

Kompendium wiedzy na temat platformy Android!



# Android 3

## tworzenie aplikacji

**Satya Komatineni • Dave MacLean • Sayed Hashimi**

# **Spis treści**

<b>Przedmowa .....</b>	<b>21</b>
<b>O autorach .....</b>	<b>23</b>
<b>Informacje o redaktorze technicznym .....</b>	<b>25</b>
<b>Podziękowania .....</b>	<b>27</b>
<b>Słowo wstępne .....</b>	<b>29</b>
<b>Rozdział 1. Wprowadzenie do platformy obliczeniowej Android .....</b>	<b>31</b>
Nowa platforma dla nowego typu komputera osobistego .....	32
Początki historii Androida .....	33
Zapoznanie się ze środowiskiem Dalvik VM .....	36
Stos programowy Androida .....	37
Projektowanie aplikacji użytkownika końcowego	
za pomocą zestawu Android SDK .....	38
Emulator Androida .....	38
Interfejs użytkownika na platformie Android .....	39
Podstawowe składniki Androida .....	40
Zaawansowane koncepcje interfejsu użytkownika .....	41
Składniki usług w Androidzie .....	43
Składniki multimedialnych oraz telefonii w Androidzie .....	43
Pakiety Java w Androidzie .....	44
Wykorzystanie zalet kodu źródłowego Androida .....	48
Przykładowe projekty zawarte w książce .....	49
Podsumowanie .....	49
<b>Rozdział 2. Konfigurowanie środowiska programowania .....</b>	<b>51</b>
Konfigurowanie środowiska .....	52
Pobieranie zestawu JDK 6 .....	52
Pobieranie środowiska Eclipse 3.6 .....	53
Pobieranie zestawu Android SDK .....	54
Okno narzędzi .....	56
Instalowanie narzędzi ADT .....	56
Przedstawienie podstawowych składników .....	58
Widok .....	58
Aktywność .....	59

Intencja .....	59
Dostawca treści .....	59
Usługa .....	59
AndroidManifest.xml .....	60
Urządzenia AVD .....	60
Witaj, świecie! .....	60
Wirtualne urządzenia AVD .....	65
Poznanie struktury aplikacji Androida .....	67
Analiza aplikacji Notepad .....	69
Wczytywanie oraz uruchomienie aplikacji Notepad .....	69
Rozłożenie kodu na czynniki pierwsze .....	71
Badanie cyklu życia aplikacji .....	78
Usuwanie błędów w aplikacji .....	81
Uruchamianie emulatora .....	83
StrictMode .....	84
Odnośniki .....	89
Podsumowanie .....	89
<b>Rozdział 3. Korzystanie z zasobów .....</b>	<b>91</b>
Zasoby .....	91
Zasoby typu string .....	92
Zasoby typu layout .....	94
Składnia odniesienia do zasobu .....	95
Definiowanie własnych identyfikatorów zasobów do późniejszego użytku .....	97
Skompilowane oraz nieskompilowane zasoby Androida .....	98
Rodzaje głównych zasobów w Androidzie .....	99
Praca na własnych plikach zasobów XML .....	109
Praca na nieskompresowanych zasobach .....	111
Praca z dodatkowymi plikami .....	111
Przegląd struktury katalogów mieszczących zasoby .....	112
Zasoby a zmiany konfiguracji .....	112
Odnośniki .....	116
Podsumowanie .....	117
<b>Rozdział 4. Dostawcy treści .....</b>	<b>119</b>
Analiza wbudowanych dostawców Androida .....	120
Architektura dostawców treści .....	126
Implementowanie dostawców treści .....	139
Testowanie dostawcy BookProvider .....	150
Dodawanie książki .....	150
Usuwanie książki .....	150
Zliczanie książek .....	151
Wyświetlanie listy książek .....	151
Odnośniki .....	152
Podsumowanie .....	153

<b>Rozdział 5. Intencje .....</b>	<b>155</b>
Podstawowe informacje na temat intencji .....	155
Intencje dostępne w Androidzie .....	156
Przegląd struktury intencji .....	159
Intencje a identyfikatory danych URI .....	159
Działania ogólne .....	160
Korzystanie z dodatkowych informacji .....	161
Stosowanie składników	
do bezpośredniego przywoływania aktywności .....	162
Kategorie intencji .....	163
Reguły przydzielania intencji do ich składników .....	166
Działanie ACTION_PICK .....	169
Działanie ACTION_GET_CONTENT .....	171
Wprowadzenie do intencji oczekujących .....	172
Odbońniki .....	173
Podsumowanie .....	174
<b>Rozdział 6. Budowanie interfejsów użytkownika oraz używanie kontrolek .....</b>	<b>175</b>
Projektowanie interfejsów UI w Androidzie .....	175
Programowanie interfejsu użytkownika wyłącznie za pomocą kodu .....	177
Tworzenie interfejsu użytkownika wyłącznie w pliku XML .....	179
Konstruowanie interfejsu użytkownika	
za pomocą kodu oraz języka XML .....	180
FILL_PARENT a MATCH_PARENT .....	182
Standardowe kontrolki Androida .....	182
Kontrolki tekstu .....	183
Kontrolki przycisków .....	187
Kontrolka ImageView .....	195
Kontrolki daty i czasu .....	197
Kontrolka MapView .....	200
Działanie adapterów .....	200
Zapoznanie się z klasą SimpleCursorAdapter .....	200
Zapoznanie się z klasą ArrayAdapter .....	202
Wykorzystywanie adapterów wraz z kontrolkami AdapterView .....	204
Podstawowa kontrolka listy — ListView .....	205
Kontrolka GridView .....	213
Kontrolka Spinner .....	215
Kontrolka Gallery .....	217
Tworzenie niestandardowych adapterów .....	218
Inne kontrolki w Androidzie .....	223
Style i motywy .....	224
Stosowanie stylów .....	224
Stosowanie motywów .....	227

Menedżery układu graficznego .....	227
Menedżer układu graficznego LinearLayout .....	228
Menedżer układu graficznego TableLayout .....	231
Menedżer układu graficznego RelativeLayout .....	235
Menedżer układu graficznego FrameLayout .....	237
Dostosowanie układu graficznego do konfiguracji różnych urządzeń .....	239
Usuwanie błędów i optymalizacja układów graficznych .....	
za pomocą narzędzia Hierarchy Viewer .....	242
Odnośniki .....	244
Podsumowanie .....	245
<b>Rozdział 7. Praca z menu ..... 247</b>	
Menu w Androidzie .....	247
Tworzenie menu .....	249
Praca z grupami menu .....	250
Odpowiedź na wybór elementów menu .....	251
Utworzenie środowiska testowego do sprawdzania menu .....	253
Praca z innymi rodzajami menu .....	259
Rozszerzone menu .....	259
Praca z menu w postaci ikon .....	259
Praca z podmenu .....	260
Zabezpieczanie menu systemowych .....	261
Praca z menu kontekstowymi .....	261
Praca z menu alternatywnymi .....	264
Praca z menu w odpowiedzi na zmianę danych .....	268
Wczytywanie menu poprzez pliki XML .....	268
Struktura pliku XML zasobów menu .....	268
Zapełnianie plików XML zasobów menu .....	269
Tworzenie odpowiedzi dla elementów menu opartych na pliku XML .....	270
Krótkie wprowadzenie do dodatkowych znaczników menu w pliku XML .....	271
Odnośniki .....	272
Podsumowanie .....	272
<b>Rozdział 8. Praca z oknami dialogowymi ..... 273</b>	
Korzystanie z okien dialogowych w Androidzie .....	274
Projektowanie okien alertów .....	274
Projektowanie okna dialogowego zachęty .....	276
Natura okien dialogowych w Androidzie .....	281
Przeprojektowanie okna dialogowego zachęty .....	282
Praca z zarządzanymi oknami dialogowymi .....	283
Protokół zarządzanych okien dialogowych .....	283
Przekształcenie niezarządzanego okna dialogowego na zarządzane okno dialogowe .....	283
Uproszczenie protokołu zarządzanych okien dialogowych .....	285

Praca z klasą Toast .....	293
Odnośniki .....	294
Podsumowanie .....	294
<b>Rozdział 9. Praca z preferencjami i zachowywanie stanów .....</b>	<b>295</b>
Badanie struktury preferencji .....	296
Klasa ListPreference .....	296
Widok CheckBoxPreference .....	305
Widok EditTextPreference .....	307
Widok RingtonePreference .....	308
Organizowanie preferencji .....	310
Programowe zarządzanie preferencjami .....	312
Zapisywanie stanu za pomocą preferencji .....	313
Odnośniki .....	314
Podsumowanie .....	315
<b>Rozdział 10. Analiza zabezpieczeń i uprawnień .....</b>	<b>317</b>
Model zabezpieczeń w Androidzie .....	317
Przegląd pojęć dotyczących zabezpieczeń .....	317
Podpisywanie wdrażanych aplikacji .....	318
Przeprowadzanie testów zabezpieczeń środowiska wykonawczego .....	324
Zabezpieczenia na granicach procesu .....	324
Deklarowanie oraz stosowanie uprawnień .....	325
Stosowanie niestandardowych uprawnień .....	326
Stosowanie uprawnień identyfikatorów URI .....	332
Odnośniki .....	334
Podsumowanie .....	335
<b>Rozdział 11. Tworzenie i użytkowanie usług .....</b>	<b>337</b>
Użytkowanie usług HTTP .....	337
Wykorzystanie modułu HttpClient do żądań wywołania GET .....	338
Wykorzystanie modułu HttpClient do żądań wywołania POST (przykład wieloczęściowy) .....	340
Parsery SOAP, JSON i XML .....	342
Obsługa wyjątków .....	343
Problemy z wielowątkowością .....	345
Zabawa z przekroczeniami limitu czasu .....	348
Stosowanie klasy HttpURLConnection .....	349
Używanie klasy AndroidHttpClient .....	349
Stosowanie wątków drugoplanowych (AsyncTask) .....	351
Obsługa zmian konfiguracji za pomocą klasy AsyncTask .....	357
Pobieranie plików za pomocą klasy DownloadManager .....	362
Stosowanie usług w Androidzie .....	367
Usługi w Androidzie .....	368
Usługi lokalne .....	369
Usługi AIDL .....	376

Definiowanie interfejsu usługi w języku AIDL .....	376
Implementowanie interfejsu AIDL .....	379
Wywoływanie usługi z poziomu aplikacji klienckiej .....	381
Przekazywanie usługom złożonych typów danych .....	385
Przykład aplikacji użytkowej korzystającej z usług .....	395
Interfejs Tłumacz Google .....	395
Stosowanie interfejsu Tłumacz Google .....	397
Odnośniki .....	405
Podsumowanie .....	405
<b>Rozdział 12. Analiza pakietów .....</b>	<b>407</b>
Pakiet i procesy .....	407
Szczegółowa specyfikacja pakietu .....	407
Przekształcanie nazwy pakietu w nazwę procesu .....	408
Tworzenie listy zainstalowanych pakietów .....	408
Usuwanie pakietu za pomocą aplikacji Package Browser .....	409
Jeszcze raz o procesie podpisywania pakietów .....	409
Zrozumienie koncepcji podpisów cyfrowych — scenariusz 1. ....	410
Zrozumienie koncepcji podpisów cyfrowych — scenariusz 2. ....	410
Wysławienie koncepcji podpisów cyfrowych .....	410
Jak zatem tworzymy cyfrowy podpis .....	411
Implikacje wynikające z podpisywania plików .....	411
Współdzielenie danych pomiędzy pakietami .....	412
Natura współdzielonych identyfikatorów użytkownika .....	412
Schemat kodu wykorzystywanego przy współdzieleniu danych .....	413
Projekty bibliotek .....	414
Czym jest projekt bibliotek? .....	414
Twierdzenia dotyczące projektów bibliotek .....	414
Utworzenie projektu bibliotek .....	417
Tworzenie projektu testowego wykorzystującego projekt bibliotek ....	420
Odnośniki .....	425
Podsumowanie .....	426
<b>Rozdział 13. Analiza procedur obsługi .....</b>	<b>427</b>
Składniki Androida i wątkowanie .....	427
Aktywności działają w głównym wątku .....	428
Odbiorcy komunikatów działają w głównym wątku .....	429
Usługi działają w głównym wątku .....	429
Dostawcy treści działają w głównym wątku .....	429
Skutki posiadania pojedynczego głównego wątku .....	429
Pule wątków, dostawcy treści, składniki zewnętrznych usług .....	429
Narzędzia wątkowania — poznaj swój wątek .....	429
Procedury obsługi .....	431
Skutki przetrzymywania głównego wątku .....	432
Zastosowanie procedury obsługi do opóźnienia operacji w wątku głównym .....	432

Przykładowy kod źródłowy procedury obsługi opóźniającej	433
przeprowadzanie operacji .....	433
Konstruowanie odpowiedniego obiektu Message .....	435
Wysyłanie obiektów Message do kolejki .....	435
Odpowiedź na metodę zwrotną handleMessage .....	436
Stosowanie wątków roboczych .....	436
Przywoływanie wątku roboczego z poziomu menu .....	437
Komunikacja pomiędzy wątkami głównym i roboczym .....	438
Szybki przegląd — jak działa wątek? .....	440
Klasy przykładowego sterownika procedury obsługi .....	441
Plik aktywności sterującej .....	442
Plik układu graficznego .....	444
Plik menu .....	445
Plik manifest .....	445
Czas życia składnika i procesu .....	446
Cykl życia aktywności .....	446
Cykl życia usługi .....	448
Cykl życia odbiorców komunikatów .....	448
Cykl życia dostawcy treści .....	448
Instrukcje dotyczące komplikowania kodu .....	449
Utworzenie projektu za pomocą pliku ZIP .....	449
Tworzenie projektu za pomocą listingów .....	449
Odbońniki .....	450
Podsumowanie .....	450
<b>Rozdział 14. Odbiorcy komunikatów i usługi długoterminowe .....</b>	<b>453</b>
Odbiorcy komunikatów .....	453
Wysyłanie komunikatu .....	454
Tworzenie prostego odbiorcy — przykładowy kod .....	454
Rejestrowanie odbiorcy komunikatów w pliku manifeście .....	456
Wysyłanie komunikatu testowego .....	456
Wprowadzanie wielu odbiorców komunikatów .....	460
Projekt wykorzystujący odbiorców pozaprocesowych .....	462
Używanie powiadomień pochodzących od odbiorcy komunikatów .....	463
Monitorowanie powiadomień za pomocą menedżera powiadomień .....	463
Wysyłanie powiadomienia .....	464
Długoterminowi odbiorcy komunikatów i usługi .....	467
Protokół długoterminowego odbiorcy komunikatów .....	468
Klasa IntentService .....	469
Kod źródłowy klasy IntentService .....	470
Rozszerzanie klasy IntentService na odbiorcę komunikatów .....	472
Abstrakcja długoterminowej usługi wysyłającej komunikaty .....	472
Długoterminowy odbiorca komunikatów .....	474
Wyodrębnianie blokady przechodzenia	
w stan zatrzymania za pomocą klasy LightedGreenRoom .....	476

Oświetlony zielony pokój .....	478
Implementacja oświetlonego zielonego pokoju .....	478
Implementacja długoterminowej usługi .....	483
Szczegółowe informacje na temat usługi nietrwałej .....	484
Informacje dotyczące trwałej usługi .....	485
Odmiana nietrwałej usługi — ponownie dostarczane intencje .....	485
Definiowanie flag usługi w metodzie onStartCommand .....	485
Wybieranie odpowiedniego trybu usługi .....	485
Kontrolowanie blokady przechodzenia w stan zatrzymania z dwóch miejsc jednocześnie .....	486
Implementacja długoterminowej usługi .....	486
Testowanie długoterminowych usług .....	488
Instrukcje dotyczące kompilowania kodu .....	489
Utworzenie projektów za pomocą pliku ZIP .....	489
Utworzenie projektów za pomocą listingów .....	489
Odnośniki .....	491
Podsumowanie .....	492
<b>Rozdział 15. Badanie menedżera alarmów .....</b>	<b>493</b>
Podstawy menedżera alarmów — konfiguracja prostego alarmu .....	493
Uzyskanie dostępu do menedżera alarmów .....	494
Definiowanie czasu uruchomienia alarmu .....	494
Konfigurowanie odbiorcy dla alarmu .....	495
Utworzenie oczekującej intencji dostosowanej do alarmu .....	495
Ustawianie alarmu .....	496
Projekt testowy .....	497
Analiza alternatywnych wersji menedżera alarmów .....	503
Konfigurowanie powtarzalnego alarmu .....	503
Anulowanie alarmu .....	506
Praca z wieloma alarmami jednocześnie .....	508
Pierwszeństwo intencji w uruchamianiu alarmów .....	512
Trwałość alarmów .....	515
Twierdzenia dotyczące menedżera alarmów .....	515
Odnośniki .....	516
Podsumowanie .....	516
<b>Rozdział 16. Analiza animacji dwuwymiarowej .....</b>	<b>517</b>
Animacja poklatkowa .....	518
Zaplanowanie animacji poklatkowej .....	518
Utworzenie aktywności .....	519
Dodawanie animacji do aktywności .....	520
Animacja układu graficznego .....	523
Podstawowe typy animacji klatek kluczowych .....	524
Zaplanowanie środowiska testowego animacji układu graficznego .....	525

Utworzenie aktywności oraz widoku ListView .....	525
Animowanie widoku ListView .....	528
Stosowanie interpolatorów .....	531
Animacja widoku .....	533
Animacja widoku .....	533
Dodawanie animacji .....	536
Zastosowanie klasy Camera do symulowania głębi w obrazie dwuwymiarowym .....	539
Analiza interfejsu AnimationListener .....	541
Kilka uwag na temat macierzy transformacji .....	541
Odnośniki .....	542
Podsumowanie .....	543
<b>Rozdział 17. Analiza usług wykorzystujących mapy i dane o lokalizacji .....</b>	<b>545</b>
Pakiet do pracy z mapami .....	546
Uzyskanie klucza interfejsu API mapy od firmy Google .....	546
Klasy MapView i MapActivity .....	548
Dodawanie znaczników za pomocą nakładek .....	553
Pakiet do obsługi danych o położeniu geograficznym .....	559
Geokodowanie w Androidzie .....	559
Geokodowanie za pomocą wątków przebiegających w tle .....	563
Usługa LocationManager .....	566
Wyświetlanie informacji o położeniu za pomocą klasy MyLocationOverlay .....	574
Stosowanie alertów odległościowych .....	578
Odnośniki .....	583
Podsumowanie .....	583
<b>Rozdział 18. Używanie interfejsów telefonii .....</b>	<b>585</b>
Praca z wiadomościami SMS .....	585
Wysyłanie wiadomości SMS .....	585
Monitorowanie przychodzących wiadomości tekstowych .....	589
Praca z folderami wiadomości SMS .....	592
Wysyłanie wiadomości e-mail .....	593
Praca z menedżerem telefonii .....	594
Protokół inicjalizacji sesji (SIP) .....	597
Odnośniki .....	600
Podsumowanie .....	600
<b>Rozdział 19. Używanie szkieletu multimedialnego .....</b>	<b>601</b>
Stosowanie interfejsów API multimedialnych .....	601
Wykorzystywanie kart SD .....	602
Odtwarzanie multimedialnych .....	606
Odtwarzanie źródeł dźwiękowych .....	607
Odtwarzanie plików wideo .....	619

Rejestrowanie multimediarów .....	621
Analiza procesu rejestracji dźwięku za pomocą klasy MediaRecorder .....	622
Rejestracja dźwięków za pomocą klasy AudioRecord .....	626
Analiza procesu rejestracji wideo .....	630
Analiza klasy MediaStore .....	640
Rejestrowanie dźwięku za pomocą intencji .....	641
Dodawanie plików do magazynu multimediarów .....	644
Podłączenie klasy MediaScanner do całej karty SD .....	647
Odnośniki .....	647
Podsumowanie .....	648
<b>Rozdział 20. Programowanie grafiki trójwymiarowej za pomocą biblioteki OpenGL .....</b>	<b>649</b>
Historia i podstawy biblioteki OpenGL .....	650
OpenGL ES .....	651
Środowisko OpenGL ES a Java ME .....	652
M3G — inny standard grafiki trójwymiarowej środowiska Java .....	652
Podstawy struktury OpenGL .....	653
Podstawy rysowania za pomocą biblioteki OpenGL .....	654
Kamera i współrzędne .....	659
Tworzenie interfejsu pomiędzy standardem OpenGL ES a Androidem ....	663
Stosowanie klasy GLSurfaceView i klas pokrewnych .....	664
Implementacja klasy Renderer .....	664
Zastosowanie klasy GLSurfaceView z poziomu aktywności .....	667
Zmiana ustawień kamery .....	672
Wykorzystanie indeksów do dodania kolejnego trójkąta .....	675
Animowanie prostego trójkąta w bibliotece OpenGL .....	676
Stawianie czoła bibliotece OpenGL — kształty i tekstury .....	678
Rysowanie prostokąta .....	679
Praca z kształtami .....	680
Praca z teksturami .....	694
Rysowanie wielu figur geometrycznych .....	699
OpenGL ES 2.0 .....	703
Powiązania środowiska Java z bibliotekami OpenGL ES 2.0 .....	704
Etapy renderowania .....	707
Jednostki cieniąjące .....	708
Kompilowanie jednostek cieniących w programie .....	709
Uzyskiwanie dostępu do zmiennych jednostek cieniowania .....	711
Prosty trójkąt napisany w środowisku OpenGL ES 2.0 .....	711
Dodatkowe źródła dotyczące środowiska OpenGL ES 2.0 .....	715
Instrukcje związane z kompilowaniem kodu .....	715
Odnośniki .....	715
Podsumowanie .....	716

<b>Rozdział 21. Badanie aktywnych folderów .....</b>	<b>717</b>
Badanie aktywnych folderów .....	717
W jaki sposób użytkownik korzysta z aktywnych folderów .....	718
Tworzenie aktywnego folderu .....	722
Instrukcje dotyczące kompilowania kodu .....	733
Odnośniki .....	733
Podsumowanie .....	734
<b>Rozdział 22. Widżety ekranu startowego .....</b>	<b>735</b>
Architektura widżetów ekranu startowego .....	736
Czym są widżety ekranu startowego? .....	736
W jaki sposób użytkownik korzysta z widżetów ekranu startowego? .....	736
Cykl życia widżetu .....	740
Przykładowy widżet .....	745
Definiowanie dostawcy widżetu .....	747
Definiowanie rozmiaru widżetu .....	748
Pliki związane z układem graficznym widżetu .....	749
Implementacja dostawcy widżetu .....	751
Implementacja modeli widżetów .....	753
Implementacja aktywności konfiguracji widżetu .....	761
Ograniczenia i rozszerzenia widżetów .....	764
Odnośniki .....	765
Podsumowanie .....	766
<b>Rozdział 23. Wyszukiwanie w Androidzie .....</b>	<b>767</b>
Wyszukiwanie w Androidzie .....	768
Badanie procesu przeszukiwania globalnego w Androidzie .....	768
Włączanie dostawców propozycji do procesu wyszukiwania globalnego .....	774
Interakcja aktywności z przyciskiem wyszukiwania .....	777
Zachowanie przycisku wyszukiwania wobec standardowej aktywności .....	778
Zachowanie aktywności wyłączającej wyszukiwanie .....	786
Jawne wywoływanie wyszukiwania za pomocą menu .....	787
Wyszukiwanie lokalne i pokrewne aktywności .....	790
Uruchomienie funkcji type-to-search .....	797
Implementacja prostego dostawcy propozycji .....	798
Planowanie prostego dostawcy propozycji .....	798
Pliki implementacji prostego dostawcy propozycji .....	799
Implementacja klasy SimpleSuggestionProvider .....	799
Aktywność wyszukiwania dostępna w prostym dostawcy propozycji .....	803
Aktywność wywołania wyszukiwania .....	808
Użytkowanie prostego dostawcy propozycji .....	810
Implementacja niestandardowego dostawcy propozycji .....	813
Implementacja niestandardowego dostawcy propozycji .....	814
Pliki wymagane do implementacji projektu SuggestUrlProvider .....	814

Implementacja klasy SuggestUrlProvider .....	815
Implementacja aktywności wyszukiwania	
dla niestandardowego dostawcy propozycji .....	824
Plik manifest niestandardowego dostawcy propozycji .....	830
Korzystanie z niestandardowego dostawcy propozycji .....	831
Zastosowanie przycisków działania	
i danych wyszukiwania specyficznych dla aplikacji .....	835
Wykorzystanie przycisków działania w procesie wyszukiwania .....	835
Praca ze specyficznym dla aplikacji kontekstem wyszukiwania .....	838
Odnośniki .....	839
Wyszukiwanie w tabletach .....	840
Podsumowanie .....	840
<b>Rozdział 24. Analiza interfejsu przetwarzania tekstu na mowę .....</b>	<b>841</b>
Podstawy technologii przetwarzania tekstu na mowę w Androidzie .....	841
Używanie wyrażeń do śledzenia toku wypowiedzi .....	846
Zastosowanie plików dźwiękowych do przetwarzania tekstu na mowę .....	848
Zaawansowane funkcje silnika TTS .....	854
Konfiguracja strumieni audio .....	855
Stosowanie ikon akustycznych .....	855
Odtwarzanie ciszy .....	856
Wybór innych mechanizmów przetwarzania tekstu na mowę .....	856
Stosowanie metod językowych .....	857
Odnośniki .....	858
Podsumowanie .....	859
<b>Rozdział 25. Ekrany dotykowe .....</b>	<b>861</b>
Klasa MotionEvent .....	861
Obiekt MotionEvent .....	862
Wielokrotne wykorzystywanie obiektów MotionEvent .....	873
Stosowanie klasy VelocityTracker .....	874
Analiza funkcji przeciągania .....	876
Wielodotykowość .....	879
Funkcja wielodotykowości przed wersją 2.2 Androida .....	879
Funkcja wielodotykowości w systemach poprzedzających wersję 2.2 .....	887
Obsługa map za pomocą dotyku .....	888
Gesty .....	891
Gest ściskania .....	891
Klasy GestureDetector i OnGestureListener .....	895
Niestandardowe gesty .....	898
Aplikacja Gestures Builder .....	898
Odnośniki .....	905
Podsumowanie .....	905

---

<b>Rozdział 26. Czujniki .....</b>	<b>907</b>
Czym jest czujnik? .....	907
Wykrywanie czujników .....	908
Jakie informacje możemy uzyskać na temat czujnika? .....	909
Pobieranie zdarzeń generowanych przez czujniki .....	911
Problemy pojawiające się podczas uzyskiwania danych z czujników ...	914
Interpretowanie danych czujnika .....	921
Czujniki oświetlenia .....	921
Czujniki zbliżeniowe .....	922
Termometry .....	922
Czujniki ciśnienia .....	923
Żyroskopy .....	923
Akcelerometry .....	924
Magnetometry .....	930
Współpraca akcelerometrów z magnetometrami .....	931
Czujniki orientacji w przestrzeni .....	931
Deklinacja magnetyczna i klasa GeomagneticField .....	938
Czujniki grawitacji .....	939
Czujniki przyśpieszenia liniowego .....	939
Czujniki wektora obrotu .....	939
Czujniki komunikacji bliskiego pola .....	939
Odnośniki .....	950
Podsumowanie .....	951
<b>Rozdział 27. Analiza interfejsu kontaktów .....</b>	<b>953</b>
Koncepcja konta .....	954
Szybki przegląd ekranów związanych z kontami .....	954
Związek pomiędzy kontami a kontaktami .....	957
Wyliczanie kont .....	957
Aplikacja Kontakty .....	958
Wyświetlanie kontaktów .....	958
Wyświetlanie szczegółów kontaktu .....	959
Edytowanie szczegółów kontaktu .....	960
Umieszczanie zdjęcia powiązanego z kontaktem .....	962
Eksportowanie kontaktów .....	962
Różne typy danych kontaktowych .....	964
Analiza kontaktów .....	964
Badanie treści bazy danych SQLite .....	965
Nieprzetworzone kontakty .....	965
Tabela danych .....	967
Kontakty zbiorcze .....	968
view_contacts .....	971
contact_entities_view .....	971

Praca z interfejsem kontaktów .....	972
Eksploracja kont .....	972
Badanie kontaktów zbiorczych .....	980
Badanie nieprzetworzonych kontaktów .....	989
Przeglądanie danych nieprzetworzonego kontaktu .....	994
Dodawanie kontaktu oraz szczegółowych informacji o nim .....	998
Kontrola agregacji .....	1001
Konsekwencje synchronizacji .....	1002
Odbońniki .....	1002
Podsumowanie .....	1003
<b>Rozdział 28. Wdrażanie aplikacji na rynek — Android Market i nie tylko ..... 1005</b>	
Jak zostać wydawcą? .....	1006
Postępowanie zgodnie z zasadami .....	1006
Konsola programisty .....	1009
Przygotowanie aplikacji do sprzedaży .....	1012
Testowanie działania na różnych urządzeniach .....	1012
Obsługa różnych rozmiarów ekranu .....	1012
Przygotowanie pliku AndroidManifest.xml	
do umieszczenia w sklepie Android Market .....	1013
Lokalizacja aplikacji .....	1014
Przygotowanie ikony aplikacji .....	1015
Problemy związane z zarabianiem pieniędzy na aplikacjach .....	1016
Kierowanie użytkowników z powrotem do sklepu .....	1016
Usługa licencyjna systemu Android .....	1017
Przygotowanie pliku .apk do wysłania .....	1018
Wysyłanie aplikacji .....	1018
Korzystanie ze sklepu Android Market .....	1022
Alternatywy dla serwisu Android Market .....	1023
Odbońniki .....	1024
Podsumowanie .....	1024
<b>Rozdział 29. Koncepcja fragmentów oraz inne pojęcia dotyczące tabletów ..... 1025</b>	
Czym jest fragment? .....	1026
Kiedy należy stosować fragmenty? .....	1027
Struktura fragmentu .....	1027
Cykl życia fragmentu .....	1028
Przykładowa aplikacja ukazująca cykl życia fragmentu .....	1033
Klasy FragmentTransaction i drugoplanowy stos fragmentów .....	1042
Przejścia i animacje zachodzące podczas transakcji fragmentu .....	1044
Klasa FragmentManager .....	1045
Ostrzeżenie dotyczące stosowania odniesień do fragmentów .....	1046
Klasa ListFragment i węzeł <fragment> .....	1047
Wywoływanie odrębnej aktywności w razie potrzeby .....	1051
Trwałość fragmentów .....	1054

Fragmenty wyświetlające okna dialogowe .....	1054
Podstawowe informacje o klasie DialogFragment .....	1055
Przykładowa aplikacja wykorzystująca klasę DialogFragment .....	1060
Inne formy komunikowania się z fragmentami .....	1073
Stosowanie metod startActivity() i setTargetFragment() .....	1074
Tworzenie niestandardowych animacji	
za pomocą klasy ObjectAnimator .....	1075
Odnośniki .....	1078
Podsumowanie .....	1078
<b>Rozdział 30. Analiza klasy ActionBar .....</b>	<b>1079</b>
Anatomia klasy ActionBar .....	1080
Aktywność paska działania wyświetlającego zakładki .....	1081
Implementacja bazowych klas aktywności .....	1082
Wprowadzenie jednolitego zachowania klas ActionBar .....	1084
Implementacja obiektu nasłuchującego zdarzeń z zakładek .....	1087
Implementacja aktywności przechowującej pasek zakładek .....	1088
Przewijalny układ graficzny zawierający widok debugowania .....	1090
Pasek działania a interakcja z menu .....	1091
Plik manifest Androida .....	1093
Badanie aktywności przechowującej pasek zakładek .....	1093
Aktywność paska działania w trybie wyświetlania listy .....	1094
Utworzenie klasy SpinnerAdapter .....	1095
Utworzenie obiektu nasłuchującego listy .....	1095
Konfigurowanie paska działania w trybie wyświetlania listy .....	1096
Zmiany w klasie BaseActionBarActivity .....	1097
Zmiany w pliku AndroidManifest.xml .....	1097
Badanie aktywności zawierającej pasek działania	
w trybie wyświetlania listy .....	1098
Aktywność przechowująca standardowy pasek działania .....	1099
Aktywność przechowująca standardowy pasek działania .....	1100
Zmiany w klasie BaseActionBarActivity .....	1101
Zmiany w pliku AndroidManifest.xml .....	1101
Badanie aktywności przechowującej standardowy pasek działania .....	1102
Odnośniki .....	1102
Podsumowanie .....	1104
<b>Rozdział 31. Dodatkowe zagadnienia związane z wersją 3.0 systemu .....</b>	<b>1105</b>
Widżety ekranu startowego oparte na listach .....	1105
Nowe widoki zdalne w wersji 3.0 systemu .....	1106
Praca z listami stanowiącymi część widoku zdalnego .....	1107
Działający przykład	
— testowy widżet ekranu startowego oparty na liście .....	1121
Testowanie widżetu wyświetlającego listę .....	1130

Funkcja przeciągania .....	1131
Podstawowe informacje	
o funkcji przeciągania w wersji 3.0 Androida .....	1131
Przykładowa aplikacja prezentująca funkcję przeciągania .....	1133
Testowanie przykładowej aplikacji wykorzystującej funkcję	
przeciągania .....	1145
Odnośniki .....	1146
Podsumowanie .....	1147
<b>Skorowidz .....</b>	<b>1149</b>

# Przedmowa

Wszystko już się kiedyś wydarzyło i wszystko wydarzy się ponownie w przyszłości. Teoria emergencji (wyłaniania się) wyjaśnia, w jaki sposób z oddziaływań między prostszymi elementami wyłaniają się złożone systemy i wzorce.

Także my już tu kiedyś byliśmy.

Gdy w 1985 roku rozpoczynałem przygodę z programowaniem, dostępnych było wiele różnorodnych komputerów osobistych. Gdy zdobywałem pierwsze szlify w pracy z komputerem Apple II C, moi znajomi korzystali z platform Commodore 128s, Tandy CoCo 3s lub Atari. Każdy z nas rozwijał się w tym samym środowisku, jednak rzadko kiedy mogliśmy dzielić się wynikami pracy. Gdy zaczęły się pojawiać przystępne cenowo klony komputera firmy IBM, które obsługiwały system DOS firmy Microsoft, programiści dostrzegli potencjał tworzącego się właśnie rynku i nastąpiła szybka ewolucja środowiska DOS. Ostatecznie firma Microsoft zdobyła dominującą pozycję na rynku komputerów PC, którą cieszy się do dzisiaj.

Gdy w 2003 roku zacząłem zajmować się programowaniem mobilnych aplikacji, sytuacja bardzo przypominała tę z 1985 roku. Mogliśmy urzeczywistniać swoje pomysły za pomocą różnorodnych środowisk, począwszy od .NET CF firmy Microsoft, poprzez Java Micro Edition, a skończywszy na BREW. Jednak, podobnie jak w przypadku gier, które kiedyś tworzyliśmy z przyjaciółmi, napisane aplikacje funkcjonowały tylko w określonym środowisku.

Teraz, w 2011 roku, firma Google, dzięki polityce udostępniania systemu Android producentom sprzętu, zdaje się stawać Microsoftem w Świecie Urządzeń Mobilnych. Prawdopodobnie dlatego wybrałeś tę książkę i zacząłeś czytać przedmowę. Albo jesteś studentem historii, albo — podobnie jak ja — masz szczęście w niej uczestniczyć.

Jeśli tak — mam dla Ciebie dobre wieści! Pracowaliśmy bardzo ciężko, przygotowując nowe wydanie tej książki, aby udostępnić Ci narzędzia umożliwiające implementację pomysłów krążących Ci po głowie. Przeprowadzimy Cię przez podstawy, począwszy od konfigurowania środowiska, a skończywszy na wdrożeniu aplikacji w sklepie Android Marketplace. Oczywiście, jest to bardzo rozległa podróż, dlatego będziemy podróżować przede wszystkim najbardziej uczęszczanymi szlakami. Pokażemy Ci jednak mnóstwo zasobów, za pomocą których możesz zwiedzać szeroki świat Androida na własną rękę.

Powodzenia i szczęśliwej drogi!

— Dylan Phillips



# O autorach



**Satya Komatineni** ([www.satyakomatineni.com](http://www.satyakomatineni.com)) ma ponad dwudziestoletnie doświadczenie w programowaniu dla dużych i mniejszych przedsiębiorstw. Satya Komatineni opublikował ponad 30 artykułów dotyczących projektowania stron WWW przy użyciu technologii Java, .NET oraz baz danych. Jest częstym prelegentem na konferencjach przemysłowych dotyczących innowacyjnych technologii oraz regularnie umieszcza wpisy na blogach w serwisie [java.net](http://java.net). Jest także twórcą AspireWeb ([www.activeintellect.com/aspire](http://www.activeintellect.com/aspire)) — uproszczonego narzędzia o jawnym kodzie źródłowym, służącego do projektowania stron WWW w języku Java, oraz Aspire Knowledge Central — sieciowego systemu operacyjnego o jawnym kodzie źródłowym, nastawionego na efektywność oraz możliwość publikowania przez poszczególne osoby. Autor jest również członkiem wielu programów SBIR (ang. *Small Business Innovation Research Program* — program rozwoju innowacji w małych przedsiębiorstwach). Uzyskał stopień licencjata inżynierii elektrycznej na Uniwersytecie Andhra w Visakhapatnamie oraz tytuł magistra inżynierii elektrycznej w Indyjskim Instytucie Technologicznym w Nowym Delhi. Można się z nim skontaktować, pisząc na adres [satya.komatineni@gmail.com](mailto:satya.komatineni@gmail.com).



**Dave MacLean** jest inżynierem i architektem oprogramowania. Obecnie mieszka i pracuje w Jacksonville na Florydzie. Od 1980 roku zajmuje się programowaniem w wielu językach oraz projektowaniem — począwszy od systemów automatyzacji robotów, na systemach przechowywania danych skończywszy, od automatycznie obsługiwanych aplikacji sieciowych do procesorów transakcji EDI. Dave MacLean pracował dla takich firm, jak Sun Microsystems, IBM, Trimble Navigation, General Motors oraz kilku mniejszych przedsiębiorstw. Ukończył studia na Uniwersytecie Waterloo w Kanadzie, uzyskując tytuł inżyniera projektowania systemów. Prowadzi blog dostępny pod adresem <http://davemac327.blogspot.com>, natomiast jego adres kontaktowy to [davemac327@gmail.com](mailto:davemac327@gmail.com).



**Sayed Y. Hashimi** urodził się w Afganistanie, obecnie zaś przebywa w Jacksonville na Florydzie. Ma bogate doświadczenie w dziedzinie ochrony zdrowia, finansów, logistyki oraz architektury zorientowanej na usługi. Zawodowo autor projektuje wielkoskalowe aplikacje rozproszone, wykorzystując różne języki oraz platformy, w tym C/C++, MFC, J2EE oraz .NET. Publikował artykuły w największych czasopismach poświęconych oprogramowaniu oraz napisał kilka innych popularnych książek dla wydawnictwa Apress. Sayed Y. Hashimi posiada tytuł magistra inżynierii uzyskany na Uniwersytecie Floryda. Można się z nim skontaktować na stronie [www.sayedhashimi.com](http://www.sayedhashimi.com).

Zapraszamy na naszą stronę <http://androidbook.com>.



# Informacje o redaktorze technicznym



**Dylan Phillips** jest inżynierem i architektem oprogramowania, pracującym w branży rozwiązań mobilnych już od 10 lat. Dzięki bogatemu doświadczeniu, rozciągającemu się od pracy w środowisku J2ME, poprzez .NET Compact Framework, aż do systemu Android, z radością dostrzegł potencjał w dostosowywaniu urządzeń wykorzystujących Androida do różnorodnych zapotrzebowan konsumentów. Można się z nim skontaktować poprzez adres [mykoan@hotmail.com](mailto:mykoan@hotmail.com), [@mykoan](https://twitter.com/mykoan) na Twitterze albo na obiedzie w jednej z licznych restauracji Pho House rozrzuconych po całych Stanach Zjednoczonych.



# **Podziękowania**

Napisanie tej książki wymagało wielkiego wysiłku nie tylko z naszej — autorów — strony, lecz również od części bardzo utalentowanego zespołu wydawnictwa Apress, a także ze strony redaktora technicznego. Chcieliśmy zatem podziękować Steve'owi Anglinowi, Matthew Moodiemu, Corbin Collins, Heather Lang, Tracy Brown, Mary Behr oraz Brigid Duffy z wydawnictwa Apress. Chcieliśmy także wyrazić nasze uznanie dla redaktora technicznego, Dylana Phillipsa, za pracę, jaką włożył w tę książkę. Jego komentarze oraz poprawki były bezcenne. W trakcie poszukiwań odpowiedzi na forum programistycznym Androida często pomocą służyły nam Dianne Hackborn, Nick Pelly, Brad Fitzpatrick oraz inni członkowie Android Team. Byli gotowi do pomocy o każdej porze dnia oraz w weekendy, za co chcemy im wszystkim powiedzieć: „Dziękujemy!”. To zdecydowanie oni są najczęściej pracującym zespołem w świecie urządzeń mobilnych. Społeczność użytkowników systemu Android jest bardzo aktywna i rozbudowana. Nieraz ludzie ci służyli pomocą w znajdowaniu odpowiedzi na trudne pytania, udzielali także pozytycznych rad. Mamy nadzieję, że ta książka w jakiś sposób przyda się całej tej społeczności.

Jesteśmy także głęboko wdzięczni naszym rodzinom za wyrozumiałość podczas przedłużającego się pisania książki.



# Słowo wstępne

Czy kiedykolwiek chciałeś być Rodinem<sup>1</sup>? Siedzieć z długiem w dłoni i ciosać skałę, formując ją na kształt własnej wizji? Większość programistów dążących najpopularniejszymi nurtami trzymała się z daleka od mocno ograniczonych urządzeń mobilnych ze strachu przed niemożnością wyrzeźbienia użytecznej aplikacji. Ale te czasy już minęły.

System Android pozwala nam na pracę z nieprawdopodobną liczbą programowalnych urządzeń. W tej książce pragniemy potwierdzić podejrzenia, jakoby system Android znamomie nadawał się do pisania aplikacji. Jeśli programujesz w Javie, zyskujesz olbrzymią szansę na korzyści płynące z tej eksytyjącej, pełnej możliwości, wielozadaniowej platformy obliczeniowej. Cieszymy się z Androida, ponieważ stanowi on zaawansowaną platformę, wprowadzającą wiele nowych paradygmatów w kwestii projektowania szkieletów (pomimo ograniczeń urządzeń mobilnych).

Jest to nasza trzecia, dotychczas najlepsza, edycja książki poświęconej Androidowi. *Pro Android 3* jest rozległym przewodnikiem programistycznym. W tym wydaniu udoskonaliliśmy, przepracowaliśmy stylistycznie oraz poprawiliśmy wszystkie elementy książki *Android 2. Tworzenie aplikacji* w celu stworzenia gruntownie uaktualnionego przewodnika, służącego zarówno początkującym, jak i zaawansowanym programistom — będącego wynikiem trzech lat pracy. Omawiamy ponad 100 zagadnień podzielonych na 31 rozdziałów. Niniejsze wydanie książki obejmuje wersje 2.3 oraz 3.0 systemu Android, które są zoptymalizowanymi wersjami dla, odpowiednio, telefonów i tabletów.

W tym wydaniu poświęciliśmy więcej uwagi wewnętrznym mechanizmom Androida poprzez omówienie wątków, procesów, długoterminowych usług, odbiorców komunikatów oraz menedżerów alarmów. Omawiamy również o wiele więcej kontrolek interfejsu użytkownika. Zamieściliśmy ponad 150 stron poświęconych wersji 3.0 systemu, gdzie omówiliśmy fragmenty, dialogi fragmentów, klasę *ActionBar*, a także funkcję przeciagania. Znacznie rozwinięliśmy rozdziały poświęcone czujnikom i usługom. Rozdział omawiający środowisko OpenGL został zaktualizowany pod kątem obsługi wersji OpenGL ES 2.0.

Zasadniczymi elementami tej książki są objaśnienia pojęć, listingi oraz samouczki. Wszystkie rozdziały zostały podporządkowane tej filozofii. Każdy samodzielny samouczek został opatrzony komentarzem eksperta. Wszystkie projekty zawarte w książce są dostępne do pobrania i można je bez trudu zimportować do środowiska Eclipse. Ciężko również pracowaliśmy nad tym, aby każdy pokazany tu kod mógł zostać bezproblemowo skompilowany. Lista plików składających

---

<sup>1</sup> August François-René Rodin — francuski rzeźbiarz, ur. 12 listopada 1840 r. w Paryżu, zm. 17 listopada 1917 r. w Meudon. W swoich pracach łączył elementy symbolizmu i impresjonizmu. Był prekursorem nowoczesnego rzeźbiarstwa — przyp. red.

się na każdy projekt w danym rozdziale została jasno przedstawiona, dzięki czemu własnoręczne utworzenie projektu staje się jeszcze prostsze.

Tematyka książki obejmuje takie kluczowe pojęcia, jak zasoby, intencje, dostawcy treści, procesy, wątki, kontrolki interfejsu użytkownika, odbiorcy wiadomości, usługi oraz usługi długoterminowe. Osoby dopiero poznające środowisko OpenGL znajdą tu mnóstwo materiałów dotyczących wersji OpenGL ES 1.0 oraz 2.0. Wiele miejsca poświęciliśmy funkcjom przetwarzania tekstu na mowę, czujnikom oraz wielodotykowości. Szeroko również omówiliśmy zagadnienia dotyczące wersji 3.0 Androida, wśród których można znaleźć informacje o fragmentach, dialogach fragmentów, klasie `ActionBar` i funkcji przeciągania.

Na koniec warto wspomnieć, że wyszliśmy w tej książce poza podstawowe zagadnienia, że na każdy temat zadawaliśmy trudne pytania oraz że udokumentowaliśmy wyniki (mnogość tematów zawartych w tej książce widać w szczegółowym spisie treści). Aktualizujemy również na bieżąco pomocniczą stronę ([www.androidbook.com](http://www.androidbook.com)), publikując najnowsze oraz przyszłościowe materiały dotyczące zestawu Android SDK. W razie pojawienia się jakichkolwiek pytań w trakcie czytania książki od uzyskania szybkiej odpowiedzi dzieli Cię tylko jeden e-mail.

# Wprowadzenie do platformy obliczeniowej Android

Urządzenia komputerowe stają się coraz bardziej spersonalizowane i przystępne. Urządzenia przenośne w dużej mierze przekształciły się w platformy obliczeniowe. Telefony komórkowe nie służą już wyłącznie do rozmawiania — od pewnego czasu posiadają możliwość przenoszenia danych oraz multimedialnych. Bez różnicy, czy mówimy o telefonie, czy tablecie, urządzenia przenośne mają olbrzymie możliwości obliczeniowe, uprawniające je do uzyskania statusu komputera osobistego (ang. *Personal Computer* — PC). Wielu znanych producentów, takich jak ASUS, HP czy Dell, produkuje różnorodne urządzenia działające pod kontrolą systemu Android. Konkurencja pomiędzy poszczególnymi systemami operacyjnymi, platformami obliczeniowymi, językami programowania oraz ramowymi modelami projektowania przenosi się na urządzenia mobilne i coraz częściej właśnie ich dotyczy.

Widać także zwiększenie się liczby programów tworzonych dla urządzeń przenośnych, jako że coraz więcej aplikacji użytkowych zaczyna mieć odpowiedniki dla urządzeń mobilnych. W tej książce zademonstrujemy sposoby zastosowania języka Java do pisania programów dla urządzeń obsługujących platformę Android firmy Google (<http://developer.android.com/index.html>). Jest to środowisko o jawnym kodzie źródłowym, służące do tworzenia aplikacji dla urządzeń przenośnych.

**Uwaga!**

Android jest niezwykle interesujący, ponieważ wprowadza wiele nowych paradygmatów projektowania struktury aplikacji (pomimo ograniczeń platformy mobilnej).

W tym rozdziale przedstawimy cechy Androida oraz narzędzia SDK Android. Krótko scharakteryzujemy jego najważniejsze elementy, w syntetyczny sposób zaprezentujemy tematykę poszczególnych rozdziałów, pokażemy, w jaki sposób korzystać z kodu źródłowego Androida, oraz przedstawimy zalety projektowania aplikacji na tę platformę.

## Nowa platforma dla nowego typu komputera osobistego

Wspaniałą wieścią dla programistów jest informacja, że wyspecjalizowane urządzenia, takie jak telefony komórkowe, mogą zostać obecnie zaliczone do grona platform obliczeniowych ogólnego przeznaczenia (rysunek 1.1). Począwszy od wersji Android 3.0, do tej listy możemy oficjalnie dodać tablety. W ten sposób programowanie dla urządzeń przenośnych staje się dostępne dla języków programowania ogólnego przeznaczenia, dzięki czemu powiększają się zakres oraz udziały w rynku aplikacji przeznaczonych dla tych urządzeń.



**Rysunek 1.1.** Handheld jest nowym rodzajem komputera osobistego

Platforma Android umożliwia urzeczywistnienie tej idei uniwersalnych komputerów w przypadku urządzeń typu handheld. Jest to wszechstronne środowisko z systemem operacyjnym opartym na Linuksie, który zarządza urządzeniami, pamięcią oraz procesami. Biblioteki Java Androida zapewniają obsługę funkcji telefonu, wideo, przetwarzania tekstu na mowę, grafiki, łączności, programowania interfejsu użytkownika oraz wielu innych aspektów urządzeń.

### Uwaga!

Chociaż Android został zaprojektowany pod kątem urządzeń przenośnych oraz urządzeń typu tablet, posiada strukturę w pełni wyposażonego systemu operacyjnego. Firma Google udostępnia tę strukturę programistom języka Java poprzez zestaw SDK (ang. *Software Development Kit* — zestaw do tworzenia oprogramowania) o nazwie Android SDK. Praca na tym zestawie sprawia, że wcale nie ma się wrażenia, iż tworzy się aplikację dla urządzenia przenośnego, ponieważ można korzystać z większości bibliotek klas używanych na stacji roboczej lub serwerze — łącznie z relacyjnymi bazami danych.

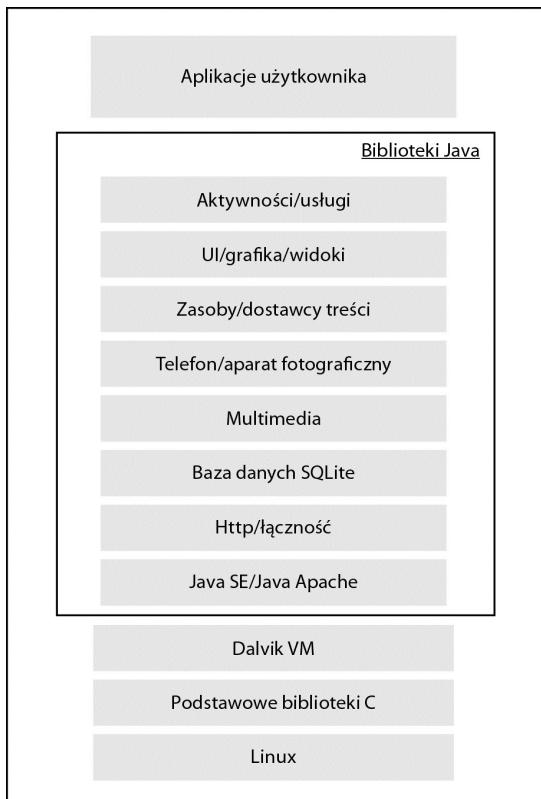
Zestaw Android SDK zapewnia obsługę znacznej części platformy Java Standard Edition (Java SE), z wyjątkiem narzędzia Abstract Window Toolkit (AWT) oraz Swing. Zamiast tych narzędzi Android SDK został zaopatrzony we własny, *obszerny, nowoczesny szkielet interfejsu użytkownika*. Językiem programowania jest Java, zatem niezbędne jest środowisko JVM (ang. *Java Virtual Machine* — wirtualna maszyna Javy), w którym odbywa się interpretowanie uruchomionego kodu bajtowego. Dzięki środowisku JVM uzyskujemy niezbędną optymalizację, pozwalającą osiągnąć wydajność porównywalną do wydajności aplikacji skompilowanych w takich językach, jak C oraz C++. Android zawiera własne, zoptymalizowane środowisko JVM, umożliwiające uruchomienie skompilowanych plików klasy Java w celu określenia takich ograniczeń, jak rozmiar pamięci RAM, ograniczenia dotyczące czasu wykonywania aplikacji itp.

niczeń urządzenia typu handheld, jak pojemność pamięci, szybkość procesora oraz moc. Ta wirtualna maszyna, nazwana Dalvik VM, zostanie dokładniej omówiona w podrozdziale „Zapoznanie się ze środowiskiem Dalvik VM”.

**Uwaga!**

Podobieństwo języka Java do jego wersji stosowanej w komputerach PC oraz jego prostota w połączeniu z rozbudowaną biblioteką klas Androida sprawiają, że jest to bardzo atrakcyjna platforma programistyczna.

Na rysunku 1.2 został ukazany stos programowy Androida (więcej informacji na ten temat można znaleźć w podrozdziale „Stos programowy Androida”).



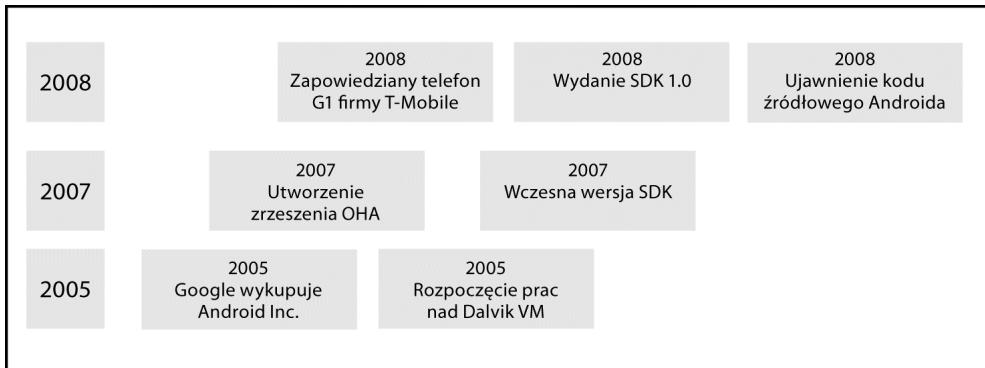
Rysunek 1.2. Wysokopoziomowy widok stosu programowego Androida

## Początki historii Androida

Dla telefonów komórkowych stworzono wiele różnych systemów operacyjnych, takich jak Symbian OS, Windows Mobile firmy Microsoft, Mobile Linux, iPhone OS (napisany na podstawie systemu Mac OS X), Moblin (firmy Intel) oraz wiele innych opatentowanych środowisk. Do tej pory żaden z tych systemów nie stał się formalnym standardem. Dostępne interfejsy API oraz środowiska projektowania aplikacji dla urządzeń przenośnych są zbyt ograniczone i pozostają w tyle w porównaniu z analogicznymi strukturami dostępnymi dla stacji roboczych.

W przeciwieństwie do pozostałych systemów operacyjnych, Android miał być otwarty, przystępny, o jawnym kodzie źródłowym oraz, co ważniejsze, miał zapewniać nowoczesny, scentralizowany i spójny szkielet projektowania.

W 2005 roku firma Google wykupiła małe przedsiębiorstwo Android Inc., które rozpoczęło projektowanie platformy Android (rysunek 1.3). Wśród najważniejszych pracowników firmy Android Inc. byli w owym czasie Andy Rubin, Rich Miner, Nick Sears oraz Chris White.



**Rysunek 1.3.** Początki historii Androida

Pod koniec 2007 roku grupa czołowych przedsiębiorstw utworzyła wokół platformy Android kластer przemysłowy Open Handset Alliance (<http://www.openhandsetalliance.com>). Niektórzy członkowie tego to:

- Sprint Nextel,
- T-Mobile,
- Motorola,
- Samsung,
- Sony Ericsson,
- Toshiba,
- Vodafone,
- Google,
- Intel,
- Texas Instruments.

Do 2011 roku liczba członków tej grupy znacznie się zwiększyła (jest ich obecnie ponad 80), co można sprawdzić na stronie zrzeszenia Open Handset Alliance.

Zgodnie z informacjami zawartymi w witrynie klastra jednym z jego celów jest szybkie wprowadzanie innowacji oraz lepsza odpowiedź na potrzeby konsumentów w przestrzeni mobilnej, a jednym z pierwszych ważnych osiągnięć był Android. Został on zaprojektowany w celu zaspokojenia potrzeb operatorów sieci komórkowych, producentów urządzeń oraz twórców aplikacji. Członkowie zrzeszenia zobowiązały się udostępnić tę istotną własność intelektualną po przez zastosowanie w stosunku do Androida warunków licencji Apache License 2.0<sup>1</sup>.

<sup>1</sup> Apache License 2.0 jest licencją wolnego oprogramowania autorstwa Apache Software Foundation. Licencja ta dopuszcza użycie kodu źródłowego zarówno na potrzeby wolnego oprogramowania, jak i zamkniętego oprogramowania komercyjnego — przyp. red.

Zestaw Android SDK został wydany jako „wczesna wersja” w listopadzie 2007 roku. We wrześniu 2008 roku firma T-Mobile zapowiedziała wydanie T-Mobile G1, pierwszego smartfonu bazującego na platformie Android. Kilka dni później firma Google ogłosiła wydanie zestawu Android SDK Release Candidate 1.0<sup>2</sup>. W październiku 2008 roku firma Google udostępniła kod źródłowy platformy Android w ramach licencji Apache. Pod koniec 2010 roku firma Google wydała zestaw Android SDK w wersji 2.3 dla smartfonów. Zestawowi temu nadano nazwę kodową Gingerbread. W marcu 2011 roku został on zaktualizowany do wersji 2.3.3. Na początku 2011 roku została wydana zoptymalizowana wersja Androida (w wersji 3.0) przeznaczona dla tabletów, nosząca nazwę kodową Honeycomb. Jednym z pierwszych tabletów działających pod kontrolą tej wersji systemu operacyjnego jest Motorola XOOM.

Jednym z najważniejszych celów twórców Androida było umożliwienie współpracy różnych aplikacji ze sobą, a także wielokrotnego wykorzystywania składników jednej aplikacji przez inną. Takie używanie fragmentów innych programów dotyczy nie tylko usług, lecz również danych oraz interfejsu UI (ang. *User Interface* — interfejs użytkownika). W efekcie Android posiada szereg funkcji konstrukcyjnych, dzięki którym stał się w rzeczywistości sposób otwarty.

Android wcześnie przyciągnął wielu zwolenników. Utrzymał również zainteresowanie programistów, gdyż posiada w pełni rozwinięte narzędzia, dzięki którym można wykorzystywać — poprzez model przetwarzania w chmurze (ang. *cloud computing*) — udostępnione zasoby sieciowe. Twórcy Androida usprawnili również funkcjonowanie lokalnych magazynów danych w samym urządzeniu przenośnym. Na ciepłe przyjęcie Androida wpłynęła również możliwość obsługi relacyjnych baz danych przez urządzenia przenośne.

Android w wersjach 1.0 oraz 1.1 (2008 rok) nie posiadał możliwości obsługi klawiatury programowej, więc urządzenia musiały być wyposażone w fizyczne przyciski. Funkcja ta została wprowadzona w zestawie Android SDK 1.5 w kwietniu 2009 roku wraz z innymi dodatkami, takimi jak zaawansowane możliwości nagrywania multimedialnych, widżety oraz aktywne foldery.

We wrześniu 2009 roku pojawiła się wersja 1.6 systemu Android, a w przeciągu miesiąca została wydana wersja opatrzona numerem 2.0, dzięki czemu nastąpił przedświąteczny wysyp urządzeń obsługujących ten system. W tej wersji zaprezentowano funkcje zaawansowanego wyszukiwania danych oraz przetwarzania tekstu na mowę.

Dzięki obsłudze języka HTML 5 system Android 2.0 posiada interesujące możliwości wykorzystania stron WWW. Interfejs API kontaktów uległ znacznemu usprawnieniu. Dodano obsługę formatu Flash. Codziennie wydaje się coraz więcej aplikacji opartych na Androidzie, pojawiają się również coraz nowsze rodzaje niezależnych sieciowych sklepów z aplikacjami. Można już zakupić od dawna wyczekiwane komputery typu tablet, bazujące na systemie Android.

W wersji 2.3 Androida wśród najważniejszych funkcji można znaleźć takie, jak zdalne usuwanie zabezpieczonych danych przez administratorów, możliwość korzystania z aparatu oraz kamery w warunkach słabego oświetlenia czy korzystanie z hotspotów WiFi. Warto też zwrócić uwagę na znaczną poprawę wydajności, usprawnione działanie interfejsu Bluetooth, możliwość opcjonalnej instalacji aplikacji na karcie SD, możliwość korzystania ze środowiska OpenGL ES 2.0, usprawnione tworzenie kopii zapasowych, poprawioną funkcję wyszukiwania, obsługę standardu NFC (ang. *Near Field Communication* — komunikacja bliskiego pola) umożliwiającą przeprowadzanie operacji na kartach kredytowych, znacznie usprawnioną obsługę czujników oraz wykrywania ruchu (podobnie jak w przypadku konsoli Wii), czat wideo oraz poprawiony Android Market.

<sup>2</sup> Release Candidate to niemal finalna wersja oprogramowania, w której mogą jeszcze zostać wprowadzone drobne poprawki — przyp. tłum.

Najnowsze wcielenie Androida, oznaczone numerem 3.0, jest przeznaczone do obsługi urządzeń typu tablet oraz o wiele późniejszych procesorów dwurdzeniowych, takich jak Nvidia Tegra2. Najważniejszą funkcją udostępnioną w tej wersji jest obsługa większych wyświetlaczów. Wprowadzono zupełnie nową koncepcję prezentowania treści w aplikacjach, zwaną „fragmentami”. Te cechy stanowią atrakcyjnością Androida 3.0. Wprowadzono również więcej funkcji spotykanych dotychczas w komputerach stacjonarnych, na przykład klas Actionbar lub możliwość przeciągania elementów. Znacznej modernizacji uległy widżety ekranu startowego. Dostępnych jest teraz więcej kontrolek interfejsu użytkownika. W zakresie grafiki trójwymiarowej środowisko OpenGL zostało zaopatrzone w interfejs Renderscript, dalej rozwijający wersję ES 2.0. Jest to znakomite wprowadzenie na rynek dla tabletów pracujących w systemie Android.

## Zapoznanie się ze środowiskiem Dalvik VM

Z uwagi na swoje zaangażowanie w projekt Android firma Google skoncentrowała się na wdrażaniu technik optymalizacyjnych dla niskonapięciowych urządzeń typu handheld. Urządzenia te są opóźnione w stosunku do ich większych odpowiedników o jakieś osiem do dziesięciu lat, jeśli chodzi o pamięć oraz szybkość. Mają także ograniczoną moc obliczeniową. Wymagania dotyczące wydajności są bardzo surowe, przez co projektanci muszą optymalizować wszystkie możliwe elementy aplikacji. Jeżeli przyjrzeć się liście pakietów Androida, można zauważyć, że są one doskonale wyposażone i że jest ich bardzo wiele.

Powyższe problemy sprawiły, że firma Google musiała ponownie przyjrzeć się w nowym świetle standardowej implementacji JVM. Osobą odpowiedzialną za implementację JVM firmy Google jest Dan Bornstein, twórca Dalvik VM (Dalvik jest nazwą islandzkiego miasteczka). Dalvik VM pobiera wygenerowane pliki klas Java i przetwarza je na jeden lub więcej plików wykonywalnych Dalvik (.dex). Następnie wykorzystuje powtarzające się informacje z różnych plików klas i w ten sposób wydajnie zmniejsza zużycie pamięci o połowę w stosunku do tradycyjnego pliku .jar (nieskompresowanego).

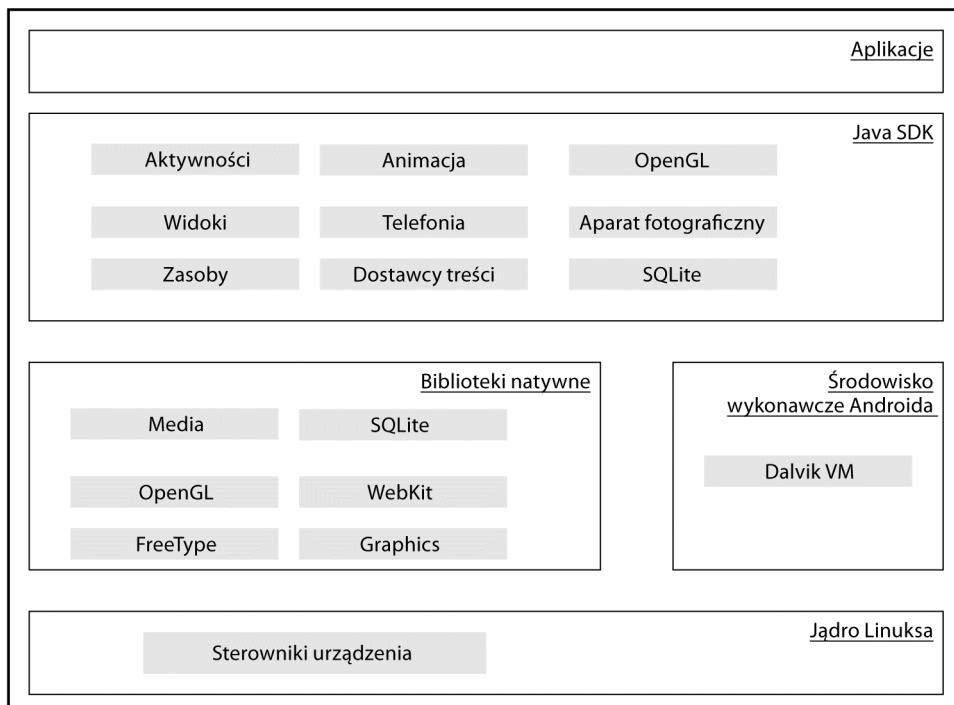
Firma Google usprawniła także zarządzanie zbędnymi plikami w środowisku Dalvik JVM, jednak we wczesnych edycjach zdecydowała się pominąć kompilator JIT (ang. *Just-In-Time Compiler*; kompilatory tego typu dokonują tzw. komplikacji w locie, tłumacząc kod bajtowy na kod maszynowy danego procesora — *przyp. red.*). Został on dodany w wersji 2.3. Z raportów wynika, że w pewnych miejscach potrafi on przyspieszyć wydajność nominalną od dwóch do pięciu razy, a w przypadku aplikacji ogólnego przeznaczenia — od 10 do 20%.

Dalvik VM wykorzystuje inną metodę generowania kodu maszynowego, w której podstawowymi jednostkami przechowywania danych są rejestrzy, a nie stosy. Firma Google ma nadzieję, że w ten sposób liczba instrukcji zostanie zmniejszona o 30%. Należy zauważyć, że w przypadku Androida ostateczny plik wykonywalny oparty jest nie na kodzie bajtowym Java, a na plikach .dex, właśnie dzięki środowisku Dalvik VM. Oznacza to, że nie można bezpośrednio uruchomić kodu bajtowego Java; najpierw należy uruchomić pliki klas Java, a następnie dokonać ich konwersji na linkowalne pliki .dex.

Takie restrykcyjne podejście do problemów z wydajnością dotyczy także pozostałych elementów zestawu Android SDK. Na przykład szeroko wykorzystuje on język XML do definiowania wyglądu interfejsu użytkownika, ale przed zapisaniem na urządzeniu pliki XML zostają przekształcone na pliki binarne. Android posiada specjalne mechanizmy umożliwiające korzystanie z tych plików XML.

## Stos programowy Androida

Do tej pory zajmowaliśmy się historią Androida oraz jego funkcjami optymalizacyjnymi, w tym także środowiskiem Dalvik VM; wspomnieliśmy również o dostępności stosu programowego Java. W tym podrozdziale zajmiemy się aspektem projektowania w Androide. Najlepszym miejscem, od którego można rozpocząć, jest rysunek 1.4.



**Rysunek 1.4.** Szczegółowy stos programowy zestawu Android SDK

Rdzeniem platformy Android jest jądro Linuksa, zapewniające obsługę sterowników urządzenia, dostęp do zasobów, zarządzanie energią oraz innymi zadaniami systemu operacyjnego. Sterowniki urządzenia obejmują ekran, aparat fotograficzny, klawiaturę, WiFi, pamięć flash, audio oraz komunikację IPC (ang. *Inter-Process Communication* — komunikacja międzyprocesowa; pojęcie to oznacza wymianę danych pomiędzy procesami systemu operacyjnego). Chociaż rdzeniem systemu jest Linux, większość aplikacji — jeśli nie wszystkie — w urządzeniach takich jak Motorola Droid jest projektowana w języku Java oraz uruchamiana w środowisku Dalvik VM.

Na kolejnym poziomie, ponad rdzeniem Linuksa, umieszczono dużą liczbę bibliotek C/C++, wśród których znajdują się biblioteki OpenGL, WebKit, FreeType, SSL (ang. *Secure Sockets Layer*; protokół służący do bezpiecznej transmisji zaszyfrowanego strumienia danych), biblioteka wykonawcza języka C (libc), SQLite oraz Media. Biblioteka systemowa języka C oparta na systemie Berkeley Software Distribution (BSD) jest dopasowana (zmniejszono ją o ponad połowę w stosunku do pierwotnego rozmiaru) do urządzeń posiadających wbudowany system basujący na Linuksie. Biblioteki multimedii są oparte na standardzie OpenCORE PacketVideo ([www.packetvideo.com/](http://www.packetvideo.com/)). Zapewniają one obsługę nagrywania oraz odtwarzania formatów audio

i wideo. Biblioteka Surface Manager kontroluje dostęp do systemu wyświetlania, a także obsługuje grafikę dwu- oraz trójwymiarową. Prawdopodobnie wraz z nowymi wersjami systemu będą dodawane kolejne biblioteki natywne.

Biblioteka WebKit odpowiada za obsługę przeglądarki; to właśnie ona obsługuje przeglądarki Google Chrome oraz Safari. Biblioteka FreeType zajmuje się obsługą czcionek. SQLite ([www.sqlite.org/](http://www.sqlite.org/)) jest relacyjną bazą danych dostępną na samym urządzeniu przenośnym. Jest to także forma niezależnej, posiadającej jawny kod źródłowy technologii relacyjnej bazy danych, niezwiązanej bezpośrednio z Androidem. Można również pobrać narzędzia przeznaczone dla bazy SQLite i używać ich do baz danych Androida.

Większość szkieletu aplikacji uzyskuje dostęp do tych bibliotek podstawowych poprzez środowisko Dalvik VM, które stanowi bramę do platformy Android. Jak zostało wyjaśnione w poprzednich podrozdziałach, środowisko Dalvik zostało zoptymalizowane do jednoczesnego uruchamiania wielu instancji wirtualnych maszyn. Podczas uzyskiwania dostępu do podstawowych bibliotek przez aplikacje Java każda z tych aplikacji otrzymuje własnąinstancję maszyny VM.

Główne biblioteki interfejsu API środowiska Java w Androidzie obejmują telefony, zasoby, lokacje, interfejs użytkownika, dostawców (dane) treści oraz menedżery pakietów (instalacja, zabezpieczenia i tak dalej). Programiści projektują aplikacje dla użytkownika końcowego w górnej warstwie tego interfejsu API. Przykładami takich aplikacji są Home, Contacts, Phone, Browser i tak dalej.

Android posiada również własną bibliotekę do obsługi grafiki Google 2D — Skia, którą napisano w językach C i C++. Skia jest również elementem rdzenia przeglądarki Google Chrome. Jednak interfejsy API odpowiedzialne za grafikę trójwymiarową bazują w Androidzie na implementacji pakietu OpenGL ES grupy Khronos (<http://www.khronos.org>). Pakiet ten zawiera podzbiory funkcji OpenGL, których adresatami są wbudowane systemy.

Jeśli zaś chodzi o multimedia, Android obsługuje najpopularniejsze formaty obrazów, dźwięków oraz wideo. Z perspektywy sieci bezprzewodowych dostępne są interfejsy API obsługujące sieci Bluetooth, EDGE, 3G, WiFi oraz telefonie GSM (ang. *Global System for Mobile Communication* — globalny system komunikacji mobilnej), w zależności od parametrów sprzętowych urządzeń.

## Projektowanie aplikacji użytkownika końcowego za pomocą zestawu Android SDK

W niniejszym podrozdziale zostaną zaprezentowane wysokopoziomowe interfejsy Java, służące do projektowania aplikacji przeznaczonych dla użytkownika końcowego na platformie Android. Krótko omówimy emulator Androida, podstawowe składniki środowiska Android, programowanie interfejsu UI, usługi, multimedia, telefonię, animacje oraz technologię OpenGL.

### Emulator Androida

Zestaw Android SDK wyposażono we wtyczkę organizacji Eclipse, nazwaną narzędziami ADT (ang. *Android Development Tools* — narzędzia projektowe dla środowiska Android). To środowisko IDE (ang. *Integrated Development Environment* — zintegrowane środowisko projektowe) służy do projektowania, usuwania błędów oraz testowania aplikacji Java (szczegółowe informacje na temat narzędzi ADT znajdują się w rozdziale 2.). Można również używać zestawu Android SDK bez narzędzi ADT; wykorzystywane są wtedy narzędzia wiersza poleceń. Obydwie metody

pozwalały na uruchamianie, usuwanie błędów oraz testowanie aplikacji. W 90% przypadków do projektowania aplikacji nie będzie potrzebne nawet fizyczne urządzenie. Emulator Androida naśladuje większość funkcji urządzenia. Ograniczenia emulatora są związane z połączeniami USB, wykonywaniem zdjęć, nagrywaniem sekwencji wideo, obsługą słuchawek, symulacją baterii, połączenia Bluetooth, WiFi, NFC oraz z obsługą środowiska OpenGL ES 2.0.

Emulator Androida spełnia swoje zadanie dzięki posiadającej jawnego kod źródłowy technologii „emulacji procesora”, nazwanej QEMU, zaprojektowanej przez Fabrice'a Bellarda (<http://bellard.org/qemu>). Ta sama technologia umożliwia emulację jednego systemu operacyjnego wewnętrz drugiego bez względu na działanie procesora. QEMU pozwala na emulację na poziomie jednostki centralnej.

Dzięki emulatorowi Androida procesor przechodzi w tryb technologii ARM (ang. *Advanced RISC Machine* — zaawansowana maszyna RISC). ARM jest architekturą procesora 32-bitowego, działającego w oparciu o technologię RISC (ang. *Reduced Instruction Set Computer* — komputer z ograniczonym zestawem instrukcji). Technologia ta umożliwia uzyskanie prostoty projektowania oraz szybkości poprzez ograniczenie liczby instrukcji w zestawie. Emulator powoduje uruchomienie systemu Linux, dostępnego na platformie Android, na takim symulowanym procesorze.

Technologia ARM jest szeroko stosowana w handheldach oraz w innych urządzeniach elektronicznych, w których istotne jest niskie zużycie energii. Większość rynku urządzeń przenośnych wykorzystuje procesory oparte na tej architekturze.

Więcej informacji dotyczących emulatora można znaleźć w dokumentacji zestawu Android SDK, dostępnej na stronie <http://developer.android.com/guide/developing/tools/emulator.html>.

## Interfejs użytkownika na platformie Android

Android wykorzystuje szkielet interfejsu UI bardzo podobny do analogicznych szkieletów używanych w komputerach osobistych, jednak jest on nowocześniejszy i bardziej asynchroniczny. W istocie interfejs UI Androida stanowi czwartą generację szkieletów interfejsów użytkownika, jeżeli uznać tradycyjny interfejs API systemu Windows, napisany w języku C, za pierwszą generację, a stworzony w języku C++ zbiór klas MFC (ang. *Microsoft Foundation Classes*) za drugą. Szkielet UI biblioteki Swing, napisany w języku Java, można uznać za trzecią generację. Zapewniono tu o wiele większą elastyczność projektowania, niż to miało miejsce w przypadku zbioru MFC. Interfejs UI Androida, JavaFX, Microsoft Silverlight oraz język XUL (ang. *XML User Interface Language* — język XML interfejsu użytkownika) należą do czwartego pokolenia szkieletu UI, w którym interfejs użytkownika jest deklaracyjny oraz tworzony niezależnie.

### Uwaga!

W Androidzie programujemy aplikacje za pomocą współczesnego paradymatu interfejsu UI, nawet jeśli urządzenie docelowe jest handheldem.

Programowanie w interfejsie UI Androida wiąże się z zadeklarowaniem interfejsu przy użyciu plików XML. Następnie można wczytać takie definicje widoków XML jako okna w aplikacji interfejsu użytkownika. Nawet listy opcji aplikacji są wczytywane z plików XML. Ekranы lub okna w Androidzie są często nazywane **aktywnościami**, składającymi się z wielu widoków, potrzebnych użytkownikowi do wykonania czynności. **Widoki** są podstawowymi blokami budulcowymi interfejsu użytkownika, można je ze sobą łączyć w celu uzyskania **grup widoków**. Widoki wykorzystują w swoim wnętrzu znajome Czytelnikowi pojęcia kanw, rysowania oraz interakcji użytkownika. Aktywność obsługująca takie złożone widoki, zawierająca widoki lub

grupy widoków, jest logicznym, zamiennym składnikiem interfejsu UI w Androidzie. W wersji Android 3.0 wprowadzono nową koncepcję interfejsu użytkownika — **fragmenty**, dzięki której programiści mogą dostosowywać widoki i funkcje do wyświetlaczów tabletów. Tablety posiadają na tyle duże ekranы, aby można było przeprowadzać czynności wykorzystujące wiele obszarów na ekranie, a fragmenty wprowadzają właśnie taki podział wyświetlacza na części.

Jedną z najważniejszych koncepcji szkieletu Androida jest zarządzanie cyklem życia okien aktywności. Do tego służą protokoły, dzięki którym Android może modyfikować stan okien aktywności, kiedy użytkownik je ukrywa, przywraca, zatrzymuje i zamyka. Te podstawowe zasady staną się bardziej zrozumiałe po przeczytaniu rozdziału 2., stanowiącego również wprowadzenie do konfigurowania środowiska projektowego Android.

## Podstawowe składniki Androida

Szkielet interfejsu UI, wraz z innymi elementami platformy Android, opiera się na nowej koncepcji, zwanej **intencją** (ang. *intent*). Intencja jest połączeniem takich pomysłów, jak informacje wywoływane w oknach, działania, modele publikowania i (lub) subskrybowania, komunikacja międzyprocesowa, a także rejestyry aplikacji. Poniżej przedstawiono przykład wykorzystania klasy Intent do wywołania lub uruchomienia przeglądarki internetowej:

```
public static void invokeWebBrowser(Activity activity)
{
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("http://www.google.com"));
    activity.startActivity(intent);
}
```

Ten przykładowy kod powoduje, że Android — poprzez intencję — otwiera odpowiednie okno, w którym będzie wyświetlana zawartość strony WWW. W zależności od listy dostępnych przeglądarek zainstalowanych w urządzeniu Android wybierze najodpowiedniejszą. Intencje zostały szczegółowo omówione w rozdziale 5.

Android zapewnia także rozbudowaną obsługę **zasobów**, obejmujących znajome kategorie elementów oraz plików, na przykład ciągi tekstowe oraz mapy bitowe, jak również mniej znane składniki, takie jak definicje widoku oparte na języku XML. Są one wykorzystywane w nowoczesny, łatwy, intuicyjny oraz wygodny dla użytkownika sposób. Poniżej został zamieszczony przykład, w którym identyfikatory zasobów zostają automatycznie wygenerowane dla zasobów zdefiniowanych w plikach XML:

```
public final class R {
    public static final class attr { }
    public static final class drawable {
        public static final int myanimation=0x7f020001;
        public static final int numbers19=0x7f02000e;
    }

    public static final class id {
        public static final int textViewId1=0x7f080003;
    }
    public static final class layout {
        public static final int frame_animations_layout=0x7f030001;
        public static final int main=0x7f030002;
    }
    public static final class string {
```

```
    public static final int hello=0x7f070000;
}
}
```

Każdy identyfikator wygenerowany w tej klasie odpowiada albo elementowi znajdującemu się w pliku XML, albo samemu plikowi. Można teraz używać tych wygenerowanych identyfikatorów zamiast definicji XML. Taka pośrednia droga jest bardzo przydatna podczas procesu lokalizacji (w rozdziale 3. zostały szczegółowo omówione zasoby oraz plik *R.java*).

Kolejną nową koncepcją w Androidzie jest **dostawca treści**. Jest to abstrakcyjne ujęcie źródła danych, przyjmujące formę wystawcy oraz adresata usług RESTful. Stanowiąc jego podstawę baza danych SQLite sprawia, że jest to potężne narzędzie dla projektantów aplikacji. W rozdziale 4. omówimy zagadnienie dostawców treści, zaś w rozdziałach 3., 4. i 5. wyjaśnimy, w jaki sposób intencje, zasoby oraz dostawcy treści gwarantują otwartość platformy Android.

## Zaawansowane koncepcje interfejsu użytkownika

Stwierdziliśmy już, że decydującą rolę w opisie interfejsu UI Androida stanowi język XML. Zobaczmy, w jaki sposób można dzięki niemu stworzyć prosty układ graficzny, zawierający widok tekstu:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android>
<TextView android:id="@+id/textViewId"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

W celu załadowania układu graficznego do okna aktywności wykorzystamy identyfikator wygenerowany dla tego pliku XML (proces ten zostanie przeanalizowany w rozdziale 6.). Android obsługuje także menu (to zagadnienie zostanie rozwinięte w rozdziale 7.) — od standardowych do kontekstowych. Praca przy takich menu jest bardzo wygodna, gdyż są one również wczytywane jako pliki XML, a ich identyfikatory zasobów są generowane automatycznie. Menu można deklarować w pliku XML w następujący sposób:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa wykorzystuje domyślną kategorię. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_clear"
            android:orderInCategory="10"
            android:title="wyczyść" />
        <item android:id="@+id/menu_show_browser"
            android:orderInCategory="5"
            android:title="wyświetl przeglądarkę" />
    </group>
</menu>
```

Android obsługuje okna dialogowe, z których wszystkie są asynchroniczne. Mogą one stanowić nowego rodzaju wyzwanie dla projektantów przyzwyczajonych do synchronicznych, modalnych okien dialogowych stosowanych w niektórych szkieletach okienkowych. Menu zajmiemy się szerzej w rozdziale 7., a oknami dialogowymi w rozdziale 8., poświęconym również sposobem korzystania z protokołów asynchronicznych okien dialogowych.

Android obsługuje także animacje w formie stosu interfejsu UI, opartego na widokach oraz rysowanych obiektach. Są dostępne dwa rodzaje animacji: **animowane przejścia** (ang. *tweening*) oraz rysowane klatka po klatce. Animowane przejścia polegają na rysowaniu obrazów znajdujących się pomiędzy kluczowymi klatkami animacji. Osiąga się to poprzez zmianę średnich wartości w regularnych odstępach czasu oraz ponowne rysowanie powierzchni. Animacja klatka po klatce występuje wtedy, gdy jest rysowana seria klatek w regularnych odstępach czasowych. Android umożliwia wykorzystanie obydwu technik animacji poprzez wywoływanie zwrotne, stosowanie interpolatorów oraz macierzy transformacji.

Ponadto istnieje możliwość zdefiniowania tych animacji w pliku zasobów XML. W poniższym przykładzie seria ponumerowanych obrazów jest odtwarzana w animacji klatka po klatce:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/numbers11" android:duration="50" />
    ...
    <item android:drawable="@drawable/numbers19" android:duration="50" />
</animation-list>
```

Dostępne biblioteki graficzne obsługują standardowe macierze transformacji, dzięki czemu otrzymujemy funkcje skalowania, przenoszenia oraz obracania. Klasa *Camera* w bibliotece graficznej posiada funkcje głębi oraz rzutowania, pozwalające na symulację trzech wymiarów w płaszczyźnie dwuwymiarowej (animacja została omówiona w rozdziale 16.).

W Androidzie jest dostępna również grafika trójwymiarowa dzięki zaimplementowaniu standardów OpenGL ES 1.0 oraz 2.0. Standardy te, podobnie jak OpenGL, są płaskimi interfejsami API opartymi na języku C. Ponieważ interfejs API środowiska Android SDK jest zaprogramowany w języku Java, musi korzystać z łącznika Java, żeby uzyskać dostęp do biblioteki OpenGL ES. Taki łącznik został już zdefiniowany w środowisku Java ME poprzez specyfikację JSR 239 dla biblioteki OpenGL ES, a w Androidzie została zaimplementowana ta sama technologia. Dla kogoś niezaznajomionego z programowaniem OpenGL może to być dość trudne do nauczenia się, jednak w rozdziale 20. zostały omówione podstawy, pozwalające na rozpoczęcie programowania w OpenGL dla systemu Android. Począwszy od wersji 3.0 Androida, został wprowadzony mechanizm skryptowy, rozbudowujący środowisko OpenGL do wersji ES 2.0.

Android jest związany z wieloma koncepcjami, które krążą wokół idei *informacji w zasięgu ręki*, dostępnych na ekranie startowym. Pierwszą taką koncepcją są **aktywne foldery**. Dzięki nim istnieje możliwość opublikowania zbioru elementów na ekranie startowym w formie folderu. Zawartość tego zbioru jest odświeżana zgodnie z aktualizacją tworzących go danych. Dane te mogą się znajdować w telefonie lub pochodzić z internetu (aktywne foldery zostały omówione w rozdziale 21.).

Drugą koncepcją są **widżety ekranu startowego**. Są one stosowane do zobrazowania informacji na ekranie startowym za pomocą interfejsu UI widżetu. Informacje te mogą ulegać zmianie w regularnych odstępach czasu. Za przykład może posłużyć wyświetlanie liczby wiadomości e-mail w bazie danych. W rozdziale 22. przedstawiono tematykę widżetów ekranu startowego. W wersji 3.0 widżety ekranu startowego zostały bardziej rozbudowane i zawierają widoki w postaci list, które są aktualizowane w trakcie modyfikowania danych związanych z tymi widżetami. Usprawnienia te zostały omówione w rozdziale 31.

**Zintegrowane przeszukiwanie Androida** jest trzecią koncepcją związaną z ekranem startowym. Za pomocą zintegrowanego przeszukiwania można wyszukiwać zawartość zarówno w urządzeniu, jak i w internecie. Przeszukiwanie Androida to także możliwość uruchamiania poleceń w oknie wyszukiwania. Koncepcję tą zajmiemy się w rozdziale 23.

W Androidzie została również udostępniona obsługa ekranu dotykowego oraz gestów opartych na ruchach palców po wyświetlaczu urządzenia. Każdy rodzaj ruchu po ekranie można zapisać jako gest. Następnie taki gest zostaje powiązany w aplikacji z określonymi czynnościami. Ekrany dotykowe oraz gesty zostały dokładnie przeanalizowane w rozdziale 25.

Coraz ważniejszym elementem obsługi urządzeń mobilnych stają się czujniki. Są one omówione w rozdziale 26.

Kolejną niezbędną innowacją wymaganą dla urządzeń mobilnych jest dynamiczna natura ich konfiguracji. Na przykład bardzo łatwo zmienić tryb przeglądania pomiędzy orientacją pionową a poziomą. Można również zadokować urządzenie przenośne i korzystać z niego jak z laptopa. W Androidzie 3.0 zostało wprowadzone pojęcie fragmentów, dzięki którym można skutecznie definiować takie różnorodne zachowania. Rozdział 29. został poświęcony fragmentom.

Omówiliśmy również nową funkcję pasków menu, zaprezentowaną w wersji 3.0, której przyjrzyjmy się w rozdziale 30. Koncepcja pasków menu w Androidzie zrównuje ją z paradigmatem pasków menu znanych z komputerów stacjonarnych. W rozdziale 25. omówiliśmy funkcję przeciągania elementów (dawny sposób), temat ten poruszoно także w rozdziale 31. (przeciąganie elementów sposobem wprowadzonym w Androidzie 3.0).

Poza środowiskiem Android SDK dostępnych jest wiele niezależnych innowacji, usprawniających oraz ułatwiających proces projektowania. Przykładami są narzędzia XML/VM, PhoneGap oraz Titanium.

## Składniki usług w Androidzie

Podstawowym założeniem twórców platformy Android jest bezpieczeństwo. Zabezpieczenia są uwzględniane na wszystkich etapach cyklu życia aplikacji — począwszy od rozoważań na temat reguł czasu projektowania, a skończywszy na testach granicy rozruchowej. Kwestią bezpieczeństwa i uprawnień zajmiemy się w rozdziale 10.

W rozdziale 11. zaprezentujemy sposoby tworzenia oraz wykorzystywania usług w Androidzie, w szczególności usług HTTP. Zostanie w nim również omówiona komunikacja międzyprocesowa (komunikowanie się aplikacji pomiędzy sobą w obrębie jednego urządzenia).

Jednym z ciekawszych składników zestawu Android SDK jest usługa zorientowana na położenie. Dzięki niej projektanci interfejsów API mogą wyświetlać oraz kontrolować mapy, a także otrzymywać w czasie rzeczywistym informacje o lokalizacji urządzenia. Koncepcje te zostały omówione w rozdziale 17.

## Składniki multimedii oraz telefonii w Androidzie

Android został zaopatrzony w interfejsy API pozwalające na wykorzystywanie składników audio, wideo oraz telefonii. Interfejs API telefonii zostanie omówiony w rozdziale 18. W rozdziale 19. przyjrzymy się dokładnie interfejsom API audio i wideo. Wraz z wersją 2.0 do Androida wprowadzono silnik przetwarzania tekstu na mowę Pico. Został mu poświęcony rozdział 24.

Ostatnia, lecz nie najmniej istotna rzecz, o której należy wspomnieć, to fakt, że Android łączy te wszystkie koncepcje w aplikacji poprzez utworzenie pliku XML, definiującego zawartość pakietu tej aplikacji. Plik ten jest znany jako manifest aplikacji (*AndroidManifest.xml*). Przykład:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ai.android.HelloWorld"
```

```
        android:versionCode="1"
        android:versionName="1.0.0">
<application android:icon="@drawable/icon" android:label="@string/app_name">
    <activity android:name=".HelloWorld"
              android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
</manifest>
```

W pliku manifeście zostają zdefiniowane aktywności, następuje rejestracja dostawców treści oraz usług, a także są deklarowane uprawnienia. W dalszej części książki podczas omawiania różnorodnych koncepcji będą pojawiały się kolejne szczegóły dotyczące pliku manifestu.

## Pakiety Java w Androidzie

Można szybko ocenić zawartość platformy Android, przyjrzałszy się strukturze pakietów Java. Ponieważ Android różni się od standardowej wersji JDK, warto wiedzieć, jakie składniki są obsługiwane, a jakie zostały pominięte. Poniższa lista stanowi krótkie omówienie ważnych pakietów dołączonych do zestawu Android SDK:

- **android.app.** Jest to implementacja modelu Application dla Androida. Wśród podstawowych klas można znaleźć Application, definiującą semantykę rozpoczęcia oraz zatrzymania aplikacji. Dostępnych jest również wiele innych klas związanych z aktywnością, fragmentów, kontrolek, okien dialogowych, ostrzeżeń oraz powiadomień.
- **android.bluetooth.** Dostarcza dużą liczbę klas pozwalających na wykorzystanie funkcji Bluetooth. Wśród głównych klas należy wymienić BluetoothAdapter, BluetoothDevice, BluetoothSocket, BluetoothServerSocket oraz BluetoothClass. Klasa BluetoothAdapter pozwala kontrolować lokalnie zainstalowany adapter Bluetooth. Można na przykład go włączyć, wyłączyć lub skonfigurować w stan wykrywania. Klasa BluetoothDevice symbolizuje zdalne urządzenie Bluetooth, do którego następuje podłączenie. Są wykorzystywane dwa gniazda Bluetooth do ustanowienia połączenia pomiędzy urządzeniami. Klasa BluetoothClass reprezentuje rodzaj urządzenia, z którym zostaje ustanowione połączenie.
- **android.content.** Pakiet ten stanowi implementację koncepcji dostawców treści. Dzięki nim można wyodrębnić dostęp do danych z magazynu danych. W pакiecie tym mieszczą się również podstawowe koncepcje dotyczące intencji oraz identyfikatorów URI (ang. *Uniform Resource Identifier* — ujednolicony identyfikator zasobów) Androida.
- **android.content.pm.** Implementacja klas związanych z menedżerem pakietów. Menedżer pakietów zbiera informacje na temat uprawnień, zainstalowanych pakietów, dostawców oraz usług, aplikacji oraz składników, takich jak aktywności.
- **android.content.res.** Zapewnia dostęp do ustrukturyzowanych oraz nieustrukturyzowanych plików zasobów. Podstawowymi klasami są AssetManager (dla nieustrukturyzowanych zasobów) oraz Resources.

- **android.database.** Wprowadza koncepcję abstrakcyjnej bazy danych. Podstawowym interfejsem jest biblioteka Cursor.
- **android.database.sqlite.** Wprowadza koncepcje z pakietu *android.database* w formie fizycznej bazy danych SQLite. Podstawowymi klasami są tutaj SQLiteCursor, SQLiteDatabase, SQLiteQuery, SQLiteQueryBuilder oraz SQLiteStatement, jednak większa część interakcji będzie przeprowadzana z klasami abstrakcyjnego pakietu *android.database*.
- **android.gesture.** Pakiet ten obejmuje większość klas i interfejsów wymaganych do pracy ze zdefiniowanymi gestami. Podstawowe klasy to: Gesture, GestureLibrary, GestureOverlayView, GestureStore, GestureStroke i GesturePoint. Klasa Gesture jest zbiorem klas GestureStroke oraz GesturePoint. Gesty są przechowywane w klasie GestureLibrary. Biblioteki gestów znajdują się w klasie GestureStore. Każdy gest posiada nazwę, dzięki czemu można go zidentyfikować jako działanie.
- **android.graphics.** Zostały tu umieszczone klasy Bitmap, Canvas, Camera, Color, Matrix, Movie, Paint, Path, Rasterizer, Shader, SweepGradient oraz Typeface.
- **android.graphics.drawable.** Wprowadza implementację protokołów rysowania oraz rysunki tła, a także umożliwia animowanie rysowanych obiektów.
- **android.graphics.drawable.shapes.** Wprowadziła takie biblioteki kształtów, jak ArcShape, OvalShape, PathShape, RectShape oraz RoundRectShape.
- **android.hardware.** Zaimplementowane zostają klasy związane z fizycznym aparatem fotograficznym. Klasa Camera jest reprezentacją sprzętowego aparatu, podczas gdy klasa android.graphics.Camera dotyczy koncepcji graficznej, zupełnie niezwiązanej z fizycznym urządzeniem.
- **android.location.** Przechowuje klasy Address, GeoCoder, Location, LocationManager oraz LocationProvider. Klasa Address reprezentuje uproszczony język XAL (ang. *Extensible Address Language* — rozszerzalny język adresów). Dzięki klasie GeoCoder można uzyskać współrzędne długości oraz szerokości geograficznej po wpisaniu adresu oraz odwrotnie. Klasa Location symbolizuje długość i szerokość geograficzną.
- **android.media.** Zawiera klasy MediaPlayer, MediaRecorder, Ringtone, AudioManager oraz FaceDetector. Klasa MediaPlayer, zapewniająca obsługę strumienia danych, służy do odtwarzania plików audio i wideo. Z kolei klasa MediaRecorder umożliwia rejestrowanie dźwięku oraz obrazu. Dzięki klasie Ringtone istnieje możliwość nagrywania krótkich fragmentów dźwiękowych, służących jako dzwonki lub powiadomienia. Do kontroli poziomu głośności wykorzystuje się klasę AudioManager, natomiast do rozpoznawania twarzy na mapie bitowej używana jest klasa FaceDetector.
- **android.net.** Dzięki temu pakietowi zostają zaimplementowane podstawowe sieciowe interfejsy API poziomu gniazda. Wśród podstawowych klas znajdują się Uri, ConnectivityManager, LocalSocket oraz LocalServerSocket. Warto również zauważyć, że Android obsługuje protokół HTTPS na poziomie przeglądarki oraz w warstwie sieciowej. W przeglądarce jest również obsługiwany język JavaScript.
- **android.net.wifi.** Zarządza połączeniami WiFi. Podstawowymi klasami są WifiManager oraz WifiConfiguration. Klasa WifiManager jest odpowiedzialna za wyświetlanie listy skonfigurowanych sieci oraz obecnie aktywnej sieci WiFi.

- **android.opengl.** Zawiera użytkowe klasy, powiązane z operacjami OpenGL ES w wersjach 1.0 i 2.0. Podstawowe klasy OpenGL ES są zaimplementowane w różnych zestawach pakietów, zapożyczonych ze specyfikacji JSR 239. Tymi pakietami są *javax.microedition.khronos.opengles*, *javax.microedition.khronos.egl* oraz *javax.microedition.khronos.nio*. Są to cienkie pakiety otaczające implementację OpenGL ES grupy Khronos, napisaną w językach C oraz C++.
- **android.os.** Pakiet ten reprezentuje usługi systemowe dostępne poprzez język programowania Java. Niektóre z istotnych klas to *BatteryManager*, *Binder*, *FileObserver*, *Handler*, *Looper* oraz *PowerManager*. *Binder* jest klasą umożliwiającą komunikację międzyprocesową. Dzięki klasie *FileObserver* system śledzi zmiany dokonywane w plikach. Klasa typu *Handler* są wykorzystywane do uruchamiania zadań w wątkach wiadomości, natomiast klasa *Looper* używa się do uruchamiania wątków wiadomości.
- **android.preference.** Dzięki temu pakietowi użytkownicy mogą zarządzać ustawieniami danej aplikacji w jednolity sposób. Podstawowymi klasami są *PreferenceActivity*, *PreferenceScreen*, a także różne klasy związane z samymi ustawieniami, na przykład *CheckBoxPreference* oraz *SharedPreferences*.
- **android.provider.** Składa się z zestawu predefiniowanych dostawców treści, przylegających do interfejsu *android.content.ContentProvider*. Do tego pakietu przynależą klasa *Contacts*, *MediaStore*, *Browser* oraz *Settings*. W tym zestawie interfejsów oraz klas są przechowywane metadane przygotowane dla podstawowych struktur danych.
- **android.sax.** Znajduje się tu wydajny zestaw analitycznych klas SAX (ang. *Simple API for XML* — prosty interfejs API dla języka XML). Podstawowymi klasami są *Element*, *RootElement* oraz pewna liczba interfejsów *ElementListener*.
- **android.speech.** Znajdują się tu stałe stosowane przy rozpoznawaniu mowy.
- **android.speech.tts.** Umożliwia konwersję tekstu na mowę. Główną klasą jest *TextToSpeech*. Istnieje możliwość wpisania tekstu oraz przekazania wystąpieniu tej klasy instrukcji przeczytania tekstu. Można skonfigurować wywoływanie monitora, na przykład po zakończeniu czytania. W Androidzie jest zastosowany silnik Pico TTS (ang. *Text to Speech*) firmy SVOX.
- **android.telephony.** Obiera klasy *CellLocation*, *PhoneNumberUtils* oraz *TelephonyManager*. Klasa *CellLocation* pozwala na określenie lokalizacji urządzenia, numeru telefonu, nazwy operatora sieci, rodzaju sieci, rodzaju telefonu oraz numeru seryjnego SIM (ang. *Subscriber Identity Module* — moduł identyfikacji abonenta).
- **android.telephony.gsm.** Pozwala na uzyskanie położenia geograficznego urządzenia na podstawie odległości od wieży operatora, a także przechowuje klasy obsługujące wysyłanie wiadomości SMS. W nazwie pakietu widnieje skrót GSM, gdyż ta właśnie technologia (globalnego systemu komunikacji mobilnej) jako pierwsza zdefiniowała standard wysyłania danych w formacie SMS.
- **android.telephony.cdma.** Wprowadza obsługę telefonii CDMA.
- **android.text.** Zawiera klasy przetwarzania tekstu.

- **android.text.method.** Obecne tu klasy umożliwiają wprowadzanie tekstu w celu kontroli zmian.
- **android.text.style.** Dostarcza wiele klas, dzięki którym można zastosować style do wprowadzanego tekstu.
- **android.utils.** Mieści klasy Log, DebugUtils, TimeUtils oraz Xml.
- **android.view.** Są tu umieszczone klasy Menu, View, ViewGroup, zestaw obiektów nasłuchujących oraz metod zwrotnych.
- **android.view.animation.** Zawiera obsługę animacji przejść. Wśród głównych klas znajdują się: klasa Animation, zestaw interpolatorów animacji oraz zestaw klas przeznaczonych dla animatora, między innymi AlphaAnimation, ScaleAnimation, TranslationAnimation i RotationAnimation. W wersji Android 3.0 został wprowadzony pakiet *android.animation*, który pełni podobną rolę, ale posiada szersze zastosowanie, ponieważ działa na obiektach, a nie na widokach.
- **android.view.inputmethod.** Zostaje dzięki niemu zaimplementowana architektura szkieletu metody wprowadzania danych.
- **android.webkit.** Przechowuje klasy reprezentujące przeglądarkę internetową. Podstawowymi klasami są WebView, CacheManager oraz CookieManager.
- **android.widget.** Znajdują się tu wszystkie kontrolki interfejsu UI, przeniesione w większości z klasy View. Głównymi klasami są Button, Checkbox, Chronometer, AnalogClock, DatePicker, DigitalClock, EditText, ListView, FrameLayout, GridView, ImageButton, MediaController, ProgressBar, RadioButton, RadioGroup, RatingButton, Scroller, ScrollView, Spinner, TabWidget, TextView, TimePicker, VideoView oraz ZoomButton.
- **com.google.android.maps.** Posiada klasy niezbędne do obsługi Google Maps, czyli MapView, MapController oraz MapActivity.

To część z najważniejszych pakietów, specyficznych dla Androida. Patrząc na tę listę, można dostrzec głębię podstawowej platformy Androida.

**Uwaga!**

Łącznie interfejs API Androida zawiera powyżej 40 pakietów oraz ponad 700 klas, a z każdą nową wersją te liczby stale rosną.

Ponadto Android posiada olbrzymią liczbę pakietów w przestrzeni nazw *java.\**. Wyróżnić można pakiety *awt.font*, *io*, *lang*, *lang.annotation*, *lang.ref*, *lang.reflect*, *math*, *net*, *nio*, *nio.channels*, *nio.channels.spi*, *nio.charset*, *security*, *security.acl*, *security.cert*, *security.interfaces*, *security.spec*, *sql*, *text*, *util*, *util.concurrent*, *util.concurrent.atomic*, *util.concurrent.locks*, *util.jar*, *util.logging*, *util.prefs*, *util.regex* oraz *util.zip*. Następne pakiety pochodzą z przestrzeni nazw *javax: crypto*, *crypto.spec*, *microedition.khronos.egl*, *microedition.khronos.opengles*, *net*, *net.ssl*, *security.auth*, *security.auth.callback*, *security.auth.login*, *security.auth.x500*, *security.cert*, *sql*, *xml* oraz *xmlparsers*. Jakby tego było mało, Android został wyposażony w wiele pakietów z takich przestrzeni nazw, jak *org.apache.http.\**, a także *org.json*, *org.w3c.dom*, *org.xml.sax*, *org.xml.sax.ext*, *org.xml.sax.helpers*, *org.xmlpull.v1* oraz *org.xmlpull.v1.sax2*. Razem te liczne pakiety tworzą rozbudowaną platformę obliczeniową, umożliwiającą pisanie aplikacji dla urządzeń typu handheld.

## Wykorzystanie zalet kodu źródłowego Androida<sup>3</sup>

We wczesnych edycjach Androida jego dokumentacja była miejscami nieco niezadowalająca. Kod źródłowy Androida mógł zostać wykorzystany do uzupełnienia braków.

Szczegóły dotyczące dystrybucji źródła Androida zostały opublikowane na stronie <http://source.android.com>. Kod źródłowy został ujawniony w październiku 2008 roku. Jednym z celów zre-szenia OHA było uczynienie z Androida darmowej oraz całkowicie konfigurowalnej platformy dla urządzeń przenośnych.

Jak zostało wyjaśnione, Android jest platformą, a nie pojedynczym projektem. Zakres oraz liczbę projektów można zobaczyć na stronie <http://source.android.com/projects>.

Kod źródłowy Androida oraz wszystkie projekty z nim związane są zarządzane przez system kontroli kodu źródłowego Git. Git (<http://git.or.cz/>) jest systemem o jawnym kodzie źródłowym, zaprojektowanym tak, żeby szybko i wygodnie móc zajmować się dużymi oraz małymi projektami. Pod kontrolą systemu Git znajduje się także jądro Linuksa oraz projekt Ruby on Rails. Pełną listę projektów związanych z Androidem, znajdujących się w repozytorium Git, można znaleźć na stronie <http://android.git.kernel.org/>.

Te projekty można pobierać za pomocą narzędzi oferowanych przez system Git, opisanych na stronie produktu. Niektóre z najważniejszych projektów dotyczą środowiska Dalvik, frameworks/base (plik android.jar), jądra Linuksa oraz wielu zewnętrznych bibliotek, takich jak biblioteki Apache HTTP (apache-http). Przechowywane są tu również podstawowe aplikacje Androida. Oto niektóre z nich: AlarmClock, Browser, Calculator, Calendar, Camera, Contacts, Email, GoogleSearch, HTML Viewer, IM, Launcher, Mms, Music, PackageInstaller, Phone, Settings, SoundRecorder, Stk, Sync, Updater oraz VoiceDialer.

Wśród projektów Androida znajdują się także projekty Provider. **Projekty Provider** są niczym bazy danych Androida, łączące dane z usługami RESTful. Wśród tych projektów można znaleźć takie, jak CalendarProvider, ContactsProvider, DownloadProvider, DrmProvider, Google-ContactsProvider, GoogleSubscribedFeedsProvider, ImProvider, MediaProvider, SettingsProvider, Subscribed FeedsProvider oraz TelephonyProvider.

Programiści będą najbardziej zainteresowani kodem źródłowym, znajdującym się w pliku android.jar (w przypadku pobrania całej platformy należy zatrzymać się na stronie <http://source.android.com/source/downloading.html>). Kod źródłowy tego pliku można pobrać z następującego adresu: <http://git.source.android.com/?p=platform/frameworks/base.git;a=snapshot;h=HEAD;sf=tgz>.

Z powyższego adresu można pobierać inne projekty Git. W systemie Windows do rozpakowania tego pliku służy aplikacja pkzip. Chociaż można pobrać i rozpakować pliki zawierające kod źródłowy, być może wygodniej będzie przejrzeć je w internecie, jeżeli nie ma potrzeby sprawdzania tego kodu w środowisku IDE. System Git pozwala przeglądać pliki online. Na przykład

---

<sup>3</sup> W trakcie tłumaczenia książki strona <http://android.git.kernel.org> została zaatakowana przez nieznanych sprawców i od tego czasu wszelkie dostępne na niej zasoby są niedostępne dla użytkowników. Prawdopodobnie strona ta zostanie ponownie oddana do użytku w niedalekiej przyszłości, do tego czasu osoby pragnące przejrzeć kod źródłowy Androida muszą skorzystać z alternatywnego źródła. Na stronie <http://source.android.com/source/downloading.html> znajdziemy instrukcję korzystania z narzędzia Repo, pozwalającego na pobieranie i przeglądanie wspomnianego kodu źródłowego — przyp. tłum.

pliki źródłowe *android.jar* można przejrzeć pod adresem <http://android.git.kernel.org/?p=platform/frameworks/base.git;a=summary>.

Jednak po odwiedzeniu tej strony należy wykonać jeszcze kilka czynności. Trzeba wybrać opcję *grep* z rozwijanego menu i wpisać nazwę w polu wyszukiwania. Następnie na liście wyników należy kliknąć nazwę pliku, żeby otworzyć jego kod źródłowy w przeglądarce. Można w ten sposób szybko zajrzieć do kodu źródłowego.

Czasami poszukiwanego pliku może nie być w katalogu *frameworks/base* lub określonym projekcie. W takim wypadku należy znaleźć listę projektów i przeszukiwać każdy z nich krok po kroku. Taka lista jest dostępna tutaj: <http://android.git.kernel.org/>.

Nie można przeszukiwać wszystkich projektów za pomocą polecenia *grep*, należy zatem się dowiedzieć, do jakich kategorii Androida należą poszczególne projekty. Na przykład biblioteki graficzne projektu Skia są dostępne tutaj: <http://android.git.kernel.org/?p=platform/external/skia.git;a=blob;f=src/core/SkMatrix.cpp>.

Plik *SkMatrix.cpp* zawiera kod źródłowy macierzy transformacyjnej, przydatnej w animacji: <http://android.git.kernel.org/?p=platform/external/skia.git;a=blob;f=src/core/SkMatrix.cpp>.

## Przykładowe projekty zawarte w książce

W niniejszej książce umieściliśmy bardzo dużo działających przykładowych projektów. Od rozdziału 2. aż do 28. wszystkie informacje dotyczą aplikacji tworzonych dla smartfonów, i pod tym właśnie kątem wszelkie zawarte w nich projekty były testowane na różnych wersjach Androida, skończywszy na wersji 2.3. Jakby nie patrzeć, na rynku istnieje nieprzyczwicie wiele rodzajów smartfonów obsługujących system Android.

Większość projektów, jeśli nie wszystkie, będzie działała w niezmienionej formie na tabletach obsługujących system Android 3.0, chociaż mogą one wyglądać niezgodnie z oczekiwaniemi. W trakcie tworzenia projektów naszym głównym celem było zaprezentowanie poszczególnych koncepcji oraz pakietów systemu Android, a w niektórych przypadkach również zademonstrowanie działania pewnych funkcji w starszych wersjach Androida. Łatwo wdrożyć te koncepcje podczas tworzenia aplikacji dla tabletów. W razie potrzeby nasze projekty bez problemu zintegrowałiby się z innymi funkcjami specyficznymi dla wersji 3.0 Androida, jednak dołączenie tych nowych funkcji w projektach odciągnęłoby naszą uwagę od koncepcji, które pragniemy objaśnić.

Rozdziały 29. – 31. zostały poświęcone Androidowi w wersji 3.0, więc zawarte w nich projekty zostały specjalnie zaprojektowane i przetestowane w tym systemie.

## Podsumowanie

W tym rozdziale chcieliśmy wzbudzić u Czytelnika zainteresowanie Androidem. Osoby programujące w środowisku Java mają znakomitą okazję, aby wyciągnąć korzyści z tej ekscytującej, rozbudowanej platformy obliczeniowej ogólnego przeznaczenia. Zapraszamy w podróż przez resztę książki — celem tej wędrówki jest dogłębne zrozumienie zestawu Android SDK.



# Konfigurowanie środowiska programowania

W poprzednim rozdziale omówiliśmy historię Androida oraz zarysowaliśmy konsepcje, które zostaną omówione w dalszej części książki. W tym momencie Czytelnik prawdopodobnie może zechcieć już zająć się kodem. Rozpoczniemy od przedstawienia elementów potrzebnych do tworzenia aplikacji w środowisku Android SDK oraz od przygotowania tego środowiska. Następnie szczegółowo przeanalizujemy aplikację „Witaj, świecie!” oraz rozłożymy na czynniki pierwsze nieco bardziej złożony fragment kodu. W dalszej kolejności objaśnimy cykl życia aplikacji w Androidzie, a na końcu poświęcimy chwilę tematowi wyszukiwania błędów w aplikacji za pomocą narzędzi AVD (ang. *Android Virtual Devices* — wirtualne urządzenia Androida).

Do tworzenia aplikacji przeznaczonych dla Androida wymagane jest posiadanie zestawu JDK (ang. *Java SE Development Kit* — zestaw do projektowania w środowisku Java SE), środowiska Android SDK oraz środowiska projektowego. Inaczej mówiąc, można pisać aplikacje za pomocą najprostszego edytora tekstowego, ale na potrzeby tworzenia projektów omówionych w tej książce lepsze będzie powszechnie dostępne środowisko IDE Eclipse. Android SDK wymaga zestawu JDK w wersji co najmniej 5 (korzystaliśmy z JDK 6) oraz środowiska Eclipse w wersji nie wcześniejszej niż 3.4 (używaliśmy wersji Eclipse 3.5, noszącej nazwę Galileo, oraz wersji 3.6, nazwanej Helios).

Żeby ułatwić sobie życie, można zainstalować narzędzia ADT (ang. *Android Development Tools* — narzędzia projektowe Androida). Jest to wtyczka środowiska Eclipse, umożliwiająca tworzenie aplikacji przeznaczonych dla Androida w środowisku IDE Eclipse. W istocie wszystkie przykłady w tej książce zostały zaprojektowane w środowisku Eclipse za pomocą narzędzi ADT.

Zestaw Android SDK składa się z dwóch głównych składników. Są to narzędzia i pakiety. Podczas jego pierwszej instalacji otrzymujemy do dyspozycji wyłącznie podstawowe narzędzia. Są to przeważnie pliki wykonywalne oraz pomocnicze, wspierające proces tworzenia aplikacji. Pakietami nazywamy pliki, które są unikatowe dla danej wersji Androida (nazywanej platformą), lub dodatki przeznaczone dla określonej platformy. Do platform zaliczamy Androida w wersjach od 1.5 do 3.0.

Na dodatki składają się takie narzędzia, jak interfejs API Google Maps, validator licencji przeznaczonych dla Android Market (ang. *Market License Validator*), a nawet dodatki pochodzące od producentów telefonów, jak na przykład wtyczka Galaxy Tab firmy Samsung. Po zainstalowaniu pakietu SDK będzie można następnie wykorzystać jedno z narzędzi do pobrania i skonfigurowania platform oraz dodatków. Zaczynajmy!

## Konfigurowanie środowiska

Żeby móc tworzyć aplikacje dla Androida, należy zapewnić sobie środowisko projektowe. W tym podrozdziale zajmiemy się omówieniem procesu pobierania aplikacji JDK 6, środowiska Eclipse, zestawu Android SDK (narzędzia i pakiety) oraz dodatku ADT. Pomożemy także skonfigurować środowisko Eclipse, tak aby można było w nim tworzyć aplikacje dla Androida.

Środowisko Android SDK jest kompatybilne z systemami Windows (Windows XP, Windows Vista oraz Windows 7), Mac OS X (jedynie z procesorami Intel) oraz Linux (również wyłącznie z procesorami Intel). W tym rozdziale omówimy proces konfigurowania środowiska we wszystkich wymienionych rodzajach systemów (w przypadku Linuksa jedynie dla wariantu Ubuntu). W kolejnych rozdziałach nie będziemy się zajmować różnicami pomiędzy poszczególnymi systemami operacyjnymi.

### Pobieranie zestawu JDK 6

Pierwszym potrzebnym składnikiem jest zestaw Java Development Kit. Środowisko Android SDK wymaga co najmniej wersji 5 zestawu JDK; przykłady w książce były tworzone z wykorzystaniem wersji 6. Dla systemów Windows aplikacja JDK 6 jest dostępna na oficjalnej stronie firmy Sun ([www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html)) — należy ją pobrać i zainstalować. Wystarczy edycja standardowa aplikacji JDK, jej wersje Bundle nie są wymagane. Zestaw JDK dla systemu Mac OS X można znaleźć w witrynie Apple (<http://developer.apple.com/java/download/>); należy stamtąd wybrać plik dla odpowiedniej wersji systemu i zainstalować go. Aby uzyskać dostęp do zestawu JDK, należy bezpłatnie zarejestrować się jako programista, a na stronie pobrań kliknąć odnośnik do Javy znajdujący się po prawej stronie okna. Żeby zainstalować JDK w systemie Linux, należy otworzyć okno terminalu i wpisać następujące polecenie:

```
sudo apt-get install sun-java6-jdk
```

Polecenie to spowoduje zainstalowanie aplikacji JDK oraz wszystkich wymaganych dodatkowych składników, takich jak środowisko JRE (ang. *Java Runtime Environment* — środowisko uruchomieniowe Java). Jeżeli tak się nie stanie, oznacza to prawdopodobnie, że należy dodać nowe źródło oprogramowania (ang. *software source*) i spróbować wykonać powyższe polecenie ponownie. Na stronie <https://help.ubuntu.com/community/Repositories/Ubuntu> wyjaśniono zasadę działania źródeł oprogramowania oraz sposób dodawania połączenia do oprogramowania pochodzącego z niezależnego źródła. Proces ten jest odmienny dla różnych wersji Linuksa. Po jego przeprowadzeniu należy spróbować ponownie wykonać widoczne powyżej polecenie.

Wraz z wprowadzeniem wersji Ubuntu 10.04 (Lucid Lynx) zalecane jest korzystanie raczej z zestawu OpenJDK, a nie Oracle/Sun JDK. Aby go zainstalować, stosujemy następujące polecenie:

```
sudo apt-get install openjdk-6-jdk
```

Jeżeli aplikacja nie zostanie znaleziona, należy — tak jak zostało wcześniej wspomniane — skonfigurować oprogramowanie wydane przez niezależnego wydawcę i ponownie uruchomić polecenie. Spowoduje to automatyczne dodanie wszystkich pakietów wymaganych przez zestaw JDK. Istnieje możliwość jednokrotnego posiadania zestawów OpenJDK oraz Oracle/Sun JDK. W celu przełączania pomiędzy aktywnymi wersjami środowiska Java zainstalowanymi w systemie Ubuntu uruchamiamy poniższe polecenie w interpreterze powłoki:

```
sudo update-alternatives --config java
```

a następnie wybieramy wersję środowiska Java, która ma pozostać domyślną.

Po zainstalowaniu środowiska JDK należy skonfigurować zmienną środowiskową `JAVA_HOME`, tak żeby wskazywała folder instalacyjny JDK. Na komputerze z zainstalowanym systemem Windows XP można tego dokonać, otwierając menu *Start* i klikając prawym przyciskiem myszy ikonę *Mój komputer*. Z menu należy wybrać opcję *Właściwości*, a następnie przejść do zakładki *Zaawansowane* i kliknąć przycisk *Zmienne środowiskowe*. W dalszej kolejności trzeba kliknąć przycisk *Nowa*, żeby dodać zmienną, lub *Edycja*, by poprawić istniejącą zmienną. Wartość zmiennej `JAVA_HOME` będzie wyglądała mniej więcej następująco: `C:\Program Files\Java\jdk1.6.0_23`. W systemach Windows Vista oraz Windows 7 uzyskiwanie dostępu do okna *Zmienne środowiskowe* wygląda nieco inaczej. Trzeba wybrać menu *Start*, prawym przyciskiem myszy kliknąć ikonę *Komputer* i wybrać z menu opcję *Właściwości*, wybrać łącze *Zaawansowane ustawienia systemu*, a następnie użyć przycisku *Zmienne środowiskowe*.... Kolejne czynności wymagane do ustanowienia lub zmiany zmiennej środowiskowej `JAVA_HOME` są identyczne jak w systemie Windows XP. W systemie Mac OS X zmienną środowiskową `JAVA_HOME` konfiguruje się w pliku `.profile`, umieszczonem w katalogu `HOME`. Należy utworzyć lub edytować ten plik i dodać następujący wiersz:

```
export JAVA_HOME=ścieżka_do_katalogu_JDK
```

gdzie w miejscu `ścieżka_do_katalogu_JDK` będzie prawdopodobnie `/Library/Java/Home`. W systemie Linux należy edytować plik `.profile` oraz dodać taki sam wiersz jak w przypadku systemu Mac OS X, z tym że docelową ścieżką, którą należy dodać, będzie najprawdopodobniej `/usr/lib/jvm/java-6-sun` lub `/usr/lib/jvm/java-6-openjdk`. Niektórzy wolą stosować plik `.bashrc` zamiast pliku `.profile`; w obydwu przypadkach powyższe polecenie powinno działać.

## Pobieranie środowiska Eclipse 3.6

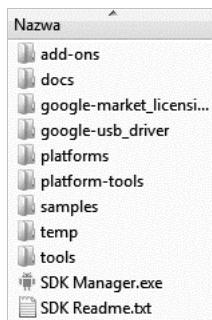
Po zainstalowaniu pakietu JDK można pobrać środowisko Eclipse IDE for Java Developers (edycja dla Java EE nie jest wymagana; będzie działać, lecz zajmuje o wiele więcej miejsca i zawiera elementy, których nie będziemy potrzebować). Przykłady przygotowane z myślą o tej książce zostały napisane z wykorzystaniem środowiska Eclipse 3.6 (w systemie Windows). Wszystkie wersje Eclipse są dostępne pod adresem [www.eclipse.org/downloads/](http://www.eclipse.org/downloads/). Pliki środowiska są skompresowane w formacie `.zip`, można je wypakować w dowolnym miejscu. Najprościej jest wypakować je do partycji `C:\`, co spowoduje utworzenie katalogu `C:\Eclipse`. Można w nim znaleźć plik wykonywalny `eclipse.exe`. W przypadku systemu Mac OS X pliki można rozpakować do katalogu `Applications`, zaś w Linuksie — do katalogu `HOME` lub poprosić administratora o ich umieszczenie w publicznym miejscu, do którego istnieje łatwy dostęp. We wszystkich przypadkach plik wykonywalny środowiska Eclipse zostaje umieszczony w utworzonym folderze. Można również znaleźć i zainstalować środowisko Eclipse poprzez Centrum Oprogramowania Linuksa, w którym dodawane są nowe aplikacje, chociaż być może nie uda się tam znaleźć jego najnowszej wersji.

Podczas pierwszego uruchomienia środowiska Eclipse pojawi się monit o określenie ścieżki do przestrzeni roboczej. Żeby nie komplikować sobie zbytnio życia, warto wpisać jak najprostszy adres, jak na przykład C:\android, lub umieścić tę ścieżkę w podkatalogu swojego profilowego katalogu. Jeżeli komputer jest współużytkowany przez kilka osób, dobrze jest umieścić folder przestrzeni roboczej gdzieś w katalogu swojego profilu.

## Pobieranie zestawu Android SDK

Zasadniczym składnikiem, niezbędnym w trakcie tworzenia aplikacji dla Androida, jest środowisko programistyczne Android SDK. Jak już zostało wcześniej wspomniane, zestaw SDK posiada pewne podstawowe narzędzia, do których następnie dołączamy kolejne składowe złożone z potrzebnych czy przydatnych pakietów. Wśród narzędzi jest dostępny emulator, zatem nie trzeba posiadać urządzeń przenośnego z systemem Android do projektowania aplikacji. Wśród nich można również znaleźć program instalacyjny, pozwalający na wybranie pakietów, które mają zostać pobrane.

Zestaw Android SDK jest dostępny pod adresem <http://developer.android.com/sdk>. Podobnie jak w przypadku środowiska Eclipse, jest on skompresowany w formie pliku .zip, zatem należy go wypakować do wybranej lokacji. W systemach Windows można umieścić te pliki na przykład w partycji C:, a po rozpakowaniu powinien pojawić się folder android-sdk-windows (lub podobnie nazwany), w którym będą pliki przedstawione na rysunku 2.1. W przypadku systemów Mac OS X oraz Linux zestaw Android SDK można wypakować do katalogu HOME. Łatwo zauważać, że wersja przeznaczona dla systemów Mac OS X i Linux nie posiada pliku wykonywalnego *SDK Manager*. W przypadku wspomnianych systemów zamiast tego pliku uruchamiamy program znajdujący się w katalogu *tools/android*.

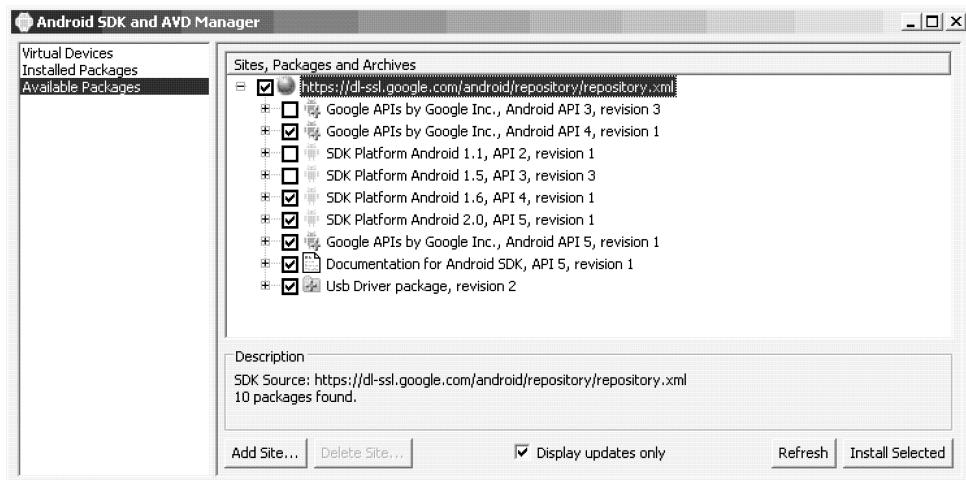


Rysunek 2.1. Zawartość folderu Android SDK

Alternatywnym rozwiązaniem (wyłącznie w przypadku systemu Windows) jest pobranie instalatora w formacie .exe, a nie skompresowanego do formatu .zip. Instalator sprawdzi obecność zestawu Java JDK, wypakuje wymagane pliki oraz uruchomi aplikację *SDK Manager*, co ułatwi pobranie pozostałych plików.

Bez względu na to, czy korzystamy z instalatora, czy bezpośrednio uruchamiamy aplikację *SDK Manager*, następnym etapem jest zainstalowanie niektórych pakietów. W trakcie instalacji zestawu Android SDK nie zawiera on żadnej platformy (na przykład różnych wersji systemu Android). Instalowanie platform jest bardzo proste. Po uruchomieniu aplikacji *SDK Manager* należy wybrać *Window/Android SDK and AVD Manager*, kliknąć element *Available Packages*, zaznaczyć adres źródła <https://dl-ssl.google.com/android/repository/repository.xml>, a następnie

wybrać potrzebne platformy i dodatki, na przykład *Android 2.3.3* (rysunek 2.2). Aby środowisko działało, należy koniecznie dodać narzędzia powiązane z daną platformą. Ponieważ już niebawem będziemy z niej korzystać, warto teraz dodać platformę przynajmniej w wersji 1.6.



Rysunek 2.2. Dodawanie pakietów do środowiska Android SDK

Teraz wystarczy kliknąć przycisk *Install Selected*. Trzeba zatwierdzić każdy element, zaznaczając opcję *Accept*<sup>1</sup>, a następnie zatwierdzić przyciskiem *Install Accepted*. Android pobierze wybrane pakiety i platformy. Dodatki *Google APIs* służą do projektowania aplikacji wykorzystujących Google Maps. Istnieje możliwość przeglądania zainstalowanych dodatków po kliknięciu opcji *Installed Packages*, widocznej na rysunku 2.2 w lewym górnym rogu okna. Jeśli będzie trzeba, w każdej chwili można tu wrócić i zainstalować następne pakiety.

## Aktualizowanie zmiennej środowiskowej PATH

Środowisko Android SDK zawiera katalog narzędzi, który warto umieścić w zmiennej systemowej PATH. Trzeba będzie również dodać do niej katalog z narzędziami obsługującymi platformy. Instalację tego katalogu omówiliśmy przed chwilą. Dodajmy teraz te narzędzia lub upewnijmy się, że są właściwie umieszczone. Przy okazji ułatwimy sobie pracę, dodając także katalog *bin* zestawu JDK. W systemie Windows należy wrócić do okna zmiennych środowiskowych. Następnie trzeba edytować zmienną PATH, dodać na końcu średnik (;), wpisać ścieżkę do folderu *tools* Androida SDK, po kolejnym średniku wpisać ścieżkę do folderu zawierającego narzędzia obsługujące platformy, a po następnym średniku umieścić wpis %JAVA\_HOME%\bin. Następnie wystarczy kliknąć przycisk OK. W przypadku systemów Mac OS X oraz Linux należy edytować plik *.profile* i dodać ścieżkę do folderu *tools*, a także ścieżkę do narzędzi powiązanych z platformami oraz parametr \$JAVA\_HOME/bin do zmiennej PATH. W systemie Linux powinno to wyglądać mniej więcej tak:

```
export PATH=$PATH:$HOME/android-sdk-linux_x86/tools:$HOME/android-sdklinux_x86/
➥platform-tools:$JAVA_HOME/bin
```

Należy się jeszcze upewnić, że część polecenia, która dotyczy ścieżki do katalogu *tools*, będzie wskazywała miejsce zainstalowania katalogu.

<sup>1</sup> Lub zaakceptować wszystkie jednocześnie, zaznaczając *Accept All* — przyp. tłum.

## Okno narzędzi

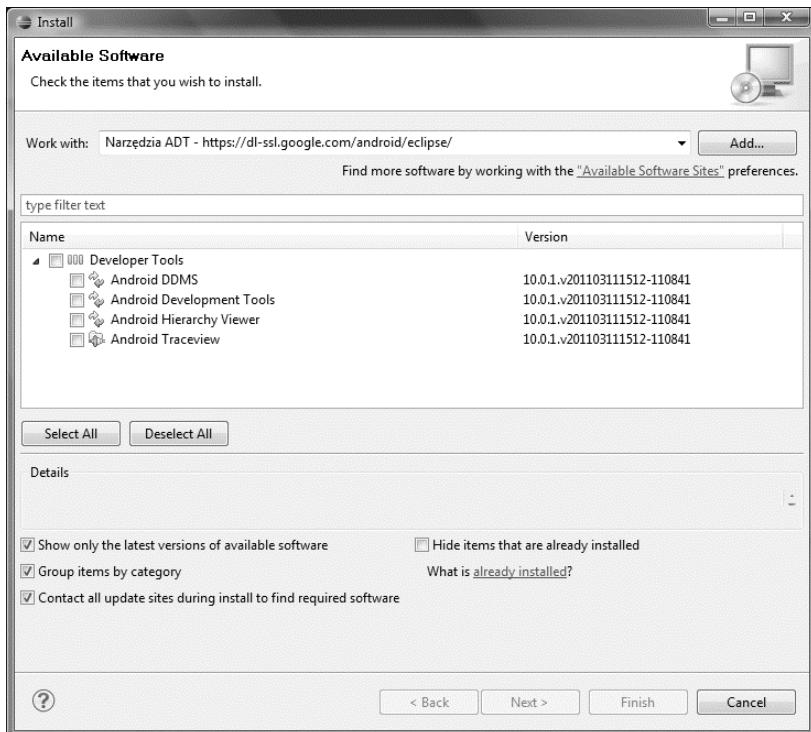
W dalszej części książki pojawią się momenty, gdy trzeba będzie uruchamiać pewne programy z wiersza poleceń. Są one częścią środowiska JDK lub Android SDK. Dzięki umieszczeniu ich w zmiennej systemowej PATH nie będzie trzeba wpisywać pełnej ścieżki do nich, jednak do uruchomienia tych programów konieczne jest otwarcie okna narzędzi. W następnych rozdziałach będziemy korzystać z takiego okna. Najprostszym sposobem jego uruchomienia w systemie Windows jest kliknięcie menu *Start/Wyszukaj*, a następnie wpisanie cmd w polu tekstowym i kliknięcie przycisku *OK*. W systemie Mac OS X należy wybrać aplikację *Terminal* w folderze *Applications* z poziomu menedżera plików *Finder* lub z poziomu *Dock*. W systemie Linux aplikacja *Terminal* znajduje się w menu *Applications/Accessories*.

Trzeba wspomnieć o jeszcze jednej sprawie dotyczącej różnic pomiędzy systemami operacyjnymi: niekiedy trzeba znać adres IP stacji roboczej. W systemie Windows należy uruchomić wiersz poleceń i wpisać polecenie ipconfig. Wśród wyników będzie widniał wpis dotyczący IPv4 (lub podobny), a obok zostanie wyświetlony adres IP danego komputera. Wygląda on mniej więcej tak: 192.168.1.25. W systemach Mac OS X oraz Linux należy uruchomić wiersz poleceń i wpisać ifconfig. Adres IP jest umieszczony obok wpisu inet addr. Może też być widoczne połączenie sieciowe przy nazwie *localhost* lub *lo*. Adres IP tego połączenia to 127.0.0.1. Jest to specjalny typ połączenia sieciowego, wykorzystywany przez system operacyjny, i nie ma nic wspólnego z adresem IP stacji roboczej. Należy poszukać wiersza, w którym widoczny jest inny adres IP.

## Instalowanie narzędzi ADT

Teraz należy zainstalować narzędzia ADT — wtyczkę środowiska Eclipse usprawniającą tworzenie aplikacji dla Androida. Dokładniej mówiąc, narzędzia ADT łączą się ze środowiskiem Eclipse, dzięki czemu można tworzyć i testować aplikacje przeznaczone dla systemu Android oraz wyszukiwać w nich błędy. Żeby zainstalować tę wtyczkę, należy skorzystać z funkcji *Install New Software...*, dostępnej w aplikacji Eclipse (instrukcje dotyczące aktualizacji wtyczek można znaleźć w dalszej części podróżnika). Żeby zainstalować narzędzia ADT, należy uruchomić środowisko Eclipse i wykonać następujące czynności:

1. W pasku narzędzi wybierz opcję *Help*, a następnie kliknij opcję *Install New Software...* (w poprzednich wersjach aplikacji Eclipse była ona nazwana *Software Updates*).
2. Zaznacz pole tekstowe *Work with...*, wpisz <https://dl-ssl.google.com/android/eclipse/> i naciśnij klawisz *Enter/Return*. Aplikacja połączy się z witryną i wyświetli listę pokazaną na rysunku 2.3.
3. Powinien się pojawić węzeł *Developer Tools*, podzielony na trzy podzadane kategorie: *Android DDMS*, *Android Development Tools* oraz *Android Hierarchy Viewer*. Zaznacz węzeł nadzędny oraz upewnij się, że elementy podzadane są również zaznaczone, a następnie kliknij przycisk *Next*. Prawdopodobnie numery wersji będą wyższe niż przedstawione na rysunku, ale to nie szkodzi. Mogą się tutaj również pojawić dodatkowe narzędzia.
4. Ukaże się okno potwierdzenia instalacji wtyczek. Kliknij *Next*. Odnosi się to również do aplikacji *Android Traceview*, dodanej w wersji Android 3.0.
5. W kolejnym oknie trzeba będzie zapoznać się z licencją narzędzi ADT, a także z licencjami dotyczącymi narzędzi potrzebnych do zainstalowania wtyczki. Przejrzyj licencje, zaznacz opcję *I accept the terms of license agreements* i kliknij przycisk *Finish*.



Rysunek 2.3. Instalacja narzędzi ADT za pomocą funkcji Install New Software w środowisku Eclipse

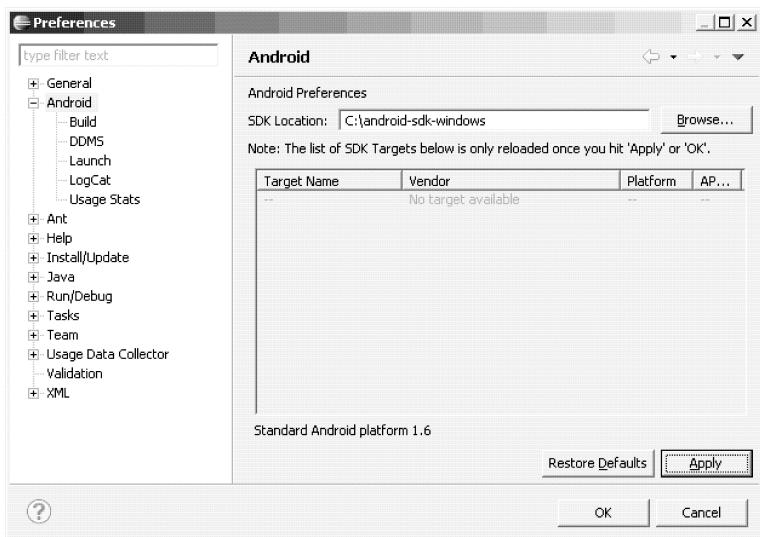
W tym momencie aplikacja Eclipse zacznie pobierać i instalować narzędzia programistyczne. Żeby nowe wtyczki pojawiły się w oknie Eclipse, należy ponownie uruchomić aplikację.

W przypadku posiadania starszej wersji narzędzi ADT należy otworzyć menu *Help* i wybrać opcję *Check for Updates*. Powinna zostać wyświetlona aktualna wersja wtyczek ADT, których instalacja przebiega tak, jak opisano, począwszy od punktu 3. powyższych instrukcji.

**Uwaga!**

Aplikacja Android Hierarchy Viewer została dodana do narzędzi Developer Tools wraz z wersją 2.3 Androida. Będzie więc ona dostępna w trakcie przeprowadzania nowej instalacji. Jeśli jednak aktualizujemy narzędzia ADT, być może nie będzie jej widać na liście. Jeśli nie jest widoczna, po zaktualizowaniu pozostałych elementów należy przejść do zakładki *Install New Software...* i wybrać adres <https://dl-ssl.google.com/android/eclipse/> z menu *Works With*. W środkowym oknie powinien się pojawić węzeł *Android Hierarchy Viewer*, który można teraz osobno zainstalować.

Ostatnim etapem aktywacji narzędzi ADT w obrębie środowiska Eclipse jest odniesienie ich do zestawu Android SDK. W tym celu należy w środowisku Eclipse otworzyć menu *Window* i wybrać opcję *Preferences* (w systemie Mac OS X opcja ta jest dostępna w menu *Eclipse*). W oknie dialogowym *Preferences* należy wybrać węzeł *Android* i wpisać ścieżkę katalogu Android SDK (rysunek 2.4), a następnie kliknąć przycisk *Apply*. W międzyczasie może się pojawić okno dialogowe, w którym można zaznaczyć opcję wysyłania do firmy Google statystyk dotyczących wykorzystania programu Android SDK. Wybór należy do Czytelnika. Teraz wystarczy kliknąć *OK*, żeby zamknąć okno *Preferences*.



**Rysunek 2.4.** Powiązanie narzędzi ADT z zestawem Android SDK

SDK Manager można uruchomić z poziomu środowiska Eclipse. Należy w tym celu wybrać zakładkę *Window/Android SDK and AVD Manager*. Powinno zostać wyświetlone okno przedstawione na rysunku 2.2, chociaż prawdopodobnie nie będą widoczne wszystkie opcje dostępne podczas osobnego uruchamiania aplikacji SDK Manager.

Już niemal nadszedł czas na zapoznanie się z pierwszą aplikacją dla Androida — najpierw jednak musimy zapoznać się z podstawowymi pojęciami odnoszącymi się do aplikacji tworzonych dla tej platformy.

## Przedstawienie podstawowych składników

Szkielet każdej aplikacji zawiera pewne kluczowe składniki, z którymi muszą się zapoznać projektanci, zanim zaczną pisać programy oparte na tym szkielecie. Na przykład do napisania aplikacji w środowisku J2EE (Java 2 Platform Enterprise Edition) wymagana jest znajomość technologii JSP (JavaServer Pages) oraz serwletów. Podobnie w przypadku aplikacji pisanych dla Androida — należy znać pojęcia aktywności, widoków, intencji, dostawców treści, usług oraz przeznaczenie pliku *AndroidManifest.xml*. W tym podrozdziale omówimy krótko każde z tych pojęć, a bardziej szczegółowe informacje zostaną przedstawione w kolejnych rozdziałach.

### Widok

Widoki są elementami interfejsu użytkownika tworzącymi jego podstawowe bloki budulcowe. Mogą one przybrać kształt przycisku, etykiety, pola tekstowego oraz wielu innych składników interfejsu UI. Jeżeli Czytelnik wie, czym są widoki w platformach J2EE oraz Swing, szybko zrozumie widoki w Androidzie. Widoki często są wykorzystywane jako kontenery dla innych widoków, co zazwyczaj oznacza istnienie hierarchii widoków w interfejsie użytkownika. Ostatecznie wszystkie elementy widoczne na ekranie są widokami.

## Aktywność

Aktywność jest pojęciem interfejsu użytkownika. Aktywność przeważnie jest reprezentacją pojedynczego okna aplikacji. Zazwyczaj zawarty jest w niej przynajmniej jeden widok, ale niekiedy musi tak być. Określenie „aktywność” dość dokładnie wskazuje jej przeznaczenie — jest to obiekt pomagający użytkownikowi wykonać daną czynność. Taką czynnością może być przeglądanie, tworzenie lub edycja danych. Większość aplikacji tworzonych dla systemu Android zawiera kilka aktywności.

## Intencja

Uogólniając, słowo intencja oznacza intencję, zamiar wykonania jakiejś pracy. W terminie tym mieści się kilka pojęć, więc najlepszym sposobem jego zrozumienia jest wykorzystanie intencji w praktyce. Intencje są wykorzystywane w następujących celach:

- nadawanie komunikatu,
- uruchamianie usługi,
- rozpoczęwanie aktywności,
- wyświetlanie strony WWW lub listy kontaktów,
- wybieranie lub odbieranie połączenia telefonicznego.

Intencje nie zawsze są inicjowane przez aplikację — są także wykorzystywane przez system do powiadamiania aplikacji o określonych zdarzeniach (na przykład o otrzymaniu wiadomości tekstuowej).

Intencje można podzielić na jawnie oraz niejawne. Jeżeli zostanie wyraźnie określone, że adres URL ma być widoczny, system automatycznie zdecyduje, jaki składnik będzie dotyczył intencji. Istnieje także możliwość określenia konkretnej informacji, w jaki sposób powinna być potraktowana intencja. Intencje luźno łączą działanie z jego uchwytem.

## Dostawca treści

Współdzielenie danych pomiędzy aplikacjami urządzenia przenośnego jest powszechnie stosowaną praktyką. Android definiuje więc standardowy mechanizm współużytkowania danych (takich jak listy kontaktów) przez aplikacje bez konieczności odsłaniania podstawowych magazynów, struktury oraz implementacji. Dzięki dostawcom treści można ujawniać dane oraz pozwalać jednym aplikacjom korzystać z zasobów innych programów.

## Usługa

Usługi Androida są podobne do usług obecnych w systemie Windows lub na innych platformach — są to procesy działające w tle, które potencjalnie mogą trwać przez długi czas. W Androidzie są zdefiniowane dwa rodzaje usług: usługi lokalne oraz usługi zdalne. Usługi lokalne są elementami dostępnymi wyłącznie dla aplikacji je obsługującej. Z drugiej strony usługi zdalne są przeznaczone dla innych aplikacji, łączących się z nimi w sposób zdalny.

Przykładem usługi jest składnik wykorzystywany przez aplikację pocztową do sprawdzania, czy pojawiły się nowe wiadomości. Usługa ta jest lokalna, jeżeli nie jest używana przez inne aplikacje znajdujące się w urządzeniu. Jeżeli korzysta z niej kilka usług, można ją zaimplementować w formie usługi zdalnej. Jak zostało wyjaśnione w rozdziale 11., jest to związane z różnicą pomiędzy funkcjami `startService()` oraz `bindService()`.

Istnieje możliwość stosowania istniejących usług, jak również pisania własnych za pomocą rozszerzania klasy `Service`.

## AndroidManifest.xml

Plik `AndroidManifest.xml`, podobny do pliku `web.xml` w świecie J2EE, określa zawartość oraz zachowanie aplikacji. Na przykład znajduje się w nim lista aktywności oraz usług danej aplikacji, a także uprawnień i właściwości wymaganych do jej uruchomienia.

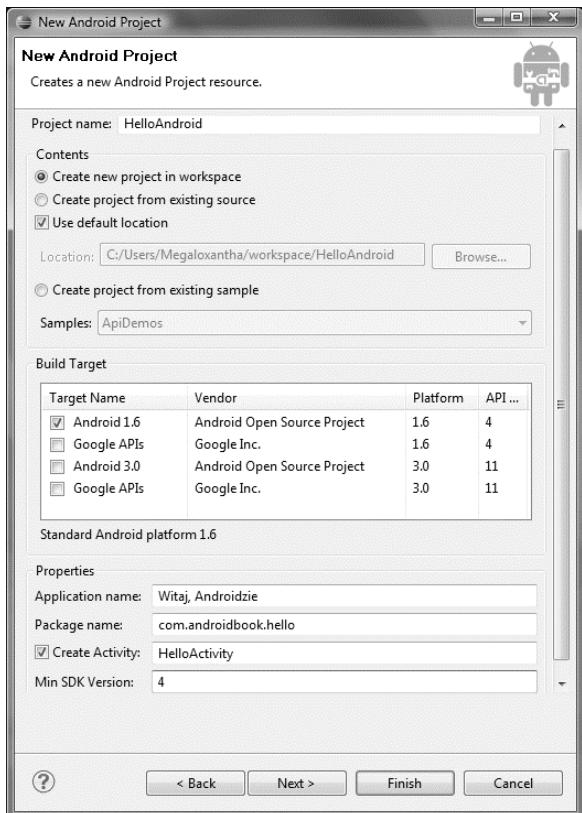
## Urządzenia AVD

Urządzenie AVD (ang. *Android Virtual Device* — wirtualne urządzenie Androida) pozwala programistom na przetestowanie aplikacji bez konieczności połączenia się z rzeczywistym urządzeniem (zazwyczaj telefonem lub tabletem). Można tworzyć różne konfiguracje urządzeń AVD, zdolne do emulowania różnych modeli istniejących urządzeń.

## Witaj, świecie!

Teraz możemy rozpocząć pisanie pierwszej aplikacji dla Androida. Na początek utworzymy prosty program „Witaj, świecie!”. Szkielet aplikacji zbudujemy w następujący sposób:

1. Uruchom środowisko *Eclipse* i wybierz *File/New/Project....* W oknie dialogowym *New Project* otwórz węzeł *Android*, a następnie wybierz opcję *Android Project*, po czym kliknij przycisk *Next*. Ujrzyś okno *New Android Project*, zaprezentowane na rysunku 2.5 (być może dostęp do projektu Android istnieje w menu *New*, dzięki czemu można nieco szybciej otwierać nowe projekty). Jeżeli istnieje taka możliwość, możesz również skorzystać z przycisku *New Android Project* na pasku narzędzi.
2. Wpisz, zgodnie z rysunkiem 2.5, nazwę projektu *HelloAndroid*. Musimy w jakiś sposób odróżniać nazwę tego projektu od innych projektów tworzonych w środowisku Eclipse, należy więc wybierać takie nazwy, które na pierwszy rzut oka na listę projektów będą łatwe do rozpoznania. Warto również zauważać, że domyślne umiejscowienie projektu jest związane z lokalizacją przestrzeni roboczej środowiska Eclipse. Kreator nowego projektu doda nazwę nowej aplikacji do obszaru roboczego. W naszym przypadku, jeśli przestrzeń robocza jest *C:\android*, nowy projekt zostanie umieszczony w katalogu *C:\android\HelloAndroid*.
3. Na razie zostaw sekcję *Contents* bez zmian, ponieważ w przestrzeni roboczej chcemy utworzyć nowy projekt umieszczony w domyślnej lokacji.
4. Zaznacz *Android 1.6* w oknie *Build Target*, tak jak zostało to pokazane na rysunku 2.5. Ta wersja Androida będzie służyła za bazę naszej aplikacji. Program ten będzie można uruchomić na późniejszych wersjach platformy, na przykład 2.1 albo 2.3, ale wersja 1.6 posiada wszystkie wymagane funkcje, zatem zostanie ona naszą wersją docelową. Zasadniczo najlepiej jest wybierać najniższą dopuszczalną wersję, ponieważ w ten sposób maksymalizujemy liczbę urządzeń, na których tworzona aplikacja zadziała zgodnie z oczekiwaniami.
5. Wprowadź *Witaj, Androidzie* jako nazwę aplikacji. Nazwa ta będzie pojawiała się wraz z ikoną aplikacji, w pasku tytułowym oraz na listach aplikacji. Powinna być opisowa, ale nie za długa.



Rysunek 2.5. Okno kreatora New Android Project

6. Jako nazwę pakietu wykorzystaj `com.androidbook.hello`. Aplikacja musi mieć nazwę podstawowego pakietu, a w naszym przykładzie wygląda ona właśnie tak. Nazwa ta będzie służyła jako identyfikator aplikacji i nie może się ona powtarzać wśród innych programów. Z tego powodu najlepiej rozpocząć nazwę pakietu od nazwy swojej domeny. Jeżeli nie posiadasz domeny, postaraj się być jak najbardziej kreatywny, aby jak najsukuteczniej unikać możliwości zduplikowania nazwy pakietu przez innych twórców. Nie należy jednak korzystać z nazw pakietu rozpoczynających się od członów `com.google`, `com.android`, `android` lub `com.example`, gdyż zostały one zastrzeżone przez firmę Google i nie będzie można umieścić takich aplikacji w sklepie Android Market.
7. W polu *Create Activity* wprowadź nazwę aktywności `HelloActivity`. Android zostaje w ten sposób poinformowany, że należy wywołać tę aktywność w momencie uruchomienia aplikacji. Aplikacja może zawierać inne aktywności, jednak to ta będzie pierwsza, jaką ujrzy użytkownik po uruchomieniu aplikacji.
8. Na koniec, dzięki wartości 4 w polu *Min SDK Version* Android „wie”, że aplikacja wymaga co najmniej wersji 1.6 systemu operacyjnego. Z technicznego punktu widzenia można określić minimalną wersję niższą od wartości *Build Target*. Jeżeli aplikacja wymaga funkcji niedostępnych w starszych wersjach Androida, trzeba rozwiązać ten problem w delikatny sposób, ale jest to możliwe. W przypadku większości aplikacji wartość pola *Min SDK Version* będzie zgodna z wartością *Build Target*.

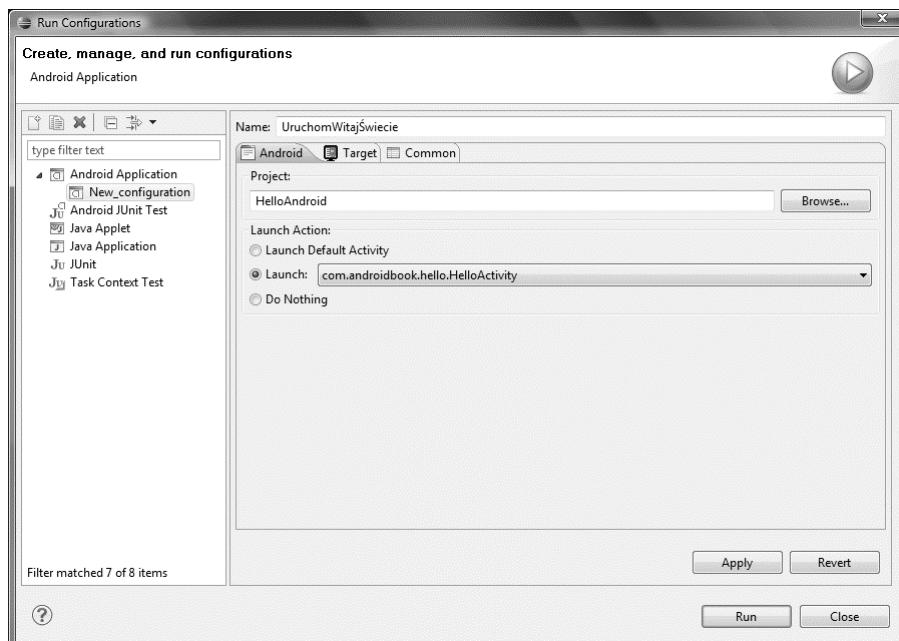
9. Kliknij przycisk *Finish*, dzięki czemu narzędzia ADT wygenerują szkielet projektu. Teraz otwórz plik *HelloActivity.java* w folderze *src* i zmodyfikuj metodę *onCreate()* w następujący sposób:

```
/** Called when activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    /* Tworzy deklarację widoku TextView i wyświetla napis „Witaj, świecie!” */
    TextView tv = new TextView(this);
    tv.setText("Witaj, świecie!");
    /* Przyłącza widok treści do deklaracji widoku TextView */
    setContentView(tv);
}
```

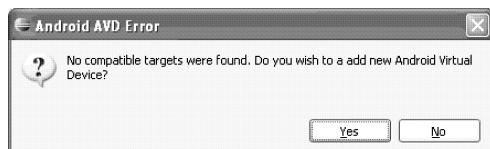
Prawdopodobnie trzeba będzie własnoręcznie dodać w kodzie instrukcję `import android.widget.TextView;`, aby pozbyć się komunikatu o błędzie, wyświetlonego przez środowisko Eclipse. Teraz wystarczy zapisać plik *HelloActivity.java*.

Żeby uruchomić aplikację, należy utworzyć konfigurację uruchomieniową środowiska Eclipse; będzie też potrzebne wirtualne urządzenie do przetestowania aplikacji. Szybko opiszemy wymagane czynności, a następnie zajmiemy się bardziej szczegółowo urządzeniami AVD. Konfigurację uruchomieniową tworzy się w następujący sposób:

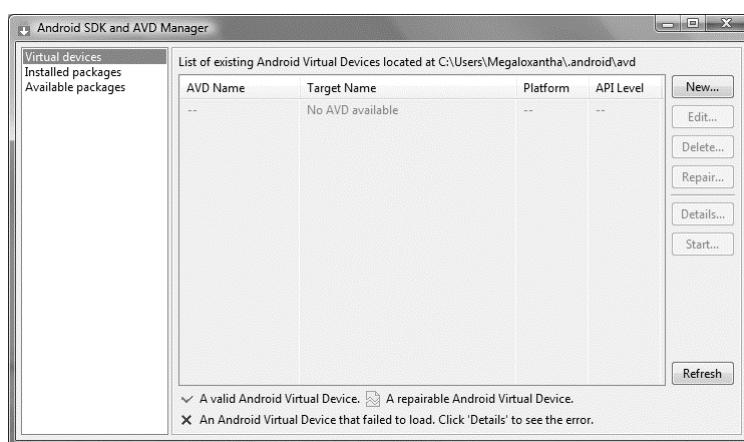
1. Wybierz główne menu *Run*, a następnie podelement *Run Configurations....*
2. W oknie dialogowym *Run Configurations* kliknij dwukrotnie opcję *Android Application*, znajdująca się w panelu po lewej stronie. Kreator utworzy nową konfigurację nazwaną *New Configuration*.
3. Zmień nazwę tej konfiguracji na *UruchomWitajŚwiecie*
4. Kliknij przycisk *Browse...* i zaznacz projekt *HelloAndroid*.
5. W części okna nazwanej *Launch Action* zaznacz opcję *Launch* i z rozwijanej listy wybierz *com.androidbook>HelloActivity*. Okno powinno wyglądać podobnie jak na rysunku 2.6.
6. Kliknij *Apply*, a następnie *Run*. Już niemal gotowe! Środowisko Eclipse jest przygotowane do uruchomienia aplikacji, ale potrzebuje jeszcze urządzenia, na którym zostanie ona sprawdzona. Pojawi się okno z ostrzeżeniem, takie jak na rysunku 2.7, że nie zostały znalezione kompatybilne urządzenia. Kliknij *Yes*, aby stworzyć własne urządzenie.
7. Następnie zostanie wyświetlone okno zawierające listę dostępnych urządzeń AVD (rysunek 2.8). Zwróc uwagę, że mamy do czynienia z tym samym oknem co przedstawione na rysunku 2.2. Poprzednio zostało wyświetlone podczas instalowania pakietów, tym razem jednak mamy do czynienia z wirtualnymi urządzeniami. Musisz tu dodać urządzenie pasujące do aplikacji. Kliknij przycisk *New....*
8. Wypełnij pola w oknie *Create new Android Virtual Device AVD*, tak jak zostało pokazane na rysunku 2.9. Podaj nazwę urządzenia *Gingerbread*, jako docelowy system z listy *Target* wybierz platformę *Android 2.3 — API Level 9* (lub jakąś inną wersję), ustaw rozmiar pamięci karty na 10 MB, włącz możliwość wykonywania migawek (opcja *Enabled* w polu *Snapshot*) oraz wybierz skórkę (ang. *Skin*) urządzenia w formacie *HVGA*. Kliknij *Create AVD*. Menedżer może powiadomić Cię o udanym utworzeniu urządzenia AVD. Kliknij krzyżyk w prawym górnym rogu okna, aby zamknąć okno narzędzia *Android SDK and AVD Manager*.



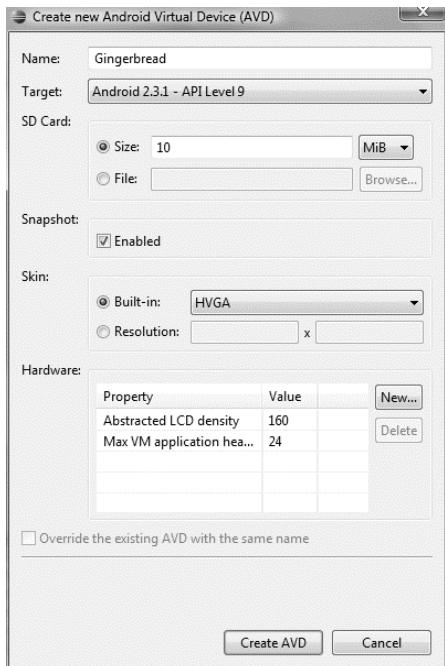
Rysunek 2.6. Tworzenie konfiguracji uruchomieniowej środowiska Eclipse, pozwalającej na uruchomienie aplikacji „Witaj, świecie!”



Rysunek 2.7. Informacja o błędzie: ostrzeżenie o braku kompatybilnych urządzeń i zapytanie o utworzenie nowego



Rysunek 2.8. Okno zawierające listę istniejących urządzeń AVD



**Rysunek 2.9.** Konfigurowanie wirtualnego urządzenia AVD

**Uwaga!**

Wybrałyśmy nowszą wersję środowiska SDK dla naszego urządzenia AVD, ale równie dobrze można skorzystać ze starszej wersji. Urządzenia AVD oparte na nowszym zestawie SDK współpracują z aplikacjami napisanymi w starszym środowisku programistycznym. Odwrotna możliwość nie wchodzi oczywiście w rachubę: aplikacja wymagająca funkcji zawartych w nowszym środowisku SDK nie zadziała na urządzeniu AVD opartym na starszej wersji SDK.

9. Teraz wybierz utworzone urządzenie AVD z listy. Zwróć uwagę, że po kliknięciu przycisku *Refresh* lista zostanie odświeżona. Kliknij *OK*.
10. W ten sposób uruchomisz swoją pierwszą aplikację na emulatorze (pokazaliśmy ją na rysunku 2.10)!

**Uwaga!**

Emulacja rozruchu urządzenia może zająć emulatorowi kilka minut. Zazwyczaj po załadowaniu systemu operacyjnego pojawi się ekran blokady systemu. Należy wtedy kliknąć przycisk *Menu* lub przewinąć suwak blokady, aby odblokować urządzenie AVD. Po odblokowaniu systemu powinna się pojawić aplikacja *HelloAndroidApp* na wirtualnym ekranie, co zostało przedstawione na rysunku 2.10. Dodatkowo należy też mieć świadomość, że emulator uruchamia w tle również inne aplikacje, więc co jakiś czas może wyskakiwać informacja o błędzie lub ostrzeżenie. Jeżeli pojawi się komunikat o błędzie, zazwyczaj można go zignorować i przejść do kolejnego etapu rozruchu. Na przykład jeżeli pojawi się informacja „application abc is not responding” („aplikacja abc przestała odpowiadać”), można albo zaczekać na jej uruchomienie, albo po prostu zmusić emulator do jej zamknięcia. Zasadniczo warto poczekać i pozwolić, żeby emulator uruchomił się bez błędów.



Rysunek 2.10. Aplikacja HelloAndroidApp uruchomiona na emulatorze

Wiadomo już, w jaki sposób utworzyć nową aplikację w Androidzie oraz jak ją uruchomić na emulatorze. Teraz przyjrzymy się uważniej urządzeniom AVD, po czym zagłębimy się w świat artefaktów oraz struktury aplikacji Androida.

## Wirtualne urządzenia AVD

Wirtualne urządzenie Androida (ang. *Android Virtual Device* — AVD) reprezentuje konfigurację wybranego modelu urządzenia. Na przykład można utworzyć urządzenie AVD symbolizujące telefon starszego rodzaju, działający zgodnie z wersją 1.5 środowiska SDK oraz posiadający kartę SD 32 MB. Cała koncepcja oparta jest na możliwości tworzenia urządzeń AVD obsługujących tworzone aplikacje oraz emulowania tych urządzeń w celu projektowania i testowania aplikacji. Definiowanie (oraz zmienianie) urządzeń AVD jest bardzo łatwym procesem do przeprowadzenia oraz umożliwia błyskawiczne testowanie aplikacji w różnych konfiguracjach. W poprzednim podrozdziale przedstawiliśmy sposób tworzenia urządzenia AVD w środowisku Eclipse. Można stworzyć większą liczbę urządzeń AVD, klikając *Window/Android SDK and AVD Manager*, a następnie wybierając węzeł *Virtual Devices* w panelu po lewej stronie ekranu. Poniżej został także opisany sposób tworzenia tych urządzeń z poziomu wiersza poleceń.

Do utworzenia urządzenia AVD wykorzystywany jest plik wsadowy *android*, umieszczony w katalogu *tools* (*c:\android-sdk-windows\tools*). Dzięki temu plikowi możliwe jest także zarządzanie utworzonymi urządzeniami AVD. Można je na przykład przeglądać oraz przenosić. Spis poleceń dostępnych dzięki plikowi *android* zostaje wyświetlony po wpisaniu w wierszu polecenia *android -help*. Na razie stworzymy urządzenie AVD.

Pliki urządzeń AVD domyślnie są przechowywane w katalogu profilu użytkownika (na wszystkich platformach) w folderze *.android\AVD*. Znajduje się tam nasze urządzenie AVD, stworzone do uruchomienia aplikacji „Witaj, świecie!”. Istnieje również możliwość przeniesienia

(lub edytowania) urządzeń AVD do innej lokalizacji. Stwórzmy teraz folder, w którym będzie przechowywany obraz naszego urządzenia AVD, na przykład *c:\avd\*. Kolejnym etapem jest utworzenie listy dostępnych docelowych wersji Androida za pomocą następującego polecenia, wprowadzonego w oknie narzędzi:

```
android list target
```

W wyniku tego polecenia zostanie wygenerowana lista wszystkich zainstalowanych wersji Androida, gdzie każdy jej element będzie posiadał własny identyfikator. Teraz należy ponownie uruchomić plik *android* w celu wygenerowania urządzenia AVD. Trzeba otworzyć wiersz poleceń i wpisać następujące polecenie (należy wprowadzić własną ścieżkę do plików AVD oraz podać wartość argumentu *-t*, odpowiadającego identyfikatorowi wersji zainstalowanego środowiska SDK):

```
android create avd -n CupcakeMaps -t 2 -c 16M -p C:\avd\CupcakeMaps\
```

W tabeli 2.1 zostały objaśnione parametry narzędzia *android*.

**Tabela 2.1.** Parametry przypisane do pliku *android.bat*

Argument/polecenie	Opis
create avd	Polecenie utworzenia urządzenia AVD.
n	Nazwa urządzenia AVD.
t	Wersja środowiska SDK. Dla każdej docelowej wersji należy skorzystać z polecenia <i>android list target</i> , aby poznać jej identyfikator.
c	Pojemność karty SD wyrażona w bajtach. Wartość K oznacza kilobajty, a M — megabajty.
p	Ścieżka tworzonego urządzenia. Ten argument nie jest wymagany.
A	Umożliwia wykonywanie migawek. Jest to argument dodatkowy. Migawki zostaną omówione w podrozdziale „Uruchamianie emulatora”.

Wykonanie powyższego polecenia spowoduje wygenerowanie pliku urządzenia AVD; powinno zostać wyświetlane okno podobne do pokazanego na rysunku 2.11. Warto zwrócić uwagę, że po wpisaniu polecenia *create avd* system zapyta, czy utworzyć niestandardowy profil sprzętowy. Na razie wpiszmy *No*, dobrze jest jednak wiedzieć, że po udzieleniu odpowiedzi *Yes* będzie można skorzystać z wielu opcji konfiguracji urządzenia AVD, takich jak rozmiar ekranu, obecność aparatu i tak dalej.

Nawet jeżeli została określona alternatywna ścieżka dla pliku *CupcakeMaps* w programie *android.bat*, istnieje także kopia pliku *CupcakeMaps.ini* w folderze macierzystym *.android\AVD*. Jest to przemyślane działanie, dzięki któremu po kliknięciu *Window/Android SDK and AVD Manager* w środowisku Eclipse będą dostępne wszystkie urządzenia AVD, także te utworzone w wierszu polecień.

Spójrzmy ponownie na rysunek 2.4. Każda wersja Androida posiada określony poziom interfejsu API. Android 1.6 posiada poziom 4. interfejsu API, natomiast w Androidzie 2.1 przybiera on wartość 7. Wartości poziomów API nie są tożsame z identyfikatorami wersji docelowych, które są określone za pomocą parametru *-t* w poleceniu *android create avd*. Należy zawsze stosować polecenie *android list target* w celu uzyskania właściwej wartości identyfikatora, która będzie wykorzystana w poleceniu *android create avd*.

```
cmd C:\Windows\system32\cmd.exe
Microsoft Windows [Wersja 6.0.6002]
Copyright © 2006 Microsoft Corporation. Wszelkie prawa zastrzeżone.
C:\Users\Megaloxantha>cd c:\avd
c:\avd>android create avd -n CupcakeMaps -t 2 -c 16M -p c:\avd\CupcakeMaps
Created AVD 'CupcakeMaps' based on Google APIs <Google Inc.>
c:\avd>dir
    Wolumin w stacji C to Vista
    Numer seryjny voluminu: A826-8C46
    Katalog: c:\avd
2011-06-21  10:57    <DIR>          .
2011-06-21  10:57    <DIR>          ..
2011-06-21  10:57    <DIR>          CupcakeMaps
              0 plików   0 bajtów
              3 katalogów  13 134 159 872 bajtów wolnych
c:\avd>
```

Rysunek 2.11. Ekran wynikowy utworzenia urządzenia AVD za pomocą pliku android.bat

Należy również mieć na uwadze, że wybór interfejsu *Google API* z listy *SDK Target* daje dostęp do funkcji korzystania z map w urządzeniu AVD, podczas gdy wybranie interfejsu *Android 1.5* lub późniejszego nie zapewni takiej możliwości. O wiele więcej uwagi poświęcimy mapom w rozdziale 17.

## Poznanie struktury aplikacji Androida

Chociaż poszczególne aplikacje Androida będą się różniły rozmiarami oraz złożonością, ich struktura będzie podobna. Na rysunku 2.12 przedstawiono strukturę utworzonej niedawno aplikacji „Witaj, świecie!”.



Rysunek 2.12. Struktura aplikacji „Witaj, świecie!”

Aplikacje dla Androida składają się z elementów niezbędnych oraz opcjonalnych. W tabeli 2.2 zostały wymienione składniki aplikacji tworzonej dla Androida.

**Tabela 2.2.** Elementy składowe aplikacji systemu Android

Element składowy	Opis	Wymagany?
<i>AndroidManifest.xml</i>	Plik deskryptora aplikacji. Są w nim zdefiniowane aktywności, dostawcy usług, usługi oraz adresaci intencji, czyli elementy związane z daną aplikacją. Można w nim również zadeklarować uprawnienia wymagane przez aplikację, a także przydzielić określone uprawnienia dla innych aplikacji, korzystających z usług danego programu. Ponadto może być tu zamieszczona instrumentacja wykorzystywana do testowania danej aplikacji lub innych programów.	Tak
<i>src</i>	Folder przechowujący kod źródłowy aplikacji.	Tak
<i>assets</i>	Luźny zbiór plików i folderów.	Nie
<i>res</i>	Folder zawierający zasoby aplikacji. Jest to folder nadzędny wobec węzłów <i>drawable</i> , <i>anim</i> , <i>layout</i> , <i>menu</i> , <i>values</i> , <i>xml</i> oraz <i>raw</i> .	Tak
<i>drawable</i>	Folder mieszczący w sobie pliki obrazów lub deskryptorów obrazów używanych przez aplikację.	Nie
<i>anim</i>	W folderze tym są umieszczone pliki deskryptora napisane w języku XML, opisujące animacje wykorzystywane przez aplikację.	Nie
<i>layout</i>	Mieszcza się tu widoki aplikacji. Bardziej opłaca się tworzenie widoków poprzez deskryptory języka XML niż poprzez pisanie kodu.	Nie
<i>menu</i>	Folder zawierający pliki deskryptorów list menu aplikacji.	Nie
<i>values</i>	Przechowywane są w nim pozostałe zasoby wykorzystywane przez aplikację. Przykładowymi zasobami mogą być ciągi znaków, tablice, style oraz kolory.	Nie
<i>xml</i>	Znajdują się tu dodatkowe pliki XML wykorzystywane przez aplikację.	Nie
<i>raw</i>	Folder z dodatkowymi danymi — prawdopodobnie nieopisany w języku XML — wymaganymi przez aplikację.	Nie

Jak zostało pokazane w tabeli 2.2, aplikacja systemu Android składa się z trzech zasadniczych elementów: deskryptora aplikacji, zbioru zasobów oraz kodu źródłowego aplikacji. Jeżeli zignorować na chwilę plik *AndroidManifest.xml*, można zauważyć prostotę aplikacji: logika biznesowa przybiera formę kodu, a cała reszta to zasoby. Taka nieskomplikowana struktura przypomina szkielet aplikacji J2EE, w którym zasobom odpowiadają strony JSP, logice biznesowej — serwlety, a odpowiednikiem pliku *AndroidManifest.xml* jest plik *web.xml*.

Można również porównać modele projektowania w środowiskach J2EE oraz Android. W przypadku J2EE widoki są budowane za pomocą języka znaczników. W Androidzie wykorzystano tę samą filozofię, ale stosowanym językiem jest XML. Jest to korzystne rozwiązanie, gdyż nie ma konieczności wplatań widoku do głównego kodu; wygląd i zachowanie aplikacji można zmieniać poprzez edytowanie znaczników.

Należy również pamiętać o kilku ograniczeniach dotyczących zasobów. Po pierwsze, Android obsługuje jedynie liniową listę plików, znajdującą się w predefiniowanych plikach umieszczonych w folderze *res*. Na przykład nie może uzyskać dostępu do zagnieżdżonych folderów znajdujących się w katalogu *layout* (tak samo w przypadku pozostałych folderów podlegających do folderu *res*). Po drugie, istnieją pewne podobieństwa pomiędzy folderem *assets* oraz folderem *raw*, umieszczonym w katalogu *res*. W obydwu katalogach mogą być przechowywane nieskompresowane pliki, ale dane znajdujące się w folderze *raw* są uznawane za zasoby, a w folderze *assets* już nie. Zatem pliki z katalogu *raw* będą zlokalizowane, dostępne poprzez identyfikatory zasobów i tak dalej. Jednak informacje znajdujące się w katalogu *assets* są traktowane jako dane ogólnego przeznaczenia, pozbawione ograniczeń oraz obsługi zasobów. Warto zwrócić na to uwagę, gdyż pozbawienie danych znajdujących się w katalogu *assets* miana zasobów umożliwia utworzenie własnej hierarchii plików i folderów w jego wnętrzu (więcej informacji na temat zasobów znajduje się w rozdziale 3.).

**Uwaga!**

Dosyć wyraźnie widać, że w Androidzie całkiem często stosuje się język XML. Powszechnie wiadomo, że jest to dość rozbudowany język, rodzi się zatem pytanie, czy korzystanie z niego, gdy celem jest urządzenie posiadające ograniczone zasoby, ma sens. Okazuje się, że kod XML, używany podczas projektowania aplikacji, jest w rzeczywistości komplikowany do kodu binarnego przy użyciu narzędzia AAPR (ang. *Android Asset Packaging Tool* — narzędzie pakowania zasobów Androida). Zatem podczas instalowania aplikacji na urządzeniu pliki są konwertowane i przechowywane w formie kodu binarnego. Podczas uruchomienia plik jest odczytywany w tej formie i nie jest konwertowany ponownie na plik XML. Ta metoda łączy zalety obydwu technologii — można pracować z językiem XML i nie martwić się o ilość cennych zasobów urządzenia.

## Analiza aplikacji Notepad

Do tej pory nie tylko pokazaliśmy, w jaki sposób utworzyć oraz uruchomić w emulatorze nową aplikację dla Androida, lecz staraliśmy się, żeby Czytelnik zrozumiał elementy jej struktury. Teraz przyjrzymy się aplikacji Notepad, umieszczonej w pakiecie Android SDK. Jej poziom złożoności plasuje się pomiędzy naszą aplikacją „Witaj, świecie!” a w pełni rozwiniętą aplikacją dla Androida, zatem analiza jej składników pozwoli niejako pojąć realny proces projektowania w środowisku SDK. Będzie to szybka analiza aplikacji Notepad. Na początku może być trudno zrozumieć niektóre z pojęć, ale bez obaw — w następnych rozdziałach poświęcimy im znacznie więcej uwagi.

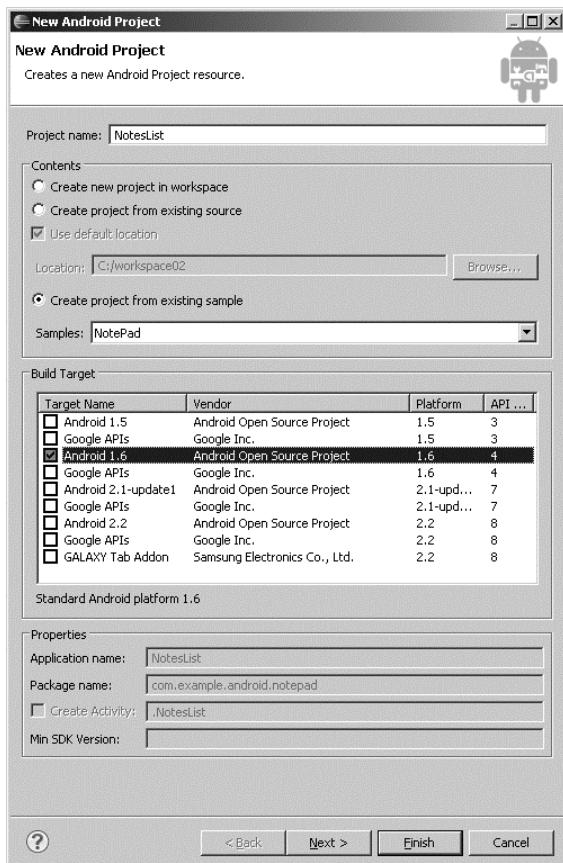
## Wczytanie oraz uruchomienie aplikacji Notepad

W tym podrozdziale wyjaśnimy, w jaki sposób załadować aplikację Notepad w środowisku Eclipse i uruchomić ją na emulatorze. Przed rozpoczęciem należy wiedzieć, że w aplikacji Notepad zaimplementowano kilka różnych opcji. Użytkownik może na przykład utworzyć nową notatkę, edytować istniejącą, usunąć ją, przejrzeć listę notatek i tak dalej. Po uruchomieniu

aplikacji nie będzie w niej żadnych zapisanych notatek, więc użytkownik ujrzy pustą listę. Po wciśnięciu przycisku *Menu* zostanie wyświetlona lista czynności, a wśród nich opcja dodania nowej notatki. Po utworzeniu nowego pliku można go edytować lub usunąć za pomocą odpowiedniej opcji.

Żeby wczytać przykładową aplikację Notepad w środowisku Eclipse, należy wykonać następujące czynności:

1. Uruchom program Eclipse.
2. Otwórz *File/New/Project*.
3. W oknie dialogowym *New Project* wybierz *Android/Android Project* i kliknij *Next*.
4. W kolejnym oknie wpisz *NotesList* jako nazwę projektu, wybierz opcję *Create project from existing sample*, następnie zaznacz pole *Android 1.6* na liście *Build Target*. Z rozwijanej listy wybierz aplikację *Notepad*. Zwróc uwagę, że jest ona umiejscowiona w folderze *platforms\android-1.6\samples* pakietu Android SDK, który wcześniej pobrałeś. Po wybraniu tej aplikacji zostanie automatycznie odczytany plik *AndroidManifest.xml* i zostaną wypełnione pozostałe pola w tym oknie dialogowym (rysunek 2.13).



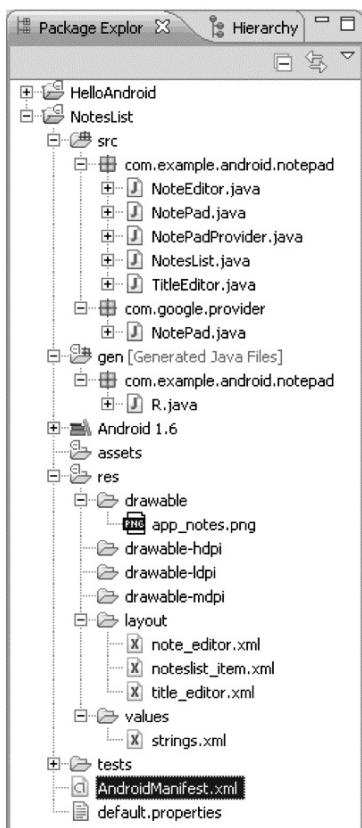
Rysunek 2.13. Tworzenie aplikacji Notepad

5. Kliknij przycisk *Finish*.

Teraz aplikacja NotesList powinna być dostępna w środowisku Eclipse. Jeżeli zostaną wyświetlane jakieś informacje o problemach związanych z tym projektem, można spróbować użyć opcji *Clean* z menu *Project*, aby je usunąć. Żeby uruchomić aplikację, można utworzyć aplikację uruchomieniową (podobnie jak to zrobiliśmy przy okazji programu „Witaj, świecie!”) lub kliknąć prawym przyciskiem ikonę projektu, wybrać opcję *Run As*, a następnie *Android Application*. Spowoduje to uruchomienie emulatora i zainstalowanie na nim aplikacji. Po wczytaniu emulatora wystarczy odblokować ekran emulatora, żeby została wyświetlona aplikacja NotesList. Aby się z nią zaznajomić, można po niej pomyszkować przez kilka minut.

## Rozłożenie kodu na czynnikie pierwsze

Przyjrzyjmy się teraz strukturze aplikacji (rysunek 2.14).



Rysunek 2.14. Struktura aplikacji Notepad

Jak widać, program zawiera kilka plików *java*, obrazów *.png*, trzy widoki (w folderze *layout*) oraz plik *AndroidManifest.xml*. Gdyby to była aplikacja wiersza poleceń, należałoby poszukać pliku, w którym jest umieszczona metoda *Main*. Zatem co jest odpowiednikiem metody *Main* w Androidzie?

W środowisku Android jest definiowana początkowa aktywność, zwana także aktywnością szczytowego poziomu. Jeżeli przyjrzeć się zawartości pliku *AndroidManifest.xml*, można tam znaleźć

jednego dostawcę oraz trzy aktywności. Aktywność NotesList wyznacza filtr intencji dla akcji android.intent.action.MAIN, a także dla kategorii android.intent.category.LAUNCHER. Po uruchomieniu aplikacji Androida zostaje ona wczytana przez urządzenie i jest odczytywany plik *AndroidManifest.xml*. Zostają wyszukane i uruchomione aktywności posiadające filtr intencji, który składa się z aktywności MAIN oraz kategorii LAUNCHER, tak jak pokazano poniżej.

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

Po odnalezieniu właściwej aktywności urządzenie musi powiązać ją z rzeczywistą klasą. Dokonuje tego poprzez połączenie nazwy głównego pakietu z nazwą aktywności, w naszym przypadku będzie to com.example.android.notepad.NotesList (listing 2.1).

---

#### **Listing 2.1.** Plik AndroidManifest.xml

---

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.notepad"
>
    <application android:icon="@drawable/app_notes"
        android:label="@string/app_name"
    >
        <provider android:name="NotePadProvider"
            android:authorities="com.google.provider.NotePad"
        />
        <activity android:name="NotesList" android:label="@string/title_notes_list">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.GET_CONTENT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
            </intent-filter>
        </activity>
    ...
</manifest>
```

---

Nazwa głównego pakietu aplikacji jest zdefiniowana jako atrybut elementu <manifest> w pliku *AndroidManifest.xml*, a każda aktywność posiada atrybut nazwy.

Po określeniu początkowej aktywności zostaje ona uruchomiona. Następuje również wywołanie metody *onCreate()*. Przyjrzyjmy się elementowi *NotesList.onCreate()*, przedstawionemu na listingu 2.2.

**Listing 2.2.** Metoda onCreate

---

```

public class NotesList extends ListActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setDefaultKeyMode(DEFAULT_KEYS_SHORTCUT);
        Intent intent = getIntent();
        if (intent.getData() == null) {
            intent.setData(Notes.CONTENT_URI);
        }

        getListView().setOnCreateContextMenuListener(this);

        Cursor cursor = managedQuery(getIntent().getData(), PROJECTION, null, null,
            Notes.DEFAULT_SORT_ORDER);

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
            R.layout.noteslist_item, cursor, new String[] { Notes.TITLE },
            new int[] { android.R.id.text1 });
        setListAdapter(adapter);
    }
}

```

---

Aktywności w Androidzie są przeważnie uruchamiane przez intencje, a także przez inne aktywności. Metoda `onCreate()` sprawdza, czy intencja bieżącej aktywności zawiera dane (notatki). Jeżeli nie zawiera, zostaje ustanowiony identyfikator URI, dzięki któremu zostają pobrane dane. W rozdziale 4. zademonstrujemy, że Android uzyskuje dostęp do danych poprzez dostawców treści korzystających z identyfikatorów URI. W tym przypadku identyfikator URI dostarcza wystarczająco wiele informacji, żeby pobrać dane z bazy danych. Stała `Notes.CONTENT_URI` jest zdefiniowana jako element `static final` w pliku `Notepad.java`, na przykład w taki sposób:

```

public static final Uri CONTENT_URI =
    Uri.parse("content://" + AUTHORITY + "/notes");

```

Klasa `Notes` znajduje się wewnątrz klasy `Notepad`. Na razie wystarczy wiedzieć, że przedstawiony powyżej identyfikator URI sprawia, że dostawca treści pobiera wszystkie notatki. Gdyby identyfikator ten wyglądał następująco:

```

public static final Uri CONTENT_URI =
    Uri.parse("content://" + AUTHORITY + "/notes/11");

```

to używany dostawca treści zwróciłby notatkę posiadającą identyfikator o wartości 11. Temat dostawców treści oraz identyfikatorów URI zostanie poruszony w rozdziale 4.

Klasa `NotesList` jest dopełnieniem klasy `ListActivity`, definiującej sposób wyświetlania danych w postaci listy. Składniki listy są zarządzane poprzez wewnętrzną klasę `ListView` (element interfejsu UI), wyświetlającą notatki w oknie listy. Po wstawieniu identyfikatora URI do intencji danej aktywności aktywność ta zgłasza gotowość do zbudowania kontekstowego menu dla notatek. Jeżeli Czytelnik starał się poznać tę aplikację, zauważył zapewne, że w zależności od wybranego elementu wyświetlane jest menu kontekstowe. Jeśli na przykład zostanie zaznaczona notatka, zostaną wyświetcone opcje *Edit note* oraz *Edit title*. Jeżeli notatka nie zostanie zaznaczona, dostępna będzie opcja *Add note*.

Widzimy następnie, że aktywność wykonuje zarządzaną kwerendę, w wyniku czego pojawia się kursor. Określenie „zarządzana kwerenda” oznacza, że Android będzie zarządzał przywołanym kursorem. Innymi słowy, w razie usunięcia lub ponownego wczytania do pamięci ani aplikacja, ani aktywność nie będą musiały obsługiwać takich aspektów zarządzania kursorem, jak jego pozycjonowanie, wczytywanie lub usunięcie z pamięci. Interesujące parametry elementu `managedQuery()` zostały opisane w tabeli 2.3.

**Tabela 2.3.** Parametry elementu `Activity.managedQuery`

Parametr	Typ danych	Opis
URI	Uri	Identyfikator URI dostawcy treści
projection	String[]	Zwracana kolumna (nazwy kolumn)
selection	String	Opcjonalna klauzula where
selectionArgs	String[]	Wybierane argumenty, w przypadku gdy kwerenda zawiera znaki zapytanego
sortOrder	String	Kolejność sortowania zestawu wynikowego

Elementy `managedQuery()` oraz bliźniaczy `query()` omówimy w dalszej części tego podrozdziału oraz w rozdziale 4. Na razie istotna jest informacja, że kwerendy w Androidzie zwracają dane tabularyczne. Parametr `projection` pozwala określić interesujące nas kolumny. Można także ograniczyć wynikowy zestaw oraz posortować go za pomocą klauzuli sortowania, używanych w języku SQL (na przykład `asc` lub `desc`). Należy zauważyć także, że kwerenda w Androidzie musi zwrócić kolumnę o nazwie `_ID`, żeby móc obsługiwać wyświetlanie pojedynczych rekordów. Ponadto należy znać typ danych zwracanych przez dostawcę treści — czy kolumna zawiera dane typu `string`, `int`, `binary` i tak dalej.

Po wykonaniu kwerendy zwrócony kursor jest przekazywany konstruktorowi elementu `SimpleCursorAdapter`, przekształcającemu rekordy zestawu danych w elementy interfejsu użytkownika (`ListView`). Przyjrzyjmy się bliżej parametrom przekazywanym do konstruktora elementu `SimpleCursorAdapter`:

```
SimpleCursorAdapter adapter =
    new SimpleCursorAdapter(this, R.layout.noteslist_item,
    cursor, new String[] { Notes.TITLE }, new int[] { android.R.id.text1 });
```

W szczególności zwróćmy uwagę na drugi parametr: identyfikator widoku reprezentującego elementy w metodzie `ListView`. Jak się będzie można przekonać w rozdziale 3., Android zawiera automatycznie generowaną klasę użytkową, w której znajdują się odniesienia do zasobów projektu. Jest to tak zwana klasa R (ang. `resources` — zasoby). Mieści się ona w pliku `R.java`, który można dostrzec na rysunku 2.14. Podczas komplikowania projektu narzędziem AAPT tworzy klasę R z zasobów umieszczonych w folderze `res`. Na przykład można umieścić wszystkie zasoby składające się z ciągów znaków w folderze `values`, a narzędziem AAPT wygeneruje identyfikator `public static` dla każdego z tych zasobów. Android obsługuje w ten sposób wszystkie zasoby. Na przykład w konstruktorze elementu `SimpleCursorAdapter` aktywności `NotesList` przekazuje identyfikator widoku, który umożliwia wyświetlanie elementu listy notatek. Dzięki tej klasie użytkowej nie ma potrzeby umieszczania zasobów wewnętrz głównego kodu oraz uzyskuje się możliwość sprawdzania odniesień w trakcie komplikacji. Inaczej mówiąc, jeżeli zasób zostanie usunięty, klasa R straci do niego odniesienie i żaden kod powiązany z tym zasobem nie zostanie skompilowany.

Przyjrzyjmy się kolejnej koncepcji Androida, o której wspomnieliśmy nieco wcześniej: metodzie `onListItemClick()` w klasie `NotesList` (listing 2.3).

#### **Listing 2.3.** Metoda `onListItemClick`

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
    Uri uri = ContentUris.withAppendedId(getIntent().getData(), id);

    String action = getIntent().getAction();
    if (Intent.ACTION_PICK.equals(action) ||
Intent.ACTION_GET_CONTENT.equals(action)) {
        setResult(RESULT_OK, new Intent().setData(uri));
    } else {
        startActivity(new Intent(Intent.ACTION_EDIT, uri));
    }
}
```

Metoda `onListItemClick()` jest wywoływana po zaznaczeniu notatki przez użytkownika. Metoda ta implementuje dwa sposoby jej wykorzystania. W przypadku pierwszego z nich aktywność związana z listą notatek może zostać wywołana za pomocą intencji, zatem użytkownik może wybrać określoną notatkę, która będzie zwrócona do aktywności wywołującej. W drugim przypadku wystarczy po prostu spojrzeć na listę notatek; po zaznaczeniu notatki bieżąca aktywność wywoła aktywność odpowiedzialną za edycję wybranej notatki. Po zaznaczeniu notatki metoda ta tworzy identyfikator URI poprzez dodanie identyfikatora danej notatki do bazowego identyfikatora URI. Jeżeli nasza aktywność została wywołana za pomocą intencji w celu zaznaczenia lub pobrania zawartości notatki, wywołujemy metodę `setResult()` zwracającą obiektem wywołującemu identyfikator URI danej notatki. W drugim przypadku identyfikator ten zostaje przekazany metodzie `startActivity()` wraz z nową intencją. Użycie metody `startActivity()` jest jednym ze sposobów uruchomienia aktywności: aktywność zostaje rozpoczęta, jednak po jej zakończeniu nie zostaje wyświetlony raport z wynikami. Inną możliwością uruchomienia aktywności jest użycie metody `startActivityForResult()`. Za jej pomocą można rozpocząć aktywność i wykorzystać wywoływanie zwrotne, aby zostać poinformowanym o jej zakończeniu, w celu uzyskania wyników. Na przykład aktywność wywołująca proces zaznaczania notatki w aplikacji `NotesList` mogłaby wykorzystać metodę `startActivityForResult()`, dzięki czemu istnieje możliwość powiadomienia w momencie, gdy aktywność aplikacji `NotesList` będzie wywoływać metodę `setResult()`.

W tym momencie można zacząć się zastanawiać, jak wygląda interakcja użytkownika względem aktywności. Na przykład jeżeli uruchomiona aktywność uruchamia następną aktywność, a ta z kolei uruchamia jeszcze inną aktywność (i tak dalej), to z którą aktywnością pracuje użytkownik? Czy może kontrolować jednocześnie wszystkie aktywności, czy może jest ograniczony do jednej? Okazuje się, że aktywności posiadają zdefiniowany cykl życia. Są one utrzymywane w stosie aktywności, na którego szczytce znajduje się uruchomiona aktywność. Jeżeli aktywność uruchomi inną aktywność, pierwsza uruchomiona aktywność przesunie się w dół stosu, a nowa zostanie umieszczona na jego szczytce. Aktywności znajdujące się na niższych poziomach stosu mogą się znajdować w stanie wstrzymania lub zatrzymania. Wstrzymana aktywność jest częściowo lub całkowicie widoczna dla użytkownika; aktywność zatrzymana jest dla niego niewidoczna. System może usunąć ze stosu wstrzymane lub zatrzymane aktywności, jeżeli się okaże, że trzeba zwolnić zasoby zajmowane przez te aktywności.

Przejdźmy teraz do trwałości danych. Notatki tworzone przez użytkownika zapisywane są w rzeczywistej bazie danych urządzenia. Ścisłe mówiąc, magazynem notatek programu Notepad jest baza danych SQLite. Wcześniej wspomniana metoda `managedQuery()` służy do określania danych w bazie danych poprzez dostawcę treści. Prześledźmy, w jaki sposób identyfikator URI, dostarczony metodzie `managedQuery()`, powoduje wykonanie kwerendy w bazie SQLite. Przypomnijmy, że identyfikator URI przekazany metodzie `managedQuery()` wygląda następująco:

```
public static final Uri CONTENT_URI =
Uri.parse("content://" + AUTHORITY + "/notes");
```

Identyfikatory URI treści zawsze przybierają następującą formę: `content://`, następnie uprawnienie (`AUTHORITY`), a na końcu segment ogólny (zależny od kontekstu). Ponieważ identyfikator URI nie zawiera rzeczywistych informacji, w jakiś sposób musi wpływać na wykonanie kodu generującego dane. Jaki jest związek pomiędzy tym identyfikatorem a kodem? W jaki sposób odniesienie URI wpływa na kod produkujący informacje? Czy identyfikator URI jest usługą HTTP lub sieciową? Okazuje się, że identyfikator URI, a dokładniej jego część związana z uprawnieniami, jest skonfigurowany w pliku *AndroidManifest.xml* jako dostawca treści, na przykład następująco:

```
<provider android:name="NotePadProvider"
    android:authorities="com.google.provider.NotePad"/>
```

Kiedy Android trafi na identyfikator URI, który należy przeanalizować, odczytuje jego część związaną z uprawnieniami i sprawdza klasę `ContentProvider` skonfigurowaną dla tych uprawnień. Aplikacja Notepad posiada klasę `NotePadProvider`, umieszczoną w pliku *AndroidManifest.xml*, skonfigurowaną dla uprawnienia `com.google.provider.NotePad`. Na listingu 2.4 został przedstawiony niewielki wycinek tej klasy.

#### **Listing 2.4. Klasa NotePadProvider**

---

```
public class NotePadProvider extends ContentProvider
{
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
    String[] selectionArgs, String sortOrder) {}

    @Override
    public Uri insert(Uri uri, ContentValues initialValues) {}

    @Override
    public int update(Uri uri, ContentValues values, String where,
    String[] whereArgs) {}

    @Override
    public int delete(Uri uri, String where, String[] whereArgs) {}

    @Override
    public String getType(Uri uri) {}

    @Override
    public boolean onCreate() {}

    private static class DatabaseHelper extends SQLiteOpenHelper {}
```

---

```

@Override
    public void onCreate(SQLiteDatabase db) {}

@Override
    public void onUpgrade(SQLiteDatabase db,
int oldVersion, int newVersion) {
    /**
     */
}
}

```

---

Klasa NotePadProvider rozszerza funkcjonalność klasy ContentProvider. Ta druga klasa definiuje sześć abstrakcyjnych metod, z których cztery są operacjami CRUB (ang. *Create*, *Read*, *Update*, *Delete* — tworzenie, odczyt, aktualizacja, usuwanie). Pozostałe dwie metody to `onCreate()` oraz `getType()`. Zwrócić uwagę, że metoda `onCreate()` jest wywoływana podczas pierwszego utworzenia dostawcy treści. Z kolei metoda `getType()` dostarcza typ MIME dla zestawu wyników (po przeczytaniu rozdziału 3. znaczenie typów MIME stanie się zrozumiałe).

Innym interesującym składnikiem klasy NotePadProvider jest wewnętrzna klasa DatabaseHelper, stanowiąca rozwinięcie klasy SQLiteOpenHelper. Rolą obydwu klas jest inicjalizacja, otwieranie oraz zamknięcie bazy danych aplikacji Notepad, a także wykonywanie innych operacji bazodanowych. Co ciekawe, klasa DatabaseHelper składa się wyłącznie z kilku wierszy kodu (listing 2.5), podczas gdy większość pracy wykonuje implementacja klasy SQLiteOpenHelper.

#### **Listing 2.5. Klasa DatabaseHelper**

---

```

private static class DatabaseHelper extends SQLiteOpenHelper {

    DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE " + NOTES_TABLE_NAME + " (" +
            + Notes._ID + " INTEGER PRIMARY KEY," +
            + Notes.TITLE + " TEXT," +
            + Notes.NOTE + " TEXT," +
            + Notes.CREATED_DATE + " INTEGER," +
            + Notes.MODIFIED_DATE + " INTEGER" +
            + ");");
    }

    /**
     */
}

```

---

Jak zostało przedstawione na listingu 2.5, metoda `onCreate()` generuje tabelę aplikacji Notepad. Należy zwrócić uwagę, że konstruktor klasy wywołuje konstruktor superklasy za pomocą nazwy tabeli. Superklasa wywoła metodę `onCreate()` jedynie w wypadku, gdy taka tabela nie istnieje w bazie danych. Warto również zauważyć, że jedną z kolumn w tabeli aplikacji Notepad jest `_ID`, omówiona kilka stron wcześniej.

Przyjrzyjmy się teraz jednej z operacji CRUD: metodzie `insert()`, przedstawionej na listingu 2.6.

#### **Listing 2.6.** Metoda insert()

```
//...
SQLiteDatabase db = mOpenHelper.getWritableDatabase();
long rowId = db.insert(NOTES_TABLE_NAME, Notes.NOTE, values);
if (rowId > 0) {
    Uri noteUri = ContentUris.withAppendedId(
        NotePad.Notes.CONTENT_URI, rowId);
    getContext().getContentResolver().notifyChange(noteUri, null);
    return noteUri;
}
```

Metoda `insert()` wykorzystuje swoje wewnętrzne wystąpienie `DatabaseHelper`, żeby uzyskać dostęp do bazy danych, a następnie wstawia rekord notatek. Zwrócony identyfikator krotki zostaje następnie dołączony do identyfikatora URI, a taki nowy identyfikator zostaje zwrócony aplikacji wywołującej.

Czytelnik do tej pory powinien już być zaznajomiony ze strukturą aplikacji Androida. Poruszanie się w aplikacji Notepad, a także innych przykładowych programach nie powinno sprawiać problemów. Dobrym pomysłem jest uruchomienie przykładowych aplikacji i zapoznanie się z ich działaniem. Zajmijmy się teraz ogólnym cyklem życia aplikacji dla systemu Android.

## **Badanie cyklu życia aplikacji**

Cykl życia aplikacji utworzonej na Androida jest ściśle zarządzany przez system na podstawie potrzeb użytkownika, dostępnych zasobów i tak dalej. Użytkownik może zechcieć otworzyć na przykład przeglądarkę internetową, ale ostatecznie to system decyduje, czy aplikacja zostanie uruchomiona. Chociaż system jest głównym zarządcą, postępuje zgodnie z pewnymi zdefiniowanymi oraz logicznymi wytycznymi, pozwalającymi określić, czy aplikacja ma zostać wczytana, wstrzymana lub zatrzymana. Jeżeli użytkownik korzysta aktualnie z aktywności, system wyznaczy tej aplikacji wysoki priorytet. Z drugiej strony, jeżeli aktywność nie jest widoczna, a system zdecyduje, że należy zamknąć aplikację w celu zwolnienia zasobów, to zostanie zamknięty program posiadający mniejszy priorytet.

Porównajmy to z cyklem życia aplikacji sieciowych, stworzonych w środowisku J2EE. Są one w sposób luźny zarządzane przez kontener, w którym są uruchomione. Na przykład aplikacja może zostać usunięta z pamięci, w przypadku gdy jest ona bezczynna przez określony czas. Z reguły jednak kontener nie będzie umieszczał aplikacji w pamięci oraz usuwał jej stamtąd ze względu na obciążenie oraz (lub) dostępność zasobów. Zazwyczaj dostępna jest wystarczająca ilość zasobów, żeby jednocześnie mogło pozostawać uruchomionych wiele aplikacji. W przypadku Androida zasoby są bardziej ograniczone, więc system musi posiadać większą kontrolę nad aplikacjami.

### **Uwaga!**

W Androidzie każda aplikacja jest uruchomiona w oddzielnym procesie, posiadającym własną wirtualną maszynę. W ten sposób zapewniono środowisko chronionej pamięci. Poprzez przydzielenie aplikacji do indywidualnych procesów system może określić ich priorytet. Na przykład uruchomiony w tle proces wykonujący zadanie znacznie pochłaniające zasoby procesora nie może blokować przychodzącego połączenia telefonicznego.

Koncepcja cyklu życia aplikacji jest logiczna, jednak podstawowa struktura aplikacji systemu Android komplikuje sprawę. Gwoli ściśleści, architektura aplikacji jest zorientowana na składniki oraz integrację. Pozwala to na wzbogacenie doznań użytkownika, możliwość bezproblemowego wielokrotnego korzystania z aplikacji oraz łatwość jej integracji, jednak przed menedżerem cyklu życia stoi bardzo skomplikowane zadanie.

Rozważmy typowy scenariusz. Użytkownik rozmawia z kimś przez telefon i musi otworzyć wiadomość e-mail, żeby odpowiedzieć na zadane przez rozmówcę pytanie. Przechodzi do ekranu głównego, otwiera aplikację pocztową, klikając adres łącza do witryny zawierającej poszukiwaną wiadomość i przytaczając jej fragment ze strony internetowej. W takim przypadku wymagane są cztery aplikacje: ekranu głównego, telefonu, pocztowa oraz przeglądarka. Użytkownik w sposób ciągły może zmieniać te aplikacje, jednak w tle system zapisuje oraz przywraca ich stan. Przykładowo po kliknięciu adresu łącza w wiadomości e-mail system zapisuje metadane uruchomionej aktywności tej wiadomości, zanim przekaże aktywności przeglądarki dane potrzebne do przekierowania na adres URL. Tak naprawdę system zapisuje metadane każdej aktywności przed uruchomieniem następnej, dzięki czemu może do niej wrócić (na przykład gdy użytkownik wraca do poprzedniej strony). Jeżeli wystąpi problem z ilością pamięci, zostanie zamknięty proces wykonujący aktywność, a w razie konieczności zostanie wznowiony.

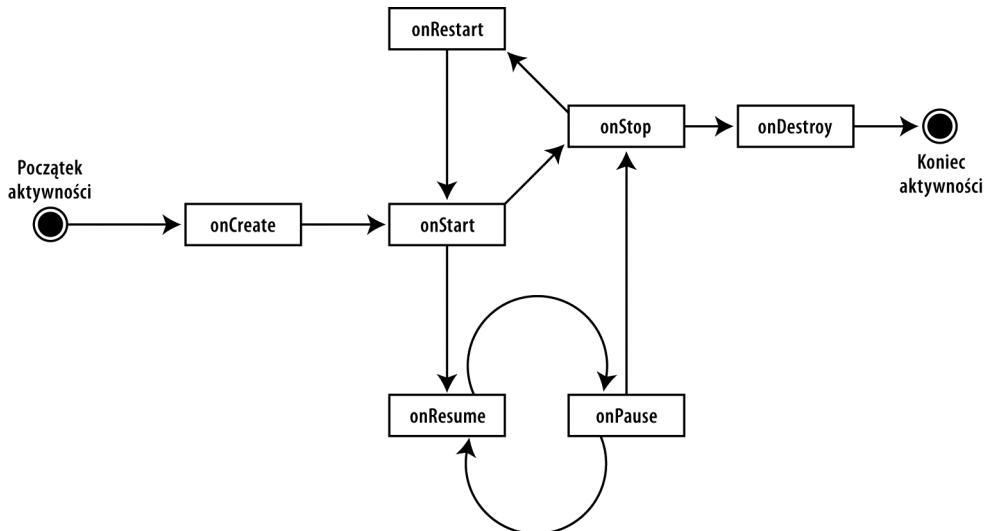
System Android jest wrażliwy na cykl życia aplikacji oraz jej elementów składowych. Zatem żeby stworzyć stabilną aplikację, należy zrozumieć zdarzenia cyklu życia oraz nauczyć się nimi posługiwać. Procesy korzystające z danej aplikacji oraz jej składników natrafiają na różnorodne zdarzenia cyklu życia i istnieje możliwość zaimplementowania wywołań zwrotnych, zajmujących się zmianami ich stanu. Na początek warto zapoznać się z wywołaniami cyklu życia aktywności (listing 2.7).

#### **Listing 2.7.** Metody cyklu życia aktywności

```
protected void onCreate(Bundle savedInstanceState);  
protected void onStart();  
protected void onRestart();  
protected void onResume();  
protected void onPause();  
protected void onStop();  
protected void onDestroy();
```

Na listingu 2.7 zostały wypisane metody, które są wywoływane podczas cyklu życia aktywności. Dla stworzenia stabilnej struktury aplikacji istotne jest zrozumienie, kiedy dana metoda jest wywoływana przez system. Nie wszystkie metody muszą być implementowane. Jeżeli zostaną użyte wszystkie wywołania, należy również stworzyć analogiczne wersje dla superklas. Na rysunku 2.15 zostały pokazane przejścia pomiędzy stanami aktywności.

System może uruchamiać oraz zatrzymywać aktywności w zależności od tego, co się w nim dzieje. Metoda `onCreate()` jest wywoływana podczas pierwszego utworzenia aktywności. Po tej metodzie zawsze pojawia się metoda `onStart()`, jednak wywołanie metody `onCreate()` nie zawsze następuje *przed* wywołaniem `onCreate()`, gdyż metoda ta może zostać wywołana w przypadku zatrzymania aplikacji. Po wywołaniu metody `onStart()` aktywność nie jest jeszcze dostępna dla użytkownika. Po metodzie `onStart()` wywoływana jest metoda `onResume()`, w momencie gdy aktywność znajduje się na pierwszym planie i jest dostępna dla użytkownika. To właśnie teraz użytkownik może bezpośrednio korzystać z aplikacji.



**Rysunek 2.15.** Zmiany stanów aktywności

Kiedy użytkownik zdecyduje się skorzystać z innej aktywności, system przywoła metodę `onPause()` dla opuszczanej aktywności. Z tego miejsca może zostać wywołana metoda `onResume()` lub `onStop()`. Ta pierwsza metoda jest wywoływana na przykład wtedy, gdy użytkownik przywróci aktywność na pierwszy plan. Jeżeli stanie się ona niewidoczna dla użytkownika, zostanie wywołana metoda `onStop()`. Jeżeli po tym wywołaniu aktywność zostanie przywrócona na pierwszy plan, nastąpi przywołanie metody `onRestart()`. Jeżeli aktywność znajduje się w stosie używanych aktywności, lecz jest niewidoczna dla użytkownika, a system postanowi ją zakończyć, zostanie wywołana metoda `onDestroy()`.

Omówiony model stanów aktywności może wydawać się skomplikowany, jednak umieszczenie wszystkich metod nie jest konieczne. Tak naprawdę najczęściej będą wykorzystywane metody `onCreate()`, `onResume()` oraz `onPause()`. Pierwsza metoda będzie służyć do tworzenia interfejsu UI danej aktywności. W tej metodzie dane będą wiązane z widżetami, a procedury obsługi zdarzeń — z elementami interfejsu użytkownika. Metoda `onPause()` jest wykorzystywana w przypadku konieczności przechowywania istotnych danych w magazynie aplikacji. Jest to ostatnia bezpieczna metoda wywoływana przed zamknięciem aplikacji. Metody `onStop()` oraz `onDestroy()` nie zawsze są wywoływane, więc nie należy na nie liczyć w przypadku tworzenia szczególnie ważnych programów.

Jakie wnioski powinny się nasuwać z powyższych wywodów? System zarządza aplikacją i może w każdej chwili uruchomić, zatrzymać lub przywrócić każdy z jej składników. Chociaż składniki te są kontrolowane przez system, nie są one całkowicie oddzielone od aplikacji. Innymi słowy, jeżeli system uruchomi aktywność w aplikacji, można liczyć na kontekst aplikacji w tej aktywności. Na przykład istnieje możliwość posiadania zmiennych globalnych, współdzielonych przez aktywności w aplikacji. Taką zmienną globalną tworzy się poprzez napisanie rozszerzenia klasy `android.app.Application`, a następnie inicjowanie jej w metodzie `onCreate()` (listing 2.8). Aktywności oraz inne składniki aplikacji będą uzyskiwały dostęp do tych odniesień bez obaw, że nie zostaną uruchomione. Koncepcja ta zostanie szczegółowo omówiona w rozdziale 11.

**Listing 2.8.** Rozszerzenie klasy Application

```
public class MyApplication extends Application
{
    // zmienna globalna
    private static final String myGlobalVariable;

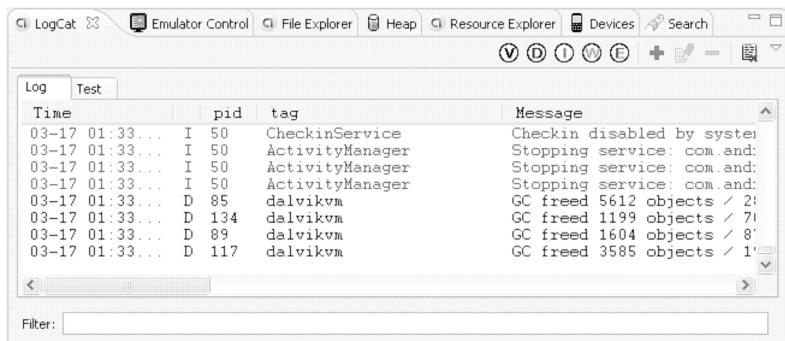
    @Override
    public void onCreate()
    {
        super.onCreate();
        // ... tutaj następuje inicjacja zmiennych globalnych
        myGlobalVariable = loadCacheData();
    }

    public static String getMyGlobalVariable() {
        return myGlobalVariable;
    }
}
```

Do tej pory omówiliśmy podstawy tworzenia aplikacji w systemie Android, uruchamianie programu na emulatorze, ogólną budowę aplikacji oraz kilka najpowszechniejszych funkcji spotykanych w tych programach. Nie pokazaliśmy jednak, w jaki sposób należy rozwiązywać problemy pojawiające się podczas pisania aplikacji. W ostatnim podrozdziale omówimy usuwanie błędów z programu.

## Usuwanie błędów w aplikacji

Po napisaniu kilku wierszy pierwszej aplikacji wiele osób z pewnością zacznie się zastanawiać, czy będzie możliwe przeprowadzenie sesji usuwania błędów podczas korzystania z aplikacji. Odpowiedź brzmi: „tak”. Zestaw Android SDK został zaopatrzony w wiele narzędzi pozwalających na sprawdzanie aplikacji pod kątem błędów. Są one zintegrowane ze środowiskiem Eclipse (niewielki przykład można ujrzeć na rysunku 2.16).



**Rysunek 2.16.** Narzędzia do usuwania błędów, które można wykorzystać podczas tworzenia aplikacji

Jednym z takich narzędzi jest *LogCat*. Aplikacja ta wyświetla komunikaty dziennika tworzone podczas korzystania z klas `android.util.Log`, `System.out.println`, wyjątków i tak dalej. Podczas gdy klasa `System.out.println` działa i informacje są wyświetlane w oknie *LogCat*,

do wyświetlenia komunikatów z aplikacji należy użyć klasy `android.util.Log`. Są w niej zdefiniowane znajome metody informacyjne, ostrzeżeń oraz błędów, które można filtrować w oknie *LogCat*. Przykładem polecenia Log jest:

```
Log.v("string TAG", "Ta rozwleka wiadomość zostanie zapisana w dzienniku");
```

Szczególnie interesującą funkcją narzędzia *LogCat* jest możliwość przeglądania komunikatów dziennika podczas testowania aplikacji na emulatorze. Nic jednak nie stoi na przeszkodzie, aby przeglądać je w przypadku podłączonego rzeczywistego urządzenia do stacji roboczej, gdy znajduje się w trybie debugowania. W rzeczywistości są one przechowywane w taki sposób, że możemy odzyskać większość najnowszych wiadomości już po odłączeniu urządzenia, w trakcie ich rejestrowania. W momencie podłączenia urządzenia przy włączonym oknie *LogCat* ujrzymy kilkaset najnowszych wpisów dziennika.

Należy zdawać sobie sprawę z dwóch faktów dotyczących debugowania aplikacji na fizycznym urządzeniu. Po pierwsze, aplikacja musi być ustawiona w trybie usuwania błędów w pliku *AndroidManifest.xml*. W tym celu należy dodać atrybut `android:debuggable="true"` w znaczniku `<application>`. Na szczęście wtyczka ADT wykona to automatycznie. Podczas tworzenia aplikacji testowych uruchamianych na emulatorze albo w przypadku bezpośredniego wdrażania aplikacji ze środowiska Eclipse na urządzenie fizyczne atrybut ten uzyskuje wartość `true`. W przypadku eksportowania aplikacji przeznaczonej już do użytkowania wtyczka ADT nie uruchomi trybu debugowania. Warto zauważyć, że po samodzielnym ustawieniu wartości tego atrybutu w pliku *AndroidManifest.xml* nie ulegnie on zmianie, bez względu na okoliczności. Druga ważna informacja jest taka, że urządzenie musi znajdować się w trybie debugowania USB. Opcję tę znajdziemy, wybierając z menu *Ustawienia* opcję *Aplikacje/Dla programistów*. Należy tutaj zaznaczyć opcję *Debugowanie USB*.

Chociaż narzędzie *LogCat* jest bardzo pomocne podczas przeglądania komunikatów dziennika, w trakcie działania aplikacji zdecydowanie warto mieć nad nią więcej kontroli oraz informacji na jej temat. W środowisku Eclipse istnieją dwie perspektywy, z którymi warto się zapoznać: **DDMS i Debug**. Skrót DDMS można rozwinąć jako *Dalvik Debug Monitor Server* (serwer monitora debugowania w środowisku Dalvik). Za pomocą tej perspektywy możemy obserwować poszczególne elementy aplikacji uruchomionej na emulatorze lub urządzeniu fizycznym — obserwować jej wątki, sterty (lub pamięć) wewnętrz aplikacji, a także uzyskać dostęp do eksploratora plików oraz kontrolera emulatora, dzięki któremu możemy symulować zdarzenia generowane przez system GPS, przychodzące połączenia telefoniczne lub wiadomości SMS. Eksplorator plików umożliwia przeglądanie systemu plików w urządzeniu, a nawet przenoszenie danych pomiędzy urządzeniem (lub emulatorem) a stacją roboczą. Można również wymuszać odzyskiwanie pamięci, usuwanie aplikacji z pamięci oraz wykonywać migawki.

Z poziomu perspektywy *DDMS* możemy wybrać jedną z uruchomionych aplikacji i podłączyć ją w trybie testowania. Zostanie wtedy uruchomiona perspektywa *Debug*. Debugowanie aplikacji rozpoczynamy również z poziomu perspektywy Java poprzez kliknięcie w jej obszarze prawym przyciskiem myszy i wybór opcji *Debug As/Android Application*; w ten sposób również zyskamy dostęp do perspektywy *Debug*. W każdym bądź razie środowisko Eclipse posiada funkcje umożliwiające śledzenie wątków, ustanawianie i usuwanie punktów kontrolnych w kodzie, kontrolowanie zmiennych oraz sprawdzanie lub pomijanie instrukcji. Jest to potężne narzędzie, pozwalające na rozwiązywanie problemów z aplikacjami.

Te narzędzia można przeglądać poprzez zaznaczenie perspektywy *DDMS* lub *Debug* w środowisku Eclipse. Każde z tych narzędzi można również uruchomić, wybierając *Window>Show View/Other/Android*. Jeżeli na przykład chcemy umieścić okno *LogCat* lub *File Explorer* w perspektywie Java, wystarczy wybrać *Window>Show View*, aby je dodać.

Istnieje także możliwość dokładnego śledzenia aplikacji za pomocą klasy `android.os.Debug`, zawierającej metodę rozpoczęcia śledzenia (`Debug.startMethodTracing(nazwa_bazowa)`) oraz zakończenia śledzenia (`Debug.stopMethodTracing()`). W urządzeniu (lub w emulatorze) zostanie utworzony plik śledzenia, dokładniej powstanie on na karcie SD. Nazwa tego pliku powstaje według wzorca `nazwa_bazowa.trace`. Można go następnie skopiować do stacji roboczej i obserwować dane wyjściowe znacznika za pomocą narzędzia `traceview`, znajdującego się w katalogu `tools` zestawu SDK, gdzie jedynym argumentem jest nazwa pliku śledzenia. Rozdział 19. został poświęcony kartom SD oraz metodom kopiowania z nich plików.

Mamy również do dyspozycji kilka innych narzędzi debugowania, które można wykorzystać z poziomu wiersza poleceń (lub okna narzędzi). Polecenie `adb` (ang. *Android Debug Bridge* — most debugowania w systemie Android) pozwala na instalowanie, aktualizowanie oraz usuwanie aplikacji. Można uruchomić powłokę na emulatorze lub w urządzeniu i wyprowadzić stamtąd szereg linuksowych poleceń, zrozumiałych dla systemu Android. Na przykład w ten sposób przeglądamy system plików, listę procesów, czytamy wpisy dziennika, a nawet łączymy się z bazami danych SQLite i wykonujemy polecenia języka SQL.

Kolejną przydatną techniką jest uruchomienie narzędzia Emulator Control, które z oczywistych względów współpracuje wyłącznie z emulatorem. Aby je uruchomić (gdy emulator jest już włączony), należy wpisać następujące polecenie w oknie narzędzi:

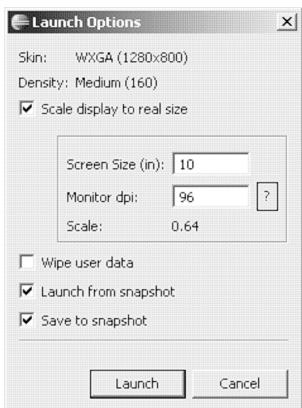
```
telnet localhost port#
```

gdzie `port#` jest numerem portu, na którym nasłuchuje emulator. Wartość tego parametru jest zazwyczaj podana w pasku tytułowym emulatora i często wynosi ona 5554. Po uruchomieniu konsoli emulatora możemy wpisywać polecenia pozwalające na symulowanie zdarzeń związanych z systemem GPS, wiadomościami SMS, a nawet na zmianę sieci i poziomu naładowania baterii.

## Uruchamianie emulatora

Pokazaliśmy wcześniej, w jaki sposób można uruchomić emulator z poziomu projektu w środowisku Eclipse. W większości przypadków chcemy najpierw włączyć emulator, a następnie wdrożyć i przetestować aplikację w już uruchomionym emulatorze. Aby go uruchomić w dowolnym momencie, musimy najpierw przejść do narzędzia *Android SDK and AVD Manager*, albo uruchamiając je bezpośrednio w katalogu `tools` pakietu *Android SDK*, albo wybierając je w oknie *Window* środowiska Eclipse. Gdy już uruchomimy menedżer, klikamy zakładkę *Virtual devices*, widoczną w panelu po lewej stronie, wybieramy właściwe urządzenie AVD z listy w prawym oknie i klikamy przycisk *Start*.

Po jego wcisnięciu pojawi się okno dialogowe *Launch Options* (rysunek 2.17). Możemy w nim definiować rozmiar okna emulatora oraz zmieniać opcje jego rozruchu i zamykania. Podczas pracy z urządzeniami AVD imitującymi urządzenia posiadające małe lub średnie wyświetlacze będziemy często ograniczać się do domyślnego rozmiaru ekranu. Jednak w przypadku dużych i bardzo dużych rozmiarów ekranu, na przykład takich jak w tabletach, domyślne wymiary wyświetlacza mogą nie pasować do rozmiaru monitora stacji roboczej. W takim przypadku możemy zaznaczyć opcję *Scale display to real size* (skaluj wyświetlacz do rzeczywistego rozmiaru) i wstawić odpowiednią wartość. Nazwa tej opcji może być nieco myląca, ponieważ tablety mogą posiadać inne gęstości wyświetlacza od stacji roboczej, natomiast emulator nie potrafi dokładnie odwzorować fizycznych parametrów wyświetlacza na ekranie monitora. Przykładowo na mojej stacji roboczej, podczas symulowania tabletu obsługującego platformę Honeycomb



Rysunek 2.17. Okno dialogowe Launch Options

o rozmiarach 10 cali, „rzeczywisty rozmiar” 10 cali zostaje przeskalowany o współczynnik 0,64 i rozmiar mojego monitora, który jest nieco większy od 10 cali. Na podstawie wielkości i gęstości monitora należy wybrać najbardziej odpowiednią wartość.

Okno dialogowe *Launch Options* pozwala również konfigurować migawki. Zapisanie migawki (*Save to snapshot*) nieco wydłuży czas zamknięcia emulatora. Jak sama nazwa wskazuje, bieżący stan emulatora zostaje zapisany w pliku-obrazie migawki, który można wykorzystać podczas następnego uruchomienia emulatora, w celu pominięcia całej sekwencji rozruchu systemu. Uruchamianie emulatora przebiega znacznie szybciej w obecności migawki i rekompensuje czas potrzebny na jej zapisanie — zasadniczo rozpoczęźmy pracę od miejsca, w którym ją zakończyliśmy. Jeżeli chcemy uruchomić emulator w jego pierwotnym stanie, zaznaczamy opcję *Wipe user data*. Możemy również usunąć zaznaczenie opcji *Launch from snapshot*, aby umożliwić zapisywanie danych wraz z jednociesnym przeprowadzaniem całego procesu rozruchu. Ewentualnie istnieje możliwość utworzenia optymalnej migawki i pozostawienia wyłącznie opcji *Lauch from snapshot*; w ten sposób dana migawka będzie ciągle wykorzystywana, co spowoduje przyspieszenie zarówno procesu uruchamiania emulatora, jak też jego zamknięcia, ponieważ nie będzie za każdym razem tworzony nowy obraz migawki. Plik migawki jest przechowywany w tym samym katalogu co reszta plików-obrazów urządzenia AVD. Aby mieć możliwość korzystania z funkcji migawek, musimy zaznaczyć odpowiednią opcję podczas tworzenia urządzenia AVD.

## StrictMode

Wraz z wydaniem Androida w wersji 2.3 została wprowadzona nowa funkcja debugowania, nazwana *StrictMode*. Opcja ta — według firmy Google — została wykorzystana do wprowadzenia setek usprawnień w aplikacjach tej firmy stworzonych z myślą o tym systemie. Do czego więc właściwie ona służy? Będzie powiadala o naruszeniach zasad powiązanych z wątkami oraz wirtualną maszyną. Po wykryciu naruszenia zasad funkcja wyświetli alert z odniesieniem do stosu, w którym znajdowała się aplikacja w momencie naruszenia zabezpieczeń. Za pomocą alertu można wymusić zamknięcie aplikacji lub jedynie zapisać treść alertu w dzienniku i pozwolić aplikacji na dalsze działanie. Obecnie trudno określić szczegóły wspomnianych zasad, spodziewamy się też, że firma Google będzie dodawała kolejne zasady wraz z rozwojem Androida.

Obecnie są dostępne dwa rodzaje zasad w obrębie funkcji `StrictMode`. Pierwszy z nich jest związany z wątkami i jego podstawowym zadaniem jest współpraca z głównym wątkiem (zwanym również wątkiem interfejsu użytkownika). Prowadzenie zapisu oraz odczytu danych dyskowych w obrębie głównego wątku nie jest dobrym rozwiązaniem, podobnie jak uzyskiwanie za jego pomocą dostępu do sieci. Firma Google dodała punkty zaczepienia funkcji `StrictMode` do kodu odpowiedzialnego za operacje zapisu-odczytu oraz operacje sieciowe. Jeżeli uruchomimy funkcję `StrictMode` w jednym z wątków, który próbuje uzyskać dostęp do przestrzeni dyskowej lub sieci, zostaniemy o tym powiadomieni. Musimy wybrać, które aspekty zasad `ThreadPolicy` spowodują wywołanie alertu, oraz rodzaj alertu. Wśród naruszeń zasad, na które możemy zwracać uwagę, znajdziemy takie jak niestandardowo powolne wywołania, odczyty danych z dysku, zapisy danych na dysku oraz dostęp do sieci. Spośród rodzajów alertów mamy do wyboru zapis komunikatu w narzędziu `LogCat`, wyświetlenie okna dialogowego, błyszczenie wyświetlacza, zapis komunikatu w pliku dziennika `DropBox` lub zawieszenie działania aplikacji. Najczęściej spotykamy się z zapisywaniem informacji w narzędziu `LogCat` oraz zawieszeniem działania aplikacji. Na listingu 2.9 przedstawiono przykładowy sposób konfigurowania funkcji `StrictMode` pod kątem zasad dotyczących wątków.

#### **Listing 2.9.** Konfigurowanie zasad `ThreadPolicy` funkcji `StrictMode`

```
StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
    .detectDiskReads()
    .detectDiskWrites()
    .detectNetwork()
    .penaltyLog()
    .build());
```

Zwróćmy uwagę, że za pomocą klasy `Builder` można w naprawdę prosty sposób ustanowić funkcję `StrictMode`. Wszystkie metody tej klasy, definiujące zasady, zwracają odniesienie do obiektu `Builder`, zatem można stworzyć z nich łańcuch, podobnie jak na listingu 2.9. Ostatnia z wywoływanych metod, `build()`, zwraca obiekt `ThreadPolicy`, który jest argumentem oczekiwany przez metodę `setThreadPolicy()` funkcji `StrictMode`. Zwróćmy uwagę, że metoda `setThreadPolicy()` jest statyczna, zatem tak naprawdę nie musimy tworzyć obiektu `StrictMode`. Metoda `setThreadPolicy()` bada bieżący wątek pod kątem zasad, zatem wszystkie następne działania wątku zostaną porównane z obiektem `ThreadPolicy` i w razie potrzeby zostanie wyświetlony alert. W powyższym kodzie zasady są tak zdefiniowane, że alert zostanie wygenerowany w przypadku odczytywania i zapisywania danych dyskowych oraz dostępu do sieci i zostanie on zapisany w dzienniku `LogCat`. Zamiast wypisywania poszczególnych metod wykrywania możemy zastosować metodę `detectAll()`. Nie ma również przeszkód, by dodawać lub wymieniać metody odpowiedzialne za ostrzeżenia. Na przykład możemy wprowadzić metodę `penaltyDeath()`, która spowoduje zawieszenie działania aplikacji zaraz po zapisaniu komunikatu w narzędziu `LogCat` (z kolei za to zdarzenie jest odpowiedzialna metoda `penaltyLog()`).

Ponieważ ten rodzaj funkcji `StrictMode` dotyczy wątku, dla danego wątku funkcja ta uruchamia się jednorazowo. Z tego powodu można uruchomić funkcję `StrictMode` na początku metody `onCreate()`, przypisanej do głównej aktywności, działającej w głównym wątku, i od tego momentu funkcja ta śledziłaby wszystkie działania przeprowadzane w tym wątku. W zależności od rodzaju poszukiwanego naruszenia pierwsza aktywność może dość szybko uruchomić funkcję `StrictMode`. Możemy ją również uruchomić dla aplikacji poprzez rozszerzenie klasy `Application` i dodanie konfiguracji funkcji `StrictMode` do metody `onCreate()` całej aplikacji.

Potencjalnie każdy element obecny w wątku może uruchomić funkcję `StrictMode`, zdecydowanie jednak nie musimy wywoływać kodu konfiguracyjnego we wszystkich miejscach; jeden raz całkowicie wystarczy.

Analogicznie do zasad `ThreadPolicy`, funkcja `StrictMode` zawiera zasady `VmPolicy`. Służą one do sprawdzania wycieków pamięci, w przypadku gdy obiekt bazy SQLite lub dowolny inny obiekt typu `Closeable` zostanie zakończony przed zamknięciem. Zasady `VmPolicy` są tworzone w podobny sposób za pomocą klasy `Builder`, co zostało przedstawione na listingu 2.10. Jedyna różnica pomiędzy zasadami `ThreadPolicy` a `VmPolicy` polega na niemożności wyświetlenia alertu jako okna dialogowego w tym drugim przypadku.

---

**Listing 2.10.** Konfigurowanie zasad `VmPolicy` funkcji `StrictMode`

---

```
StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
    .detectLeakedSqlLiteObjects()
    .penaltyLog()
    .penaltyDeath()
    .build());
```

---

Ponieważ proces konfiguracji jest przeprowadzany w wątku, funkcja `StrictMode` będzie wykrywała naruszenia nawet w przypadku kontrolnego przepływu pomiędzy obiektami. Po wystąpieniu naruszenia zasad bezpieczeństwa możemy się zdziwić, gdy zauważymy, że kod jest ciągle przetwarzany w głównym wątku, ale otrzymujemy też ślad stosu, dzięki któremu możemy odkryć, co się stało. Następnie można spróbować rozwiązać problem poprzez przeniesienie danego fragmentu kodu do osobnego wątku. Można również pozostawić kod bez zmian. Wszystko zależy od programisty. Oczywiście, najprawdopodobniej będzie trzeba wyłączyć funkcję `StrictMode` tuż przed wydaniem aplikacji na rynek; nie byłoby korzystne, aby programy zawieszaly się użytkownikom z powodu alertów.

Istnieje kilka sposobów wyłączenia funkcji `StrictMode` przed wdrożeniem aplikacji do użytkowania. Najprostszym rozwiązaniem jest usunięcie wywołań, jednak w późniejszych etapach staje się ono coraz bardziej skomplikowane. Można również wprowadzić logikę dwuwartościową na poziomie aplikacji i przeprowadzić test przed wywołaniem kodu funkcji `StrictMode`. W takim przypadku ustanowienie wartości `false` tuż przed okreaniem aplikacji światu w skuteczny sposób wyłączyłoby tę funkcję. Bardziej eleganckim rozwiązaniem jest wykorzystanie trybu debugowania aplikacji, zdefiniowanego w pliku `AndroidManifest.xml`. Jednym z atrybutów znacznika `<application>` w tym pliku jest `android:debuggable`. Jego wartość można ustawić jako `true` w trakcie debugowania aplikacji, w wyniku czego na obiekcie `ApplicationInfo` zostaje ustanowiona flaga, co można następnie odczytać w kodzie. Na listingu 2.11 pokazano, w jaki sposób można skorzystać z tej informacji, aby w trybie debugowania aplikacja posiadała aktywną funkcję `StrictMode` (a jeśli aplikacja nie będzie w trybie debugowania, funkcja ta zostanie zdezaktywowana).

---

**Listing 2.11.** Ustanawianie funkcji `StrictMode` wyłącznie w trybie debugowania

---

```
// Wraca tutaj, jeśli aplikacja nie znajduje się w trybie debugowania
ApplicationInfo appInfo = context.getApplicationInfo();
int appFlags = appInfo.flags;
if ((appFlags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
    // Tutaj przeprowadzana jest konfiguracja funkcji StrictMode
}
```

---

Podczas pisania aplikacji w środowisku Eclipse wtyczka ADT automatycznie ustanawia atrybut debugowania, co stanowi spore ułatwienie. W trakcie wdrażania aplikacji ze środowiska Eclipse do emulatora lub bezpośrednio do urządzenia fizycznego atrybut ten otrzyma wartość `true`, co spowodowałoby uruchomienie kodu funkcji `StrictMode` w powyższym kodzie. Podczas eksportowania aplikacji do wersji przeznaczonej do użytkowania wartość atrybutu `debuggable` zostanie zmieniona na `false`. Należy jednak pamiętać, że po ręcznej zmianie tego atrybutu nie będzie on automatycznie modyfikowany.

Wszystko to brzmi bardzo ładnie i elegancko, ale nie zadziała na wersji Androida starszej od 2.3. Aby móc jawnie używać funkcji `StrictMode`, musimy wykorzystywać środowisko obsługujące Androida w wersji 2.3 lub nowsze. W przypadku wprowadzenia powyższych kodów do środowiska starszego od wersji 2.3 zaczną powstawać błędy weryfikacji, ponieważ ta klasa po prostu nie istnieje w tym środowisku.

Aby wykorzystywać funkcję `StrictMode` w starszych wersjach Androida (do wersji 2.3), należy zastosować mechanizm refleksji. Dzięki temu można wywołać metody tej funkcji w sposób pośredni, jeśli są dostępne. Jeśli nie są dostępne, to możesz ponieść sromotną porażkę. Najprostsze rozwiązań zostało przedstawione na listingu 2.12; wywołujemy specjalną metodę, stworzoną wyłącznie dla starszych wersji Androida.

#### **Listing 2.12.** Wykorzystanie funkcji StrictMode za pomocą refleksji

---

```
try {
    Class sMode = Class.forName("android.os.StrictMode");
    Method enableDefaults = sMode.getMethod("enableDefaults");
    enableDefaults.invoke(null);
}
catch(Exception e) {
    // Funkcja StrictMode nieobsługiwana na tym urządzeniu; zaniechanie
    Log.v("StrictMode", "... nieobsługiwana. Pomijanie...");
}
```

---

W ten sposób można określić, czy klasa `StrictMode` istnieje. Jeżeli istnieje, nastąpi wywołanie metody `enableDefaults()`. Jeżeli klasa ta nie zostanie znaleziona, zostaje wywołany nasz blok `catch` wraz z wyjątkiem `ClassNotFoundException`. Jeżeli funkcja `StrictMode` istnieje, nie powinny pojawiać się wyjątki, ponieważ jedną z jej metod jest `enableDefaults()`. Metoda ta sprawia, że funkcja `StrictMode` wyłapuje wszystkie naruszenia zasad i zapisuje je w dzienniku `LogCat`. Ponieważ ta metoda jest statyczna, pierwszy argument przyjmuje wartość `null` podczas jej wywoływania.

Mogą się zdarzać sytuacje, w których zapisywanie wszystkich naruszeń jest niepożądane. Nic nie stoi na przeszkodzie, aby dołączać funkcję `StrictMode` do wątków innych od głównego, i to właśnie wtedy możemy ustanowić mniejszą liczbę alertów. Dobrym przykładem byłoby monitorowanie wątku, który służy do odczytu danych. W takim przypadku możemy albo nie wywoływać metody `detectDiskReads()` w obiekcie `Builder`, albo wywołać metodę `detectAll()`, a następnie `permitDiskReads()` w tym obiekcie. Istnieją również analogiczne metody `zezwoleń` dla pozostałych opcji zasad. Ale gdybyśmy chcieli dokonać czegoś podobnego w wersjach Androida starszych od 2.3, to czy istnieje na to jakiś sposób? Oczywiście, że tak!

Jeżeli funkcja `StrictMode` nie jest dostępna dla danej aplikacji, w przypadku próby jej aktywowania zostanie wyświetlony komunikat o błędzie `VerifyError`. Jeżeli umieścimy tę funkcję w klasie i następnie otrzymamy taki komunikat o błędzie, nie musimy się przejmować, gdy nie

będzie ona dostępna, a gdy będzie dostępna — wykorzystamy ją. Na listingu 2.13 widzimy przykładową klasę StrictModeWrapper, którą można dodać do aplikacji, natomiast listing 2.14 przedstawia kod wewnątrz aplikacji służący do konfigurowania funkcji StrictMode.

---

**Listing 2.13.** Stosowanie funkcji StrictMode w Androidzie starszym od wersji 2.3

```
import android.content.Context;
import android.content.pm.ApplicationInfo;
import android.os.StrictMode;

public class StrictModeWrapper {
    public static void init(Context context) {
        // sprawdza, czy atrybut android:debuggable posiada wartość true
        int appFlags = context.getApplicationInfo().flags;
        if ((appFlags & ApplicationInfo.FLAG_DEBUGGABLE) != 0) {
            StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
                .detectDiskReads()
                .detectDiskWrites()
                .detectNetwork()
                .penaltyLog()
                .build());
            StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                .detectLeakedSqlLiteObjects()
                .penaltyLog()
                .penaltyDeath()
                .build());
        }
    }
}
```

---

Jak widać, mamy tu do czynienia z takim samym kodem jak wcześniej, tutaj jednak łączymy w całość wszystkie zdobyte wcześniej informacje. Wreszcie, aby skonfigurować funkcję StrictMode w aplikacji, wystarczy dodać do niej kod widoczny na listingu 2.14:

---

**Listing 2.14.** Wywoływanie funkcji StrictMode w Androidzie starszym od wersji 2.3

```
try {
    StrictModeWrapper.init(this);
}
catch(Throwable throwable) {
    Log.v("StrictMode", "... jest nieosiągalna. Zaniechanie...");
}
```

---

Zwróćmy uwagę, że this jest lokalnym kontekstem dowolnego obiektu, którym się zajmujemy, na przykład z wnętrza metody onCreate() będącej częścią głównej aktywności. Kod z listingu 2.14 będzie działał z dowolną wersją systemu Android.

W ramach ćwiczenia Czytelnik może uruchomić środowisko Eclipse i stworzyć kopię aplikacji Notepad, wygenerowanej we wcześniejszej części rozdziału. Następnie można dodać nową klasę w katalogu src, wykorzystującą kod z listingu 2.13. Wewnątrz metody onCreate() w pliku NotesList.java należy teraz dodać taki kod jak na listingu 2.14, po czym uruchomić program na emulatorze obsługującym Androida w wersji starszej od 2.3. To samo można następnie sprawdzić dla wersji 2.3 lub późniejszej. Jeżeli funkcja StrictMode będzie niedostępna, w oknie

*LogCat* pojawi się informacja o jej braku, jednak aplikacja powinna dalej nieprzerwanie działać. W przypadku obecności funkcji StrictMode w oknie *LogCat* powinny od czasu do czasu pojawiać się informacje o naruszeniach zasad podczas korzystania z aplikacji Notepad.

## Odbośniki

Poniżej przedstawiamy pomocne odnośniki do tematów, które Czytelnik może zechcieć poznać dokładniej.

- <http://developer.motorola.com/docstools/> jest witryną firmy Motorola, na której można znaleźć dodatki do urządzeń oraz inne narzędzia programistyczne przystosowane do mikrotelefonów tego producenta, w tym takie jak MOTODEV Studio — alternatywę dla środowiska Eclipse.
- <http://developer.htc.com/> to witryna firmy HTC przeznaczona dla programistów w systemie Android.
- <http://innovator.samsungmobile.com/platform.main.do?platformId=1> stanowi stronę firmy Samsung dla programistów w systemie Android, na której można znaleźć dodatek zestawu Android SDK dla tabletu Samsung Galaxy Tab.
- <http://developer.android.com/guide/developing/tools/index.html> zawiera dokumentację programistyczną dla uprzednio opisanych narzędzi debugujących.
- <http://appinventor.gglabes.com/about/index.html> jest stroną środowiska App Inventor, kolejnej alternatywy służącej do tworzenia aplikacji dla systemu Android. Za stworzenie tego środowiska odpowiada firma Google Labs i jest ono przeznaczone dla osób niebędących programistami. Aplikacje są tutaj tworzone w sposób graficzny, podobnie jak logika stojąca za interfejsem użytkownika.
- <http://code.google.com/p/android-ui-utils/> zawiera łącza do użytecznych narzędzi, takich jak Android Asset Studio, które jest aplikacją sieciową służącą do tworzenia różnorodnych rodzajów ikon dla systemu Android. Warto zwrócić uwagę, że do obsługi aplikacji Android Asset Studio wymagane jest uruchomienie przeglądarki Google Chrome.
- <http://www.droiddraw.org/> — narzędzie do projektowania interfejsów użytkownika, w którym do tworzenia układów graficznych jest wykorzystywana funkcja przeciągania.

## Podsumowanie

W tym rozdziale zademonstrowaliśmy, w jaki sposób należy skonfigurować środowisko projektowe do tworzenia aplikacji dla systemu Android. Opisaliśmy podstawowe elementy budulcowe interfejsu API Androida, a także wprowadziliśmy pojęcia widoków, aktywności, intencji, dostawców treści oraz usług. W dalszej części przeanalizowaliśmy strukturę aplikacji Notepad pod kątem wspomnianych już bloków budulcowych oraz składników aplikacji. Następnie omówiliśmy istotę cyklu życia aplikacji pisanych na Androida. Na końcu wspomnieliśmy o narzędziach do usuwania błędów zaimplementowanych w zestawie Android SDK, zintegrowanych ze środowiskiem Eclipse.

A teraz wprowadzimy podstawy projektowania dla Androida. Następny rozdział został poświęcony zasobom.



# Korzystanie z zasobów

W rozdziale 2. skrótnie omówiliśmy strukturę aplikacji tworzonej dla Androida oraz wspomnieliśmy o pewnych kluczowych pojęciach. Opisaliśmy także zestaw Android SDK, narzędzia ADT środowiska Eclipse oraz możliwość uruchamiania aplikacji na emulatorach urządzeń AVD.

W tym oraz w kilku następnych rozdziałach będziemy kontynuować szczegółowe przedstawianie podstaw pracy z zestawem Android SDK. Zajmiemy się zasobami, dostawcami treści oraz intencjami. Są to trzy elementy niezbędne do naużenia się programowania dla systemu Android, a ich solidne opanowanie pozwoli na zrozumienie materiału opisanego w następnych rozdziałach.

Android wymaga dostępu do zasobów w celu definiowania elementów interfejsu użytkownika w deklaracyjny sposób. Metoda ta nie różni się wiele od stosowania znaczników w języku HTML. W tym sensie projektowanie interfejsu UI w Androidzie jest dosyć nowatorskie. Możliwe jest także określanie lokalizacji tych zasobów. W tym rozdziale zajmiemy się opisem wielkiej różnorodności zasobów dostępnych w Androidzie oraz wyjaśnimy, w jaki sposób można ich używać.

## Zasoby

Zasoby odgrywają kluczową rolę w architekturze Androida. W Androidzie zasobem może być plik (na przykład plik muzyczny) lub wartość (przykładowo nazwa okna dialogowego), powiązane z wykonywalną aplikacją. Te pliki i wartości są z nią powiązane w sposób umożliwiający ich modyfikowanie bez konieczności ponownego komplilowania aplikacji.

Znanymi Czytelnikowi rodzajami zasobów są ciągi znaków, kolory oraz mapy bitowe. Zamiast umieszczać na przykład ciągi znaków w kodzie aplikacji, można wykorzystać ich identyfikatory. W ten sposób możliwe staje się zmienianie tekstu w zasobie bez potrzeby ingerowania w kod źródłowy.

Istnieje bardzo wiele różnorodnych rodzajów zasobów w Androidzie. W tym rozdziale postaramy się omówić większość z nich. Rozpoczniemy od przedyskutowania bardzo powszechnego rodzaju zasobów: ciągu znaków.

## Zasoby typu string

Android umożliwia definiowanie ciągów znaków (ang. *string*) w co najmniej jednym pliku zasobów XML. Pliki te, zawierające definicje zasobów typu *string*, są umieszczone w podkatalogu */res/values*. Nazwy plików XML mogą być dowolne, jednak najczęściej spotykany będzie plik zatytułowany *strings.xml*. Listing 3.1 przedstawia przykład pliku zawierającego zasób typu *string*.

**Listing 3.1.** Przykładowy plik strings.xml

---

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Witaj</string>
    <string name="app_name">Witaj, nazwo aplikacji</string>
</resources>
```

---

**Uwaga!**

Zwróćmy uwagę, że w pewnych edycjach środowiska Eclipse węzeł *<resources>* musi zostać zdefiniowany pod specyfikacją *xmlns*. Wydaje się, że nie ma znaczenia, na co ta specyfikacja wskazuje; wystarczy, że istnieje. Poniższe dwie wariacje tej specyfikacji powinny działać bez zarzutu:

```
<resources xmlns="http://schemas.android.com/apk/res/android" >
lub
<resources xmlns="domyślna przestrzeń nazw" >
```

Po utworzeniu lub zaktualizowaniu takiego pliku narzędzie ADT automatycznie utworzy lub zaktualizuje klasę Java, umieszczoną w głównym pakiecie aplikacji nazwanym *R.java*, o unikalne identyfikatory dwóch widocznych na listingu ciągów znaków.

Zwróćmy uwagę na lokalizację pliku *R.java* w poniższym przykładzie. Utworzyliśmy wysokopoziomową strukturę katalogów dla projektu nazwanego *MyProject*:

```
\MyProject
  \src
    \com\mycompany\android\my-root-package
    \com\mycompany\android\my-root-package\another-package
  \gen
    \com\mycompany\android\my-root-package\R.java
  \assets
  \res
  \AndroidManifest.xml
... itd.
```

**Uwaga!**

Bez względu na liczbę plików zasobów istnieje tylko jeden plik *R.java*.

Plik *R.java* zaktualizowany o zasoby z listingu 3.1 zostałby wzbogacony o wpisy widoczne na listingu 3.2:

**Listing 3.2.** Przykładowa zawartość pliku R.java

```
package com.mycompany.android.my-root-package;
public final class R {
    ...inne wpisy w zależności od projektu i aplikacji

    public static final class string
    {
        ...inne wpisy w zależności od projektu i aplikacji

        public static final int hello=0x7f040000;
        public static final int app_name=0x7f040001;

        ...inne wpisy w zależności od projektu i aplikacji
    }
    ...inne wpisy w zależności od projektu i aplikacji
}
```

---

Przede wszystkim należy zwrócić uwagę, w jaki sposób zdefiniowano szczytową klasę głównego pakietu w pliku *R.java*: `public static final class R`. W tej zewnętrznej klasie R Android definiuje klasę wewnętrzną, dokładniej `static final class string`. Klasa ta służy plikowi *R.java* jako przestrzeń nazw do przechowywania identyfikatorów zasobów typu `string`.

Dwie klasy `static final ints`, określone nazwami zmiennych `hello` oraz `app_name`, są identyfikatorami zasobów reprezentującymi odpowiednie zasoby typu `string`. Można stosować te identyfikatory w dowolnym miejscu kodu źródłowego za pomocą następującej struktury:

`R.string.hello`

Należy zwrócić uwagę, że te wygenerowane identyfikatory wskazują typ danych `int`, a nie `string`. Większość metod korzystających z ciągów znaków uznaje także identyfikatory zasobów za dane wejściowe. W razie konieczności Android przekształci dane `int` w dane typu `string`.

Jest jedynie kwestią ustalonej konwencji, że większość przykładowych aplikacji zestawu Android SDK definiuje ciągi znaków w jednym pliku *strings.xml*. Android radzi sobie z dowolną liczbą takich plików, pod warunkiem że ich struktura wygląda tak, jak przedstawiono na listingu 3.1, oraz że znajdują się w podkatalogu */res/values*.

Można łatwo prześledzić strukturę takiego pliku. Obecny jest główny węzeł `<resources>`, pod którym umieszczone są podrzędne elementy `<string>`. Każdy element `<string>` lub węzeł posiada właściwość `name`, która staje się atrybutem `id` w pliku *R.java*.

Żeby się przekonać, co się dzieje z plikami zasobów typu `string` w tym podkatalogu, można w nim umieścić plik zawierający kod pokazany poniżej i nazwać go *strings1.xml* (listing 3.3):

**Listing 3.3.** Przykład dodatkowego pliku strings1.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello1">Witaj 1</string>
    <string name="app_name1">Witaj, nazwo aplikacji 1</string>
</resources>
```

---

W trakcie komplikacji narzędzie ADT zweryfikuje unikalność tych identyfikatorów i umieści je jako dwie dodatkowe stałe w pliku *R.java*: `R.string.hello1` oraz `R.string.app_name1`.

## Zasoby typu layout

W Androidzie wygląd ekranu często jest wczytywany z pliku XML w formie zasobu. Pliki te są nazywane zasobami typu layout (ang. *layout* — układ graficzny). **Zasoby typu layout** są kluczowymi elementami programowania interfejsu użytkownika w aplikacji tworzonej dla Androida. Spójrzmy na fragment kodu z przykładowej aktywności Androida, widoczny na listingu 3.4:

**Listing 3.4.** Stosowanie pliku układu graficznego

---

```
public class HelloWorldActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView tv = (TextView) this.findViewById(R.id.text1);
        tv.setText("Wpisz tu jakiś tekst");
    }
    ...
}
```

---

Wiersz `setContentView(R.layout.main)` wskazuje na istnienie statycznej klasy `R.layout`, w której znajduje się stała `main` (typ `integer`) odnosząca się do widoku zdefiniowanego przez plik XML stanowiący zasób układu graficznego. Plik ten nosi nazwę `main.xml`. Musi on zostać umieszczony w podkatalogu zasobów `layout`. Innymi słowy, powyższa instrukcja wymaga od programisty utworzenia pliku `/res/layout/main.xml` i umieszczenia w nim niezbędnych definicji dotyczących układu graficznego. Zawartość pliku `main.xml` może wyglądać tak jak na listingu 3.5.

**Listing 3.5.** Przykładowy plik układu graficznego main.xml

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
<Button android:id="@+id/b1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

---

Plik układu graficznego zaprezentowany na listingu 3.5 definiuje główny węzeł, nazwany `LinearLayout`, w którym umieszczony jest element `TextView`, a po nim `Button`. Węzeł ten rozmieszcza elementy podrzędne w pionie lub w poziomie — w tym przypadku w pionie.

Dla każdego ekranu (lub aktywności) trzeba definiować oddzielne pliki układu graficznego. Gwoli ściśleści, każdy układ graficzny wymaga oddzielnego pliku. W przypadku tworzenia dwóch ekranów prawdopodobnie potrzebne będą dwa pliki układu graficznego, na przykład */res/layout/screen1\_layout.xml* oraz */res/layout/screen2\_layout.xml*.

**Uwaga!**

Każdy plik umieszczony w podkatalogu */res/layout/* generuje unikatową stałą na podstawie nazwy tego pliku (rozszerzenie zostaje pominięte). W przypadku zasobów typu *layout* istotna jest liczba plików, w przypadku zasobów typu *string* ważna jest liczba — stanowiących zasoby — poszczególnych ciągów znaków, znajdujących się *wewnątrz* plików.

Jeżeli na przykład w podkatalogu */res/layout/* zostały utworzone dwa pliki o nazwach *file1.xml* oraz *file2.xml*, w pliku *R.java* pojawią się następujące wpisy (listing 3.6):

**Listing 3.6.** Kilka stałych dla różnych plików układu graficznego

```
public static final class layout {  
    ...jakies inne pliki  
    public static final int file1=0x7f030000;  
    public static final int file2=0x7f030001;  
}
```

Zdefiniowane w tych plikach widoki, na przykład *TextView* (listing 3.5), dostępne są w kodzie Java poprzez wygenerowane w pliku *R.java* identyfikatory ich zasobów:

```
TextView tv = (TextView)this.findViewById(R.id.text1);  
tv.setText("Wpisz tu jakiś tekst");
```

W tym przykładzie widok *TextView* lokalizowany jest za pomocą metody *findViewById* klasy *Activity*. Stała *R.id.text1* nawiązuje do identyfikatora zdefiniowanego dla widoku *TextView*. Identyfikator ten w pliku układu graficznego wygląda następująco:

```
<TextView android:id="@+id/text1"  
...  
</TextView>
```

Wartość atrybutu *id* wskazuje na to, że stała *text1* zostanie użyta do jednoznacznego rozpoznawania tego widoku wśród innych widoków obsługiwanych przez tę aktywność. Znak *+* w wyrażeniu *@+id/text1* oznacza, że identyfikator *text1* zostanie utworzony, w przypadku gdy jeszcze nie istnieje. Składnia identyfikatora zasobu jest bardziej skomplikowana. Zajmiemy się nią w następnym punkcie.

## Składnia odniesienia do zasobu

Bez względu na rodzaj (dotychczas omówiliśmy zasoby typu *string* oraz *layout*) wszystkie zasoby są identyfikowane (są tworzone do nich odniesienia) poprzez atrybut *id* kodu Java. Składnia stosowana do wiązania tego atrybutu z zasobem pliku XML jest określana mianem **składni odniesienia do zasobu**. Formalna struktura składni wspomnianego wyżej identyfikatora *@+id/text1* wygląda następująco:

```
@[package:]type/name
```

Składnik type odnosi się do jednej z przestrzeni nazw określonych dla zasobów, dostępnych w pliku *R.java*. Wśród nich znajdują się takie jak:

- R.drawable,
- R.id,
- R.layout,
- R.string,
- R.attr,
- R.plural,
- R.array.

Odpowiadającymi im typami w składni odniesienia do zasobów XML są odpowiednio:

- drawable,
- id,
- layout,
- string,
- attr,
- plurals,
- string-array.

Element name w składni @*[package:]type/name* to nazwa nadawana zasobowi (na przykład *text1* na listingu 3.5); jest ona reprezentowana również jako stała int w pliku *R.java*.

Jeżeli nie zostanie zdefiniowany żaden pakiet w składni @*[package:]type/name*, to para *type/name* zostanie przetworzona na podstawie lokalnych zasobów oraz lokalnego pakietu *R.java* aplikacji.

Jeżeli wpiszemy android:type/name, identyfikator odniesienia zostanie utworzony przy użyciu pakietu android, a dokładnie pliku *android.R.java*. W celu zlokalizowania odpowiedniego pliku *R.java* przetwarzającego odniesienie możemy w miejscu składnika package użyć nazwy dowolnego pakietu Java. Znając te informacje, przyjrzyjmy się przykładom. W trakcie przeglądania listingu 3.7 zwróćmy uwagę, że znajdujący się po lewej stronie człon identyfikatora (*android:id*) nie jest częścią składni. Stanowi on jedynie sposób przydzielania identyfikatora do kontrolki takiej jak *TextView*.

---

#### **Listing 3.7.** Analiza składni odniesień do zasobów

---

```
<TextView android:id="text">
// Błąd komplikacji, gdyż identyfikator nie przyjmie nieprzetworzonych ciągów znaków
// tekstowych.
```

```
<TextView android:id="@text">
// Niewłaściwa składnia. W członie @text brakuje nazwy typu.
// Składnia powinna wyglądać następująco: @id/text/ @+id/text lub @string/string1.
// Zostanie wyświetlony błąd „No Resource type specified” (nie określono typu zasobu).
```

```
<TextView android:id="@id/text">
// Błąd: nie został odnaleziony zasób odpowiadający identyfikatorowi „text”,
// chyba że przedtem „text” został zdefiniowany jako identyfikator.
```

```
<TextView android:id="@+id/text">
// Błąd: zasób nie jest publiczny.
// Wskazuje na to, że nie ma takiego identyfikatora w obiekcie android.R.id.
// Oczywiście byłoby to poprawne, gdyby to plik R.java pakietu android definiował
// identyfikator z tą nazwą.

<TextView android:id="@+id/text">
// Sukces: zostaje utworzony identyfikator o nazwie „text” w lokalnym pakiecie R.java.
```

---

W składni @+id/text symbol + posiada specjalne znaczenie. System zostaje w ten sposób poinformowany, że identyfikator text może jeszcze nie istnieć — w takim przypadku należy go utworzyć oraz nadać nazwę text.

## Definiowanie własnych identyfikatorów zasobów do późniejszego użytku

Dwoma głównymi mechanizmami przydzielania atrybutu id są wygenerowanie nowego identyfikatora lub wykorzystanie już utworzonego przez pakiet Android. Możliwe jest jednak również własnoręczne określanie identyfikatorów i późniejsze ich wykorzystanie we własnych pakietach.

Omówiony uprzednio wiersz <TextView android:id="@+id/text"> wskazuje na to, że atrybut id nazwany text będzie używany, jeżeli jest już utworzony. Jeżeli ten atrybut nie istnieje, zostanie utworzony. Zatem kiedy atrybut id, taki jak wspomniany text, który już istnieje w pliku *R.java*, może zostać ponownie wykorzystany?

Możliwe, że ktoś chciałby wstawić taką stałą jak *R.id.text* do pliku *R.java*, lecz plik ten nie jest modyfikowalny. Nawet gdyby istniała taka możliwość, jest on odtwarzany za każdym razem, gdy coś zostaje zmienione, dodane lub usunięte w podkatalogu */res/\**.

Rozwiązaniem jest użycie znacznika zasobów *item* do zdefiniowania atrybutu id bez dołączania konkretnego zasobu. Na listingu 3.8 został pokazany przykład:

**Listing 3.8.** Predefiniowanie identyfikatora

---

```
<resources>
<item type="id" name="text"/>
</resources>
```

---

Element *type* dotyczy rodzaju zasobu — w tym przypadku atrybutu id. Kiedy ten atrybut znajduje się na miejscu, powinna działać definicja widoku, widoczna na listingu 3.9:

**Listing 3.9.** Wykorzystywanie predefiniowanego identyfikatora

---

```
<TextView android:id="@+id/text">
...
</TextView>
```

---

## Skompilowane oraz nieskompilowane zasoby Androida

Android obsługuje zasoby głównie za pomocą dwóch kategorii plików: *plików XML* oraz *plików nieskompresowanych* (na przykład obrazy, pliki audio i wideo). Pokazaliśmy wcześniej, że nawet w przypadku plików XML zasoby są czasami definiowane jako wartości wewnętrz tych plików (na przykład ciągi znaków), a czasami cały plik XML może być zasobem (przytoczyć można przykład zasobu typu *layout*).

Pliki XML można podzielić na kolejne dwie kategorie: część z nich zostaje skompilowana do formatu binarnego, inne natomiast zostają skopiowane na urządzenie w niezmienionej postaci. Przykłady, z którymi mieliśmy do czynienia — pliki XML zawierające zasoby typu *string* oraz pliki XML zawierające zasoby typu *layout* — zostają skompilowane do formatu binarnego przed dołączeniem do pakietu instalacyjnego. Pliki te posiadają predefiniowane formaty, zgodnie z którymi węzły są przekształcane w identyfikatory.

Można także wskazać, że niektóre pliki XML nie będą posiadały ściśle ustalonego formatu struktury; nie będą one interpretowane ani przekształcane w identyfikatory zasobów. Może jednak ciągle istnieć potrzeba skompilowania ich do formatu binarnego, co jest równoznaczne z uzyskaniem komfortu lokalizacji. W celu uzyskania tego efektu można umieścić te pliki w podkatalogu */res/xml/*, co spowoduje ich skompilowanie do formatu binarnego. W takim przypadku do odczytywania węzłów XML należy używać czytników XML Androida.

Jeżeli jednak pliki (w tym pliki XML) zostaną umieszczone w katalogu */res/raw/*, nie zostaną przekształcone do formatu binarnego. Konieczne są w tym przypadku jawne interfejsy API oparte na technologii przesyłania strumieniowego w celu obsłużenia odczytu tych plików. Do tej kategorii należą pliki audio i wideo.

**Uwaga!**

Warto zauważyć, że dzięki temu, iż katalog *raw* jest podkatalogiem katalogu */res/\**, nawet te nieskompilowane pliki audio i wideo mogą korzystać z zalet lokalizacji w taki sam sposób jak inne rodzaje zasobów.

Jak zostało wspomniane w tabeli 2.1 (rozdział 2.), pliki zasobów są przechowywane w różnych podkatalogach, w zależności od ich typów. Poniżej wypisaliśmy kilka istotnych podkatalogów węzła *res* wraz z rodzajami przechowywanych w nich zasobów:

- **anim** — skompilowane pliki animacji;
- **drawable** — mapy bitowe;
- **layout** — definicje widoku bądź interfejsu UI;
- **values** — tablice, kolory, wymiary, ciągi znaków oraz style;
- **xml** — skompilowane własne pliki XML;
- **raw** — nieskompilowane nieskompresowane pliki.

Kompilator zasobów w narzędziu AAPT kompiluje wszystkie zasoby poza znajdującymi się w katalogu *raw* i umieszcza je w końcowym pliku *.apk*. Plik ten zawiera kod i zasoby aplikacji. Jest powiązany z plikiem *.jar* środowiska Java (skrót „apk” rozwija się jako „Android Package”, czyli „pakiet systemu Android”). To właśnie plik *.apk* jest instalowany w urządzeniu.

**Uwaga!**

Chociaż analizator składni zasobów XML umożliwia nazwanie zasobu na przykład `hello-string`, pojawi się błąd komplikacji pliku `R.java`. Można tego uniknąć, zmieniając nazwę zasobu na `hello_string` (myślnik zostaje zastąpiony podkreśnikiem).

## Rodzaje głównych zasobów w Androidzie

Po przedstawieniu podstawowych informacji dotyczących zasobów zajmiemy się wyliczeniem części pozostałych zasobów obsługiwanych przez Androida, ich reprezentacji w języku XML oraz sposobem ich wykorzystywania w kodzie Java (można używać tego podrozdziału jako skróconej instrukcji obsługi podczas tworzenia plików zasobów dla poszczególnych rodzajów zasobów). Zaczniemy od szybkiego przejrzenia rodzajów zasobów oraz ich funkcji (tabela 3.1).

**Tabela 3.1.** Rodzaje zasobów

Typ zasobu	Lokalizacja	Opis
Kolory	/res/values/any-file	Reprezentuje identyfikatory kolorów, wskazujące na kody kolorów. Identyfikatory tych zasobów są umieszczone w pliku <code>R.java</code> jako <code>R.color.*</code> . Węzłem XML w pliku jest <code>/resources/color</code> .
Ciągi znaków	/res/values/any-file	Reprezentuje zasoby typu <code>string</code> . Dzięki tym zasobom istnieje możliwość korzystania, poza prostymi ciągami znakowymi, z ciągów znaków sformatowanych w środowisku Java oraz ze znaczników nieprzetworzonego języka HTML. Identyfikatory tych zasobów są umieszczone w pliku <code>R.java</code> jako <code>R.string.*</code> . Węzłem XML w pliku jest <code>/resources/string</code> .
Tablice ciągów znaków	/res/values/any-file	Reprezentuje zasoby składające się z tablic ciągów znaków. Identyfikatory tych zasobów są zdefiniowane w pliku <code>R.java</code> jako <code>R.array.*</code> . Węzeł XML w pliku wygląda następująco: <code>/resources/string-array</code> .
Wielokrotności	/res/values/any-file	Reprezentuje zbiór kilku ciągów znaków, z których każdy jest odpowiedni dla wartości odpowiadającej liczbie na przykład jakichś elementów. Chodzi o to, że w poszczególnych językach sposób zapisania zdania zależy od liczby elementów, o których mowa w tym zdaniu — zdania mówiące o jednym, o kilku, o wielu albo o ani jednym elemencie brzmią różnie. Identyfikator zasobu jest widoczny w pliku <code>R.java</code> jako <code>R.plural.*</code> . Węzeł w pliku to <code>/resources/plurals</code> .
Wymiary	/res/values/any-file	Reprezentuje wymiary lub rozmiary różnych elementów oraz widoków w Androidzie. Obsługuje piksele, cale, milimetry, piksele niezależne od gęstości oraz piksele niezależne od skali. Identyfikatory tych zasobów są umieszczone w pliku <code>R.java</code> jako <code>R.dimens.*</code> . Węzłem XML w pliku jest <code>/resources/dimens</code> .

**Tabela 3.1.** Rodzaje zasobów — ciąg dalszy

Typ zasobu	Lokalizacja	Opis
Obrazy	/res/drawable/ multiple-files	Reprezentuje zasoby obrazów. Obsługiwany typami są pliki .jpg, .gif, .png itd. Każdy obraz znajduje się w oddzielnym pliku i otrzymuje własny identyfikator oparty na nazwie tego pliku. Identyfikatory tych zasobów są umieszczone w pliku R.java jako R.drawable.*. Obsługiwane są także tak zwane obrazy rozciągalne, w których część obrazu ulega rozciągnięciu, podczas gdy pozostałe jego fragmenty nie ulegają zmianie. Taki rozciągalny obraz jest znany również jako plik 9-patch (.9.png).
Kolorowe obiekty rysowane	/res/values/any-file także /res/drawable/ multiple-files	Reprezentuje prostokąty kolorów, które mogą być używane jako tło widoków lub inne zwykłe elementy rysowane (ang. <i>drawable</i> ), takie jak mapy bitowe. Można stosować ten rodzaj zasobu, zamiast wybierać na tło jednokolorową mapę bitową. W języku Java odpowiednikiem jest utworzenie kolorowego prostokąta i skonfigurowanie go jako tła widoku. Służy do tego znacznik wartości <drawable>. Identyfikatory tych zasobów są umieszczone w pliku R.java jako R.drawable.*. Węzłem w pliku XML jest /resources/drawable. Android obsługuje także zaokrąglone prostokąty oraz prostokąty o wypełnieniu w formie gradientu kolorów poprzez pliki XML umieszczone w podkatalogu /res/drawable. Pliki takie posiadają główny znacznik XML <shape>. Te zasoby również są zamieszczone w pliku R.java jako R.drawable.*. Nazwa każdego pliku w tym przypadku jest tłumaczona na unikatowy identyfikator.
Własne pliki XML	/res/xml/*.xml	Android dopuszcza własne pliki XML jako zasoby. Pliki te są przetwarzane przez kompilator AAPT. Identyfikatory tych zasobów są dostępne w pliku R.java jako R.xml.*.
Własne, nieskompresow ane zasoby	/res/raw/*.*	Android dopuszcza w tym katalogu posiadanie własnych, nieskompilowanych plików binarnych lub tekstowych. Każdy plik otrzymuje specyficzny identyfikator zasobu. Te pliki zasobów są eksponowane w pliku R.java jako R.raw.*.
Własne, nieskompresow ane pliki dodatkowe	/assets/*.*/*.*	Android dopuszcza posiadanie własnych plików we własnych podkatalogach, mieszczących się w podkatalogu /assets. Nie są to faktyczne zasoby, lecz nieskompresowane pliki. W przeciwieństwie do pozostałych podkatalogów węzła /res istnieje możliwość tworzenia tu własnego drzewa katalogowego. Pliki te nie generują identyfikatorów zasobów. Trzeba podać relatywną nazwę ścieżki, począwszy od podkatalogu /assets — bez uwzględniania jego nazwy w ścieżce.

Na następnych stronach każdy z wymienionych w powyższej tabeli zasobów zostanie dokładniej omówiony oraz zaprezentowany we fragmentach kodu XML oraz Java.

**Uwaga!**

Jeśli przyjrzeć się sposobowi tworzenia identyfikatorów zasobów, to wygląda na to — chociaż nigdzie oficjalnie tego nie napisano — że są generowane na podstawie nazwy pliku, pod warunkiem że te pliki XML znajdują się w którymkolwiek miejscu podkatalogu *res/values*. Jeżeli tam się znajdują, sprawdzana jest jedynie zawartość pliku, by dowiedzieć się, czy nadaje się do utworzenia identyfikatorów.

## Tablice ciągów znaków

Istnieje możliwość zdefiniowania ciągów znaków w postaci tablicy jako zasób w każdym pliku umieszczonym w podkatalogu */res/values*. Wykorzystywany jest w tym celu węzeł XML nazwany *string-array*. Jest to węzeł potomny, którego rodzicem jest *resources*, podobnie jak ma to miejsce w przypadku węzła *string*. Listing 3.10 prezentuje przykład definiowania tablicy w pliku zasobów.

**Listing 3.10.** Definiowanie tablicy ciągów znaków

---

```
<resources ...>
....Inne zasoby
<string-array name="test_array">
    <item>raz</item>
    <item>dwa</item>
    <item>trzy</item>
</string-array>
....Inne zasoby
</resources>
```

---

Po zdefiniowaniu takiego zasobu tablicy ciągów znaków możemy pobrać tę tablicę w kodzie Java, co zostało pokazane na listingu 3.11.

**Listing 3.11.** Odczytywanie tablicy ciągów znaków w kodzie Java

---

```
//Uzyskuje dostęp do obiektu zasobów z poziomu aktywności
Resources res = your-activity.getResources();
String strings[] = res.getStringArray(R.array.test_array);

//Wyświetla ciągi znaków
for (String s: strings)
{
    Log.d("example", s);
}
```

---

## Wielokrotności

Zasób *plurals* składa się ze zbioru ciągów znaków. Te ciągi znaków stanowią różnorodne sposoby numerycznego określenia liczby jakichś elementów, na przykład jajek w gnieździe. Rozważmy poniższy przykład:

Jest 1 jajko.

Są 2 jajka.  
Jest 0 jajek.  
Jest 100 jajek.

Zwróćmy uwagę, że zdania są identyczne dla liczb 0 i 100, jednak wyglądają inaczej w przypadku liczb 1 i 2. Taka odmienność zapisu zdań może zostać odwzorowana za pomocą zasobu plurals. Na listingu 3.12 widzimy, w jaki sposób można wewnątrz pliku zasobu zaprezentować te trzy odmiany zdania na podstawie liczby elementów.

#### **Listing 3.12.** Definiowanie wielokrotności w pliku zasobów

---

```
<resources...>
<plurals name="eggs_in_a_nest_text">
    <item quantity="one">Jest 1 jajko.</item>
    <item quantity="few">Są %d jajka.</item>
    <item quantity="other">Jest %d jajek.</item>
</plurals>
</resources>
```

---

Zauważmy, w jaki sposób te trzy odmiany zostały zdefiniowane jako pięć elementów jednej wielokrotności. Teraz możemy wykorzystać pokazany na listingu 3.13 kod Java do wyświetlenia ciągu znaków odnoszącego się do liczby jakichś elementów, o których mowa w zdaniu. Pierwszym parametrem metody `getQuantityString()` jest identyfikator zasobu wielokrotności. Za pomocą drugiego parametru wybieramy potrzebny ciąg znaków. Jeżeli wartość liczby elementów wynosi 1, 2, 3 lub 4, nie modyfikujemy ciągu znaków w żaden sposób. Jeżeli ta wartość będzie inna, należy wprowadzić trzeci parametr, którego wartość będzie zastępowała zmienną `%d`. Za każdym razem, gdy będziemy chcieli formatować ciągi znaków w zasobie wielokrotności, wymagane będą przynajmniej te trzy parametry.

#### **Listing 3.13.** Wyświetlanie ciągów znaków zawartych w zasobie wielokrotności

---

```
Resources res = your-activity.getResources();
String s1 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 0,0);
String s2 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 1,1);
String s3 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 2,2);
String s4 = res.getQuantityString(R.plurals.eggs_in_a_nest_text, 10,10);
```

---

Dzięki temu fragmentowi kodu podanie dowolnej wartości jako liczby elementów spowoduje wyświetlenie odpowiedniego ciągu znaków, stanowiącego zdanie we właściwej formie gramatycznej.

Czy istnieją jednak jakieś inne zastosowania atrybutu `quantity` występującego w węźle `item`? Żeby zrozumieć zastosowanie tych zasobów, zalecamy przejrzenie kodów źródłowych plików `Resources.java` i `PluralRules.java`, które są dostępne w kodzie systemu Android. Wśród zamieszczonych na końcu rozdziału odnośników można znaleźć odniesienia do wyciągów z tych plików źródłowych.

---

<sup>1</sup> Wartość `few` odnosi się do gramatyki języka polskiego, gdzie oddzielną odmianę uzyskują zdania zawierające cyfry 2, 3, 4 oraz wszelkie liczby, które kończą się cyframi 2, 3, 4 (za wyjątkiem cyfr 12, 13, 14) — przyp. tłum.

Podsumowując, w języku angielskim mamy do czynienia tylko z wartościami „jeden” i „wiele”, podczas gdy w języku polskim (i czeskim) dochodzi jeszcze wartość „kilka” (reprezentująca przedział liczb 2 – 4).

## Dodatkowe informacje na temat zasobów typu string

Na początku tego rozdziału omówiliśmy skrótnie zasoby ciągów znaków. Przyjrzymy się im teraz dokładniej, przeanalizujemy także ciągi znaków języka HTML oraz metody podstawiania zmiennych w zasobach typu `string`.

### Uwaga!

Większość struktur interfejsu użytkownika umożliwia korzystanie z zasobów typu `string`. Jednak w przeciwieństwie do innych szkieletów interfejsu UI, Android pozwala na szybkie powiązanie identyfikatorów z zasobami ciągu znaków poprzez plik `R.java`. Zatem stosowanie ciągów znaków w formie zasobów ułatwia pracę.

Rozpoczniemy od definiowania w pliku zasobów XML zwykłych ciągów znaków, cytowanych ciągów znaków, ciągów znaków formatowanych znacznikami HTML oraz podstawialnych ciągów znaków (listing 3.14).

**Listing 3.14.** Składnia języka XML stosowana do definiowania zasobów typu `string`

---

```
<resources>
    <string name="simple_string">Prosty ciąg znaków</string>
    <string name="quoted_string">"cytowany ciąg znaków 'xyz'"</string>
    <string name="double_quoted_string">\"cudzysłów\"</string>
    <string name="java_format_string">
        Witaj %2$s formatowanie java. %1$s ponownie
    </string>
    <string name="tagged_string">
        Witaj <b><i>ukończy Androidzie</i></b>, jesteś pogrubiony.
    </string>
</resources>
```

---

Plik XML zasobów typu `string` musi zostać umieszczony w podkatalogu `/res/values`. Nazwa pliku nie ma znaczenia.

Należy zauważać, że cytowane ciągi znaków muszą zostać wstawione pomiędzy znaki cytowania lub zacytowane w alternatywny sposób. Definicje typu `string` pozwalają także na stosowanie standardowych sekwencji formatowania w języku Java.

Android dopuszcza również stosowanie wewnętrz węzła XML `<string>` takich elementów, jak `<b>` czy `<i>`, oraz innych prostych znaczników formatowania tekstu w języku HTML. Można utworzyć taki złożony ciąg znaków do sformatowania tekstu przed jego wstawieniem do widoku tekstu.

Każda z tych metod została zaprezentowana na listingu 3.15.

**Listing 3.15.** Stosowanie zasobów typu `string` w kodzie Java

---

```
//Odczytuje prosty ciąg znaków i wstawia go do widoku tekstu
String simpleString = activity.getString(R.string.simple_string);
textView.setText(simpleString);
```

```
//Odczytuje ciąg znaków i wstawia go do widoku tekstu
String quotedString = activity.getString(R.string.quoted_string);
textView.setText(quotedString);

//Odczytuje ciąg znaków w cudzysłowie i wstawia go do widoku tekstu
String doubleQuotedString = activity.getString(R.string.double_quoted_string);
textView.setText(doubleQuotedString);

//Odczytuje ciąg znaków sformatowany w języku Java
String javaFormatString = activity.getString(R.string.java_format_string);
//Konwertuje sformatowany ciąg znaków poprzez przeniesienie argumentów
String substitutedString = String.format(javaFormatString, "Hello" , "Android");
//Umieszcza dane wyjściowe w widoku tekstu
textView.setText(substitutedString);

//Odczytuje z zasobu ciąg znaków sformatowany w języku HTML i umieszcza go w widoku
// tekstu
String htmlTaggedString = activity.getString(R.string.tagged_string);
// Konwertuje go do postaci ciągu tekstowego nadającego się do umieszczenia w widoku
// tekstu
// Klasa android.text.Html umożliwia rysowanie ciągów znaków w kodzie „html”
// Jest to klasa ściśle zdefiniowana przez Android i nie obsługuje wszystkich znaczników
// html
Spanned textSpan = android.text.Html.fromHtml(htmlTaggedString);
// Umieszcza tekst w widoku tekstu
textView.setText(textSpan);
```

---

Po zdefiniowaniu ciągów znaków jako zasobu można umieścić go bezpośrednio w takim widoku, jak `TextView`, w definicji układu graficznego XML tego widoku. Na listingu 3.16 został pokazany przykład, w którym ciąg znaków sformatowany za pomocą znaczników HTML jest skonfigurowany jako zawartość tekstowa widoku `TextView`.

---

**Listing 3.16.** Stosowanie zasobów typu `string` w języku XML

---

```
<TextView android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:textAlign="center"
          android:text="@string/tagged_string"/>
```

---

Widok `TextView` natychmiast rozpoznaje formatowanie HTML i odpowiednio je przetwarza, co jest bardzo przydatne, gdyż w ten sposób można szybko formatować w widokach atrakcyjnie wyglądający tekst jako część układu graficznego.

## Zasoby typu Color

Podobnie jak w przypadku zasobów typu `string`, można stosować identyfikatory odniesienia również do wskazywania kolorów w sposób pośredni. Dzięki temu istnieje możliwość umieszczenia kolorów i tworzenia kompozycji graficznych. Po zdefiniowaniu kolorów oraz przypisaniu im identyfikatorów w plikach zasobów stają się one dostępne w kodzie Java poprzez te identyfikatory. Analogicznie jak w przypadku identyfikatorów zasobów typu `string`, które są dostępne w przestrzeni nazw `<pakiet>.R.string`, identyfikatory kolorów znajdują się w przestrzeni nazw `<pakiet>.R.color`.

Android definiuje również podstawowy zestaw kolorów we własnych plikach zasobów. Ich identyfikatory dostępne są w przestrzeni nazw `android.R.color`. Lista stałych kolorów dostępnych w przestrzeni nazw `android.R.color` została umieszczona pod następującym adresem (w języku angielskim):

<http://developer.android.com/reference/android/R.color.html>

Listing 3.17 prezentuje przykłady określania koloru w pliku zasobów XML.

#### **Listing 3.17.** Składnia języka XML do definiowania zasobów typu Color

---

```
<resources>
    <color name="red">#f00</color>
    <color name="blue">#0000ff</color>
    <color name="green">#f0f0</color>
    <color name="main_back_ground_color">#fffff00</color>
</resources>
```

---

Wpisy przedstawione na listingu 3.17 muszą się znajdować w pliku umieszczonym w podkatalogu `/res/values`. Nazwa pliku może być dowolna. Android odczyta wszystkie pliki, a następnie je przetworzy oraz odszuka oddzielne węzły, takie jak `<resources>` oraz `<color>`, w celu określenia identyfikatorów.

Na listingu 3.18 został pokazany sposób zastosowania zasobu typu `color` w kodzie Java.

#### **Listing 3.18.** Zasoby typu color w kodzie Java

---

```
int mainBackGroundColor
    = activity.getResources().getColor(R.color.main_back_ground_color);
```

---

Z kolei na listingu 3.19 zaprezentowano przykład wykorzystania zasobu typu `color` w definicji widoku.

#### **Listing 3.19.** Zastosowanie kolorów w definicji widoku

---

```
<TextView android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:textColor="@color/ red"
          android:text="Przykładowy tekst napisany czerwoną czcionką." />
```

---

## **Zasoby typu dimension**

Przykładami wymiarów wykorzystywanych podczas tworzenia układu graficznego w języku XML lub Java są piksele,cale oraz punkty. Można stosować zasoby wymiarów do tworzenia stylów oraz rozmieszczania elementów interfejsu użytkownika bez konieczności ingerencji w kod źródłowy aplikacji.

Na listingu 3.20 przedstawiono sposób korzystania z zasobów typu `dimension` w języku XML.

**Listing 3.20.** Składnia języka XML służąca do definiowania zasobów typu dimension

```
<resources>
    <dimen name="mysize_in_pixels">1px</dimen>
    <dimen name="mysize_in_dp">5dp</dimen>
    <dimen name="medium_size">100sp</dimen>
</resources>
```

---

Wymiary można definiować w następujących jednostkach:

- px — piksele,
- in — cale,
- mm — milimetry,
- pt — punkty,
- dp — piksele niezależne od gęstości na podstawie wartości 160 dpi (liczba pikseli na cal) ekranu (wymiary zostają dopasowane do upakowania pikseli na ekranie),
- sp — piksele niezależne od skali (wymiary umożliwiające użytkownikowi powiększanie/pomniejszanie; szczególnie przydatne w przypadku czcionek).

W celu uzyskania wymiaru w języku Java należy uzyskać dostęp do wystąpienia obiektu klasy Resources. Osiągamy to poprzez wywołanie funkcji getResources w obiekcie activity (listing 3.21). Po otrzymaniu obiektu klasy Resources można go użyć do wyszukania zasobu typu dimension, korzystając z identyfikatora tego zasobu (ponownie listing 3.21).

**Listing 3.21.** Stosowanie zasobów typu dimension w kodzie Java

```
float dimen = activity.getResources().getDimension(R.dimen.mysize_in_pixels);
```

---

**Uwaga!**

W metodzie tej stosowana jest pełna nazwa *Dimension*, podczas gdy w przestrzeni nazw pliku *R.java* używana jest skrócona forma *dimen*.

Podobnie jak w języku Java, wobec odniesienia do zasobu w środowisku XML stosuje się nazwę *dimen* zamiast pełnej nazwy *dimension* (listing 3.22).

**Listing 3.22.** Używanie zasobów typu dimension w kodzie XML

```
<TextView android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:textSize="@dimen/medium_size"/>
```

---

## Zasoby typu image

Android tworzy identyfikatory zasobów dla plików obrazów umieszczonych w podkatalogu */res/drawable*. Obsługiwany rozszerzeniami plików są *.gif*, *.jpg* oraz *.png*. Identyfikator każdego umieszczonego w tym podkatalogu pliku obrazu jest tworzony na podstawie jego nazwy. Jeżeli na przykład plik nosi nazwę *sample\_image.jpg*, identyfikatorem tego zasobu będzie *R.drawable.sample\_image*.

**Ostrzeżenie**

Jeżeli dwa pliki będą miały takie same nazwy, system wyświetli komunikat o błędzie. Poza tym podkatalogi umieszczone w węźle `/res/drawable` będą ignorowane. Żaden plik umieszczony w takim podkatalogu nie będzie odczytywany.

W definicjach układu graficznego pisanych w kodzie XML można tworzyć odniesienia do obrazów znajdujących się w podkatalogu `/res/drawable`, jak zostało to zaprezentowane na listingu 3.23.

**Listing 3.23.** Stosowanie zasobów obrazów w języku XML

---

```
<Button
    android:id="@+id/button1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Dzwoni"
    android:background="@drawable/sample_image"
/>
```

---

Można także uzyskać dostęp do obrazu programowo — za pomocą kodu Java — i określić go jako obiekt interfejsu użytkownika, na przykład jako przycisk (listing 3.24).

**Listing 3.24.** Używanie zasobów typu image w środowisku Java

---

```
// Wywołuje funkcję getDrawable, żeby pobrała obraz
BitmapDrawable d = activity.getResources().getDrawable(R.drawable.sample_image);

// Można teraz użyć obiektu rysowanego do ustawienia tła
button.setBackgroundDrawable(d);

// lub można wyznaczyć tło bezpośrednio poprzez identyfikator zasobu
button.setBackgroundResource(R.drawable.sample_image);
```

---

**Uwaga!**

Przedstawione powyżej techniki ustawiania tła odwołują się wstecz aż do nadrzędnej klasy `View`. W wyniku tego większość elementów interfejsu użytkownika będzie korzystała z tego tła.

Android obsługuje również specjalny format obrazu, zwany obrazem **rozciągalnym**. Jest to rodzaj obrazu `.png`, w którym można definiować fragmenty obrazu jako statyczne lub rozciągalne. Do określania tych rejonów służy narzędzie Draw9-patch, znajdujące się w pakiecie Android SDK (więcej informacji na jego temat — w języku angielskim — można znaleźć na stronie <http://developer.android.com/guide/developing/tools/draw9patch.html>).

Po przygotowaniu obrazu `.png` może być on używany tak samo jak każdy inny typ obrazu. Ten szczególny typ przydaje się jako tło dla przycisków, które ulegają rozciągnięciu w celu dopasowania do tekstu.

## Zasoby typu color-drawable

Rysunek w Androidzie jest jednym z rodzajów rysowanych zasobów. Drugim obsługiwany typem zasobu tego rodzaju jest tak zwany kolorowy obiekt rysowany (ang. *color-drawable*); w istocie jest to prostokąt wypełniony kolorem.

**Ostrzeżenie**

W dokumentacji Androida znajduje się stwierdzenie, że istnieje możliwość rysowania prostokątów o zaokrąglonych narożnikach. Nam się jednak nie udało tego dokonać. Zamiast tego przedstawiliśmy poniżej alternatywne rozwiązanie tego problemu. Dokumentacja sugeruje także, że tworzoną klasą Java jest `PaintDrawable`, jednak w wyniku działania kodu otrzymujemy nazwę `ColorDrawable`.

W celu zdefiniowania takiego prostokąta wypełnionego kolorem należy określić element środowiska XML poprzez nazwę węzła `drawable` w dowolnym pliku XML znajdującym się w podkatalogu `/res/values`. Na listingu 3.25 wymieniono kilka przykładów.

---

**Listing 3.25.** Składnia języka XML służąca do definiowania zasobów typu color-drawable

---

```
<resources>
    <drawable name="red_rectangle">#f00</drawable>
    <drawable name="blue_rectangle">#0000ff</drawable>
    <drawable name="green_rectangle">#f0f0</drawable>
</resources>
```

---

Listingi 3.26 oraz 3.27 przedstawiają kolejno zastosowanie zasobu typu `color-drawable` w języku Java oraz języku XML.

---

**Listing 3.26.** Zastosowanie zasobów typu color-drawable w kodzie Java

---

```
// Wczytuje obiekt rysowany
ColorDrawable redDrawable =
(ColorDrawable)
activity.getResources().getDrawable(R.drawable.red_rectangle);

// Ustawia ten obiekt jako tło widoku tekstu
textView.setBackgroundDrawable(redDrawable);
```

---

---

**Listing 3.27.** Korzystanie z zasobów typu color-drawable w kodzie XML

---

```
<TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textAlign="center"
        android:background="@drawable/red_rectangle"/>
```

---

W celu zaokrąglenia rogów obiektu `Drawable` można zastosować pominięty w dokumentacji znacznik `<shape>`. Musi być on jednak umieszczony w oddzielnym pliku, w katalogu `/res/drawable`. Sposób użycia znacznika `<shape>` umieszczonego w pliku `/res/drawable/my_rounded_rectangle.xml` został zaprezentowany na listingu 3.28.

**Listing 3.28.** Definiowanie prostokąta o zaokrąglonych rogach

---

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#f0600000"/>
    <stroke android:width="3dp" color="#ffff8080"/>
    <corners android:radius="13dp" />
    <padding android:left="10dp" android:top="10dp"
        android:right="10dp" android:bottom="10dp" />
</shape>
```

---

Można następnie wykorzystać ten zasób jako tło, tak jak we wcześniejszym przykładzie z widokiem tekstu, co zostało pokazane na listingu 3.29:

**Listing 3.29.** Wykorzystanie obiektu Drawable w kodzie Java

---

```
// Wczytuje obiekt rysowany
GradientDrawable roundedRectangle =
(GradientDrawable)
activity.getResources().getDrawable(R.drawable.my_rounded_rectangle);

// Ustawia ten obiekt jako tło widoku tekstu
textView.setBackgroundDrawable(roundedRectangle);
```

---

**Uwaga!**

Nie trzeba obsadzać bazowego zwracanego obiektu `Drawable` obiektem klasy `GradientDrawable`; zrobiliśmy to jednak, żeby pokazać przemianę znacznika `<shape>` w ten właśnie element. Jest to istotna informacja, ponieważ można sprawdzić w dokumentacji interfejsu API Java, jakie znaczniki języka XML ta klasa definiuje.

Na koniec warto wspomnieć, że obraz mapy bitowej, umieszczony w podkatalogu `drawable`, jest przetwarzany przez klasę `BitmapDrawable`. Wartość „rysowanego” zasobu, jak choćby prostokąta widocznego na listingu 3.29, jest przetwarzana przez klasę `ColorDrawable`. Plik XML zawierający znacznik `<shape>` jest przetwarzany przez klasę `GradientDrawable`.

## Praca na własnych plikach zasobów XML

Oprócz dotychczas omawianych zasobów strukturalnych, w Androidzie dopuszczalne jest również korzystanie z własnych plików XML jako zasobów. Dzięki temu takie pliki zyskują pewne zalety właściwe zasobom. Możliwe jest uzyskanie szybkiego odniesienia do tych plików w postaci wygenerowanych identyfikatorów zasobów. Poza tym pojawia się możliwość określenia lokalizacji tych plików. Można je również skutecznie kompilować i przechowywać w urządzeniu.

W taki sposób odczytywane pliki są umieszczone w podkatalogu `/res/xml`. Na listingu 3.30 został zaprezentowany przykład pliku `/res/xml/test.xml`:

**Listing 3.30.** Przykładowy plik XML

---

```
<rootelem1>
    <subelem1>
        Kod aplikacji Witaj, świecie! z podelementu xml
    </subelem1>
</rootelem1>
```

---

Tak jak w przypadku pozostałych plików zasobów XML Androida, narzędzie AAPT skompiluje ten plik, zanim umieści go w pakuie aplikacji. Do sprawdzenia składni zawartego w nim kodu potrzebne będzie wprowadzenie instancji `XmlPullParser`. Można wykorzystać instancję implementacji `XmlPullParser`, stosując poniższy fragment kodu (przykład z listingu 3.31 dotyczy kontekstu `activity`):

---

**Listing 3.31.** Odczytywanie pliku XML

---

```
Resources res = activity.getResources();
XmlResourceParser xpp = res.getXml(R.xml.test);
```

---

Zwracany obiekt `XmlResourceParser` jest instancją obiektu klasy `XmlPullParser` — dzięki niemu zostaje zaimplementowana również funkcja `java.util.AttributeSet`. Listing 3.32 przedstawia większy wycinek kodu odczytującego plik `test.xml`.

---

**Listing 3.32.** Zastosowanie wystąpienia `XmlPullParser`

---

```
private String getEventsFromAnXMLFile(Activity activity)
throws XmlPullParserException, IOException
{
    StringBuffer sb = new StringBuffer();
    Resources res = activity.getResources();
    XmlResourceParser xpp = res.getXml(R.xml.test);

    xpp.next();
    int eventType = xpp.getEventType();
    while (eventType != XmlPullParser.END_DOCUMENT)
    {
        if(eventType == XmlPullParser.START_DOCUMENT)
        {
            sb.append("*****Start document");
        }
        else if(eventType == XmlPullParser.START_TAG)
        {
            sb.append("\nStart tag "+xpp.getName());
        }
        else if(eventType == XmlPullParser.END_TAG)
        {
            sb.append("\nEnd tag "+xpp.getName());
        }
        else if(eventType == XmlPullParser.TEXT)
        {
            sb.append("\nText "+xpp.getText());
        }
        eventType = xpp.next();
    }//eof-while
    sb.append("\n*****End document");
    return sb.toString();
}//eof-function
```

---

Na listingu 3.32 można zobaczyć, w jaki sposób uzyskać obiekt `XmlPullParser` i wykorzystać go do nawigacji wśród elementów dokumentu XML oraz w jaki sposób zastosować dodatkowe metody tego obiektu do otrzymania szczegółowych informacji na temat tych elementów.

Jeżeli powyższy kod ma zadziałać, należy utworzyć wspomniany wcześniej plik XML i wywołać funkcję `getEventsFromAnXMLFile` z dowolnego elementu menu lub za pomocą kliknięcia przyciskiem myszy. Otrzymamy ciąg znaków, który można następnie skopiować do dziennika za pomocą metody Log.d.

## Praca na nieskompresowanych zasobach

Poza własnymi plikami XML można również używać nieskompresowanych typów plików. Zasoby te są umieszczone w podkatalogu `/res/raw` i stanowią pliki audio, video lub tekstowe, które wymagają zlokalizowania lub utworzenia odniesienia w formie identyfikatorów zasobów. W przeciwieństwie do zasobów znajdujących się w podkatalogu `/res/xml`, pliki te nie są komplikowane, lecz są przenoszone do pakietu aplikacji w niezmienionej postaci, jednak każdy z tych plików będzie posiadał wygenerowany identyfikator w pliku `R.java`. W przypadku umieszczenia pliku tekstuowego w podkatalogu `/res/raw/test.txt` kod potrzebny do jego odczytania będzie wyglądał tak jak na listingu 3.33.

**Listing 3.33.** Odczytywanie nieskompresowanego zasobu

---

```
String getStringFromRawFile(Activity activity)
    throws IOException
{
    Resources r = activity.getResources();
    InputStream is = r.openRawResource(R.raw.test);
    String myText = convertStreamToString(is);
    is.close();
    return myText;
}

String convertStreamToString(InputStream is)
throws IOException
{
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    int i = is.read();
    while (i != -1)
    {
        baos.write(i);
        i = is.read();
    }
    return baos.toString();
}
```

---

**Ostrzeżenie**

Pliki posiadające taką samą nazwę powodują wygenerowanie błędu komplikacji we wtyczce ADT. Tak samo jest w przypadku wszystkich identyfikatorów zasobów tworzonych na podstawie nazw plików.

## Praca z dodatkowymi plikami

Istnieje jeszcze jeden katalog, w którym można przechowywać pliki dołączane do pakietu aplikacji — `/assets`. Znajduje się on na tym samym poziomie co katalog `/res`, co oznacza, że nie jest jednym z podkatalogów tego węzła. Pliki znajdujące się w tym katalogu nie posiadają

identyfikatorów wygenerowanych przez plik *R.java*. Ich odczytanie wymaga podania ścieżki dostępu. Jest to relatywna ścieżka, rozpoczynająca się od katalogu */assets*. Aby uzyskać dostęp do tych plików, korzysta się z klasy *AssetManager*, co zostało przedstawione na listingu 3.34:

#### **Listing 3.34.** Odczytywanie dodatkowego pliku

---

```
// Uwaga: w kodzie nie zostały pokazane wyjątki
String getStringFromAssetFile(Activity activity)
{
    AssetManager am = activity.getAssets();
    InputStream is = am.open("test.txt");
    String s = convertStreamToString(is);
    is.close();
    return s;
}
```

---

## Przegląd struktury katalogów mieszczących zasoby

Jako podsumowanie na listingu 3.35 umieszczamy ogólną strukturę katalogową zasobów:

#### **Listing 3.35.** Katalogi zasobów

---

```
/res/values/strings.xml
    /colors.xml
    /dimens.xml
    /attrs.xml
    /styles.xml
/drawable/*.*.png
    /*.*.jpg
    /*.*.gif
    /*.*.9.png
/anim/*.*.xml
/layout/*.*.xml
/raw/*.*
/xml/*.*.xml
/assets/*.*/*.*
```

---

**Uwaga!**

Jedynie katalog */assets* może posiadać własne drzewo podkatalogów, gdyż nie jest podkatalogiem węzła */res*. Żaden inny podkatalog nie może mieć plików na niższym poziomie struktury. Jest to spowodowane sposobem, w jaki plik *R.java* generuje identyfikatory dla plików.

## Zasoby a zmiany konfiguracji

Zasoby okazują się bardzo pomocne w dostosowywaniu właściwości aplikacji w zależności od lokalizacji użytkownika. Można na przykład założyć, że w zależności od języka, którym posługuję się użytkownik (określonego jako region świata), będzie się zmieniać zawartość ciągu znaków. Zasoby systemu Android rozwijają tę ideę dla wszystkich możliwych opcji konfiguracyjnych urządzenia, nie tylko tych dotyczących języka. Innym przykładem zmiany

konfiguracji jest reakcja aplikacji na obrócenie urządzenia z pozycji pionowej do poziomej. Przypomnijmy, że orientacja pionowa jest znana jako tryb portretowy, a orientacja pozioma nosi nazwę trybu krajobrazowego.

System Android pozwala na wybór różnych zestawów układów graficznych związanych z identyfikatorem tego samego zasobu, w zależności od orientacji wyświetlacza. Można tego dokonać za pomocą oddzielnych katalogów dla każdej konfiguracji. Przykładowe katalogi zostały zaprezentowane na listingu 3.36.

#### **Listing 3.36.** Katalogi alternatywnych zasobów

```
\res\layout\main_layout.xml  
\res\layout-port\main_layout.xml  
\res\layout-land\main_layout.xml
```

Nawet jeśli utworzymy trzy oddzielne pliki układów graficznych i każdy umieścimy w osobnym katalogu, wszystkie wygenerują tylko jeden wspólny identyfikator układu graficznego w pliku *R.java*. Identyfikator ten będzie wyglądał następująco:

```
R.layout.main_layout
```

Jeżeli jednak wczytamy układ graficzny odpowiadający temu identyfikatorowi, uzyskamy układ optymalnie odpowiadający ułożeniu urządzenia.

W powyższym przykładzie rozszerzenia *-port* i *-land* katalogu noszą nazwę **kwalifikatorów konfiguracji**. Kwalifikatory te są **niezależne od wielkości liter** oraz są oddzielone myślnikiem (-) od nazwy katalogu zasobu. Zasoby definiowane w takich katalogach zawierających kwalifikatory konfiguracji są nazywane **alternatywnymi zasobami**. Z kolei obiekty znajdujące się w katalogu zasobów pozbawionym kwalifikatorów określane są jako **zasoby domyślne**.

Poniżej zamieszczono spis dostępnych kwalifikatorów konfiguracji.

- **mccAAA** — AAA jest kodem MCC (ang. *Mobile Country Code* — kod kraju w telefonii mobilnej),
- **mncAAA** — AAA jest kodem operatora/sieci,
- **pl-rPL** — język i region,
- **small, normal, large, xlarge** — rozmiar ekranu,
- **long, notlong** — typ ekranu,
- **port, land** — tryb portretowy lub krajobrazowy,
- **car, desk** — rodzaj dokowania,
- **night, notnight** — noc lub dzień,
- **ldpi, mdpi, hdpi, xhdpi, nodpi** — gęstość ekranu,
- **notouch, stylus, finger** — reakcja ekranu na dotyk,
- **keysexposed, keyssoft, keyshidden** — rodzaj klawiatury,
- **nokeys, qwerty, 12key** — liczba przycisków,
- **navexposed, navhidden** — przyciski nawigacji odsłonięte lub ukryte,
- **nonav, dpad, trackball, wheel** — rodzaj urządzenia sterującego,
- **v3, v4, v7** — poziom interfejsu API.

Za pomocą tych kwalifikatorów możemy utworzyć różnorakie katalogi zasobów, których kilka przykładów możemy zobaczyć na listingu 3.37.

#### **Listing 3.37.** Dodatkowe katalogi zawierające alternatywne zasoby

---

```
\res\layout-mcc312-mnc222-en-rUS  
\res\layout-ldpi  
\res\layout-hdpi  
\res\layout-car
```

---

Możemy sprawdzić aktualny region językowy poprzez uruchomienie aplikacji *Custom Locale*, dostępnej w emulatorze. Oto ścieżka dostępu: *Ekran startowy/lista aplikacji/Custom Locale*.

System Android, korzystając z danego identyfikatora, stosuje algorytm służący do wybrania właściwego zasobu. Aby lepiej zrozumieć reguły rządzące tym procesem, Czytelnik może przejrzeć odpowiednie adresy URL, zamieszczone w podrozdziale „Odnośniki”, postaramy się jednak już teraz wyjaśnić niektóre z nich.

Podstawowa zasada polega na tym, że kwalifikatory występujące na listingu 3.37 są przetwarzane w kolejności ich występowania. Spójrzmy na katalogi zamieszczone na listingu 3.38.

#### **Listing 3.38.** Różne odmiany plików układów graficznych

---

```
\res\layout\main_layout.xml  
\res\layout-port\main_layout.xml  
\res\layout-en\main_layout.xml
```

---

Na listingu 3.38 widać, że plik *main\_layout.xml* jest dostępny w dwóch dodatkowych wersjach: dla języka oraz dla orientacji wyświetlacza. Sprawdźmy teraz, którą wersję układu graficznego wybierze system, gdy urządzenie znajduje się w trybie portretowym. Nawet jeżeli ustawimy urządzenie w orientacji pionowej, Android wybierze układ graficzny z katalogu *layout-en*, ponieważ wśród kwalifikatorów konfiguracji wersja dotycząca języka ma wyższy priorytet od wersji związanej z ułożeniem ekranu. W podrozdziale „Odnośniki” umieściliśmy łącza do zasobów SDK, gdzie można znaleźć pełną listę kwalifikatorów konfiguracji oraz kolejność ich przetwarzania.

Przyjrzymy się dokładniej regułom pierwszeństwa, przeprowadzając eksperymenty na kilku zasobach ciągów znaków. Należy zwrócić uwagę, że ciągi znaków opierają się na pojedynczych identyfikatorach, podczas gdy zasoby układów graficznych są zależne od plików. W celu przetestowania kolejności przetwarzania kwalifikatorów konfiguracji wobec ciągów znaków stworzymy pięć identyfikatorów zasobów, które mogą występować w następujących konfiguracjach: *default*, *en*, *en\_us*, *port* oraz *en\_port*. Te identyfikatory to:

- **teststring\_all** — identyfikator ten znajdzie się we wszystkich odmianach katalogu *values*, włącznie z domyślną.
- **testport\_port** — ten identyfikator będzie obecny w konfiguracji domyślnej oraz w odmianie *-port*.
- **t1\_enport** — zostanie umieszczony w konfiguracji domyślnej oraz w odmianach *-en* i *-port*.

- t1\_1\_en\_port — znajdziemy go wyłącznie w konfiguracji domyślnej oraz odmianie -en-port.
- t2 — ten identyfikator będzie dostępny wyłącznie w konfiguracji domyślnej.

Na listingu 3.39 przedstawiono wszystkie odmiany katalogu *values*.

**Listing 3.39.** Odmiany ciągów znaków zależne od konfiguracji

---

```
// values(strings.xml
<resources xmlns="http://schemas.android.com/apk/res/android">
    <string name="teststring_all">teststring w katalogu głównym</string>
    <string name="testport_port">testport-port</string>
    <string name="t1_enport">t1 w katalogu głównym</string>
    <string name="t1_1_en_port">t1_1 w katalogu głównym</string>
    <string name="t2">t2 w katalogu głównym</string>
</resources>

// values-en(strings_en.xml
<resources xmlns="http://schemas.android.com/apk/res/android">
    <string name="teststring_all">teststring-en</string>
    <string name="t1_enport">t1_en</string>
    <string name="t1_1_en_port">t1_1_en</string>
</resources>

// values-en-rUS(strings_en_us.xml
<resources xmlns="http://schemas.android.com/apk/res/android">
    <string name="teststring_all">test-en-us</string>
</resources>

// values-port(strings_port.xml
<resources xmlns="http://schemas.android.com/apk/res/android">
    <string name="teststring_all">test-en-us-port</string>
    <string name="testport_port">testport-port</string>
    <string name="t1_enport">t1_port</string>
    <string name="t1_1_en_port">t1_1_port</string>
</resources>

// values-en-port(strings_en_port.xml
<resources xmlns="http://schemas.android.com/apk/res/android">
    <string name="teststring_all">test-en-port</string>
    <string name="t1_1_en_port">t1_1_en_port</string>
</resources>
```

---

Na listingu 3.40 widzimy plik *R.java* wygenerowany dla tych zasobów.

**Listing 3.40.** Plik R.java obsługujący różne odmiany konfiguracji ciągów znaków

---

```
public static final class string {
    public static final int teststring_all=0x7f050000;
    public static final int testport_port=0x7f050004;
    public static final int t1_enport=0x7f050001;
    public static final int t1_1_en_port=0x7f050002;
    public static final int t2=0x7f050003;
}
```

---

Od razu widać, że chociaż zdefiniowaliśmy mnóstwo ciągów znaków, system wygenerował zaledwie pięć identyfikatorów zasobów. Jeżeli wczytamy teraz wartości tych identyfikatorów, wynikiem będzie następujące zachowanie (testowaliśmy je w konfiguracji en\_US oraz w trybie portretowym):

- **teststring\_all** — ten identyfikator znajduje się we wszystkich pięciu odmianach katalogu *values*. Z tego powodu zostaje wybrana konfiguracja values-en-rUS. Zgodnie z regułami pierwszeństwa zdefiniowany język posiada priorytet ponad odmianami: domyślną, en, port oraz en-port.
- **testport\_port** — ten identyfikator znajdziemy wyłącznie w konfiguracji domyślnej oraz odmianie –port. Ponieważ nie znajduje się w żadnym katalogu posiadającym kwalifikator –en, wychodzi na to, że kwalifikator -port posiada pierwszeństwo przed konfiguracją domyślną i zostanie wybrana wartość z tej odmiany. Gdyby ten identyfikator został umieszczony w którymś z katalogów zawierających kwalifikator –en, to właśnie stamtąd pochodziłby odpowiedni ciąg znaków.
- **t1\_enport** — ten identyfikator jest obecny w trzech konfiguracjach: domyślnej, -en i –port. Ponieważ w tym samym czasie mamy do czynienia z odmianami –en i –port, zostanie wybrana wartość z tej pierwszej.
- **t1\_1\_en\_port** — ten identyfikator znajdziemy w czterech odmianach: domyślnej, -port, -en oraz –en-port. Ponieważ jest dostępny w konfiguracji –en-port, zostanie wybrana wartość właśnie stąd, a pozostałe odmiany zostaną zignorowane.
- **t2** — ten identyfikator jest dostępny jedynie w konfiguracji domyślnej, więc z niej zostanie pobrana wartość ciągu znaków.

Zestaw Android SDK uwzględnia bardziej złożony algorytm wybierania konfiguracji, warto się więc z nim zaznajomić. Jednak powyższy przykład powinien dać ogólne pojęcie na jego temat. Podstawą jest poznanie reguł pierwszeństwa jednych odmian konfiguracji nad innymi. W następnym podrozdziale znajduje się odpowiedni odnośnik do informacji o pakiecie SDK.

## Odbońniki

W trakcie poznawania tajników zasobów Androida mogą się przydać poniższe odnośniki; opisaliśmy, co można znaleźć po ich kliknięciu.

- <http://developer.android.com/guide/topics/resources/index.html> — ten adres URL stanowi mapę po dokumentacji dotyczącej zasobów.
- <http://developer.android.com/guide/topics/resources/available-resources.html> — znajdziemy tu opisane różnorodne rodzaje zasobów.
- <http://developer.android.com/reference/android/content/res/Resources.html> — umieszczono tutaj opis różnorodnych metod służących do odczytywania zasobów.
- <http://developer.android.com/reference/android/R.html> — opis zasobów zdefiniowanych w rdzeniu systemu Android.
- <http://www.androidbook.com/item/3542> — nasze badania dotyczące zasobów wielokrotności, tablic ciągów znaków oraz zasobów alternatywnych, jak również odniesienia do innych materiałów.
- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — z tego adresu możemy pobrać projekt środowiska Eclipse, w którym zostało ukazanych wiele koncepcji zawartych w tym rozdziale. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_Zasoby*.

## Podsumowanie

Podsumujmy ten rozdział poprzez wyliczenie opisanych tematów. Czytelnik mógł poznać rodzaje zasobów obsługiwanych przez Androida oraz metody ich tworzenia w plikach XML. Można się było dowiedzieć, w jaki sposób są tworzone identyfikatory zasobów oraz jak je umieścić w kodzie Java. Czytelnicy przekonali się także, że tworzenie identyfikatorów zasobów jest wygodną metodą, ułatwiającą zarządzanie zasobami w Androidzie. Poza tym można było zrozumieć, w jaki sposób należy pracować z zasobami nieskompresowanymi oraz dodatkowymi plikami. Poruszyliśmy również dość ogólnie zagadnienie alternatywnych zasobów, zasobów wielokrotności oraz tablic ciągów znaków.

Mając taką wiedzę, w następnym rozdziale można zająć się dostawcami treści.



# Dostawcy treści

Koncepcja **dostawców treści** w Androidzie oznacza abstrakcyjną warstwę łączącą usługom dostęp do danych. Dzięki dostawcom treści dostęp do źródeł danych jest podobny do takiego jak w przypadku architektury REST. Dobrym przykładem są strony WWW.

Witryna internetowa przekazuje do przeglądarki informacje na temat danych dostępnych pod określonym adresem URL. Podobnie dostawca treści zapewnia opis danych przekazywanych obsługiwanej aktywności. W tym znaczeniu dostawca treści służy jako osłona danych. Przykładem źródła danych, które można umieścić w dostawcy treści, jest baza danych SQLite.

## Uwaga!

Skrót REST oznacza *REpresentational State Transfer*, czyli reprezentacyjny transfer stanu. Jest to skomplikowana nazwa dla bardzo prostej koncepcji, z którą wszyscy (na przykład użytkownicy sieci WWW) są dobrze zaznajomieni. Kiedy ktoś wpisuje adres URL w przeglądarce i otrzymuje w odpowiedzi usługę sieciową (wyświetlenie strony), wykonuje operację „zapytania” wobec tej usługi. Zapytanie to jest oparte właśnie na architekturze REST. Innym przykładem jest wypełnienie formularza na stronie WWW. Przesłanie tego formularza do serwera może spowodować zmianę stanu tego serwera albo „zaktualizowanie” jego zawartości. Operacje te również są oparte na architekturze REST. Zazwyczaj koncepcji tej jest przeciwstawiane pojęcie usług sieciowych SOAP (ang. *Simple Object Access Protocol* — protokół wywoływanego zdalnego dostępu do obiektów). Więcej informacji na temat architektury REST można znaleźć na stronie Wikipedii [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer).

Aby odczytać dane zawarte w dostawcy treści lub je w nim zapisać, należy skorzystać z zestawu identyfikatorów URI, również zgodnych z założeniami architektury REST. Żeby na przykład odczytać zbiór tytułów książek znajdujących się w dostawcy treści, który stanowi „opakowanie” bazy danych o książkach, potrzebny byłby następujący identyfikator URI:

`content://com.android.book.BookProvider/books`

Aby odczytać dane wybranej książki (książkę numer 23), identyfikator URI musiałby wyglądać następująco:

`content://com.android.book.BookProvider/books/23`

W tym rozdziale pokażemy, w jaki sposób identyfikatory URI są powiązane z podstawowymi mechanizmami dostępu do bazy danych. Każda aplikacja zainstalowana w urządzeniu może korzystać z tych URI, żeby uzyskać dostęp do danych i je modyfikować. W związku z tym dostawcy treści pełnią istotną rolę w procesie współdzielenia danych pomiędzy aplikacjami.

Jednak ściślej mówiąc, obowiązki dostawców treści dotyczą w większym stopniu mechanizmów opakowania danych niż zapewniania do nich dostępu. Aby uzyskać dostęp do źródeł danych, potrzebny będzie rzeczywisty mechanizm dostępu do danych, na przykład baza SQLite lub dostęp sieciowy. Zatem abstrakcyjny obiekt dostawcy treści potrzebny jest jedynie w przypadku współużytkowania danych na zewnątrz lub pomiędzy aplikacjami. Przy wewnętrznym dostępie do danych aplikacja może korzystać z dowolnego, odpowiedniego mechanizmu przechowywania i dostępu, takiego jak:

- **Preferencje.** Zestawy par klucz – wartość mogących przechowywać preferencje danych.
- **Pliki.** Wewnętrzne pliki aplikacji, które mogą być przechowywane w wymiennym magazynie danych.
- **SQLite.** Baza danych SQLite, do której dostęp uzyskuje wyłącznie pakiet generujący daną bazę.
- **Sieć.** Mechanizm umożliwiający odczytywanie lub przechowywanie danych w internecie.

#### Uwaga!

Pomimo dużej liczby mechanizmów udostępniania danych obsługiwanych przez system Android niniejszy rozdział dotyczy bazy SQLite oraz abstrakcyjnych dostawców treści, ponieważ stanowią one podstawę technologii współdzielenia danych. Występuje ona o wiele powszechniej w strukturze Androida niż w innych szkieletach interfejsów UI. Mechanizm sieciowy zostanie omówiony w rozdziale 11., a mechanizm preferencji w rozdziale 9.

## Analiza wbudowanych dostawców Androida

Android wyposażono w wiele rodzajów wbudowanych dostawców treści. Dokumentacja tych dostawców jest dostępna w pakiecie *android.provider* zestawu SDK, a ich listę można znaleźć pod adresem <http://developer.android.com/reference/android/provider/package-summary.html>.

Poniżej została wymieniona część z dostawców treści omówionych na powyższej stronie WWW:

*Browser (Przeglądarka)*

*CallLog (Dziennik połączeń)*

*Contacts (Kontakty)*

*People (Osoby)*

*Phones (Telefony)*

*Photos (Zdjęcia)*

*Groups (Grupy)*

*MediaStore (Dane multimedialne)*

*Audio (Audio)*

- Albums (Albumy)*
- Artists (Wykonawcy)*
- Genres (Gatunki)*
- Playlists (Listy odtwarzania)*
- Images (Obrazy)*
- Thumbnails (Minitaturaly)*
- Video (Wideo)*
- Settings (Ustawienia)*

**Uwaga!**

W zależności od używanej wersji Androida lista dostawców może się składać z innych elementów. Zadaniem powyższej listy jest zaprezentowanie dostępnych elementów, jednak nie należy jej uznawać za bezwzględny punkt odniesienia.

Bazy danych są elementami najwyższego poziomu, natomiast elementami na niższych poziomach są tabele. Zatem pozycje Browser, CallLog, Contacts, MediaStore oraz Settings są oddzielnymi bazami danych SQLite, zdefiniowanymi jako dostawcy. Bazy te zazwyczaj posiadają rozszerzenie *.db* i są dostępne jedynie z poziomu pakietu implementacyjnego. Każda próba użycia dostępu spoza tego pakietu musi nastąpić poprzez interfejs dostawcy treści.

## Analiza baz danych na emulatorze oraz dostępnych urządzeniach

Wielu dostawców treści w Androidzie wykorzystuje bazy danych SQLite (<http://www.sqlite.org/>), zatem do badania tych baz można użyć narzędzi dostępnych zarówno w Androidzie, jak i w środowisku SQLite. Część tych narzędzi umieszczono w folderze *\katalog-instalacyjny-android-sdk\tools*, inne można znaleźć w katalogu *\android-sdk-install-directory\platform-tools*.

Jednym z narzędzi dostępnych w urządzeniu jest zdalna powłoka, umożliwiająca uruchomienie programu bazy SQLite w wierszu poleceń dla wybranej bazy danych. W dalszej części podpunktu omówimy metodę korzystania z tej aplikacji w celu przeanalizowania wbudowanych baz danych Androida.

**Uwaga!**

W rozdziale 2. można znaleźć informacje dotyczące położenia katalogu „narzędzi” oraz przywoływanego wiersza polecień w różnych systemach operacyjnych. W tym oraz w większości pozostałych rozdziałów podajemy przykłady głównie dla systemu Windows. W dalszej części podrоздziału będziemy często korzystać z narzędzi wiersza polecen. Rozdział 2. zawiera również instrukcję, w jaki sposób skonfigurować ścieżkę do katalogu z narzędziami w różnych systemach operacyjnych, więc Czytelnik nie musi się nią przejmować, trzeba jedynie znać nazwę pliku wykonywalnego lub wsadowego.

System Android posiada aplikację wiersza polecień znaną jako Android Debug Bridge (adb), którą można znaleźć w katalogu jako plik:

*platform-tools\adb.exe*

Jest to specjalne narzędzie, z którym większość pozostałych aplikacji musi się połączyć, zanim uzyska dostęp do urządzenia. Żeby jednak zadziałało, trzeba najpierw uruchomić emulator lub połączyć urządzenie. Za pomocą poniższego polecenia można sprawdzić, czy jest uruchomiony jakiś emulator lub urządzenie:

**adb devices**

Jeśli emulator nie działa, można go uruchomić za pomocą poniższego polecenia:

```
emulator.exe @avdname
```

Argument @avdname jest nazwą urządzenia AVD (w rozdziale 2. napisaliśmy o konieczności posiadania wirtualnego urządzenia AVD oraz omówiliśmy sposób jego utworzenia). Do wyświetlenia listy istniejących urządzeń wirtualnych służy polecenie:

```
android list avd
```

Na ekranie pojawi się lista dostępnych urządzeń AVD. Jeżeli w środowisku Eclipse zaprojektowano i uruchomiono przynajmniej jedną aplikację, musiało również zostać utworzone przynajmniej jedno urządzenie AVD. Powyższe polecenie spowoduje wyświetlenie nazwy przynajmniej tego urządzenia.

Poniżej zaprezentowano dane wyświetlane po wpisaniu powyższego polecenia (niektóre mogą ulec zmianie w zależności od ścieżki do katalogu *tools* — na przykład *i:\android* — oraz od wersji Androida):

```
I:\android\tools>android list avd
Available Android Virtual Devices:
  Name: avd
  Path: I:\android\tools\..\avds\avd3
  Target: Google APIs (Google Inc.)
            Based on Android 1.5 (API level 3)
  Skin: HVGA
  Sdcard: 32M
-----
  Name: titanium
  Path: C:\Documents and Settings\Satya\.android\avd\titanium.avd
  Target: Android 1.5 (API level 3)
  Skin: HVGA
```

Jak już wspomniano, urządzenia AVD zostały szczegółowo omówione w rozdziale 2.

Można również uruchomić emulator za pomocą wtyczki środowiska Eclipse. Dzieje się to automatycznie po uruchomieniu programu lub podczas sprawdzania błędów. Po uruchomieniu emulatora można jeszcze raz wywołać listę podłączonych urządzeń za pomocą polecenia:

```
adb devices
```

Powinny zostać wyświetcone informacje podobne do następującej:

---

```
List of devices attached
emulator-5554 device
```

---

Listę dostępnych opcji i poleceń można wywołać za pomocą poniższego polecenia:

```
adb help
```

Pod widocznym poniżej adresem została zamieszczona lista wielu opcji rozruchowych narzędzia adb:

<http://developer.android.com/guide/developing/tools/adb.html>

Istnieje możliwość uruchomienia za pomocą aplikacji adb okna powłoki w podłączonym urządzeniu:

```
adb shell
```

**Uwaga!** Jest to powłoka ash systemu Unix, pozbawiona jest jednak kilku poleceń. Dostępne jest polecenie ls, brakuje natomiast instrukcji find, grep oraz awk.

Dostępny zestaw poleceń powłoki zostaje wyświetlony po wpisaniu następującego polecenia w oknie zachęty powłoki:

```
#ls /system/bin
```

Symbol # jest znakiem zachęty powłoki. W celu zachowania zwięzłości będziemy go pomijać w następnych przykładach. Po wpisaniu powyższego polecenia pojawi się lista poleceń przedstawionych w tabeli 4.1. (Należy mieć na uwadze fakt, że tabela ta jest zaprezentowana wyłącznie w celach demonstracyjnych i nie zostały w niej wymienione wszystkie polecenia. W zależności od wersji zestawu Android SDK na liście mogą występować różne elementy).

Żeby zobaczyć katalogi i pliki podstawowego poziomu, wystarczy wpisać:

```
ls -l
```

Lista baz danych znajduje się w katalogu:

```
ls /data/data
```

Tu się znajduje lista wszystkich pakietów zainstalowanych w urządzeniu. Przyjrzyjmy się na przykład zawartości pakietu com.android.providers.contacts:

```
ls /data/data/com.android.providers.contacts/databases
```

Zostanie wyświetlony plik contacts.db, będący bazą danych SQLite (plik ten oraz ścieżka do niego zależą od rodzaju urządzenia oraz wersji systemu).

**Uwaga!** Powinniśmy wspomnieć, że w Androidzie można tworzyć bazy danych podczas pierwszej próby uzyskania do nich dostępu. Oznacza to, że powyższy plik może nie być widoczny, w przypadku gdy aplikacja korzystająca z „kontaktów” nie została jeszcze uruchomiona.

Gdyby w powłoce ash było dostępne polecenie find, istniałaby możliwość wyszukiwania wszystkich plików \*.db. Nie można tego wykonać w prosty sposób za pomocą samego polecenia ls. Najprostszym sposobem jest wpisanie:

```
ls -R /data/data/*databases
```

Dzięki temu poleceniu dowiadujemy się, że Android zawiera następujące bazy danych (jak zwykle liczba i rodzaje elementów listy mogą się różnić w zależności od edycji zestawu Android SDK):

```
alarms.db  
contacts.db  
downloads.db  
internal.db  
settings.db  
mmssms.db  
telephony.db
```

**Tabela 4.1.** Zestaw dostępnych polecień powłoki

dumpcrash	sh	date
am	hciattach	dd
dumpstate	sdptool	cmp
input	logcat	cat
itr	servicemanager	dmesg
monkey	dbus-daemon	df
pm	debug_tool	getevent
svc	flash_image	getprop
ssltest	installd	hd
debuggerd	dvz	id
dhpcd	hostapd	ifconfig
hostapd_cli	htclogkernel	insmod
fillup	mountd	ioctl
linker	qemud	kill
logwrapper	radiooptions	ln
telnetd	toolbox	log
iftop	hcid	lsmod
mkdosfs	route	ls
mount	setprop	mkdir
mv	sleep	dumpsys
notify	setconsole	service
netstat	smd	playmp3
printenv	stop	sdutil
reboot	top	rild
ps	start	dalvikvm
renice	umount	dexopt
rm	vmstat	surfaceflinger
rmdir	wipe	app_process
rmmod	watchprops	mediaserver
sendevent	sync	system_server
schedtop	netcfg	
ping	chmod	

Istnieje możliwość otwarcia bazy danych za pomocą aplikacji sqlite3 w powłoce adb, jeśli wpisze się następujący wiersz:

```
sqlite3 /data/data/com.android.providers.contacts/databases/contacts.db
```

Zamknięcie aplikacji następuje po wpisaniu polecenia:

```
sqlite>.exit
```

Należy zwrócić uwagę, że znakiem zachęty aplikacji adb jest #, natomiast w przypadku sqlite3 jest to sqlite>. Informacje dotyczące różnych poleceń aplikacji sqlite3 dostępnych wewnętrz powłoki adb można znaleźć pod adresem <http://www.sqlite.org/sqlite.html>, jednak omówimy tu kilka ważniejszych poleceń, dzięki czemu odwiedziny tej witryny nie będą konieczne. Lista tabel zostanie wyświetlona po wpisaniu:

```
sqlite>.tables
```

Polecenie to jest skrótną wersją kodu:

```
SELECT name FROM sqlite_master
WHERE type IN ('table','view') AND name NOT LIKE 'sqlite_%'
UNION ALL
SELECT name FROM sqlite_temp_master
WHERE type IN ('table','view')
ORDER BY 1
```

Jak można się domyślić, element sqlite\_master jest główną tabelą, zarządzającą pozostałymi tabelami i widokami bazy danych. Poniższy wiersz wywołuje instrukcję create dla tabeli people, znajdującej się w pliku contacts.db:

```
.schema people
```

Jest to jeden ze sposobów uzyskania nazw kolumn tabeli SQLite. Zostaną również wyświetlane typy danych zawartych w kolumnach. Podczas pracy z dostawcami usług typy danych będą pełniły ważną funkcję, gdyż od nich zależą metody dostępu do bazy danych.

Jednak analizowanie wyników instrukcji create jedynie w celu poznania nazw kolumn oraz typów danych w nich zawartych jest dosyć żmudnym zajęciem. Na szczęście istnieje inny sposób: można wyizolować plik contacts.db z pakietu, a następnie obejrzeć tabelę za pomocą dowolnego interfejsu GUI (ang. *Graphical User Interface* — graficzny interfejs użytkownika) obsługującego bazę danych SQLite w wersji 3. Dzięki poniższemu wierszowi, wpisanemu w oknie poleceń systemu operacyjnego, możliwe jest uzyskanie pliku contacts.db:

```
adb pull /data/data/com.android.providers.contacts/databases/contacts.db
c:/somelocaledir/contacts.db
```

Podczas zbierania materiałów do książki korzystaliśmy z darmowej aplikacji Sqldeman (<http://sqldeman.com/>), całkiem dobrze sprawującego się interfejsu graficznego dla baz danych SQLite. Kilkakrotnie przerwał działanie, poza tym jednak okazał się użytkowniczym narzędziem do przeglądania baz danych systemu Android.

## Krótki elementarz baz danych SQLite

Przedstawione poniżej przykładowe instrukcje SQLite mogą pomóc w nauce sprawnego posługiwania się bazami danych SQLite:

```
// Pokazuje nagłówki kolumn w oknie narzędzia
sqlite>.headers on
```

```
// Zaznacza wszystkie krótkie tabeli
select * from table1;
```

```
// Zlicza krótki tabeli
select count(*) from table1;

// Zaznacza określoną kategorię kolumn
select col1, col2 from table1;

// Zaznacza różne wartości w kolumnie
select distinct col1 from table1;

// Zlicza ilość unikatowych wartości
select count(col1) from (select distinct col1 from table1);

// Grupuje elementy w określonej kolejności
select count(*), col1 from table1 group by col1;

// Regularne wewnętrznełączenie (ang. inner join)
select * from table1 t1, table2 t2
where t1.col1 = t2.col1;

// Lewe zewnętrznełączenie (ang. outer join)
// Pobiera wszystko z t1, nawet jeśli nie ma krotek w t2
select * from table1 t1 left outer join table2 t2
on t1.col1 = t2.col1
where ....
```

## Architektura dostawców treści

Czytelnik już teraz wie, w jaki sposób przeglądać zawartość dostawców treści za pomocą narzędzi dostępnych w Androidzie oraz odpowiednich interfejsów GUI. Obecnie zajmiemy się analizą niektórych elementów struktury dostawców treści oraz ich powiązaniemi z innymi rodzajami abstrakcyjnych obiektów umożliwiających dostęp do danych.

Technologia dostawców treści ma swoje odpowiedniki w następujących mechanizmach:

- stronach WWW,
- architekturze REST,
- usługach sieciowych,
- procedurach składowanych.

Każdy dostawca treści zostaje zarejestrowany w urządzeniu jako strona WWW poprzez ciąg znaków (podobny do nazwy domeny, tu jednak nazywany **upoważnieniem**). Taki niepowtarzalny ciąg znaków jest podstawą dla zestawu identyfikatorów URI, udostępnianych przez każdego dostawcę treści. W podobny sposób strona WWW wraz z domeną posiada adresy URL, odsyłające do jej dokumentów lub, ogólnie, treści.

Rejestracja upoważnienia przebiega w pliku *AndroidManifest.xml*. Poniżej zaprezentowano dwa przykłady rejestrowania dostawców treści w tym pliku:

```
<provider android:name="SomeProvider"
          android:authorities="com.your-company.SomeProvider" />
<provider android:name="NotePadProvider"
          android:authorities="com.google.provider.NotePad"
/>
```

Upoważnienie pełni funkcję nazwy domeny dla danego dostawcy treści. Na podstawie powyższych przykładów rejestracji upoważnień dostawcy treści będą honorować adresy URL rozpoczynające się od prefiksu upoważnienia:

```
content://com.your-company.SomeProvider/
content://com.google.provider.NotePad/
```

Widac więc, że „dostawcy treści”, na przykład strony WWW, posiadają podstawową nazwę domeny, zachowującą się jak początek adresu URL.

#### Uwaga!

Należy zwrócić uwagę, że dostawcy treści w Androidzie nie muszą posiadać pełnej, złożonej nazwy upoważnienia. Obecnie jest ona zalecana wyłącznie dla dostawców treści wydawanych przez niezależnych producentów. Dlatego właśnie czasami niektórzy dostawcy treści opisywani są jednym słowem, na przykład *contacts*, w przeciwieństwie do *com.google.android.contacts* (w przypadku dostawcy treści od niezależnego producenta).

Dostawcy treści zapewniają także adresy URL oparte na architekturze REST, służące do odczytywania lub modyfikowania danych. Na bazie powyższej rejestracji identyfikator URI służący do rozpoznawania katalogu lub zbioru notatek bazy danych *NotePadProvider* będzie wyglądał następująco:

```
content://com.google.provider.NotePad/Notes
```

Identyfikator URI dotyczy określonej notatki (mianowicie *content://com.google.provider.NotePad/Notes/#*), gdzie # oznacza atrybut id tej notatki. Poniżej wypisano inne przykłady identyfikatorów URI akceptowanych przez dostawców treści:

```
content://media/internal/images
content://media/external/images
content://contacts/people/
content://contacts/people/23
```

Zwróćmy uwagę, że „multimedia” tych dostawców (*content://media*) oraz ich „kontakte” (*content://contacts*) nie posiadają pełnej, złożonej struktury. Wynika to z faktu, że dostawcy ci zostali dostarczeni przez niezależnych producentów i są kontrolowani przez Androida.

Dostawcy treści wykazują również właściwości usług sieciowych. Poprzez swoje identyfikatory URI dostawca treści przedstawia wewnętrzne dane w formie usługi. Jednak inaczej niż w przypadku wywołań usług sieciowych opartych na protokole SOAP na końcu adresu URL, wewnętrzne dane dostawcy treści nie są typowymi danymi. Bardziej przypominają zestaw wynikowy dostępny w interfejsie JDBC. Także tutaj podobieństwa dotyczą wyłącznie koncepcji. Nie chcemy, aby Czytelnik odniósł wrażenie, że te dane są tożsame z obiektem *ResultSet* interfejsu JDBC.

Aby wywołanie przebiegło pomyślnie, wymagana jest znajomość struktury zwracanych wierszy i kolumn. W punkcie „Struktura typów MIME w Androidzie” zostaną omówione wbudowane mechanizmy, pozwalające na określenie typu MIME (ang. *Multipurpose Internet Mail Extensions* — uniwersalne rozszerzenia poczty internetowej) danych reprezentowanych przez identyfikatory URI.

Poza podobieństwem do stron WWW, architektury REST oraz usług sieciowych identyfikatory URI dostawców treści wykazują również powinowactwo z nazwami procedur składowanych w bazach danych. Procedury te pozwalają na oparty na usługach dostęp do relacyjnych, podstawowych danych. Identyfikatory URI są podobne do procedur składowanych, gdyż

ich wywołanie powoduje przekazanie kursora. Jednak obydwa mechanizmy różnią się tym, że w przypadku dostawcy treści dane wejściowe wywoływanej usługi są zazwyczaj zagnieżdżone bezpośrednio w identyfikatorze URI.

Powyższe porównanie technologii ma na celu ukazanie szerszego zakresu funkcjonalności dostawców treści.

## Struktura identyfikatorów URI dostawców treści

Porównaliśmy dostawcę treści do strony WWW, ponieważ reaguje on na przychodzące identyfikatory URI. Aby zatem odczytać dane umieszczone w dostawcy treści, wystarczy wywołać jego identyfikator URI. Jednak w przypadku dostawców treści dane są odczytywane w formie zbioru krotek i kolumn, reprezentowanych przez obiekt cursor. W takim kontekście przeanalizujemy strukturę identyfikatora URI, żeby się dowiedzieć, w jaki sposób odczytywać dane.

Identyfikatory URI w Androidzie przypominają nieco identyfikatory URI protokołu HTTP, różnią się tylko początkowym członem `content` oraz ogólną strukturą:

`content:///*/*/*`

lub

`content://authority-name/path-segment1/path-segment2/itd...`

Poniżej został ukazany przykładowy identyfikator URI, odnoszący się do notatki nr 23 w bazie notatek:

`content://com.google.provider.NotePad/notes/23`

Po członie `content:` umieszczono niepowtarzalny identyfikator upoważnienia, który jest używany do zlokalizowania dostawcy w rejestrze dostawców. W powyższym przykładzie członem stanowiącym upoważnienie jest `com.google.provider.NotePad`.

Człon `/notes/23` jest sekcją określającą ścieżkę, inną dla każdego dostawcy. Człyony `notes` oraz `23` nazywane są segmentami ścieżki. Zadaniem dostawcy jest określenie oraz zinterpretowanie sekcji oraz segmentów ścieżki w danym identyfikatorze URI.

Proces projektowania dostawcy treści polega przeważnie na deklarowaniu stałych w klasie Java lub interfejsie Java, umieszczonych w tej samej implementacji pakietu, w której znajduje się dostawca. Co więcej, pierwszy człon sekcji ścieżki może wskazywać na zbiór obiektów. Na przykład człon `/notes` wskazuje zbiór lub katalog notatek, a segment `/23` precyzuje interesującą nas notatkę.

Po otrzymaniu identyfikatora URI zadaniem dostawcy treści jest odczytanie wierszy wskazywanych przez ten identyfikator. Zadaniem dostawcy jest również zmiana zawartości takiego identyfikatora za pomocą którejś z metod zmiany stanu: wstawiania, aktualizowania lub usuwania.

## Struktura typów MIME w Androidzie

Tak samo jak w przypadku strony WWW przekazującej danemu adresowi URL typ MIME (dzięki czemu przeglądarka uruchamia właściwą aplikację do przeglądania zawartości strony), dostawca treści przekazuje typ MIME określonym identyfikatorowi URI. Zapewnia to elastyczność przeglądania danych. Znając typ danych, można przypisać kilka różnych programów do jego obsługi. Na przykład umieszczony na dysku plik tekstowy można otworzyć w kilku

rodzajach edytorów tekstu. W zależności od systemu operacyjnego może pojawić się nawet opcja wyboru któregoś z edytorów.

Typy MIME działają w Androidzie podobnie jak w przypadku protokołu HTTP. Dostawca otrzymuje żądanie typu MIME obsługiwanej identyfikatora URI, a następnie przekazuje składający się z dwóch części ciąg znaków, który służy do identyfikowania tego typu MIME zgodnie ze standardową konwencją sieciową. Standardy typów MIME można znaleźć pod adresem:

<http://tools.ietf.org/html/rfc2046>

Według specyfikacji typu MIME składa się on z dwóch części: typu oraz podtypu. Poniżej zaprezentowano przykłady znanych par typu MIME:

```
text/html  
text/css  
text/xml  
text/vnd.curl  
application/pdf  
application/rtf  
application/vnd.ms-excel
```

P pełną listę zarejestrowanych typów i podtypów można przejrzeć w witrynie organizacji IANA (ang. *Internet Assigned Numbers Authority* — Urząd Przydzielania Numerów Internetowych):

<http://www.iana.org/assignments/media-types/>

Podstawowymi zarejestrowanymi typami treści są:

```
application  
audio  
example  
image  
message  
model  
multipart  
text  
video
```

Każdy główny typ posiada podtypy. Jeżeli jednak producent wykorzystuje zastrzeżony format danych, nazwa podtypu rozpoczyna się od vnd. Na przykład arkusze kalkulacyjne Microsoft Excel są identyfikowane jako podtyp o nazwie vnd.ms-excel, podczas gdy standard pdf nie jest zastrzeżony, dlatego jego identyfikator nie posiada żadnego przedrostka określającego producenta.

Pewne podtypy posiadają przedrostek x- w nazwie; są to podtypy niestandardowe, których nie trzeba rejestrować. Są one uznawane za prywatne wartości, które są dwustronnie definiowane pomiędzy dwoma współpracującymi agentami. Poniżej znajduje się kilka przykładów:

```
application/x-tar  
audio/x-aiff  
video/x-msvideo
```

Android posługuje się konwencją podobną do definiowania typów MIME. Prefiks vnd wskazuje na to, że typy i podtypy są niestandardowymi formami określonymi przez producenta. W celu zapewnienia niepowtarzalności Android idzie dalej w kierunku rozgraniczenia wieloczęściowych typów i podtypów przypominających specyfikację domenową. Co więcej, typ MIME w Androidzie przybiera dwie formy dla każdej treści: jedną dla określonego rekordu, a drugą dla wielu rekordów.

Dla pojedynczego rekordu typ MIME wygląda następująco:

vnd.android.cursor.item/vnd.yourcompanyname.contenttype

Dla krotek lub zbioru rekordów wygląda on tak:

vnd.android.cursor.dir/vnd.yourcompanyname.contenttype

Dwa przykłady:

// Pojedyncza notatka

vnd.android.cursor.item/vnd.google.note

// Zbiór lub katalog notatek

vnd.android.cursor.dir/vnd.google.note

### Uwaga!

Nasuwa się wniosek, że Android rozpoznaje natywnie „katalog” elementów oraz „pojedyncze” elementy. Elastyczność programistyczna jest ograniczona do podtypów.

Na przykład takie elementy jak opcje listy zależą od tego, co zostanie przekazane w kursorze jako jeden z „głównych” typów MIME.

Typy MIME są powszechnie stosowane w Androidzie, zwłaszcza w intencjach, w przypadku których system właśnie dzięki typom danych MIME może wybrać właściwą aktywność i ją przywołać. Typy MIME są niezmiennie wydzielane z ich identyfikatorów URI dzięki dostawcom treści. Należy pamiętać o trzech kwestiach podczas pracy z tymi typami:

- Zarówno typ, jak i podtyp muszą w sposób jednoznaczny reprezentować wskazywany element. Jak stwierdziliśmy, typ jest ściśle ustalony w postaci katalogu elementów lub pojedynczego elementu. W kontekście systemu Android nie są one zbyt otwartymi obiekttami.
- Jeżeli typ i podtyp nie są standardowe, należy poprzedzić je prefiksem vnd (zazwyczaj dzieje się to w przypadku określonych rekordów).
- Przeważnie przydziela się im przestrzeń nazw zgodnie z potrzebą.

Podsumowując, główny typ MIME zbioru elementów zwróconych przez obiekt cursor powinien zawsze wyglądać tak jak vnd.android.cursor.dir, natomiast analogiczny przypadek dla pojedynczego elementu powinien przybrać postać vnd.android.cursor.item. Większe pole do manewru istnieje w przypadku podtypu, na przykład vnd.google.note; po przedrostku vnd. można wpisać dowolną nazwę.

## Odczytywanie danych za pomocą identyfikatorów URI

Wiadomo teraz, że do odczytu danych zawartych w dostawcy treści należy wykorzystać identyfikatory URI tego dostawcy. Ponieważ identyfikatory te są niepowtarzalne dla każdego dostawcy, bardzo ważne jest ich udokumentowanie oraz udostępnienie ich listy programistom do wglądu, a następnie do ich wywoływania. Zaimplementowana w Androidzie technologia dostawców dokonuje tego poprzez definiowanie stałych, reprezentujących te ciągi znaków URI.

Przyjrzyjmy się trzem identyfikatorom URI, zdefiniowanym przez klasy pomocnicze w środowisku Android SDK:

MediaStore.Images.Media.INTERNAL\_CONTENT\_URI

MediaStore.Images.Media.EXTERNAL\_CONTENT\_URI

Contacts.People.CONTENT\_URI

Ich odpowiednikami w postaci tekstowych ciągów znaków są:

```
content://media/internal/images
content://media/external/images
content://contacts/people/
```

Dostawca MediaStore definiuje dwa identyfikatory URI, a dostawca Contacts określa jeden identyfikator. Można zauważyć, że te stałe są definiowane za pomocą drzewa hierarchii. Przykładowy identyfikator URI dla kontaktów jest zdefiniowany jako `Contacts.People`. →`CONTENT_URI`. Spowodowane jest to faktem, że bazy danych kontaktów mogą korzystać z wielu tablic do reprezentowania jednostek w dostawcy `Contacts`. Kategoria `People` jest jedną z tablic lub zbiorów. Każda podstawowa jednostka bazy danych może posiadać własny identyfikator URI, wszystkie są jednak zakończone za pomocą podstawowej nazwy upoważnienia (w przypadku dostawcy kontaktów jest to `contacts://contacts`).

#### Uwaga!

W odniesieniu `Contacts.People.CONTENT_URI` element `Contacts` jest pakietem Java, a składnik `People` stanowi interfejs wewnętrz tego pakietu. Zwróćmy również uwagę na fakt, iż identyfikatory `Contacts` oraz `Contacts.people` stały się przestarzałe w wersji 2.0 Androida, a ich nowe odpowiedniki zostały omówione w rozdziale 27. Jednak identyfikatory te ciągle są przydatne, zwłaszcza do omawiania koncepcji dostawców treści.

Biorąc pod uwagę takie identyfikatory URI, kod służący do odczytania jednego wiersza dostawcy treści zawierającego imiona ludzi wygląda następująco:

```
Uri peopleBaseUri = Contacts.People.CONTENT_URI;
Uri myPersonUri = Uri.withAppendedId(Contacts.People.CONTENT_URI, "23");

// Kwerenda dla tego rekordu
// managedQuery jest metodą w klasie Activity
Cursor cur = managedQuery(myPersonUri, null, null, null);
```

Zwróćmy uwagę, że identyfikator `Contacts.People.CONTENT_URI` jest zdefiniowany jako stała w klasie `People`. W tym przykładzie do głównego identyfikatora URI zostaje dodany identyfikator określonej osoby i następuje wywołanie metody `managedQuery`.

Jako część kwerendy w identyfikatorze URI można wprowadzić kolejność sortowania, zaznaczanie określonych kolumn oraz klauzulę `WHERE`. W powyższym przykładzie te parametry mają wartość `null`.

#### Uwaga!

Dostawca treści powinien wyświetlić listę obsługiwanych przez niego kolumn poprzez zaimplementowanie zestawu interfejsów lub wyświetlenie nazw tych kolumn w postaci stałych. Jednak klasa lub interfejs, które definiują kolumny w formie stałych, powinny także w jasny sposób określać typ kolumny za pomocą odpowiednio dobranej konwencji nazewnictwa, komentarzy lub dokumentacji, ponieważ nie istnieje formalny sposób określania typu kolumny za pomocą stałych.

Na podstawie wcześniejszych przykładów na listingu 4.1 przedstawiono metodę wywołania kursora, zawierającego określoną listę kolumn z tabeli `People` umieszczonej wewnątrz dostawcy treści `contacts`.

**Listing 4.1.** Wywołanie obiektu Cursor z dostawcy treści

```
// Tabela określająca zwracane kolumny
string[] projection = new string[] {
    People._ID,
    People.NAME,
    People.NUMBER,
};

// Odczytuje bazę identyfikatorów URI tablicy People w dostawcy treści Contacts
// np. content://contacts/people/
Uri mContactsUri = Contacts.People.CONTENT_URI;

// Najlepsza metoda odczytywania kwerendy; zwraca zarządzaną kwerendę
Cursor managedCursor = managedQuery( mContactsUri,
    projection, //Która kolumna będzie zwrocona
    null, //klauzula WHERE
    Contacts.People.NAME + " ASC"); // klauzula sortowania
```

---

Zwróćmy uwagę, że obiekt `projection` jest jedynie tablicą ciągów znaków, reprezentującą nazwy kolumn. Zatem bez znajomości nazw tych kolumn trudno będzie go utworzyć. Ich nazwy można znaleźć w tej samej klasie, która dostarcza identyfikator URI, w tym przypadku w klasie `People`. Zobaczmy, jakie są w niej zdefiniowane nazwy pozostałych kolumn:

```
CUSTOM_RINGTONE
DISPLAY_NAME
LAST_TIME_CONTACTED
NAME
NOTES
PHOTO_VERSION
SEND_TO_VOICE_MAIL
STARRED
TIMES_CONTACTED
```

Informacje na temat tych kolumn można znaleźć w dokumentacji pakietu SDK dotyczącej klasy `android.provider.Contacts.PeopleColumns`. Dokumentacja ta jest dostępna pod adresem:

<http://developer.android.com/reference/android/provider/Contacts.PeopleColumns.html>

Jak już wcześniej zasugerowaliśmy, baza danych typu `contacts` zawiera wiele tabel, z których każda jest reprezentowana przez klasę lub interfejs, co umożliwia opisanie tych kolumn oraz ich typów. Przejrzyjmy się pakietowi `android.providers.Contacts`, którego kod źródłowy można przejrzeć na stronie:

<http://developer.android.com/reference/android/provider/Contacts.html>

Pakiet zawiera następujące zagnieżdżone klasy lub interfejsy:

```
ContactMethods
Extensions
Groups
Organizations
People
```

```
Phones
Photos
Settings
```

Każda z tych klas reprezentuje nazwę tabeli w bazie danych *contacts.db*, natomiast wszystkie tabele mają opisać strukturę swoich własnych identyfikatorów URI. W dodatku dla każdej klasy jest zdefiniowany odpowiedni interfejs *Columns*, umożliwiający identyfikację nazw kolumn, na przykład *PeopleColumns*.

Spójrzmy jeszcze na otrzymany obiekt *Cursor* — może nie zawierać żadnych rekordów. Nazwy, typ oraz kolejność kolumn są określone dla każdego dostawcy. Jednak każdy zwrócony wiersz posiada domyślną kolumnę *\_id*, stanowiącą niepowtarzalny identyfikator tego wiersza.

## Korzystanie z kurSORA systemu Android

Oto kilka faktów na temat kurSORA systemu Android:

- KurSOR jest zbiorem krotek.
- Należy skorzystać z metody *moveToFirst()* przed odczytem jakichkolwiek danych, ponieważ kurSOR jest ustawiony przed pierwszą krotką.
- Niezbędna jest znajomość nazw kolumn.
- Niezbędna jest znajomość typów kolumn.
- Wszystkie metody pola dostępu opierają się na numerze kolumny, zatem należy najpierw przekształcić nazwę kolumny na jej numer.
- Obiekt *Cursor* jest kurSorem swobodnym (może poruszać się do przodu, do tyłu oraz przeskakiwać).
- Dzięki powyższej właściwości istnieje możliwość wykorzystania go do zliczania krotek.

Istnieje wiele metod, dzięki którym można sterować kurSorem w Androidzie. Na listingu 4.2 pokazaliśmy, w jaki sposób można sprawdzić, czy kurSOR jest pusty, oraz w jaki sposób go przemieszczać krotką po krotce, w przypadku gdy nie jest pusty.

**Listing 4.2.** Sterowanie kurSorem za pomocą pętli while

---

```
if (cur.moveToFirst() == false)
{
    // brak wierszy, pusty kurSOR
    return;
}

// KurSOR wskazuje pierwszy wiersz
// Uzyskajmy dostęp do kilku kolumn
int nameColumnIndex = cur.getColumnIndex(People.NAME);
String name = cur.getString(nameColumnIndex);

// Niech kurSOR sprawdza wiersze po kolei

while(cur.moveToNext())
{
    // KurSOR pomyślnie przeniesiony
    // poła dostępu
}
```

---

Na początku listingu 4.2 założyliśmy, że kursor jest ulokowany przed pierwszą krotką. Żeby umieścić kursor na pozycji pierwszego wiersza, stosujemy metodę `moveToFirst()` wobec obiektu `cursor`. Jeżeli kursor jest pusty, otrzymujemy wartość `false`. Korzystamy więc wielokrotnie z metody `moveToNext()`, żeby przesuwać kursor.

Aby się dowiedzieć, gdzie w danej chwili znajduje się kursor, można skorzystać z poniższych metod:

```
isBeforeFirst()  
isAfterLast()  
isClosed()
```

Można je również wykorzystywać w pętli `for` (listing 4.3) zamiast w przedstawionej na listingu 4.2 pętli `while`.

#### **Listing 4.3.** Sterowanie kursorem za pomocą pętli `for`

---

```
//Najpierw uzyskujemy indeksy spoza pętli  
int nameColumn = cur.getColumnIndex(People.NAME);  
int phoneColumn = cur.getColumnIndex(People.NUMBER);  
  
//Ruch kursora jest teraz zależny od indeksów kolumn  
for(cur.moveToFirst();!cur.isAfterLast();cur.moveToNext())  
{  
    String name = cur.getString(nameColumn);  
    String phoneNumber = cur.getString(phoneColumn);  
}
```

---

Kolejność występowania indeksów w kolumnach zdaje się być przyjęta nieco arbitralnie. Z tego powodu zalecamy pozyskiwanie w jawnym sposób indeksów z kurSORA w celu uniknięcia niespodzianek. Żeby określić liczbę krotek objętych kursorem, stosuje się dostępną w Androidzie metodę `getCount()`.

### **Praca z klauzulą WHERE**

Istnieją dwa sposoby wprowadzenia klauzuli `WHERE` w dostawcach treści:

- za pomocą identyfikatora URI,
- za pomocą kombinacji klauzuli `string` oraz zestawu wymiennych argumentów, stanowiących tablice ciągów znaków.

Obydwie wymienione metody przedstawimy w formie przykładowych kodów.

#### **Wprowadzanie klauzuli WHERE za pomocą identyfikatora URI**

Wyobraźmy sobie, że chcemy odczytać notatkę o identyfikatorze 23 z bazy notatek Google. Do uzyskania kurSORA zawierającego jeden wiersz odpowiadający wierszowi nr 23 tablicy notatek można napisać kod zaprezentowany na listingu 4.4.

#### **Listing 4.4. Wprowadzanie klauzul WHERE języka SQL za pomocą identyfikatora URI**

---

```
Activity someActivity;  
//...inicjalizacja aktywności someActivity  
String noteUri = "content://com.google.provider.NotePad/notes/23";
```

```
Cursor managedCursor = someActivity.managedQuery( noteUri,
    projection, // Które kolumny zostaną przekazane
    null, // Klauzula WHERE
    null); // Klauzula sortowania
```

Pozostawiłyśmy wartość `null` w argumencie klauzuli `WHERE`, będącej częścią metody `managedQuery`, ponieważ w tym przypadku założyliśmy, że dostawca notatek jest w stanie sam określić wartość obiektu `id` pożąданej notatki. Wartość ta jest umieszczona w identyfikatorze URI. Użyliśmy identyfikatora URI jako pojemnika do wprowadzenia klauzuli `WHERE`. Staje się to zrozumiałe, gdy się zwróci uwagę, w jaki sposób dostawca treści implementuje związaną z nim metodę `kwhere`. Poniżej umieściliśmy fragment kodu takiej metody:

```
// Uzyskuje identyfikator notatki z przychodzącego identyfikatora uri, wyglądającego jak
// content://.../notes/23
int noteId = uri.getPathSegments().get(1);

// Źródło od konstruktora kwerend utworzenia zapytania
// Określa nazwę tabeli
queryBuilder.setTables(NOTES_TABLE_NAME);

// Wykorzystuje wartość noteID do wstawienia klauzuli WHERE
queryBuilder.appendWhere(Notes._ID + " = " + noteId);
```

Zauważmy, w jaki sposób obiekt `id` notatki jest uzyskiwany z identyfikatora URI. Klasa `Uri`, reprezentująca nadchodzący element `uri`, posiada metodę pozwalającą na wydobycie fragmentów identyfikatora następujących po głównej części `content://com.google.provider.NotePad`. Fragmenty te noszą nazwę **segmentów ścieżki**; są to ciągi znaków oddzielonych znakiem `/` — na przykład `/seg1/seg3/seg4/` — indeksowane na podstawie pozycji. Dla naszego identyfikatora URI pierwszym segmentem może być `23`. Segment ten należy dodać do klauzuli `WHERE`, określonej w klasie `QueryBuilder`. Ostatecznie równoważna instrukcja wyboru wyglądałaby następująco:

```
select * from notes where _id = 23
```

#### Uwaga!

Klasy `Uri` oraz `UriMatcher` używane są do rozpoznawania identyfikatorów URI oraz wydobywania z nich parametrów (klasa `UriMatcher` zostanie omówiona w podpunkcie „*Stosowanie klasy UriMatcher do rozpoznawania identyfikatorów URI*”). W pakiecie `android.database.sqlite` znajduje się pomocnicza klasa `SQLiteQueryBuilder`, pozwalająca na konstruowanie kwerend SQL wykonywanych przez klasę `SQLiteDatabase` w wystąpieniu bazy danych SQLite.

### **Stosowanie jawnych klauzul WHERE**

Po zaprezentowaniu metody, w której identyfikator URI służy do wprowadzania klauzuli `WHERE`, nadszedł czas na pokazanie drugiego sposobu, umożliwiającego wysyłanie listy jawnych kolumn oraz odpowiadających im wartości w formie tej klauzuli. Przyjrzyjmy się ponownie metodzie `managedQuery` z klasy `Activity`, przedstawionej na listingu 4.4. Oto jej sygnatura:

```
public final Cursor managedQuery(Uri uri,
    String[] projection,
    String selection,
```

```
String[] selectionArgs,  
String sortOrder)
```

Zwróćmy uwagę na argument `selection`, którego zadeklarowanym typem jest `String`. Ten ciąg znaków pełni funkcję filtra (w istocie klauzuli `WHERE`) określającego wiersze, które zostaną otrzymane i sformatowane do postaci klauzuli `WHERE` języka SQL (sama klauzula `WHERE` zostaje pominięta). Parametr `null` zwróci wszystkie krotki ze wskazanego identyfikatora URI. W ciągu znaków wyboru można wstawić znaki zapytania, które będą zastępowane przez wartości argumentu `selectionArgs` w kolejności ich pojawiania się. Typem wartości będzie `String`.

Ponieważ istnieją dwie metody określania klauzuli `WHERE`, mogą pojawić się problemy ze stwierdzeniem, w jaki sposób dostawca treści wykorzystał te klauzule oraz która z nich uzyskuje pierwszeństwo, jeżeli są wykorzystywane obydwa mechanizmy.

Na przykład można wysłać kwerendę dotyczącą notatki numer 23 na obydwa sposoby:

```
// metoda URI  
managedQuery("content://com.google.provider.NotePad/notes/23"  
,null  
,null  
,null  
,null);
```

lub

```
// jawna klauzula WHERE  
managedQuery("content://com.google.provider.NotePad/notes"  
,null  
, "_id=?"  
,new String[] {23}  
,null);
```

Zgodnie z konwencją metodę URI stosuje się zawsze tam, gdzie istnieje taka możliwość, natomiast jawne definiowanie klauzuli `WHERE` używane jest w specjalnych przypadkach.

## Wstawianie rekordów

Do tej pory zajmowaliśmy się odczytywaniem danych z dostawców treści za pomocą identyfikatorów URI. Teraz omówimy techniki wstawiania, aktualizowania oraz usuwania danych.

### Uwaga!

W trakcie objaśniania zagadnienia dostawców treści szeroko wykorzystujemy przykłady zaczerpnięte z prototypowej aplikacji Notepad, którą firma Google zamieściła jako część samouczka. Nie jest jednak wymagana doskonała znajomość tego programu. Nawet osoby, które go nie znają, powinny bez większego problemu zrozumieć przytaczane tu przykłady. W dalszej części rozdziału zaprezentujemy pełny kod przykładowego dostawcy treści.

Klasa `android.content.ContentValues` w Androidzie służy do przechowywania wstawianej wartości pojedynczego rekordu. Klasa `ContentValues` stanowi katalog par klucz – wartość, na przykład nazwa kolumny i jej wartości. Rekordy są wstawiane najpierw poprzez wprowadzenie rekordu do klasy `ContentValues`, a następnie klasa `android.content.ContentResolver` wstawia ten rekord za pomocą identyfikatora URI.

**Uwaga!**

Klasa ContentResolver jest niezbędna, ponieważ na tym etapie nie wymaga się wstawiania rekordu do bazy danych. Zamiast tego wstawią się rekord do dostawcy określonego przez identyfikator URI. Klasa ContentResolver rozpoznaje właściwego dostawcę treści i przekazuje mu obiekt umieszczonego w klasie ContentValues.

Poniżej zaprezentowano przykład umieszczenia pojedynczego wiersza z notatkami w klasie ContentValues:

```
ContentValues values = new ContentValues();
values.put("title", "Nowa notatka");
values.put("note", "To jest nowa notatka");
```

*// obiekt values jest teraz przygotowany do wstawienia*

Odniesienie do klasy ContentResolver otrzymuje się poprzez zapytanie klasy Activity:

```
ContentResolver contentResolver = activity.getContentResolver();
```

Teraz wystarczy wskazać identyfikator URI klasie ContentResolver, żeby wstawić wiersz. Identyfikatory te są zdefiniowane w klasie odpowiadającej tabeli Notes. Na przykładzie aplikacji Notepad identyfikatorem jest:

```
Notepad.Notes.CONTENT_URI
```

Możemy teraz wykorzystać posiadany identyfikator URI oraz klasę ContentValues i utworzyć żądanie wstawienia wiersza:

```
Uri uri = contentResolver.insert(Notepad.Notes.CONTENT_URI, values);
```

Otrzymujemy identyfikator URI wskazujący na nowo wstawiony rekord. Identyfikator ten będzie posiadał następującą strukturę:

```
Notepad.Notes.CONTENT_URI/new_id
```

## Dodawanie pliku do dostawcy treści

Czasami może zaistnieć potrzeba przechowania pliku w bazie danych. Standardowym sposobem jest zapisanie takiego pliku na dysku, a następnie aktualizacja rekordu bazy danych, który ma wskazywać nazwę tego pliku.

Android korzysta z tego protokołu i go automatyzuje poprzez definiowanie specyficznej procedury zachowywania oraz odczytywania tych plików. Stosuje się konwencję, wedle której nazwa z odniesieniem pliku jest zapisywana w rekordzie znajdującym się w specjalnie zarezerwowanej do tego celu kolumnie \_data.

Po wstawieniu rekordu do tabeli Android odsyła żądającemu programowi identyfikator URI. Po zapisaniu rekordu za pomocą tego mechanizmu należy zapisać plik w tej lokacji. Aby można było to zrobić, Android umożliwia klasie ContentResolver uzyskanie wartości parametru uri z rekordu bazy danych i zwrócenie zapisywального strumienia danych wynikowych. Następnie Android umieszcza wewnętrzny plik oraz magazynuje odniesienie do jego nazwy w polu kolumny \_data.

Można rozwinąć przykład z aplikacją Notepad o zapisanie obrazu z określonej notatki. Należałyby w takim przypadku stworzyć dodatkową kolumnę nazwaną \_data oraz zastosować po raz pierwszy instrukcję insert, żeby otrzymać identyfikator URI. Poniżej zademonstrowano opisany fragment protokołu:

```
ContentValues values = new ContentValues();
values.put("title", "Nowa notatka");
values.put("note", "To jest nowa notatka");

// Korzysta z rozpoznawania treści, żeby wstawić rekord
ContentResolver contentResolver = activity.getContentResolver();
Uri newUri = contentResolver.insert(Notepad.Notes.CONTENT_URI, values);
```

Po uzyskaniu identyfikatora URI rekordu poniższy kod zostanie użyty do zażądania od klasy ContentResolver odniesienia do wyjściowego strumienia danych:

```
...
// Program rozpoznający treść bezpośrednio uzyskuje dostęp do wyjściowego
// strumienia
// Klasa ContentResolver ukrywa dostęp do pola _data, w którym jest przechowywane
// odniesienie do prawdziwego pliku.
OutputStream outStream = activity.getContentResolver().openOutputStream(newUri);
someSourceBitmap.compress(Bitmap.CompressFormat.JPEG, 50, outStream);
outStream.close();
```

Następnie wyjściowy strumień zostaje wykorzystany przez kod do zapisu danych.

## Aktualizowanie oraz usuwanie

Dotychczas zajmowaliśmy się kwerendami oraz instrukcją wstawiania danych. Instrukcje aktualizowania oraz usuwania danych nie są skomplikowane. Wykonanie operacji aktualizacji jest podobne do wykonywania operacji wstawienia, czyli zmienione wartości kolumn przechodzą przez klasę ContentValues. Sygnatura metody update w obiekcie ContentResolver wygląda następująco:

```
int numberOfRowsUpdated =
activity.getContentResolver().update(
    Uri uri,
    ContentValues values,
    String whereClause,
    String[] selectionArgs )
```

Argument whereClause ogranicza aktualizację do właściwych krotek. Sygnatura metody delete wygląda analogicznie:

```
int numberOfRowsDeleted =
activity.getContentResolver().delete(
    Uri uri,
    String whereClause,
    String[] selectionArgs )
```

Oczywiście w metodzie delete argument ContentValues staje się zbędny, gdyż nie ma potrzeby określania kolumn podczas usuwania rekordu.

Niemal wszystkie wywołania z klas managedQuery oraz ContentResolver są ostatecznie kierowane do klasy provider. Wiedza, w jaki sposób dostawca implementuje każdą z tych metod, pozwala wysnuć wnioski na temat techniki wykorzystywania ich przez klienta. W następnej sekcji omówimy od podstaw implementację przykładowego dostawcy treści, nazwanego BookProvider.

## Implementowanie dostawców treści

Przedyskutowaliśmy metody interakcji z dostawcą treści pod kątem pracy na danych, jednak jeszcze nie pokazaliśmy, jak napisać nowego dostawcę. W celu utworzenia dostawcy treści należy rozszerzyć klasę `android.content.ContentProvider` i zaimplementować następujące główne metody:

```
query
insert
update
delete
getType
```

Przed implementacją tych metod należy skonfigurować kilka rzeczy. Omówimy implementację dostawcy treści w formie szczegółowego opisu każdego z następujących etapów:

1. Zaprojektuj bazę danych, identyfikatory URI, nazwy kolumn i tak dalej. Utwórz klasę metadanych, w której będą zdefiniowane stałe dla wszystkich elementów metadanych.
2. Rozszerz abstrakcyjną klasę `ContentProvider`.
3. Zaimplementuj metody: `query`, `insert`, `update`, `delete` oraz `getType`.
4. Zarejestruj dostawcę w pliku manifeście.

## Planowanie bazy danych

Aby przedstawić to zagadnienie, pokażemy, jak zbudować bazę danych zawierającą kolekcję książek. Zawiera ona tylko jedną tabelę `books`, a jej kolumny noszą nazwy `name`, `isbn` oraz `author`. Nazwy tych kolumn stanowią metadane. Zostaną one zdefiniowane w klasie Java. Klasa ta, nazwana `BookProviderMetaData`, jest zaprezentowana na listingu 4.5. Najistotniejsze elementy tej klasy zostały zaznaczone pogrubieniem.

**Listing 4.5.** Definiowanie metadanych bazy danych: klasa `BookProviderMetaData`

---

```
public class BookProviderMetaData
{
    public static final String AUTHORITY = "com.androidbook.provider.BookProvider";

    public static final String DATABASE_NAME = "book.db";
    public static final int DATABASE_VERSION = 1;
    public static final String BOOKS_TABLE_NAME = "books";

    private BookProviderMetaData() {}

    // wewnętrzna klasa opisująca obiekt BookTable
    public static final class BookTableMetaData implements BaseColumns
    {
        private BookTableMetaData() {}
        public static final String TABLE_NAME = "books";

        // definicje identyfikatora URI oraz typu MIME
        public static final Uri CONTENT_URI =
            Uri.parse("content://" + AUTHORITY + "/books");

        public static final String CONTENT_TYPE =
```

```
"vnd.android.cursor.dir/vnd.androidbook.book";\n\npublic static final String CONTENT_ITEM_TYPE =\n    "vnd.android.cursor.item/vnd.androidbook.book";\n\npublic static final String DEFAULT_SORT_ORDER = "modified DESC";\n\n// Tu rozpoczynają się dodatkowe kolumny\n// typ string\npublic static final String BOOK_NAME = "name";\n\n// typ string\npublic static final String BOOK_ISBN = "isbn";\n\n// typ string\npublic static final String BOOK_AUTHOR = "author";\n\n// Liczba całkowita z metody System.currentTimeMillis()\npublic static final String CREATED_DATE = "created";\n\n// Liczba całkowita z metody System.currentTimeMillis()\npublic static final String MODIFIED_DATE = "modified";\n}\n}
```

---

Klasa BookProviderMetaDta rozpoczyna działanie od zdefiniowania swojego upoważnienia — przybiera ono postać com.androidbook.provider.BookProvider. Posłuży nam ono do za-rejestrowania dostawcy w pliku manifeście Androida. Ciąg znaków upoważnienia stanowi po-czątkową część identyfikatora URI tego dostawcy.

Następnie klasa ta definiuje jedną tabelę (books) jako wewnętrzną klasę BookTableMetaDta. Klasa BookTableMetaDta tworzy identyfikator URI, potrzebny do rozpoznawania kolekcji ksiązek. Biorąc pod uwagę wspomniane w poprzednim akapicie upoważnienie, identyfikator URI dla kolekcji książek będzie wyglądał następująco:

content://com.androidbook.provider.BookProvider/books

Identyfikator ten jest wskazywany przez stałą:

BookProviderMetaDta.BookTableMetaDta.CONTENT\_URI

Klasa BookProviderMetaDta przechodzi do definiowania typów MIME zbioru książek oraz pojedynczych egzemplarzy. W implementacji dostawcy te stałe posłużą do przekazywania typów MIME przychodzących identyfikatorom URI.

Następnie zostaje zdefiniowany zestaw kolumn: name, isbn, author, created (czas utworzenia) oraz modified (data ostatniej aktualizacji).

**Uwaga!** Typy danych w kolumnach powinny być określone za pomocą komentarzy w kodzie.

Klasa metadanych BookTableMetaDta dziedziczy również elementy klasy BaseColumns, która dostarcza standardowe pole \_id reprezentujące identyfikator wiersza. Mając już przygotowane metadane, jesteśmy gotowi stawić czoła implementacji dostawcy.

## Rozszerzanie klasy ContentProvider

Implementacja naszego przykładu dostawcy treści BookProvider niesie ze sobą konieczność rozszerzenia klasy ContentProvider oraz przesłonięcia metody `onCreate()` w celu utworzenia bazy danych, a następnie wprowadzenia metod `query`, `insert`, `update`, `delete` i `getType`. W tym podpunkcie omówimy proces konfigurowania oraz tworzenia bazy danych, natomiast nieco dalej przedstawimy kolejno metody `query`, `insert`, `update`, `delete` i `getType`. Na listingu 4.6 został zaprezentowany pełny kod źródłowy tej klasy. Najważniejsze sekcje zostały zaznaczone pogrubieniem.

Metoda `query` żąda określonego przez nią zestawu kolumn. Przypomina to klauzulę `select`, która żąda nazw kolumn wraz z ich odpowiednikami `as` (czasami zwanyimi **synonimami**). Android wykorzystuje obiekt `map`, który wywołuje mapę projekcji, reprezentującą nazwy tych kolumn oraz ich synonimy. Musimy skonfigurować taką mapę, żeby wprowadzić ją w późniejszej implementacji metody `query`. W kodzie implementacji dostawcy (listing 4.6) odpowiedzialny za to fragment został umieszczony na początku jako część konfiguracji mapy projektu.

Większość implementowanych przez nas metod pobiera identyfikatory URI w postaci danych wejściowych. Chociaż wszystkie identyfikatory obsługiwane przez tego dostawcę posiadają taki sam początek adresu, jego końcówka będzie w każdym identyfikatorze inna — tak samo jak w przypadku strony WWW. Każdy identyfikator musi różnić się końcowymi członami w celu określania różnych danych lub dokumentów. Zilustrujemy to przykładem:

```
Uri1: content://com.androidbook.provider.BookProvider/books
Uri2: content://com.androidbook.provider.BookProvider/books/12
```

Zauważmy, w jaki sposób dostawca treści BookProvider rozróżnia poszczególne identyfikatory. Zaprezentowaliśmy prosty przypadek. Gdyby nasz dostawca przechowywał nie tylko książki, zostałyby utworzone identyfikatory również dla tych obiektów.

Implementacja dostawcy potrzebuje mechanizmu umożliwiającego odróżnianie poszczególnych identyfikatorów URI; w tym celu została wprowadzona klasa `UriMatcher`. Musimy więc skonfigurować ten obiekt wraz ze wszystkimi wariacjami identyfikatora URI. Kod ten został umieszczony na listingu 4.6 zaraz po fragmencie definiującym mapę. Klasa `UriMatcher` zostanie dokładniej omówiona w podpunkcie „*Stosowanie klasy UriMatcher do rozpoznawania identyfikatorów URI*”.

Kod z listingu 4.6 powoduje następnie przesłonięcie metody `onCreate()`, co ułatwia utworzenie bazy danych. Kod źródłowy implementuje następnie metody `insert()`, `query()`, `update()`, `getType()` oraz `delete()`. Cały kod został zaprezentowany na jednym listingu, jednak jego poszczególne aspekty zostaną omówione w oddzielnych podpunktach.

### **Listing 4.6.** Implementacja dostawcy treści BookProvider

---

```
public class BookProvider extends ContentProvider
{
    // Dolicza pomocniczy znacznik. Nie ma znaczenia dla dostawców.
    private static final String TAG = "BookProvider";

    // Konfiguruje mapę projekcji.
    // Mapy projekcji są podobne do aliasu kolumny „as” w języku sql,
    // za pomocą której można zmieniać nazwy kolumn.
    private static HashMap<String, String> sBooksProjectionMap;
```

```

static
{
    sBooksProjectionMap = new HashMap<String, String>();
    sBooksProjectionMap.put(BookTableMetaData._ID, BookTableMetaData._ID);

    // kolumny name, isbn, author
    sBooksProjectionMap.put(BookTableMetaData.BOOK_NAME
                           , BookTableMetaData.BOOK_NAME);
    sBooksProjectionMap.put(BookTableMetaData.BOOK_ISBN
                           , BookTableMetaData.BOOK_ISBN);
    sBooksProjectionMap.put(BookTableMetaData.BOOK_AUTHOR
                           , BookTableMetaData.BOOK_AUTHOR);

    // kolumny created, modified
    sBooksProjectionMap.put(BookTableMetaData.CREATED_DATE
                           , BookTableMetaData.CREATED_DATE);
    sBooksProjectionMap.put(BookTableMetaData.MODIFIED_DATE
                           , BookTableMetaData.MODIFIED_DATE);
}

// Konfiguruje identyfikatory URI.
// Mechanizm umożliwiający identyfikowanie wzorców wszystkich przychodzących
// identyfikatorów URI.
private static final UriMatcher sUriMatcher;
private static final int INCOMING_BOOK_COLLECTION_URI_INDICATOR = 1;
private static final int INCOMING_SINGLE_BOOK_URI_INDICATOR = 2;
static {
    sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    sUriMatcher.addURI(BookProviderMetaData.AUTHORITY
                       , "books"
                       , INCOMING_BOOK_COLLECTION_URI_INDICATOR);
    sUriMatcher.addURI(BookProviderMetaData.AUTHORITY
                       , "books/#",
                       INCOMING_SINGLE_BOOK_URI_INDICATOR);
}

/**
 * Konfiguracja/tworzenie bazy danych.
 * Klasa ta pomaga otwierać, tworzyć i aktualizować plik bazy danych.
 */
private static class DatabaseHelper extends SQLiteOpenHelper {

    DatabaseHelper(Context context) {
        super(context, BookProviderMetaData.DATABASE_NAME, null
              , BookProviderMetaData.DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        Log.d(TAG, "wywołana wewnętrzna metoda oncreate");
        db.execSQL("CREATE TABLE " + BookTableMetaData.TABLE_NAME + " (" +
                  + BookProviderMetaData._ID + " INTEGER PRIMARY KEY," +
                  + BookTableMetaData.BOOK_NAME + " TEXT," +
                  + BookTableMetaData.BOOK_ISBN + " TEXT," +
                  + BookTableMetaData.BOOK_AUTHOR + " TEXT," +

```

```
+ BookTableMetaData.CREATED_DATE + " INTEGER,"  
+ BookTableMetaData.MODIFIED_DATE + " INTEGER"  
+ ");");  
}  
  
@Override  
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
    Log.d(TAG, "wywolana wewnetrzna metoda onupgrade");  
    Log.w(TAG, "Aktualizacja bazy danych z wersji " + oldVersion + " do wersji "  
        + newVersion + ", w wyniku ktorej wszystkie stare dane zostana usuniete");  
    db.execSQL("DROP TABLE IF EXISTS " + BookTableMetaData.TABLE_NAME);  
    onCreate(db);  
}  
}  
  
private DatabaseHelper mOpenHelper;  
  
// Zajmuje się kwestią wywoływanego metody OnCreate.  
  
@Override  
public boolean onCreate() {  
    Log.d(TAG, "wywolana glowna metoda onCreate");  
    mOpenHelper = new DatabaseHelper(getContext());  
    return true;  
}  
  
@Override  
public Cursor query(Uri uri, String[] projection, String selection  
                    , String[] selectionArgs, String sortOrder)  
{  
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();  
  
    switch (sUriMatcher.match(uri))  
    {  
        case INCOMING_BOOK_COLLECTION_URI_INDICATOR:  
            qb.setTables(BookTableMetaData.TABLE_NAME);  
            qb.setProjectionMap(sBooksProjectionMap);  
            break;  
  
        case INCOMING_SINGLE_BOOK_URI_INDICATOR:  
            qb.setTables(BookTableMetaData.TABLE_NAME);  
            qb.setProjectionMap(sBooksProjectionMap);  
            qb.appendWhere(BookTableMetaData._ID + "="  
                           + uri.getPathSegments().get(1));  
            break;  
  
        default:  
            throw new IllegalArgumentException("Nieznany ident. URI" + uri);  
    }  
  
    // Jeżeli kolejność sortowania nie jest określona, należy skorzystać  
    // z domyślnej wartości.  
    String orderBy;  
    if (TextUtils.isEmpty(sortOrder)) {  
        orderBy = BookTableMetaData.DEFAULT_SORT_ORDER;  
    } else {
```

```
        orderBy = sortOrder;
    }

    // Otwiera bazę danych i uruchamia kwerendę.
    SQLiteDatabase db =
        mOpenHelper.getReadableDatabase();
    Cursor c = qb.query(db, projection, selection,
                        selectionArgs, null, null, orderBy);

    // Przykładowy sposób zliczania.
    int i = c.getCount();

    // Informuje kursor, który identyfikator URI ma być obserwowany
    // na wypadek zmiany źródła danych.
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}

@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
    case INCOMING_BOOK_COLLECTION_URI_INDICATOR:
        return BookTableMetaData.CONTENT_TYPE;

    case INCOMING_SINGLE_BOOK_URI_INDICATOR:
        return BookTableMetaData.CONTENT_ITEM_TYPE;

    default:
        throw new IllegalArgumentException("Nieznany ident. URI " + uri);
    }
}

@Override
public Uri insert(Uri uri, ContentValues initialValues) {
    // Sprawdza żądany identyfikator Uri.
    if (sUriMatcher.match(uri) != INCOMING_BOOK_COLLECTION_URI_INDICATOR) {
        throw new IllegalArgumentException("Nieznany ident. URI " + uri);
    }

    ContentValues values;
    if (initialValues != null) {
        values = new ContentValues(initialValues);
    } else {
        values = new ContentValues();
    }

    Long now = Long.valueOf(System.currentTimeMillis());

    // Należy się upewnić, że wszystkie są skonfigurowane.
    if (values.containsKey(BookTableMetaData.CREATED_DATE) == false) {
        values.put(BookTableMetaData.CREATED_DATE, now);
    }

    if (values.containsKey(BookTableMetaData.MODIFIED_DATE) == false) {
        values.put(BookTableMetaData.MODIFIED_DATE, now);
    }
}
```

```

if (values.containsKey(BookTableMetaData.BOOK_NAME) == false) {
    throw new SQLException(
        "Nieudana próba wstawienia wiersza z powodu braku nazwy książki " + uri);
}

if (values.containsKey(BookTableMetaData.BOOK_ISBN) == false) {
    values.put(BookTableMetaData.BOOK_ISBN, "Nieznany numer ISBN");
}
if (values.containsKey(BookTableMetaData.BOOK_AUTHOR) == false) {
    values.put(BookTableMetaData.BOOK_ISBN, "Nieznany autor");
}

SQLiteDatabase db = mOpenHelper.getWritableDatabase();
long rowId = db.insert(BookTableMetaData.TABLE_NAME
                      , BookTableMetaData.BOOK_NAME, values);
if (rowId > 0) {
    Uri insertedBookUri = ContentUris.withAppendedId(
        BookTableMetaData.CONTENT_URI, rowId);
    getContext().getContentResolver().notifyChange(insertedBookUri, null);
    return insertedBookUri;
}

throw new SQLException("Nieudana próba umieszczenia wiersza w " + uri);
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case INCOMING_BOOK_COLLECTION_URI_INDICATOR:
            count = db.delete(BookTableMetaData.TABLE_NAME, where, whereArgs);
            break;

        case INCOMING_SINGLE_BOOK_URI_INDICATOR:
            String rowId = uri.getPathSegments().get(1);
            count = db.delete(BookTableMetaData.TABLE_NAME
                , BookTableMetaData._ID + "=" + rowId
                + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : ""))
                , whereArgs);
            break;

        default:
            throw new IllegalArgumentException("Nieznany ident. URI " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}

@Override
public int update(Uri uri, ContentValues values, String where, String[] whereArgs)
{
    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case INCOMING_BOOK_COLLECTION_URI_INDICATOR:
            count = db.update(BookTableMetaData.TABLE_NAME,
                values, where, whereArgs);

```

```
        break;

    case INCOMING_SINGLE_BOOK_URI_INDICATOR:
        String rowId = uri.getPathSegments().get(1);
        count = db.update(BookTableMetaData.TABLE_NAME
            , values
            , BookTableMetaData._ID + "=" + rowId
            + (!TextUtils.isEmpty(where) ? " AND (" + where + ')' : "") 
            , whereArgs);
        break;

    default:
        throw new IllegalArgumentException("Nieznany ident. URI " + uri);
    }

    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
}
```

---

## Wypełnianie kontraktów typów MIME

Dostawca treści BookProvider musi posiadać także zaimplementowaną metodę `getType()`, przekazującą typ MIME danego identyfikatora URI. Podobnie jak w przypadku innych metod dostawcy treści jest ona przeciążona w odniesieniu do nadchodzących identyfikatorów URI. W ten sposób jej zadaniem jest rozróżnianie typów identyfikatorów URI: czy dany typ określa kolekcję książek, czy też tylko jeden egzemplarz.

Jak wcześniej wspomnieliśmy, zastosujemy klasę `UriMatcher` do określenia typu identyfikatora URI. Klasa `BookTableMetaData` posiada zdefiniowane stałe, przekazywane w zależności od identyfikatora URI. Listing 4.6 przedstawia implementację tej metody.

## Implementowanie metody query

W dostawcy treści metoda `query` przekazuje zbiór wierszy, w zależności od przychodzących identyfikatorów URI oraz klauzuli `WHERE`.

Również ta metoda wykorzystuje klasę `UriMatcher` do rozpoznawania typu identyfikatora URI. Jeżeli typ identyfikatora URI odpowiada pojedynczemu elementowi, metoda ta otrzymuje kod reprezentujący książkę w następujący sposób:

1. Wydobywa segmenty ścieżki za pomocą metody `getPathSegments()`.
2. Numeruje segmenty identyfikatora URI w celu znalezienia pierwszej części ścieżki, która jest identyfikatorem książki.

Metoda `query` wykorzystuje następnie utworzone na początku listingu 4.6 mapy do identyfikowania otrzymywanych kolumn. Ostatecznie kursor jest przekazywany programowi żądającemu. W trakcie tego procesu metoda `query` wykorzystuje klasę `SQLiteQueryBuilder` do sformułowania oraz wykonania kwerendy (listing 4.6).

## Implementowanie metody insert

Zadaniem metody `insert` w dostawcy treści jest wstawianie rekordu do bazy danych oraz przekazywanie identyfikatora URI wskazującego ten rekord.

Także w tym przypadku metoda `UriMatcher` służy do identyfikacji adresów URI. Kod sprawdza najpierw, czy identyfikator ten wskazuje właściwy typ danych. Jeśli nie, zostaje wyświetlony wyjątek (listing 4.6).

Następnie zostają zweryfikowane obowiązkowe oraz opcjonalne parametry kolumn. Jeżeli nie zostały zdefiniowane wartości dla niektórych kolumn, mogą zostać wstawione domyślne.

W dalszej kolejności zostaje zastosowana klasa `SQLiteDataBase` do wstawienia rekordu oraz utworzenia jego identyfikatora. Na końcu zostaje skonstruowany adres URI na podstawie zwróconego identyfikatora z bazy danych.

## Implementowanie metody update

W dostawcy treści metoda `update` aktualizuje rekord (lub rekordy) na podstawie przekazanych wartości kolumny, jak i klauzuli `WHERE`. Przekazuje ona liczbę zaktualizowanych krotek.

Podobnie jak w pozostałych metodach, także i tutaj klasa `UriMatcher` służy do rozpoznawania identyfikatorów URI. Jeżeli identyfikator ten wskazuje zbiór danych, następuje wykonanie klauzuli `WHERE`, dzięki czemu zostanie zaktualizowana jak największa liczba rekordów. Jeżeli identyfikator URI określa pojedynczy egzemplarz, otrzymujemy identyfikator danej książki, zdefiniowany jako dodatkowa klauzula `WHERE`. Na końcu kod wyświetla liczbę uaktualnionych rekordów (listing 4.6). W rozdziale 21. zostaną gruntownie przeanalizowane konsekwencje użytej tu metody `notifyChange`. Należy zwrócić uwagę, w jaki sposób metoda ta pozwala ogłosić światu, że dane obecne pod określonym adresem URI zostały zaktualizowane. Teoretycznie można zastosować taką samą technikę przy metodzie `insert`, stwierdzając, że katalog `.../books` został zmieniony po wstawieniu rekordu.

## Implementowanie metody delete

Metoda `delete` w dostawcy treści służy do usuwania rekordu (lub rekordów) na podstawie przekazywanej klauzuli `WHERE`. Następnie otrzymujemy liczbę usuniętych wierszy.

Tak samo jak w przypadku metod omówionych w poprzednich podpunktach, do określania typu identyfikatora URI służy metoda `UriMatcher`. Jeżeli identyfikator ten wskazuje zbiór danych, wykorzystywana jest klauzula `WHERE`, dzięki czemu usuwana jest jak największa liczba rekordów. Jeżeli wartość tej klauzuli będzie wynosić `null`, zostaną usunięte wszystkie rekordy. Jeżeli identyfikator URI określa pojedynczy egzemplarz, otrzymujemy identyfikator danej książki, zdefiniowany jako dodatkowa klauzula `WHERE`. Na końcu kod wyświetla liczbę usuniętych rekordów (listing 4.6).

## Stosowanie klasy UriMatcher do rozpoznawania identyfikatorów URI

Wspomnieliśmy kilkakrotnie o klasie `UriMatcher`, a zatem przyjrzyjmy się jej dokładniej. Niemal wszystkie metody w dostawcy treści są przeciążone w odniesieniu do rodzajów identyfikatorów URI. Na przykład do odczytania pojedynczego egzemplarza książki i całej listy książek jest używana ta sama metoda `query()`. Metoda ta musi znać rodzaj wywoływanego identyfikatora URI. Klasa `UriMatcher` jest pomocna w identyfikacji typów adresów URI.

Działa to następująco: instancja klasy UriMatcher zostaje powiadomiona o oczekiwanych wzorach typów identyfikatora URI. Każdy wzorzec otrzyma również niepowtarzalny numer. Po zarejestrowaniu tych wzorców klasa UriMatcher będzie sprawdzała, czy przychodzące identyfikatory im odpowiadają.

Jak już wcześniej stwierdziliśmy, nasz dostawca treści BookProvider zawiera dwa wzorce identyfikatorów URI: jeden dla kolekcji książek, a drugi dla pojedynczych egzemplarzy. Kod przedstawiony na listingu 4.7 rejestruje obydwa wzorce za pomocą klasy UriMatcher. Zostaje przypisana wartość 1 dla kolekcji książek i wartość równa 2 dla pojedynczego egzemplarza (same wzorce identyfikatorów URI zostają zdefiniowane w metadanych tabeli books).

---

**Listing 4.7.** Rejestrowanie wzorców identyfikatorów URI za pomocą klasy UriMatcher

```
private static final UriMatcher sUriMatcher;
// Definiuje identyfikatory dla każdego typu adresu uri
private static final int INCOMING_BOOK_COLLECTION_URI_INDICATOR = 1;
private static final int INCOMING_SINGLE_BOOK_URI_INDICATOR = 2;

static {
    sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    // Rejestruje wzorzec dla kolekcji książek
    sUriMatcher.addURI(BookProviderMetaData.AUTHORITY
        , "books"
        , INCOMING_BOOK_COLLECTION_URI_INDICATOR);
    // Rejestruje wzorzec dla jednej książki
    sUriMatcher.addURI(BookProviderMetaData.AUTHORITY
        , "books/#",
        INCOMING_SINGLE_BOOK_URI_INDICATOR);
}
```

---

Po utworzeniu rejestracji można zobaczyć, jaką rolę odgrywa klasa UriMatcher w implementacji metody query:

```
switch (sUriMatcher.match(uri)) {
    case INCOMING_BOOK_COLLECTION_URI_INDICATOR:
    ...
    case INCOMING_SINGLE_BOOK_URI_INDICATOR:
    ...
    default:
        throw new IllegalArgumentException("Nieznany ident. URI " + uri);
}
```

Zwróćmy uwagę na sposób przekazywania przez metodę match tej samej liczby, która została wcześniej zarejestrowana. Konstruktor klasy UriMatcher wykorzystuje liczbę całkowitą względem głównego identyfikatora URI. UriMatcher przekazuje tę wartość, w przypadku gdy w adresie URL nie ma ani segmentów ścieżki, ani upoważnień. Wartość NO\_MATCH jest zwracana również w przypadku niepasujących wzorców identyfikatorów. Można utworzyć klasę UriMatcher nieposiadającą wzorców dopasowania; w takim przypadku Android inicjalizuje wewnętrznie wartość NO\_MATCH w klasie UriMatcher. Kod z listingu 4.7 można zatem napisać również w następujący sposób:

```
static {
    sUriMatcher = new UriMatcher();
    sUriMatcher.addURI(BookProviderMetaData.AUTHORITY
```

```

        , "books"
        , INCOMING_BOOK_COLLECTION_URI_INDICATOR);

sUriMatcher.addURI(BookProviderMetaData.AUTHORITY
                    , "books/#",
                    INCOMING_SINGLE_BOOK_URI_INDICATOR);
}

```

## Korzystanie z map projekcji

Dostawca treści pełni rolę pośrednika pomiędzy abstrakcyjnym a rzeczywistym zestawem kolumn w bazie danych, mimo to zestawy te mogą się różnić. Podczas tworzenia kwerend potrzebna jest mapa łącząca określane przez klienta kolumny klauzuli WHERE z rzeczywistymi bazodanowymi kolumnami. Takie **mapy projekcji** są konfigurowane za pomocą klasy `SQLiteQueryBuilder`.

Poniżej przytaczamy informacje zawarte w dokumentacji Android SDK, traktujące o metody mapowania `public void setProjectionMap(Map columnMap)` dostępnej w klasie `QueryBuilder`:

*Klasa ta ustanawia mapę projekcji dla kwerendy. Mapa projekcji wiąże nazwy kolumn przekazywane kwerendzie przez program żądający z bazodanowymi nazwami kolumn. Jest to funkcja przydatna podczas zmieniania nazw kolumn, jak i służąca do usuwania niejasności w nazwach kolumn podczas wykonywania operacji łączenia. Można na przykład połączyć nazwę name z people.name. Jeżeli mapa projekcji jest konfigurowana, musi zawierać nazwy wszystkich kolumn, które mogą być wywoływane przez użytkownika, nawet jeśli klucz i wartość są takie same.*

Nasz dostawca treści `BookProvider` konfiguruje mapę projekcji w następujący sposób:

```

sBooksProjectionMap = new HashMap<String, String>();
sBooksProjectionMap.put(BookTableMetaData._ID, BookTableMetaData._ID);

// kolumny name, isbn, author
sBooksProjectionMap.put(BookTableMetaData.BOOK_NAME
                        , BookTableMetaData.BOOK_NAME);
sBooksProjectionMap.put(BookTableMetaData.BOOK_ISBN
                        , BookTableMetaData.BOOK_ISBN);
sBooksProjectionMap.put(BookTableMetaData.BOOK_AUTHOR
                        , BookTableMetaData.BOOK_AUTHOR);

// kolumny created, modified
sBooksProjectionMap.put(BookTableMetaData.CREATED_DATE
                        , BookTableMetaData.CREATED_DATE);
sBooksProjectionMap.put(BookTableMetaData.MODIFIED_DATE
                        , BookTableMetaData.MODIFIED_DATE);

```

Teraz konstruktor kwerendy stosuje zmienną `sBooksProjectionMap` w następujący sposób:

```

queryBuilder.setTables(BookTableMetaData_TABLE_NAME);
queryBuilder.setProjectionMap(sNotesProjectionMap);

```

## Rejestrowanie dostawcy

Ostatnim etapem jest rejestracja dostawcy treści w pliku `AndroidManifest.xml` za pomocą struktury ukazanej na listingu 4.8:

**Listing 4.8.** Rejestrowanie dostawcy

```
<provider android:name=".BookProvider"  
    android:authorities="com.androidbook.provider.BookProvider "/>
```

---

## Testowanie dostawcy BookProvider

Skoro już otrzymaliśmy dostawcę treści BookProvider, zaprezentujemy przykładowy kod, pozwalający na przetestowanie koncepcji dostawcy treści. Przyjrzymy się procesom dodawania oraz usuwania informacji o książce, zliczania liczby książek oraz wyświetlania wszystkich książek.

Należy pamiętać, że są to tylko wycinki z przykładowego projektu i jako takie nie dadzą się skompilować, gdyż brakuje tu kilku dodatkowych plików, niezbędnych do działania projektu. Sądzimy jednak, że te przykładowe fragmenty kodu mogą się okazać cenne podczas demonstrowania uprzednio omawianych koncepcji.

Na końcu rozdziału zamieściliśmy odnośnik do przykładowego projektu, który można pobrać oraz przetestować w środowisku Eclipse.

## Dodawanie książki

Kod z listingu 4.9 powoduje wstawienie nowej książki do bazy danych.

**Listing 4.9.** Testowanie funkcji dodawania za pomocą dostawcy treści

```
public void addBook(Context context)  
{  
    String tag = "Testowanie dostawcy BookProvider";  
    Log.d(tag,"Dodawanie ksiazki");  
    ContentValues cv = new ContentValues();  
    cv.put(BookProviderMetaData.BookTableMetaData.BOOK_NAME, "book1");  
    cv.put(BookProviderMetaData.BookTableMetaData.BOOK_ISBN, "isbn-1");  
    cv.put(BookProviderMetaData.BookTableMetaData.BOOK_AUTHOR, "author-1");  
  
    ContentResolver cr = context.getContentResolver();  
    Uri uri = BookProviderMetaData.BookTableMetaData.CONTENT_URI;  
    Log.d(tag,"identyfikator uri wstawianej ksiazki:" + uri);  
    Uri insertedUri = cr.insert(uri, cv);  
    Log.d(tag,"identyfikator Uri wstawiania:" + insertedUri);  
}
```

---

## Usuwanie książki

Kod widoczny na listingu 4.10 powoduje usunięcie ostatniego rekordu z książkowej bazy danych. Z kolei listing 4.11 ukazuje przykład działania metody getCount() widocznej w poniższym kodzie.

**Listing 4.10.** Testowanie funkcji usuwania za pomocą dostawcy treści

---

```
public void removeBook(Context context)
{
    String tag = "Testowanie dostawcy BookProvider";
    int i = getCount(context); //Spójrzmy na funkcję getCount z listingu 4.11
    ContentResolver cr = context.getContentResolver();
    Uri uri = BookProviderMetaData.BookTableMetaData.CONTENT_URI;
    Uri delUri = Uri.withAppendedPath(uri, Integer.toString(i));
    Log.d(tag, "Identyfikator Uri usuwania:" + delUri);
    cr.delete(delUri, null, null);
    Log.d(tag, "Nowa zliczona wartość:" + getCount(context));
}
```

---

Zauważmy, że mamy tu do czynienia z krótkim przykładem, pokazującym mechanizm usuwania za pomocą identyfikatora URI. Algorytm uzyskiwania ostatniego identyfikatora URI nie zawsze musi być stuoprocentowo skuteczny. Powinien jednak poprawnie działać w przypadku dodawania pięciu rekordów, a następnie usuwania ich od końca, jednego po drugim. W typowej aplikacji dążylibyśmy do wyświetlenia rekordów w formie listy oraz poproszenia użytkownika o zaznaczenie rekordu przeznaczonego do usunięcia. W ten sposób uzyskalibyśmy dokładny identyfikator tego rekordu.

## Zliczanie książek

Kod z listingu 4.11 pobiera kursor i zlicza zawarte w nim rekordy.

**Listing 4.11.** Zliczanie rekordów w tabeli

---

```
private int getCount(Context context)
{
    Uri uri = BookProviderMetaData.BookTableMetaData.CONTENT_URI;
    Activity a = (Activity)context;
    Cursor c = a.managedQuery(uri,
        null, //projekcja
        null, //ciąg znaków wyboru
        null, //argumenty tablicy zawierającej ciągi znaków wyboru
        null); // kolejność sortowania
    int numberofRecords = c.getCount();
    c.close();
    return numberofRecords;
}
```

---

## Wyświetlanie listy książek

Z pomocą kodu z listingu 4.12 pobierane są wszystkie rekordy z bazy książek.

**Listing 4.12.** Wyświetlanie listy książek

---

```
public void showBooks(Context context)
{
    String tag = "Testowanie dostawcy BookProvider";
    Uri uri = BookProviderMetaData.BookTableMetaData.CONTENT_URI;
```

```
Activity a = (Activity)context;
Cursor c = a.managedQuery(uri,
    null, //projekcja
    null, //ciąg znaków wyboru
    null, //argumenty tablicy ciągów znaków wyboru
    null); //kolejność sortowania

int iName = c.getColumnIndex(
    BookProviderMetaData.BookTableMetaData.BOOK_NAME);

int iIsbn = c.getColumnIndex(
    BookProviderMetaData.BookTableMetaData.BOOK_ISBN);
int iAuthor = c.getColumnIndex(
    BookProviderMetaData.BookTableMetaData.BOOK_AUTHOR);

//Raportowanie indeksów
Log.d(tag,"name,isbn,author:" + iName + iIsbn + iAuthor);

//przechodzi po wierszach w oparciu o indeksy
for(c.moveToFirst();!c.isAfterLast();c.moveToNext())
{
    //Zbiera wartości
    String id = c.getString(1);
    String name = c.getString(iName);
    String isbn = c.getString(iIsbn);
    String author = c.getString(iAuthor);

    //Raportuje wiersz lub zapisuje go do dziennika
    StringBuffer cbuf = new StringBuffer(id);
    cbuf.append(",").append(name);
    cbuf.append(",").append(isbn);
    cbuf.append(",").append(author);
    Log.d(tag, cbuf.toString());
}

//Raportuje liczbę przeczytanych wierszy
int numberofRecords = c.getCount(context);
Log.d(tag,"Ilosc rekordow:" + numberofRecords);

//Zamyka kursor
//W idealnym przypadku powinno to zostać wykonane w
//bloku finally
c.close();
}
```

---

## Odbośniki

Poniżej przedstawiamy kilka dodatkowych zasobów, które mogą pomóc w zdobywaniu wiedzy na tematy przedstawione w tym rozdziale:

- <http://developer.android.com/guide/topics/providers/content-providers.html>
  - znajdziemy tu dokumentację dotyczącą dostawców treści.

- <http://developer.android.com/reference/android/content/ContentProvider.html> — zamieszczono tutaj opis interfejsu API klasy ContentProvider, z którego możemy się dowiedzieć o kontraktach tej klasy.
- <http://developer.android.com/reference/android/content/UriMatcher.html> — ten adres URL prowadzi do strony zawierającej użyteczne informacje o klasie UriMatcher.
- <http://developer.android.com/reference/android/database/Cursor.html> — dzięki informacjom tu zawartym nauczymy się odczytywać informacje z dostawcy treści lub bezpośrednio z bazy danych.
- <http://www.sqlite.org/sqlite.html> — strona domowa silnika SQLite, na której znajdziemy wiele informacji na jego temat oraz narzędzi pozwalających na pracę z bazami danych SQLite.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — z tego adresu możemy pobrać testowy projekt stworzony specjalnie z myślą o niniejszym rozdziale. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R04\_DostawcyTreści*.

## Podsumowanie

W niniejszym rozdziale opisaliśmy najważniejsze kwestie dotyczące identyfikatorów URI, typów MIME oraz dostawców treści. Nauczyliśmy się wykorzystywać bazę danych SQLite do tworzenia dostawców związanych z identyfikatorami URI. Kiedy dane zostaną w taki sposób odsłonięte, każda aplikacja systemu Android może je wykorzystać.

Taka zdolność uzyskiwania dostępu do danych oraz ich aktualizowania za pomocą identyfikatorów URI, bez względu na granice procesów, wpisuje się znakomicie w technologię środowiska zorientowanego na usługi oraz przetwarzanie rozproszone, co opisaliśmy w rozdziale 1.

W kolejnym rozdziale omówimy pojęcie intencji, które poprzez identyfikatory URI danych oraz identyfikatory URI typu MIME są związane z dostawcami treści (jak również z innymi składnikami Androida). Wiedza zdobyta w tym rozdziale bardzo się przyda do zrozumienia zagadnienia intencji — identyfikatory URI danych odgrywają tu kluczową rolę.



# Intencje

W Androidzie wprowadzono pojęcie *intencji*. Intencje służą do przywoływania składników, do których zaliczamy w Androidzie aktywności (składniki interfejsu użytkownika), usługi (kod przetwarzany w tle), odbiorców komunikatów (kod generujący odpowiedzi na nadawane wiadomości) oraz dostawców treści (kod, który wychwytuje dane).

## Podstawowe informacje na temat intencji

Chociaż pod pojęciem **intencji** najczęściej rozumiemy mechanizm umożliwiający przywoływanie składników, z pojęciem tym wiąże się kilka koncepcji. Intencje wykorzystuje się do wywołania zewnętrznych aplikacji z poziomu innej aplikacji. Służą do wywoływania wewnętrznych lub zewnętrznych składników danej aplikacji. Można dzięki nim generować zdarzenia, na które mogą odpowiadać inne elementy o podobnym modelu publikowania i subskrybowania. Dzięki intencjom można również generować alerty.

**Uwaga!**

Czym jest intencja? Najkrótsza odpowiedź jest taka: intencją nazywamy akcję oraz powiązane z nią dane.

Na najprostszym poziomie intencja jest działaniem, które Android może przeprowadzić lub **przywołać**. Przywoływane działanie zależy od tego, co jest dla niego zarejestrowane. Wyobraźmy sobie następującą aktywność:

```
public class BasicViewActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.some-view);
    }
} // klasa eof
```

Układ graficzny some-view musi wskazywać odpowiedni plik w katalogu `/res/layout`. Android pozwala następnie na zarejestrowanie tej aktywności w pliku manifeście, dzięki czemu będzie wywoływana przez inne aplikacje. Kod rejestracji został zaprezentowany poniżej:

```
<activity android:name="BasicViewActivity"
          android:label="Testy podstawowego widoku">
    <intent-filter>
        <action android:name="com.androidbook.intent.action.ShowBasicView"/>
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

W powyższym procesie rejestracji bierze udział nie tylko aktywność, lecz również działanie, dzięki któremu można tę aktywność wywołać. Projektant aktywności przeważnie wybiera nazwę dla działania i definiuje je jako część filtra intencji dla tej aktywności. W dalszej części rozdziału pojawi się więcej informacji na temat filtrów intencji.

Po zdefiniowaniu aktywności oraz jej zarejestrowaniu wobec działania można użyć intencji do wywołania klasy BasicViewActivity:

```
public static invokeMyApplication(Activity parentActivity)
{
    String actionName= " com.androidbook.intent.action.ShowBasicView ";
    Intent intent = new Intent(actionName);
    parentActivity.startActivity(intent);
}
```

#### Uwaga!

Ogólna konwencja nazewnictwa działania wygląda następująco:  
`<nazwa-pakietu>intent.action.NAZWA_DZIAŁANIA`.

Po przywołaniu aktywności BasicViewActivity posiada ona zdolność do rozpoznania wywołującej ją intencji. Poniżej przedstawiono kod tej aktywności zmodyfikowany w taki sposób, aby została wczytana intencja wywołująca:

```
public class BasicViewActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.some_view);
        Intent intent = this.getIntent();
        if (intent == null)
        {
            Log.d("test tag", "Ta aktywnosc jest wywolywana bez uzycia intencji");
        }
    }
}//klaśa eof
```

## Intencje dostępne w Androidzie

Możemy przetestować intencje w jednej z aplikacji dołączonych do Androida. Na stronie <http://developer.android.com/guide/appendix/g-app-intents.html> została zamieszczona lista niektórych dostępnych aplikacji oraz wywołujących je intencji.

**Uwaga!**

Należy jednak pamiętać, że ta lista może ulegać zmianie w zależności od wersji systemu Android.

Wśród dostępnych aplikacji mogą się znajdować następujące:

- aplikacja przeglądarki stron WWW;
- aplikacja umożliwiająca połączenie telefoniczne;
- aplikacja wyświetlająca klawiaturę wpisywania numeru, umożliwiająca ręczne wpisanie numeru telefonicznego i wykonanie połączenia za pośrednictwem interfejsu użytkownika;
- aplikacja przedstawiająca mapę świata oraz wskazane współrzędne długości i szerokości geograficznej;
- aplikacja zawierająca szczegółowe mapy i wyświetlająca zdjęcia ulic z serwisu Google.

Na listingu 5.1 zaprezentowano kod pozwalający na przywołanie powyższych aplikacji za pomocą ich opublikowanych intencji.

**Listing 5.1.** Korzystanie z aplikacji wbudowanych w Androida

```
public class IntentsUtils
{
    public static void invokeWebBrowser(Activity activity)
    {
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse("http://www.google.com"));
        activity.startActivity(intent);
    }
    public static void invokeWebSearch(Activity activity)
    {
        Intent intent = new Intent(Intent.ACTION_WEB_SEARCH);
        intent.setData(Uri.parse("http://www.google.com"));
        activity.startActivity(intent);
    }
    public static void dial(Activity activity)
    {
        Intent intent = new Intent(Intent.ACTION_DIAL);
        activity.startActivity(intent);
    }
    public static void call(Activity activity)
    {
        Intent intent = new Intent(Intent.ACTION_CALL);
        intent.setData(Uri.parse("tel:555-555-5555"));
        activity.startActivity(intent);
    }
    public static void showMapAtLatLong(Activity activity)
    {
        Intent intent = new Intent(Intent.ACTION_VIEW);
        //geo:lat,long?z=zoomlevel&q=question-string
        intent.setData(Uri.parse("geo:0,0?z=4&q=business+near+city"));
        activity.startActivity(intent);
    }
}
```

```
public static void tryOneOfThese(Activity activity)
{
    IntentsUtils.invokeWebBrowser(activity);
}
```

---

Taki kod jest przydatny w przypadku prostej aktywności, która zawiera element menu przywołujący metodę `tryOneOfThese(activity)`. Utworzenie prostego menu jest banalne (listing 5.2).

---

**Listing 5.2.** Środowisko testowe służące do zbudowania prostego menu

---

```
public class MainActivity extends Activity
{
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        TextView tv = new TextView(this);
        tv.setText("Witaj, Androidzie. Przywitaj się");
        setContentView(tv);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        int base=Menu.FIRST; // wartość wynosi 1
        MenuItem item1 = menu.add(base,base,base,"Test");
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == 1) {
            IntentUtils.tryOneOfThese(this);
        }
        else {
            return super.onOptionsItemSelected(item);
        }
        return true;
    }
}
```

---

**Uwaga!**

W rozdziale 2. zostały zawarte informacje dotyczące utworzenia z tych plików projektu Android, skompilowania go i uruchomienia. Na początku rozdziału 7. „Praca z menu”, pokazaliśmy natomiast kod pozwalający na utworzenie menu. Można również pobrać przykładowy kod, napisany specjalnie z myślą o tym rozdziale, do którego odnośnik znajduje się na końcu tego rozdziału. Podstawowa aktywność zawarta w tym kodzie może się nieznacznie różnić od opisywanej w książce, jednak sama koncepcja nie ulega zmianie. W tym przykładowym kodzie wczytujemy również menu z pliku XML.

## Przegląd struktury intencji

Kolejnym świetnym sposobem zrozumienia koncepcji intencji jest poznanie składników przez nią przechowywanych. Intencja zawiera działanie, dane (reprezentowane przez identyfikator URI), mapę typu klucz – wartość dla dodatkowych danych oraz jawną nazwę klasy (zwaną **nazwą składnika**). Omówimy kolejno każdy z wymienionych elementów.

### Uwaga!

Jeżeli intencja zawiera w sobie nazwę składnika, jest ona określana jako intencja **jawną**. Jeśli natomiast nie przechowuje nazwy składnika, ale jest zależna od innych składników, na przykład działania lub danych, nazywana jest intencją **niejawną**. W trakcie czytania rozdziału zauważymy, że pomiędzy tymi dwoma typami istnieją drobne różnice.

## Intencje a identyfikatory danych URI

Do tej pory zajmowaliśmy się najprostszymi intencjami, w których wymagana jest jedynie nazwa działania. Aktywność ACTION\_DIAL ukazana na listingu 5.1 jest właśnie jedną z nich: żeby wywołać klawiaturę do wpisywania numeru, wystarczyła wyłącznie nazwa działania:

```
public static void dial(Activity activity)
{
    Intent intent = new Intent(Intent.ACTION_DIAL);
    activity.startActivity(intent);
}
```

Inaczej ma się sprawa z intencją ACTION\_CALL, służącą do wykonania połączenia z wybranym numerem. Intencja ta wymaga dodatkowego parametru Data (ponownie odnosimy się do listingu 5.1). Wskazuje on identyfikator URI, który z kolei przekierowuje do numeru telefonu:

```
public static void call(Activity activity)
{
    Intent intent = new Intent(Intent.ACTION_CALL);
    intent.setData(Uri.parse("tel:555-555-5555"));
    activity.startActivity(intent);
}
```

W tej intencji częścią danych związaną z działaniem jest ciąg znaków lub stała typu `string`, które przeważnie zawierają przedrostek z nazwą pakietu Java.

Tej części „danych” w intencji nie stanowią rzeczywiste dane, tylko wskaźnik do tych danych, którym jest ciąg znaków będący identyfikatorem URI. Identyfikator URI intencji może zawierać argumenty, które można uznawać za dane, na przykład adres URL.

Format tego identyfikatora może być inny dla każdej aktywności wywoływanej przez to działanie. W tym przypadku działanie CALL decyduje o tym, jakiego identyfikatora URI należy się spodziewać. Z tego identyfikatora uzyskiwany jest numer telefonu.

### Uwaga!

Wywoływana aktywność może również wykorzystać identyfikator URI jako wskaźnik do źródła danych, wydobyć te dane ze źródła i wykorzystać je. Przydaje się to w przypadku plików multimedialnych, na przykład muzyki, wideo i obrazów.

## Działania ogólne

Działania Intent.ACTION\_CALL oraz Intent.ACTION\_DIAL mogą doprowadzić do błędного wniosku, że istnieje wzajemnie jednoznaczny związek pomiędzy działaniem oraz wynikiem tego działania. Żeby udowodnić, że jest inaczej, rozpatrzmy kontrprzykład fragmentu IntentUtils umieszczonego na listingu 5.1:

```
public static void invokeWebBrowser(Activity activity)
{
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("http://www.google.com"));
    activity.startActivity(intent);
}
```

Zwróćmy uwagę, że działanie jest określone jedynie jako ACTION\_VIEW. Skąd wiadomo, którą aktywność należy przywołać na podstawie takiej ogólnej nazwy działania? W takich przypadkach Android bada nie tylko nazwę ogólnego działania, ale także charakter identyfikatora URI. Analizuje strukturę identyfikatora URI, w tym przykładzie http, a następnie sprawdza każdą zarejestrowaną aktywność pod kątem rozpoznawania tej struktury. Spośród aktywności, które mogą odpowiedzieć na ten identyfikator, system wyszukuje taką, która może odpowiedzieć na intencję VIEW, i właśnie ona zostaje wywołana. Żeby ten mechanizm mógł działać, aktywność przeglądarki powinna mieć zarejestrowaną intencję VIEW wobec schematu danych http. W pliku manifestie taka deklaracja intencji może wyglądać następująco:

```
<activity...>
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="http"/>
    <data android:scheme="https"/>
</intent-filter>
</activity>
```

Więcej informacji na temat opcji danych można zdobyć, przeglądając definicję XML elementu data filtru intencji na stronie <http://developer.android.com/guide/topics/manifest/data-element.html>. Elementy lub atrybuty węzła data są następujące:

- host
- mimeType
- path
- pathPattern
- pathPrefix
- port
- scheme

Atrybut mimeType jest powszechnie używany. Na przykład przedstawiony poniżej filtr intencji dla aktywności wyświetlającej listę notatek wskazuje typ MIME jako katalog tych notatek:

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

Deklarację tego filtru intencji można odczytać jako: „Przywołaj tę aktywność, aby przejrzeć listę notatek”.

Z drugiej strony ekran wyświetlający jedną notatkę posiada filtr intencji zadeklarowany za pomocą typu MIME wskazującego pojedynczy element:

```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

Deklarację tego filtra intencji można odczytać jako: „Przywołaj tę aktywność, aby obejrzeć pojedynczą notatkę”.

## Korzystanie z dodatkowych informacji

Poza głównymi atrybutami działań oraz danych intencja posiada również atrybuty **dodatkowe** (ang. *extra*). Atrybut dodatkowy zapewnia opcjonalne informacje składnikowi odbierającemu intencję. Dane dodatkowe przybierają postać par klucz – wartość: nazwa klucza zazwyczaj rozpoczyna się od nazwy pakietu, a wartość powinna być dowolnym, podstawowym typem danych lub dowolnym obiektem — byleby tylko zaimplementowano interfejs `android.os.Parcelable`. Takie dodatkowe informacje są reprezentowane przez klasę Androida `android.os.Bundle`.

Poniżej zaprezentowano dwie metody w klasie `Intent`, zapewniające dostęp do dodatkowych atrybutów obiektu klasy `Bundle`:

```
// Pobranie klasy Bundle z intencji
Bundle extraBundle = intent.getExtras();

// Umieszczenie klasy Bundle w intencji
Bundle anotherBundle = new Bundle();

// Wypełnienie klasy Bundle parami klucz – wartość
...
// Ustanowienie klasy Bundle w intencji
intent.putExtras(anotherBundle);
```

Funkcja `getExtras` jest zrozumiała: przekazuje obiekt klasy `Bundle` zawarty w intencji. `putExtras` sprawdza, czy intencja zawiera aktualnie obiekt `Bundle`. Jeżeli tak jest w istocie, funkcja ta przekonwertuje dodatkowe klucze i wartości z nowego obiektu klasy `Bundle` do już istniejącego. W przeciwnym razie funkcja `putExtras` utworzy taki obiekt i skopiuje pary klucz – wartość z tego obiektu do istniejącego już wystąpienia klasy `Bundle`.

### Uwaga!

Funkcja `putExtras` tworzy repliki przychodzącego obiektu `Bundle`, a nie odniesienia do nich. Zatem podczas późniejszej modyfikacji przychodzącej klasy `Bundle` nie trzeba zmieniać pakietu znajdującego się w intencji.

Istnieje wiele metod dodawania podstawowych typów do obiektu klasy `Bundle`. Poniżej zaprezentowano kilka metod dodających proste typy danych do dodatkowych danych:

```
putExtra(String name, boolean value);
putExtra(String name, int value);
putExtra(String name, double value);
putExtra(String name, String value);
```

A tu są nieco trudniejsze dodatkowe dane:

```
// Obsługa prostej tablicy  
putExtra(String name, int[]values);  
putExtra(String name, float[]values);  
  
// Obiekty serializowalne  
putExtra(String name, Serializable value);  
  
// Obsługa parcelowania  
putExtra(String name, Parcelable value);  
  
// Dodaje kolejny obiekt klasy Bundle do danego klucza  
// Obiekty klasy Bundle w obiektach klasy Bundle  
putExtra(String name, Bundle value);  
  
// Dodaje obiekty Bundle z innej intencji  
// Kopie obiektów klasy Bundle  
putExtra(String name, Intent anotherIntent);  
  
// Obsługa jawnej listy tablicowej  
putIntegerArrayListExtra(String name, ArrayList arrayList);  
putParcelableArrayListExtra(String name, ArrayList arrayList);  
putStringArrayListExtra(String name, ArrayList arrayList);
```

Po stronie odbiorcy równoważne metody pobierające (typu get) rozpoczynają działanie od odczytywania dodatkowych danych na podstawie kluczowych nazw.

Klasa Intent definiuje dodatkowe, kluczowe ciągi znaków, związane z konkretnymi działania. Pod adresem [http://developer.android.com/reference/android/content/Intent.html#EXTRA\\_ALARM\\_COUNT](http://developer.android.com/reference/android/content/Intent.html#EXTRA_ALARM_COUNT) można się zapoznać z dużą liczbą tych zawierających dodatkowe informacje stałych.

Przyjrzymy się dostępnym pod powyższym adresem URL przykładom dodatkowych danych, związanych z wysyłaniem wiadomości e-mail:

- **EXTRA\_EMAIL.** Klucz ten służy do przechowywania grupy adresów e-mail. Jego wartość to android.intent.extra.EMAIL. Powinien wskazywać tablicę ciągów znakowych, zawierającą wpisane adresy e-mail.
- **EXTRA\_SUBJECT.** Dzięki temu kluczowi możliwe jest przechowywanie nazwy tematu wiadomości e-mail. Wartością tego klucza jest android.intent.extra.SUBJECT. Powinien wskazywać ciąg znaków stanowiący temat wiadomości.

## **Stosowanie składników do bezpośredniego przywoływania aktywności**

Prześledziliśmy kilka sposobów uruchamiania aktywności za pomocą intencji. Pokazaliśmy, jak jawne działanie uruchamia aktywność oraz jak można tego dokonać, stosując ogólne działanie za pomocą identyfikatora URI. Istnieje również bardziej bezpośredni sposób uruchomienia aktywności: można określić jej klasę ComponentName, stanowiącą abstrakcję powiązaną z nazwą pakietu danego obiektu oraz nazwą klasy. Istnieje wiele metod klasy Intent pozwalających na określenie składnika:

```
setComponent(ComponentName name);
setClassName(String packageName, String classNameInThatPackage);
setClassName(Context context, String classNameInThatContext);
setClass(Context context, Class classObjectInThatContext);
```

Ostatecznie wszystkie te metody stanowią skróty do wywoływania jednej metody:

```
setComponent(ComponentName name);
```

Obiekt klasy ComponentName łączy ze sobą nazwę pakietu oraz nazwę klasy. Na przykład poniższy kod wywołuje dostępną w emulatorze aktywność `Contacts`:

```
Intent intent = new Intent();
intent.setComponent(new ComponentName(
    "com.android.contacts",
    "com.android.contacts.DialContactsEntryActivity");
startActivity(intent)
```

Zauważmy, że nazwy pakietu oraz klasy są w pełni kwalifikowane i zostają użyte do skonstruowania obiektu klasy ComponentName, zanim przejdą do klasy Intent.

Można również wykorzystać nazwę klasy bezpośrednio, bez tworzenia obiektu ComponentName. Rozważmy ponownie fragment kodu `BasicViewActivity`:

```
public class BasicViewActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.some-view);
    }
}//eof-class
```

Biorąc go pod uwagę, można wykorzystać poniższy kod do uruchomienia aktywności:

```
Intent directIntent = new Intent(activity, BasicViewActivity.class);
activity.startActivity(directIntent);
```

Jeżeli jednak każdy rodzaj intencji ma uruchamiać aktywność, należy ją zarejestrować w pliku `AndroidManifest.xml` w następujący sposób:

```
<activity android:name=".BasicViewActivity"
    android:label="Aktywność testowa">
```

Do bezpośredniego wywołania aktywności za pomocą nazwy klasy lub nazwy składnika nie są potrzebne filtry intencji. Jak już wcześniej wyjaśniliśmy, jest to tak zwana intencja jawnia. Ponieważ taka intencja definiuje do wywołania w pełni kwalifikowany składnik systemu Android, podczas przywoływania tego składnika ignorowane są pozostałe elementy intencji.

## Kategorie intencji

Aktywności można dzielić na kategorie, żeby dało się je wyszukiwać na podstawie nazwy kategorii. Na przykład podczas rozruchu system Android szuka aktywności znajdujących się w kategorii nazwanej `CATEGORY_LAUNCHER`. Pobiera następnie nazwy oraz ikony tych aktywności i umieszcza je na ekranie startowym.

Kolejny przykład: Android wyszukuje aktywność oznaczoną etykietą CATEGORY\_HOME, żeby wyświetlić ekran startowy podczas uruchamiania. Podobnie etykieta CATEGORY\_GADGET określa aktywności, które nadają się do osadzenia lub wykorzystania wewnątrz innej aktywności.

Format ciągu znaków dla kategorii takiej jak CATEGORY\_LAUNCHER jest ustanowiony zgodnie z konwencją definicji category:

```
android.intent.category.LAUNCHER
```

Znajomość ciągów znaków w definicjach category będzie potrzebna, ponieważ kategorie są rejestrowane przez aktywności w pliku *AndroidManifest.xml* jako część ich definicji filtru aktywności. Poniżej umieściliśmy przykład:

```
<activity android:name=".HelloWorldActivity"
          android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

#### Uwaga!

Aktywności mogą posiadać pewne właściwości powodujące ich ograniczanie lub uruchamianie, na przykład umożliwiające osadzenie ich w nadzędnej aktywności. Takie właściwości są definiowane poprzez kategorie.

Przejrzyjmy szybko niektóre predefiniowane kategorie oraz dowiedzmy się, do czego są wykorzystywane (tabela 5.1).

Więcej szczegółów na temat wymienionych kategorii aktywności można znaleźć pod następującym adresem, poświęconym klasie Intent:

[http://developer.android.com/reference/android/content/Intent.html#CATEGORY\\_ALTERNATIVE](http://developer.android.com/reference/android/content/Intent.html#CATEGORY_ALTERNATIVE)

Kiedy intencja służy do uruchomienia aktywności, rodzaj tej aktywności można wybrać poprzez wskazanie kategorii. Ewentualnie można wyszukać aktywności pasujące do określonej kategorii. Poniżej zamieściliśmy przykład uzyskiwania zestawu głównych aktywności, odpowiadających kategorii CATEGORY\_LAUNCHER:

```
Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);
PackageManager pm = getPackageManager();
List<ResolveInfo> list = pm.queryIntentActivities(mainIntent, 0);
```

PackageManager jest kluczową klasą, pozwalającą odkrywać aktywności dopasowane do określonych intencji bez konieczności ich wywoływanego. Interfejs ResolveInfo pozwala na wymienianie otrzymywanych aktywności oraz wywoływanie ich w miarę potrzeby. Poniżej przedstawiamy rozwinięcie wcześniejszego kodu, w którym sprawdzana jest lista dostępnych aktywności oraz wywoływana jest jedna z nich, jeśli nazwy będą się zgadzały. W celach testowych wykorzystaliśmy własną nazwę.

```
for(ResolveInfo ri: list)
{
    //ri.activityInfo.
    Log.d("test",ri.toString());
    String packagename = ri.activityInfo.packageName;
```

**Tabela 5.1.** Kategorie aktywności oraz ich omówienie

Nazwa kategorii	Opis
CATEGORY_DEFAULT	Aktywność można zadeklarować jako domyślną, aby móc ją wywołać poprzez intencję niejawną. Jeżeli nie zdefiniujemy tej kategorii wobec aktywności, trzeba będzie ją za każdym razem jawnie wywoływać za pomocą nazwy jej klasy. Dlatego właśnie znajdujemy specyfikację domyślnej kategorii w aktywnościach, które zostają wywołane za pomocą nazw ogólnych działań lub innych działań.
CATEGORY_BROWSABLE	Aktywność można zadeklarować jako odpowiednią do przeglądania, gwarantując w ten sposób, że nie będzie po uruchomieniu w przeglądarce naruszała jej reguł bezpieczeństwa.
CATEGORY_TAB	Aktywność tego rodzaju jest osadzalna w oznaczonej, nadrzędnej aktywności.
CATEGORY_ALTERNATIVE	Aktywność może zostać zadeklarowana jako alternatywna dla pewnych przeglądanych typów danych. Elementy te standardowo są pokazywane jako część menu opcji podczas oglądania dokumentu. Na przykład widok wydruku jest uznawany za alternatywny wobec widoku normalnego.
CATEGORY_SELECTED ↳_ALTERNATIVE	Aktywność może zostać zadeklarowana jako alternatywna dla pewnych typów danych. Przypomina to listę dostępnych edytorów do przeglądania dokumentu tekstowego lub napisanego w języku HTML.
CATEGORY_LAUNCHER	Przydzielenie tej kategorii do aktywności sprawi, że aktywność ta zostanie wyświetlona na ekranie startowym.
CATEGORY_HOME	Aktywność tego typu będzie ekranem startowym. Zazwyczaj powinna istnieć tylko jedna aktywność tego rodzaju. W przypadku obecności większej ich liczby system zażąda wybrania jednej z nich.
CATEGORY_PREFERENCE	Zaznaczone są w ten sposób aktywności zapewniają obsługę ustawień, zatem będą one umieszczone na ekranie preferencji.
CATEGORY_GADGET	Aktywność tego typu jest osadzalna w aktywności nadrzędnej.
CATEGORY_TEST	Jest to aktywność testowa.
CATEGORY_EMBED	Kategoria ta została zastąpiona przez kategorię CATEGORY_GADGET, pozostawiono ją jednak dla zachowania zgodności z wcześniejszymi wersjami oprogramowania.

```

String classname = ri.activityInfo.name;
Log.d("test", packagename + ":" + classname);
if (classname.equals("com.ai.androidbook.resources.TestActivity"))
{
    Intent ni = new Intent();
    ni.setClassName(packagename,classname);
    activity.startActivity(ni);
}

```

Możemy również rozpocząć aktywność opartą wyłącznie na takiej kategorii, jak `CATEGORY_LAUNCHER`:

```
public static void invokeAMainApp(Activity activity)
{
    Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
    mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);
    activity.startActivity(mainIntent);
}
```

Do intencji będzie pasowało kilka aktywności. Zatem którą z nich wybierze Android? System rozwiązuje ten problem poprzez wyświetlenie okna dialogowego *Complete action using* (dokończ działanie za pomocą), w którym są widoczne wszystkie dopuszczalne aktywności. Należy wybrać jedną z nich.

A to inny przykład zastosowania intencji służącej do wywoływania ekranu startowego:

```
// Przechodzi do ekranu startowego
Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
mainIntent.addCategory(Intent.CATEGORY_HOME);
startActivity(mainIntent);
```

Jeżeli należy użyć ekranu startowego innego niż domyślny, można napisać swój własny i zadeklarować aktywność do kategorii `HOME`. W tym przypadku powyższy kod da możliwość otwarcia swojego ekranu startowego, ponieważ zarejestrowano kilka jego aktywności:

```
// Podmienia ekran startowy na utworzony przez programistę
<intent-filter>
    <action android:value="android.intent.action.MAIN" />
    <category android:value="android.intent.category.HOME"/>
    <category android:value="android.intent.category.DEFAULT" />
</intent-filter>
```

## Reguły przydzielania intencji do ich składników

Poznaliśmy już wiele aspektów dotyczących intencji. Gwoli przypomnienia, omówiliśmy działania, identyfikatory URI danych, dane dodatkowe oraz kategorie. Biorąc pod uwagę te aspekty, Android stosuje wiele rozwiązań w procesie przydzielania intencji do docelowych aktywności za pomocą filtrów intencji.

Podstawowe znaczenie ma przydzielenie nazwy składnika do intencji. Jeżeli tak się stanie, mamy do czynienia z jawną intencją. W takim przypadku znaczenie ma wyłącznie nazwa składnika; każdy inny aspekt lub atrybut intencji zostaje zignorowany. W przypadku niejawnnej intencji istnieje wiele różnych reguł określających obiekty docelowe.

Podstawowa reguła jest taka, że każde przychodzące działanie intencji, jej kategoria oraz charakterystyka danych *muszą* pasować do odpowiednich obiektów określonych w filtrze intencji (lub we właściwy sposób prezentować te obiekty). W przeciwnieństwie do intencji, filtr intencji może definiować wiele działań, kategorii oraz charakterystyk danych. A zatem jeden filtr może spełniać założenia wielu intencji, co oznacza, że jedna aktywność może stać się odpowiedź na różne intencje. Jednak znaczenie terminu „dopasowanie” jest różne dla poszczególnych rodzajów elementów: działań, atrybutów danych oraz kategorii. Przyjrzyjmy się więc kryteriom dopasowania każdego elementu niejawnnej intencji.

## Działanie

Jeżeli intencja zawiera w sobie działanie, filtr intencji musi posiadać je na swojej liście działań lub nie posiadać żadnej listy działań. Jeśli zatem filtr intencji *nie zdefiniuje żadnego działania*, będzie on *dopasowany* do dowolnego przychodzącego działania.

Jeżeli w filtrze intencji zostanie zdefiniowane przynajmniej jedno działanie, musi ono odpowiadać przychodzącemu działaniu intencji.

## Dane

Jeżeli w filtrze intencji nie zostaną zdefiniowane charakterystyki danych, nie będzie on dopasowany do przychodzącej intencji, przenoszącej jakiekolwiek dane lub ich atrybuty. Oznacza to, że będzie wyszukiwał wyłącznie intencje niepowiązane z danymi.

Brak danych i brak działania (w filtrze) mają zupełnie odwrotne działanie. Jeżeli w filtrze nie zostanie zdefiniowane żadne działanie, każdy obiekt będzie dopasowany. Jeżeli w filtrze nie zostaną umieszczone dane, każdy bit informacji z intencji będzie niedopasowany.

## Typ danych

Aby typ danych przychodzącej intencji był dopasowany, musi się znajdować na liście typów danych określonych w filtrze intencji. Typ danych zamieszczony w intencji musi być również obecny w filtrze intencji.

Typ danych z przychodzącej intencji może zostać określony na jeden z dwóch sposobów. Po pierwsze, jeśli identyfikator URI danych jest identyfikatorem URI pliku lub treści, dostawca treści lub sam system automatycznie rozpoznają typ danych. Drugie rozwiązanie polega na sprawdzeniu jawnego typu danych intencji. Aby to się powiodło, przychodząca intencja nie może posiadać ustanowionego identyfikatora URI danych, ponieważ jest on nadawany automatycznie w momencie wywołania metody `setType` wobec intencji.

Jako część specyfikacji typów MIME Android pozwala również na wprowadzenie symbolu gwiazdki (\*), zastępującego wszystkie możliwe podtypy.

Ponadto typ danych rozróżnia wielkość liter.

## Schemat danych

Aby schemat danych był dopasowany, musi się znajdować na liście schematów w filtrze intencji oraz odpowiadać schematowi znajdującemu się w przychodzącej intencji. Innymi słowy, przychodzący schemat danych musi być odzwierciedlony w filtrze intencji.

W przychodzącej intencji schemat stanowi pierwszy człon identyfikatora URI danych. W przypadku intencji nie ma żadnego sposobu na ustanowienie schematu. Wywodzi się on wprost z identyfikatora URI danych intencji i wygląda mniej więcej tak: <http://www.jakasstrona.com/→jakasciezka>.

Jeżeli schemat danych z przychodzącego identyfikatora URI intencji rozpoczyna się od członów `content:` lub `file:`, jest on dopasowany bez względu na schemat filtra, domenę czy ścieżkę. Zgodnie z dokumentacją zestawu SDK wynika to z faktu, iż każdy składnik powinien móc odczytywać dane z tych dwóch rodzajów adresów URL, które w swej istocie są lokalne. Inaczej mówiąc, od wszystkich składników oczekuje się obsługi tych dwóch typów adresów URL.

Schemat również rozróżnia wielkość liter.

## Uprawnienia do danych

Jeżeli w filtrze nie zostaną uwzględnione żadne uprawnienia, wszelkie uprawnienia (lub nazwy domenowe) przychodzących identyfikatorów URI będą dopasowane. Jeżeli w filtrze zdefiniowano jakieś uprawnienie, na przykład `www.jakasstrona.com`, to do danego identyfikatora URI intencji powinien pasować jeden schemat oraz jedno uprawnienie.

Jeśli na przykład uprawnieniem określonym w filtrze intencji jest `www.jakasstrona.com`, a schematem jest `https`, adres `http://www.jakasstrona.com/jakassciezka` będzie niedopasowany, ponieważ `http` nie jest w tym przypadku obsługiwany formatem.

Uprawnienie także rozróżnia wielkość liter.

## Ścieżka danych

Brak ścieżek w filtrze intencji oznacza dopasowanie do każdej ścieżki znajdującej się w przychodzących identyfikatorze URI. Jeżeli w filtrze określono ścieżkę, na przykład `jakassciezka`, przychodzącemu identyfikatorowi URI danych w intencji powinien odpowiadać jeden schemat, jedno uprawnienie i jedna ścieżka danych.

Innymi słowy, schemat, uprawnienie i ścieżka współpracują ze sobą w celu sprawdzenia poprawności przychodzącego identyfikatora URI, na przykład `http://www.jakasstrona.com/→jakassciezka`. Zatem elementy `path`, `authority` i `scheme` nie działają oddzielnie, lecz współpracują ze sobą.

Podobnie jak we wcześniejszych przypadkach, ścieżka rozróżnia wielkość liter.

## Kategorie intencji

Każda kategoria zawarta w przychodzącej intencji musi być wymieniona na liście kategorii filtru intencji. Większa liczba kategorii w filtrze nie jest niczym złym. Jeżeli filtr *nie zawiera żadnych kategorii*, dopasowana będzie *wyłącznie intencja nieposiadająca żadnej zadeklarowanej kategorii*.

Istnieje jednak pewne zastrzeżenie. Android uwzględnia wszystkie niejawne intencje przekazane do metody `startActivity()`, tak jakby posiadały przynajmniej jedną kategorię: `android.intent.category.DEFAULT`. Kod tworzący tę metodę będzie wyszukiwał jedynie te aktywności, dla których zdefiniowano kategorię `DEFAULT`, ale tylko wtedy, gdy przychodząca intencja jest niejawna. Zatem każda aktywność, która będzie wywołana za pomocą niejawniej intencji, musi zawierać domyślną kategorię w filtrze intencji.

Nawet jeśli aktywność nie posiada domyślnej kategorii zadeklarowanej w filtrze intencji, w przypadku gdy znamy jej jawne nazwy składników, będziemy mogli ją uruchomić tak jak program wywołujący. Jeżeli samodzielnie wyszukujemy w sposób jawnym pasujące intencje, bez posiadania domyślnej kategorii jako kryterium wyszukiwania, to w ten sposób uruchomimy aktywności.

W tym sensie kategoria `DEFAULT` jest artefaktem implementacyjnym metody `startActivity`, a nie naturalnym składnikiem filtra intencji.

Istnieje jeszcze dodatkowy problem. System Android uznaje, iż domyślna kategoria jest niepotrzebna, w przypadku gdy aktywność ma być uruchamiana jedynie z poziomu ekranów programów wywołujących. Zatem w filtrach intencji takich aktywności zazwyczaj umieszczane są wyłącznie kategorie `MAIN` i `LAUNCHER`. Jednak kategoria `DEFAULT` może również zostać opcjonalnie zdefiniowana dla tych aktywności.

## Działanie ACTION\_PICK

Jak na razie zajmowaliśmy się intencjami lub działaniami, które w głównej mierze wywoływały inną aktywność bez uzyskiwania wyników. Przyjrzymy się nieco bardziej zaawansowanemu działaniu, w którym po jego wywołaniu otrzymujemy wartość. Takim działaniem jest ACTION\_PICK.

Działanie ACTION\_PICK polega na uruchomieniu aktywności, która wyświetla listę elementów. Aktywność powinna następnie umożliwić użytkownikowi wybranie jednego z elementów. Gdy tak się stanie, aktywność przekaże programowi żądającemu identyfikator URI wybranego elementu. Dzięki temu można wielokrotnie korzystać z funkcjonalności interfejsu użytkownika do wybierania elementów określonego typu.

Należy wskazać zbiór wybieranych elementów za pomocą typu MIME, który określa kursor treści w Androidzie. Typ MIME takiego identyfikatora URI powinien wyglądać mniej więcej następująco:

```
vnd.android.cursor.dir/vnd.google.note
```

Aktywność powinna uzyskać dane z tego dostawcy treści na podstawie identyfikatora URI. Z tego właśnie powodu dane powinny być umieszczane w dostawcach treści wszędzie, gdzie jest to możliwe.

W żadnym działaniu, które przekazuje dane w ten sposób, nie można zastosować metody `startActivity()`, ponieważ nie przekazuje ona żadnego wyniku. Wynika to z faktu, że ta metoda otwiera nową aktywność, która przyjmuje postać modalnego okna dialogowego w oddzielnym wątku, i pozostawia główny wątek dla innych zdarzeń. Innymi słowy, metoda `startActivity()` jest wywołaniem asynchronicznym, nieposiadającym żadnych wywołań zwrotnych, które wskazywałyby na to, co się stało z przywołaną aktywnością. Jeśli przewidujemy otrzymywanie danych wynikowych, można wykorzystać wariant metody `startActivity()`, nazywany `startActivityForResult()`, który zawiera wywołanie zwrotne.

Przyjrzymy się sygnaturze metody `startActivityForResult()` klasy `Activity`:

```
public void startActivityForResult(Intent intent, int requestCode)
```

Metoda ta uruchamia aktywność, która ma wyświetlić wynik. Po zakończeniu aktywności jej źródłowa metoda `onActivityResult()` zostanie wywołana wraz z danym argumentem `requestCode`. Sygnaturą tej metody wywoływania zwrotnego jest:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
```

Parametr `requestCode` reprezentuje dane przekazane metodzie `startActivityForResult()`. Jego wartościami mogą być `RESULT_OK`, `RESULT_CANCELLED` lub kod niestandardowy. Nazwy kodów niestandardowych powinny się zaczynać od członu `RESULT_FIRST_USER`. Parametr `Intent` zawiera dodatkowe dane, które wywołana aktywność powinna przekazać jako wynik. W przypadku działania `ACTION_PICK` otrzymane w intencji dane wskazują identyfikator URI jednego elementu.

Listing 5.3 przedstawia wywołanie aktywności przekazującej wynik.

### Uwaga!

Kod z listingu 5.3 opiera się na założeniu, że zainstalowano przykładowy projekt Notepad, dostępny w zestawie Android SDK. Na końcu rozdziału umieściliśmy odnośnik do wskazówek dotyczących pobrania tej aplikacji. Będzie to pomocne dla osób, które jeszcze nie posiadają zestawu SDK.

**Listing 5.3.** Przekazywanie danych wynikowych po wywołaniu działania

```
public class SomeActivity extends Activity
{
    .....
    .....
    public static void invokePick(Activity activity)
    {
        Intent pickIntent = new Intent(Intent.ACTION_PICK);
        int requestCode = 1;
        pickIntent.setData(Uri.parse(
            "content://com.google.provider.NotePad/notes"));
        activity.startActivityForResult(pickIntent, requestCode);
    }

    protected void onActivityResult(int requestCode
        ,int resultCode
        ,Intent outputIntent)
    {
        // Fragment ten służy do poinformowania nadzędnej klasy (Activity)
        // o fakcie, że aktywność zakończyła działanie oraz że klasa bazowa
        // może przeprowadzić niezbędne porządkи
        super.onActivityResult(requestCode, resultCode, outputIntent);
        parseResult(this, requestCode, resultCode, outputIntent);
    }

    public static void parseResult(Activity activity
        , int requestCode
        , int resultCode
        , Intent outputIntent)
    {
        if (requestCode != 1)
        {
            Log.d("Test", "To wywołał ktoś inny. Nie my.");
            return;
        }
        if (resultCode != Activity.RESULT_OK)
        {
            Log.d("Test", "Kod wyniku nie zgadza się:" + resultCode);
            return;
        }
        Log.d("Test", "Kod wyniku zgadza się:" + resultCode);
        Uri selectedUri = outputIntent.getData();
        Log.d("Test", "Wynikowy identyfikator uri:" + selectedUri.toString());

        // Wyświetla notatkę
        outputIntent.setAction(Intent.ACTION_VIEW);
        startActivity(outputIntent);
    }
}
```

---

Stałe RESULT\_OK, RESULT\_CANCELED oraz RESULT\_FIRST\_USER są zdefiniowane w klasie Activity. Ich wartości numeryczne są następujące:

```
RESULT_OK = -1;
RESULT_CANCELED = 0;
RESULT_FIRST_USER = 1;
```

Aby funkcja PICK działała, element, który jej odpowiada, powinien zawierać kod jawnie odpowiadający wymaganiom intencji PICK. Zobaczmy, w jaki sposób cel ten osiągnięto w przykładowej aplikacji Notepad. Po wybraniu elementu z listy system sprawdza, czy intencja, która wywołała aktywność, jest intencją PICK. Jeśli tak jest, w nowej intencji ustanawia się identyfikator URI danych, przekazywany poprzez metodę setResult():

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
    Uri uri = ContentUris.withAppendedId(getIntent().getData(), id);

    String action = getIntent().getAction();
    if (Intent.ACTION_PICK.equals(action) ||
        Intent.ACTION_GET_CONTENT.equals(action))
    {
        // Program wywołujący czeka na zwrócenie notatki wybranej przez
        // użytkownika. Jedna z nich została kliknięta, więc będzie teraz zwrócona.
        setResult(RESULT_OK, new Intent().setData(uri));
    } else {
        // Uruchamia aktywność odpowiedzialną za przeglądanie/edycję bieżącego elementu.
        startActivityForResult(new Intent(Intent.ACTION_EDIT, uri));
    }
}
```

## Działanie ACTION\_GET\_CONTENT

Działanie ACTION\_GET\_CONTENT jest podobne do działania ACTION\_PICK. W przypadku tego drugiego określa się identyfikator URI, który wskazuje zbiór elementów, na przykład kolekcję notatek. Działanie ma pobrać jedną z notatek i przekazać ją programowi wywołującemu. W przypadku działania ACTION\_GET\_CONTENT potrzebny jest element określonego typu MIME. Android przeszukuje wtedy zarówno aktywności zdolne do utworzenia takich elementów, jak i aktywności, w których można wybierać elementy spełniające warunek właściwego typu MIME.

Z pomocą działania ACTION\_GET\_CONTENT można wybrać notatkę ze zbioru notatek obsługiwanych przez aplikację Notepad w następujący sposób:

```
public static void invokeGetContent(Activity activity)
{
    Intent pickIntent = new Intent(Intent.ACTION_GET_CONTENT);
    int requestCode = 2;
    pickIntent.setType("vnd.android.cursor.item/vnd.google.note");
    activity.startActivityForResult(pickIntent, requestCode);
}
```

Zwróćmy uwagę na sposób, w jaki typ intencji zostaje dopasowany do typu MIME pojedynczej notatki. Porównamy to z kodem ACTION\_PICK, w którym na wejściu jest identyfikator URI danych. Kod znajduje się poniżej:

```
public static void invokePick(Activity activity)
{
    Intent pickIntent = new Intent(Intent.ACTION_PICK);
    int requestCode = 1;
    pickIntent.setData(Uri.parse(
        "content://com.google.provider.NotePad/notes"));
    activity.startActivityForResult(pickIntent, requestCode);
}
```

Żeby aktywność zareagowała na działanie ACTION\_GET\_CONTENT, musi ona posiadać zarejestrowany filtr intencji wskazujący na to, że aktywność ta będzie zdolna obsłużyć element o danym typie MIME. Aplikacja Notepad spełnia ten wymóg w następujący sposób:

```
<activity android:name="NotesList" android:label="@string/title_notes_list">
...
<intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
...
</activity>
```

Reszta kodu obsługującego metodę onActivityResult() jest taka sama jak w przykładzie z działaniem ACTION\_PICK. Jeżeli istnieje wiele aktywności mogących przekazać ten sam typ MIME, zostanie wyświetlony ekran wyboru, na którym można wskazać jedną z nich.

## Wprowadzenie do intencji oczekujących

W Androidzie istnieje odmiana intencji, zwana intencją oczekującą (ang. *pending intent*). W tym przypadku pewien składnik może przechowywać intencję do przyszłego użytku w lokacji, z której będzie można ją ponownie przywołać. Na przykład w menedżerze alarmów chcemy uruchomić usługę w momencie włączenia się alarmu. Android dokonuje tego poprzez utworzenie osłonowej intencji oczekującej wokół intencji. Intencja ta jest przechowywana w bezpiecznym miejscu, tak że nawet w przypadku wygaśnięcia procesu wywoływanego zostanie wysłana do docelowego miejsca. W trakcie tworzenia intencji oczekującej Android przechowuje wystarczająco wiele informacji na temat źródłowego procesu, aby poświadczenia zabezpieczeń mogły zostać sprawdzone na etapie jej wysyłania lub przywoływania.

Zobaczmy, w jaki sposób możemy stworzyć intencję oczekującą.

```
Intent regularIntent;
PendingIntent pi = PendingIntent.getActivity(context, 0, regularIntent, ...);
```

### Uwaga!

Drugi argument metody PendingIntent.getActivity() nosi nazwę requestCode i w naszym przykładzie jego wartość wynosi 0. Argument ten służy do rozróżniania dwóch intencji oczekujących, wywodzących się z tej samej intencji. Zagadnienie to zostało omówione o wiele dokładniej w rozdziale 15., gdzie zajmujemy się intencjami oczekującymi menedżerów alarmów.

Nazwa metody PendingIntent.getActivity() wywołuje pewne wątpliwości. Jaka jest tu rola aktywności? Poza tym dlaczego nie skorzystano z metody create podczas tworzenia intencji oczekującej, a zamiast tego wprowadzono metodę get?

Aby zrozumieć pierwsze zagadnienie, musimy zagłębić się nieco bardziej w sposób użytkowania standardowej intencji. Zwyczajna intencja może zostać wykorzystana do uruchomienia aktywności lub usługi albo do przywołania odbiorcy komunikatów (w dalszej części książki zaznajomimy się dokładniej z usługami oraz odbiorcami komunikatów). W każdym wymienionym przypadku sposób wykorzystania intencji jest nieco odmienny. Aby pogodzić je ze sobą, kontekst Androida (superklasa aktywności) zawiera trzy oddzielne metody. Są to:

```
startActivity(intent)
startService(intent)
sendBroadcast(intent)
```

W jaki sposób te metody pozwalają na określenie, czy należy rozpocząć aktywność, usługę, czy też przygotować odbiorcę wiadomości na transmisję, jeśli trzeba przechować intencje do późniejszego wykorzystania? Właśnie dlatego trzeba jawnie określić przeznaczenie intencji oczekującą w momencie jej tworzenia. Teraz poniższe trzy metody stają się zrozumiałe:

```
PendingIntent.getActivity(context, 0, intent, ...)
PendingIntent.getService(context, 0, intent, ...)
PendingIntent.getBroadcast(context, 0, intent, ...)
```

Pozostała nam jeszcze kwestia metody get. Android przechowuje intencje oraz umożliwia ich ponowne zastosowanie. Jeżeli dwukrotnie wywołamy oczekującą intencję za pomocą tego samego obiektu intencji, otrzymamy identyczną intencję oczekującą. Stanie się to nieco bardziej zrozumiałe, jeśli przypatrzymy się pełnej sygnaturze metody `PendingIntent.getActivity()`. Oto ona:

```
PendingIntent.getActivity(Context context, //pierwotny kontekst
    int requestCode, //1,2,3 itd.
    Intent intent, //oryginalna intencja
    int flags ) //flagi
```

Jeżeli naszym celem jest uzyskanie kolejnej kopii intencji oczekującej, musimy dostarczyć inną wartość argumentu `requestCode`. Konieczność ta została wyjaśniona o wiele dokładniej w rozdziale 15., podczas omawiania menedżerów alarmów. Dwie „intencje” są uznawane za identyczne, jeżeli ich wewnętrzne elementy są takie same. Powyższe stwierdzenie nie dotyczy dodatkowych obiektów.

Dla intencji oczekującej rodzaj czynności jest definiowany za pomocą flag. Chodzi tu o takie czynności, jak przekazanie wartości null, nadpisanie elementów dodatkowych i tak dalej. Więcej wiadomości na temat istniejących rodzajów flag znajdziemy pod następującym adresem:

<http://developer.android.com/reference/android/app/PendingIntent.html>

Zazwyczaj, aby intencja zachowała się w sposób domyślny, przekazujemy wartość 0 argumentom `requestCode` oraz `flags`.

## Odbośni

Pod poniższymi adresami można znaleźć więcej materiałów (w języku angielskim), uzupełniających informacje z tego rozdziału:

- <http://developer.android.com/reference/android/content/Intent.html> — pod tym adresem znajdziemy ogólne informacje dotyczące intencji. Poznamy tu najpopularniejsze działania, obiekty dodatkowe itd.
- <http://developer.android.com/guide/appendix/g-app-intents.html> — znajduje się tu lista intencji wykorzystywanych w różnych aplikacjach firmy Google. Dowiemy się, w jaki sposób wywołać intencje takich aplikacji, jak Browser, Map, Dialer czy Google Street View.
- <http://developer.android.com/reference/android/content/IntentFilter.html> — tu znajdziemy informacje dotyczące filtrów intencji, przydatne podczas rejestrowania tych filtrów.

- <http://developer.android.com/guide/topics/intents/intents-filters.html> — tu omówiono reguły określające filtry intencji.
- <http://developer.android.com/resources/samples/get.html> — za pomocą tego łącza możemy pobrać przykładowy kod aplikacji Notepad. Bez wczytania tego projektu nie przetestujemy niektórych intencji.
- <http://developer.android.com/resources/samples/NotePad/index.html> — wersja online kodu źródłowego aplikacji Notepad.
- <http://www.openintents.org/> — witryna, której zadaniem jest próba zebrania intencji tworzonych przez różnych wydawców.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — z tego adresu możemy pobrać testowy projekt, zaprojektowany specjalnie na potrzeby niniejszego rozdziału. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R05\_Intencje*.

## Podsumowanie

W tym rozdziale zdefiniowaliśmy najważniejsze elementy dotyczące intencji w systemie Android. Przejrzeliśmy różnorodne scenariusze wykorzystania intencji, a także wyjaśniliśmy związki istniejące pomiędzy intencjami a identyfikatorami URI treści. Wytlumaczyliśmy również, w jaki sposób można wykorzystać intencje do wywoływania aktywności przekazujących wyniki. Wprowadziłmy także pojęcie intencji oczekujących. Pojęcie to zostanie dokładniej przeanalizowane w rozdziałach 15. i 22.

# Budowanie interfejsów użytkownika oraz używanie kontrollek

Do tego momentu zajmowaliśmy się podstawami Androida, lecz nie poruszaliśmy tematu interfejsu użytkownika (UI). W tym rozdziale omówimy interfejsy użytkownika oraz kontrolki. Rozpoczniemy od ogólnej filozofii projektowania interfejsów UI w Androidzie, a następnie dokonamy analizy standardowych kontrollek, dostępnych w zestawie Android SDK. Przyjrzymy się także menedżerom układu graficznego i adapterom widoków. Dyskusję zakończymy na omówieniu aplikacji Hierarchy Viewer — narzędzia służącego do wyszukiwania błędów oraz optymalizowania interfejsów UI w Androidzie.

## Projektowanie interfejsów UI w Androidzie

Projektowanie interfejsu użytkownika w Androidzie jest przyjemne. Wynika to z faktu, że proces ten jest względnie łatwy. Mamy do dyspozycji prosty do zrozumienia szkielet oraz niewielki zestaw predefiniowanych kontrollek. Dostępny obszar ekranu jest zazwyczaj ograniczony. Android wykonuje za programistę ciężką pracę, która zwykle jest związana z jakością projektowania i tworzenia interfejsu użytkownika. Te elementy oraz stwierdzenie, że użytkownik przeważnie chce wykonać w danej chwili tylko jedną czynność, sprawiają, że możemy w łatwy sposób zaprojektować dobry interfejs UI, sprawiający dobre wrażenie na użytkowniku.

Zestaw Android SDK został zaopatrzony w grupę kontrollek, które można wykorzystać do budowania aplikacji. Podobnie jak w przypadku innych zestawów projektowych (SDK) dostępne są pola tekstowe, przyciski, listy, siatki i tak dalej. Dodatkowo Android posiada zbiór kontrollek dopasowanych do urządzeń przenośnych.

Podstawą kontrollek standardowych są dwie klasy: `android.view.View` oraz `android.view.ViewGroup`. Nazwa pierwszej z nich sugeruje, że klasa `View` reprezentuje widok View ogólnego zastosowania. Kontrolki standardowe znacznie rozwijają tę klasę. Klasa  `ViewGroup` także jest widokiem, zawiera w sobie jednak

dwa inne widoki. Jest to klasa bazowa dla wielu klas układu graficznego. Podobnie jak pakiet Swing, tak i Android wykorzystuje pojęcie układu graficznego do zarządzania rozmieszczeniem kontrolek wewnątrz pojemnika widoku. Jak się przekonamy, stosowanie układów graficznych ułatwia nam kontrolowanie pozycji oraz orientacji kontrolek w interfejsach UI.

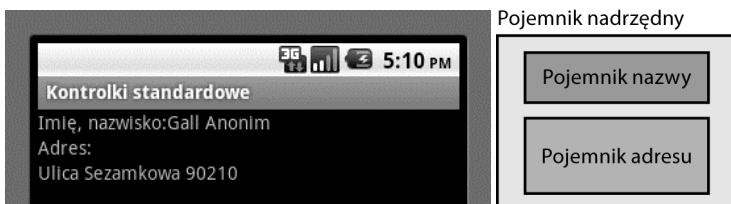
Do projektowania interfejsów UI w Androidzie można podejść na kilka sposobów. Można konstruować je, korzystając tylko z kodu. Można także definiować je, stosując same znaczniki języka XML. Istnieje nawet możliwość połączenia obydwu technik — zdefiniowanie interfejsu użytkownika w języku XML, a następnie odniesienie się do niego i modyfikowanie go w kodzie. W celach demonstracyjnych zaprojektujemy prosty interfejs użytkownika, korzystając ze wszystkich trzech metod.

Zanim rozpoczniemy, musimy ustalić pewną nomenklaturę. W niniejszej książce oraz pozostałej literaturze poświęconej Androidowi podczas omawiania procesu projektowania interfejsu użytkownika pojawiają się takie pojęcia, jak *widok* (ang. *view*), *kontrolka* (ang. *control*), *widżet* (ang. *widget*), *pojemnik* lub *kontener* (ang. *container*) oraz *układ graficzny* (ang. *layout*). Osoby stykające się po raz pierwszy z programowaniem w środowisku Android lub z projektowaniem interfejsów UI w ogóle mogą nie być zaznajomione z tymi pojęciami. Zanim rozpoczniemy, zostaną one pokrótko omówione (tabela 6.1).

**Tabela 6.1.** Nomenklatura interfejsu graficznego

Pojęcie	Opis
Widok, widżet, kontrolka	Każde z tych pojęć reprezentuje element interfejsu użytkownika. Z przykładów można wymienić przycisk, siatkę, listę, okno, okno dialogowe i tak dalej. W tym rozdziale pojęcia „widok”, „widżet” oraz „kontrolka” są stosowane jako synonimy.
Pojemnik, kontener	W tym widoku przechowywane są inne widoki. Na przykład siatkę można uznać za pojemnik, ponieważ przechowuje komórki będące widokami.
Układ graficzny	Wizualne rozmieszczenie pojemników oraz widoków, w którym mogą zostać zawarte inne układy graficzne.

Rysunek 6.1 przedstawia zrzut ekranu aplikacji, którą wkrótce zaprojektujemy. Obok zrzutu znajduje się schemat hierarchii kontrolek i pojemników aplikacji w układzie graficznym.



**Rysunek 6.1.** Interfejs użytkownika oraz układ graficzny aktywności

Podczas omawiania przykładowych programów będziemy się odnosić do tej hierarchii układu graficznego. Na razie wystarczy wiedza, że nasza aplikacja posiada jedną aktywność. Interfejs użytkownika tej aktywności składa się z trzech pojemników: obejmującego imię i nazwisko osoby, obejmującego adres oraz pojemnika zewnętrznego, nadzawanego wobec dwóch pozostałych.

## Programowanie interfejsu użytkownika wyłącznie za pomocą kodu

Pierwszy przykład, umieszczony na listingu 6.1, pokazuje, w jaki sposób należy skonstruować interfejs użytkownika wyłącznie za pomocą kodu. Można spróbować to zrobić, tworząc nowy projekt Androida zawierający aktywność nazwaną MainActivity, a następnie kopując kod z tego listingu do klasy MainActivity.

**Uwaga!**

Na końcu rozdziału podajemy adres URL, z którego można pobrać omawiane tu projekty. W ten sposób, zamiast przepisywać kod zawarty w książce, Czytelnik może zaimportować projekt do środowiska Eclipse.

**Listing 6.1.** Utworzenie prostego interfejsu użytkownika wyłącznie za pomocą kodu

```
package com.androidbook.controls;
import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup.LayoutParams;
import android.widget.LinearLayout;
import android.widget.TextView;
public class MainActivity extends Activity
{
    private LinearLayout nameContainer;
    private LinearLayout addressContainer;
    private LinearLayout parentContainer;

    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        createNameContainer();
        createAddressContainer();
        createParentContainer();
        setContentView(parentContainer);
    }
    private void createNameContainer()
    {
        nameContainer = new LinearLayout(this);
        nameContainer.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
            LayoutParams.WRAP_CONTENT));
        nameContainer.setOrientation(LinearLayout.HORIZONTAL);
        TextView nameLbl = new TextView(this);
```

```
nameLbl.setText("Imię, nazwisko: ");

TextView nameValue = new TextView(this);
nameValue.setText("Gall Anonim");

nameContainer.addView(nameLbl);
nameContainer.addView(nameValue);
}

private void createAddressContainer()
{
    addressContainer = new LinearLayout(this);

    addressContainer.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
        LayoutParams.WRAP_CONTENT));
    addressContainer.setOrientation(LinearLayout.VERTICAL);

    TextView addrLbl = new TextView(this);
    addrLbl.setText("Adres:");

    TextView addrValue = new TextView(this);
    addrValue.setText("Ulica Sezamkowa 90210");

    addressContainer.addView(addrLbl);
    addressContainer.addView(addrValue);
}

private void createParentContainer()
{
    parentContainer = new LinearLayout(this);

    parentContainer.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
        LayoutParams.FILL_PARENT));
    parentContainer.setOrientation(LinearLayout.VERTICAL);

    parentContainer.addView(nameContainer);
    parentContainer.addView(addressContainer);
}
}
```

---

Jak widać na listingu 6.1, aktywność zawiera trzy układy graficzne `LinearLayout`. Wspomniliśmy wcześniej, że obiekty typu `layout` są przystosowane do pozycjonowania innych obiektów w danej części ekranu. Na przykład układ graficzny `LinearLayout` określa, czy obiekt ma być umieszczony w pionie, czy w poziomie. W obiektach typu `layout` można umieszczać dowolne typy widoków — nawet inne układy graficzne.

W obiekcie `nameContainer` są umieszczone dwie kontrolki `TextView`: jedna wyświetla etykietę `Imię, nazwisko`, a druga przechowuje dane imienia i nazwiska (w naszym przykładzie *Gall Anonim*). Analogicznie obiekt `addressContainer` również zawiera dwie kontrolki `TextView`. Różnica pomiędzy tymi pojednikami jest taka, że ten pierwszy został umieszczony poziomo, a obiekt drugi — pionowo. Obydwa pojedniki znajdują się we wnętrzu obiektu `parentContainer`, stanowiącego podstawowy widok aktywności. Po utworzeniu pojedników aktywność przypisuje treść widoku do widoku głównego poprzez wywołanie metody `setContentView(parent →Container)`. Kiedy trzeba wyświetlić interfejs użytkownika tej aktywności, główny widok jest

wywoływany do wyświetlenia na ekranie. Widok ten następnie wywołuje pojemniki podrzędne, te z kolei wywołują swoje pojemniki podrzędne i tak dalej — aż do momentu wyświetlenia całego interfejsu.

Ponadto na listingu 6.1 widać kilka kontrolek `LinearLayout`. Dwie z nich są ułożone pionowo, a jedna poziomo. Tym rodzynkiem jest kontrolka `nameContainer`. Oznacza to, że dwie kontrolki `TextView` pojawiają się tuż obok siebie w poziomie. Obiekt `addressContainer` jest umieszczony pionowo, a zatem dwie kolejne kontrolki `TextView` są na sobie ułożone. Również pojemnik `parentContainer` jest zorientowany w pionie, dlatego obiekt `nameContainer` pojawia się ponad obiektem `addressContainer`. Zauważmy subtelną różnicę pomiędzy dwoma umieszczonymi pionowo pojemnikami `addressContainer` i `parentContainer`: ten drugi został tak skonfigurowany, żeby zająć całą długość i szerokość ekranu:

```
parentContainer.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
    LayoutParams.FILL_PARENT));
```

Natomiast obiekt `addressContainer` otacza zawartą w nim treść w pionie:

```
addressContainer.setLayoutParams(new LayoutParams(LayoutParams.FILL_PARENT,
    LayoutParams.WRAP_CONTENT));
```

Inaczej mówiąc, parametr `WRAP_CONTENT` oznacza, że widok zajmuje w danym wymiarze wyłącznie tyle przestrzeni, ile jest konieczne, a poza tym jest ograniczony rozmiarem swojego widoku nadzawanego. W przypadku kontrolki `addressContainer` oznacza to, że będzie ona zajmowała dwie linijki tekstu, ponieważ potrzebuje tylko tyle miejsca.

## Tworzenie interfejsu użytkownika wyłącznie w pliku XML

Zaprogramujmy teraz identyczny interfejs UI za pomocą języka XML (listing 6.2). Przypominamy informację z rozdziału 3., że pliki XML układu graficznego przechowywane są w katalogu zasobów (`/res/`), w podkatalogu `layout`. W celu wypróbowania tego przykładu należy utworzyć nowy projekt w środowisku Eclipse. Domyslnie zostanie utworzony plik układu graficznego `main.xml`, zlokalizowany w folderze `/res/layout`. Po dwukrotnym kliknięciu nazwy tego pliku edytor tekstowy środowiska Eclipse wyświetli jego zawartość. Prawdopodobnie na samej górze tego widoku będzie widniał mniej więcej taki ciąg znaków: „Hello World, MainActivity!” lub coś podobnego. Aby ujrzeć kod XML tego pliku, należy kliknąć zakładkę `main.xml` u dołu widoku. Staną się widoczne kontrolki `LinearLayout` oraz `TextView`. Korzystając z zakładek `Layout` oraz `main.xml`, należy odtworzyć kod z listingu 6.2 w pliku `main.xml` i zachować zmiany.

**Listing 6.2.** Utworzenie interfejsu użytkownika wyłącznie w języku XML

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <!-- KONTENER IMIENIA I NAZWISKA -->
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal" android:layout_width="fill_parent"
        android:layout_height="wrap_content">

        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="Imię, nazwisko:" />
```

```
<TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content" android:text="Gall Anonim" />

</LinearLayout>

<!-- KONTENER ADRESU -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="wrap_content">

    <TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Adres:" />

    <TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Ulica Sezamkowa 90210" />
</LinearLayout>

</LinearLayout>
```

---

W katalogu *src* tego projektu istnieje domyślny plik *.java*, zawierający definicję klasy *Activity*. Dwukrotne kliknięcie nazwy tego pliku spowoduje wyświetlenie jego treści. Zwróćmy uwagę na instrukcję *setContentView(R.layout.main)*. Fragment kodu XML umieszczony na listingu 6.2 w połączeniu z wywołaniem *setContentView(R.layout.main)* spowoduje wyświetlenie takiego samego interfejsu użytkownika jak w przypadku listingu 6.1. Nie trzeba omawiać pliku XML, warto jednak zwrócić uwagę, że zdefiniowano trzy widoki pojemników. Pierwszy, *LinearLayout*, jest odpowiednikiem pojemnika nadzawanego. Jego położenie zostaje ustalone na pionowe poprzez zdefiniowanie odpowiedniej właściwości — *android:orientation="vertical"*. W pojemniku nadzawanym są umieszczone dwa podzadane elementy *LinearLayout*, reprezentujące, odpowiednio, pojemniki *nameContainer* oraz *addressContainer*.

Uruchomienie tej aplikacji spowoduje wygenerowanie takiego samego układu graficznego jak w poprzednim przykładzie. Wyświetlane etykiety i wartości będą takie same jak na rysunku 6.1.

## Konstruowanie interfejsu użytkownika za pomocą kodu oraz języka XML

Kod na listingu 6.2 stanowi raczej wydumany przykład — trwałe zakodowanie wartości kontrolek *TextView* w układzie graficznym XML nie ma sensu. Najlepiej byłoby zaprojektować interfejs użytkownika w języku XML, a następnie utworzyć odniesienia do kontrolek. Taka technika pozwala na dołączanie dynamicznie zmieniających się danych do kontrolek zdefiniowanych w trakcie tworzenia projektu. W istocie jest to zalecana metoda. Budowanie układów graficznych w języku XML, a następnie stosowanie kodu do wypełniania tych układów dynamicznymi danymi jest całkiem prostą czynnością.

Na listingu 6.3 został pokazany ten sam interfejs UI, lecz z nieco zmodyfikowanym kodem XML. Przydzielono tutaj identyfikatory kontrolek *TextView*, dzięki czemu możemy się do nich odnieść w kodzie.

**Listing 6.3.** Utworzenie interfejsu użytkownika w kodzie XML z dołączonymi identyfikatorami

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <!-- KONTENER IMIENIA I NAZWISKA -->
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal" android:layout_width="fill_parent"
        android:layout_height="wrap_content">

            <TextView android:layout_width="wrap_content"
                android:layout_height="wrap_content" android:text="@string/name_text" />

            <TextView android:id="@+id/nameValue"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content" />
        </LinearLayout>
    <!-- KONTENER ADRESU -->
    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="wrap_content">

        <TextView android:layout_width="fill_parent"
            android:layout_height="wrap_content" android:text="@string/addr_text" />

        <TextView android:id="@+id/addrValue"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" />
    </LinearLayout>
</LinearLayout>

```

Poza dodaniem identyfikatorów do kontrolek `TextView`, które później wypełnimy danymi z kodu, możemy także użyć etykietowych kontrolek `TextView`. Ich wartości są wypełniane tekstem z pliku zasobów. Są to kontrolki `TextView` nieposiadające identyfikatora zawartego w atrybutie `android:text`. Jeżeli przypomnijmy sobie informacje z rozdziału 3., właściwe ciągi znaków, przeznaczone dla tych kontrolek, znajdują się w pliku `strings.xml` umieszczonym w folderze `/res/values`. Listing 6.4 przedstawia taki — wykorzystany w naszym przykładzie — plik `strings.xml`.

**Listing 6.4.** Plik strings.xml współpracujący z kodem z listingu 6.3

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Kontrolki standardowe</string>
    <string name="name_text">Imię i nazwisko:</string>
    <string name="addr_text">Adres:</string>
</resources>;

```

Kod umieszczony na listingu 6.5 demonstruje, w jaki sposób uzyskać odniesienia do kontrolek zdefiniowanych w kodzie XML w celu skonfigurowania ich właściwości. Możemy wstawić go do metody `onCreate()` zawartej w naszej aktywności.

**Listing 6.5.** Tworzenie odniesień do kontrolek w zasobach w czasie działania

```
setContentView(R.layout.main);

TextView nameValue = (TextView) findViewById(R.id.nameValue);
nameValue.setText("Gall Anonim");
TextView addrValue = (TextView) findViewById(R.id.addrValue);
addrValue.setText("Ulica Sezamkowa 90210");
```

---

Powyższy kod nie jest skomplikowany, jednak należy zauważyc, że zanim wywołamy metodę `findViewById()`, wczytujemy zasób poprzez wywołanie `setContentView(R.layout.main)` — nie możemy odnieść zasobów do widoków, jeżeli zasoby nie zostały jeszcze załadowane.

Twórcy systemu Android wykonali kawał dobrej roboty, umożliwiając konfigurowanie właściwie każdego aspektu kontrolek za pomocą kodu lub języka XML. Preferowanym rozwiązańiem jest ustanawianie atrybutów kontrolek w pliku XML, bez konieczności używania kodu. Jednak będzie jeszcze wiele okazji do wykorzystania kodu, na przykład w przypadku ustanawiania wartości, która będzie wyświetlana użytkownikowi.

## FILL\_PARENT a MATCH\_PARENT

Stała `FILL_PARENT` została wycofana w wersji 2.2 i zastąpiono ją stałą `MATCH_PARENT`. Zmiana dotyczy jednak wyłącznie nazwy. Wartość tej stałej ciągle wynosi -1. Podobnie w przypadku układów graficznych tworzonych w języku XML — argument `fill_parent` został zastąpiony argumentem `match_parent`. Zatem jaka wartość jest tu właściwie stosowana? Zamiast stałej `FILL_PARENT` lub `MATCH_PARENT` moglibyśmy po prostu wprowadzić wartość -1 i nic by się nie stało. Nie jest ona jednak odczytywana w prosty sposób, poza tym nie istnieje równoważna, nienazwana wartość, którą można by zastosować do układów graficznych tworzonych w języku XML. Znamy lepsze rozwiązanie.

W zależności od poziomu interfejsu API, który chcemy wykorzystać w aplikacji, możemy stworzyć program wykorzystujący wersję Androida starszą od 2.2 i liczyć na kompatybilność w przód albo przygotować aplikację pod kątem wersji co najmniej 2.2 i ustawić w argumencie `minSdkVersion` najstarszą wersję systemu, na jakiej nasze dzieło będzie pracować. Jeśli na przykład wystarczą nam funkcje zawarte w wersji 1.6 Androida, to właśnie dla niej tworzymy aplikację i wykorzystujemy argumenty `FILL_PARENT` oraz `fill_parent`. Powinna ona bez problemu działać również w nowszych wersjach systemu. Jeżeli wymagana jest funkcjonalność wersji 2.2 Androida, piszemy przystosowany do niej program, stosujemy `MATCH_PARENT` i `match_parent`, natomiast w argumencie `minSdkVersion` wstawiamy wartość starszej wersji interfejsów API, na przykład 4 (odpowiada ona wersji 1.6 Androida). Możemy wdrożyć aplikację napisaną pod wersję 2.2 Androida do jego starszej wersji, musimy jednak uważać na stosowane klasy oraz (lub) metody, które są w niej niedostępne. Zawsze znajdzie się jakieś rozwiązanie, choćby stosowanie refleksji lub klas osłonowych, aby zniwelować różnice pomiędzy wersjami Androida. Nie zajmujemy się tu jednak tym zagadnieniem.

## Standardowe kontrolki Androida

Zajmiemy się teraz omówieniem standardowych kontrolek, dostępnych w zestawie Android SDK. Rozpoczniemy od kontrolek tekstu, a następnie przejdziemy do przycisków, pól wyboru, przycisków opcji, list, siatek, kontrolki daty i czasu oraz widoku mapy. Przedyskutujemy także kwestię kontrolek układu graficznego.

## Kontrolki tekstu

Prawdopodobnie kontrolki tekstu są pierwszym rodzajem kontrolek, na które natykają się programiści. Android posiada pełny, lecz nieprzytaczający ogromem zestaw kontrolek tekstu. W kolejnych podpunktach omówimy kontrolki `TextView`, `EditText`, `AutoCompleteTextView` oraz `MultiAutoCompleteTextView`. Na rysunku 6.2 pokazaliśmy działanie tych kontrolek.



Rysunek 6.2. Kontrolki tekstu w Androidzie

### TextView

Widzieliśmy już prostą specyfikację kontrolki `TextView` w języku XML (listing 6.3) oraz sposób jej definiowania w kodzie (listing 6.4). Zwrócmy uwagę na sposób określania identyfikatora, szerokości, wysokości oraz wartości tekstu w pliku XML oraz mechanizm ustanawiania wartości za pomocą metody `setText()`. Kontrolka `TextView` wyświetla tekst, jednak nie pozwala na jego edycję. Można by wysnuć wniosek, że jest to jedynie zwykła etykieta. Nieprawda. Kontrolka ta posiada kilka interesujących właściwości, dzięki którym staje się bardzo przydatna. Jeżeli na przykład wiadomo, że zawartością kontrolki `TextView` będzie adres URL lub adres e-mail, można ustawić właściwość `autoLink` wobec obiektu `email|web`, dzięki czemu kontrolka znajdzie i podświetli dany adres URL lub e-mail. Co więcej, kiedy użytkownik kliknie jeden z podświetlonych elementów, system uruchomi aplikację pocztową z otwartą do edycji wiadomością z już wpisany adresem e-mail lub przeglądarkę stron WWW z wpisany adresem URL. W języku XML atrybut ten znajdowałby się wewnątrz znacznika `TextView` i wyglądałby następująco:

```
<TextView ... android:autoLink="email|web" ... />
```

gdzie określamy ograniczony zbiór wartości, takich jak `web`, `email`, `phone`, `map` lub `none` (domyślnie) albo `all`. Jeżeli chcemy ustawić właściwość `autoLink` w kodzie, a nie w pliku XML, odpowiednie wywołanie metody nosi nazwę `setAutoLinkMask()`. Odczytuje ona wartości typu `int`, reprezentujące podobne do widzianego wcześniej połączenie wartości, na przykład `Linkify.EMAIL_ADDRESSES|Linkify.WEB_ADDRESSES`. W tym celu kontrolka `TextView` wykorzystuje klasę `android.text.util.Linkify`. Na listingu 6.6 przedstawiono przykład automatycznego korzystania z łączy za pomocą kodu.

**Listing 6.6.** Zastosowanie klasy Linkify z kontrolką TextView

```
TextView tv = (TextView) this.findViewById(R.id.tv);
tv.setAutoLinkMask(Linkify.ALL);
tv.setText("Odwiedź moją stronę, http://www.androidbook.com
lub napisz do mnie na adres davemac327@gmail.com.");
```

---

Zwróćmy uwagę, że opcje automatycznego korzystania z łączy konfigurujemy w kontrolce TextView przed ustawieniem tekstu. Jest to ważne, ponieważ ustawienie tych opcji po wpisaniu tekstu nie wpłynie na ten istniejący tekst. Ponieważ hiperłącza dodajemy w tekście za pomocą kodu, nasz fragment języka XML dotyczący kontroliki TextView z listingu 6.6 nie wymaga żadnych dodatkowych atrybutów i może być taki prosty:

```
<TextView android:id="@+id/tv" android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>
```

Jeśli chcemy, możemy przywołać statyczną metodę addLinks() klasy Linkify w celu znalezienia łączy oraz dodania ich do kontrolek TextView lub Spannable. Zamiast korzystać z metody setAutoLinkMask(), można wpisać poniższy wiersz już *po* wstawieniu tekstu:

```
Linkify.addLinks(tv, Linkify.ALL);
```

Kliknięcie takiego łącza powoduje wywołanie domyślnej intencji tego działania. Na przykład po kliknięciu adresu URL zostanie uruchomiona przeglądarka internetowa z wklejonym adresem. Kliknięcie numeru telefonu otworzy ekran wybierania numeru itd. Klasa Linkify może sprzągać te czynności bez najmniejszego problemu.

Klasa Linkify umożliwia również wykrywanie niestandardowych wzorców, określa, czy dany obiekt może zareagować na kliknięcie, a także decyduje, w jaki sposób uruchomić intencję wywołującą jakieś działanie po kliknięciu. Nie będziemy wnikać w szczegóły, warto jednak wiedzieć, że takie opcje są dostępne.

Istnieje dużo więcej funkcjonalności kontrolki TextView, takich jak atrybuty fontów, minLines i maxLines, a także wiele, wiele innych. Większość z nich posiada dość oczywiste przeznaczenie i zachęcamy Czytelników do eksperymentowania z nimi. Należy jednak pamiętać, że niektóre funkcje tej klasy nie znajdą zastosowania dla pól tylko do odczytu. Do obsługi pól edytowalnych służą odpowiednie podklasy, z których jedna została omówiona poniżej.

## EditText

Kontrolka EditText jest kontrolką klasy TextView. Jak sugeruje jej nazwa, umożliwia edytowanie tekstu. EditText nie jest tak rozbudowana jak niektóre kontrolki tego typu dostępne w internecie, jednak użytkownicy urządzeń obsługujących Androida nie będą raczej tworzyli w nich rozbudowanych dokumentów — co najwyżej kilka akapitów. Zatem klasa ta posiada ograniczony, lecz rozsądnie dobrany zestaw funkcji, który może nawet zaskoczyć niejedną osobę. Przykładowo jedną z najważniejszych właściwości kontrolki EditText jest `inputType`. Możemy skonfigurować właściwość `inputType` w taki sposób, aby flaga `textAutoCorrect` poprawiała pospolitsze błędy w trakcie pisania. Właściwość `textCapWords` powoduje przekształcanie małych liter w duże na początku zdania, wyrazów itd. Istnieją również pewne opcje stosowane wyłącznie dla numerów telefonów, hasel itp.

Istnieją starsze, obecnie uznane za przestarzałe, sposoby definiowania dużych liter, tekstu mieszczącego się w wielu wierszach oraz wielu innych cech. Jeżeli zostaną one określone przy braku właściwości `inputType`, będą normalnie odczytywane, jednak jeśli gdziekolwiek właściwość `inputType` zostanie zdefiniowana, starsze typy właściwości będą ignorowane.

Dawnym domyślnym zachowaniem kontrolki `EditText` było wyświetlanie tekstu w jednym wierszu oraz, w razie potrzeby, w kolejnych wierszach. Inaczej mówiąc, jeżeli użytkownik wypełnił tekstem cały pierwszy wiersz, pojawiał się wiersz drugi, trzeci i tak dalej. Można było jednak wymusić korzystanie tylko z jednego wiersza poprzez ustawienie wartości `true` we właściwości `singleLine`. W takim wypadku użytkownik musiał zmieścić cały tekst w jednym wierszu. W przypadku właściwości `inputType`, jeżeli nie zdefiniujemy właściwości `textMultiLine`, kontrolka `EditText` będzie domyślnie ograniczona wyłącznie do jednej linii tekstu. Jeśli więc chcemy wprowadzić dawny typ domyślnego zachowania, umożliwiający pisanie w wielu wierszach, musimy we właściwości `inputType` ustawić flagę `textMultiLine`.

Jedną z przyjemniejszych funkcji kontrolki `EditText` jest możliwość określenia tekstu podpowiedzi. Taka podpowiedź będzie wyświetlana nieco jaśniej i zniknie w momencie, gdy użytkownik zacznie wpisywać swój tekst. Zadaniem podpowiedzi jest powiadomienie użytkownika o przeznaczeniu danego pola tekstowego, bez konieczności zaznaczania i usuwania domyślnego tekstu. W pliku XML atrybut ten wygląda następująco: `android:hint="Tutaj wprowadzamy tekst podpowiedzi"` lub `android:hint="@string/nazwa_podpowiedzi"`, gdzie `nazwa_podpowiedzi` stanowi nazwę ciągu znaków umieszczonego w pliku `/res/values/strings.xml`. Pisząc kod, wywołujemy metodę `setHint()`, której argumentem jest ciąg znaków `CharSequence` lub identyfikator zasobu.

## AutoCompleteTextView

Kontrolka `AutoCompleteTextView` jest obiektem klasy `TextView` z funkcjonalnością automatycznego wypełniania. Innymi słowy, podczas pisania tekstu przez użytkownika w oknie `TextView` będą wyświetlane sugestie dokonania wyrazu. Na listingu 6.7 zaprezentowano kod kontrolki `AutoCompleteTextView`.

**Listing 6.7.** Stosowanie kontrolki `AutoCompleteTextView`

```
<AutoCompleteTextView android:id="@+id/actv"
    android:layout_width="fill_parent" android:layout_height="wrap_content" />

AutoCompleteTextView actv = (AutoCompleteTextView) this.findViewById(R.id.actv);

ArrayAdapter<String> aa = new ArrayAdapter<String>(this,
    android.R.layout.simple_dropdown_item_1line,
new String[] {"Angielski", "Hebrajski", "Hindi", "Hiszpański", "Niemiecki", "Grecki" });

actv.setAdapter(aa);
```

Kontrolka `AutoCompleteTextView` przedstawiona na listingu 6.7 sugeruje użytkownikowi określony język. Jeżeli na przykład zostanie wpisany tekst `an`, kontrolka zasugeruje język angielski. Po wpisaniu `gr` zostanie zaproponowany język grecki i tak dalej.

Kontrolki z funkcją podpowiedzi lub podobne kontrolki z funkcją automatycznego wypełniania składają się z dwóch części: kontrolki widoku tekstu oraz kontrolki odpowiedzialnej za wyświetlanie podpowiedzi. Taka jest ogólna koncepcja. Żeby móc stosować taką kontrolkę, należy

utworzyć najpierw ją, a następnie listę podpowiedzi, którą trzeba przypisać do tej kontrolki. Trzeba też określić sposób wyświetlania podpowiedzi. Ewentualnie można utworzyć drugą kontrolkę w celu wyświetlania podpowiedzi, a następnie powiązać obydwie kontrolki ze sobą.

Automatyczne wprowadzanie tekstu w Androidzie jest proste, czego dowodem jest listing 6.7. Żeby korzystać z kontrolki AutoCompleteTextView, można ją zdefiniować w pliku układu graficznego, a następnie odnosić się do niego w aktywności. Następnie tworzy się klasę przejściową (ang. *adapter class*), przechowującą podpowiedzi, oraz definiuje się identyfikator kontrolki, która będzie je wyświetlała (w tym przypadku prostą listę elementów). Drugi parametr przedstawionego na listingu 6.7 obiektu ArrayAdapter wskazuje klasie przejściowej, że podpowiedzi mają być przedstawiane w postaci prostej listy. Ostatnim krokiem jest powiązanie klasy przejściowej z kontrolką AutoCompleteTextView za pomocą metody `setAdapter()`. Nie przejmujmy się na razie adapterami. Zajmiemy się nimi w dalszej części rozdziału.

## **MultiAutoCompleteTextView**

Osoby korzystające z kontrolki AutoCompleteTextView wiedzą, że oferuje ona jedynie podpowiedzi dla całego tekstu w oknie widoku. Inaczej mówiąc, podczas pisania zdania nie będą wyświetlane sugestie dla każdego wyrazu oddzielnie. Do takich zastosowań jest przeznaczona kontrolka MultiAutoCompleteTextView. Służy ona do wyświetlania podpowiedzi w trakcie wpisywania tekstu przez użytkownika. Na rysunku 6.2 pokazano, że użytkownik wpisał wyraz Angielski, a po przecinku Ni, co powoduje wyświetlenie sugerowanego wyrazu Niemiecki. Jeżeli użytkownik będzie wypisywał nazwy innych języków, aplikacja wyświetli kolejne sugestie.

Kontrolki MultiAutoCompleteTextView używa się tak samo jak obiektu AutoCompleteTextView. Różnica polega na konieczności określenia miejsca wyświetlania kolejnej podpowiedzi. Na przykład na rysunku 6.2 pokazano, że sugestie mogą być proponowane na początku zdania oraz po przecinku. Konieczne jest zatem umieszczenie tokenizera analizującego zdanie oraz wskazującego kontrolce MultiAutoCompleteTextView, kiedy ma wyświetlić kolejną podpowiedź. Na listingu 6.8 zaprezentowano plik XML oraz kod kontrolki MultiAutoCompleteTextView.

---

### **Listing 6.8. Stosowanie kontrolki MultiAutoCompleteTextView**

---

```
<MultiAutoCompleteTextView android:id="@+id/mactv"
    android:layout_width="fill_parent" android:layout_height="wrap_content" />

MultiAutoCompleteTextView mactv = (MultiAutoCompleteTextView) this
    .findViewById(R.id.mactv);
ArrayAdapter<String> aa2 = new ArrayAdapter<String>(this,
    android.R.layout.simple_dropdown_item_1line,
    new String[] {"Angielski", "Hebrajski", "Hindi", "Hiszpański", "Niemiecki",
    "Grecki" });
mactv.setAdapter(aa2);
mactv.setTokenizer(new MultiAutoCompleteTextView.CommaTokenizer());
```

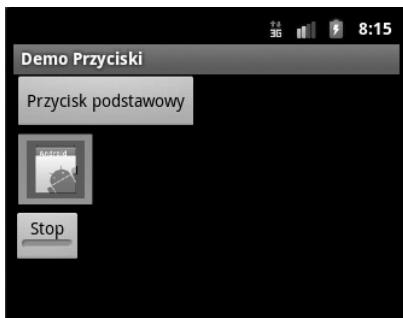
---

Jedyną istotną różnicą pomiędzy listingu 6.7 a 6.8 jest zastosowanie klasy MultiAutoCompleteTextView oraz wywołanie metody `setTokenizer()`. Ponieważ w tym wypadku uwzględniono obiekt CommaTokenizer, po wprowadzeniu przecinka w polu *EditText* znów pojawi się okno podpowiedzi korzystające z tablicy ciągów znaków. Inne znaki nie spowodują wyświetlenia pola sugestii. Zatem jeśli nawet użytkownik wpisze wyrazy Francuski Hiszpań, nie pojawi się sugestia dokończenia drugiego słowa, ponieważ przed nim nie ma przecinka.

Innym rodzajem tokenizera obsługiwianego przez Androida jest ten przeznaczony dla adresów e-mail. Nosi on nazwę Rfc822Tokenizer. W razie potrzeby zawsze można utworzyć swój własny tokenizer.

## Kontrolki przycisków

Przyciski są standardem w każdym środowisku obsługującym widżety, a Android nie jest wyjątkiem. Do dyspozycji mamy typowy zestaw przycisków oraz kilka dodatków. W następnych podpunktach zajmiemy się trzema rodzajami kontrolek przycisków: przyciskiem podstawowym, przyciskiem obrazkowym oraz przyciskiem przełączania. Na rysunku 6.3 został pokazany interfejs użytkownika zawierający wszystkie trzy rodzaje przycisków. Są to kolejno: przycisk podstawowy, przycisk obrazkowy oraz przycisk przełączania.



Rysunek 6.3. Kontrolki przycisków w Androidzie

Zajmijmy się najpierw przyciskiem podstawowym.

### Kontrolka Button

Klasą przycisku podstawowego w Androidzie jest `android.widget.Button`. Niewiele można powiedzieć na temat tego rodzaju przycisków, poza omówieniem obsługi zdarzeń wyzwalanych kliknięciem. Na listingu 6.9 został zaprezentowany fragment układu graficznego zapisanego w języku XML dla kontrolki Button, a także kod Java, który można wprowadzić do metody `onCreate()` naszej aktywności. Taki podstawowy przycisk będzie wyglądał tak jak górny przycisk widoczny na rysunku 6.3.

**Listing 6.9.** Obsługa zdarzeń wyzwalanych kliknięciem w obiekcie Button

```
<Button android:id="@+id/ccbtn1"
    android:text="@string/basicBtnLabel"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />

Button btn = (Button)this.findViewById(R.id.ccbtn1);
btn.setOnClickListener(new OnClickListener()
{
    public void onClick(View v)
    {
        Intent intent = new Intent(Intent.ACTION_VIEW,
            Uri.parse("http://www.androidbook.com"));
    }
});
```

```
        startActivity(intent);
    }
});
```

---

Na listingu 6.9 zaprezentowano sposób rejestrowania zdarzenia wywołanego kliknięciem. Dokonuje się tego poprzez wywołanie metody `setOnClickListener()` wobec interfejsu `onClick Listener`. Aby obsłużyć zdarzenia wywoływanego kliknięciem obiektu (przycisku) `btn`, na bieżąco utworzono anonimowy obiekt nasłuchujący. Kliknięcie przycisku powoduje wywołanie metody `onClick()` obiektu nasłuchującego, co w naszym przypadku powoduje otwarcie okna przeglądarki.

Wraz z wydaniem środowiska Android SDK w wersji 1.6 wprowadzono łatwiejszy sposób konfigurowania obsługi kliknięcia przycisku (przycisków). Na listingu 6.10 został ukazany fragment XML dla obiektu `Button`, w którym określamy atrybut procedury obsługi, a także kod Java stanowiący procedurę obsługi kliknięcia.

#### **Listing 6.10. Konfigurowanie procedury obsługi kliknięcia dla przycisku**

---

```
<Button ... android:onClick="myClickHandler" ... />

public void myClickHandler(View target) {
    switch(target.getId()) {
        case R.id.ccbtn1:
        ...
    }
}
```

---

Dla obiektu klasy `View`, reprezentującego naciśnięty przycisk, nastąpi wywołanie funkcji obsługi kliknięcia wraz z zestawem usług oczekiwanych od tej funkcji. Należy zwrócić uwagę, w jaki sposób instrukcja `switch` zawarta w omawianej metodzie obsługi kliknięcia wykorzystuje identyfikatory zasobów przycisku do uruchomienia procesu. Stosowanie tej metody oznacza, że obiekty klasy `Button` nie będą jawnie tworzone w kodzie oraz że ta sama metoda może być wykorzystywana także do obsługi wielu przycisków. Dzięki temu struktura interfejsu staje się bardziej zrozumiała i przejrzysta. Metoda ta działa również w przypadku pozostałych typów przycisków. Nie zadziała ona jednak w wersji 1.5 Androida i starszych. Nie pojawi się informacja o błędzie; po prostu kliknięcie przycisku nie wywoła żadnej reakcji.

## **Kontrolka ImageButton**

Przyciski obrazkowe są dostępne w Androidzie dzięki klasie `android.widget.ImageButton`. Używanie tego rodzaju obiektu przypomina korzystanie z przycisku podstawowego (listing 6.11). Przycisk obrazkowy będzie przypominał środkowy przycisk, widoczny na rysunku 6.3.

#### **Listing 6.11. Stosowanie kontrolki ImageButton**

---

```
<ImageButton android:id="@+id/imageBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
    android:onClick="myClickHandler"
    android:src="@drawable/icon" />

ImageButton btn = (ImageButton)this.findViewById(R.id.imageBtn);
btn.setImageResource(R.drawable.icon);
```

---

W tym fragmencie kodu utworzyliśmy w pliku XML przycisk obrazkowy. Obraz dla tego przycisku znajduje się w zasobach typu `drawable`. Plik z tym obrazem musi się znajdować w katalogu `/res/drawable`. W naszym przypadku wykorzystaliśmy domyślną ikonę Androida. Na listingu 6.11 pokazaliśmy również dynamiczny sposób konfiguracji przycisku obrazkowego poprzez wywołanie metody `setImageResource()` na przycisku i przekazanie jej identyfikatora zasobu. Warto zapamiętać, że wystarczy zastosować tylko jeden z tych dwóch sposobów. Nie trzeba definiować przycisku obrazkowego jednocześnie w kodzie oraz w pliku XML.

Interesującą funkcją przycisku obrazkowego jest możliwość ustawienia przezroczystego tła. W ten sposób dowolny obraz można ustawić tak, aby zachowywał się jak przycisk.

Ponieważ przycisk obrazkowy może się zasadniczo różnić od zwykłego przycisku, można do stosować jego wygląd, gdy znajduje się w dwóch pozostałych stanach. Warto bowiem przypomnieć, że oprócz normalnego stanu przyciski mogą się znaleźć w stanie uaktywnienia oraz zostać wciśnięte. Stan **uaktywnienia** oznacza po prostu, że przycisk znajduje się w stanie gotowości. Możemy uaktywnić przycisk za pomocą klawiszy strzałek klawiatury lub D-pada<sup>1</sup>. Przycisk jest **wciśnięty**, gdy jego wygląd zmienia się po wciśnięciu, ale użytkownik nie zdąży go jeszcze puścić. Aby zdefiniować trzy obrazy dla jednego przycisku oraz przypisać każdy z nich do określonego stanu, konfigurujemy selektor. Jest to niewielki plik XML, umieszczony w katalogu `/res/drawable` projektu. Jest to zachowanie cokolwiek sprzeczne z logiką, ponieważ w katalogu tym umieszczamy plik XML, a nie rysunek. Mimo to właśnie tutaj musi się znaleźć selektor. Zawartość pliku selektora została ukazana na listingu 6.12.

#### **Listing 6.12.** Wykorzystanie selektora wraz z kontrolką `ImageButton`

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_pressed="true"
        android:drawable="@drawable/button_pressed" /> <!-- wciśnięty -->
    <item android:state_focused="true"
        android:drawable="@drawable/button_focused" /> <!-- uaktywniony -->
    <item android:drawable="@drawable/icon" /> <!-- domyślny -->
</selector>
```

Należy zapamiętać kilka informacji o selektorze. Po pierwsze, nie definiujemy znacznika `<resources>` spotykanego w innych plikach XML. Po drugie, istotna jest kolejność wymieniania w selektorze obrazów przypisanych do poszczególnych stanów przycisku. Android sprawdza dopasowanie każdego obiektu umieszczonego w selektorze zgodnie z ich kolejnością, my natomiast chcemy, aby obraz przypisany do normalnego stanu przycisku był stosowany wyłącznie wtedy, gdy przycisk nie jest wciśnięty ani uaktywniony. Jeżeli obraz przypisany do normalnego stanu przycisku zostanie wskazany na początku selektora, będzie zawsze wybierany, nawet jeśli przycisk zostanie uaktywniony lub wciśnięty. Oczywiście pliki obiektów typu `drawable` muszą się znajdować w katalogu `/res/drawables`. W końcu, definiując przycisk za pomocą pliku XML, powinniśmy ustawić powiązanie z selektorem we właściwości `android:src`, tak jakbyśmy mieli do czynienia z zasobem typu `drawable`, na przykład:

```
<Button ... android:src="@drawable/imagebuttonselector" ... />
```

<sup>1</sup> tzw. krzyżak, kontroler stosowany w konsolach do gier — przyp. red.

## Kontrolka ToggleButton

Kontrolka ToggleButton, taka jak pole wyboru lub przycisk opcji, reprezentuje kategorię przycisków dwustanowych. Przycisk taki może się znajdować w stanie włączonym lub wyłączenym. Domyslnym zachowaniem przycisku ToggleButton jest wyświetlanie zielonego paska w stanie włączonym i wyszarzonego w stanie wyłączenym. Co więcej, tekst przycisku brzmi `On`, gdy przycisk jest włączony, i zmienia się na `Off` po jego wyłączeniu. Istnieje możliwość modyfikowania tekstu pojawiającego się w poszczególnych stanach kontrolki ToggleButton, jeżeli domyslnie ustawienia nie pasują do tworzonej aplikacji. Jeśli na przykład taki przycisk ma umożliwiać kontrolę procesu przebiegającego w tle, można umieścić wyrazy *Uruchom* oraz *Zatrzymaj* poprzez zdefiniowanie właściwości `android:textOn` oraz `android:textOff`.

Na listingu 6.13 ukazano przykład. Przycisk przełączania jest widoczny na dole rysunku 6.3 i znajduje się w pozycji *On*, więc na etykiecie umieszczonej pod nim widnieje napis *Stop*.

**Listing 6.13.** Przycisk przełączania w Androidzie

---

```
<ToggleButton android:id="@+id/cctglBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Przycisk przełączania"
    android:textOn="Uruchom"
    android:textOff="Zatrzymaj"/>
```

---

Ponieważ teksty wyświetlane podczas włączenia lub wyłączenia przycisku ToggleButton są oddzielnymi atrybutami, atrybut `android:text` właściwie nie jest wykorzystywany. Jest on dostępny, ponieważ został odziedziczony (po obiekcie `TextView`), jednak w tym przypadku okazuje się niepotrzebny.

## Kontrolka CheckBox

Kontrolka CheckBox jest kolejnym przyciskiem dwustanowym, umożliwiającym użytkownikowi wybranie stanu. Różnica polega na tym, że w wielu sytuacjach użytkownicy nie postrzegają jej jako przycisku bezpośrednio wywołującego akcję. Jednak z punktu widzenia programisty Androida takie pole wyboru jest przyciskiem i można na nim wykonywać te same czynności co na przycisku.

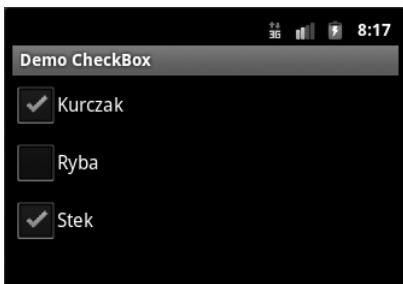
W Androidzie pole wyboru jest tworzone poprzez ustanowienie instancji klasy `android.widget.CheckBox` (listing 6.14 oraz rysunek 6.4).

**Listing 6.14.** Tworzenie pól wyboru

---

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <CheckBox android:id="@+id/chickenCB" android:text="Kurczak" android:checked="true"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <CheckBox android:id="@+id/fishCB" android:text="Ryba"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <CheckBox android:id="@+id/steakCB" android:text="Stek" android:checked="true"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

---



**Rysunek 6.4.** Zastosowanie kontrolki CheckBox

Zarządzanie stanem pola wyboru odbywa się za pomocą wywołania metod `setChecked()` lub `toggle()`. Informacje o stanie uzyskiwane są dzięki wywołaniu metody `isChecked()`.

Jeżeli po zaznaczeniu pola wyboru lub usunięciu tego zaznaczenia ma nastąpić określone wydarzenie, można je zarejestrować poprzez wywołanie metody `setOnCheckedChangeListener()` wraz z implementacją interfejsu `OnCheckedChangeListener`. Konieczne będzie także zaimplementowanie metody `onCheckedChanged()`, wywoływanej po zaznaczeniu lub usunięciu zaznaczenia pola wyboru. Na listingu 6.15 przedstawiamy kod zajmujący się obsługą kontrolki CheckBox.

**Listing 6.15.** Stosowanie kontrolek CheckBox w kodzie

```
public class CheckBoxActivity extends Activity {
    /** Wywołane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.checkbox);

        CheckBox fishCB = (CheckBox)findViewById(R.id.fishCB);

        if(fishCB.isChecked())
            fishCB.toggle(); // jeżeli pole wyboru było zaznaczone, zaznaczenie zostaje usunięte

        fishCB.setOnCheckedChangeListener(
            new CompoundButton.OnCheckedChangeListener() {

                @Override
                public void onCheckedChanged(CompoundButton arg0, boolean isChecked) {
                    Log.v("CheckBoxActivity", "Pole wyboru Ryba jest teraz "
                        + (isChecked?"zaznaczone":"niezaznaczone"));
                }
            });
    }
}
```

Ciekawą możliwością konfiguracji obiektu nasłuchującego `OnCheckedChangeListener` jest przekazywanie nowego stanu przycisku CheckBox. Alternatywnie można by wykorzystać element nasłuchujący `onClickListener`, tak jak pokazano w przypadku podstawowych przycisków. Podczas wywołania metody `onClick()` trzeba by wtedy samodzielnie zdefiniować nowy stan przycisku, odpowiednio oddając jego właściwości, a także zaprogramować wywołanie

metody `isChecked()` dla tego stanu. Listing 6.16 pokazuje w analogiczny sposób, jak mógłby wyglądać kod w przypadku dodania wiersza `android:onClick="myClickHandler"` do definicji XML przycisku CheckBox (pamiętajmy, że funkcja ta zadziała dopiero od wersji 1.6 Androida).

---

**Listing 6.16.** Stosowanie przycisków CheckBox wraz z właściwością `android:onClick`

```
public void myClickHandler(View view) {  
    switch(view.getId()) {  
        case R.id.steakCB:  
            Log.v("CheckBoxActivity", "Pole zaznaczenia Stek jest teraz " +  
                (((CheckBox)view).isChecked()?"zaznaczone":"niezaznaczone"));  
    }  
}
```

---

## Kontrolka RadioButton

Kontrolki przycisków opcji są integralnym elementem każdego środowiska projektowego interfejsów UI. Przycisk opcji daje użytkownikowi kilka możliwości wyboru, ale tylko jedna z nich może zostać zaznaczona. Żeby taki model działał skutecznie, przyciski opcji przeważnie należą do grupy. W takiej grupie w danym momencie może być zaznaczony tylko jeden przycisk opcji.

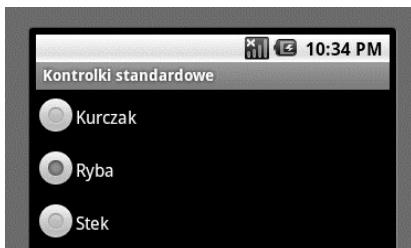
Żeby utworzyć grupę przycisków opcji w Androidzie, należy najpierw stworzyć element `RadioGroup`, a następnie wypełnić go tymi przyciskami. Na listingu 6.17 został zaprezentowany przykład, a rysunek 6.5 stanowi jego ilustrację.

---

**Listing 6.17.** Stosowanie widżetów RadioButton w Androidzie

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical" android:layout_width="fill_parent"  
    android:layout_height="fill_parent">  
  
<RadioGroup android:id="@+id/rBtnGrp" android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:orientation="vertical" >  
  
<RadioButton android:id="@+id/chRBtn" android:text="Kurczak"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"/>  
  
<RadioButton android:id="@+id/fishRBtn" android:text="Ryba" android:checked="true"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"/>  
  
<RadioButton android:id="@+id/stkRBtn" android:text="Stek"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"/>  
  
</RadioGroup>  
</LinearLayout>
```

---



**Rysunek 6.5.** Używanie przycisków opcji

W Androidzie grupa opcji jest implementowana za pomocą klasy `android.widget.RadioGroup`, a przycisk opcji — poprzez klasę `android.widget.RadioButton`.

Zwróćmy uwagę, że wszystkie przyciski opcji wewnętrz grupy opcji są domyślnie niezaznaczone, chociaż istnieje możliwość zaznaczenia jednego z nich w definicji XML, co zrobiliśmy w przypadku opcji *Ryba*. Żeby zaprogramować takie predefiniowane zaznaczenie jednego z przycisków opcji, można utworzyć odniesienie do tego przycisku i wywołać metodę `setChecked()`:

```
RadioButton rbtn = (RadioButton)this.findViewById(R.id.stkRBtn);
rbtn.setChecked(true);
```

Można również wykorzystać metodę `toggle()` do przełączania stanów przycisku. Podobnie jak w przypadku kontrolki `CheckBox`, po wywołaniu metody `setOnCheckedChangeListener()` wraz z implementacją interfejsu `OnCheckedChangeListener` przy każdym zdarzeniu zaznaczenia przycisku opcji lub cofnięcia zaznaczenia będzie wyświetlane powiadomienie. Istnieje tu jednak pewna różnica. W istocie mamy tutaj do czynienia z inną klasą niż poprzednio. Tym razem, patrząc z technicznego punktu widzenia, jest to klasa `RadioGroup.OnCheckedChangeListener`, a nie — jak poprzednio — `CompoundButton.OnCheckedChangeListener`.

W elemencie `RadioGroup` mogą zostać umieszczone również inne widoki, nie tylko przyciski opcji. Na przykład na listingu 6.18 po ostatnim przycisku opcji została dodana kontrolka `TextView`. Warto też zauważyć, że przycisk opcji został umieszczony poza grupą opcji.

**Listing 6.18.** Grupa opcji zawierająca nie tylko przyciski opcji

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <RadioButton android:id="@+id/anotherRadBtn"
        android:text="Na zewnątrz"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <RadioGroup android:id="@+id/radGrp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
        <RadioButton android:id="@+id/chRBtn"
            android:text="Kurczak"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <RadioButton android:id="@+id/fishRBtn"
            android:text="Ryba"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
    </RadioGroup>
</LinearLayout>
```

```
        android:text="Ryba"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
<RadioButton android:id="@+id/stkRBtn"
        android:text="Stek"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

<TextView android:text="Moje ulubione danie"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</RadioGroup>

</LinearLayout>
```

---

Listing 6.18 stanowi dowód, że można umieścić kontrolki niebędące częścią klasy RadioGroup wewnętrznie grupy opcji. Powinniśmy także wiedzieć, że grupa opcji może wymuszać zaznaczenie tylko jednego przycisku jedynie wobec przycisków opcji znajdujących się w tym pojemniku. Inaczej mówiąc, przycisk opcji o identyfikatorze anotherRadBtn nie będzie objęty działaniem grupy opcji przedstawionej na listingu 6.18, ponieważ nie jest jej elementem podlegającym.

Istnieje możliwość programowego sterowania obiektami klasy RadioGroup. Na przykład można w ten sposób uzyskać odniesienie do grupy opcji oraz dodać przycisk opcji (lub inny rodzaj kontrolki). Koncepcja ta została zademonstrowana na listingu 6.19.

#### **Listing 6.19.** Dodanie w kodzie kontrolki RadioButton do pojemnika RadioGroup

---

```
RadioGroup radGrp = (RadioGroup)findViewById(R.id.radGrp);
RadioButton newRadioBtn = new RadioButton(this);
newRadioBtn.setText("Wieprzowina");
radGrp.addView(newRadioBtn);
```

---

Po zaznaczeniu przez użytkownika przycisku opcji w grupie opcji nie będzie można usunąć tego zaznaczenia za pomocą powtórnego kliknięcia. Jedynym sposobem usunięcia zaznaczenia wszystkich przycisków opcji w tej grupie jest wywołanie metody `clearCheck()` w obiekcie RadioGroup.

Oczywiście, Czytelnik może zechcieć wykorzystać klasę RadioGroup do czegoś bardziej interesującego. Prawdopodobnie nie chce za każdym razem sprawdzać, czy każdy przycisk RadioButton jest zaznaczony. Na szczęście klasa RadioGroup posiada kilka metod, które mogą się tu przydać. Przedstawiamy je na listingu 6.20. Odpowiednik XML tego kodu znajduje się na listingu 6.18.

#### **Listing 6.20.** Wykorzystanie klasy RadioGroup w sposób programowy

---

```
public class RadioGroupActivity extends Activity {
    protected static final String TAG = "RadioGroupActivity";

    /** Wywołana podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.radiogroup);
```

```
RadioGroup radGrp = (RadioGroup) findViewById(R.id.radGrp);

int checkedRadioButtonId = radGrp.getCheckedRadioButtonId();

radGrp.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(RadioGroup arg0, int id) {
        switch(id) {
            case -1:
                Log.v(TAG, "Wybrane elementy wyczyszczone!");
                break;
            case R.id.chRBtn:
                Log.v(TAG, "Wybrano kurczaka");
                break;
            case R.id.fishRBtn:
                Log.v(TAG, "Wybrano rybę");
                break;
            case R.id.stkRBtn:
                Log.v(TAG, "Wybrano stek");
                break;
            default:
                Log.v(TAG, "He?");
                break;
        }
    }
});
```

Zawsze można pobrać najnowszy zaznaczony obiekt RadioButton za pomocą metody `getCheckedRadioButtonId()`, która zwraca identyfikator zaznaczonego obiektu lub wartość -1, jeśli żadna opcja nie zostało zaznaczona (być może nie wprowadzono domyślnej opcji, a użytkownik jeszcze żadnej nie wybrał). Poprzednio zademonstrowaliśmy ją w metodzie `onCreate()`, w rzeczywistości jednak powinniśmy ją wykorzystać w odpowiednim momencie w celu odczytania bieżącego zaznaczenia użytkownika. Możemy również wprowadzić obiekt nasłuchujący, który by od razu informował o wybraniu przez użytkownika jakieś opcji. Zwróćmy uwagę, że metoda `onCheckedChanged()` przyjmuje parametr `RadioGroup`, co pozwala nam na użycie tego samego obiektu nasłuchującego `OnCheckedChangeListener` dla wielu klas `RadioGroup`. Był może niektórzy Czytelnicy zauważyli wartość -1 instrukcji `switch`. Opcja ta zostanie wybrana, jeśli przyciski wyboru opcji klasy `RadioGroup` zostaną wyczyszczone programistycznie.

## Kontrolka ImageView

Jedną z najważniejszych kontrolek, których jeszcze nie omówiliśmy, jest `ImageView`. Jest ona stosowana do wyświetlania obrazów, pochodzących z plików, dostawców treści lub zasobów, na przykład typu `drawable`. Można również zdefiniować wyłącznie kolor, który kontrolka `ImageView` będzie wyświetlała. Na listingu 6.21 zaprezentowano kilka kontrolek `ImageView`, a następnie przedstawiono przykładowy kod ukazujący proces tworzenia tej klasy.

**Listing 6.21.** Kontrolki ImageView w pliku XML oraz w kodzie

```
<ImageView android:id="@+id/image1"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:src="@drawable/icon" />

<ImageView android:id="@+id/image2"
    android:layout_width="125dip" android:layout_height="25dip"
    android:src="#555555" />

<ImageView android:id="@+id/image3"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />

<ImageView android:id="@+id/image4"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:src="@drawable/manatee02"
    android:scaleType="centerInside"
    android:maxWidth="35dip" android:maxHeight="50dip"
/>

ImageView imgView = (ImageView)findViewById(R.id.image3);

imgView.setImageResource( R.drawable.icon );

imgView.setImageBitmap(BitmapFactory.decodeResource(
    this.getResources(), R.drawable.manatee14 ) );

imgView.setImageDrawable(
    Drawable.createFromPath("/mnt/sdcard/dave2.jpg") );

imgView.setImageURI(Uri.parse("file:///mnt/sdcard/dave2.jpg"));
```

---

W tym przykładzie definiujemy cztery obrazy za pomocą języka XML. Pierwszy stanowi po prostu ikonę naszej aplikacji. Drugi jest szarym paskiem, szerokim, ale niezbyt wysokim. Trzecia definicja nie wskazuje źródła obrazu w kodzie XML, natomiast przypisuje identyfikator (*image3*), za pomocą którego można programowo ustawić obraz. Czwarty obraz jest kolejnym z zasobów typu *drawable*, dla którego nie tylko określamy ścieżkę do pliku źródłowego, lecz również jego maksymalne rozmiary, a także wskazujemy, co ma się stać z tym obrazem, jeśli przekroczy nałożone ograniczenia rozmiarów. W tym przypadku klasa *ImageView* wyśrodkuje go i przeskaliuje do założonych rozmiarów.

W kodzie Java z listingu 6.21 widzimy kilka sposobów ustawiania obrazu *image3*. Oczywiście, najpierw musimy uzyskać odniesienie do kontrolki *ImageView* za pomocą identyfikatora zasobów. Pierwsza metoda ustawiania, *setImageResource()*, zwyczajnie wykorzystuje identyfikator obrazu do jego zlokalizowania oraz dostarczenia go kontrolce *ImageView*. Druga metoda ustawiania korzysta z klasy *BitmapFactory* do wczytania zasobu obrazu do obiektu *Bitmap*, a następnie ustanawia kontrolkę *ImageView* wobec tego obiektu. Warto wiedzieć, że w obiekcie *Bitmap* można wprowadzać pewne modyfikacje przed wczytaniem go do kontrolki *ImageView*, jednak w naszym przykładowym kodzie niczego nie zmieniamy. Ponadto klasa *BitmapFactory* zawiera kilka metod służących do tworzenia obiektu *Bitmap*, na przykład z tablicy bajtów albo z klasy *InputStream*. Moglibyśmy wykorzystać metodę *InputStream* do odczytania obrazu z serwera sieciowego, utworzyć obraz *Bitmap*, a następnie ustawić klasę *ImageView*.

Trzecie ustawienie określa obiekt `Drawable` jako źródło obrazu. W naszym przykładzie źródło to wskazuje plik znajdujący się na karcie SD. Żeby opisywany kod zadziałał, trzeba umieścić na karcie SD plik z odpowiednią nazwą. Podobnie jak miało to miejsce w przypadku klasy `BitmapFactory`, klasa `Drawable` posiada kilka metod pozwalających na tworzenie obiektów typu `Drawable`, w tym ze strumienia XML.

Ostatnia metoda ustawiania obrazu polega na pobraniu identyfikatora URI obrazu i wykorzystaniu go jako źródła obrazu. Nie należy jednak sądzić, że identyfikator URI każdego obrazu nadaje się do tego celu. Ta metoda służy do wykorzystywania obrazów dostępnych lokalnie, znajdujących się w urządzeniu, a nie obrazów wyszukiwanych w internecie. Aby takie obrazy internetowe mogły być źródłami dla kontrolki `ImageView`, najlepiej zastosować klasy `BitmapFactory` oraz `InputStream`.

## Kontrolki daty i czasu

Kontrolki daty i czasu są standardem w wielu zestawach widżetów. W Androidzie zawarto kilka kontrolek związańych z datą i czasem, niektóre z nich zostaną omówione w następnych podpunktach. W szczególności zajmiemy się kontrolkami `DatePicker`, `TimePicker`, `DigitalClock` oraz `AnalogClock`.

### Kontrolki DatePicker oraz TimePicker

Zgodnie z nazwami kontrolka `DatePicker` służy do wybierania daty, natomiast kontrolka `TimePicker` umożliwia ustawianie godziny. Listing 6.22 oraz rysunek 6.6 przedstawiają przykłady tych kontrolek.

**Listing 6.22.** Kontrolki DatePicker oraz TimePicker w kodzie XML

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView android:id="@+id/dateDefault"
        android:layout_width="fill_parent" android:layout_height="wrap_content" />

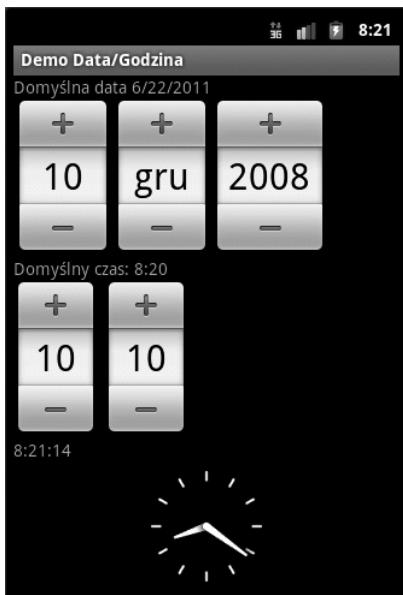
    <DatePicker android:id="@+id/datePicker"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />

    <TextView android:id="@+id/timeDefault"
        android:layout_width="fill_parent" android:layout_height="wrap_content" />

    <TimePicker android:id="@+id/timePicker"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />

</LinearLayout>
```

Jeżeli przyjrzeć się układowi graficznemu w kodzie XML, to można stwierdzić, że definiowanie tych kontrolek nie jest skomplikowane. Podobnie jak w przypadku innych kontrolek w Androidzie, także i w tym przypadku można zaprogramować je, żeby uruchamiały się lub żeby można było pobierać z nich dane. Na przykład kodinicjalizacji tych kontrolek może wyglądać tak jak na listingu 6.23.



Rysunek 6.6. Interfejsy UI kontrolek DatePicker i TimePicker

**Listing 6.23.** Inicjalizacja daty w kontrolce DatePicker oraz godziny w kontrolce TimePicker

---

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.datetimepicker);  
  
    TextView dateDefault = (TextView)findViewById(R.id.dateDefault);  
    TextView timeDefault = (TextView)findViewById(R.id.timeDefault);  
  
    DatePicker dp = (DatePicker)this.findViewById(R.id.datePicker);  
    // Wartość miesiąca rozpoczyna się od zera. Trzeba dodać 1 do wyświetlonej wartości  
    dateDefault.setText("Domyślana data " + (dp.getMonth() + 1) + "/" +  
        dp.getDayOfMonth() + "/" + dp.getYear());  
    // A tutaj odejmujemy 1 od wartości Grudzień (12), żeby był wyświetlany właściwy miesiąc  
    dp.init(2008, 11, 10, null);  
  
    TimePicker tp = (TimePicker)this.findViewById(R.id.timePicker);  
  
    java.util.Formatter timeF = new java.util.Formatter();  
    timeF.format("Domyślny czas %d:%02d", tp.getCurrentHour(),  
        tp.getCurrentMinute());  
    timeDefault.setText(timeF.toString());  
  
    tp.setIs24HourView(true);  
    tp.setCurrentHour(new Integer(10));  
    tp.setCurrentMinute(new Integer(10));  
}  
}
```

---

Kod na listingu 6.23 ustawia datę na 10 grudnia 2008 roku. Zwróćmy uwagę, że dla nazw miesięcy wartość wewnętrzna rozpoczyna się od zera, co oznacza, że styczeń posiada wartość 0, a grudzień — 11. W przypadku klasy TimePicker wybrano godzinę 10:10. Warto wiedzieć, że kontrolka obsługuje wyświetlanie czasu w formacie dwudziestoczątowogodzinnym. Jeżeli w tych kontrolkach nie zostaną ustawione żadne wartości, domyślnymi będą aktualne data i czas, skonfigurowane w urządzeniu.

Android wykorzystuje również te kontrolki jako okna dialogowe, na przykład DatePickerDialog oraz TimePickerDialog. Kontrolki te przydają się, w przypadku gdy mają zostać wyświetlone użytkownikowi w celu zmuszenia go do dokonania jakiegoś wyboru. Okna dialogowe zostały szczegółowo omówione w rozdziale 8.

## Kontrolki DigitalClock i AnalogClock

Na rysunku 6.7 przedstawiliśmy dostępne w Androidzie kontrolki DigitalClock oraz AnalogClock.



**Rysunek 6.7.** Zastosowanie kontrolek AnalogClock i DigitalClock

Jak widać, w zegarze cyfrowym można dodatkowo odczytać sekundy. Zegar analogowy w Androidzie posiada dwie wskazówki, jedna wskazuje godziny, a druga — minuty. Aby umieścić te zegary w układzie graficznym, możemy wykorzystać węzły XML widoczne na listingu 6.24.

### **Listing 6.24.** Dodawanie obiektów DigitalClock i AnalogClock w języku XML

---

```
<DigitalClock
    android:layout_width="wrap_content" android:layout_height="wrap_content" />

<AnalogClock
    android:layout_width="wrap_content" android:layout_height="wrap_content" />
```

---

Kontrolki te pozwalają jedynie na wyświetlanie bieżącego czasu, nie zapewniają jednak możliwości modyfikowania daty ani czasu. Są to zatem zwykłe zegary, których jedną funkcją jest wyświetlanie aktualnej godziny. Zatem w przypadku potrzeby zmiany daty lub czasu należy stosować kontrolki DatePicker i TimePicker lub DatePickerDialog i TimePickerDialog. Przydatnym szczegółem jest, że obydwie kontrolki — DigitalClock oraz AnalogClock — będą automatycznie aktualizować czas, bez konieczności ustawiania czegokolwiek. Oznacza to, że w przypadku zegara cyfrowego sekundy będą same odliczane, a w zegarze analogowym wskazówki będą się poruszały samoistnie, bez potrzeby zapewniania dodatkowej obsługi.

## Kontrolka MapView

Kontrolka com.google.android.maps.MapView umożliwia wyświetlanie mapy. Można utworzyć egzemplarz tej kontrolki w pliku XML układu graficznego lub w kodzie Java, jednak wykorzystująca ją aktywność musi rozszerzyć klasę MapActivity. Klasa ta obsługuje przetwarzanie wielowątkowych żądań ładowania mapy, przeprowadzanie procesu buforowania i tak dalej.

Na listingu 6.25 został zaprezentowany przykład utworzenia obiektu MapView.

**Listing 6.25.** Utworzenie kontrolki MapView w pliku XML układu graficznego

---

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <com.google.android.maps.MapView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="myAPIKey"
    />

</LinearLayout>
```

---

Szczegółowe informacje na temat kontrolki MapView zostały zawarte w rozdziale 17., w którym opiszemy usługi oparte na wyznaczaniu położenia geograficznego. Znajdują się tam również informacje, w jaki sposób uzyskać własny klucz API mapowania.

## Działanie adapterów

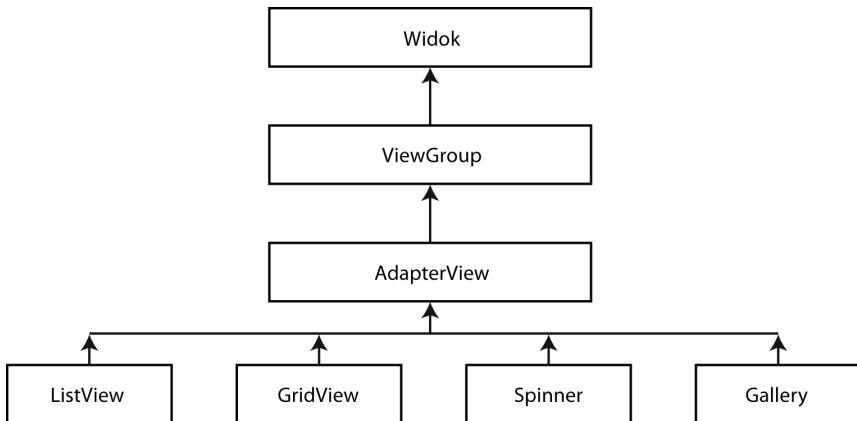
Zanim zajmiemy się kontrolkami listy w Androidzie, musimy przedstawić pojęcie adaptera. Kontrolki listy służą do wyświetlania zbiorów danych. Jednak zamiast używać jednego typu kontrolki zarówno do obsługi wyświetlania, jak i zarządzania danymi, Android dzieli te dwa zadania pomiędzy kontrolki listy i adaptery. Kontrolki listy są rozszerzeniem klasy android.widget.AdapterView i dzielą się na następujące kategorie: ListView, GridView, Spinner oraz Gallery (rysunek 6.8).

Sama klasa AdapterView rozszerza klasę android.widget.ViewGroup, co oznacza, że widoki ListView, GridView i inne są kontrolkami-pojemnikami. Innymi słowy, kontrolki listy wyświetlają zbiór widoków potomnych. Zadaniem adaptera jest zarządzanie danymi pojemnika AdapterView oraz dostarczenie mu potomnych widoków. Przyjrzyjmy się, jak to działa, analizując adapter SimpleCursorAdapter.

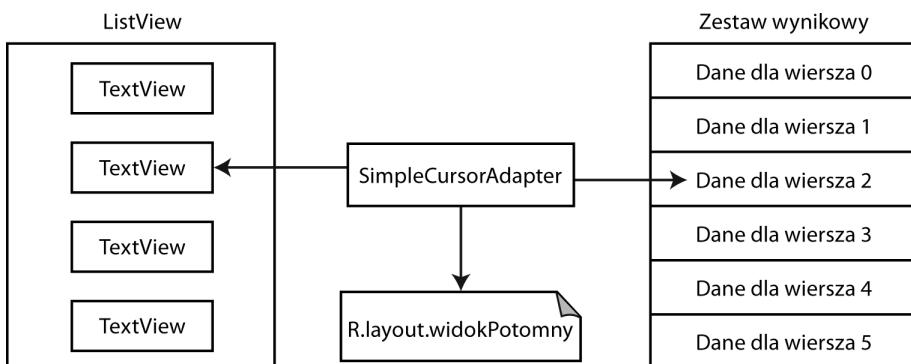
## Zapoznanie się z klasą SimpleCursorAdapter

Adapter SimpleCursorAdapter został naszkicowany na rysunku 6.9.

Zrozumienie tego rysunku odgrywa niebagatelną rolę. Po lewej stronie widzimy klasę AdapterView. W tym przykładzie jest to pojemnik ListView utworzony z potomnych widoków TextView. Po prawej stronie mamy do czynienia z danymi; tu są one reprezentowane przez zestaw wynikowy z wierszami danych, pochodzący z zapytania wysłanego do dostawcy treści.



Rysunek 6.8. Hierarchia klasy AdapterView



Rysunek 6.9. Klasa SimpleCursorAdapter

Aby odwzorować dane w kontrolce `ListView`, klasa `SimpleCursorAdapter` musi posiadać dostęp do identyfikatora potomnego układu graficznego. Ten potomny układ musi opisywać układ graficzny wszystkich elementów danych (widocznych po prawej stronie), które mają zostać wyświetlane po lewej stronie. W tym przypadku układ graficzny nie różni się od układów, które prezentowaliśmy podczas omawiania aktywności, musi on tylko opisywać układ graficzny pojedynczego wiersza z pojemnika `ListView`. Jeśli na przykład posiadamy zestaw wynikowy dostarczony od dostawcy treści `Contacts`, a w pojemniku `ListView` chcemy wyświetlać wyłącznie nazwę danego kontaktu, trzeba określić układ graficzny opisujący wygląd takiego pola zawierającego nazwę kontaktu. Aby w każdym wierszu pojemnika `ListView` wyświetlać nazwę i obraz, które pochodzą z zestawu wynikowego, taki układ graficzny musiałby definiować sposób wyświetlania nazwy oraz obrazu.

Nie oznacza to wcale, że trzeba dostarczać oddzielną specyfikację układu graficznego dla każdego pola w zestawie wynikowym ani że w zestawie wynikowym mają znaleźć się fragmenty danych, za pomocą których trzeba wypełnić wszystkie wiersze pojemnika `ListView`. Jako przykład za moment pokażemy, w jaki sposób można wybierać wiersze za pomocą pól wyboru umieszczonego w widoku `ListView`, gdzie te pola nie muszą być zestawami danych z zestawu wynikowego. Zademonstrujemy także, w jaki sposób uzyskać dostęp do danych z zestawu wynikowego, jeśli te dane nie są częścią pojemnika `ListView`. A chociaż cały czas rozmawiamy o widokach `ListView`,

`TextView`, kurSORach i zestawach wynikowych, należy pamiętać, że koncepcja adapterów posiada ogólniejszy charakter. Wracając do rysunku 6.9, obszar po lewej stronie może być galerią, natomiast prawa strona — prostą tabelą z obrazami. Na razie nie utrudnijmy sobie jednak zadania i przyjrzyjmy się dokładniej klasie `SimpleCursorAdapter`.

Konstruktor klasy `SimpleCursorAdapter` wygląda następująco:

```
SimpleCursorAdapter(Context context, int layout, Cursor c, String[] from, int[] to)
```

Adapter ten przekształca krotkę w kurSORze do widoku podziemnego względem kontrolki pojemnika. Definicja widoku potomnego została umieszczona w zasobie XML (parametr `childLayout`). Zwróćmy uwagę, że krotka w kurSORze może zawierać wiele kolumn. Aby wskazać, które mają zostać zaznaczone w adapterze `SimpleCursorAdapter`, definiuje się tablicę z nazwami kolumn. W tym celu stosuje się parametr `from`.

W podobny sposób, ponieważ każda wybrana kolumna musi zostać odwzorowana w obiekcie klasy `View` układu graficznego, należy utworzyć identyfikatory dla parametru `to`. Pomiędzy wybraną kolumną a kontrolką `View` wyświetlającą dane w kolumnie istnieje odwzorowanie typu jeden do jednego, zatem tablice wartości parametrów `from` i `to` muszą zawierać tę samą liczbę elementów. Jak już wcześniej wspomnieliśmy, widok potomny może być innym typem widoku; to wcale nie musi być kontrolka `TextView`. Można zamiast tego wprowadzić na przykład kontrolkę `ImageView`.

Widok `ListView` i nasz adapter współpracują ze sobą w przemyślany sposób. Kiedy pojemnik `ListView` próbuje wyświetlić wiersz danych, wywołuje metodę `getView()` adaptera i przekazuje położenie wyświetlanego wiersza. Adapter w odpowiedzi tworzy odpowiedni widok potomny za pomocą układu graficznego ustalonego w swoim konstruktorze, uwzględniając dane pobrane z właściwego rekordu pochodzącego z zestawu wynikowego. Zatem widok `ListView` nie musi obsługiwać danych po stronie adaptera. Widok ten wyłącznie wywołuje potrzebne potomne widoki. Jest to punkt krytyczny, gdyż w ten sposób pojemnik `ListView` nie musi tworzyć oddzielnego widoku potomnego dla każdego wiersza danych. Pojemnik `ListView` tworzy tylko tyle widoków potomnych, ile trzeba wyświetlić. Z technicznego punktu widzenia, jeżeli przewidujemy wyświetlanie tylko dziesięciu wierszy, widok `ListView` mógłby utworzyć tylko dziesięć potomnych układów graficznych, nawet jeśli zestaw wynikowy składałby się z setek rekordów. W rzeczywistości system może wywoływać więcej widoków potomnych, ponieważ zazwyczaj Android przechowuje dodatkowe obiekty na wypadek potrzeby szybszego wyświetlenia nowego wiersza. Wyniosek wynika z tego taki, że potomne widoki pojemnika `ListView` mogą ulegać ciągłe przetwarzaniu. Zajmiemy się tym dokładniej w dalszej części książki.

Na rysunku 6.9 można zauważyc pewną elastyczność w stosowaniu adapterów. Ponieważ kontrolka listy korzysta z adaptera, można podstawić różne rodzaje adapterów w zależności od rodzaju danych oraz widoków podziemnych. Jeżeli na przykład klasa `AdapterView` nie będzie zapełniana danymi z dostawcy treści lub bazy danych, nie ma potrzeby, żeby używać adaptera `SimpleCursorAdapter`. Można wtedy zastosować jeszcze prostszy adapter —  `ArrayAdapter`.

## Zapoznanie się z klasą `ArrayAdapter`

Klasa  `ArrayAdapter` jest najprostszym adaptorem dostępnym w Androidzie. Jej grupą docelową są kontrolki listy. Działanie obiektów tej klasy opiera się na założeniu, że kontrolki `TextView` reprezentują elementy listy (na przykład widoki potomne). Utworzenie adaptera  `ArrayAdapter` może być bardzo proste:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(
    this, android.R.layout.simple_list_item_1,
    new String[]{"Dave", "Satya", "Dylan"});
```

W dalszym ciągu przekazujemy kontekst (np. `this`) oraz identyfikator zasobu potomnego układu graficznego. Zamiast jednak przekazywać tablicę `from` specyfikacji pola danych, jako rzeczywiste dane przekazujemy tablicę ciągów znaków. Nie przekazujemy kursora ani tablicy identyfikatorów zasobów obiektu `View`. Zakładamy tutaj, że potomny układ graficzny składa się z pojedynczej kontrolki `TextView` oraz że będzie on wykorzystywany przez klasę `ArrayAdapter` jako miejsce docelowe dla ciągów znaków przechowywanych w tablicy danych.

Zaproponujemy teraz przyjemny skrót dla identyfikatora zasobu `childLayout`. Zamiast tworzyć własny plik układu graficznego do obsługi obiektów listy, możemy skorzystać z predefiniowanych układów graficznych Androida. Zauważmy, że przedrostkiem w identyfikatorze potomnego układu graficznego jest `android..` Zamiast przeszukiwać lokalny katalog `/res`, Android przeszukuje swój własny. Można przejrzeć ten folder poprzez otwarcie katalogu zawierającego zestaw Android SDK i wybranie `platforms/<wersja-androida>/data/res/layout`. Znajdziemy tu element `simple_list_item_1.xml`, w którym widać definicję prostej kontrolki `TextView`. To właśnie tę kontrolkę wykorzystuje klasa `ArrayAdapter` do utworzenia widoku (w metodzie `getView()`), który zostanie przekazany pojemnikowi `ListView`. Warto przejrzeć zawarte tu katalogi, żeby znaleźć predefiniowane układy graficzne dla wszelakich rodzajów zastosowań. W dalszej części książki wykorzystamy jeszcze niektóre z nich.

Klasa  `ArrayAdapter` posiada również inne konstruktory. Jeżeli potomny układ graficzny nie jest prostym widokiem `TextView`, można przekazać identyfikator układu graficznego wiersza oraz identyfikator kontrolki `TextView` otrzymującej dane. Jeśli nie mamy przygotowanej do przekazania tablicy ciągów znaków, możemy zastosować metodę `createFromResource()`. Listing 6.26 stanowi przykład, w którym tworzymy klasę `Adapter` dla obiektu typu `Spinner`:

**Listing 6.26.** Utworzenie adaptera  `ArrayAdapter` z pliku zasobów typu `string`

```
<Spinner android:id="@+id/spinner"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />

Spinner spinner = (Spinner) findViewById(R.id.spinner);

ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
    R.array.planets, android.R.layout.simple_spinner_item);

adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);

spinner.setAdapter(adapter);

<?xml version="1.0" encoding="utf-8"?>
<!-- Plik ten znajduje się w /res/values/planets.xml -->
<resources>
    <string-array name="planets">
        <item>Merkury</item>
        <item>Wenus</item>
        <item>Ziemia</item>
        <item>Mars</item>
        <item>Jowisz</item>
```

```
<item>Saturn</item>
<item>Uran</item>
<item>Neptun</item>
</string-array>
</resources>
```

Listing 6.26 składa się z trzech części. Pierwsza stanowi układ graficzny obiektu `Spinner` zapisany w pliku XML. Druga część, napisana w języku Java, ukazuje nam, w jaki sposób można utworzyć klasę  `ArrayAdapter`, której źródło danych zostało zdefiniowane w pliku zasobów typu `String`. Za pomocą tej metody możemy nie tylko uzewnętrznić zawartość listy w pliku XML, lecz również korzystać ze zlokalizowanych wersji list. Obiektami typu `Spinner` zajmiemy się nieco później, na razie wystarczy nam wiedzieć, że obiekt tego typu posiada widok pozwalający na wyświetlenie aktualnie wybranej wartości oraz widok listy elementów, które można wybrać. W zasadzie mamy tu do czynienia z listą rozwijalną. Trzecią część listingu 6.26 stanowi plik zasobów umieszczony w katalogu `/res/values/planets.xml`, który jest wczytywany w celu uruchomienia klasy  `ArrayAdapter`.

Warto wspomnieć, że klasa  `ArrayAdapter` pozwala na dynamiczne modyfikowanie wykorzystywanych danych. Na przykład metoda `add()` dodaje nową wartość na końcu tablicy. Metoda `insert()` wprowadza nową wartość w określonej pozycji tablicy, natomiast metoda `remove()` usuwa obiekt z tablicy. Możemy także wywołać metodę `sort()`, która uporządkuje tablicę. Oczywiście, po wykonaniu tych wszystkich czynności tablica danych zostaje zdesynchronizowana z pojemnikiem  `ListView`, zatem należy wtedy wywołać metodę  `notifyDataSetChanged()` adaptera. W ten sposób zsynchronizujemy ponownie kontrolkę  `ListView` z adapterem.

Poniższa lista podsumowuje rodzaje adapterów dostępnych w systemie Android:

- `ArrayAdapter<T>`. Adapter ten znajduje się na szczytce ogólnej tabeli własnych obiektów. Jest przeznaczony do stosowania z kontrolkami  `ListView`.
- `CursorAdapter`. Adapter ten, również używany przy kontrolkach  `ListView`, dostarcza dane listy poprzez kurSOR.
- `SimpleAdapter`. Jak sama nazwa sugeruje, mamy do czynienia z prostym adapterem. Zazwyczaj używany jest do zapełniania listy danymi statycznymi (również z zasobów).
- `ResourceCursorAdapter`. Ten adapter rozszerza klasę  `CursorAdapter` i tworzy widoki z zasobów.
- `SimpleCursorAdapter`. Adapter ten rozszerza klasę  `ResourceCursorAdapter` i tworzy widoki  `TextView`/ `ImageView` z kolumn w kurSORZE. Widoki są zdefiniowane w zasobach.

Przedstawiliśmy wystarczająco dobrze zagadnienie adapterów, aby zaprezentować rzeczywiście przykłady korzystania z nich oraz z kontrolkami listy (znanych także pod nazwą  `AdapterView`). Do dzieła.

## Wykorzystywanie adapterów wraz z kontrolkami `AdapterView`

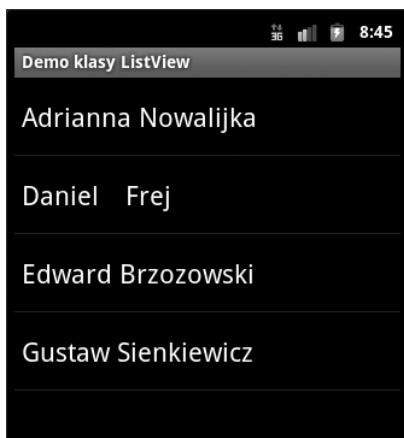
Po zapoznaniu się z tematyką adapterów czas zaprzerć je do pracy i dostarczyć im dane przesypane do kontrolek listy. W tym podrozdziale rozpoczniemy od omówienia pierwszej kontrolki tego typu —  `ListView`. Następnie przyjrzymy się mechanizmowi tworzenia własnego adaptera, a na końcu opiszemy inne rodzaje kontrolek listy:  `GridView`, obiekty typu  `Spinner` i galerie.

## Podstawowa kontrolka listy — ListView

Kontrolka ListView wyświetla pionowo listę elementów. Inaczej mówiąc, jeśli posiadamy listę przeglądanych elementów i przekracza ona rozmiary ekranu, możemy ją przewinąć, aby zobaczyć pozostałe elementy. Przeważnie stosuje się tę kontrolkę poprzez napisanie nowej aktywności, rozszerzającej klasę android.app.ListActivity. Klasa ListActivity zawiera kontrolkę ListView, a dane są w niej umieszczane poprzez wywołanie metody setListAdapter(). Jak już wcześniej omówiono, adaptery łączą kontrolki listy z danymi i biorą udział w przygotowaniu widoków potomnych dla tych kontrolek. Elementy w pojemniku ListView można kliknąć, co spowoduje natychmiastową odpowiedź, można też je zaznaczać, dzięki czemu można później pracować na zbiorze wybranych elementów. Rozpoczniemy od najprostszych czynności i stopniowo będziemy dodawać nowe funkcje.

### Wyświetlanie wartości w kontrolce ListView

Rysunek 6.10 przedstawia kontrolkę ListView w jej najprostszej postaci.



**Rysunek 6.10.** Zastosowanie kontrolki ListView

W kolejnym ćwiczeniu wypełnimy cały ekran kontrolką ListView, więc nie trzeba jej nawet określić w pliku układu graficznego *main.xml*. Na listingu 6.27 umieszczono kod Java kontrolki ListActivity.

#### Listing 6.27. Dodawanie elementów do kontrolki ListView

```
public class ListDemoActivity extends ListActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        Cursor c = managedQuery(People.CONTENT_URI,
                               null, null, null, People.NAME);

        String[] cols = new String[] {People.NAME};
```

```
    int[] views = new int[] {android.R.id.text1};

    SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
        android.R.layout.simple_list_item_1,
        c, cols, views);
    this.setAdapter(adapter);
}
}
```

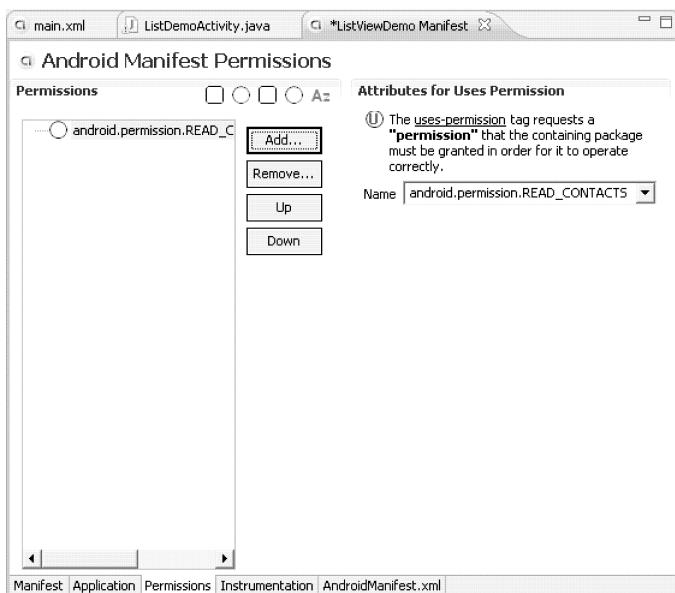
Kod z listingu 6.27 powoduje utworzenie kontrolki `ListView`, zapełnionej pobraną z urządzenia listą kontaktów. W tym przykładzie żądamy od urządzenia listy kontaktów. W celach demonstracyjnych zaznaczamy wszystkie pola pojemnika `Contacts` (na przykład za pomocą pierwszego parametru `null` w metodzie `managedQuery()`) i stosujemy sortowanie według wartości pola `People.NAME` (do czego służy ostatni parametr we wspomnianej metodzie `managedQuery()`). Następnie tworzymy projekcję (kolumn), dzięki której wybieramy wyłącznie nazwy kontaktów dla pojemnika `ListView` — projekcja definiuje interesujące nas kolumny. Kolejnym etapem jest utworzenie tablicy identyfikatorów zasobów (widoków), która pozwalałyby na odzwierciedlanie nazwy kolumny (`People.NAME`) wobec kontrolki `TextView` (`android.R.id.text1`). W kolejnym etapie tworzymy adaptera kurSORA i konfigurujemy adapter listy. Klasa adaptera jest przy stosowana do przeglądania krotek w danych źródłowych i pobierania nazw kontaktów w sposób umożliwiający zapełnienie interfejsu użytkownika.

Musimy wykonać jeszcze jedną czynność, zanim aplikacja zadziała. Ponieważ w tym ćwiczeniu aplikacja uzyskuje dostęp do listy kontaktów telefonu, należy przydzielić jej odpowiednie uprawnienia. Informacje dotyczące zabezpieczeń przedstawiono w rozdziale 10., teraz więc wyjaśnimy jedynie, w jaki sposób udostępnić dane kontrolce `ListView`. Należy dwukrotnie kliknąć nazwę pliku `AndroidManifest.xml` w projekcie, a następnie wybrać zakładkę `Permissions`. W dalszej kolejności trzeba kliknąć przycisk `Add...`, zaznaczyć opcję `Uses Permission` i na końcu kliknąć `OK`. Należy przewinąć listę `Name` aż do pozycji `android.permission.READ_CONTACTS`. Okno środowiska Eclipse powinno wyglądać tak jak na rysunku 6.11. Można teraz zapisać plik `AndroidManifest.xml` i uruchomić aplikację w emulatorze. Może zaistnieć potrzeba dodania kontaktów za pomocą aplikacji Kontakty, zanim jakiekolwiek nazwiska pojawią się w naszym przykładowym programie.

Zauważmy, że metoda `onCreate()` nie ustanawia widoku treści danej aktywności. Ponieważ bazowa klasa `ListActivity` zawiera już kontrolkę `ListView`, należy jedynie zapewnić jej dostęp do danych. Wykorzystaliśmy w tym przykładzie kilka skrótów; pierwszy polegał na zastosowaniu głównego układu graficznego w pojemniku `ListView`. Wykorzystaliśmy również predefiniowany układ graficzny Androida w potomnym widoku (identyfikator `android.R.layout.simple_list_item_1`), w którym znajduje się predefiniowana kontrolka `TextView` (`android.R.id.text1`). Podsumowując, ta konfiguracja wcale nie jest skomplikowana.

## Elementy reagujące na kliknięcie w pojemniku `ListView`

Oczywiście, po uruchomieniu tej przykładowej aplikacji łatwo zauważyc, że możemy przewijać listę kontaktów w górę i w dół, ale nic poza tym. W jaki sposób można zrobić coś bardziej interesującego z tą aplikacją, na przykład uruchomić aplikację Kontakty po kliknięciu przez użytkownika jednego z elementów pojemnika `ListView`? Na listingu 6.28 znajdziemy modyfikację wcześniejszego kodu, umożliwiającą reagowanie na czynności użytkownika.



Rysunek 6.11. Modyfikowanie pliku AndroidManifest.xml, umożliwiające uruchomienie aplikacji

#### **Listing 6.28.** Przyjmowanie danych wprowadzanych przez użytkownika w pojemniku ListView

```
public class ListViewActivity2 extends ListActivity implements OnItemClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        ListView lv = getListView();

        Cursor c = managedQuery(People.CONTENT_URI,
                               null, null, null, People.NAME);

        String[] cols = new String[]{People.NAME};
        int[] views = new int[] {android.R.id.text1};

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
                android.R.layout.simple_list_item_1,
                c, cols, views);
        this.setAdapter(adapter);
        lv.setOnItemClickListener(this);
    }

    @Override
    public void onItemClick(AdapterView<?> adView, View target, int position, long id) {
        Log.v("ListViewActivity", "w metodzie onItemClick z " + ((TextView)
        ↪target).getText())
    +
        ". Pozycja = " + position + ". Id = " + id);
        Uri selectedPerson = ContentUris.withAppendedId(
            People.CONTENT_URI, id);
    }
}
```

```
        Intent intent = new Intent(Intent.ACTION_VIEW, selectedPerson);
        startActivity(intent);
    }
}
```

W efekcie nasza aktywność implementuje interfejs `onItemClickListener`, co oznacza, że będziemy otrzymywać wywołanie zwrotne za każdym razem, gdy użytkownik kliknie jakiś element pojemnika `ListView`. Jak widać po zapoznaniu się z metodą `onItemClick()`, otrzymujemy wiele informacji na temat klikniętego elementu, w tym takie jak kliknięty widok, położenie klikniętego elementu w pojemniku `ListView` oraz zgodny z adapterem identyfikator tego elementu. Ponieważ wiemy, że pojemnik `ListView` składa się z kontrolek `TextView`, zakładamy, że otrzymaliśmy właśnie taką kontrolkę i że generujemy ją przez wywołanie metody `getText()`, służącej do odczytania nazwy kontaktu. Wartość położenia reprezentuje umiejscowienie elementu na pełnej liście obiektów w pojemniku `ListView` i jest ona liczona od zera. Zatem pierwszy element na liście posiada przypisaną wartość 0.

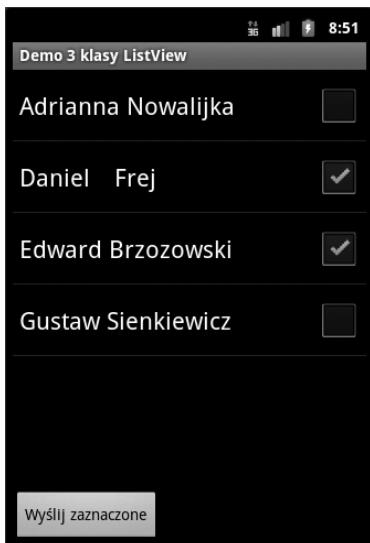
Wartość identyfikatora całkowicie zależy od adaptera oraz źródła danych. W naszym przykładzie wysyłamy zapytania do dostawcy treści `Contacts`, zatem — zgodnie z adapterem — mamy tu do czynienia z identyfikatorem `_ID` rekordu od dostawcy treści. Jednak w innych przypadkach źródło danych może nie pochodzić od dostawcy treści, więc nie należy sądzić, że możemy zawsze tworzyć identyfikator URI, jak w omawianym przykładzie. Jeśli korzystaliśmy z adaptera  `ArrayAdapter` odczytującego wartości z pliku XML zasobów, uzyskany przez nas identyfikator będzie prawdopodobnie stanowił pozycję danej wartości w tablicy danych oraz, w istocie, może być dokładnie wartością tej pozycji.

Podczas omawiania klasy  `ArrayAdapter` wspominaliśmy, że metoda  `notifyDataSetChanged()` służy do synchronizowania adaptera z pojemnikiem  `ListView` w przypadku modyfikowania danych. Przeprowadźmy mały eksperyment na naszym przykładzie. Kliknijmy jeden z elementów listy, co spowoduje wyświetlenie aplikacji `Kontakty`. Edytujmy teraz ten kontakt i zmieńmy jego nazwę. Należy kliknąć przycisk *Gotowe*, a następnie *Cofnij*, aby wrócić do naszej aplikacji. Zauważmy, że nazwa kontaktu w pojemniku  `ListView` została automatycznie zaktualizowana. Świetne, nieprawdaż? Pojemnik  `ListView` został automatycznie zaktualizowany za pomocą klasy  `SimpleCursorAdapter` i dostawcy treści  `Contacts`. Jednak w przypadku klasy  `ArrayAdapter` trzeba samodzielnie przywołać metodę  `notifyDataSetChanged()`.

To nie było wcale takie trudne. Utworzyliśmy własny pojemnik  `ListView` zawierający nazwy kontaktów, a po kliknięciu danego elementu została uruchomiona aplikacja `Kontakty` z informacjami o wybranej osobie. A co w przypadku, gdy chcemy najpierw zaznaczyć kilka nazwisk i w jakiś sposób działać na takiej grupie? W następnej aplikacji zmodyfikujemy układ graficzny listy i dodamy do niej pola wyboru, następnie zaś wprowadzimy do interfejsu użytkownika przycisk pozwalający na przetwarzanie podgrupy zaznaczonych elementów.

## Dodawanie innych kontrolek do pojemnika `ListView`

Jeżeli chcemy wprowadzić dodatkowe kontrolki do głównego układu graficznego, możemy stworzyć własny plik XML układu graficznego, wstawić do niego pojemnik  `ListView` i dodać pożądane kontrolki. Można na przykład wstawić w interfejsie UI przycisk poniżej kontrolki  `ListView`, który pozwala na wysłanie listy zaznaczonych elementów, co zostało pokazane na rysunku 6.12.



**Rysunek 6.12.** Dodatkowy przycisk umożliwiający użytkownikowi wysłanie listy zaznaczonych elementów

Główny układ graficzny naszej aplikacji został umieszczony na listingu 6.29 i zawiera definicję interfejsu aktywności — kontrolek `ListView` oraz `Button`.

#### **Listing 6.29.** Przesłonięcie kontrolki `ListView`, do której odnosi się klasa `ListActivity`

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Ten plik umieszczony jest w podkatalogu /res/layout/list.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">

    <ListView android:id="@+id/list"
        android:layout_width="fill_parent"
        android:layout_height="0dip"
        android:layout_weight="1"/>

    <Button android:id="@+id/btn" android:onClick="doClick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Wyślij zaznaczone" />

</LinearLayout>
```

Zwróćmy uwagę na specyfikację identyfikatora dla pojemnika `ListView`. Musielibyśmy wprowadzić wartość `@+id/list`, ponieważ aktywność `ListActivity` oczekuje znalezienia pojemnika `ListView` w tak nazwanym układzie graficznym. Gdybyśmy polegali na domyślnym pojemniku `ListView` utworzonym przez aktywność `ListActivity`, posiadałby on właśnie taki identyfikator.

Inną kwestią, na którą warto zwrócić uwagę, jest sposób określenia wysokości pojemnika `ListView` w układzie `LinearLayout`. Chcemy, aby przycisk był widoczny na ekranie przez cały czas, bez względu na liczbę elementów dostępnych w widoku `ListView`, i nie chcemy przecież

przewijać ekranu na sam dół, aby znaleźć tam ten przycisk. W tym celu ustawiamy wartość argumentu `layout_height` na 0, a następnie wprowadzamy właściwość `layout_weight`, dzięki której kontrolka może zająć całe dostępne miejsce w nadziednym pojemniku. Dzięki tej sztuczce rezerwujemy miejsce na przycisk oraz pozostawiamy możliwość przewijania pojemnika `ListView`. Więcej informacji o układach graficznych i wagach znajdziemy w dalszej części rozdziału.

Implementacja tej aktywności będzie przypominała kod widoczny na listingu 6.30.

**Listing 6.30.** Odczytywanie danych wprowadzanych przez użytkownika w aktywności `ListViewActivity`

---

```
public class ListViewActivity3 extends ListActivity
{
    private static final String TAG = "ListViewActivity3";
    private ListView lv = null;
    private Cursor cursor = null;
    private int idCol = -1;
    private int nameCol = -1;
    private int notesCol = -1;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list);

        lv = getListView();

        cursor = managedQuery(People.CONTENT_URI,
            null, null, null, People.NAME);

        String[] cols = new String[]{People.NAME};
        idCol = cursor.getColumnIndex(People._ID);
        nameCol = cursor.getColumnIndex(People.NAME);
        notesCol = cursor.getColumnIndex(People.NOTES);

        int[] views = new int[]{android.R.id.text1};

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_multiple_choice,
            cursor, cols, views);

        this.setListAdapter(adapter);

        lv.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
    }

    public void doClick(View view) {
        int count=lv.getCount();
        SparseBooleanArray viewItems = lv.getCheckedItemPositions();
        for(int i=0; i<count; i++) {
            if(viewItems.get(i)) {
                cursor.moveToPosition(i);
                long id = cursor.getLong(idCol);
                String name = cursor.getString(nameCol);
                String notes = cursor.getString(notesCol);
            }
        }
    }
}
```

```
        Log.v(TAG, name + " jest zaznaczony/a. Uwagi: " + notes +
        ". Polozenie = " + i + ". Id = " + id);
    }
}
}
```

Wracamy tutaj do wywoływania metody `setContentView()` w celu ustawienia interfejsu użytkownika w aktywności. Natomiast w konfiguracji adaptera przekazujemy kolejny predefiniowany widok wobec elementu pojemnika `ListView` (`android.R.layout.simple_list_item_multiple_choice`), w wyniku czego każdy wiersz zawiera kontrolki `TextView` i `CheckBox`. Jeżeli zajrzymy do pliku zawierającego układ graficzny, zauważymy kolejną podklasę kontrolki `TextView`, noszącą nazwę `CheckedTextView`. Ten specjalny rodzaj kontrolki `TextView` został stworzony z myślą o pojemnikach `ListView`. Mówiliśmy przecież, że w tym folderze, zawierającym predefiniowane pliki układów graficznych, można znaleźć interesujące rzeczy! Warto zauważać, że identyfikator tej kontrolki posiada wartość `text1`, którą musieliśmy przekazać w tablicy widoków konstruktorowi klasy `SimpleCursorAdapter`.

Ponieważ chcemy, aby użytkownik mógł zaznaczać poszczególne wiersze, wprowadzamy tryb wybierania `CHOICE_MODE_MULTIPLE`. Domyślną wartością tego trybu jest `CHOICE_MODE_NONE`. Ostatnią możliwością, jaką możemy wybrać, jest `CHOICE_MODE_SINGLE`. Aby użyć tego ostatniego trybu, trzeba by wprowadzić inny układ graficzny, najprawdopodobniej `android.R.layout.simple_list_item_single_choice`.

W tym przykładzie zaimplementowaliśmy podstawowy przycisk, wywołujący metodę `onClick()` naszej aktywności. Aby nie utrudniać sprawy, nazwy elementów zaznaczanych przez użytkownika będą zapisywane w oknie *LogCat*. Dobrą wieścią jest, że wprowadzenie takiego rozwiązania jest bardzo proste, z drugiej strony jednak Android wyewoluował do tego stopnia, że jego implementacja może zależeć od wersji systemu. Ukażane tu rozwiązanie polegające na wykorzystaniu pojemnika `ListView` działa od wersji 1 Androida (chociaż przy wywoływaniu zwrótnym przycisku korzystamy ze skrótu dostępnego od wersji 1.6 Androida). Oznacza to, że metoda `getCheckedItemPositions()` jest stara, ale ciągle skuteczna. W wyniku jej działania otrzymujemy tablicę określającą, czy dany element został zaznaczony, czy nie. Zatem teraz można sprawdzić wszystkie elementy za pomocą metody `array.viewItems.get(i)`. Metoda ta przekaże wartość `true`, jeśli dany wiersz w pojemniku `ListView` został zaznaczony. Dostęp do danych można uzyskać za pomocą kursora. Zatem zamiast wyszukiwać dane w pojemniku `ListView`, sprawdzamy informacje zawarte w kursorze. Widok `ListView` powie nam, w którym miejscu adaptera należy szukać.

Po uzyskaniu numeru pozycji zaznaczonego elementu możemy użyć metody `moveToPosition()` kursora, aby przygotować aplikację do odczytu danych. Istnieje inna metoda, spełniająca niemal identyczne zadanie — `getItemAtPosition()` klasy `ListView`. W naszym przypadku element przekazany przez tę metodę przekształciłby się na obiekt `CursorWrapper`. Jak już wcześniej stwierdziliśmy, w innych przypadkach możemy otrzymać odmienne typy obiektów. Obiekt `CursorWrapper` pojawia się tylko dlatego, że pracujemy z dostawcą treści. Należy rozumieć źródło danych oraz adapter, żeby wiedzieć, czego się spodziewać.

Możemy następnie wykorzystać obiekt `Cursor` (lub `CursorWrapper`, jeśli go otrzymaliśmy) do odczytania informacji powiązanych z zaznaczonym wierszem pojemnika `ListView`. Za- uważmy, że w naszym przykładzie odczytujemy nie tylko nazwę kontaktu, ale także uwagi na jego temat, chociaż nigdzie ich nie odzwierciedlaliśmy w tym kontenerze. Jest tak, ponieważ

podczas konfigurowania kurSORA dla adaptera wybraliśmy wszystkie dostępne pola. W praktyce nie trzeba wybierać wszystkich pól, powinniśmy ograniczać zapytania tylko do potrzebnych elementów. Ale w tym konkretnym przypadku wysłaliśmy zapytania dotyczące większej liczby pól, niż było potrzebne do wyświetlania w pojemniku ListView. W łatwy więc sposób uzyskalismy dostęp do tych pól w trakcie wywoływania zwrotnego przycisku.

## Alternatywna metoda odczytywania zaznaczonych elementów w pojemniku ListView

W wersji 1.6 AndroIDA wprowadzono inną metodę odczytywania listy zaznaczonych elementów w pojemniku ListView; mowa tu o metodzie `getCheckItemIds()`. Z kolei w wersji 2.2 została ona wycofana i zastąpiona metodą `getCheckedItemIds()`. Nastąpiła nieznaczna zmiana nazwy, ale sposób korzystania z tej klasy jest zasadniczo taki sam. Poza tym w tej wersji zmodyfikowano sposób obsługiwanego kontaktów. W kolejnym przykładzie wykorzystamy system Android 2.2, aby pokazać działanie tej metody. Listing 6.31 prezentuje odpowiedni kod Java, natomiast plik XML układu graficznego `list.xml` może być taki sam jak na listingu 6.29.

**Listing 6.31.** Alternatywny sposób odczytywania danych wprowadzanych przez użytkownika w klasie ListActivity

---

```
public class ListViewActivity4 extends ListActivity
{
    private static final String TAG = "ListViewActivity4";
    private static final Uri CONTACTS_URI = ContactsContract.Contacts.CONTENT_URI;
    private SimpleCursorAdapter adapter = null;
    private ListView lv = null;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list);

        lv = getListView();

        String[] projection = new String[] {ContactsContract.Contacts._ID,
            ContactsContract.Contacts.DISPLAY_NAME};
        Cursor c = managedQuery(CONTACTS_URI,
            projection, null, null, ContactsContract.Contacts.DISPLAY_NAME);

        String[] cols = new String[] {ContactsContract.Contacts.DISPLAY_NAME};
        int[] views = new int[] {android.R.id.text1};

        adapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_multiple_choice,
            c, cols, views);

        this.setListAdapter(adapter);

        lv.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE);
    }

    public void doClick(View view) {
        if(!adapter.hasStableIds()) {
```

```

        Log.v(TAG, "Dane sa niestabilne");
        return;
    }
    long[] viewItems = lv.getCheckedItemIds();
    for(int i=0; i<viewItems.length; i++) {
        Uri selectedPerson = ContentUris.withAppendedId(
            CONTACTS_URI, viewItems[i]);
        Log.v(TAG, selectedPerson.toString() + " jest zaznaczony/a.");
    }
}

```

W tej przykładowej aplikacji po kliknięciu przycisku następuje wywołanie zwrotnej metody `getCheckedItemIds()`. Tym razem uzyskujemy tablicę identyfikatorów rekordów (z adaptera), które zostały zaznaczone w widoku `ListView`, podczas gdy w poprzedniej aplikacji otrzymaliśmy tablicę pozycji zaznaczonych elementów pojemnika `ListView`. Teraz można pominąć widok `ListView` oraz kursor, ponieważ identyfikatory w połączeniu z dostawcami treści pozwalają na podjęcie dowolnej pożądanej akcji. W tym przykładzie konstruujemy po prostu identyfikator URI, reprezentujący określony rekord z dostawcy treści `Contacts`, i zapisujemy ten identyfikator w dzienniku `LogCat`. Moglibyśmy bezpośrednio operować na danych za pomocą dostawcy treści. Mechanizm ten działa również dobrze w przypadku starszych wersji dostawcy treści `Contacts` oraz wprowadzonej w wersji 1.6 Androida metody `getCheckItemIds()`.

Kolejną różnicą jest zaznaczenie tylko kilku pól na etapie tworzenia obiektu `Cursor`. Jest to całkowicie naturalne rozwiązanie, ponieważ nie trzeba odczytywać większej ilości danych, niż jest to konieczne. Ostatnią rzeczą, na którą warto zwrócić uwagę, jest fakt, że metoda `getCheckedItemIds()` wymaga stabilności danych przechowywanych w adapterze. Zatem bardzo zalecamy wywołanie metody `hasStableIds()` w adapterze, zanim wywołamy metodę `getCheckedItemIds()` w pojemniku `ListView`. W omawianym przykładzie skorzystaliśmy ze skrótu — dany fakt został zwyczajnie odnotowany w dzienniku. Aplikacja użytkowa powinna jakoś na ten fakt zareagować, na przykład uruchomić wątek przebiegający w tle, służący do wykonywania powtórzeń oraz wyświetlający okno dialogowe informujące użytkownika o trwaniu procesu przetwarzania.

Powyżej zademonstrowaliśmy różne scenariusze korzystania z kontrolki `ListView`. Pokazaliśmy, jak wiele pracy wykonują adaptery podczas obsługi pojemników `ListView`. Teraz zajmiemy się pozostałymi rodzajami kontrolek listy, począwszy od widoku `GridView`.

## Kontrolka `GridView`

Większość narzędzi do tworzenia widżetów ma przynajmniej jedną kontrolkę definiującą siatkę. Android posiada kontrolkę `GridView`, dzięki której dane są wyświetlane w takiej siatce. Pamiętajmy, że poprzez „dane” rozumiemy tu tekst, rysunki i tak dalej.

Kontrolka `GridControl` wyświetla informacje w siatce. Algorytm wykorzystania tej kontrolki polega na zdefiniowaniu siatki w pliku XML układu graficznego (listing 6.32), a następnie połączaniu danych z tą siatką za pomocą klasy `android.widget.ListAdapter`. Należy też dodać etykietę `Uses Permission` do pliku `AndroidManifest.xml`, w przeciwnym wypadku przykładowy kod nie zadziała.

**Listing 6.32.** Definiowanie kontrolki GridView w pliku XML układu graficznego oraz w kodzie Java

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Plik ten jest umieszczony w podkatalogu /res/layout/gridview.xml -->
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/dataGrid"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10px"
    android:verticalSpacing="10px"
    android:horizontalSpacing="10px"
    android:numColumns="auto_fit"
    android:columnWidth="100px"
    android:stretchMode="columnWidth"
    android:gravity="center"
    />

public class GridViewActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gridview);
        GridView gv = (GridView)this.findViewById(R.id.gridView);

        Cursor c = managedQuery(People.CONTENT_URI,
            null, null, null, People.NAME);

        String[] cols = new String[] {People.NAME};
        int[] views = new int[] {android.R.id.text1};

        SimpleCursorAdapter adapter = new SimpleCursorAdapter(this,
            android.R.layout.simple_list_item_1 ,c,cols,views);

        gv.setAdapter(adapter);
    }
}
```

---

Na listingu 6.32 zdefiniowano prostą kontrolkę GridView w pliku XML układu graficznego. Siatka ta zostaje wczytana do widoku treści aktywności. Wygenerowany interfejs użytkownika można ujrzeć na rysunku 6.13.

Siatka przedstawiona na rysunku 6.13 wyświetla nazwy kontaktów przechowywanych w urządzeniu. Postanowiliśmy umieścić kontrolki TextView z tymi nazwami, jednak również dobrze można w ich miejsce wstawić obrazy lub inne kontrolki. Ponownie skorzystaliśmy z możliwości predefiniowanych układów graficznych. W rzeczywistości przykład ten jest bardzo podobny do kodu zamieszczonego na listingu 6.27, istnieje jednak pomiędzy nimi kilka istotnych różnic. Po pierwsze, klasa GridViewActivity rozszerza klasę Activity, nie ListActivity. Po drugie, musimy wywołać metodę `setContentView()` w celu ustanowienia układu graficznego dla pojemnika GridView — nie ma tu żadnych domyślnych widoków. Na koniec warto zauważyć, że w celu ustawnienia adaptera wywołujemy metodę `setAdapter()` na obiekcie GridView, a nie metodę `setListAdapter()` wobec klasy Activity.



Rysunek 6.13. Kontrolka GridView wypełniona nazwami kontaktów

Niewątpliwie zdążyliśmy zauważyć, że adapterem siatki jest `ListAdapter`. Listy są zazwyczaj jednowymiarowe, podczas gdy siatki posiadają dwa wymiary. Wynika z tego wniosek, że siatka tak naprawdę wyświetla dane w postaci listy. Okazuje się także, że lista jest wyświetlana wierszami. Inaczej mówiąc, lista zostaje układana najpierw w jednym rzędzie, następnie w drugim i tak dalej.

Podobnie jak przedtem, mamy tu do czynienia z kontrolką listy, która współpracuje z adapterem zarządzającym danymi oraz z generowaniem widoków potomnych. Techniki stosowane wcześniej powinny również działać z kontrolkami `GridView`. Jedyna różnica polega na sposobie wybierania. W przeciwieństwie do kodu widocznego na listingu 6.30, nie ma możliwości wprowadzenia funkcji wielokrotnego wyboru.

## Kontrolka Spinner

Kontrolka `Spinner` pełni funkcję rozwijanego menu. Zazwyczaj stosuje się ją do wybierania opcji ze stosunkowo krótkiej listy. Jeżeli lista jest zbyt długa do wyświetlania, zostaje automatycznie dodany pasek przewijania. Można ją utworzyć w pliku XML układu graficznego w tak prosty sposób:

```
<Spinner
    android:id="@+id/spinner" android:prompt="@string/spinnerprompt"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
/>
```

Chociaż z technicznego punktu widzenia obiekt typu `Spinner` jest kontrolką listy, przypomina on bardziej prosty widok `TextView`. Innymi słowy, kiedy kontrolka `Spinner` pozostaje nieaktywna, wyświetlana jest tylko jedna wartość. Zadaniem tej kontrolki jest umożliwienie użytkownikowi wyboru z zestawu predefiniowanych wartości: po kliknięciu niewielkiej strzałki lista wyświetla się i użytkownik może wybrać z niej jakiś element. Lista ta jest zapełniana tak samo jak w przypadku pozostałych kontrolek listy, to znaczy za pomocą adaptera. Ponieważ kontrolka typu `Spinner` często jest stosowana w formie rozwijanego menu, powszechnym rozwiązaniem jest pobieranie przez adapter listy opcji z pliku zasobów. Przykładowy sposób wykorzystania kontrolki `Spinner` wraz z plikiem zasobów został pokazany na listingu 6.33. Zwróćmy uwagę na nowy atrybut `android:prompt`, ustanawiający zachętą na szczycie listy opcji. Właściwy tekst zachęty znajduje się w pliku `/res/values/strings.xml`. Jak można się spodziewać, klasa `Spinner` posiada również odpowiednią metodę, pozwalającą na umieszczenie zachęty w kodzie.

**Listing 6.33.** Kod tworzący obiekt klasy Spinner z pliku zasobów

```
public class SpinnerActivity extends Activity {
    /** Wywołane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.spinner);

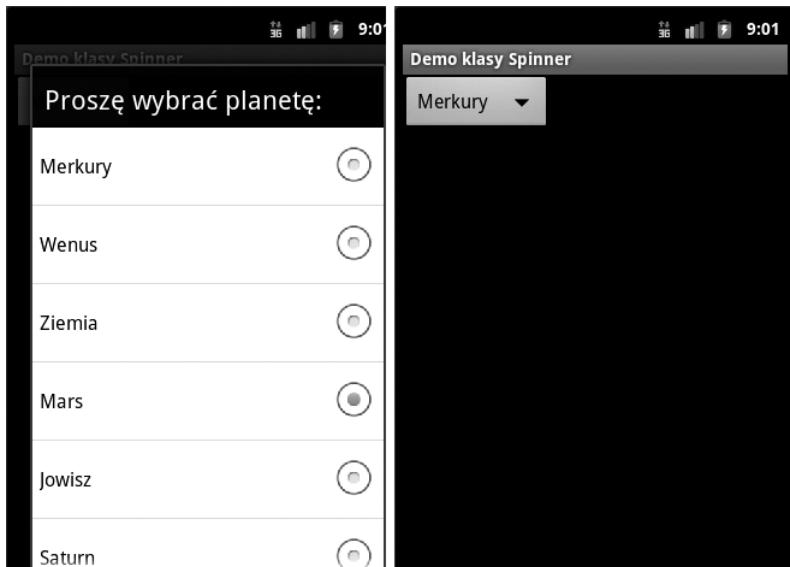
        Spinner spinner = (Spinner)findViewById(R.id.spinner);

        ArrayAdapter<CharSequence> adapter = ArrayAdapter.createFromResource(this,
            R.array.planets, android.R.layout.simple_spinner_item);

        adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);

        spinner.setAdapter(adapter);
    }
}
```

Być może Czytelnik pamięta, że plik *planets.xml* widniał również na listingu 6.26. W omawianym przykładzie ukazujemy sposób utworzenia kontrolki Spinner. Po skonfigurowaniu adaptera zostaje on dołączony do tego obiektu. Na rysunku 6.14 widzimy obiekt Spinner w akcji.



**Rysunek 6.14.** Obiekt Spinner umożliwiający wybór planety

Jedną z cech odróżniających ten obiekt od pozostałych kontrollek list jest obecność dodatkowego układu graficznego, z którego należy korzystać podczas pracy z klasą Spinner. Na lewym zrzucie ekranu z rysunku 6.14 widzimy normalny tryb działania kontrolki Spinner — widoczny jest tu bieżący wybór. W tym przypadku wybrano planetę Saturn. Obok znajduje się strzałka informująca, że mamy do czynienia z kontrolką typu Spinner, a jej naciśnięcie spowoduje wyświetlenie się listy dostępnych wartości. Pierwszy układ graficzny, dostarczany w postaci para-

metru metodzie `ArrayAdapter.createFromResource()`, definiuje wygląd obiektu `Spinner` w trybie normalnym. Na prawym rzucie ekranu z rysunku 6.14 widzimy ten obiekt w trybie listy rozwijanej, oczekujący na wybór nowej wartości przez użytkownika. Układ graficzny tej listy jest ustawiany za pomocą metody `setDropDownViewResource()`. Jest to kolejny przykład sytuacji, gdzie wykorzystujemy dwa predefiniowane układy graficzne, zatem jeżeli chcemy zapoznać się z ich definicjami, powinniśmy odwiedzić katalog `/res/layout` Androida. Oczywiście, możemy również utworzyć własne definicje układów graficznych, aby osiągnąć z góry zamierzony, niestandardowy efekt.

## Kontrolka Gallery

Kontrolka `Gallery` tworzy listę przewijaną w poziomie, która eksponuje elementy widoczne w środkowej części tej listy. Kontrolka ta przeważnie jest wykorzystywana do tworzenia galerii obrazów, gdzie nawigacja między obrazami odbywa się w trybie dotykowym. Można ją utworzyć w pliku XML układu graficznego lub w kodzie Java:

```
<Gallery  
    android:id="@+id/gallery"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
/>
```

Kontrolka typu `Gallery` jest najczęściej używana do wyświetlania obrazów, zatem adapter zostanie najprawdopodobniej dostosowany do ich obsługi. W następnym punkcie, dotyczącym niestandardowych adapterów, zaprezentujemy adapter wyspecjalizowany do obsługi obrazów. Wygląd kontrolki `Gallery` został zaprezentowany na rysunku 6.15.



Rysunek 6.15. Galeria zdjęć krów morskich

## Tworzenie niestandardowych adapterów

Standardowe adaptery systemu Android są łatwe w użyciu, posiadają jednak pewne ograniczenia. Problem ten został rozwiązany za pomocą abstrakcyjnej klasy `BaseAdapter`, którą można rozszerzyć w przypadku konieczności utworzenia niestandardowego adaptera. Adaptery takie okazują się przydatne w przypadku potrzeby wdrożenia niestandardowych sposobów zarządzania danymi lub w celu zapewnienia większej kontroli nad wyświetlaniem potomnych widoków. Stosowanie niestandardowych adapterów pozwala również na zwiększenie wydajności, gdyż możemy stosować techniki buforowania w pamięci podręcznej. Pokażemy teraz, w jaki sposób można utworzyć taki niestandardowy adapter.

Na listingu 6.34 widzimy przykładowy plik XML oraz kod Java tworzące niestandardowy adapter. W omawianym przykładzie adapter posłuży do obsługi zdjęć krów morskich, zatem nawiązemy go `ManateeAdapter`. Utworzymy go również we wnętrzu aktywności.

**Listing 6.34.** Nasz niestandardowy adapter: `ManateeAdapter`

---

```
<?xml version="1.0" encoding="utf-8"?>
<!- Plik ten znajduje się w /res/layout/gridviewcustom.xml -->
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dip"
    android:verticalSpacing="10dip"
    android:horizontalSpacing="10dip"
    android:numColumns="auto_fit"
    android:gravity="center"
/>

public class GridViewCustomAdapter extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.gridviewcustom);

        GridView gv = (GridView) findViewById(R.id.gridView);
        ManateeAdapter adapter = new ManateeAdapter(this);

        gv.setAdapter(adapter);
    }

    public static class ManateeAdapter extends BaseAdapter {
        private static final String TAG = "ManateeAdapter";
        private static int convertViewCounter = 0;
        private Context mContext;
        private LayoutInflater mInflater;

        static class ViewHolder {
            ImageView image;
        }
    }
}
```

```
private int[] manatees = {
    R.drawable.manatee00, R.drawable.manatee01, R.drawable.manatee02,
    R.drawable.manatee03, R.drawable.manatee04, R.drawable.manatee05,
    R.drawable.manatee06, R.drawable.manatee07, R.drawable.manatee08,
    R.drawable.manatee09, R.drawable.manatee10, R.drawable.manatee11,
    R.drawable.manatee12, R.drawable.manatee13, R.drawable.manatee14,
    R.drawable.manatee15, R.drawable.manatee16, R.drawable.manatee17,
    R.drawable.manatee18, R.drawable.manatee19, R.drawable.manatee20,
    R.drawable.manatee21, R.drawable.manatee22, R.drawable.manatee23,
    R.drawable.manatee24, R.drawable.manatee25, R.drawable.manatee26,
    R.drawable.manatee27, R.drawable.manatee28, R.drawable.manatee29,
    R.drawable.manatee30, R.drawable.manatee31, R.drawable.manatee32,
    R.drawable.manatee33 };

private Bitmap[] manateeImages = new Bitmap[manatees.length];
private Bitmap[] manateeThumbs = new Bitmap[manatees.length];

public ManateeAdapter(Context context) {
    Log.v(TAG, "Tworzenie adaptera ManateeAdapter");
    mContext = context;
    mInflater = LayoutInflater.from(context);

    for(int i=0; i<manatees.length; i++) {
        manateeImages[i] = BitmapFactory.decodeResource(
            context.getResources(), manatees[i]);
        manateeThumbs[i] = Bitmap.createScaledBitmap(manateeImages[i],
            100, 100, false);
    }
}

@Override
public int getCount() {
    Log.v(TAG, "w getCount()");
    return manatees.length;
}

public int getViewTypeCount() {
    Log.v(TAG, "w getViewTypeCount()");
    return 1;
}

public int getItemViewType(int position) {
    Log.v(TAG, "w getItemViewType() dla pozycji " + position);
    return 0;
}

@Override
public View getView(int position, View convertView, ViewGroup parent) {
    ViewHolder holder;

    Log.v(TAG, "w getView dla pozycji " + position +
        ", convertView is " +
        ((convertView == null)?"null":"ponowne przetwarzanie"));

    if (convertView == null) {
        convertView = mInflater.inflate(R.layout.gridimage, null);
    }
}
```

```
        convertViewCounter++;
        Log.v(TAG, convertViewCounter + " convertViews zostaly utworzone");

        holder = new ViewHolder();
        holder.image = (ImageView) convertView.findViewById(R.id.gridImageView);

        convertView.setTag(holder);
    } else {
        holder = (ViewHolder) convertView.getTag();
    }

    holder.image.setImageBitmap( manateeThumbs[position] );

    return convertView;
}

@Override
public Object getItem(int position) {
    Log.v(TAG, "w getItem() dla pozycji " + position);
    return manateeImages[position];
}

@Override
public long getItemId(int position) {
    Log.v(TAG, "w getItemId() dla pozycji " + position);
    return position;
}
}
}
```

Po uruchomieniu aplikacji powinniśmy zobaczyć widok przedstawiony na rysunku 6.16.



Rysunek 6.16. Widok siatki zawierający zdjęcia krów morskich

Chociaż przedstawiony kod wydaje się względnie prosty, wiele elementów wymaga tu objaśnienia. Zaczniemy od klasy `Activity`, bardzo przypominającej te, z którymi pracowaliśmy wcześniej w tym punkcie. Widać też główny układ graficzny z pliku `gridviewcustom.xml`, w którym zawarta jest wyłącznie definicja kontrolki `GridView`. Musimy uzyskać odniesienie do tej kontrolki z wnętrza układu graficznego, zatem definiujemy i ustanawiamy widok `gv`. Tworzymy nasz obiekt `ManateeAdapter`, przekazujemy mu kontekst i ustanawiamy go wobec kontrolki `GridView`. Na razie nie zrobiliśmy niczego odkrywczego, jednak bez wątpienia zauważymy, że na etapie tworzenia nasz niestandardowy adapter nie wykorzystuje nawet części tych parametrów co adaptery predefiniowane. Wynika to głównie z faktu, że mamy całkowitą kontrolę nad tym konkretnym adapterem i wykorzystujemy go tylko w tej jednej aplikacji. Gdybyśmy chcieli utworzyć adapter ogólniejszego przeznaczenia, najprawdopodobniej wprowadzilibyśmy więcej parametrów. Idźmy zatem dalej.

Zadaniem adaptera jest zarządzanie przekazywaniem danych do obiektów typu `View` Androida. Obiekty te są wykorzystywane przez kontrolkę listy (w tym przypadku `GridView`). Dane pochodzą z jakiegoś źródła danych. We wcześniejszych przykładach dane były dostarczane poprzez obiekt kursora, który był przekazywany adapterowi. W omawianym przypadku nasz niestandardowy adapter ma wszystkie informacje o danych oraz ich źródłach. Jest w stanie skonstruować interfejs użytkownika, jeśli otrzyma takie żądanie od kontrolki. Może również przekazać do ponownego wykorzystania widoki, które przestaną być potrzebne. Może wydawać się dziwne, że adapter musi mieć zdolność konstruowania widoków, ale ostatecznie wszystko razem nabiera sensu.

W trakcie tworzenia instancji omawianego niestandardowego adaptera `ManateeAdapter` zwykle przekazuje się mu kontekst, który będzie w nim przechowywany. Jego przechowywanie bardzo często okazuje się przydatne. Drugim zadaniem tego adaptera jest przechowanie klasy `Inflater`. To pozwala na poprawę wydajności w momencie utworzenia nowego widoku, zwracanego kontrolce listy. Trzecim typowym zadaniem adaptera jest utworzenie obiektu `ViewHolder`, przechowującego obiekty typu `View` dla zarządzanych danych. W omawianym przykładzie przechowujemy po prostu widok `ImageView`, ale gdyby trzeba było obsłużyć dodatkowe pola, wprowadzilibyśmy je do definicji obiektu `ViewHolder`. Gdybyśmy na przykład posiadaли pojemnik `ListView`, na którego każdy wiersz składałby się jeden widok `ImageView` i dwie kontrolki `TextView`, obiekt ten przechowywałby dokładnie jeden widok `ImageView` i dwie kontrolki `TextView`.

Ponieważ omawiany adapter służy do obsługi obrazów krów morskich, ustanawiamy tablicę identyfikatorów tych zasobów, za pomocą których zostaną utworzone mapy bitowe. Definiujemy również tablicę map bitowych, które będą stanowiły listę danych.

Jak wynika z kodu konstruktora klasy `ManateeAdapter`, zapisujemy kontekst, tworzymy i przechowujemy klasę `Inflater`, a następnie iterujemy poprzez identyfikatory zasobów obrazów i budujemy tablicę bitmap. Ta ostatnia będzie stanowiła nasze dane.

Jak już się wcześniej dowiedzieliśmy, ustanowienie adaptera spowoduje, że kontrolka `GridView` będzie wywoływała wobec niego metody definiujące wyświetlane w niej dane. Na przykład kontrolka `gv` będzie wywoływać metodę `getCount()` adaptera w celu określenia liczby wyświetlanych obiektów. Będzie także wywoływana metoda `getViewTypeCount()` służąca do określenia, jak wiele różnych typów widoków może być wyświetlanych wewnętrz pojemnika `GridView`. W naszym przykładzie przypisujemy jej wartość 1. Jeżeli jednak chcielibyśmy uwzględnić kontener `ListView` i wprowadzić separatory pomiędzy wiersze z danymi, potrzebne byłoby dwa typy danych, a wtedy metoda `getViewTypeCount()` powinna zwracać wartość 2.

Możemy zaplanować dowolną liczbę różnych typów widoków, byle tylko metoda ta przekazywała odpowiednią wartość. Pokrewną metodą jest `getItemViewType()`. Przed momentem stwierdziliśmy, że adapter może przekazywać większą liczbę typów widoków. Żeby jednak uprościć sprawę, wynikiem działania metody `getItemViewType()` może być wyłącznie liczba całkowita, dzięki czemu może wskazywać, jaki typ widoku dotyczy danego fragmentu danych. Jeżeli zatem otrzymujemy dwa typy widoków, metoda `getItemViewType()` za pomocą wartości 0 i 1 wskazywałaby, który typ jest w danej chwili potrzebny. W przypadku obecności trzech typów danych dostępne byłyby wartości 0, 1 i 2.

Jeżeli omawiany adapter obsługuje separatory w pojemniku `ListView`, muszą być one traktowane jako część danych. Oznacza to, że separator zajmuje pozycję danych. Po wywołaniu metody `getView()` przez kontrolkę listy w celu odczytania właściwego widoku dla tej pozycji metoda ta przekaże separator jako widok w miejsce normalnych danych. Jeśli zaś chodzi o typ danych dla tej właśnie pozycji, to metoda `getItemViewType()` przekaże odpowiednią wartość całkowitą, odpowiadającą temu typowi widoku. W przypadku korzystania z separatorów należy także zaimplementować metodę `isEnabled()`. Przekazywałaby ona wartość `true` dla elementów listy, a `false` dla separatorów, ponieważ ten drugi typ danych nie powinien być zaznaczany ani reagować na kliknięcia.

Najbardziej interesujące w klasie `ManateeAdapter` jest wywołanie metody `getView()`. Po określaniu przez widok `gv` liczby dostępnych obiektów, rozpoczyna on wysyłanie zapytań o dane. Teraz możemy mówić o wielokrotnym wykorzystywaniu widoków. Kontrolka listy może pokazywać tyle elementów potomnych, ile zmieści się na ekranie. Oznacza to, że wywoływanie metody `getView()` dla każdego fragmentu danych dostępnych w adapterze nie ma sensu. Czynność ta nabiera sensu dopiero w przypadku jej wywoływania dla takiej liczby elementów, która zmieści się na ekranie. Kiedy `gv` pobiera widoki z adaptera, określa jednocześnie, jak wiele elementów można wyświetlić na wyświetlaczu o danych rozmiarach. Gdy wyświetlacz zostanie już wypełniony danymi, `gv` zaprzestaje wywoływania metody `getView()`.

Jeżeli przyjrzeć się oknu *LogCat* po uruchomieniu tej przykładowej aplikacji, widać różnorodne wywołania, stwierdzimy jednak również, że metoda `getView()` zaprzestała wywoływania, zanim zażądano wszystkich obrazów. Jeżeli zaczniemy teraz przewijać widok `GridView` w górę i w dół, w dzienniku *LogCat* pojawi się więcej wywołań metody `getView()`. Okaże się także, że po utworzeniu określonej liczby widoków potomnych zostaje ona wywołana wraz z parametrem `convertView` posiadającym wartość inną od `null`. Oznacza to, że Android wykorzystuje ponownie stare widoki potomne — co znacznie poprawia wydajność.

Jeżeli wartość parametru `convertView` będzie niezerowa, oznacza to, że pojemnik `gv` ponownie wykorzystuje dany widok. W ten sposób unikamy nadmiernego obciążania układu graficznego XML, nie trzeba też odnajdywać kontrolki `ImageView`. Poprzez połączenie obiektu `ViewHolder` z uzyskiwanym obiektem `View` proces odświeżania widoku może zostać przeprowadzony o wiele szybciej następnym razem, gdy tylko ten widok będzie ponownie potrzebny. Jedyne, co trzeba zrobić w metodzie `getView()`, to ponownie uzyskać obiekt `ViewHolder` i przydzielić właściwe dane do widoku.

Chcieliśmy w tym przykładzie pokazać, że w widoku nie muszą być koniecznie umieszczone dokładnie te same dane, które są zawarte w źródle tych danych. Metoda `createScaledBitmap()` służy do tworzenia mniejszych wersji obrazów, które będą następnie wyświetlane jako miniaturki. Polega to na tym, że kontrolka listy nie wywołuje metody `getItems()`. Zostaje ona wywołana przez inny kod, który pod wpływem określonych działań użytkownika może w jakiś sposób zmodyfikować dane znajdujące się w kontrolce listy. Znowu widać, jak ważne jest zrozumienie

przeprowadzanych działań w przypadku adapterów. Nie zawsze trzeba polegać na danych przechowywanych w widoku stanowiącym część kontroli listy, utworzonym w adapterze przez metodę `getView()`. Czasami należy wywołać metodę `getItem()` adaptera, żeby otrzymać rzeczywiste dane, na których można operować. A czasami, podobnie jak w poprzednich przykładach z kontrolką `ListView`, obsługę danych zapewni kursor. Wszystko zależy od adaptera oraz od pochodzenia danych. Chociaż w tym przykładzie wykorzystaliśmy metodę `createScaledBitmap()`, w wersji 2.2 Androida wprowadzono kolejną klasę, która tutaj może okazać się przydatna — `ThumbnailUtils`. Zawiera ona pewne statyczne metody, służące do generowania miniaturek obrazów z bitmap oraz plików wideo.

Ostatnią kwestią, na którą warto zwrócić uwagę w tym przykładzie, jest wywołanie metody `getItemId()`. We wcześniejszych przykładach z kontrolkami `ListView` i kontaktami identyfikator obiektu posiadał wartość `_ID` otrzymywana od dostawcy treści. Ścisłe rzecz ujmując, w ostatnim przykładzie rolę identyfikatora spełniała informacja o położeniu obiektu. Cały sens istnienia identyfikatorów polega na zapewnieniu mechanizmu, który pozwala na odnoszenie się do danych z pozycji takiego identyfikatora. Jest to prawdą zwłaszcza wtedy, gdy dane są znacznie oddzielone od adaptera, co miało miejsce w przypadku kontaktów. Kiedy posiadamy taką bezpośrednią kontrolę nad danymi, podobnie jak w przypadku zdjęć krów morskich, oraz wiemy, w jaki sposób dostać się do właściwych danych w aplikacji, powszechnym rozwiązaniem jest wykorzystanie pozycji jako identyfikatora elementu. Jest to prawdziwe stwierdzenie zwłaszcza w naszym przypadku, gdyż nie pozwalamy na dodawanie lub usuwanie danych.

## Inne kontrolki w Androidzie

W Androidzie istnieje wiele różnych kontrolek. Dotychczas omówiliśmy kilka z nich, a kolejnymi zajmiemy się w dalszych rozdziałach (na przykład `MapView` w rozdziale 17., `VideoView` i `MediaController` w rozdziale 19., a `GLSurfaceView` w rozdziale 20.). Ponieważ kontrolki te wywodzą się z klasy `View`, stwierdzimy, że łączy je wiele wspólnych cech z dotychczas omówionymi pojednikami. Teraz jedynie wspomnijmy o kilku kontrolkach, które warto poznac samodzielnie.

Kontrolka `ScrollBar` służy do ustawiania w kontenerze typu `View` pionowego paska przewijania. Jest to bardzo przydatna kontrolka, w przypadku gdy treść nie mieści się na ekranie. W podrozdziale „Odnośniki” zamieszczono adres do bloga Romain Guya, w którym omówiono sposób korzystania z tej kontrolki.

Kontrolki `ProgressBar` i `RatingBar` przypominają suwaki. Pierwszy z nich w sposób wizualny prezentuje stopień postępu jakiejś czynności (na przykład pobieranie pliku lub odsłuchiwanie muzyki), natomiast za pomocą drugiego jest prezentowana skala oceniania za pomocą gwiazdek.

Kontrolka `Chronometer` stanowi czasomierz. Jeżeli chcemy wprowadzić funkcję stopera, można wykorzystać klasę `CountDownTimer`, nie jest ona jednak częścią klasy `View`.

`WebView` stanowi bardzo specyficzny widok pozwalający na wyświetlanie stron HTML. Jego funkcjonalność na tym się nie kończy. Kontrolka ta może również obsługiwać pliki cookies, język JavaScript oraz połączenia z kodem Java, znajdującym się w naszej aplikacji. Zanim jednak zaimplementujemy przeglądarkę internetową w tworzonym programie, warto ostrożnie rozważyć możliwość przywoływania przeglądarki wbudowanej w urządzenie. Ta wbudowana przeglądarka przeprowadzałaby wszystkie wymagane operacje.

W ten sposób kończymy wprowadzenie do kontrolek. Zajmiemy się teraz stylami i motywami modyfikującymi wygląd i zachowanie kontrolek, a następnie układami graficznymi pozwalającymi na rozmieszczanie kontrolek na ekranie.

## Style i motywy

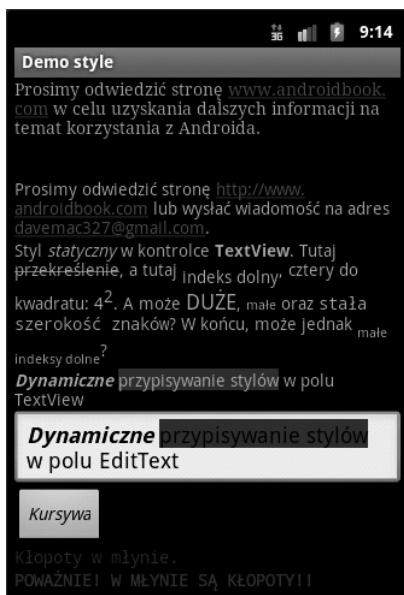
Android posiada kilka mechanizmów pozwalających na zmianę stylu widoków w aplikacji. Najpierw zajmiemy się znacznikami wprowadzanymi do ciągów znaków, a następnie zaprezentujemy sposób użycia obiektów klasy Spannable, dzięki którym zmienimy określone, wizualne atrybuty tekstu. Jednak co można zrobić w razie potrzeby kontrolowania wyglądu kontrollek za pomocą specyfikacji wspólnej dla wielu widoków lub dla całej aktywności czy aplikacji? Odpowiedzi udzielimy w trakcie omawiania stylów i motywów stosowanych w systemie Android.

### Stosowanie stylów

Czasami chcemy podświetlić lub zaznaczyć odmiennym stylem jakiś fragment treści zawartej w kontrolce klasy `View`. Można tego dokonać w sposób statyczny lub dynamiczny. Metoda statyczna polega na wstawieniu znaczników bezpośrednio do ciągu znaków w zasobach typu `string`, na przykład:

```
<string name="styledText"> Styl <i>statyczny</i> w polu <b>TextView</b>. </string>
```

Możemy następnie utworzyć odniesienie w pliku XML lub kodzie. Warto wiedzieć, że w zasobach typu `string` dostępne są znaczniki `<i>`, `<b>` oraz `<u>` języka HTML, odpowiadające, kolejno, pochyleniu czcionki, jej pogrubieniu oraz podkreśleniu. Istnieją także takie znaczniki, jak `<sup>` (indeks górnny), `<sub>` (indeks dolny), `<strike>` (przekreślenie), `<big>`, `<small>` oraz `<monospace>`. Możemy nawet tworzyć zagnieżdżenia, na przykład pomniejszone indeksy dolne. Style działają nie tylko w kontrolce `TextView`, ale także w innych, na przykład w przyciskach. Na rysunku 6.17 widzimy wygląd tekstu zmodyfikowanego za pomocą stylów i motywów, na którym można zobaczyć wiele przykładów omówionych w tym podrozdziale.



Rysunek 6.17. Przykładowe style i motywy

Programistyczne tworzenie stylów treści umieszczonej w kontrolce TextView wymaga nieco więcej wysiłku, jednak oferuje o wiele więcej możliwości (listing 6.35), ponieważ efekty zastosowania danego stylu mogą stawać się widoczne podczas działania aplikacji. Taka elastyczność może zostać jednak osiągnięta wyłącznie za pomocą obiektu klasy Spannable. Kontrolka EditText standardowo obsługuje w ten sposób wewnętrzny tekst, podczas gdy widok TextView zwykle nie korzysta z takich obiektów. Obiekt klasy Spannable jest przeważnie zwykłym ciągiem znaków, do którego można wprowadzać style. Aby kontrolka TextView przechowywała tekst w postaci obiektu Spannable, można wywołać metodę setText() w następujący sposób:

```
tv.setText("Ten tekst jest przechowywany w obiekcie klasy Spannable",
    →TextView.BufferType.SPANNABLE);
```

Następnie, podczas wywoływania metody tv.getText(), otrzymamy obiekt klasy Spannable.

Jak zostało pokazane na listingu 6.35, możemy pobrać zawartość kontrolki EditText (w postaci obiektu klasy Spannable), a następnie ustanawiać style dla poszczególnych fragmentów tekstu. Kod widoczny na listingu pogrubia tekst i pochyla go, a także generuje czerwone tło. Możemy tu zastosować wszystkie wymienione wcześniej opcje formatowania tekstu.

#### **Listing 6.35.** Dynamiczne umieszczanie stylów w treści kontrolki EditText

---

```
EditText et = (EditText) this.findViewById(R.id.et);
et.setText("Dynamiczne przypisywanie stylów zawartości pola EditText");
Spannable spn = (Spannable) et.getText();
spn.setSpan(new BackgroundColorSpan(Color.RED), 11, 31,
Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
spn.setSpan(new StyleSpan(android.graphics.Typeface.BOLD_ITALIC)
, 11, 31, Spannable.SPAN_EXCLUSIVE_EXCLUSIVE);
```

---

Te dwie techniki formatowania tekstu działają tylko w stosunku do tego widoku, do którego je zastosowano. W Androidzie istnieje również mechanizm umożliwiający definiowanie ogólnego stylu, który będzie wykorzystywany przez wiele widoków, a także mechanizm motywów, który w ogólnej zasadzie pozwala na zastosowanie danego stylu w obrębie całej aktywności lub aplikacji. Najpierw jednak musimy omówić style.

**Stylem** nazywamy zbiór atrybutów obiektu klasy View, posiadający osobną nazwę, możliwość przypisywania do widoków oraz taki, do którego możemy się później odnosić. Na przykład na listingu 6.36 widzimy plik XML, zapisany w katalogu *res/values*, który może być stosowany dla komunikatów o wszystkich rodzajach błędów.

#### **Listing 6.36.** Definiowanie stylu, który będzie wykorzystywany w wielu widokach

---

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="ErrorText">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#FF0000</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

---

Zdefiniowano tu rozmiar widoku, jak również kolor czcionki (w tym przypadku czerwony), a także jej krój. Zauważmy, że atrybut name znacznika elementu jest również atrybutem używanym w plikach układu graficznego, a wartość znacznika item nie wymaga już cudzysłowu. Możemy wykorzystać ten styl w kontrolce TextView wyświetlającej komunikaty o błędach, tak jak zostało to zaprezentowane na listingu 6.37.

---

**Listing 6.37.** Umieszczanie stylu w widoku

```
<TextView android:id="@+id/errorText"
    style="@style/ErrorText"
    android:text="W tej chwili nie ma błędów"
/>>
```

---

Istotna jest informacja, że nazwa atrybutu dla stylu w tej definicji obiektu klasy View nie rozpoczyna się od członu android:. Należy na to uważać, ponieważ wszystkie inne parametry zawierają w sobie człon android:. Jeżeli mamy w aplikacji wiele widoków współdzielących dany styl, jego zmiana w jednym miejscu jest o wiele łatwiejsza, wystarczy zmienić dane atrybuty w pojedynczym pliku zasobów. Możemy, oczywiście, również tworzyć wiele różnych stylów dla oddzielnich kontrolek. Na przykład przyciski mogą korzystać z jednego stylu, który będzie się różnił od stylu zastosowanego w tekście z menu.

Jednym z najprzyjemniejszych aspektów stylów jest możliwość utworzenia ich hierarchii. Na podstawie stylu ErrorText możemy utworzyć oddzielnny styl dla komunikatów o naprawdę groźnych błędach. Na listingu 6.38 zaprezentowaliśmy jedną z propozycji.

---

**Listing 6.38.** Definiowanie stylu na podstawie stylu nadzawanego

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="ErrorText.Danger" >
        <item name="android:textStyle">bold</item>
    </style>
</resources>
```

---

Przykład ten pokazuje, że możemy w prosty sposób nazwać nasz potomny styl, stosując nazwę stylu nadzawanego jako przedrostek. Zatem styl ErrorText.Danger jest potomny wobec stylu ErrorText i dziedziczy atrybuty rodzica. Następnie dodaje nową wartość we właściwości textStyle. W podobny sposób można utworzyć całe drzewo hierarchii stylów.

Podobnie jak mieliśmy do czynienia z układami graficznymi, system Android został wyposażony w spory zestaw predefiniowanych stylów. Aby wykorzystać któryś z nich, stosujemy następującą składnię:

style="@android:style/TextAppearance"

W ten sposób zdefiniowano domyślny styl formatowania tekstu w Androidzie. Główny plik stylów styles.xml znajdziemy w katalogu *Android SDK/platforms/<wersja-androida>/data/res/values/*. Wewnątrz tego pliku znajdziemy kilka przygotowanych stylów, które możemy wykorzystać lub rozszerzać. Jesteśmy jeszcze winni Czytelnikowi ostrzeżenie odnośnie do rozszerzania predefiniowanych stylów: wspomniana wcześniej metoda dodawania przedrostka do nazwy w przypadku tych stylów nie zadziała. Zamiast tego trzeba wykorzystać nadzędny atrybut znacznika style, na przykład tak:

```
<style name="CustomTextAppearance" parent="@android:style/TextAppearance">
    <item ... tutaj umieszczamy rozszerzenia ... />
</style>
```

Nie musimy zawsze stosować całego stylu wobec widoku. Możemy w razie potrzeby wprowadzić tylko jego fragment. Jeżeli na przykład chcemy, aby kolor tekstu w kontrolce TextView odpowiadał kolorowi systemowemu, możemy tego dokonać w poniższy sposób:

```
<EditText id="@+id/et2"
    android:layout_width="fill_parent" android:layout_height="wrap_content"
    android:textColor="?android:textColorSecondary"
    android:text="@string/hello_world" />
```

Zauważmy, że w tym przykładzie wartość atrybutu `textColor` rozpoczyna się od symbolu ?, a nie @. Dzięki temu znakowi Android poszukuje wartości stylu w bieżącym motywie. Ponieważ mamy do czynienia z członem ?android, szukamy tej wartości w motywie systemowym Androida.

## Stosowanie motywów

Problemem dotyczącym stylów jest konieczność dodawania specyfikacji atrybutu `style` `→="@style/..."` do każdej definicji widoku, do którego dany styl ma zostać zastosowany. Jeżeli chcemy wprowadzić pewne elementy formatowania w zakresie całej aktywności lub aplikacji, do tego celu najlepiej nadaje się motyw. Zasadniczo **motyw** jest stylem, który może zostać zastosowany w szerszym zakresie, natomiast pod kątem definiowania nie różni się niczym od stylu. W rzeczywistości style i motywy są dość często stosowane zamiennie, ponieważ można rozszerzyć motyw o styl albo odnosić się w motywie do stylu. Zazwyczaj potrafimy rozpoznać jedynie po nazewnictwie, czy styl pełni rolę stylu, czy też motywu.

W celu zdefiniowania motywu dla aktywności lub aplikacji musimy dodać odpowiedni atrybut w znaczniku `<activity>` lub `<application>` w pliku `AndroidManifest.xml` danego projektu. Ten kod może wyglądać następująco:

```
<activity android:theme="@style/MyActivityTheme">
<application android:theme="@style/MyApplicationTheme">
<application android:theme="@android:style/Theme.NoTitleBar">
```

Predefiniowane motywy Androida można znaleźć w tym samym katalogu co predefiniowane style. Zostały one umieszczone w pliku `themes.xml`. Po otwarciu tego pliku ujrzymy olbrzymi zbiór zdefiniowanych stylów, których nazwy rozpoczynają się od członu `Theme`. Warto także zauważyć, że w kodzie tych stylów i motywów bardzo często widać zapisy świadczące o rozszerzeniach, dlatego nie powinna dziwić taka nazwa stylu jak na przykład `Theme.Dialog.AppError`.

Na tym zakończymy omawianie zestawu kontrolek w Androidzie. Jak zostało wspomniane na początku rozdziału, opanowanie sztuki budowania interfejsów użytkownika wymaga znajomości dwóch elementów: zestawu kontrolek oraz menedżerów układu graficznego. W kolejnym podrozdziale zajmiemy się drugim z wymienionych składników.

## Menedżery układu graficznego

Android zawiera zbiór klas widoku, pełniących rolę pojemników na widoki. Te kontenerowe klasy noszą nazwę **układów graficznych** (ang. *layout*) lub **menedżerów układów graficznych** (ang. *layout manager*). Każda z nich wnosi określony sposób zarządzania rozmiarem oraz pozycją

elementów podrzędnych. Na przykład klasa *LinearLayout* umieszcza elementy potomne jeden za drugim w orientacji poziomej lub pionowej. Wszystkie menedżery układów graficznych wywodzą się z klasy *View*, zatem możemy je zagnieździć jeden w drugim.

Dostępne w zestawie Android SDK menedżery układu graficznego zdefiniowano w tabeli 6.2.

**Tabela 6.2.** Menedżery układu graficznego w Androidzie

Menedżer układu graficznego	Opis
<i>LinearLayout</i>	Rozmieszcza elementy podrzędne w pionie lub poziomie.
<i>TableLayout</i>	Rozmieszcza elementy podrzędne w postaci tabelarycznej.
<i>RelativeLayout</i>	Rozmieszcza elementy podrzędne względem innych elementów lub pojemnika nadzawanego.
<i>FrameLayout</i>	Umożliwia dynamiczne zmienianie kontrolki (kontrolek) w układzie graficznym.

Wymienione menedżery układu graficznego zostaną omówione w następnych punktach. Menedżer *AbsoluteLayout* jest już przestarzały, więc zostanie w tej książce pominięty.

## Menedżer układu graficznego *LinearLayout*

Klasa *LinearLayout* stanowi przykład najprostszego układu graficznego. Menedżer ten rozmieszcza elementy potomne w poziomie lub w pionie, zależnie od wartości właściwości *orientation*. Do tej pory zdążyliśmy już kilkakrotnie wykorzystać ten układ graficzny. Na listingu 6.39 przedstawiono konfigurację poziomą elementów potomnych.

**Listing 6.39.** Klasa *LinearLayout* z wprowadzoną konfiguracją poziomą

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">

    <!-- tutaj są dodawane elementy potomne-->

</LinearLayout>
```

Klasę *LinearLayout* można przekształcić do orientacji pionowej poprzez zmianę wartości właściwości *orientation* na *vertical*. Ponieważ menedżery układów graficznych mogą być zagnieżdżane, moglibyśmy na przykład utworzyć formularz składający się z pionowego układu graficznego zawierającego w sobie poziome menedżery. Wszystkie wiersze posiadałyby etykietę tuż obok kontrolki *EditText*. Każdy taki wiersz byłby swoim własnym poziomym układem graficznym, ale zbiór tych wierszy byłby ułożony w pionie.

## Ciężar oraz grawitacja

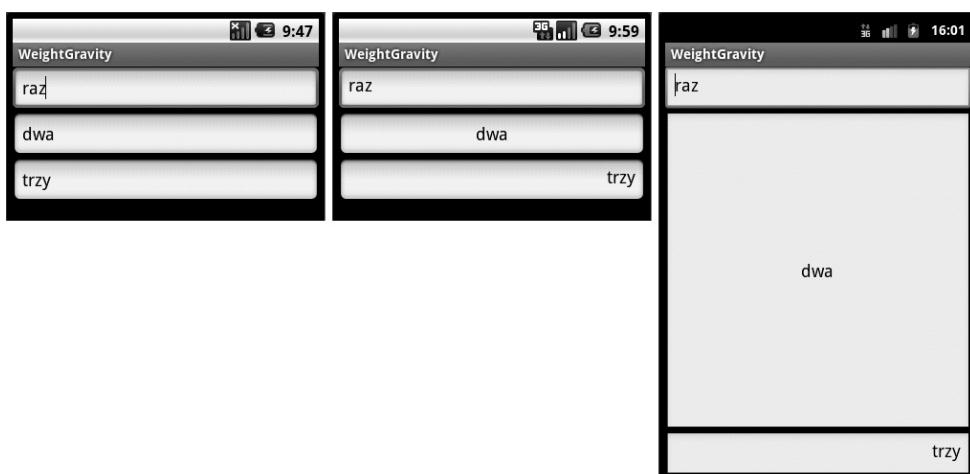
Właściwość *orientation* jest pierwszym z istotnych atrybutów rozpoznawanych przez menedżer *LinearLayout*. Innymi ważnymi właściwościami, wpływającymi na rozmiary oraz rozmieszczenie kontrolek potomnych, są **ciężar** (ang. *weight*) i **grawitacja** (ang. *gravity*). Dzięki ciężarowi przypisywany jest stopień ważności kontrolki w odniesieniu do innych kontrolek

w pojemniku. Założymy, że w pojemniku umieszczono trzy kontrolki: jedna ma zdefiniowany ciężar równy 1, pozostały dwóm została natomiast przypisana wartość 0. W takim przypadku kontrolka o wartości ciężaru 1 zajmie całą niezajętą przestrzeń pojemnika. Grawitacja w zasadzie określa sposób wyrównania elementu w układzie graficznym. Jeżeli na przykład tekst etykiety ma zostać wyrównany do prawej strony, należy przypisać atrybutowi gravity wartość right. Dostępnych jest kilka różnych wartości atrybutu gravity, na przykład left, center, right, top, bottom, center\_vertical, clip\_horizontal i inne. Szczegóły dotyczące tych oraz pozostałych wartości tej właściwości można znaleźć, korzystając z adresów umieszczonych w podrozdziale „Odnośniki”.

**Uwaga!**

Menedżery układu graficznego rozszerzają klasę android.widget.ViewGroup, podobnie jak wiele klas pojemników opartych na kontrolkach, na przykład ListView. Chociaż rozszerzają one tę samą klasę, klasy menedżerów układu graficznego służą wyłącznie do definiowania rozmiarów oraz rozmieszczenia kontrolek, a nie do interakcji użytkownika z kontrolkami potomnymi. Na przykład porównajmy obiekty LinearLayout i ListView. Na ekranie wyglądają podobnie, gdyż obydwa mogą organizować elementy potomne w orientacji pionowej. Jednak kontrolka ListView zawiera interfejsy API umożliwiające użytkownikowi zaznaczanie elementów, czego nie można powiedzieć o klasie LinearLayout. Innymi słowy, pojemnik oparty na kontrolkach (ListView) obsługuje interakcję użytkownika z elementami w nich umieszczonymi, podczas gdy menedżer układu graficznego (LinearLayout) zajmuje się jedynie określaniem rozmiarów i rozmieszczeniem potomków.

Przyjrzyjmy się przykładowi związanemu z właściwościami ciężaru i grawitacji (rysunek 6.18).



**Rysunek 6.18.** Stosowanie menedżera układu graficznego LinearLayout

Na rysunku 6.18 pokazano trzy interfejsy użytkownika wykorzystujące menedżer LinearLayout, z których każdy posiada inną konfigurację ciężaru i grawitacji. Interfejs ukazany po lewej stronie posiada domyślne ustawienia ciężaru i grawitacji. Plik XML tego układu graficznego zawiera kod zaprezentowany na listingu 6.40.

**Listing 6.40.** Trzy pola tekstowe umieszczone pionowo w menedżerze LinearLayout przy domyślnych ustawieniach ciężaru i grawitacji

---

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <EditText android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="raz"/>
    <EditText android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="dwa"/>
    <EditText android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="trzy"/>
</LinearLayout>
```

---

Środkowy interfejs na rysunku 6.18 zawiera domyślną wartość ciężaru, ale parametry argumentu `android:gravity` zostały zdefiniowane dla kontrolek w kolejności: `left`, `center`, `right`. W przykładzie po prawej atrybut `android:layout_weight` środkowego elementu wynosi 1.0, natomiast w pozostałych dwóch kontrolkach nie zmieniono domyślnej wartości 1.0 (listing 6.41). Sprawiamy w ten sposób, że środkowy element zajmie całą wolną przestrzeń pojemnika nadzawanego, a dwie skrajne kontrolki pozostaną przy swoich domyślnych rozmiarach.

**Listing 6.41.** Konfigurowanie ciężaru w menedżerze LinearLayout

---

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <EditText android:layout_width="fill_parent" android:layout_weight="0.0"
        android:layout_height="wrap_content" android:text="raz"
        android:gravity="left"/>
    <EditText android:layout_width="fill_parent" android:layout_weight="1.0"
        android:layout_height="wrap_content" android:text="dwa"
        android:gravity="center"/>
    <EditText android:layout_width="fill_parent" android:layout_weight="0.0"
        android:layout_height="wrap_content" android:text="trzy"
        android:gravity="right"
    />
</LinearLayout>
```

---

Analogicznie, jeżeli chcemy, żeby dwie kontrolki z trzech podzieliły między siebie pozostałą wolną przestrzeń, wprowadzamy im wartość ciężaru równą 1.0, w trzeciej wartości natomiast pozostawiamy ten argument niezmieniony (0.0). W końcu trzecią możliwością jest podzielenie ekranu między trzy kontrolki w równym stopniu, osiągnięte poprzez przydzielenie każdej z nich wartości ciężaru wynoszącej 1.0. W ten sposób każde pole tekstowe zostanie rozciagnięte w takim samym stopniu.

## Porównanie atrybutów android:gravity i android:layout\_gravity

Zwróćmy uwagę, że w Androidzie zdefiniowano dwa podobne atrybuty: android:gravity oraz android:layout\_gravity. Różnica polega na tym, że android:gravity jest używany przez widok, a android:layout\_gravity przez pojemnik (android.view.ViewGroup). Na przykład można ustawić wartość atrybutu android:gravity na center, aby wyśrodkować tekst zawarty w kontrolce EditText. Natomiast aby umieścić kontrolkę EditText po prawej stronie pojemnika LinearLayout, należy wpisać następujący wiersz: android:layout\_gravity="right" (rysunek 6.19 i listing 6.42).



**Rysunek 6.19.** Wprowadzanie ustawień grawitacji

**Listing 6.42.** Pokazanie różnicy pomiędzy atrybutami android:gravity a android:layout\_gravity

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <EditText android:layout_width="wrap_content" android:gravity="center"
        android:layout_height="wrap_content" android:text="raz"
        android:layout_gravity="right"/>
</LinearLayout>
```

Na rysunku 6.19 widać, że wewnętrz kontrolki EditText zawartość jest wyśrodkowana, natomiast sama kontrolka została umieszczona po prawej stronie pojemnika LinearLayout.

## Menedżer układu graficznego TableLayout

Menedżer układu graficznego TableLayout jest rozwinięciem menedżera LinearLayout. Potomne kontrolki są w nim układane w wierszach i kolumnach. Na listingu 6.43 pokazano przykład:

**Listing 6.43.** Prosty menedżer TableLayout

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

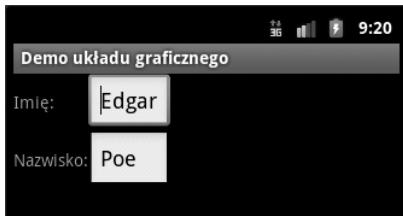
    <TableRow>
        <TextView android:text="Imię:" android:layout_width="wrap_content" android:layout_height="wrap_content" />
        <EditText android:text="Edgar" android:layout_width="wrap_content" android:layout_height="wrap_content" />
    </TableRow>
</TableLayout>
```

```
<TextView android:text="Nazwisko"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />

<EditText android:text="Poe"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />
</TableRow>

</TableLayout>
```

W celu skorzystania z menedżera TableLayout należy utworzyć jegoinstancję, a następnie umieścić w nim elementy TableRow. W nich są przechowywane kontrolki tabeli. Interfejs użytkownika utworzony na listingu 6.43 został pokazany na rysunku 6.20.



Rysunek 6.20. Menedżer układu graficznego TableLayout

Ponieważ elementy menedżera TableLayout są definiowane w wierszach, a nie w kolumnach, Android określa liczbę kolumn tabeli poprzez wyszukanie wiersza zawierającego największą liczbę komórek. Na przykład na listingu 6.44 utworzono dwa wiersze, gdzie pierwszy wiersz zawiera dwie komórki, a drugi — trzy (rysunek 6.21). W takim przypadku Android tworzy tabelę zawierającą dwa wiersze i trzy kolumny. Komórka znajdująca się w trzeciej kolumnie pierwszego wiersza jest pusta.

#### **Listing 6.44.** Definicja nieregularnej tabeli

---

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TableRow>
        <TextView android:text="Imię:"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />

        <EditText android:text="Edgar"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />

        </TableRow>

    <TableRow>
        <TextView android:text="Nazwisko:"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />

        <EditText android:text="Allan"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />

        </TableRow>

```

---

```

        android:layout_width="wrap_content" android:layout_height="wrap_content" />

<EditText android:text="Poe"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />

</TableRow>

</TableLayout>

```

---



**Rysunek 6.21.** Menedżer TableLayout reprezentujący nieregularną tabelę

Na listingach 6.43 oraz 6.44 wypełniliśmy menedżer TableLayout elementami TableRow. Chociaż jest to zwyczajne podejście, można także umieścić dowolny element android.widget.View jako potomka tabeli. Na przykład w listingu 6.45 utworzono tabelę, w której pierwszym wierszem jest kontrolka EditText (rysunek 6.22).

**Listing 6.45.** Zastosowanie kontrolki EditText zamiast TableRow

---

```

<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="0,1,2">

<EditText
    android:text="Imię i nazwisko:"/>

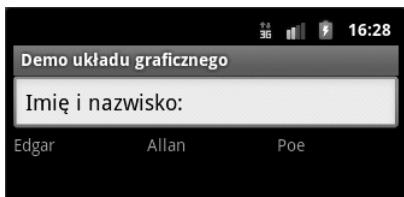
<TableRow>
    <TextView android:text="Edgar"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <TextView android:text="Allan"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <TextView android:text="Poe"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</TableRow>

</TableLayout>

```

---

Interfejs użytkownika utworzony na listingu 6.45 został ukazany na rysunku 6.22. Zauważmy, że kontrolka EditText zajmuje całą szerokość ekranu, chociaż nawet nie zawarliśmy odpowiedniego parametru w układzie graficznym XML. Wynika to z faktu, że wszystkie elementy



Rysunek 6.22. Kontrolka EditText jako potomek menedżera TableLayout

podrzędne menedżera TableLayout są zawsze rozciągane na długość wiersza. Innymi słowy, elementy potomne menedżera TableLayout mogą definiować parametr `android:layout_width`  $\Rightarrow$  `"wrap_content"` (podobnie jak to zrobiliśmy w kontrolce EditText), nie wpłynie to jednak na rzeczywisty układ graficzny — muszą przyjmować wartość `fill_parent`. Można tu jednak dostosować argument `android:layout_height`.

Ponieważ zawartość tabeli nie zawsze jest znana podczas etapu projektowania, menedżer TableLayout posiada kilka atrybutów umożliwiających kontrolowanie układu graficznego tabeli. Na przykład na listingu 6.45 widać, że parametry właściwości `android:stretchColumns` wynoszą odpowiednio 0, 1, 2. Dla menedżera TableLayout oznacza to, że w zależności od treści kolumny nr 0, 1 oraz 2 zostaną rozciągnięte. Gdyby na listingu 6.45 nie użyto argumentu `stretchColumns`, efektem byłoby złączenie trzech wyrazów „EdgarAllanPoe”. Pod względem technicznym drugi rząd zajmuje całą szerokość, ale trzy kontrolki TextView nie znajdują się na niego.

W podobny sposób można za pomocą argumentu `android:shrinkColumns` ścinać zawartość wybranych kolumn, gdy inne kolumny potrzebują więcej miejsca. Można także wprowadzić właściwość `android:collapseColumns`, dzięki której wybrane kolumny stają się niewidoczne. Należy pamiętać, że kolumny są numerowane, począwszy od cyfry 0.

Menedżer TableLayout zawiera także atrybut `android:layout_span`. Jest on używany do rozciągnięcia komórki na wiele kolumn. Atrybut ten przypomina właściwość `colspan` w języku HTML.

Nieraz pojawi się potrzeba utworzenia odstępów wewnętrznych zawartości komórki lub kontrolki. Przydatny jest wtedy atrybut `android:padding` oraz jemu podobne. Dzięki niemu możliwe jest kontrolowanie przestrzeni pomiędzy zewnętrznymi granicami widoku a jego treścią (listing 6.46).

#### **Listing 6.46.** Zastosowanie właściwości `android:padding`

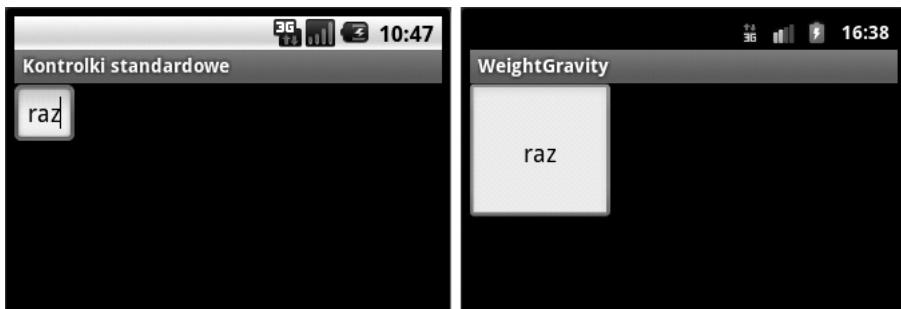
---

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <EditText android:text="raz"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:padding="40px" />
</LinearLayout>
```

---

Na listingu 6.46 zostały wyznaczone odstępy równe 40px. W ten sposób 40 pikseli białej przestrzeni oddziela treść kontrolki EditText od jej krańców. Na rysunku 6.23 zademonstrowano tę samą kontrolkę EditText, której nadano dwie różne wartości odstępów. Interfejs po lewej stronie nie ma zdefiniowanych żadnych odstępów, a interfejs po prawej posiada dopisaną linijkę `android:padding="40px"`.



Rysunek 6.23. Zastosowanie odstępów

Atrybut `android:padding` tworzy odstępy we wszystkich kierunkach: w lewo, w prawo, w górę i w dół. Można kontrolować rozmiar odstępu dla każdej strony dzięki atrybutom `android:leftPadding`, `android:rightPadding`, `android:topPadding` oraz `android:bottomPadding`.

W Androidzie można również zdefiniować atrybut `android:layout_margin`, bardzo przypominający `android:padding`. W rzeczywistości różnica pomiędzy atrybutami `android:padding`/`→ android:layout_margin` jest taka jak w przypadku `android:gravity`/`android:layout_gravity`. Oznacza to, że jeden atrybut jest przeznaczony dla widoku, a drugi dla pojemnika.

W końcu należy zwrócić uwagę, że wartość odstępu jest zawsze wyrażana jako wymiar określonego typu. Android obsługuje następujące typy wymiarów:

- **Piksele** — w skrócie px. Wymiar ten reprezentuje fizyczne piksele ekranu.
- **Cale** — w skrócie in. Jednostka ta reprezentuje rzeczywisty rozmiar obiektu na wyświetlaczu w calach.
- **Milimetry** — w skrócie mm. Jednostka ta reprezentuje rzeczywisty rozmiar obiektu na wyświetlaczu w milimetrach.
- **Punkty** — w skrócie pt. Jeden punkt jest równy  $1/72$  cala.
- **Piksele niezależne od gęstości** — w skrócie dip albo dp. W tym przypadku ekran o gęstości 160 pikseli na cal jest wykorzystywany jako ramka odniesienia, dla której wymiary obiektu są odwzorowywane na rzeczywistym ekranie. Na przykład ekran o szerokości 160 pikseli będzie odwzorowywał 1 dip na 1 piksel.
- **Piksele niezależne od skali** — w skrócie sp. Wymiar używany przeważnie przy pracy z czcionkami. Są tu brane pod uwagę ustawienia użytkownika oraz rozmiar czcionki w celu określenia rzeczywistego wymiaru.

Warto zapamiętać, że wymienione tu rodzaje wymiarów nie są przeznaczone wyłącznie do definiowania odstępów — każdy obiekt w Androidzie akceptujący wartości wymiaru (na przykład `android:layout_width` lub `android:layout_height`) będzie w stanie odczytać te jednostki.

## Menedżer układu graficznego **RelativeLayout**

Kolejnym interesującym menedżerem układu graficznego jest `RelativeLayout`. Jak sama nazwa sugeruje, menedżer ten implementuje zasady, dzięki którym pozycja kontrollek znajdujących się w pojemniku jest zależna od pojemnika lub innej umieszczonej w nim kontrolki. Odpowiedni przykład pokazano na listingu 6.47 oraz na rysunku 6.24.

**Listing 6.47.** Zastosowanie menedżera układu graficznego RelativeLayout

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">

    <TextView android:id="@+id/userNameLbl"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Nazwa użytkownika: "
        android:layout_alignParentTop="true" />

    <EditText android:id="@+id/userNameText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/userNameLbl" />

    <TextView android:id="@+id/pwdLbl"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/userNameText"
        android:text="Hasło: " />

    <EditText android:id="@+id/pwdText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/pwdLbl" />

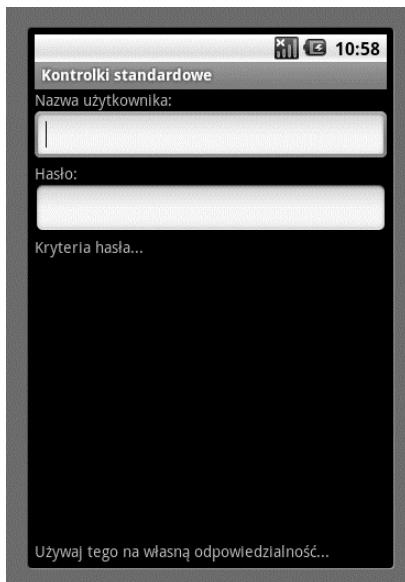
    <TextView android:id="@+id/pwdCriteria"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@+id/pwdText"
        android:text="Kryteria hasła... " />

    <TextView android:id="@+id/disclaimerLbl"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:text="Używaj tego na własną odpowiedzialność... " />

</RelativeLayout>
```

Jak widać, interfejs użytkownika przypomina prosty formularz logowania. Etykieta nazwy użytkownika została przypięta do górnej części pojemnika, ponieważ atrybutowi `android:layout_alignParentTop` przypisano wartość `true`. W podobny sposób pole wpisywania nazwy użytkownika znalazło się poniżej etykiety, gdyż wprowadzono atrybut `android:layout_below`. Etykieta hasła jest umieszczona jeszcze niżej, pod nią zaś wprowadzono pole wpisywania hasła. Ostrzeżenie znalazło się na samym dole pojemnika, gdyż w tej kontrolce atrybutowi `android:layout_alignParentBottom` nadano wartość `true`.

Poza trzema wymienionymi atrybutami dostępne są jeszcze inne, takie jak `layout_above`, `layout_toRightOf`, `layout_toLeftOf`, `layout_centerInParent` i inne. Praca z menedżerem `RelativeLayout` jest przyjemna z powodu łatwości jego obsługi. Istotnie, dla każdego, kto zacznie go używać, stanie się on ulubionym menedżerem układu graficznego — bez przerwy będzie się do niego wracać.



**Rysunek 6.24.** Interfejs użytkownika utworzony za pomocą menedżera układu graficznego RelativeLayout

## Menedżer układu graficznego FrameLayout

Menedżery układu graficznego omówione do tej pory wprowadzają różne strategie kompozycji elementów. Innymi słowy, każdy z nich w określony sposób wstawia i orientuje kontrolki potomne na ekranie. Dzięki menedżerom można wstawić na jednym ekranie jednocześnie wiele kontrolek, z których każda zajmuje określony fragment pojemnika. W Androidzie dostępny jest także menedżer układu graficznego, używany przede wszystkim do wyświetlania pojedynczego składnika — FrameLayout. Ta klasa układu graficznego służy głównie do dynamicznego wyświetlania pojedynczego widoku, można ją jednak zapełnić wieloma elementami, z których jeden będzie widoczny, a pozostałe nie. Na listingu 6.48 zademonstrowano zastosowanie menedżera FrameLayout.

**Listing 6.48.** Zapełnianie menedżera FrameLayout

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/frmLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <ImageView
        android:id="@+id/oneImgView" android:src="@drawable/one"
        android:scaleType="fitCenter"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>
    <ImageView
        android:id="@+id/twoImgView" android:src="@drawable/two"
        android:scaleType="fitCenter"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>

```

```
        android:scaleType="fitCenter"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:visibility="gone" />

    </FrameLayout>

public class FrameLayoutActivity extends Activity{
    private ImageView one = null;
    private ImageView two = null;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.listing6_48);

        one = (ImageView)this.findViewById(R.id.oneImgView);
        two = (ImageView)this.findViewById(R.id.twoImgView);

        one.setOnClickListener(new OnClickListener(){

            public void onClick(View view) {
                two.setVisibility(View.VISIBLE);

                view.setVisibility(View.GONE);
            }
        });

        two.setOnClickListener(new OnClickListener(){

            public void onClick(View view) {
                one.setVisibility(View.VISIBLE);

                view.setVisibility(View.GONE);
            }
        });
    }
}
```

---

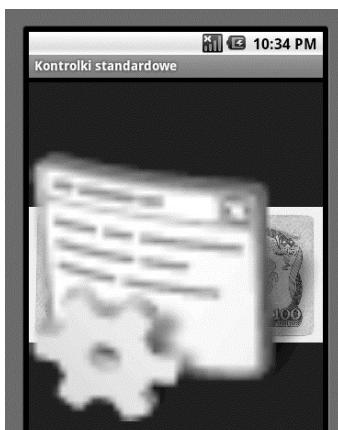
Listing 6.48 przedstawia plik układu graficznego oraz metodę `onCreate()` aktywności. Celem ćwiczenia jest wczytanie dwóch widoków `ImageView` w menedżerze `FrameLayout` w taki sposób, żeby w danym momencie był widoczny tylko jeden z nich. Na poziomie interfejsu użytkownika, kiedy użytkownik kliknie widoczny obrazek, aplikacja go schowa i wyświetli drugi obiekt.

Przyjrzyjmy się bliżej kodowi zamieszczonemu na listingu 6.48, począwszy od układu graficznego. Można zauważyc, że definiujemy menedżer `FrameLayout` zawierający dwie kontrolki `ImageView` (są one odpowiedzialne za właściwe wyświetlanie obrazów). Zwróćmy uwagę, że widoczność drugiego obiektu `ImageView` przyjmuje wartość `gone`, dzięki czemu staje się on niewidoczny. Spójrzmy teraz na metodę `onCreate()`. Rejestrujemy w niej elementy nasłuchujące, reagujące na kliknięcia widoków `ImageView`. W procedurze obsługi kliknięcia programujemy ukrycie jednego obiektu `ImageView` wraz z jednoczesnym wyświetleniem drugiego obiektu.

Jak już wcześniej wspomnieliśmy, menedżera `FrameLayout` zazwyczaj używa się podczas dynamicznego konfigurowania treści widoku w pojedynczej kontrole. Chociaż jest to standarowa praktyka, pokazaliśmy również, że kontrolka akceptuje także wiele obiektów potomnych. Na listingu 6.48 do układu graficznego dodano dwie kontrolki, jednak w danym momencie

widoczna jest tylko jedna z nich. Menedżer FrameLayout nie wymusza jednak takiego rozwiązania. Jeżeli do układu graficznego dodamy wiele kontrolek, Android po prostu utworzy ich stos, w którym jedna kontrolka będzie nałożona na drugą, ostatnia natomiast będzie się znajdować na jego szczycie. W ten sposób można stworzyć interesujący interfejs użytkownika. Na przykład na rysunku 6.25 pokazano menedżer FrameLayout, w którym są widoczne dwa widoki ImageView. Widać, że kontrolki są ułożone na stosie, a ta znajdująca się na wierzchu częściowo zasłania obiekt umieszczony za nią.

Kolejny interesujący fakt dotyczący menedżera FrameLayout jest taki, że po dodaniu do układu graficznego więcej niż jednej kontrolki rozmiar tego układu jest definiowany jako rozmiar największego elementu w pojemniku. Na rysunku 6.25 element znajdujący się na wierzchu jest w rzeczywistości znacznie mniejszy od elementu umieszczonego pod spodem, jednak ponieważ układ graficzny jest dopasowany do największego obiektu, obraz z pierwszego planu zostaje rozciągnięty.



**Rysunek 6.25.** Menedżer FrameLayout zawierający dwa widoki ImageView

Warto także pamiętać, że jeśli umieści się w menedżerze FrameLayout wiele elementów, z których część zostanie zdefiniowana jako niewidoczne, można rozważyć wykorzystanie metody `setMeasureAllChildren(true)` na układzie FrameLayout. Skoro największy element podrębny definiuje rozmiar całego układu graficznego, może się pojawić problem, gdy takim elementem okaże się obiekt niewidoczny. To znaczy, że jeżeli pojawi się na pierwszym planie, będzie widoczna jedynie jego część. Po wprowadzeniu metody `setMeasureAllChildren()` z wartością `true` wszystkie elementy powinny być prawidłowo wyświetlane. Równoważnym atrybutem XML dla układu FrameLayout jest `android:measureAllChildren="true"`.

## Dostosowanie układu graficznego do konfiguracji różnych urządzeń

Jak doskonale wiemy, że Android zawiera wiele menedżerów układu graficznego pomagających w budowaniu interfejsów użytkownika. Jeżeli ktoś już wypróbował omawiane przez nas menedżery, będzie wiedział, że można je łączyć na wiele sposobów w celu uzyskania pożądanego wyglądu i sposobu funkcjonowania danego interfejsu. Jednak nawet za ich pomocą tworzenie poprawnie działających interfejsów UI stanowi wyzwanie. Dotyczy to zwłaszcza urządzeń

przenośnych. Oczekiwania użytkowników oraz producentów urządzeń przenośnych stają się coraz bardziej wysublimowane, przez co poprzeczka stojąca przed programistą aplikacji zostaje podniesiona jeszcze wyżej.

Jednym z takich wyzwań jest tworzenie interfejsu użytkownika aplikacji, która będzie wyświetcona w różnych konfiguracjach wyświetlacza. Trzeba sobie odpowiedzieć na pytanie, jak wyglądałby taki interfejs UI na wyświetlaczu ułożonym w orientacji poziomej, a jak w pionowej? Czytelnicy, którzy jeszcze nie spotkali się z tym problemem, prawdopodobnie usilnie starają się teraz wyobrazić sobie rozwiążanie tego dosyć powszechnego scenariusza. Na szczęście Android w interesujący sposób pomaga poradzić sobie z tym problemem.

Działa to w następujący sposób: w trakcie tworzenia układu graficznego Android na podstawie konfiguracji urządzenia znajduje oraz wczytuje układy graficzne z określonych folderów. Urządzenie może się znajdować w jednej z trzech konfiguracji: pionowej (ang. *portrait*), poziomej (ang. *landscape*) lub umożliwiającej wyświetlanie kwadratowego obrazu (ang. *square*), która jest najrzadziej spotykana. Żeby układy graficzne były właściwie wyświetlane w tych różnych konfiguracjach, należy utworzyć dla nich oddzielne foldery, z których Android będzie wczytywał właściwe układy graficzne. Jak wiadomo, domyślnym folderem układu graficznego jest *res/layout*. Aby obsłużyć konfigurację pionową, należy utworzyć folder *res/layout-port*; dla trybu poziomego będzie to *res/layout-land*, a dla kwadratowego *res/layout-square*.

W tym momencie nasuwa się pytanie: jeśli zdefiniowano te trzy foldery, to czy jest potrzebny jeszcze domyślny katalog układu graficznego (*res/layout*)? Zasadniczo tak. Pamiętajmy, że logika Androida, polegająca na rozpoznawaniu zasobów, sprawdza najpierw katalog określony w konfiguracji. Jeżeli nie zostaną tam znalezione zasoby, system przechodzi do domyślnego folderu układu graficznego. Zatem można umieścić domyślne definicje układu graficznego w folderze *res/layout*, a ich odpowiednio dostosowane wersje w folderach konfiguracji.

Zauważmy, że środowisko Android SDK nie posiada interfejsów API, które pozwalałyby w sposób programistyczny określić rodzaj wczytywanej konfiguracji — system sam wybiera folder na podstawie wykrytej konfiguracji urządzenia. Można jednak w kodzie określić orientację urządzenia, na przykład w sposób przedstawiony poniżej:

```
import android.content.pm.ActivityInfo;  
...  
setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
```

W ten sposób urządzenie zostaje zmuszone do wyświetlania aplikacji w orientacji poziomej. Można śmiało wypróbować ten sposób we wcześniejszych projektach. Powyższy fragment kodu należy dodać do metody *onCreate()* danej aktywności, uruchomić ją w emulatorze i już można podziwiać odwróconą aplikację na wyświetlaczu.

Układ graficzny nie jest jedynym zasobem korzystającym z konfiguracji. Inne kwalifikatory konfiguracji urządzenia są również brane pod uwagę po znalezieniu odpowiedniego zasobu. Cała zawartość folderu *res* może posiadać odpowiedniki w każdej konfiguracji. Aby na przykład dla każdej konfiguracji przygotować obiekty rysowane, należy utworzyć katalogi *drawable-port*, *drawable-land* oraz *drawable-square*. Jednak Android jest jeszcze potężniejszy. W tabeli 6.3 wypisano pełną listę kwalifikatorów, które są wykorzystywane po znalezieniu zasobów.

Więcej informacji na temat tych kwalifikatorów można znaleźć na stronie Androida pod adresem:

<http://developer.android.com/guide/topics/resources/providing-resources.html#table2>

**Tabela 6.3.** Kwalifikatory zasobów

Kwalifikator	Opis
Numery MCC i MNC	Identyfikatory kraju oraz operatora sieci.
Język i region	Dwuliterowy kod języka pisany małymi literami, po dodaniu parametru - r także dwuliterowy kod regionu pisany dużymi literami.
Rozmiary ekranu	Daje ogólne pojęcie na temat rozmiaru ekranu; wartości: small, normal, large oraz xlarge.
Szersze/wyższe ekranы	Związane z proporcjami obrazu; wartości: long, notlong.
Orientacja ekranu	Wartości: land, port oraz square.
Gęstość pikseli na ekranie	Przybliżone gęstości: ldpi (ok. 120), mdpi (ok. 160), hdpi (ok. 240) oraz xhdpi (ok. 320). Android może dopasowywać zasoby znalezione w odpowiednich folderach, chyba że są one umieszczone w katalogu zawierającym kwalifikator nodpi.
Typ ekranu dotykowego	Wartości: finger, notouch oraz stylus.
Klawiatura	Rodzaj klawiatury. Wartości: keysexposed, keyshidden oraz keyssoft.
Rodzaj tekstowych danych wejściowych	Wartości: nokeys, qwerty oraz 12key (numeryczne).
Sterowanie przy braku klawiatury dotykowej	Wartości: dpad, nonav, trackball oraz wheel.
Wersja środowiska SDK	Wartości: v4 (SDK 1.6), v7 (SDK 2.1) itd.

Powysze kwalifikatory mogą być używane w wielu kombinacjach, aby uzyskać pożądane zachowanie. Nazwa katalogu zasobów może nie zawierać żadnej z wartości kwalifikatorów lub zawierać wiele takich wartości oddzielonych myślnikami. Na przykład poniżej pokazaliśmy poprawną pod względem technicznym (choć niezalecaną) nazwę katalogu zasobów typu drawable:

*drawable-mcc310-en-rUS-large-long-port-mdpi-stylus-keyssoft-qwerty-dpad-v3*

Można jednak zapisać to również w następujący sposób:

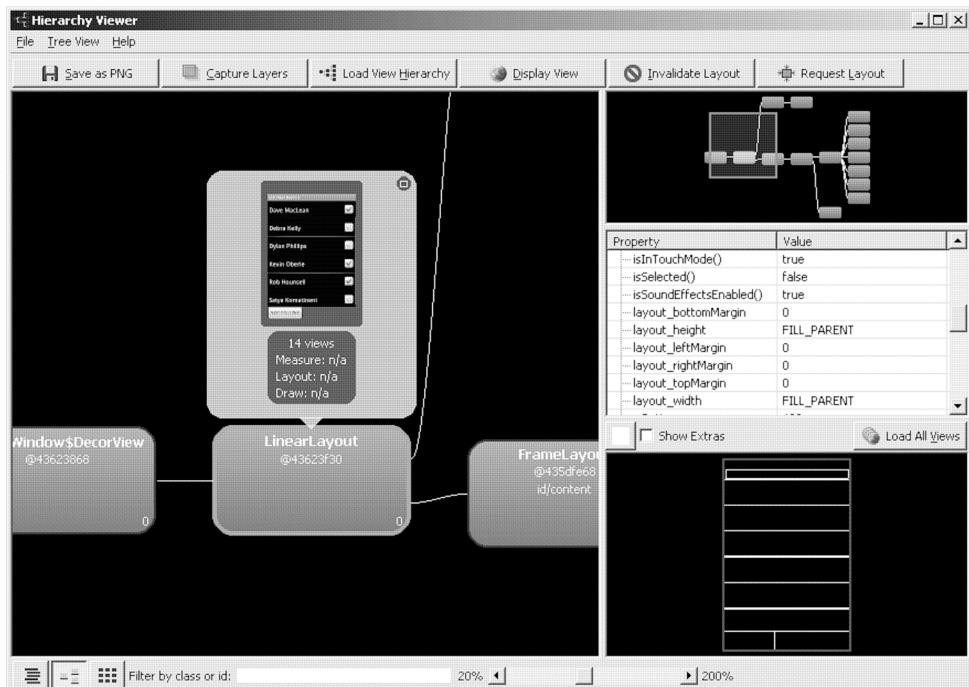
*drawable-en-rUS-land* (obrazy dla wersji angielskiej ze Stanów Zjednoczonych, w orientacji poziomej)

*values-fr* (ciąg znaków w języku francuskim)

Bez względu na liczbę kwalifikatorów wykorzystanych w zasobach aplikacji należy pamiętać, że w kodzie cały czas należy odwoływać się do zasobów w postaci R.rodzaj\_zasobu.nazwa, bez kwalifikatorów. Jeśli na przykład istnieje wiele różnych odmian pliku układu graficznego *main.xml* w różnych kwalifikowanych katalogach zasobów, w kodzie nadal będziemy się do niego odwoływać za pomocą wyrażenia R.layout.main. Android sam zajmuje się odnalezieniem właściwego pliku *main.xml*.

## Usuwanie błędów i optymalizacja układów graficznych za pomocą narzędzia Hierarchy Viewer

W zestawie Android SDK znalazło się wiele narzędzi, które znacznie ułatwiają życie projektanta aplikacji. Ponieważ zajmujemy się tematem tworzenia interfejsów użytkownika, warto zapoznać się z narzędziem Hierarchy Viewer. Narzędzie to, pokazane na rysunku 6.26, pozwala na usuwanie błędów w interfejsach UI z poziomu układu graficznego.

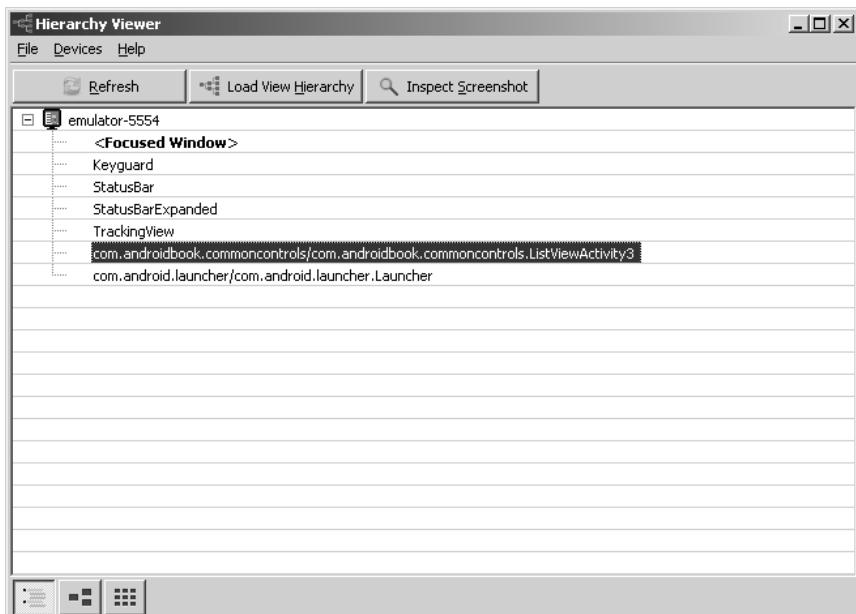


Rysunek 6.26. Widok układu graficznego w aplikacji Hierarchy Viewer

Jak widać na rysunku 6.26, narzędzie to ukazuje hierarchię widoków w formie drzewa. Koncepcja działania narzędzia jest następującą: najpierw narzędzie wczytuje układ graficzny, a następnie go analizuje pod względem określenia możliwych problemów lub próby zoptymalizowania układu graficznego pod kątem minimalizacji liczby widoków (kwestia wydajności).

W celu wyszukania błędów w interfejsie użytkownika należy uruchomić emulator i wyszukać interfejs UI, który zostanie sprawdzony. Następnie trzeba odnaleźć narzędzie Hierarchy Viewer w katalogu `/tools` środowiska Android SDK. W przypadku systemu Windows będzie się tam znajdował plik wsadowy `hierarchyviewer.bat`. Po jego uruchomieniu zostanie wyświetlony ekran urządzeń (rysunek 6.27).

Okno *Devices* wyświetla listę uruchomionych urządzeń (w naszym przypadku emulatorów). Po rozwinięciu węzła urządzenia w prawym panelu ukaże się lista ekranów dostępnych w tym urządzeniu. Żeby zobaczyć hierarchię widoków dla danego ekranu, należy go zaznaczyć (przeważnie jest to pełna nazwa aktywności, w której przedrostkiem jest nazwa pakietu aplikacji), a następnie kliknąć przycisk *Load View Hierarchy*.

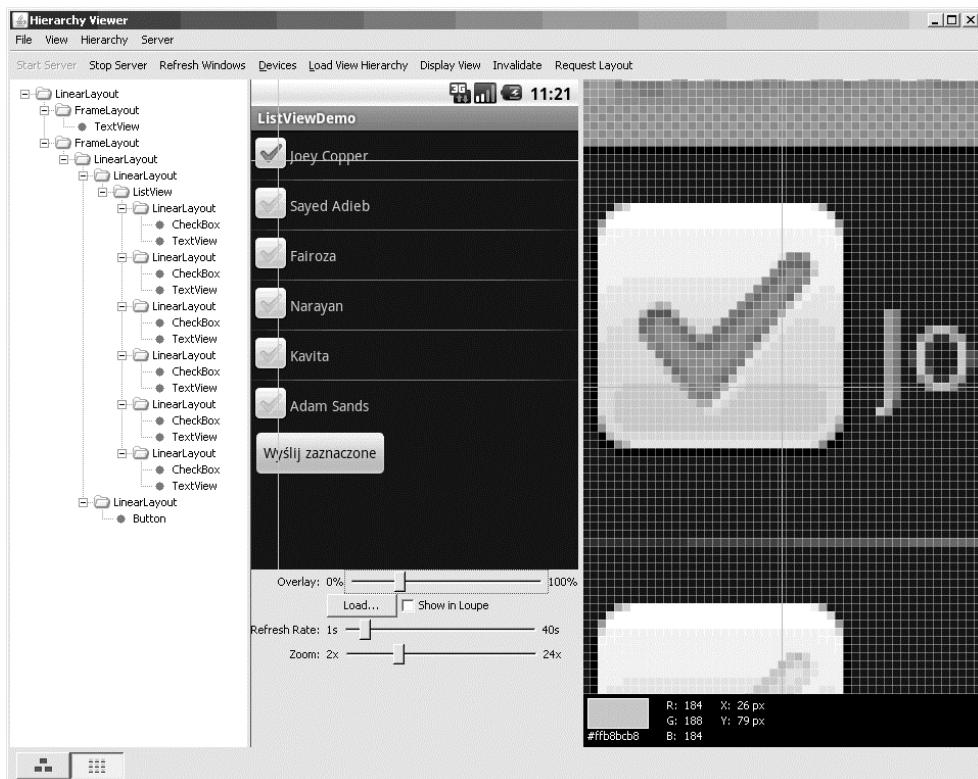


Rysunek 6.27. Ekran urządzeń w aplikacji Hierarchy Viewer

W oknie *View Hierarchy* zostanie wyświetlona hierarchia widoków w panelu po lewej stronie (rysunek 6.26). Po zaznaczeniu elementu widoku w środkowym panelu zostaną wyświetcone jego właściwości, a w obrazie szkieletowym po prawej stronie także relatywna wobec innych widoków lokalizacja tego widoku. Zaznaczony widok będzie podświetlony na czerwono. Mając w ten sposób ukazaną hierarchię widoków, możemy poszukać sposobu na zmniejszenie ich liczby, co jest równoznaczne z przyspieszeniem działania aplikacji.

Na rysunku 6.27 można dostrzec trzy przyciski w lewym dolnym rogu okna *Hierarchy Viewer*. Przycisk po lewej włącza omówiony powyżej widok drzewa. Środkowy przycisk odpowiada za wyświetlanie okna *View Hierarchy*. Przycisk po prawej wyświetla bieżący układ graficzny w widoku *Pixel Perfect*, jednak dopiero po zainicjalizowaniu tego widoku za pomocą przycisku *Inspect Screenshot*, widocznego w górnej części narzędzia. Widok ten jest bardzo interesujący, ponieważ przedstawia układ graficzny w siatce pikselowej (rysunek 6.28). Znajduje się tu kilka ciekawych elementów. Po lewej stronie jest umieszczony widok eksploratora wszystkich składników ekranu. Po kliknięciu jednego z umieszczonych tu elementów zostanie on podświetlony na czerwono w środkowym panelu. Znajdujące się w nim celowniki pozwalają wyświetlić w prawym panelu przybliżenie wybranego fragmentu ekranu (jest to **lupa**). Dostępna jest również opcja powiększenia, pozwalająca na jeszcze większe przybliżenie danego rejonu ekranu. Lupa pokazuje również dokładne współrzędne (x, y) wybranego piksela oraz wartość jego koloru.

Ostatnią bardzo interesującą funkcją w tym oknie jest przycisk *Load Overlay* oraz suwak *Overlay*. Istnieje możliwość załadowania pliku obrazu pod wyświetlonym ekranem w celu porównania go z tym obrazem (być może stanowią makietę dla tworzonego układu graficznego) oraz zwiększania lub zmniejszania jego widoczności za pomocą suwaka *Overlay*. Obraz ten pojawi się w lewym dolnym rogu. Domyślnie nie jest on pokazywany w widoku lupy, jednak można to zmienić poprzez zaznaczenie odpowiedniej opcji.



Rysunek 6.28. Tryb Pixel Perfect narzędzia Hierarchy Viewer

Wraz z wydaniem wersji 2.3 Androida aplikacja Hierarchy Viewer stała się dostępna w środowisku Eclipse. Pojawiły się w niej nowe perspektywy: *Hierarchy Viewer* oraz *Pixel Perfect*, każda posiadająca zestaw widoków rozwijających ich funkcje. Aplikacja ta działa w zasadzie tak samo jak jej samodzielna wersja, omówiona powyżej. Osoby mające problem z jej zainstalowaniem znajdą instrukcję, jak ją odszukać i wdrożyć, w rozdziale 2.

Dzięki takim narzędziom programista posiada wszechstronną kontrolę nad wyglądem oraz działaniem aplikacji.

## Odbońniki

Poniżej prezentujemy odnośniki do zagadnień, z którymi warto się dokładniej zapoznać.

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów stworzonych na potrzeby książki. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R06\_Kontrolki*. Zebrane są w nim wszystkie projekty z niniejszego rozdziału, rozmiieszczone w oddzielnich katalogach. Umieściliśmy w nim również plik *Czytaj.TXT*, w którym zamieściliśmy dokładną instrukcję importowania projektów z plików ZIP do środowiska Eclipse.

- [http://developer.android.com/reference/android/widget/LinearLayout.html#attr\\_android:id:gravity](http://developer.android.com/reference/android/widget/LinearLayout.html#attr_android:id:gravity) — na tej stronie zostały omówione różnorodne wartości parametru `gravity`, stosowanego wraz z układem graficznym `LinearLayout`.
- [www.curious-creature.org/2010/08/15/scrollviews-handy-trick](http://www.curious-creature.org/2010/08/15/scrollviews-handy-trick) — wpis Romain Guya (z zespołu Androida) wyjaśniający, jak należy korzystać z kontrolki `ScrollView`.
- <http://developer.android.com/resources/articles/index.html> — na tej stronie zamieszczono kilka technicznych artykułów pod wspólną nazwą *Layout Tricks*, których przeczytanie bardzo polecamy. Zajmują się one zagadnieniem wydajnością podczas projektowania i programowania interfejsów użytkownika w Androidzie. Warto również przejrzeć pozostałe artykuły dotyczące budowania interfejsów użytkownika.

## Podsumowanie

W tym momencie Czytelnik powinien mieć już dobre pojęcie na temat kontrolek dostępnych w zestawie SDK. Nieobce powinny być także adaptery oraz menedżery układu graficznego. Znając wymagania określonego typu ekranu, szybkie identyfikowanie kontrolek oraz menedżerów układów graficznych potrzebnych do skonstruowania wyświetlanego ekranu nie powinno być teraz trudne.

W następnym rozdziale będziemy się dalej zajmować interfejsem użytkownika — naszym celem będą menu.



# Praca z menu

Zestaw Android SDK pozwala na dokładną obsługę menu. W tym rozdziale wyjaśnimy, jak korzystać z kilku rodzajów menu obsługiwanych przez Androida: standardowych menu, podmenu, menu kontekstowych, menu w postaci ikon, menu drugorzędnych oraz menu alternatywnych. W wersji 3.0 Androida został wprowadzony pasek zadań, który znakomicie integruje się z listami menu. Pasek zadań oraz jego interakcja z menu zostały omówione w rozdziale 30.

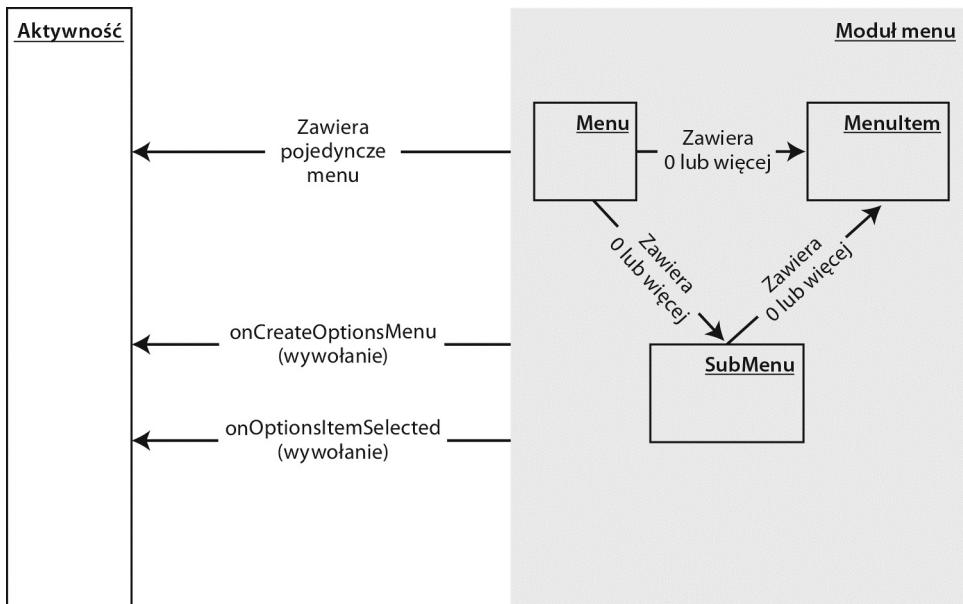
Menu, oprócz tego, że są obiektami środowiska Java, w Androidzie są traktowane jako zasoby. Podobnie jak w przypadku pozostałych rodzajów zasobów, można je wczytywać z plików XML. Dla każdego wczytanego elementu menu Android tworzy identyfikator zasobów. W tym rozdziale szczegółowo zajmujemy się takimi zasobami menu w formie XML. Pokażemy także, w jaki sposób wykorzystywać automatycznie wygenerowane identyfikatory zasobów dla każdego rodzaju elementów menu.

## Menu w Androidzie

Osoby pracujące w takim środowisku jak Swing obsługiwane przez język Java, Windows Presentation Foundation (WPF) w systemie Windows lub w jakimkolwiek innym szkielecie interfejsu użytkownika, z pewnością zetknęły się z menu.

Najważniejszą klasą obsługującą menu w Androidzie jest klasa `android.view.Menu`. Każda aktywność w Androidzie jest powiązana z tego typu obiektem menu, w którym można zawrzeć wiele elementów menu oraz podmenu.

Elementy menu są reprezentowane przez klasę `android.view.MenuItem`, a podmenu przez `android.view.SubMenu`. Związki pomiędzy nimi zostały naszkicowane na rysunku 7.1. Ścisłe rzecz biorąc, nie jest to diagram klas, lecz diagram strukturalny, zaprojektowany po to, aby pomóc w dostrzeżeniu powiązań pomiędzy różnymi klasami i funkcjami dotyczącymi menu.



**Rysunek 7.1.** Struktura klas związanych z menu w Androidzie

Na rysunku 7.1 widać, że klasa `Menu` składa się z podzbioru elementów.

Element menu posiada także nazwę (tytuł), swój niepowtarzalny identyfikator oraz identyfikator kolejności na liście (w zestawie SDK nosi on po prostu nazwę kolejności — ang. *order*), a także identyfikator (lub numer). Takie identyfikatory kolejności służą do uporządkowania elementów wewnętrz menu. Jeśli na przykład jeden z elementów posiada wartość identyfikatora kolejności równą 4, a inny element ma przyporządkowaną wartość 6, pierwszy z obiektów będzie widniał ponad drugim.

Pewne zakresy wartości są zarezerwowane dla określonych rodzajów menu. Kolejność elementów menu drugorzędnych, które są uznawane za mniej istotne od pozostałych, rozpoczyna się od wartości `0x30000` i jest definiowana przez stałą `Menu.CATEGORY_SECONDARY`. Inne rodzaje kategorii menu — na przykład menu systemowe, menu alternatywne czy menu kontenerowe — posiadają odmienne zakresy wartości kolejności.

Wartości elementów menu systemowego rozpoczynają się od `0x20000` i są definiowane przez stałą `Menu.CATEGORY_SYSTEM`. Wartości elementów menu alternatywnego rozpoczynają się od `0x40000` i są definiowane przez stałą `Menu.CATEGORY_ALTERNATIVE`. Wartości elementów menu kontenerowego rozpoczynają się od `0x10000` i są definiowane przez stałą `Menu.CATEGORY_CONTAINER`. Przyglądając się zakresom wartości tych stałych, można stwierdzić, w jakiej kolejności elementy menu pojawiają się na ekranie (rodzaje elementów menu omówimy w podrozdziale „Praca z innymi rodzajami menu”).

Możemy grupować elementy menu poprzez przydzielenie każdemu z nich identyfikatora grupy, który jest jednym z atrybutów takiego elementu. Wszystkie obiekty posiadające ten sam identyfikator grupy są uznawane za elementy jednego zbioru.

Na rysunku 7.1 pokazano również dwie wywoływane metody, dzięki którym można tworzyć i obsługiwać elementy menu: `onCreateOptionsMenu` oraz `onOptionsItemSelected`. Zajmiemy się nimi nieco dalej.

## Tworzenie menu

W środowisku Android SDK nie ma potrzeby tworzenia obiektu menu od podstaw. Ponieważ aktywność jest powiązana z pojedynczym menu, jest ono tworzone dla tej aktywności i przekazywane do metody wywoławczej `onCreateOptionsMenu` klasy tej aktywności (jak wskazuje nazwa metody, menu w Androidzie zwane są także **menu opcji**). Metoda ta umożliwia zapelnienie przekazywanego jej menu zestawem elementów menu (listing 7.1).

**Listing 7.1.** Sygnatura metody `onCreateOptionsMenu`

---

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // zapełnia elementami menu
    ...
    ...return true;
}
```

---

Po zapelnieniu menu elementami metoda powinna odesłać wartość `true`, co oznacza, że menu stało się widoczne. Jeżeli przekazaną wartością będzie `false`, menu będzie niewidoczne. Kod przedstawiony na listingu 7.2 pokazuje, w jaki sposób dodać trzy elementy do menu za pomocą identyfikatora grupy oraz identyfikatorów elementów menu i identyfikatora kolejności o wartościach wzrastających.

**Listing 7.2.** Dodawanie elementów do menu

---

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // wywołuje klasę bazową, zawierającą listy menu systemowych
    super.onCreateOptionsMenu(menu);

    menu.add(0      // Grupa
            ,1      // identyfikator elementu
            ,0      // kolejność
            ,"Dodaj"); // tytuł

    menu.add(0,2,1,"element2");
    menu.add(0,3,2,"Wyczysć");

    // Ważne jest, żeby została zwrócona wartość true, co spowoduje wyświetlenie menu
    return true;
}
```

---

Należy także wywołać implementację klasy podstawowej tej metody, aby system miał możliwość zapelnienia menu elementami menu systemowego. Źeby elementy menu systemowego były oddzielone od pozostałych elementów, Android dodaje je, począwszy od wartości kolejności równej `0x20000` (wspomnieliśmy wcześniej, że stała `Menu.CATEGORY_SYSTEM` definiuje identyfikator kolejności dla tego typu elementów. Jak na razie w żadnej wersji Androida nie zostały dodane nowe menu systemowe).

Pierwszym parametrem wymaganyem do dodania elementu jest identyfikator grupy (liczba całkowita). Drugim jest identyfikator elementu menu, odsyłany do wywoywanej funkcji po wybraniu tego elementu. Trzeci argument reprezentuje identyfikator kolejności.

Ostatnim argumentem jest nazwa lub tytuł elementu menu. Można skorzystać z zasobu typu `string` umieszczonego w pliku stałych `R.java`, zamiast wpisywać tekst. Identyfikatory grupy, elementu oraz kolejności są całkowicie opcjonalne; jeżeli nie ma potrzeby ich definiowania, można wprowadzić argument `Menu.NONE`.

## Praca z grupami menu

Pokażemy teraz, w jaki sposób można pracować z grupami menu. Listing 7.3 przedstawia sposób dodania dwóch grup menu: Grupy 1 oraz Grupy 2.

---

**Listing 7.3.** Zastosowanie identyfikatorów grup do utworzenia grup menu

---

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // Grupa 1
    int group1 = 1;
    menu.add(group1,1,1,"g1.item1");
    menu.add(group1,2,2,"g1.item2");

    // Grupa 2
    int group2 = 2;
    menu.add(group2,3,3,"g2.item1");
    menu.add(group2,4,4,"g2.item2");

    return true; // ważne, żeby została zwrócona wartość true
}
```

---

Zwróćmy uwagę, że identyfikatory elementów oraz kolejności są niezależne dla każdej grupy. Jaki jest więc pożytek z grupy? W klasie `android.view.Menu` dostępny jest zbiór metod, korzystających z identyfikatorów grupy. Za ich pomocą można kontrolować elementy menu w danej grupie:

```
removeGroup(id)
setGroupCheckable(id, checkable, exclusive)
setGroupEnabled(id,boolean enabled)
setGroupVisible(id,visible)
```

Metoda `removeGroup` usuwa wszystkie elementy z grupy o podanym identyfikatorze. Można włączać lub wyłączać elementy menu w danej grupie za pomocą metody `setGroupEnabled`. W podobny sposób, stosując metodę `setGroupVisible`, kontrolujemy widoczność grupy elementów menu.

Metoda `setGroupCheckable` jest dosyć interesująca. Dzięki niej pojawią się znak zaznaczenia obok elementu wybranego przez użytkownika. Jeżeli metoda ta zostanie zastosowana wobec całej grupy, wszystkie jej elementy uzyskają tę właściwość. W przypadku ustanowienia w tej metodzie flagi wyłączonej możliwe stanie się zaznaczenie wyłącznie jednego elementu w grupie. Pozostałe elementy grupy będą niezaznaczone.

Wiemy już, w jaki sposób zapłnić główne menu aktywności elementami oraz pogrupować je zgodnie z ich przeznaczeniem. Teraz pokażemy, jak ustawić reakcję systemu na wybór elementu menu.

## Odpowiedź na wybór elementów menu

Istnieje wiele sposobów określenia reakcji na kliknięcie elementu menu w Androidzie. Można wykorzystać metodę `onOptionsItemSelected` klasy aktywności, wprowadzić samodzielne obiekty nasłuchujące lub zastosować intencje. W kolejnych podpunktach omówimy każdą z wymienionych metod.

### Odpowiedź na kliknięcie za pomocą metody `onOptionsItemSelected`

Po kliknięciu elementu menu Android wywołuje metodę `onOptionsItemSelected` w klasie `Activity` (listing 7.4).

**Listing 7.4.** Sygnatura oraz treść metody `onOptionsItemSelected`

---

```
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
    switch(item.getItemId()) {
        ...
    }
    // dla elementów klikniętych
    return true;

    // dla pozostałych elementów
    ...return super.onOptionsItemSelected(item);
}
```

---

Podstawą działającego tu algorytmu jest sprawdzenie identyfikatora elementu menu poprzez metodę `getItemId()` klasy `MenuItem` oraz wykonanie odpowiedniej czynności. Jeżeli metoda `onOptionsItemSelected()` przetwarza element menu, przekazuje wartość `true`. Zdarzenie z menu nie będzie przenoszone dalej. Jeżeli wywołań elementu menu nie obsłuży metoda `onOptionsItemSelected()`, wywołuje ona metodę nadziedną poprzez `super.onOptionsItemSelectedSelected`. Domyślna implementacja metody `onOptionsItemSelected()` przekazuje wartość `false`, co powoduje „zwykłe” przetwarzanie zdarzenia. Taka forma przetwarzania obejmuje alternatywne metody wywołania odpowiedzi na kliknięcie elementu menu.

### Odpowiedź na kliknięcie za pomocą obiektów nasłuchujących

Odpowiedzi na kliknięcie definiuje się przeważnie za pomocą przesłonięcia metody `onOptionSelected`; jest to technika zalecana z powodu poprawienia wydajności. Jednak element menu pozwala również na zarejestrowanie obiektu nasłuchującego, zapewniającego wywołania zwrotne.

Jest to proces dwuetapowy. Pierwszy etap polega na zimplementowaniu interfejsu `OnMenuItemClickListener`. Następnie przekazuje się wystąpienie tej implementacji do elementu menu. Po kliknięciu elementu zostanie wywołana metoda `onMenuItemClick()` interfejsu `OnMenuItemClickListener` (listing 7.5).

**Listing 7.5.** Zastosowanie obiektu nasłuchującego jako wywołania zwrotnego w przypadku kliknięcia elementu menu

---

```
// Etap 1
public class MyResponse implements OnMenuItemClickListener
{
    // Jakaś zmienna lokalna, na której można pracować
    //...
    // Jakiś konstruktory
    @Override
    boolean onMenuItemClick(MenuItem item)
    {
        // Wykonuje zadanie
        return true;
    }
}

// Etap 2
MyResponse myResponse = new MyResponse(...);
menuItem.setOnMenuItemClickListener(myResponse);
...
```

---

Metoda `onMenuItemClick` zostaje wywołana po wyświetleniu elementu menu. Kod zostaje wykonany natychmiast po kliknięciu elementu, jeszcze przed wywołaniem metody `onOptionsItemSelected`. Jeżeli metoda `onMenuItemClick` przekaże wartość `true`, nie zostaną wykonane następne wywołania zwrotne — w tym także metoda `onOptionsItemSelected`. Oznacza to, że kod obiektu nasłuchującego ma pierwszeństwo przed metodą `onOptionsItemSelected`.

## Wykorzystanie intencji do wywołania odpowiedzi na kliknięcie elementu menu

Istnieje także możliwość powiązania elementu menu z intencją poprzez wykorzystanie metody `setIntent(intent)` klasy `MenuItem`. Domyślnie element menu nie jest powiązany z żadną intencją. Jednak jeśli intencja jest powiązana z takim elementem i nic innego go nie przetwarza, to domyślnym zachowaniem staje się przywołanie intencji za pomocą metody `startActivity` (`intent`). Jeśli ten sposób ma zadziałać, wszystkie procedury obsługujące — zwłaszcza metoda `onOptionsItemSelected` — powinny wywoływać nadzczną klasę metody `onOptionsItemSelected` dla elementów nieprzetwarzanych. Ewentualnie można na ten sposób spojrzeć następująco: system daje metodzie `onOptionsItemSelected` pierwszeństwo w przetworzeniu elementu listy menu (nie licząc oczywiście obiektu nasłuchującego). Zakładamy tutaj, że żaden obiekt nasłuchujący nie został bezpośrednio powiązany z tym elementem menu, ale jeśli jest inaczej, obiekt nasłuchujący przesłoni całą resztę.

Jeżeli metoda `onOptionsItemSelected` nie zostanie przesłonięta, to podstawowa klasa szkieletu Androida wykona czynności potrzebne do przywołania intencji wobec elementu menu. Jeżeli jednak metoda ta zostanie przesłonięta, ale nie interesuje nas dany element menu, należy wywołać metodę nadzczną, która z kolei zajmie się wywołaniem intencji. Podsumowując: albo nie należy przesłaniać metody `onOptionsItemSelected`, albo należy ją przesłonić i przywołać metodę nadzczną dla elementów menu, które nie są przetwarzane.

## Utworzenie środowiska testowego do sprawdzania menu

Jak na razie wszystko jest proste i zrozumiałe. Czytelnicy nauczyli się, w jaki sposób tworzyć menu oraz jak definiować dla nich odpowiedzi za pomocą różnych rodzajów wywołań. Teraz pokażemy przykładową aktywność testującą interfejsy API, które do tej pory przedstawiliśmy.

### Uwaga!

Na końcu rozdziału zamieściliśmy adres URL, z którego można pobrać odpowiedni projekt, gotowy do zimportowania w środowisku Eclipse.

Celem tego ćwiczenia jest utworzenie prostej aktywności, w której znajdzie się widok tekstowy. Widok ten będzie pełnił rolę testera. Przy każdym menu będziemy wypisywać nazwę oraz identyfikator elementu menu w tym widoku tekstowym. Efekt końcowy będzie wyglądał tak jak na rysunku 7.2.



Rysunek 7.2. Przykładowa aplikacja menu

Na rysunku 7.2 widoczne są dwa interesujące nas elementy: menu oraz widok tekstowy. Menu pojawia się u spodu ekranu. Nie będzie jednak widoczne po uruchomieniu aplikacji; konieczne będzie kliknięcie przycisku *Menu* na emulatorze lub urządzeniu, żeby menu zostało wyświetcone. Drugim zajmującym nas elementem jest widok tekstu na górze ekranu, w którym wyświetlone są wiadomości dotyczące błędów. Podczas klikania elementów dostępnych w menu ich nazwy będą wyświetlane w widoku tekstowym. Po kliknięciu elementu *Wyczysć* program usunie zawartość widoku tekstowego.

### Uwaga!

Na rysunku 7.2 nie jest przedstawiony początkowy stan aplikacji. Stanowi on ilustrację omawianych typów menu w tym rozdziale.

Żeby zaimplementować środowisko testowe, należy wykonać następujące czynności:

1. Utwórz plik XML układu graficznego, w którym zostanie umieszczony widok tekstowy.
2. Utwórz klasę Activity, która będzie przechowywać układ graficzny zdefiniowany w punkcie 1.
3. Skonfiguruj menu.
4. Dodaj standardowe elementy do menu.
5. Dodaj elementy drugorzędne do menu.
6. Utwórz odpowiedzi na kliknięcie dla tych elementów.
7. Zmodyfikuj plik *AndroidManifest.xml* w taki sposób, żeby była wyświetlana właściwa nazwa aplikacji.

Każdy z tych etapów zostanie teraz szczegółowo omówiony. Zaprezentujemy również kod potrzebny do utworzenia środowiska testowego.

## Utworzenie układu graficznego w pliku XML

Pierwszy etap polega na utworzeniu prostego pliku XML układu graficznego, zawierającego widok tekstowy (listing 7.6). Plik ten może być wczytywany do aktywności podczas jej uruchamiania.

**Listing 7.6.** Plik XML układu graficznego, zastosowany w środowisku testowym

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView android:id="@+id/textViewId"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Notatnik testowy"
    />
</LinearLayout>
```

---

## Utworzenie aktywności

W drugim etapie tworzymy aktywność, co jest również całkiem proste. Założyszy, że plik układu graficznego z pierwszego etapu jest dostępny w katalogu */res/layout/main.xml*, można skorzystać z jego identyfikatora zasobów do zapełnienia widoków aktywności (listing 7.7).

**Listing 7.7.** Klasa aktywności menu w środowisku testowym

---

```
public class SampleMenusActivity extends Activity {

    // Należy to zainicjalizować w metodzie onCreateOptionsMenu
    Menu myMenu = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

---

```

        setContentView(R.layout.main);
    }
}

```

---

W celu zachowania przejrzystości nie zawarliśmy tutaj instrukcji importu. W środowisku Eclipse można to zrobić automatycznie poprzez wybranie opcji *Source/Organize Imports* w menu kontekstowym edytora. Równie skuteczny okazuje się skrót *Ctrl+Shift+O*.

## Konfiguracja menu

Gdy już mamy widok oraz aktywność, możemy przejść do trzeciego etapu: przesłonięcia metody `onCreateOptionsMenu` i skonfigurowania menu za pomocą kodu (listing 7.8).

**Listing 7.8.** Konfigurowanie menu za pomocą kodu

---

```

@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // należy wywołać metodę nadziedną w celu dołączenia menu systemowych
    super.onCreateOptionsMenu(menu);

    this.myMenu = menu;

    // dodaje kilka standardowych menu
    addRegularMenuItem(menu);

    // dodaje kilka drugorzędnych menu
    add5SecondaryMenuItem(menu);

    // W celu uwidocznienia menu musi zostać zwrócona wartość true
    // Przy wartości false menu nie będzie widoczne
    return true;
}

```

---

Kod z listingu 7.8 wywołuje najpierw nadziedną metodę `onCreateOptionsMenu`, dzięki czemu uzyskuje ona możliwość dodania systemowych menu.

**Uwaga!**

W dotychczasowych wersjach środowiska Android SDK metoda `onCreateOptionsMenu` nie dodaje nowych elementów menu. Jednak w kolejnych edycjach może się to zmienić, więc warto wywoływać metodę nadziedną.

Programujemy następnie obiekt Menu, gdyż będzie on później modyfikowany w celach demonstracyjnych. Następnym etapem jest dodanie kilku standardowych elementów menu oraz kilku elementów drugorzędnych.

## Dodawanie elementów menu standardowego

Nadszedł czas na etap czwarty: dodanie kilku standardowych elementów do menu. Kod funkcji `addRegularMenuItem` przedstawiono na listingu 7.9.

**Listing 7.9.** Funkcja addRegularMenuItems

```
private void addRegularMenuItems(Menu menu)
{
    int base=Menu.FIRST; // wartość wynosi 1

    menu.add(base,base,base,"Dodaj");
    menu.add(base,base+1,base+1,"element2");
    menu.add(base,base+2,base+2,"Wyczysć");

    menu.add(base,base+3,base+3,"ukryj drugorzędny");
    menu.add(base,base+4,base+4,"pokaż drugorzędny");

    menu.add(base,base+5,base+5,"włącz drugorzędny");
    menu.add(base,base+6,base+6,"wyłącz drugorzędny");

    menu.add(base,base+7,base+7,"zaznacz drugorzędny");
    menu.add(base,base+8,base+8,"odznacz drugorzędny");
}
```

---

Klasa Menu definiuje kilka przydatnych stałych, wśród nich stałą Menu.FIRST. Można ją wykorzystać jako podstawę numeracji identyfikatorów menu oraz innych sekwencji liczbowych związanych z menu. Zauważmy, w jaki sposób możemy powiązać identyfikator grupy z wartością base i zwiększać jedynie wartości identyfikatorów kolejności oraz identyfikatorów poszczególnych elementów. Dodatkowo w kodzie w celach demonstracyjnych umieszczone kilka niestandardowych elementów menu, takich jak „ukryj drugorzędny”, „włącz drugorzędny” i parę innych.

## Dodawanie elementów menu drugorzędnego

Dodajmy teraz kilka elementów menu drugorzędnego, aby wykonać piąty etap (listing 7.10). Jak wcześniej wspomnieliśmy, kolejność elementów menu drugorzędnego rozpoczyna się od wartości 0x30000 i jest zdefiniowana przez stałą Menu.CATEGORY\_SECONDARY. Ponieważ wartość ta jest wyższa od analogicznej wartości elementów menu standardowego, elementy te będą się znajdować poniżej elementów standardowych w menu. Zauważmy, że jedynie kolejność rozmieszczenia odróżnia elementy standardowe od drugorzędnych. We wszystkich innych aspektach elementy te niczym się od siebie nie różnią.

**Listing 7.10.** Dodawanie elementów menu drugorzędnego

```
private void add5SecondaryMenuItems(Menu menu)
{
    // Elementy drugorzędne są wyświetlane tak samo jak inne elementy
    int base=Menu.CATEGORY_SECONDARY;

    menu.add(base,base+1,base+1,"drugorz. element 1");
    menu.add(base,base+2,base+2,"drugorz. element 2");
    menu.add(base,base+3,base+3,"drugorz. element 3");
    menu.add(base,base+3,base+3,"drugorz. element 4");
    menu.add(base,base+4,base+4,"drugorz. element 5");
}
```

---

## Odpowiedź na kliknięcie elementu menu

Po skonfigurowaniu menu przechodzimy do szóstego etapu: przypisywania odpowiedzi na kliknięcie. Po kliknięciu elementu Android wywołuje metodę `onOptionsItemSelected` klasy `Activity` poprzez przesłanie odniesienia do tego klikniętego elementu. Następnie metoda `getItemId()` klasy `MenuItem` rozpoznaje wybrany element.

Nierazdrok stosuje się instrukcje `switch` lub kombinacje `if` oraz `else`, których celem jest wywołanie różnych funkcji w odpowiedzi na kliknięcie elementu. Listing 7.11 przedstawia standardowy proces odpowiadania na kliknięcie za pomocą wywoływanej metody `onOptionsItemSelected` (nieco lepszy sposób wykonania tej samej czynności został zaprezentowany w podrozdziale „Wczytywanie menu poprzez pliki XML”, gdzie będą stosowane symboliczne nazwy dla identyfikatorów elementów menu).

**Listing 7.11.** Odpowiedź na kliknięcie elementu menu

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == 1) {
        appendText("\nwitaj");
    }
    else if (item.getItemId() == 2) {
        appendText("\nlement2");
    }
    else if (item.getItemId() == 3) {
        emptyText();
    }
    else if (item.getItemId() == 4) {
        // ukryj drugorzędny
        this.appendMenuItemText(item);
        this.myMenu.setGroupVisible(Menu.CATEGORY_SECONDARY, false);
    }
    else if (item.getItemId() == 5) {
        // pokaż drugorzędny
        this.appendMenuItemText(item);
        this.myMenu.setGroupVisible(Menu.CATEGORY_SECONDARY, true);
    }
    else if (item.getItemId() == 6) {
        // włącz drugorzędny
        this.appendMenuItemText(item);
        this.myMenu.setGroupEnabled(Menu.CATEGORY_SECONDARY, true);
    }
    else if (item.getItemId() == 7) {
        // wyłącz drugorzędny
        this.appendMenuItemText(item);
        this.myMenu.setGroupEnabled(Menu.CATEGORY_SECONDARY, false);
    }
    else if (item.getItemId() == 8) {
        // zaznacz drugorzędny
        this.appendMenuItemText(item);
        myMenu.setGroupCheckable(Menu.CATEGORY_SECONDARY, true, false);
    }
    else if (item.getItemId() == 9) {
        // usuń zaznaczenie drugorzędnego
        this.appendMenuItemText(item);
        myMenu.setGroupCheckable(Menu.CATEGORY_SECONDARY, false, false);
    }
}
```

```
        }
    else {
        this.appendMenuItemText(item);
    }
// powinien zwrócić wartość true, jeśli element jest przetwarzany
return true;
}
```

---

Kod przedstawiony na listingu 7.11 prezentuje również przeprowadzanie operacji na poziomie grupy menu; wywołania tych metod zostały zaznaczone pogrubieniem. Szczegóły klikniętego elementu są wyświetlane w widoku `TextView`. Na listingu 7.12 zostały wypisane funkcje pozwalające umieszczać te informacje w kontrolce `TextView`. Zwróćmy uwagę na dodatkową metodę klasy `MenuItem`, pozwalającą wyświetlić tytuł elementu.

**Listing 7.12.** Funkcje umożliwiające wypisywanie danych w kontrolce testowej `TextView`

---

```
// Dany ciąg znaków tekstowych dodany do kontrolki TextView
private void appendText(String text) {
    TextView tv = (TextView)this.findViewById(R.id.textViewId);
    tv.setText(tv.getText() + text);
}

// Dany element menu wyświetla swój tytuł w kontrolce TextView
private void appendMenuItemText(MenuItem menuItem) {
    String title = menuItem.getTitle().toString();
    TextView tv = (TextView)this.findViewById(R.id.textViewId);
    tv.setText(tv.getText() + "\n" + title);
}
// Wyczyszczenie zawartości kontrolki TextView
private void emptyText() {
    TextView tv = (TextView)this.findViewById(R.id.textViewId);
    tv.setText("");
}
```

---

## Modyfikowanie pliku `AndroidManifest.xml`

Ostatnim etapem tworzenia środowiska testowego jest aktualizacja pliku `AndroidManifest.xml`. Ten generowany automatycznie podczas tworzenia nowego projektu plik jest umieszczony w katalogu głównym projektu.

Wewnątrz tego pliku jest przeprowadzany proces rejestrowania klasy `Activity` (na przykład `SampleMenusActivity`) oraz definiowany tytuł aktywności. Jak widać na rysunku 7.2, nasza aktywność nosi nazwę *Przykładowa aplikacja menu*. Wiersz odpowiedzialny za tytuł został na listingu 7.13 pogrubiony.

**Listing 7.13.** Plik `AndroidManifest.xml` przygotowany do testowania

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="your-package-name-goes-here "
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="Przykładowe menu">
```

---

```

<activity android:name=".SampleMenusActivity"
    android:label="Przykładowa aplikacja menu">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
</application>
</manifest>

```

---

Za pomocą umieszczonych w tym podrozdziale fragmentów kodu można w szybki sposób utworzyć środowisko testowe pozwalające na sprawdzenie możliwości menu. Pokazaliśmy, jak utworzyć prostą aktywność uruchamianą wraz z widokiem tekstowym, a także w jaki sposób utworzyć menu i przypisać jego elementom reakcję na kliknięcie. Większość menu jest zaprojektowana na podstawie tego prostego, lecz funkcjonalnego wzorca. Rysunek 7.2 pokazuje, jakiego rodzaju interfejsu UI należy się spodziewać po zakończeniu ćwiczenia. Przypominamy jednak, że efekt końcowy może się różnić od przedstawionego na rysunku (najczęściej jednak nie będzie się różnił), ponieważ nie zademonstrowaliśmy sposobu, w jaki należy dodawać ikony do menu. Nawet po wstawieniu ikon interfejs środowiska testowego może się nieznacznie różnić od naszej wersji, ponieważ Czytelnik może zastosować inne obrazy.

## Praca z innymi rodzajami menu

Do tej pory zajmowaliśmy się prostszymi, chociaż funkcjonalnymi rodzajami menu. W trakcie korzystania ze środowiska SDK można zauważyc, że Android obsługuje również menu w formie ikon, podmenu, menu kontekstowe oraz menu alternatywne. Ten ostatni typ występuje wyłącznie w Androidzie. W kolejnych punktach zostaną przedstawione niestandardowe rodzaje menu.

### Rozszerzone menu

Na rysunku 7.2 w prawym dolnym rogu ekranu widnieje element menu zatytułowany *Więcej*. W żadnym fragmencie kodu nie umieściliśmy tego elementu, zatem skąd on się tu wziął?

Jeżeli aplikacja posiada więcej elementów menu, niż może zaprezentować na ekranie, Android wyświetla w menu element *Więcej*, dzięki któremu użytkownik może przejść do pozostałych elementów. Takie **rozszerzone menu** pojawia się automatycznie, gdy trzeba wyświetlić zbyt dużą liczbę elementów na małej przestrzeni. Rozszerzone menu mają jednak ograniczenie: nie mogą przetwarzać ikon. Po kliknięciu przycisku *Więcej* zostanie wyświetcone menu bez ikon.

## Praca z menu w postaci ikon

Skoro już wspomnieliśmy o menu zawierających ikony, przyjrzyjmy się im uważniej. W repertuarze menu obsługiwany jest nie tylko tekst, lecz również obrazy lub ikony. Do reprezentowania elementów menu można używać samych ikon, jak i ikon wraz z tekstem. Podczas stosowania ikon pojawia się jednak kilka ograniczeń. Po pierwsze, nie ma możliwości stosowania ikon w rozszerzonych menu, jak już zostało wspomniane w poprzednim punkcie. Po drugie, elementy menu zawierające ikony nie obsługują funkcji ich zaznaczania. Po trzecie, jeżeli tekst w elemencie menu jest za długi, zostanie obcięty o pewną liczbę znaków w zależności od rozmiaru wyświetlacza (ograniczenie to dotyczy także elementów menu zawierających sam tekst).

Utworzenie elementu menu w formie ikony jest bardzo proste. Najpierw buduje się standardowy element menu, a następnie w celu wybrania obrazu stosuje się metodę `setIcon` w klasie `MenuItem`. Konieczne jest wprowadzenie identyfikatora zasobu obrazu, trzeba więc umieścić obraz lub ikonę w katalogu `/res/drawable`. Jeśli na przykład plik nosi nazwę `balony`, jego identyfikator będzie następujący: `R.drawable.balony`.

Poniżej umieszczono przykład:

```
// Dodaje element menu i zapamiętuje go, żeby następnie móc dodać do niego ikonę.  
MenuItem item8 = menu.add(base,base+8,base+8,"Oznacz drugorzędny");  
item8.setIcon(R.drawable.balony);
```

Podczas dodawania elementów do menu rzadko kiedy trzeba pilnować, żeby lokalna zmienna była przekazywana przez metodę `menu.add`. Jednak w tym przypadku otrzymany obiekt musi być zapamiętyany w celu dodania ikony do obiektu menu. Z powyższego przykładu widać także, że typem zwracanym przez metodę `menu.add` jest `MenuItem`.

Ikona będzie widoczna tak długo, jak długo wyświetlany będzie obiekt menu w głównym oknie aplikacji. Jeżeli obiekt ten będzie wyświetlany w rozszerzonym menu, ikona zostanie pominięta i widoczny stanie się sam tekst. Pokazany na rysunku 7.2 element menu, wyświetlający ikonę przedstawiającą balony, jest przykładem omówionego rodzaju obiektu.

## Praca z podmenu

Przyjrzyjmy się teraz podmenu w Androidzie. Na rysunku 7.1 zostały naszkicowane związki strukturalne pomiędzy obiektem klasy `SubMenu` a obiektami klas `Menu` i `MenuItem`. W obiekcie `Menu` może się znajdować wiele obiektów klasy `SubMenu`. Każdy obiekt `SubMenu` jest dodawany do obiektu `Menu` poprzez wywołanie metody `Menu.addSubMenu` (listing 7.14). Elementy są dodawane do listy podmenu tak samo jak w przypadku zwykłego menu. Wynika to z faktu, że obiekt `SubMenu` wywodzi się z obiektu klasy `Menu`. Nie można jednak umieszczać podmenu w obiekcie `SubMenu`.

### Listing 7.14. Dodawanie podmenu

---

```
private void addSubMenu(Menu menu)  
{  
    // Elementy drugorzędne są pokazywane tak jak pozostałe obiekty  
    int base=Menu.FIRST + 100;  
    SubMenu sm = menu.addSubMenu(base,base+1,Menu.NONE,"podmenu");  
    sm.add(base,base+2,base+2,"podelement1");  
    sm.add(base,base+3,base+3, "podelement2");  
    sm.add(base,base+4,base+4, "podelement3");  
  
    // Ikony elementów podmenu nie są obsługiwane  
    item1.setIcon(R.drawable.icon48x48_2);  
  
    // Ten poniżej jest poprawny, jednak...  
    sm.setIcon(R.drawable.icon48x48_1);  
  
    // ...spowoduje wyświetlenie wyjątku wykonawczego  
    //sm.addSubMenu("spróbuj tego");  
}
```

---

**Uwaga!**

Ponieważ obiekt SubMenu jest podklassą obiektu Menu, obsługuje metodę addSubMenu. Kompilator nie wyświetli błędu przy próbie dołączenia podmenu do innego podmenu, pojawi się jednak wyjątek wykonawczy.

W dokumentacji zestawu Android SDK widnieje także informacja, że podmenu nie obsługują ikon elementów menu. Gdy do elementu menu zostanie dodana ikona, a następnie zostanie on przeniesiony do podmenu, ikona ta zostanie zignorowana, nawet jeśli nie pojawił się żaden błąd komplikacji lub błęd wykonalowy. Jednak samo podmenu może posiadać ikonę.

## Zabezpieczanie menu systemowych

Większość aplikacji systemu Windows posiada takie menu, jak *Plik*, *Edycja*, *Widok*, *Otwórz*, *Zamknij* oraz *Wyjście*. Są to tak zwane menu systemowe. Zestaw Android SDK sugeruje wstawianie podobnego zestawu menu do każdego menu opcji. Jednak żadna z aktualnych wersji środowiska Android SDK nie dołącza tych menu w procesie tworzenia menu. Można sobie wyobrazić, że systemowe menu zostaną zaimplementowane w przyszłych wersjach środowiska programistycznego. Według dokumentacji programiści powinni w odpowiedni sposób przygotowywać kod, żeby można było wstawić menu systemowe, gdy będą dostępne. W tym celu należy wywołać metodę `onCreateOptionsMenu` w klasie nadrzędnej, dzięki czemu możliwe stanie się dodawanie menu systemowych do grupy definiowanej przez stałą `CATEGORY_SYSTEM`.

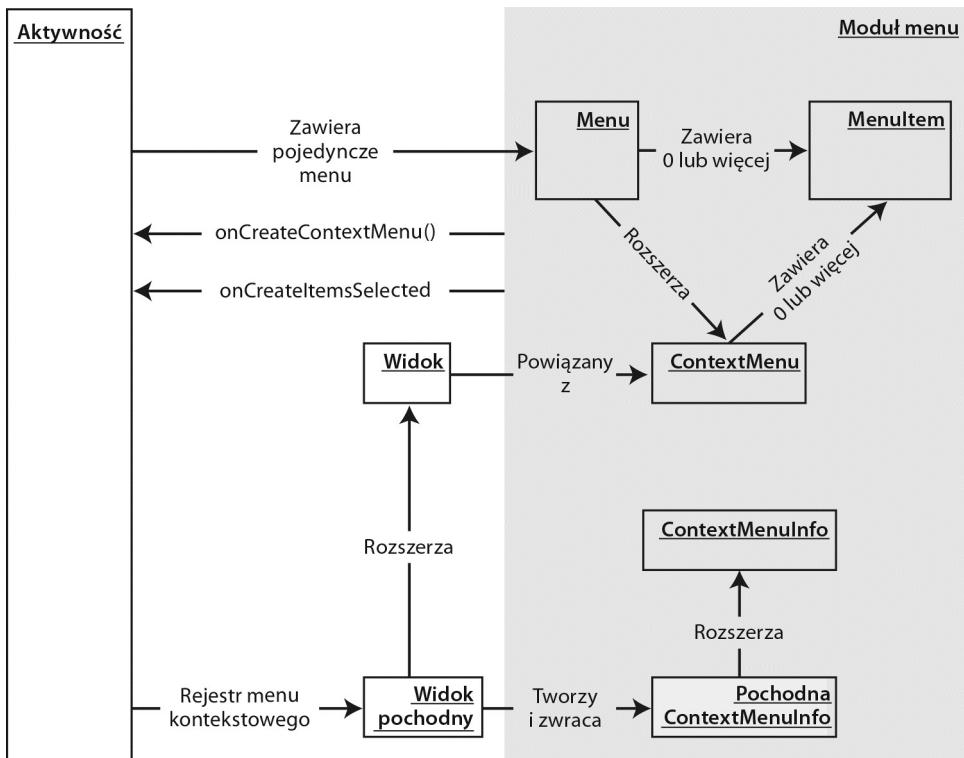
## Praca z menu kontekstowymi

Użytkownicy aplikacji biurowych z pewnością natknęli się na menu kontekstowe. Na przykład w programach systemu Windows dostęp do takiego menu uzyskuje się poprzez kliknięcie prawym przyciskiem dowolnego elementu interfejsu użytkownika. W Androidzie został zaadaptowany ten sam pomysł, korzystający z działania zwanego **długim kliknięciem**. W technice tej przycisk urządzenia wskazującego jest przytrzymywany nieco dłużej niż w przypadku zwykłego kliknięcia.

W takich handheldach jak telefony komórkowe kliknięcia myszy zostały zaimplementowane na najróżniejsze sposoby, w zależności od rodzaju sterowania. Jeżeli telefon jest zaopatrzony w kółko sterujące kursem, jego wcisnięcie jest odpowiednikiem kliknięcia myszy. W urządzeniach posiadających panel dotykowy jego naciśnięcie lub stuknięcie pełni rolę kliknięcia. Natomiast w przypadku urządzeń posiadających klawisze sterujące kursem oraz przycisk wykonania akcji naciśnięcie tego przycisku jest równoznaczne kliknięciu przyciskiem myszy. Niezależnie od sposobu zaimplementowania myszy w urządzeniu, dłuższe przytrzymanie elementu odpowiedzialnego za kliknięcie spowoduje wykonanie długiego kliknięcia.

Pod względem struktury menu kontekstowe różni się od omawianego wcześniej standardowego menu opcji (rysunek 7.3). Menu kontekstowe charakteryzują pewne niuanse, których brak w menu standardowym.

Na rysunku 7.3 widać, że menu kontekstowe reprezentowane jest przez klasę `ContextMenu` w architekturze menu Androida. Podobnie jak klasa `Menu`, tak i `ContextMenu` może obejmować wiele elementów menu. Aby dodać elementy do menu kontekstowego, wykorzystuje się te same metody klasy `Menu`. Największą różnicą pomiędzy klasami `Menu` a `ContextMenu` jest rodzaj obiektu będącego właściwicielem danego rodzaju menu. Standardowe menu przynależy do aktywności, podczas gdy menu kontekstowe — do widoku. Należało się tego spodziewać, ponieważ długie kliknięcia, uruchamiające menu kontekstowe, są stosowane wobec klikniętego



Rysunek 7.3. Aktywności, widoki i menu kontekstowe

widoku. Zatem aktywność może zawierać tylko jedno menu opcji, ale wiele menu kontekstowych. Ponieważ aktywność może obejmować wiele widoków, a każdy z nich może posiadać własne menu kontekstowe, maksymalna liczba menu kontekstowych w aktywności jest równa liczbie zawartych w niej widoków.

Chociaż właściwym menu kontekstowym jest widok, metoda potrzebna do zapełnienia takiego menu znajduje się w klasie `Activity`. Nosi ona nazwę `activity.onCreateContextMenu()` i z działania przypomina metodę `activity.onCreateOptionsMenu()`. Ta wywoływana metoda przenosi ze sobą również widok, w którym menu kontekstowe będzie zapełnione.

Istnieje jeszcze jeden godny uwagi problem z menu kontekstowymi. Chociaż metoda `onCreateOptionsMenu()` jest automatycznie wywoływana dla każdej aktywności, nie dotyczy to metody `onCreateContextMenu()`. Widok w aktywności *nie musi* posiadać menu kontekstowego. Na przykład mogą być obecne trzy widoki w aktywności, lecz może istnieć potrzeba wyłączenia menu kontekstowego tylko dla jednego z nich. Jeżeli dany widok ma posiadać menu kontekstowe, musi on zostać zarejestrowany wraz z aktywnością do pełnienia roli właściciela tego menu. Dokonuje się tego poprzez metodę `activity.registerForContextMenu(view)`, omówioną w podpunkcie „Rejestrowanie widoku dla menu kontekstowego”.

Zwróćmy teraz uwagę na przedstawioną na rysunku 7.3 klasę `ContextMenuInfo`. Obiekt tego typu jest przekazywany do metody `onCreateContextMenu`. Jest to jeden ze sposobów przekazywania przez widok dodatkowych informacji do tej metody. Żeby widok mógł tego dokonać, musi przesłonić metodę `getContextViewInfo()` i zwrócić pochodną klasę `ContextMenuInfo` wraz

z danymi reprezentującymi dodatkowe informacje. Żeby w pełni zrozumieć tę interakcję, można zjrzeć do kodu źródłowego klasy `android.view.View`.

**Uwaga!** Zgodnie z dokumentacją środowiska Android SDK menu kontekstowe nie obsługują skrótów, ikon ani podmenu.

Skoro znamy już ogólną strukturę menu kontekstowych, przyjrzyjmy się przykładowemu kodowi pokazującemu, w jaki sposób krok po kroku zaimplementować menu kontekstowe:

1. Zarejestruj widok dla danego menu kontekstowego w metodzie `onCreate()` aktywności.
2. Zapełnij menu kontekstowe za pomocą metody `onCreateContextMenu()`. Musisz dokończyć pierwszy etap, zanim ta metoda zostanie wywołana przez system Android.
3. Zdefiniuj odpowiedzi na kliknięcia poszczególnych elementów menu kontekstowego.

## Rejestrowanie widoku dla menu kontekstowego

Pierwszym etapem w implementacji menu kontekstowego jest zarejestrowanie widoku dla tego menu w metodzie `onCreate()` aktywności. Po utworzeniu środowiska testowego, omówionego we wcześniejszej części rozdziału, można zarejestrować widok `TextView` dla menu kontekstowego w tym środowisku za pomocą kodu widocznego na listingu 7.15. Najpierw należy znaleźć widok `TextView`, a następnie wywołać w aktywności metodę `registerForContextMenu`, jako argument wstawiając ten widok `TextView`. W ten sposób widok zostanie skonfigurowany dla menu kontekstowych.

**Listing 7.15.** Rejestrowanie widoku `TextView` dla menu kontekstowego

---

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TextView tv = (TextView)this.findViewById(R.id.textViewId);
    registerForContextMenu(tv);
}
```

---

## Zapełnianie menu kontekstowego

Po zarejestrowaniu przykładowego widoku `TextView` dla menu kontekstowych Android wywoła metodę `onCreateContextMenu()`, w której argumentem będzie ten widok. To właśnie tutaj można zapełnić menu kontekstowe odpowiednimi elementami. Dzięki wywołanej metodzie `onCreateContextMenu()` dostępne stają się trzy potrzebne argumenty.

Pierwszym argumentem jest domyślnie utworzona klasa `ContextMenu`, drugi argument to widok (na przykład `TextView`), który wygenerował wywoływanie zwrotne, a trzeci argument stanowi klasa `ContextMenuInfo`, którą pokróćce omówiliśmy podczas analizowania rysunku 7.3. W wielu prostych przypadkach można po prostu zignorować trzeci argument, jednak niektóre widoki mogą przenosić dzięki niemu dodatkowe informacje. W takich przypadkach trzeba będzie umieścić klasę `ContextMenuInfo` w podklasie, a następnie zastosować dodatkowe metody, umożliwiające odczyt danych.

Przykładami klas wywodzących się z klasy ContextMenuInfo są AdapterContextMenuInfo oraz ExpandableContextMenuInfo. Widoki, które są powiązane w Androidzie z bazodanowymi kursorami, wykorzystują klasę AdapterContextMenuInfo do przekazywania identyfikatora krotki do widoku, w którym menu kontekstowe będzie wyświetlane. W pewnym sensie można stosować tę klasę do dalszego zwiększania przejrzystości obiektu kryjącego się pod kliknięciem myszy, nawet w danym widoku.

Na listingu 7.16 została zaprezentowana metoda onCreateContextMenu().

#### **Listing 7.16.** Metoda onCreateContextMenu()

---

```
@Override  
public void onCreateContextMenu(ContextMenu menu, View v, ContextMenuInfo menuInfo)  
{  
    menu.setHeaderTitle("Przykładowe menu kontekstowe");  
    menu.add(200, 200, 200, "element1");  
}
```

---

### **Tworzenie odpowiedzi na kliknięcie elementu menu kontekstowego**

Trzecim etapem implementacji menu kontekstowego jest zdefiniowanie odpowiedzi na kliknięcie jego elementów. Mechanizm tworzenia odpowiedzi w menu kontekstowym jest analogiczny do przeprowadzania tej czynności w menu opcji. Dostępna jest wywoływana metoda podobna do onOptionsItemSelected(), nazwana onContextItemSelected(). Obydwie są dostępne w klasie Activity. Listing 7.17 przedstawia zastosowanie metody onContextItemSelected().

#### **Listing 7.17.** Tworzenie odpowiedzi dla menu kontekstowego

---

```
@Override  
public boolean onContextItemSelected(MenuItem item)  
{  
    if (item.getItemId() = jakis-identyfikator-elementu-menu)  
    {  
        // przetwarza ten element menu  
        return true;  
    }  
    ... inne przetwarzanie wyjątku  
}
```

---

### **Praca z menu alternatywnymi**

Do tej pory nauczyliśmy się tworzyć i obsługiwać menu, podmenu oraz menu kontekstowe. W Androidzie zaprezentowano nową koncepcję — **menu alternatywne** — pozwalającą elementom takiego menu stanowić część standardowych menu, podmenu oraz menu kontekstowych. Menu alternatywne pozwalają wielu aplikacjom w Androidzie na wzajemne użytkowanie. Takie menu alternatywne stanowią część międzyaplikacyjnego systemu komunikacyjnego lub struktury użytkowania.

W szczególności menu alternatywne pozwalają na zamieszczanie menu jednej aplikacji wewnątrz drugiej. Po wybraniu menu alternatywnego docelowa aplikacja lub aktywność uruchamia się za pośrednictwem adresu URL w celu przekazania danych wymaganych przez żądającą aktywność. Następnie wywołana aktywność wykorzysta adres URL danych, które zostały przekazane za pomocą intencji. Żeby dobrze pojąć działanie menu alternatywnych, trzeba najpierw zrozumieć pojęcia dostawców treści, identyfikatorów URI treści, typów MIME treści oraz intencji (rozdział 4. i 5.).

Ogólna zasada działania jest następująca: wyobraźmy sobie, że tworzymy ekran, który ma za zadanie wyświetlać dane. Najprawdopodobniej ekran ten będzie aktywnością. W tej aktywności będzie dostępne menu opcji pozwalających na różne sposoby modyfikowania tych danych. Założymy także na chwilę, że pracujemy nad dokumentem lub notatką, definiowaną przez identyfikator URI oraz odpowiadający mu typ MIME. Jako programiści chcemy, żeby urządzenie posiadało więcej aplikacji umożliwiających edytowanie lub wyświetlanie tych danych. Chcemy sprawić, żeby menu tych programów były wyświetlane jako część menu tworzonego dla naszej aktywności.

Żeby przyłączyć elementy menu alternatywnego do naszego menu, należy wykonać następujące czynności podczas konfigurowania menu w metodzie `onCreateOptionsMenu`:

1. Utwórz intencję, której identyfikator URI danych jest ustanowiony dla pokazywanego w bieżącym momencie identyfikatora URI danych.
2. Przydziel tę intencję do kategorii `CATEGORY_ALTERNATIVE`.
3. Wyszukaj aktywności, które pozwalają na obsługę typu danych definiowanych przez identyfikator URI.
4. Intencje wywołujące te aktywności dodaj jako elementy menu.

Wymienione etapy mówią nam wiele na temat natury aplikacji w Androidzie, więc zastanowimy się nad każdym z nich. Jak już wiemy, przyłączanie elementów menu alternatywnego do standardowego menu jest przeprowadzane w metodzie `onCreateOptionsMenu`:

```
@Override public boolean onCreateOptionsMenu(Menu menu)
{
}
```

Zastanówmy się, jaki kod tworzy tę funkcję. Najpierw musimy poznać identyfikator URI danych, na których zamierzamy pracować w danej aktywności. Można go uzyskać w następujący sposób:

```
this.getIntent().getData()
```

Technika ta działa, ponieważ klasa `Activity` posiada metodę `getIntent()`, przekazującą identyfikator URI danych, dla których została przywołana aktywność. Taką aktywnością może być główna aktywność wywołana przez menu główne; w takim przypadku może nie posiadać intencji i metoda `getIntent()` zwróci wartość `null`. Pisząc kod, należy zabezpieczyć się przed podobną sytuacją.

Teraz naszym celem jest odnalezienie innych programów, które potrafią pracować z tego rodzaju danymi. Wyszukiwanie przeprowadzamy, wstawiając intencję w miejsce argumentu. Poniżej przedstawiamy kod pozwalający na skonstruowanie odpowiedniej intencji:

```
Intent criteriaIntent = new Intent(null, getIntent().getData());
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

Po skonstruowaniu intencji dodamy także kategorię interesujących nas działań. Gwoli ściśleści, interesują nas jedynie te aktywności, którą mogą być wywołane jako część menu alternatywnego. Jesteśmy już gotowi do zaprogramowania obiektu Menu, aby wyszukała pasujące aktywności i dodała je jako opcje w menu (listing 7.18).

---

**Listing 7.18.** Zapełnianie menu elementami menu alternatywnego

```
// Wyszukuje pasujące aktywności i wypełnia nimi menu.
menu.addIntentOptions(
    Menu.CATEGORY_ALTERNATIVE, // Grupa
    Menu.CATEGORY_ALTERNATIVE, // Unikatowe identyfikatory, które chcemy dodać.
    Menu.CATEGORY_ALTERNATIVE, // Kolejność
    getComponentName(),        // Nazwa klasy wyświetlającej menu Name
    null,                      // --tutaj, to jest ta klasa.
    criteriaIntent,            // Uprzednio utworzona intencja, która
    null);                     // opisuje nasze wymagania.
                                // Bez szczegółów.
                                // Bez flag.
                                // Zwracane elementy menu
```

---

Zanim omówimy każdą linijkę kodu, wyjaśnimy, co rozumiemy pod pojęciem „pasujące aktywności”. **Pasująca aktywność** to taka aktywność, która potrafi przetworzyć przekazany jej identyfikator URI. Informacje dotyczące obsługiwanych identyfikatorów URI są zazwyczaj rejestrowane w plikach manifestach aktywności za pomocą identyfikatorów URI, działań i kategorii. W Androidzie znajduje się mechanizm pozwalający używać obiektu Intent do wyszukiwania pasujących aktywności, jeśli ma się dane te atrybuty.

Przyjrzymy się teraz uważniej listingowi 7.18. Metoda addIntentOptions w klasie Menu wyszukuje aktywności pasujące do identyfikatora URI intencji oraz atrybutów kategorii. Następnie metoda dodaje te aktywności do właściwej grupy w menu, korzystając z identyfikatorów elementów menu oraz identyfikatorów kolejności. Tym aspektem działania metody zajmują się trzy pierwsze atrybuty. Na listingu 7.18 grupą, od której zaczniemy dodawanie nowych elementów menu, jest Menu.CATEGORY\_ALTERNATIVE. Ta sama stała jest używana jako wartość bazowa dla identyfikatorów elementów oraz kolejności.

Kolejny argument wskazuje na w pełni kwalifikowaną nazwę składnika aktywności, której częścią jest nasze menu. W kodzie jest zastosowana pomocnicza metoda getComponentName(), wywodząca się z klasy Activity. **Nazwa składnika** (ang. *component name*) stanowi po prostu nazwę pakietu oraz klasy i jest ona wymagana, ponieważ za każdym razem, gdy jest dodawany nowy element menu, element ten będzie musiał wywoływać docelową aktywność. Żeby tego dokonać, system wymaga źródłowej aktywności, która uruchomiła aktywność docelową. Następnym argumentem jest tablica intencji, która powinna być stosowana jako filtr wobec zwracanych intencji. W naszym przykładzie wprowadziliśmy wartość null.

Kolejny argument wskazuje obiekt criteriaIntent, który dopiero co skonstruowaliśmy. Jest to kryterium wyszukiwania, którego chcemy użyć. Następujący po nim argument jest flagą typu Menu.FLAG\_APPEND\_TO\_GROUP — wskazującą na to, czy elementy menu mają być dodawane do istniejącej grupy menu, czy też mają być podmieniane. Domyślana wartość wynosi 0, co wskazuje na to, że elementy grupy menu mają być podmieniane.

Ostatnim argumentem na listingu 7.18 jest tablica dodanych elementów menu. Można korzystać z takiego odniesienia do dodanych elementów, w przypadku gdy trzeba je w jakiś sposób zmodyfikować już po ich dodaniu.

Wszystko brzmi prosto i pięknie. Pozostaje jednak kilka pytań bez odpowiedzi. Na przykład: jakie będą nazwy dodanych elementów? Dokumentacja Androida jest w tym temacie wyjątkowo uboga, zatem poszperaliśmy trochę w kodzie źródłowym, żeby dowiedzieć się, jak ta funkcja w rzeczywistości działa poza wzrokiem użytkownika (w rozdziale 1. opisaliśmy, w jaki sposób uzyskać dostęp do kodu źródłowego Androida).

Okazuje się, że klasa `Menu` jest jedynie interfejsem, więc nie widzieliśmy jej kodu źródłowego. Klasą implementującą interfejs `Menu` jest `MenuBuilder`. Na listingu 7.19 umieściliśmy kod podobnej metody, `addIntentOptions`, z klasy `MenuBuilder` (wstawiliśmy ten kod wyłącznie w celach poglądowych; nie będziemy go szczegółowo objaśniać).

#### **Listing 7.19.** Metoda `MenuBuilder.addIntentOptions`

```
public int addIntentOptions(int group, int id, int categoryOrder,
                            ComponentName caller,
                            Intent[] specifics,
                            Intent intent, int flags,
                            MenuItem[] outSpecificItems)
{
    PackageManager pm = mContext.getPackageManager();
    final List<ResolveInfo> lri =
        pm.queryIntentActivityOptions(caller, specifics, intent, 0);
    final int N = lri != null ? lri.size() : 0;

    if ((flags & FLAG_APPEND_TO_GROUP) == 0) {
        removeGroup(group);
    }

    for (int i=0; i<N; i++) {
        final ResolveInfo ri = lri.get(i);
        Intent rintent = new Intent(
            ri.specifyIndex < 0 ? intent : specifics[ri.specifyIndex]);
        rintent.setComponent(new ComponentName(
            ri.activityInfo.applicationInfo.packageName,
            ri.activityInfo.name));
        final MenuItem item = add(group, id, categoryOrder, ri.loadLabel(pm));
        item.setIntent(rintent);
        if (outSpecificItems != null && ri.specifyIndex >= 0) {
            outSpecificItems[ri.specifyIndex] = item;
        }
    }
    return N;
}
```

Zauważmy, że jeden wiersz na listingu 7.19 został pogrubiony; ta część kodu jest odpowiedzialna za konstruowanie elementu menu. Zadanie określenia tytułu menu zostaje przekazane klasie `ResolveInfo`. W kodzie źródłowym tej klasy widać, że filtr deklarujący tę intencję powinien posiadać powiązany z nią tytuł. Poniżej znajduje się przykład:

```
<intent-filter android:label="Tytuł menu ">
    ...
    <category android:name="android.intent.category.ALTERNATE" />
    <data android:mimeType="jakiś typ danych" />
</intent-filter>
```

Wartość `label` filtru intencji staje się nazwą menu. Można sprawdzić zachowanie tej klasy na podstawie przykładowej aplikacji Notepad.

## Praca z menu w odpowiedzi na zmianę danych

Do tej pory zajmowaliśmy się menu statycznymi; zostają one skonfigurowane na początku i nie zmieniają się dynamicznie w zależności od zawartości ekranu. Żeby utworzyć menu dynamiczne, należy zastosować dostępną w Androidzie metodę `onPrepareOptionsMenu`. Przypomina ona metodę `onCreateOptionsMenu` — z wyjątkiem faktu, że jest wywoływana za każdym razem, gdy zostaje przywołane menu. Jest ona stosowana na przykład, gdy niektóre menu lub grupy menu mają być wyłączone na podstawie danych wyświetlanych na ekranie. Warto o tym pamiętać podczas projektowania menu.

Musimy zająć się jeszcze jednym istotnym aspektem menu, zanim przejdziemy do okien dialogowych. Android umożliwia tworzenie menu za pomocą plików XML. Następny podrozdział poświęcono właśnie zapoznaniu się z obsługą takich menu mających postać plików XML.

## Wczytywanie menu poprzez pliki XML

Aż do tej chwili tworzyliśmy nasze menu za pomocą kodu Java. Nie jest to najwygodniejszy sposób, ponieważ każde menu wymaga kilku identyfikatorów oraz stałych zdefiniowanych dla tych identyfikatorów. Niewątpliwie po pewnym czasie staje się to nużące.

Zamiast tego można zdefiniować menu w plikach XML; Android pozwala na to, ponieważ menu są uznawane za zasoby. Tworzenie menu za pomocą plików XML ma kilka zalet, takich jak możliwość nadania nazwy menu, automatyczne tworzenie kolejności menu oraz przyznawanie identyfikatorów i tak dalej. Można także wprowadzić obsługę lokalizacji wobec tekstu zawartego w menu.

Żeby zaprojektować menu oparte na pliku XML, należy wykonać następujące czynności:

1. Zdefiniuj plik XML ze znacznikami menu.
2. Umieść plik w podkatalogu `/res/menu`. Nazwa pliku może być dowolna. Nie ma ograniczeń co do liczby plików. Android wygeneruje automatycznie identyfikator tego pliku.
3. Wykorzystaj identyfikator zasobów tego menu, aby wczytać plik XML do menu.
4. Zdefiniuj odpowiedzi na kliknięcie za pomocą identyfikatora zasobów wygenerowanego dla każdego elementu menu.

W poniższych punktach omówimy każdy z tych etapów oraz zaprezentujemy fragmenty kodu.

## Struktura pliku XML zasobów menu

Najpierw przyjrzymy się plikowi XML zawierającemu definicje menu (listing 7.20). Wszystkie pliki menu rozpoczynają się od znacznika `menu`, po którym następuje seria znaczników `group`. Każdy znacznik `group` odpowiada grupie elementów menu, omówionej na początku rozdziału.

Identyfikator grupy można określić za pomocą wyrażenia @+id. W każdej grupie menu będą umieszczone elementy menu, których identyfikatory będą powiązane z nazwami symbolicznymi. W dokumentacji środowiska Android SDK można znaleźć wszystkie argumenty dla tych znaczników języka XML.

#### **Listing 7.20.** Plik XML zawierający definicje menu

---

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">

        <item android:id="@+id/menu_testPick"
            android:orderInCategory="5"
            android:title="Część testowa" />
        <item android:id="@+id/menu_testGetContent"
            android:orderInCategory="5"
            android:title="Test pobierania treści" />
        <item android:id="@+id/menu_clear"
            android:orderInCategory="10"
            android:title="Wyczyść" />
        <item android:id="@+id/menu_dial"
            android:orderInCategory="7"
            android:title="Dzwon" />
        <item android:id="@+id/menu_test"
            android:orderInCategory="4"
            android:title="@string/test" />
        <item android:id="@+id/menu_show_browser"
            android:orderInCategory="5"
            android:title="Wyświetl przeglądarkę" />
    </group>
</menu>
```

---

Plik XML menu, którego kod widać na listingu 7.20, posiada jedną grupę menu. Na podstawie definicji identyfikatora zasobu @+id/menuGroup\_main tej grupie zostanie automatycznie przydzielony identyfikator zasobów menuGroup\_main w pliku R.java. W analogiczny sposób wszystkie potomne elementy menu uzyskują własne identyfikatory, tworzone na bazie definicji symbolicznych identyfikatorów zasobów z tego pliku XML.

## Zapełnianie plików XML zasobów menu

Załóżmy, że nasz plik XML nosi nazwę *moje\_menu.xml*. Należy umieścić ten plik w podkatalogu */res/menu*. Dzięki temu automatycznie zostanie wygenerowany identyfikator zasobów *R.menu.moje\_menu*.

Spójrzmy teraz, w jaki sposób możemy wykorzystać ten identyfikator do zapełnienia menu opcji. W Androidzie jest dostępna klasa *android.view.MenuInflater* służąca do umieszczania obiektów Menu z plików XML. Klasa *MenuInflater* posłuży nam do zapełnienia menu za pomocą identyfikatora *R.menu.moje\_menu*:

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    MenuInflater inflater = getMenuInflater(); // z aktywności
```

```
inflater.inflate(R.menu.moje_menu, menu);

// Musi być zwrócona wartość true, żeby menu było widoczne
return true;
}
```

W powyższym kodzie najpierw pobieramy klasę `MenuInflater` z klasy `Activity`, a następnie każemy jej przesyłać zawartość pliku XML bezpośrednio do menu.

## Tworzenie odpowiedzi dla elementów menu opartych na pliku XML

Nie mieliśmy jeszcze do czynienia z wyjątkową zaletą tej metody — staje się ona widoczna dopiero wtedy, gdy przychodzi czas na tworzenie odpowiedzi dla elementów menu. Odpowiedź dla elementów umieszczonych w pliku XML tworzy się podobnie jak w przypadku programowania w języku Java, istnieje jednak mała różnica. Tak jak wcześniej, elementami menu zajmuje się wywoływana metoda `onOptionsItemSelected`. Tym razem będziemy musieli skorzystać z pomocy zasobów Androida (rozdział 3. został poświęcony między innymi zasobom). W punkcie „Struktura pliku XML zasobów menu” wspomnieliśmy, że Android generuje nie tylko identyfikator zasobu dla pliku XML, lecz również niezbędne identyfikatory elementów, dzięki którym są one rozróżniane. Pod względem tworzenia odpowiedzi dla elementów menu jest to olbrzymia zaleta, ponieważ nie trzeba jawnie tworzyć identyfikatorów tych elementów i zarządzać nimi.

Żeby w pełni zrozumieć implikacje, w przypadku menu napisanych w języku XML nie trzeba definiować stałych dla tych identyfikatorów oraz nie należy się martwić o ich unikatowość, ponieważ tą kwestią zajmuje się proces generowania identyfikatorów zasobów. Widać to w poniższym kodzie:

```
private void onOptionsItemSelected (MenuItem item)
{
    this.appendMenuItemText(item);
    if (item.getItemId() == R.id.menu_clear)
    {
        this.emptyText();
    }
    else if (item.getItemId() == R.id.menu_dial)
    {
        // coś robi
    }
    else if (item.getItemId() == R.id.menu_testPick)
    {
        // coś robi
    }
    else if (item.getItemId() == R.id.menu_testGetContent)
    {
        // coś robi
    }
    else if (item.getItemId() == R.id.menu_show_browser)
    {
        // coś robi
    }
    ...itd.
}
```

Zauważmy, w jaki sposób nazwy elementów menu z pliku zasobów XML automatycznie wygenerowały identyfikatory elementów w przestrzeni nazw R.id.

## Krótkie wprowadzenie do dodatkowych znaczników menu w pliku XML

Podczas konstruowania plików XML należy znać dostępne znaczniki. Informacje na ich temat można szybko zdobyć, przeglądając wersje demonstracyjne interfejsów API, dostępne w zestawie Android SDK. Wśród nich znajduje się zbiór menu, które pomagają zrozumieć wszystkie aspekty programowania w Androidzie. W podkatalogu /res/menu znajduje się wiele przykładowych plików XML zasobów menu. Omówimy pokrótce niektóre kluczowe znaczniki.

### Znacznik kategorii grupy

W pliku XML można określić kategorię grupy za pomocą znacznika menuCategory:

```
<group android:id="@+id/some-group-id"
       android:menuCategory="secondary">
```

### Znacznik zaznaczania

Do kontrolowania zaznaczania na poziomie grupy można zastosować znacznik checkableBehavior:

```
<group android:id="@+id/noncheckable_group"
       android:checkableBehavior="none">
```

Znacznik checked służy do kontrolowania zaznaczania na poziomie pojedynczych elementów:

```
<item android:id="..."
      android:title="..."
      android:checked="true" />
```

### Tagi do symulowania podmenu

Podmenu jest reprezentowane jako element wewnętrzny menu:

```
<item android:title="Wszystko poza grupą">
    <menu>
        <item...>
    </menu>
</item>
```

### Znacznik ikony menu

Dzięki znacznikowi icon można powiązać obraz z elementem menu:

```
<item android:id="..."
      android:icon="@drawable/jakis-plik" />
```

## Znacznik włączania i wyłączania menu

Można włączać i wyłączać element menu za pomocą znacznika `enabled`:

```
<item android:id="..."  
      android:enabled="true"  
      android:icon="@drawable/jakis-plik" />
```

## Skróty dla elementu menu

Można ustanawiać skróty dla elementów menu poprzez znacznik `alphabeticShortcut`:

```
<item android:id="..."  
      android:alphabeticShortcut="a"  
      ...  
</item>
```

## Widoczność menu

Widoczność elementu menu jest kontrolowana za pomocą flagi `visible`:

```
<item android:id="..."  
      android:visible="true"  
      ...  
</item>
```

## Odbońniki

W trakcie nauki korzystania z klas menu Androida przydatny może się okazać poniższy adres URL. Można tam znaleźć projekty środowiska Eclipse, powiązane z niniejszym rozdziałem.

- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — z tego adresu możemy pobrać projekt testowy, ukazujący koncepcje omówione w tym rozdziale. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R07\_Menu*.

## Podsumowanie

W niniejszym rozdziale zaprezentowano sposób korzystania z różnorodnych rodzajów menu dostępnych w Androidzie: menu standardowych, kontekstowych, alternatywnych oraz opartych na języku XML. W niektórych rozdziałach, na przykład w 8. („Praca z oknami dialogowymi”), 16. („Analiza animacji dwuwymiarowej”) czy 20. („Programowanie grafiki trójwymiarowej za pomocą biblioteki OpenGL”), przedstawimy sposób wykorzystania menu XML do przetestowania omawianych w nich funkcji. Natomiast wprowadzone w wersji 3.0 Androida paski zadań oraz ich interakcja z menu są przedmiotami rozważań w rozdziale 30.

# Praca z oknami dialogowymi

System Android zapewnia rozbudowaną obsługę okien dialogowych. Wśród jawnie obsługiwanych typów okien dialogowych wyróżniamy okna alertów, listy kompleta cyjne, okna dialogowe pojedynczego wyboru, okna wielokrotnego wyboru, okna informujące o postępie działania oraz obiekty TimePicker i DatePicker (lista ta może się różnić w zależności od wersji Androida). Istnieje również możliwość stosowania niestandardowych okien dialogowych w razie potrzeby. Zasadniczym celem tego rozdziału jest omówienie nie samych okien dialogowych Androida, lecz raczej architektury kryjącej się pod tym pojęciem. W wersji 3.0 systemu dodano okna dialogowe oparte na fragmentach. Ten aspekt został omówiony w rozdziale 29. Spodziewamy się, że wraz z upływem czasu okna dialogowe oparte na fragmentach będą w coraz większym stopniu zastępować omawiane tu tradycyjne okna. Jednak te klasyczne okna dialogowe nie są jeszcze przestarzałe i ciągle stanowią standard w urządzeniach obsługujących system Android.

Okna dialogowe w Androidzie są asynchroniczne, dzięki czemu zapewnia się większą elastyczność. Jednak osoby przyzwyczajone do pracy w środowiskach programistycznych obsługujących okna dialogowe synchroniczne (na przykład Microsoft Windows) mogą uznać okna dialogowe asynchroniczne za nieco nieintuicyjne.

Dzięki omówieniu podstaw tworzenia oraz stosowania okien dialogowych w Androidzie wprowadzimy pewną intuicyjną abstrakcję, ułatwiającą pracę z asynchronicznymi oknami dialogowymi. Następnie za pomocą tej abstrakcji zaimplementujemy kilka przykładowych okien dialogowych. W znajdująącym się na końcu rozdziału podrozdziale „Odbońniki” zamieściliśmy również adres URL strony zawierającej przygotowany projekt, utworzony z myślą o oknach dialogowych. Warto pobrać ten projekt w celu poeksperymentowania z objaśnianymi tu konцепcjami oraz kodem.

## Korzystanie z okien dialogowych w Androidzie

Osoby pracujące w środowiskach obsługujących synchroniczne okna dialogowe (zwłaszcza modalne) muszą zmienić tok myślenia podczas projektowania okien dialogowych w Androidzie. W tym systemie okna dialogowe są asynchroniczne, a do tego również **zarządzane**. Oznacza to, że są wielokrotnie wykorzystywane pomiędzy wywołaniami, być może w celu zwiększenia wydajności.

### Projektowanie okien alertów

Dyskusję rozpoczęliśmy od okien alertów. Zasadniczo **okna alertów** wyświetlają proste informacje dotyczące poprawności wpisywanych danych lub czasami ostrzeżeń przed błędami (prawdziwymi lub rzekomymi). Prześledźmy poniższy przykład, często spotykany na stronach HTML:

```
if (validate(pole1) == false)
{
    // wskazuje poprzez okno alertu, że format nie jest poprawny
    showAlert("Dane w pole1 są wpisane w niewłaściwym formacie.");
    // przechodzi do tego pola
    //...i kontynuuje
}
```

Tego typu program najlepiej utworzyć w programie JavaScript za pomocą funkcji `alert`, która powoduje wyświetlenie prostego, synchronicznego okna dialogowego, zawierającego informację oraz przycisk `OK`. Po kliknięciu przycisku `OK` program działa dalej. To okno dialogowe jest uznawane za modalne oraz synchroniczne, ponieważ uzyskamy dostęp do następnej linijki kodu dopiero po wykonaniu funkcji `alert`.

Okno typu `alert` przydaje się podczas sprawdzania błędów. Jednak w Androidzie takie bezpośrednie funkcje lub okna dialogowe nie są dostępne. Zamiast nich jest obsługiwany konstruktor okien alertów, aplikacja ogólnego przeznaczenia służąca do tworzenia oraz używania okien alertów. Można zatem samemu stworzyć okno alertu, korzystając z klasy `android.app.AlertDialog.Builder`. Dzięki tej klasie można konstruować okna dialogowe pozwalające użytkownikowi wykonywać następujące czynności:

- odczytywać wiadomości oraz odpowidać *Tak* lub *Nie*;
- wybierać element z listy;
- wybierać wiele elementów z listy;
- obserwować postęp działania aplikacji;
- wybrać jedną z opcji;
- odpowiedzieć na zachętą, zanim program wznowi działanie.

Pokażemy, w jaki sposób skonstruować jedno z wymienionych okien dialogowych oraz jak wywołać je z elementu menu. Algorytm odnoszący się do wszystkich wspomnianych powyżej okien dialogowych składa się z następujących etapów:

1. Utwórz obiekt klasy `Builder`.
2. Ustaw takie parametry wyświetlania, jak liczba przycisków, lista elementów i tak dalej.
3. Ustanów metody wywoływanego zwracanego dla przycisków.

4. Zaprogramuj obiekt Builder, żeby skonstruował okno dialogowe. Typ utworzonego okna dialogowego zależy od informacji zawartych w tym obiekcie.
5. Zastosuj metodę `dialog.show()` do wyświetlenia okna dialogowego.

Na listingu 8.1 został przedstawiony kod, w którym zaimplementowano wymienione etapy.

#### **Listing 8.1.** Budowanie oraz wyświetlanie okna alertu

---

```
public class Alerts
{
    public static void showAlert(String message, Context ctx)
    {
        // Tworzenie konstruktora
        AlertDialog.Builder builder = new AlertDialog.Builder(ctx);
        builder.setTitle("Okno alertu");

        // Dodanie przycisków i obiektu nasłuchującego
        EmptyOnClickListener pl = new EmptyOnClickListener();
        builder.setPositiveButton("OK", pl);

        // Utworzenie okna dialogowego
        AlertDialog ad = builder.create();

        // Wyświetlenie
        ad.show();
    }

    public class EmptyOnClickListener
    implements android.content.DialogInterface.OnClickListener {
        public void onClick(DialogInterface v, int buttonId)
        {
        }
    }
}
```

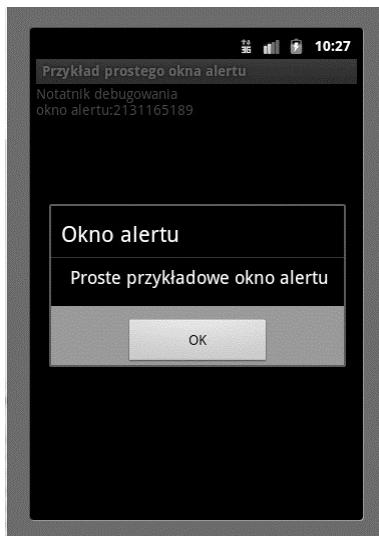
---

Kod z listingu 8.1 można wywołać w odpowiedniej aktywności testowej (na przykład w przygotowanym przez nas projekcie) poprzez utworzenie elementu menu i przypisanie mu następującej odpowiedzi:

```
if (item.getItemId() == R.id.menu_simple_alert)
{
    Alerts.showAlert("Przykład prostego okna alertu", this);
}
```

Efekt końcowy może wyglądać tak jak na rysunku 8.1 (w zależności od aktywności testowej).

Kod tego alertu nie jest skomplikowany (jest on zawarty na listingu 8.1 oraz w następującym po nim wycinku kodu). Nawet fragment dotyczący obiektu nasłuchującego jest łatwy do zrozumienia. W istocie naciśnięcie przycisku nic nie powoduje.



Rysunek 8.1. Proste okno alertu

Warto jednak zauważyć, że obiekt nasłuchujący przekazał odniesienie do interfejsu `Dialog` → `Interface`. To odniesienie wskazuje rzeczywiste okno dialogowe, wobec którego następuje wywołanie zwrotne. Interfejs ten obsługuje pewną liczbę stałych wykorzystywanych przez klasy okien dialogowych, pewną liczbę interfejsów wywołań zwrotnych oraz dwie kluczowe metody. Są to klasy:

```
cancel()  
dismiss()
```

Zazwyczaj nie musimy ich wywoływać, ponieważ w razie konieczności są one automatycznie przywoływane podczas kliknięcia przycisku. Jeżeli chcemy reagować na wywołania tych metod, możemy zarejestrować odpowiadające im wywołania zwrotne. W dokumentacji zestawu SDK dotyczącej interfejsu `DialogInterface` znajdziemy pełną listę dostępnych metod zwrotnych.

Na listingu 8.1 utworzyliśmy po prostu pusty obiekt nasłuchujący, zarejestrowany dla przycisku `OK`. Jedyną nową czynnością jest brak zastosowania polecenia `new` do utworzenia okna dialogowego. Zamiast tego skonfigurowaliśmy parametry i utworzyliśmy konstruktor, który zbudował okno alertu.

## Projektowanie okna dialogowego zachęty

Po udanym utworzeniu prostego okna alertu przejdźmy do nieco bardziej skomplikowanego rodzaju okna dialogowego: do okna dialogowego zachęty. Wedle definicji obowiązującej dla języka JavaScript okno dialogowe zachęty wyświetla wskazówkę lub pytanie i oczekuje od użytkownika wpisania danych w polu edycji. Wpisany ciąg znaków jest odsyłany programowi, który wznawia działanie. Jest to znakomity przykład do prześledzenia, ponieważ możemy zaobserwować zastosowanie wielu funkcji pochodzących z klasy `Builder`. Można też zapoznać się z naturą synchroniczności, asynchroniczności, modalności oraz ni/modalności okien dialogowych w Androidzie.

Poniżej przedstawiliśmy sposób utworzenia okna dialogowego zachęty:

1. Utwórz widok układu graficznego okna dialogowego zachęty.
2. Wczytaj układ graficzny do klasy View.
3. Skonstruuj obiekt klasy Builder.
4. Ustanów widok w obiekcie Builder.
5. Skonfiguruj przyciski wraz z ich wywołaniami zwrotnymi, żeby przyjmowały wprowadzany tekst.
6. Utwórz okno dialogowe za pomocą konstruktora alertów.
7. Wyświetl okno dialogowe.

Zaprezentujemy teraz kod dla każdego z opisanych etapów.

## Plik XML układu graficznego dla okna dialogowego zachęty

Aby zbudować okno dialogowe zachęty, trzeba najpierw wpisać tekst zachęty w widoku TextView, po którym ustawia się pole edycji. Użytkownik może w nim wpisać odpowiedź na zachętą. Listing 8.2 przedstawia zawartość pliku XML układu graficznego okna dialogowego zachęty. Jeżeli plik ten zostanie nazwany *prompt\_layout.xml*, należy go umieścić w podkatalogu */res/layout*, aby Android utworzył dla niego identyfikator zasobów R.layout.prompt\_layout.

**Listing 8.2.** Plik prompt\_layout.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/promptmessage"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_marginLeft="20dip"
        android:layout_marginRight="20dip"
        android:text="Tutaj należy wpisać tekst"
        android:gravity="left"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <EditText
        android:id="@+id/editText_prompt"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent"
        android:layout_marginLeft="20dip"
        android:layout_marginRight="20dip"
        android:scrollHorizontally="true"
        android:autoText="false"
        android:capitalize="none"
        android:gravity="fill_horizontal"
        android:textAppearance="?android:attr/textAppearanceMedium" />
</LinearLayout>
```

## Konfigurowanie konstruktora alertów za pomocą widoku użytkownika

Połączmy etapy 2., 3. oraz 4. naszej instrukcji: załadowanie widoku XML oraz skonfigurowanie go w konstruktorze alertów, aby utworzyć okno dialogowe zachęty. Android zawiera klasę `android.view.LayoutInflater` pozwalającą na utworzenie widoku View z pliku XML definicji układu graficznego. Teraz pokażemy, jak wykorzystać wystąpienie metody `LayoutInflater` do zapełnienia widoku naszego okna dialogowego na podstawie pliku XML układu graficznego (listing 8.3).

**Listing 8.3.** Umieszczanie układu graficznego w oknie dialogowym

---

```
LayoutInflater li = LayoutInflater.from(activity);
// zmienna „activity” stanowi odniesienie do naszej aktywności lub kontekstu
View view = li.inflate(R.layout.prompt_layout, null);

// przywołuje konstruktor i ustanawia widok
AlertDialog.Builder builder = new AlertDialog.Builder(ctx);
builder.setTitle("Zachęta");
builder.setView(view);
```

---

Na listingu 8.3 pokazano, jak uzyskać obiekt klasy `LayoutInflater` za pomocą statycznej metody `LayoutInflater.from(ctx)`, a następnie użyć go do przekształcenia kodu XML do widoku View. Po tym należy wprowadzić do konstruktora alertów tytuł oraz utworzony przed momentem widok.

## Konfigurowanie przycisków i obiektów nasłuchujących

Przechodzimy do etapu 5.: konfigurowania przycisków. Musimy umieścić przyciski *OK* oraz *Anuluj*, żeby użytkownik mógł odpowiedzieć na zachętę. Jeżeli użytkownik kliknie przycisk *Anuluj*, program nie będzie odczytywał informacji z okna dialogowego zachęty. Po wybraniu przycisku *OK* program pobierze wartość z pola tekstowego i przeniesie ją do aktywności.

Żeby skonfigurować te przyciski, potrzebny będzie obiekt nasłuchujący, odpowiadający na wywołania zwrotne. Kod dla obiektu nasłuchującego zostanie przedstawiony w podrozdziale „Obiekt nasłuchujący okna dialogowego zachęty”, teraz zaś zajmijmy się konfiguracją przycisków. Odpowiedni kod, stanowiący rozwinięcie listingu 8.3, znajduje się na listingu 8.4.

**Listing 8.4.** Konfigurowanie przycisków OK i Anuluj

---

```
// dodaje przyciski oraz obiekt nasłuchujący
PromptListener pl = new PromptListener(view);
builder.setPositiveButton("OK", pl);
builder.setNegativeButton("Anuluj", pl);
```

---

Pisząc kod na listingu 8.4, założyliśmy, że nazwą klasy obiektu nasłuchującego jest `PromptListener`. Ten obiekt nasłuchujący został zarejestrowany dla każdego przycisku. Klasa `PromptListener` pobiera skonstruowany za pomocą kodu z listingu 8.3 układ graficzny. Jeżeli przyjrzymy się tej klasie nieco uważniej, dostrzeżemy, że zmienna `view` jest stosowana do identyfikowania kontrolek tekstowych oraz odczytywania danych wprowadzanych przez użytkownika.

## Utworzenie i wyświetlenie okna dialogowego zachęty

W końcu docieramy do etapów 6. i 7.: utworzenia oraz wyświetlenia okna dialogowego zachęty. Dzięki konstruktorowi okien dialogowych nie powinno być z tym najmniejszych problemów (listing 8.5).

**Listing 8.5.** Konstruktor okien alertów tworzy okno dialogowe

---

```
// tworzy okno dialogowe
AlertDialog ad = builder.create();
ad.show();

// zwraca zachętą
return pl.getPromptReply();
```

---

W ostatniej linii wykorzystujemy obiekt nasłuchujący do odesłania odpowiedzi na zachęzę. Teraz, zgodnie z obietnicą, przedstawimy kod klasy `PromptListener`.

## Obiekt nasłuchujący okna dialogowego zachęty

Okno dialogowe zachęty oddziałyuje z aktywnością poprzez wywołanie klasy `PromptListener` obiektu nasłuchującego. Klasa ta posiada jedną metodę, nazwaną `onClick`, a przekazany tej metodzie identyfikator przycisku rozpoznaje, który przycisk został wciśnięty. Reszta kodu jest łatwa do zrozumienia (listing 8.6). Kiedy użytkownik wpisze tekst i kliknie przycisk `OK`, wartość z pola tekstowego zostanie przeniesiona do pola `promptReply`. W przeciwnym wypadku jego wartość pozostanie null. Zwróćmy uwagę na sposób wykorzystania identyfikatora kontrolki `EditText` (`editText_prompt`) utworzonej na listingu 8.2.

**Listing 8.6.** Klasa `PromptListener` obiektu nasłuchującego

---

```
public class PromptListener
implements android.content.DialogInterface.OnClickListener
{
    // Zmienna lokalna zwracająca wartość wpisaną w zachęcie
    private String promptReply = null;

    // Przechowuje zmienną dla widoku, aby odczytać wartość zachęty
    View promptDialogView = null;

    // Przyjmuje widok w konstruktorze
    public PromptListener(View inDialogView) {
        promptDialogView = inDialogView;
    }

    // Wywołuję metodę z okien dialogowych
    public void onClick(DialogInterface v, int buttonId) {
        if (buttonId == DialogInterface.BUTTON_POSITIVE) {
            // Przycisk OK
            promptReply = getPromptText();
        }
        else {
            // Przycisk Anuluj
            promptReply = null;
        }
    }
}
```

---

```
// Metoda dostępowa do zawartości pola edycji
private String getPromptText() {
    EditText et = (EditText)
        promptDialogView.findViewById(R.id.editText_prompt);
    return et.getText().toString();
}
public String getPromptReply() { return promptReply; }
}
```

---

## Przedstawienie kompletnego kodu

Po omówieniu każdego fragmentu kodu, dzięki któremu można wyświetlić okno dialogowe zachęty, zaprezentujemy go w całości, aby Czytelnik mógł go przetestować (listing 8.7). Pomineliśmy klasę PromptListener, ponieważ jej kod został przedstawiony oddzielnie na listingu 8.6.

### Listing 8.7. Przykładowy kod okna dialogowego zachęty

---

```
public class Alerts
{
    public static String prompt(String message, Context ctx)
    {
        // wczytuje jakiś widok
        LayoutInflater li = LayoutInflater.from(ctx);
        View view = li.inflate(R.layout.prompt_layout, null);

        // generuje konstruktor i ustanawia widok
        AlertDialog.Builder builder = new AlertDialog.Builder(ctx);
        builder.setTitle("Zachęta");
        builder.setView(view);

        // dodaje przyciski i obiekt nasłuchujący
        PromptListener pl = new PromptListener(view);
        builder.setPositiveButton("OK", pl);
        builder.setNegativeButton("Anuluj", pl);

        // generuje okno dialogowe
        AlertDialog ad = builder.create();

        // wyświetla
        ad.show();

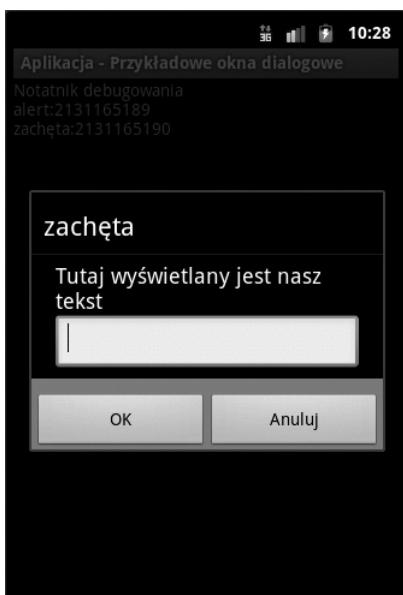
        return pl.getPromptReply();
    }
}
```

---

Kod z listingu 8.7 można wywołać poprzez utworzenie elementu menu w odpowiednim środowisku testowym oraz napisanie następującej odpowiedzi dla tego elementu:

```
if (item.getItemId() == R.id.nasz_identyfikator_elementu_menu)
{
    String reply = Alerts.showPrompt("Tutaj wpisujesz tekst", this);
}
```

Rezultat powinien przypominać ekran z rysunku 8.2.



Rysunek 8.2. Proste okno dialogowe zachęty

Jednak po przetestowaniu tego kodu Czytelnik zauważ, że okno dialogowe zachęty zawsze odsyła wartość null, nawet po wpisaniu tekstu w polu edycji. Wynika to z faktu, że metoda show() wywołuje okno dialogowe w sposób asynchroniczny:

```
ad.show(); // dialog.show  
return pl.getPromptReply(); // listener.getPromptReply()
```

Oznacza to, że wywołanie metody getPromptReply() (listing 8.6) następuje, zanim użytkownik zdąży wpisać tekst w polu edycji i kliknąć przycisku OK. Taki błąd logiczny zbliża nas do zrozumienia sedna natury okien dialogowych systemu Android.

## Natura okien dialogowych w Androidzie

Jak już wspomnieliśmy, wyświetlanie okien dialogowych w Androidzie jest procesem asynchronicznym. Po wyświetleniu okna dialogowego odpowiedzialny za to główny wątek wraca do programu i kontynuuje przetwarzanie reszty kodu. Nie oznacza to wcale, że okno dialogowe nie jest *modalne*. W istocie nadal takie jest. Kliknięcia myszą dotyczą wyłącznie okna dialogowego, podczas gdy nadzędna aktywność wraca do pętli komunikatów.

W niektórych systemach okienkowych modalne okna dialogowe zachowują się nieco inaczej. Program wywołujący pozostaje zablokowany, dopóki użytkownik nie udzieli odpowiedzi poprzez okno dialogowe (blokada programu może być czysto wirtualna). W systemach Windows wątek wysyłający komunikat rozpoczętu przesyłanie danych do okna dialogowego, a wstrzymuje je dla okna głównego. Po zamknięciu okna dialogowego wątek wraca do głównego okna. W tym przypadku wywołanie jest synchroniczne.

W przypadku urządzenia typu handheld, w którym niespodziewane zdarzenia występują częściej i główny wątek musi na nie reagować, takie podejście może się okazać nieskuteczne. Źeby zagwarantować wystarczająco krótki czas odpowiedzi, Android natychmiast zwraca główny wątek do pętli komunikatów.

W konsekwencji takiego założenia nie ma możliwości zaimplementowania prostego okna dialogowego, w którym program oczekuje odpowiedzi i wstrzymuje działanie do czasu jej uzyskania. W istocie model programowania okien dialogowych musi wyglądać inaczej pod kątem definiowania wywołań zwrotnych.

## Przeprojektowanie okna dialogowego zachęty

Przyjrzyjmy się ponownie problematycznemu fragmentowi kodu z poprzedniej implantacji okna dialogowego zachęty:

```
if (item.getItemId() == R.id.identyfikator_naszego_menu)
{
    String reply = Alerts.showPrompt("Tutaj wpisujesz tekst", this);
}
```

Udowodniliśmy, że wartość ciągu znaków w zmiennej reply będzie zawsze wynosiła null, ponieważ okno dialogowe zachęty, zainicjalizowane przez metodę Alerts.showPrompt(), nie ma możliwości odesłania wpisanej wartości do tego samego wątku. Jedynym sposobem, dzięki któremu można tego dokonać, jest bezpośrednie zaimplementowanie wywoływanej metody w aktywności, bez wykorzystywania klasy PromptListener. W tym celu należy zaimplementować metodę onClickListener w klasie Activity:

```
public class SampleActivity extends Activity
implements android.content.DialogInterface.OnClickListener
{
    // ...jakiś inny kod

    if (item.getItemId() == R.id.identyfikator_naszego_menu)
    {
        Alerts.showPrompt("Tutaj wpisujesz tekst", this);
    }
    // ...
}

public void onClick(DialogInterface v, int buttonId)
{
    // tutaj należy wprowadzić jakiś kod odpowiedzialny za odczytanie wartości ciągu
    // znaków z okna dialogowego
}
```

Jak widać, dzięki metodzie onClick można poprawnie odczytywać zmienne z wywołanego okna dialogowego, ponieważ zanim zostanie ona wywołana, użytkownik zdąży zamknąć okno dialogowe.

Taka forma korzystania z okien dialogowych jest całkowicie poprawna. Jednak twórcy Androida zapewnili dodatkowy mechanizm optymalizacji wydajności w postaci *zarządzanych okien dialogowych* — obiektów wielokrotnie wykorzystywanych przez różne wywołania. Nadal jednak trzeba stosować wywoływanie zwrotne. Tak naprawdę cała wiedza zdobyta podczas implementowania okna dialogowego zachęty przyda się podczas pracy z zarządzanymi oknami dialogowymi i ułatwi zrozumienie ich działania. Takie zarządzane okna dialogowe pozwalają również Andoidowi na kontrolowanie stanu okien dialogowych pomiędzy ich wywołaniami, dopóki stan widoku aktywności pozostaje niezmieniony.

## Praca z zarządzanymi oknami dialogowymi

Android wykorzystuje protokół zarządzania oknem dialogowym, dzięki czemu ponownie wykorzystuje uprzednio utworzone okna dialogowe, zamiast tworzyć ich nowe wystąpienia. W tym podrozdziale omówimy szczegóły protokołu zarządzania oknami dialogowymi oraz pokażemy, w jaki sposób zaimplementować alert w postaci zarządzanego okna dialogowego. Jednak naszym zdaniem protokół zarządzania oknami dialogowymi powoduje, że praca z tymi oknami jest dosyć żmudna. Utworzmy niewielką strukturę z wyrywkowych fragmentów tego protokołu, aby ułatwić pracę z zarządzanymi oknami dialogowymi.

### Protokół zarządzanych okien dialogowych

Podstawowym zadaniem protokołu zarządzanych okien dialogowych jest ponowne wykorzystanie okna dialogowego podczas jego kolejnych wywołań. Przypomina to stosowanie pul obiektów w środowisku Java. Protokół zarządzanych okien dialogowych składa się z następujących etapów:

1. Przypisz niepowtarzalny identyfikator dla każdego tworzonego i wykorzystywanej okna dialogowego. Zakładamy, że etykietą jednego z okien dialogowych jest 1.
2. Zaprogramuj wyświetlenie okna dialogowego z etykietą 1.
3. Android sprawdza, czy bieżąca aktywność zawiera już okno dialogowe oznaczone jako 1. Jeżeli tak jest, okno dialogowe zostaje wyświetlone bez konieczności ponownego tworzenia. W celach porządkowych Android wywołuje funkcję `onPrepareDialog()` przed wyświetleniem okna dialogowego.
4. Jeżeli okno dialogowe z etykietą 1 nie istnieje, Android wywołuje metodę `onCreateDialog()` poprzez przekazanie identyfikatora okna dialogowego (w tym przypadku 1).
5. Teraz przesłoń metodę `onCreateDialog()`. W tym celu utwórz okno dialogowe za pomocą konstruktora alertów, a następnie je wywołaj. Jednak najpierw musisz określić na podstawie identyfikatora, które okno dialogowe będzie tworzone. W tym celu stosuje się instrukcję `switch`.
6. Android wyświetla okno dialogowe.
7. Okno dialogowe wywołuje odpowiednie funkcje po kliknięciu przycisków.

Zastosujmy teraz protokół do ponownego zaimplementowania okna dialogowego, które nie jest zarządzane, w formie zarządzanego alertu.

### Przekształcenie niezarządzanego okna dialogowego na zarządzane okno dialogowe

W celu ponownego zaimplementowania okna alertu będziemy postępować zgodnie z etapami wypisanyimi w poprzednim podrozdziale. Rozpoczniemy od zdefiniowania niepowtarzalnego identyfikatora tego okna dialogowego w kontekście danej aktywności:

```
// unikatowy identyfikator okna dialogowego  
private static final int DIALOG_ALERT_ID = 1;
```

Wystarczająco proste. Właśnie utworzyliśmy identyfikator okna dialogowego, służący do sterowania wywołaniami zwrotnymi. Identyfikator ten pozwala nam na wykonanie poniższej czynności w odpowiedzi na kliknięcie elementu menu:

```
jakasaktywnosc.showDialog(this.DIALOG_ALERT_ID);
```

Dostępna w zestawie Android SDK metoda `showDialog` uruchamia proces wywołania metody `onCreateDialog()` klasy naszej aktywności. Android jest wystarczająco sprytny, żeby nie wywoływać wielokrotnie metody `onCreateDialog()`. Po jej wywołaniu musimy utworzyć okno dialogowe i odesłać je Androidowi. Metoda `onCreateDialog()` jest później przechowywana we wnętrzu systemu na wypadek potrzeby ponownego użycia. Poniżej przedstawiliśmy przykładowy kod, służący do utworzenia okna dialogowego na podstawie unikalnego identyfikatora:

```
public class SomeActivity extends Activity {  
    ...  
  
    @Override  
    protected Dialog onCreateDialog(int id) {  
        switch (id) {  
            case DIALOG_ALERT_ID:  
                return createAlertDialog();  
        }  
        return null;  
    }  
  
    private Dialog createAlertDialog()  
    {  
        AlertDialog.Builder builder = new AlertDialog.Builder(this);  
        builder.setTitle("Okno alertu");  
        builder.setMessage("jakis komunikat");  
        EmptyOnClickListener emptyListener = new EmptyOnClickListener();  
        builder.setPositiveButton("OK", emptyListener );  
        AlertDialog ad = builder.create();  
        return ad;  
    }  
}
```

Warto zwrócić uwagę, w jaki sposób metoda `onCreateDialog()` rozpoznaje przychodzący identyfikator w celu określenia właściwego okna dialogowego. Sama funkcja `onCreateDialog()` jest przechowywana w oddzielnej funkcji i przetwarza równolegle omówiony wcześniej proces tworzenia okna alertu. W tym kodzie również wykorzystano tę samą funkcję `EmptyOnClickListener`, która była stosowana podczas korzystania z alertu.

Ponieważ okno dialogowe jest tworzone tylko raz, potrzebny jest mechanizm pozwalający na zmianę elementów okna dialogowego podczas jego kolejnych wyświetleń. Do tego celu służy metoda `onPrepareDialog()`:

```
@Override  
protected void onPrepareDialog(int id, Dialog dialog) {  
    switch (id) {  
        case DIALOG_ALERT_ID:  
            prepareAlertDialog(dialog);  
    }  
}  
  
private void prepareAlertDialog(Dialog d) {  
    AlertDialog ad = (AlertDialog)d;  
    // tutaj należy coś zmienić w oknie dialogowym  
}
```

Jeżeli powyższy kod zostanie wstawiony, metoda `showDialog(1)` będzie działać. Nawet jeżeli ta metoda będzie przywoływana wiele razy, metoda `onCreateMethod` zostanie wywołana tylko raz. Taki sam protokół można zastosować wobec okna dialogowego zachęty.

Utworzenie odpowiedzi na wywołanie okna dialogowego to kawał roboty, jednak korzystanie z protokołu zarządzanych okien dialogowych jest jeszcze bardziej pracochłonne. Po dokładnym przestudiowaniu tego protokołu stwierdziliśmy, że wyabstrahujemy go i przekształcimy tak, żeby spełniał dwa zadania:

- spowodował, aby identyfikacja okna dialogowego oraz jego tworzenie odbywały się poza klasą aktywności obiektów;
- umieścił proces tworzenia okna dialogowego oraz jego odpowiedzi w wyspecjalizowanej klasie okna dialogowego.

W następnym podrozdziale omówimy proces projektowania takiej struktury, a następnie zastosujemy ją do ponownego utworzenia okna alertu i okna dialogowego zachęty.

## Uproszczenie protokołu zarządzanych okien dialogowych

Większość Czytelników pewnie zdążyła zauważyc, że praca z zarządzanymi oknami dialogowymi może być nieco chaotyczna, a dodatkowo może wprowadzić nieporządek do głównego kodu. Gdyby przekształcić protokół zarządzanych okien dialogowych w prostszą formę, wyglądałby on następująco:

1. Utwórz instancję wymaganego okna dialogowego, stosując instrukcję `new`. Zachowaj tę instancję jako lokalną zmienną. Nazwijmy ją `dialog1`.
2. Wyświetl okno dialogowe za pomocą metody `dialog1.show()`.
3. Zaimplementuj w aktywności jedną metodę nazwaną `dialogFinished()`.
4. W metodzie `dialogFinished()` odczytaj atrybuty zmiennej `dialog1`, takie jak `dialog1.getValue1()`.

Zgodnie z tym schematem wyświetlenie zarządzanego okna alertu będzie możliwe za pomocą następującego kodu:

```
//...class MyActivity ...
{
    // nowe okno dialogowe
    ManagedAlertDialog mad = new ManagedAlertDialog("komunikat", ..., ...);

    //...jakaś metoda menu
    if (item.getItemId() == R.id.identyfikator_naszego_menu)
    {
        // wyświetla okno dialogowe
        mad.show();
    }
    //...
    // w razie potrzeby uzyskuje dostęp do wnętrza okna dialogowego mad
    dialogFinished()
    {
        //...
        // używa wartości z okna dialogowego
        mad.getA();
    }
}
```

```

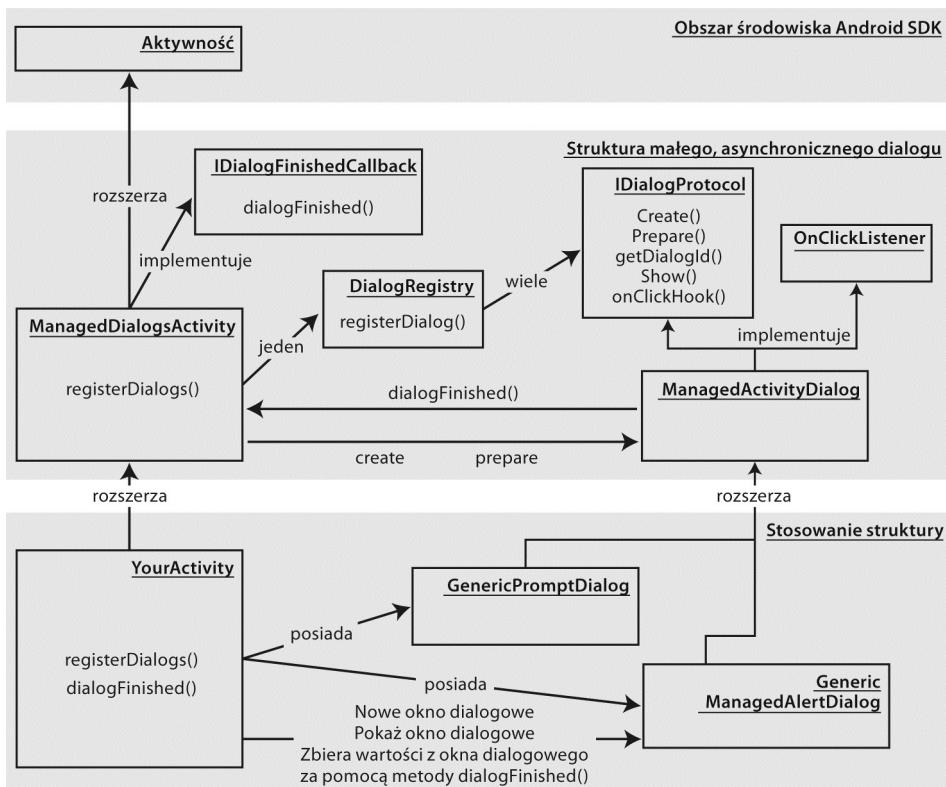
        mad.getB();
    }
}

```

Uważamy, że jest to o wiele prostszy model korzystania z okien dialogowych. Wyraźnie widać zalety tego rozwiązania:

- Nie ma potrzeby nadawania ani zapamiętywania własnych identyfikatorów.
- Nie trzeba wstawić do głównego kodu fragmentów służących do tworzenia okien dialogowych.
- Można bezpośrednio wykorzystywać obiekty pochodzące z okna dialogowego w celu odczytania wartości.

Jaki jest mechanizm działania tego abstrakcyjnego rozwiązania? W pierwszym etapie przenosimy procesy tworzenia i przygotowywania okna dialogowego do klasy rozpoznającej podstawowe okno dialogowe. Interfejs ten nosi nazwę `IDialogProtocol`. W tym interfejsie bezpośrednio zaimplementowano metodę `show()`. Takie okna dialogowe są zbierane oraz przechowywane w rejestrze bazowej klasy aktywności, a ich identyfikatory pełnią funkcje kluczy. Bazowa klasa na podstawie identyfikatorów okien dialogowych rozdziela wywołania `onCreate`, `onPrepare` oraz `onClick` i kieruje je do klasy okna dialogowego. Omawiana architektura została szczegółowo naszkicowana na rysunku 8.3.



Rysunek 8.3. Struktura prostego zarządzanego okna dialogowego

Na listingu 8.8 zaprezentowano zastosowanie tej struktury. W dalszej części rozdziału pokazaliśmy kod źródłowy najważniejszych klas (w tym GenericPromptDialog i GenericManagedAlertDialog), jednak nie zamieściliśmy w książce pełnej klasy sterownika. Na listingu 8.8 przedstawiliśmy jedynie jego najważniejsze aspekty. W podrozdziale „Odnośniki” można znaleźć adres URL do projektu, w którym klasa ta pozwala na przetestowanie uprzednio omówionych okien dialogowych.

#### **Listing 8.8.** Wersja skrócona protokołu zarządzanych okien dialogowych

```
public class MainActivity extends ManagedDialogsActivity
{
    // okno dialogowe 1
    private GenericManagedAlertDialog gmad =
        new GenericManagedAlertDialog(this,1,"InitialValue");

    // okno dialogowe 2
    private GenericPromptDialog gmpd =
        new GenericPromptDialog(this,2,"InitialValue");

    // elementy menu uruchamiające okna dialogowe
    if (item.getItemId() == R.id.identyfikator1_naszego_menu)
    {
        gmad.show();
    }
    else if (item.getItemId() == R.id.identyfikator2_naszego_menu)
    {
        gmpd.show();
    }

    // zajmuje się kwestią wywołań
    public void dialogFinished(ManagedActivityDialog dialog, int buttonId)
    {
        if (dialog.getDialogId() == gmpd.getDialogId())
        {
            // zakładając, że gmpd zawiera metodę dostępową do ciągu znaków odpowiedzi
            String replyString = gmpd.getReplyString();
        }
    }
}
```

Aby ta struktura zadziałała, należy najpierw rozszerzyć klasę `ManagedDialogsActivity`. Następnie tworzy się obiekty potrzebnych okien dialogowych, każdy pochodzący z klasy `ManagedActivityDialog`. W razie implementacji odpowiedzi na kliknięcie elementu menu wystarczy zastosować metodę `show()` wobec tych okien dialogowych. Okna dialogowe same pobierają parametry niezbędne do ich utworzenia i wyświetlenia. Chociaż przekazujemy identyfikator okna dialogowego, nie będziemy musieli już go pamiętać. Jeżeli ktoś chce, może je nawet całkiem pominąć.

Teraz zajmiemy się każdą klasą przedstawioną na rysunku 8.3. Poniższe fragmenty kodu źródłowego są również dostępne do pobrania ze strony przykładowych projektów. Jeżeli Czytelnik zechce skompilować ten kod, warto pobrać ten projekt. Jeśli jednak ktoś postanowi inaczej, w książce znajdzie większość wymaganego kodu źródłowego, ale będzie musiał samodzielnie wypełnić kilka luk.

## IDialogProtocol

Interfejs IDialogProtocol definiuje pojęcie zarządzanego okna dialogowego. Zadaniem zarządzanego okna dialogowego jest tworzenie okna dialogowego i przygotowanie go do każdego wyświetlenia. Rozsądne jest także przekazanie funkcji show() do samego okna dialogowego. Okno dialogowe musi rozpoznawać także kliknięcia przycisku oraz umożliwiać wywoływanie odpowiedniej funkcji nadzędnej po jego zamknięciu. Poniższy kod przedstawia te działania w formie zestawu funkcji:

```
public interface IDialogProtocol
{
    public Dialog create();
    public void prepare(Dialog dialog);
    public int getDialogId();
    public void show();
    public void onClickHook(int buttonId);
}
```

## ManagedActivityDialog

Abstrakcyjna klasa ManagedActivityDialog zapewnia wspólną implementację wszystkim klasom okien dialogowych, wymagającym wprowadzenia interfejsu IDialogProtocol. Pozwala bazowym klasom na przesłonięcie funkcji create oraz prepare, ale umożliwia zaimplementowanie wszystkich pozostałych metod klasy IDialogProtocol. Klasa ManagedActivityDialog informuje także nadziedną aktywność o zakończeniu działania okna dialogowego po zarejestrowaniu zdarzenia kliknięcia. Wykorzystuje wzorce szablonów uchwytów i pozwala klasom pochodnym na korzystanie z wyspecjalizowanej metody uchwytu onClickHook. Klasa ta jest również odpowiedzialna za przekierowanie metody show() do nadziednej aktywności, a tym samym implementacja tej metody staje się bardziej naturalna. Klasa ManagedActivityDialog powinna być stosowana jako klasa bazowa dla wszystkich nowych okien dialogowych (listing 8.9).

---

**Listing 8.9.** Klasa ManagedActivityDialog

```
public abstract class ManagedActivityDialog implements IDialogProtocol
    , android.content.DialogInterface.OnClickListener
{
    private ManagedDialogsActivity mActivity;
    private int mDialogId;
    public ManagedActivityDialog(ManagedDialogsActivity a, int dialogId)
    {
        mActivity = a;
        mDialogId = dialogId;
    }
    public int getDialogId()
    {
        return mDialogId;
    }
    public void show()
    {
        mActivity.showDialog(mDialogId);
    }
    public void onClick(DialogInterface v, int buttonId)
    {
        onClickHook(buttonId);
    }
}
```

---

```

        this.mActivity.dialogFinished(this, buttonId);
    }
}

```

---

## DialogRegistry

Klasa DialogRegistry pełni dwie funkcje. Tworzy mapę powiązań pomiędzy identyfikatorami okien dialogowych a ich rzeczywistymi (wbudowanymi) wystąpieniami. Kieruje także ogólne wywołania metod onCreate i onPrepare do określonych okien dialogowych poprzez mapowanie tych identyfikatorów na obiekty. Klasa ManagedDialogsActivity wykorzystuje klasę DialogRegistry jako magazyn rejestrujący nowe okna dialogowe (listing 8.10).

**Listing 8.10.** Klasa DialogRegistry

---

```

public class DialogRegistry
{
    SparseArray<IDialogProtocol> idsToDialogs
                                = new SparseArray();
    public void registerDialog(IDialogProtocol dialog)
    {
        idsToDialogs.put(dialog.getDialogId(), dialog);
    }

    public Dialog create(int id)
    {
        IDialogProtocol dp = idsToDialogs.get(id);
        if (dp == null) return null;
        return dp.create();
    }

    public void prepare(Dialog dialog, int id)
    {
        IDialogProtocol dp = idsToDialogs.get(id);
        if (dp == null)
        {
            throw new RuntimeException("Identyfikator okna dialogowego jest
→niezarejestrowany:" + id);
        }
        dp.prepare(dialog);
    }
}

```

---

## ManagedDialogsActivity

Klasa ManagedDialogsActivity jest traktowana jako klasa bazowa dla aktywności obsługujących zarządzane okna dialogowe. Utrzymuje jedno wystąpienie klasy DialogRegistry, dzięki czemu na bieżąco śledzi zarządzane okna dialogowe identyfikowane przez interfejs IDialogProtocol. Umożliwia pochodnym aktywnościom rejestrowanie ich okien dialogowych poprzez funkcję registerDialogs(). Jak widać na rysunku 8.3, jest również odpowiedzialna za przenoszenie semantyk create i prepare do właściwej instancji okna dialogowego poprzez wyszukanie jej w rejestrze okien dialogowych. Klasa ta zapewnia także metodę zwrotną dialogFinished dla każdego okna dialogowego znajdującego się w rejestrze (listing 8.11).

**Listing 8.11.** Klasa ManagedDialogsActivity

```
public class ManagedDialogsActivity extends Activity
    implements IDialogFinishedCallBack
{
    // Rejestr dla zarządzanych okien dialogowych
    private DialogRegistry dr = new DialogRegistry();

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.registerDialogs();
    }

    protected void registerDialogs()
    {
        // Nic nie robi
        // Klasy pochodne przesyłają tę metodę,
        // żeby mogły zarejestrować swoje okna dialogowe
        // Przykład:
        // registerDialog(this.DIALOG_ALERT_ID_3, gmad);

    }
    public void registerDialog(IDialogProtocol dialog)
    {
        this.dr.registerDialog(dialog);
    }

    @Override
    protected Dialog onCreateDialog(int id) {
        return this.dr.create(id);
    }
    @Override
    protected void onPrepareDialog(int id, Dialog dialog) {
        this.dr.prepare(dialog, id);
    }

    public void dialogFinished(ManagedActivityDialog dialog, int buttonId)
    {
        // Nic nie robi
        // Klasy pochodne przesyłają
    }
}
```

---

**IDialogFinishedCallBack**

Interfejs `IDialogFinishedCallBack` umożliwia klasie `ManagedActivityDialog` przekazywanie aktywności nadzędnej informacji, że użytkownik zakończył pracę z oknem dialogowym i można wywołać wobec okna dialogowego odpowiednie metody, żeby odczytać parametry. Zazwyczaj klasa `ManagedDialogsActivity` implementuje ten interfejs i zachowuje się jak klasa nadzędna wobec klasy `ManagedActivityDialog` (listing 8.12).

**Listing 8.12.** Interfejs IDialogFinishedCallBack

---

```
public interface IDialogFinishedCallBack
{
    public static int OK_BUTTON = -1;
    public static int CANCEL_BUTTON = -2;
    public void dialogFinished(ManagedActivityDialog dialog, int buttonId);
}
```

---

**GenericManagedAlertDialog**

Klasa GenericManagedAlertDialog jest implementacją okna alertu. Rozszerza ona klasę ManagedActivityDialog. Odpowiada za utworzenie rzeczywistego okna alertu za pomocą konstruktora okien alertów. Przenosi również wszystkie potrzebne jej informacje w formie zmiennych lokalnych. Ponieważ klasa GenericManagedAlertDialog implementuje proste okno alertu, nie wpływa w żaden sposób na metodę onClickHook. Podczas korzystania z tej klasy należy przede wszystkim zwrócić uwagę, że przechowuje ona wszystkie powiązane z nią informacje w jednym miejscu (listing 8.13). Oznacza to, że główny kod aktywności pozostaje sterylnie czysty.

**Listing 8.13.** Klasa GenericManagedAlertDialog

---

```
public class GenericManagedAlertDialog extends ManagedActivityDialog
{
    private String alertMessage = null;
    private Context ctx = null;
    public GenericManagedAlertDialog(ManagedDialogsActivity inActivity,
                                    int dialogId,
                                    String initialMessage)
    {
        super(inActivity,dialogId);
        alertMessage = initialMessage;
        ctx = inActivity;
    }
    public Dialog create()
    {
        AlertDialog.Builder builder = new AlertDialog.Builder(ctx);
        builder.setTitle("Okno alertu");
        builder.setMessage(alertMessage);
        builder.setPositiveButton("OK", this );
        AlertDialog ad = builder.create();
        return ad;
    }

    public void prepare(Dialog dialog)
    {
        AlertDialog ad = (AlertDialog)dialog;
        ad.setMessage(alertMessage);
    }
    public void setAlertMessage(String inAlertMessage)
    {
        alertMessage = inAlertMessage;
    }
    public void onClickHook(int buttonId)
    {
```

```
// nic nie robi  
// żadnych zmiennych lokalnych do skonfigurowania  
}  
}
```

---

## GenericPromptDialog

Klasa GenericPromptDialog przechowuje wszystkie dane niezbędne dla okna dialogowego zachęty poprzez rozszerzenie klasy ManagedActivityDialog oraz dostarczenie niezbędnych metod create i prepare (listing 8.14). Zachowuje ona także tekst wpisany przez użytkownika w oknie dialogowym zachęty jako zmienną lokalną, aby nadziedna aktywność mogła go odczytać dzięki metodzie dialogFinished.

**Listing 8.14.** Klasa GenericPromptDialog

---

```
public class GenericPromptDialog extends ManagedActivityDialog  
{  
    private String mPromptMessage = null;  
    private View promptView = null;  
    String promptValue = null;  
    private Context ctx = null;  
    public GenericPromptDialog(ManagedDialogsActivity inActivity,  
        int dialogId,  
        String promptMessage)  
    {  
        super(inActivity,dialogId);  
        mPromptMessage = promptMessage;  
        ctx = inActivity;  
    }  
    public Dialog create()  
    {  
        LayoutInflater li = LayoutInflater.from(ctx);  
        promptView = li.inflate(R.layout.promptdialog, null);  
        AlertDialog.Builder builder = new AlertDialog.Builder(ctx);  
        builder.setTitle("zachęta");  
        builder.setView(promptView);  
        builder.setPositiveButton("OK", this);  
        builder.setNegativeButton("Anuluj", this);  
        AlertDialog ad = builder.create();  
        return ad;  
    }  
    public void prepare(Dialog dialog)  
    {  
        // na razie puste  
    }  
    public void onClickHook(int buttonId)  
    {  
        if (buttonId == DialogInterface.BUTTON1)  
        {  
            // przycisk OK  
            String promptValue = getEnteredText();  
        }  
    }  
}
```

---

```

private String getEnteredText()
{
    EditText et =
        (EditText)
        promptView.findViewById(R.id.editText_prompt);
    String enteredText = et.getText().toString();
    Log.d("xx",enteredText);
    return enteredText;
}

```

---

Zaprezentowana tu struktura musi zostać nieco zmodyfikowana w przypadku przebudowy aktywności zależnej od konfiguracji urządzenia. Główna zmiana polega na odtworzeniu obiektów okien dialogowych za pomocą metod zapisywania oraz odczytywania instancji. Ponieważ omawiane typy okien dialogowych zostaną zastąpione przez okna dialogowe oparte na fragmentach (omówione w rozdziale 29.), postanowiliśmy nie wprowadzać w kodzie modyfikacji wymaganych do przechowywania okien dialogowych w zależności od konfiguracji urządzenia.

## Praca z klasą Toast

Rozpoczęliśmy ten rozdział od stwierdzenia, że okna alertu są często stosowane w procesie debugowania kodu JavaScript na błędnie działających stronach. Jeżeli musimy wprowadzić podobne rozwiązanie, powodujące sporadyczne wyświetlanie informacji o błędach, możemy w tym celu wykorzystać obiekt `Toast`.

Obiekt `Toast` przypomina okno alertu, które wyświetla informację przez określony czas, po czym znika. Można więc powiedzieć, że mamy tu do czynienia z chwilowo występującym oknem alertu.

Na listingu 8.15 zaprezentowaliśmy przykład wyświetlania komunikatu za pomocą obiektu `Toast`.

**Listing 8.15.** Zastosowanie klasy `Toast` w procesie debugowania

---

```

//Tworzy funkcję opakowującą komunikat w obiekt klasy Toast
//Wyświetla obiekt Toast
public void reportToast(String message)
{
    String s = tag + ":" + message;
    Toast mToast = Toast.makeText(activity, s, Toast.LENGTH_SHORT);
    mToast.show();
    Log.d(tag,message);
}

//Możemy w razie potrzeby przywoływać wielokrotnie
//powyższą funkcję, tak jak widać poniżej
private void testToast()
{
    reportToast("Komunikat1");
    reportToast("Komunikat2");
    reportToast("Komunikat3");
}

```

---

Widoczna na listingu 8.14 metoda `makeText()` może pobierać nie tylko obiekt aktywności, ale nawet obiekt kontekstu, na przykład przekazywany odbiorcy komunikatów lub usłudze. W ten sposób zastosowanie obiektu `Toast` wykracza poza granice aktywności.

## Odbośniki

- [\*http://developer.android.com/guide/topics/ui/dialogs.html\*](http://developer.android.com/guide/topics/ui/dialogs.html) — znakomita dokumentacja zestawu Android SDK stanowiąca wprowadzenie do pracy z oknami dialogowymi w Androidzie. Znajdziemy tu wyjaśnienie sposobu stosowania zarządzanych okien dialogowych oraz różnorodne przykłady dostępnych okien dialogowych.
- [\*http://developer.android.com/reference/android/content/DialogInterface.html\*](http://developer.android.com/reference/android/content/DialogInterface.html) — lista stałych zdefiniowanych dla dialogów.
- [\*http://developer.android.com/reference/android/app/Dialog.html\*](http://developer.android.com/reference/android/app/Dialog.html) — omówiony zbiór metod dostępnych w obiekcie klasy `Dialog`.
- [\*http://developer.android.com/reference/android/app/AlertDialog.Builder.html\*](http://developer.android.com/reference/android/app/AlertDialog.Builder.html) — dokumentacja interfejsów API dotycząca klasy `AlertDialog`.
- [\*http://developer.android.com/reference/android/app/ProgressDialog.html\*](http://developer.android.com/reference/android/app/ProgressDialog.html) — dokumentacja dotycząca klasy `ProgressDialog`.
- [\*http://developer.android.com/reference/android/app/DatePickerDialog.html\*](http://developer.android.com/reference/android/app/DatePickerDialog.html) — dokumentacja dotycząca klasy `DatePicker`.
- [\*http://developer.android.com/reference/android/app/TimePickerDialog.html\*](http://developer.android.com/reference/android/app/TimePickerDialog.html) — dokumentacja dotycząca klasy `TimePicker`.
- [\*http://developer.android.com/resources/tutorials/views/hello-datepicker.html\*](http://developer.android.com/resources/tutorials/views/hello-datepicker.html) — samouczek pozwalający na zrozumienie działania klasy `DatePicker`.
- [\*http://developer.android.com/resources/tutorials/views/hello-timepicker.html\*](http://developer.android.com/resources/tutorials/views/hello-timepicker.html) — samouczek ułatwiający pracę z klasą `TimePicker`.
- [\*ftp://ftp.helion.pl/przyklady/and3ta.zip\*](ftp://ftp.helion.pl/przyklady/and3ta.zip) — z tego adresu możemy pobrać testowy projekt, utworzony na podstawie niniejszego rozdziału. Właściwy plik znajdziesz w katalogu o nazwie `ProAndroid3_R08_OknaDialogowe`. Znajdziemy tu również przykładowe aplikacje korzystające z okien dialogowych `TimePicker` i `DatePicker`.

## Podsumowanie

W tym rozdziale zwróciliśmy uwagę na nowego rodzaju wyzwania związane z korzystaniem z okien dialogowych w Androidzie. Pokazaliśmy skutki używania asynchronicznych okien dialogowych oraz zaprezentowaliśmy, w jaki sposób można sobie ułatwić korzystanie z zarządzanych okien dialogowych. Dodatkowo w rozdziale 29. omówiliśmy działanie okien dialogowych opartych na fragmentach, elementach wprowadzonych w wersji 3.0 Androida. Ponieważ interfejs API dialogów jest wprowadzany również do starszych wersji Androida, rozpoczęcie używania okien dialogowych tego typu jest dobrym pomysłem.

# Praca z preferencjami i zachowywanie stanów

W Androidzie — podobnie jak w przypadku wielu innych środowisk SDK — istnieje obsługa preferencji. System może śledzić zarówno preferencje użytkownika aplikacji, jak i preferencje samej aplikacji. Dobrym przykładem jest aplikacja Microsoft Outlook — jej użytkownik może określić preferencje wyświetlania wiadomości e-mail w określony sposób, ale i sama aplikacja posiada pewne domyślne ustawienia, które mogą być konfigurowane przez poszczególnych użytkowników. Istnieje jednak różnica między aplikacją taką jak Microsoft Outlook a aplikacją Androida — nawet jeśli Android teoretycznie śledzi preferencje użytkowników i aplikacji, to nie wprowadza pomiędzy nimi rozróżnienia. Wynika to z faktu, że zazwyczaj aplikacje Androida działają w urządzeniu, które nie jest współdzielone przez kilku użytkowników; ludzie nie współużytkują telefonów komórkowych. Zatem w Androidzie wprowadzono termin **preferencji aplikacji**, odnoszący się zarówno do preferencji użytkownika, jak i do domyślnych preferencji aplikacji.

Osoby widzące po raz pierwszy obsługę preferencji w Androidzie są zazwyczaj bardzo pozytywnie zaskoczone. Android zapewnia wydajną i elastyczną strukturę obsługi preferencji. Dostępne są proste interfejsy API, pozwalające na ukrywanie i przechowywanie preferencji, a także przygotowane interfejsy użytkownika, za pomocą których użytkownik może zmieniać ustawienia preferencji. Dzięki огромнemu potencjału tkwiącego w strukturze preferencji Androida możemy ją również wykorzystywać do bardziej ogólnego zadania przechowywania stanu aplikacji, na przykład w taki sposób, aby aplikacja po zamknięciu i ponownym uruchomieniu otwierała się dokładnie w punkcie przerwania pracy przez użytkownika. Wszystkie wymienione funkcje omówimy w dalszej części rozdziału.

## Badanie struktury preferencji

Zanim zajmiemy się omówieniem struktury preferencji w Androidzie, ustalmy scenariusz, w którym wymagane będzie wykorzystanie preferencji, a następnie zastanówmy się, jak byśmy się zabrali za ich implementację. Założymy, że piszemy aplikację posiadającą funkcję wyszukiwania rozkładu lotów. Przypuśćmy jeszcze, że domyślnie aplikacja wyszukuje loty według kryterium ceny biletu, ale użytkownik może wprowadzić preferencję sortowania wyników pod względem liczby przesiadek lub pod względem konkretnej linii lotniczej. Jak możemy tego dokonać?

### Klasa ListPreference

Oczywiście najpierw musimy dostarczyć interfejs UI, umożliwiający przeglądanie listy dostępnych opcji. Lista taka będzie zawierała przyciski opcji, a domyślny (lub bieżący) wybór będzie już zaznaczony. Rozwiążanie tego problemu dotyczącego struktury preferencji w Androidzie wymaga bardzo niewielkiego nakładu pracy. Najpierw utworzymy plik XML, w którym zostaną opisane preferencje, a następnie wykorzystamy gotową klasę aktywności, która może wyświetlać i przechowywać preferencje. Szczegóły zostały ukazane na listingu 9.1.

**Uwaga!**

Na końcu rozdziału umieściliśmy odnośnik, za pomocą którego można pobierać projekty utworzone specjalnie dla tego rozdziału. Projekty te można importować bezpośrednio do środowiska Eclipse.

**Listing 9.1.** Plik XML preferencji lotów i powiązana z nim klasa aktywności

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/xml/flightoptions.xml -->
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="flight_option_preference"
    android:title="@string/prefTitle"
    android:summary="@string/prefSummary">

    <ListPreference
        android:key="@string/selected_flight_sort_option"
        android:title="@string/listTitle"
        android:summary="@string/listSummary"
        android:entries="@array/flight_sort_options"
        android:entryValues="@array/flight_sort_options_values"
        android:dialogTitle="@string/dialogTitle"
        androiddefaultValue="@string/flight_sort_option_default_value" />

</PreferenceScreen>

package com.androidbook.preferences.sample;

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class FlightPreferenceActivity extends PreferenceActivity
{
```

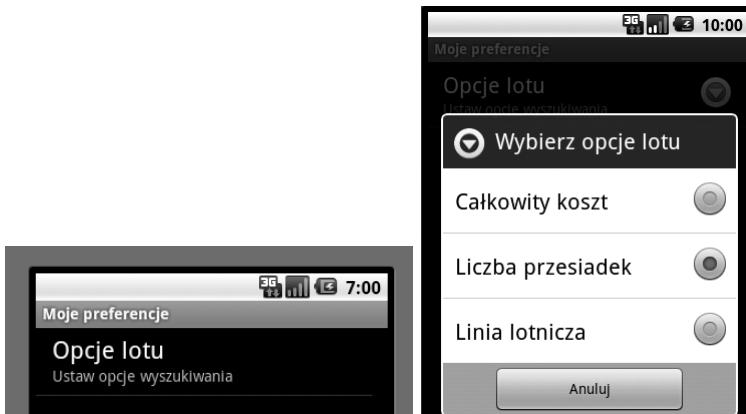
```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    addPreferencesFromResource(R.xml.flightoptions);
}
}

```

Listing 9.1 zawiera fragment kodu XML, który umożliwia ustawienie preferencji lotów. Na listingu tym znalazła się również klasa aktywności, wczytująca plik XML preferencji. Zaczniemy od kodu XML. Android posiada pełną strukturę obsługi preferencji. Oznacza to, że dzięki tej strukturze możemy definiować własne preferencje, wyświetlać je użytkownikowi i zapamiętywać jego decyzje w magazynie danych. Preferencje definiujemy w katalogu */res/xml*. Aby wyświetlić użytkownikowi preferencje, tworzymy klasę aktywności, która rozszerza predefiniowaną klasę Androida *android.preference.PreferenceActivity*, i dodajemy zasób do zbioru zasobów aktywności za pomocą metody *addPreferencesFromResource()*. Wspomniana wcześniej struktura zapewnia obsługę pozostałych funkcjonalności (zapisywanie i przechowywanie preferencji).

W naszym scenariuszu z rozkładami lotów tworzymy plik *flightoptions.xml* i umieszczaemy go w */res/xml/flightoptions.xml*. Generujemy następnie klasę aktywności *FlightPreference* → *Activity*, rozszerzającą klasę *android.preference.PreferenceActivity*. W dalszej kolejności wywołujemy metodę *addPreferencesFromResource()* i przekazujemy jej zasób *R.xml.flightoptions*. Zauważmy, że kod XML zasobów ustawień wskazuje kilka zasobów typu *string*. Aby komplikacja została przeprowadzona pomyślnie, musimy dodać do projektu kilka ciągów znaków. Wkrótce pokażemy, jak tego dokonać. Na razie przyjrzyjmy się utworzonemu na listingu 9.1 interfejsowi użytkownika (rysunek 9.1).



**Rysunek 9.1.** Interfejs UI preferencji lotów

Na rysunku 9.1 pokazano dwa widoki. Widok z lewej strony jest nazywany **ekranem preferencji**, a interfejs z prawej strony nosi nazwę **listy preferencji**. Po kliknięciu pola *Opcje lotu* pojawia się widok *Wybierz opcje lotu* w formie okna dialogowego modalnego, zawierającego przycisk opcji dla każdego ustawienia. Użytkownik wybiera opcję, która natychmiast zostaje zapisana, a widok zostaje zamknięty. Po powrocie do ekranu opcji widok odzwierciedla dokonany wybór.

Kod na listingu 9.1 definiuje klasę *PreferenceScreen*, a następnie tworzy klasę podzieloną *ListPreference*. Klasa *PreferenceScreen* posiada trzy właściwości: *key*, *title* i *summary*.

Właściwość key jest ciągiem znaków, za pomocą którego można się odnosić programistycznie do elementu (podobnie jak w przypadku właściwości android:id); title definiuje nazwę ekranu (*Opcje lotu*); a dzięki właściwości summary opisujemy przeznaczenie ekranu, umieszczone mniejszą czcionką pod nazwą ekranu (w naszym przypadku jest to *Ustaw opcje wyszukiwania*). Dla listy preferencji definiujemy właściwości key, title i summary, a także atrybuty entries, entryValues, dialogTitle i defaultValue. Podsumowaliśmy te atrybuty w tabeli 9.1.

**Tabela 9.1.** Niektóre atrybuty klasy android.preference.ListPreference

Atrybut	Opis
android:key	Nazwa lub klucz opcji (na przykład selected_flight_sort_option).
android:title	Tytuł opcji.
android:summary	Krótki opis opcji.
android:entries	Nazwy elementów listy, które mogą być wybierane w ramach opcji.
android:entryValues	Definiuje klucz lub wartość każdego elementu. Każdy element posiada tekst i wartość. Tekst jest definiowany w atrybucie entries, a wartości — w entryValues.
android:dialogTitle	Nazwa okna dialogowego — atrybut jest używany w przypadku przedstawiania widoku w postaci okna dialogowego modalnego.
android:defaultValue	Domyślna wartość opcji na liście elementów.

Aby nasz przykładowy kod zadziałał, musimy dodać lub zmodyfikować pliki, tak jak pokazano na listingu 9.2.

**Listing 9.2.** Konfigurowanie reszty projektu w naszym przykładzie

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/values/arrays.xml -->
<resources>
<string-array name="flight_sort_options">
    <item>Całkowity koszt</item>
    <item>Liczba przesiadek</item>
    <item>Linia lotnicza</item>
</string-array>
<string-array name="flight_sort_options_values">
    <item>0</item>
    <item>1</item>
    <item>2</item>
</string-array>
</resources>

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/values/strings.xml -->
<resources>
    <string name="app_name">Demonstracja preferencji</string>
    <string name="prefTitle">Moje preferencje</string>
    <string name="prefSummary">Ustaw preferencje opcji lotu</string>
    <string name="flight_sort_option_default_value">1</string>
    <string name="dialogTitle">Wybierz opcje lotu</string>
    <string name="listSummary">Ustaw opcje wyszukiwania</string>
```

```
<string name="listTitle">Opcje lotu</string>
<string name="selected_flight_sort_option">selected_flight_sort_option</string>
<string name="menu_prefs_title">Ustawienia</string>
</resources>

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/menu/mainmenu.xml -->
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item android:id="@+id/menu_prefs"
      android:title="@string/menu_prefs_title"
      />
</menu>

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

<TextView android:text="" android:id="@+id/text1"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          />

</LinearLayout>

// Jest to plik MainActivity.java
import android.app.Activity;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.TextView;

public class MainActivity extends Activity {
    private TextView tv = null;
    private Resources resources;

    /** Wywoywane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        resources = this.getResources();

        tv = (TextView)findViewById(R.id.text1);

        setOptionText();
    }
}
```

```
}

@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.mainmenu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected (MenuItem item)
{
    if (item.getItemId() == R.id.menu_prefs)
    {
        // Uruchamiają na ekranie preferencji.
        Intent intent = new Intent()
            .setClass(this,
                com.androidbook.preferences.sample.FlightPreferenceActivity.class);
        this.startActivityForResult(intent, 0);
    }
    return true;
}

@Override
public void onActivityResult(int requestCode, int resCode, Intent data)
{
    super.onActivityResult(requestCode, resCode, data);
    setOptionText();
}

private void setOptionText()
{
    SharedPreferences prefs =
        PreferenceManager.getDefaultSharedPreferences(this);
    // Jest to alternatywny sposób uzyskania dostępu do współdzielonych zasobów:
    // SharedPreferences prefs = getSharedPreferences(
    //     "com.androidbook.preferences.sample_preferences", 0);
    String option = prefs.getString(
        resources.getString(R.string.selected_flight_sort_option),
        resources.getString(R.string.flight_sort_option_default_value));
    String[] optionText = resources.getStringArray(R.array.flight_sort_options);

    tv.setText("wartość opcji wynosi " + option + " (" +
        optionText[Integer.parseInt(option)] + ")");
}

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.preferences.sample"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".MainActivity">
```

```

        android:label="@string/app_name"
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity android:name=".FlightPreferenceActivity"
        android:label="@string/prefTitle">
    <intent-filter>
        <action android:name=
"com.androidbook.preferences.sample.intent.action.FlightPreferences" />
        <category android:name="android.intent.category.PREFERENCE" />
    </intent-filter>
</activity>

</application>
<uses-sdk android:minSdkVersion="4" />

</manifest>
```

Po wprowadzeniu zmian i uruchomieniu aplikacji ujrzymy prosty komunikat tekstowy *Wartość opcji wynosi 1 (Liczba przesiadek)*. Kliknijmy przycisk *Menu*, a następnie *Settings*, aby przejść do aktywności *PreferenceActivity*. Po zakończeniu wystarczy kliknąć strzałkę cofania, aby natychmiast ujrzeć tekst informujący o wprowadzonych zmianach.

Pierwszym dodanym plikiem jest */res/values/arrays.xml*. W pliku tym znajdują się dwie tablice ciągów znaków, które są nam potrzebne do wprowadzenia możliwości wyboru opcji. Pierwsza tablica przechowuje wyświetlany tekst, w drugiej natomiast są umieszczone wartości, które otrzymamy podczas wywołania metody, oraz wartość przechowywana w pliku XML preferencji. W celach demonstracyjnych wprowadziliśmy wartości indeksów tablic 0, 1 i 2 dla obiektu *flight\_sort\_options\_values*. Możemy wprowadzić każdą wartość usprawniającą działanie aplikacji. Gdyby nasza opcja była natury numerycznej (na przykład początkowa wartość odliczania w liczniku), moglibyśmy wykorzystać wartości 60, 120, 300 i tak dalej. Wartości nie muszą być numeryczne, dopóki mają sens dla projektanta; użytkownik ich nie widzi, chyba że zostaną wyeksponowane. Widoczny jest jedynie tekst zawarty w pierwszej tablicy — *flight\_sort\_options*.

Jak już stwierdziliśmy wcześniej, struktura Androida zapewnia także przechowywanie preferencji. Na przykład po wybraniu przez użytkownika opcji sortowania Android przechowuje wybór w pliku XML, umiejscowionym w katalogu aplikacji */data* (rysunek 9.2).

Name	Size	Date
data		2010-07-20
└── data		2010-07-20
└── anr		2010-07-20
└── app		2010-07-25
└── app-private		2010-07-05
└── backup		2010-07-05
└── dalvik-cache		2010-07-25
└── data		2010-07-25
└── com.androidbook.preferences.sample		2010-07-25
└── lib		2010-07-25
└── shared_prefs		2010-07-25
└── com.androidbook.preferences.sample_preferences.xml	124	2010-07-25

Rysunek 9.2. Ścieżka do zachowanych preferencji aplikacji

Rzeczywista ścieżka pliku wygląda następująco: `/data/data/[NAZWA_PAKIETU]/shared_prefs/[NAZWA_PAKIETU]_preferences.xml`. Na listingu 9.3 widoczny jest plik `com.androidbook.preferences.sample_preferences.xml` z naszego przykładu.

#### **Listing 9.3.** Przykładowe zachowane preferencje

---

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="selected_flight_sort_option">1</string>
</map>
```

---

Widzimy, że w przypadku listy preferencji wartość wybranego elementu jest przechowywana za pomocą atrybutu key listy. Zwrócmy również uwagę, że przechowywana jest **wartość** elementu — nie tekst. Należy tu zwrócić Czytelnikowi uwagę: ponieważ w pliku XML preferencji jest przechowywana jedynie wartość, a nie tekst, to jeżeli będziemy kiedykolwiek aktualniac aplikację i zmieniać tekst opcji lub dodawać elementy do tablic ciągów znaków, wszystkie wartości przechowywane w tym pliku powinny pozostać powiązane z odpowiednim tekstem po aktualizacji. Plik XML preferencji jest zachowywany podczas aktualizowania aplikacji. Innymi słowy, jeżeli przed aktualnieniem aplikacji w tym pliku znajdowała się wartość 1 oznaczająca *Liczba przesiadek*, to po aktualizacji powinna ona oznaczać dokładnie to samo.

Następnym utworzonym przez nas plikiem jest `/res/values/strings.xml`. Dodaliśmy kilka ciągów znaków do tytułów, podsumowań i elementów menu. Umieszczono tu dwa ciągi znaków, na które szczególnie warto zwrócić uwagę. Pierwszy z nich to `flight_sort_option_default_value`. W naszym przykładzie przypisaliśmy wartość 1 argumentowi *Liczba przesiadek*. Zazwyczaj warto wybrać domyślną wartość dla każdej opcji. Jeżeli nie ustalono wartości domyślnej i użytkownik nie wybrał żadnej opcji, metody przekażą wartość `null` dla tej opcji. W takim przypadku kod musi być w stanie przetwarzać puste wartości. Drugim interesującym nas ciągiem znaków jest `selected_flight_sort_option`. Gwoli scisłości, użytkownik nie będzie widział tego ciągu znaków, nie musimy więc umieszczać go w pliku `strings.xml` w celu zapewnienia alternatywnego tekstu w innych językach. Ponieważ jednak ten ciąg znaków stanowi klucz używany przez metody do odczytania wartości, poprzez utworzenie z niego identyfikatora możemy się upewnić w czasie procesu komplikacji, że nie popełniliśmy żadnej literówki w nazwie klucza.

Trzecim dodanym przez nas plikiem jest `/res/menu/mainmenu.xml`. Zakładamy, że chcemy uzyskać dostęp do widoku preferencji poprzez menu, a nie za pomocą przycisku. Plik ten reprezentuje menu naszej aplikacji.

Czwarty plik to `/res/layout/main.xml`. Stanowi on główny interfejs użytkownika naszej aplikacji. Dotychczas omawialiśmy sposób obsługiwania preferencji za pomocą specjalnej klasy aktywności `PreferenceActivity`. Chcemy jednak, aby użytkownik mógł korzystać z preferencji w głównej aktywności, a nie w `PreferenceActivity`. Musimy więc uzyskać w jakiś sposób dostęp do preferencji z poziomu innej aktywności. W naszym przykładzie układem graficznym jest prosta kontrolka `TextView`, wyświetlająca bieżącą wartość preferencji lotu.

Następnie umieszczono kod źródłowy aktywności `MainActivity`. Jest to podstawowa aktywność, która pobiera odniesienie do preferencji i zajmuje się obsługą widoku `TextView`, a następnie wywołuje metodę odczytującą bieżącą wartość opcji, dzięki czemu możliwe jest wstawienie jej do widoku. Konfigurujemy menu oraz jego wywoływanie zwrotne. Wewnątrz wywołania zwrotnego menu uruchamiamy obiekt `Intent` wobec aktywności `FlightPreferenceActivity`.

Uruchomienie intencji wobec preferencji jest najlepszym sposobem na wyświetlenie ekranu preferencji. Aby ją uruchomić, możemy wykorzystać opcje menu lub przycisk. W następnych przykładach nie będziemy powtarzać tego kodu, ale można w nich wykorzystywać tę samą technikę, pod warunkiem że wprowadzimy odpowiednią nazwę aktywności. Po przekazaniu intencji Intent preferencji wywołujemy metodę `setOptionText()` służącą do aktualizacji kontrolki `TextView`.

Istnieją dwa sposoby postępowania z preferencjami:

- W powyższym przykładzie przedstawiliśmy prostszy sposób, polegający na wywołaniu metody `PreferenceManager.getDefaultSharedPreferences(this)`. Argument `this` jest kontekstem pozwalającym na odnalezienie domyślnie współdzielonych preferencji, natomiast sama metoda wykorzystuje nazwę pakietu tego kontekstu do określenia nazwy oraz lokalizacji pliku preferencji. Jest to zresztą ta sama preferencja co utworzona przez aktywność `PreferenceActivity`, ponieważ posiada tę samą nazwę pakietu.
- Drugim sposobem jest wywołanie metody `getSharedPreferences()` i przekazanie argumentów nazwy pliku oraz trybu. Rozwiążanie to jest widoczne na listingu 9.2, jednak ten fragment kodu jest nieaktywny, gdyż został objęty znakami komentarza. Warto zauważyć, że wymieniono jedynie podstawowy człon nazwy pliku, pominięto natomiast ścieżkę do niego oraz jego rozszerzenie. Argument trybu pozwala nam na określanie uprawnień do naszego pliku XML. W poprzednim przykładzie argument trybu był zbędny, ponieważ plik XML został utworzony wyłącznie w obrębie aktywności `PreferenceActivity`, co sprawia, że domyślne uprawnienia otrzymują wartość `MODE_PRIVATE` (np. zero). Argumenty trybu zostaną omówione w podrozdziale dotyczącym zachowywania stanu.

Następnie z poziomu metody `setOptionText()`, uwzględniając odniesienie do preferencji, wywołujemy odpowiednie metody służące do odczytywania wartości. W naszym przypadku wywołujemy metodę `getString()`, ponieważ z preferencji uzyskujemy wartość typu `string`. Pierwszym argumentem jest ciąg znaków, reprezentujący klucz opcji. Stwierdziliśmy wcześniej, że stosowanie identyfikatorów chroni programistę przed popełnieniem literówki podczas komplikacji. W miejsce pierwszego argumentu moglibyśmy również po prostu wstawić ciąg znaków `selected_flight_sort_option`, gdyż zależy nam na utworzeniu jak najmniejszej i jak najszyszej aplikacji. Drugi argument służy do określenia wartości domyślnej, w przypadku gdy nie można jej znaleźć w pliku XML preferencji. Gdy aplikacja zostaje uruchomiona po raz pierwszy, nie istnieje jeszcze plik preferencji, zatem bez określenia wartości drugiego argumentu zawsze za pierwszym razem będzie odsyłana pusta wartość. Dzieje się tak, nawet jeśli zdefiniowano domyślną wartość w specyfikacji `ListPreference`, umieszczonej w pliku `flightoptions.xml`. W naszym przykładzie ustanowiliśmy w pliku domyślną wartość w pliku XML za pomocą identyfikatora zasobu, więc kod w metodzie `setOptionText()` może posłużyć do odczytania tego identyfikatora w celu otrzymania domyślnej wartości. Zwróćmy uwagę, że gdybyśmy nie utworzyli identyfikatora domyślnej wartości, jej odczyt z obiektu `ListPreference` byłby znacznie utrudniony. Dzięki temu, że identyfikator zasobu jest wykorzystywany zarówno przez plik XML, jak i kod Java, wartość domyślną możemy zmieniać tylko w jednym miejscu (mamy na myśli plik `strings.xml`).

Poza prezentowaniem wartości preferencji wyświetlamy także jej treść. W naszym przykładzie korzystamy ze skrótu, ponieważ użyliśmy indeksu tablicy dla wartości w obiekcie `flight_sort_options_values`. Dzięki prostej konwersji wartości na typ `int` wiemy, który ciąg znaków ma zostać odczytany z argumentu `flight_sort_options`. Gdybyśmy użyli dla

argumentu `flight_sort_options_values` innego zestawu wartości, musielibyśmy określić indeks elementu stanowiącego naszą preferencję, a następnie wykorzystać ten indeks do odczytania tekstu tej preferencji z tablicy `flight_sort_options`.

Ostatnim plikiem użyтыm w naszym przykładzie jest `AndroidManifest.xml`. Teraz w aplikacji istnieją dwie aktywności, a zatem potrzebujemy dwóch znaczników aktywności. Pierwszy z nich określa standardową aktywność kategorii `LAUNCHER`. Drugi znacznik jest przeznaczony dla aktywności `PreferenceActivity`, a więc ustanawiamy nazwę działania zgodnie z konwencją dla intencji oraz kategorii `PREFERENCE`. Prawdopodobnie nie chcemy, aby aktywność `PreferenceActivity` pojawiała się wraz z pozostałymi aplikacjami na ekranie startowym Androida, dlatego nie wyznaczyliśmy jej do kategorii `LAUNCHER`. W przypadku dodawania kolejnych ekranów preferencji należy wprowadzać podobne zmiany w pliku `AndroidManifest.xml`.

Zademonstrowaliśmy jeden sposób odczytywania domyślnej wartości preferencji za pomocą kodu. Istnieje alternatywne rozwiązanie, nieco bardziej eleganckie. Moglibyśmy w metodzie `onCreate()` wykonać następującą czynność:

```
PreferenceManager.setDefaultValues(this, R.xml.flightoptions, false);
```

Następnie można odczytać wartość opcji wewnętrz metody `setOptionText()`:

```
String option = prefs.getString(  
    resources.getString(R.string.selected_flight_sort_option), null);
```

W pierwszym wywołaniu wykorzystano plik `flightoptions.xml` do znalezienia domyślnych wartości oraz wygenerowania za ich pomocą pliku XML preferencji. Jeżeli w pamięci obecne jest już wystąpienie interfejsu `SharedPreferences`, zostanie ono również zaktualizowane. Drugie wywołanie odnajdzie teraz wartość opcji `selected_flight_sort_option`, ponieważ najpierw zostały załadowane domyślne wartości.

Jeżeli zajrzymy do folderu `shared_prefs` po pierwszym uruchomieniu aplikacji, zauważymy, że został utworzony plik XML preferencji, nawet jeśli ekran preferencji nie został wyświetlony. Będzie widoczny również inny plik — `_has_set_default_values.xml`. Aplikacja otrzymuje w ten sposób informację, że utworzono i wypełniono domyślnymi wartościami plik XML preferencji. Trzeci argument metody `setDefaultValues() — false`, wskazuje, że domyślne wartości mają zostać wstawione do pliku XML preferencji tylko wtedy, jeśli wcześniej ich tam nie było. Jeżeli ustawilibyśmy wartość `true`, plik ten zawsze byłby nadpisywany wartościami domyślnymi. Android zapamiętuje tę informację na cały czas istnienia nowego pliku XML. Jeżeli użytkownik wprowadził nowe wartości preferencji przy ustawionej wartości `false` tego trzeciego argumentu, nie zostaną one przywrócone do wartości domyślnych podczas kolejnego uruchomienia aplikacji. Zauważmy jeszcze, że nie musimy wprowadzać domyślnej wartości do wywołania metody `getString()`, ponieważ powinniśmy ją zawsze uzyskiwać z pliku XML preferencji.

Aby uzyskać odniesienie do preferencji z wnętrza aktywności rozszerzającej klasę `PreferenceActivity`, możemy dokonać tego w następujący sposób:

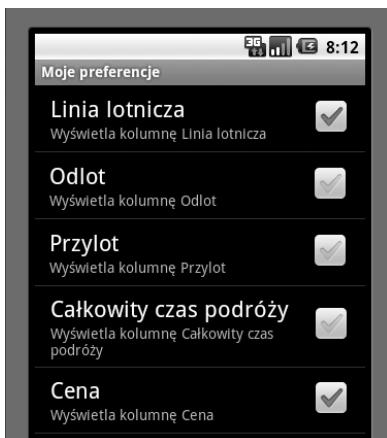
```
SharedPreferences prefs = getPreferenceManager().getDefaultsSharedPreferences(this);
```

Pokazaliśmy, w jaki sposób można wykorzystać widok `ListPreference`; teraz przyjrzymy się innym elementom interfejsu użytkownika, stosowanym w strukturze preferencji w Androidzie. Przeanalizujmy mianowicie widoki `CheckBoxPreference`, `EditTextPreference` i `RingtonePreference`.

## Widok CheckBoxPreference

Pokazaliśmy, że preferencja `ListPreference` wyświetla listę jako element swego interfejsu użytkownika. W analogiczny sposób preferencja `CheckBoxPreference` wyświetla widżet pola wyboru w postaci składnika swego interfejsu UI.

Załóżmy, że w ramach rozbudowania naszej aplikacji wyszukującej loty chcemy użytkownikowi umożliwić wybranie kolumn przed wyświetlaniem zestawu wyników. Preferencja `CheckBoxPreference` wyświetla spis dostępnych kolumn i umożliwia użytkownikowi wybranie pożądanych kolumn poprzez zaznaczenie odpowiedniego pola wyboru. Interfejs użytkownika dla tego przykładu jest przedstawiony na rysunku 9.3, a zawartość pliku XML preferencji została umieszczona na listingu 9.4.



Rysunek 9.3. Interfejs UI preferencji pola wyboru

**Listing 9.4.** Zastosowanie widoku CheckBoxPreference

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/xml/chkbox.xml -->
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="flight_columns_pref"
    android:title="Preferencje wyszukiwania lotu"
    android:summary="Wybierz kolumny wyświetlane w wynikach wyszukiwania">
    <CheckBoxPreference
        android:key="show_airline_column_pref"
        android:title="Linia lotnicza"
        android:summary="Wyświetla kolumnę Linia lotnicza" />
    <CheckBoxPreference
        android:key="show_departure_column_pref"
        android:title="Odlot"
        android:summary="Wyświetla kolumnę Odlot" />
    <CheckBoxPreference
        android:key="show_arrival_column_pref"
        android:title="Przylot"
        android:summary="Wyświetla kolumnę Przylot" />
    <CheckBoxPreference
        android:key="show_total_travel_time_column_pref"
```

```
        android:title="Całkowity czas podróży"
        android:summary="Wyświetla kolumnę Całkowity czas podróży" />
<CheckBoxPreference
    android:key="show_price_column_pref"
    android:title="Cena"
    android:summary="Wyświetla kolumnę Cena" />

</PreferenceScreen>

// CheckBoxPreferenceActivity.java

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class CheckBoxPreferenceActivity extends PreferenceActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.chkbox);
    }
}
```

---

Na listingu 9.4 został ukazany plik XML preferencji, *chkbox.xml*, oraz prosta klasa aktywności wczytująca ten plik za pomocą metody `addPreferencesFromResource()`. Jak widać, w interfejsie użytkownika znalazło się pięć pól wyboru, każde reprezentowane przez węzeł `CheckBoxPreference` w pliku XML. Każde z tych pól posiada również wartość `key`, która — jak można było się spodziewać — jest ostatecznie wykorzystywana do przechowywania stanu elementu interfejsu użytkownika w trakcie procesu zapisywania wprowadzonych zmian preferencji. Dzięki widokowi `CheckBoxPreference` zostaje ona zapisana po zmianie stanu preferencji. Innymi słowy, jeżeli użytkownik zaznacza kontrolkę preferencji lub usuwa jej zaznaczenie, jej stan zostaje zapisany. Listing 9.5 przedstawia magazyn danych preferencji dla naszego przykładu.

---

**Listing 9.5.** Magazyn danych dla preferencji w postaci pola wyboru

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <boolean name="show_total_travel_time_column_pref" value="false" />
    <boolean name="show_price_column_pref" value="true" />
    <boolean name="show_arrival_column_pref" value="false" />
    <boolean name="show_airline_column_pref" value="true" />
    <boolean name="show_departure_column_pref" value="false" />
</map>
```

---

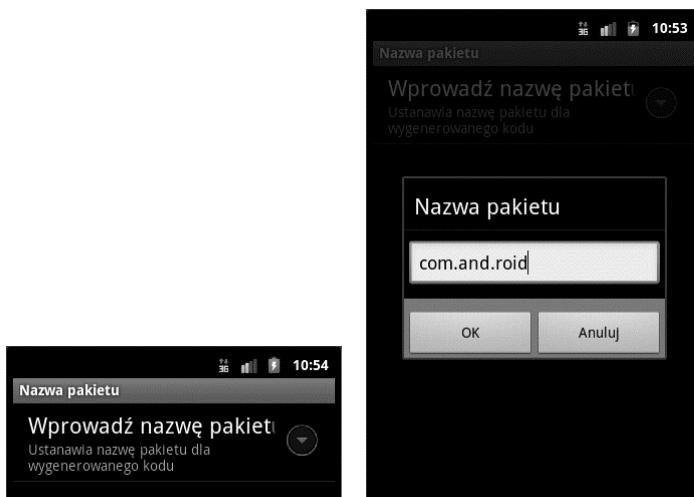
Ponownie widać, że każda preferencja zostaje zachowana poprzez atrybut `key`. Typem danych widoku `CheckBoxPreference` jest `boolean`, przyjmujący wartość `true` lub `false`: wartość `true` wskazuje zaznaczenie preferencji, wartość `false` ma przeciwnie znaczenie. Aby odczytać wartość z pola wyboru preferencji, musielibyśmy uzyskać dostęp do współdzielonej preferencji, a następnie wywołać metodę `getBoolean()` i przekazać jej atrybut `key` preferencji:

```
boolean option = prefs.getBoolean("show_price_column_pref", false);
```

Inną pozytyczną funkcją widoku CheckBoxPreference jest możliwość zmiany treści podsumowania w zależności od tego, czy pole wyboru jest zaznaczone, czy nie. W tym przypadku stosowane są atrybuty języka XML summaryOn i summaryOff. Przyjrzyjmy się teraz kontrolce EditTextPreference.

## Widok EditTextPreference

W Androidzie jest również dostępna preferencja umożliwiająca pisanie tekstu, zwana EditTextPreference. Użytkownik, zamiast zaznaczyć wybraną opcję, może wpisać własną treść. Założymy, że posiadamy aplikację służącą do generowania kodu Java. Jednym z ustawień preferencji tej aplikacji może być nazwa domyślnego pakietu, wyznaczonego dla generowanych klas. Chcemy wyświetlić użytkownikowi pole tekstowe i pozwolić mu na wpisanie nazwy takiego pakietu. Na rysunku 9.4 został zaprezentowany interfejs użytkownika takiej aplikacji, zaś na listingu 9.6 pokazaliśmy kod XML.



Rysunek 9.4. Zastosowanie widoku EditTextPreference

**Listing 9.6.** Przykładowy widok EditTextPreference

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/xml/packagepref.xml -->
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="package_name_screen"
    android:title="Nazwa pakietu"
    android:summary="Wprowadź nazwę pakietu">

    <EditTextPreference
        android:key="package_name_preference"
        android:title="Wprowadź nazwę pakietu"
        android:summary="Ustanawia nazwę pakietu dla wygenerowanego kodu"
        android:dialogTitle="Nazwa pakietu" />
</PreferenceScreen>

// EditTextPreferenceActivity.java
```

```

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class EditTextPreferenceActivity extends PreferenceActivity{

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.packagepref);
    }
}

```

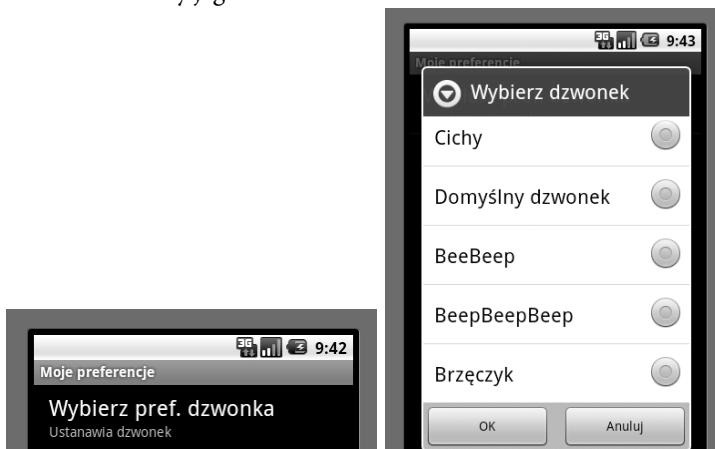
Widzimy, że na listingu 9.6 zdefiniowano klasę PreferenceScreen, zawierającą widok EditTextPreference. W wygenerowanym interfejsie użytkownika klasa PreferenceScreen jest widoczna po lewej stronie, a widok EditTextPreference — po prawej (rysunek 9.4). Po kliknięciu przez użytkownika opcji *Wprowadź nazwę pakietu* pojawi się okno dialogowe, w którym można wpisać odpowiednią nazwę. Kliknięcie przycisku *OK* spowoduje zapisanie ustawienia w magazynie preferencji.

Podobnie jak ma to miejsce w przypadku innych preferencji, możemy odczytać widok EditTextPreference z poziomu klasy aktywności za pomocą atrybutu key tej preferencji. Po wczytaniu preferencji EditTextPreference możemy konfigurować widok EditText poprzez wywołanie metody getEditText() — jeśli na przykład chcemy wprowadzić sprawdzanie wartości wpisanej przez użytkownika, jej przetwarzanie wstępne lub przetwarzanie końcowe. Aby odczytać treść widoku EditTextPreference, wykorzystujemy po prostu metodę getText().

Przyjrzyjmy się teraz widokowi RingtonePreference struktury preferencji.

## Widok RingtonePreference

Widok RingtonePreference służy do obsługi dzwonków. Jest on wstawiany do aplikacji, w przypadku gdy użytkownik matrzymać możliwość wyboru dzwonka w formie preferencji. Rysunek 9.5 ilustruje przykładowy interfejs UI widoku RingtonePreference, a na listingu 9.7 został wstawiony jego kod XML.



Rysunek 9.5. Przykładowy interfejs UI widoku RingtonePreference

**Listing 9.7.** Definiowanie preferencji RingtonePreference

```

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/xml/ringtone.xml -->
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:key="ringtone_option_preference"
        android:title="Moje preferencje"
        android:summary="Wybierz preferencje dzwonka">
    <RingtonePreference
        android:key="ring_tone_pref"
        android:title="Wybierz preferencję dzwonka"
        android:showSilent="true"
        android:ringtoneType="alarm"
        android:summary="Ustanawia dzwonek" />
</PreferenceScreen>

// RingtonePreferenceActivity.java

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class RingtonePreferenceActivity extends PreferenceActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.ringtone);
    }
}

```

Kiedy użytkownik kliknie przycisk *Wybierz preferencję dzwonka*, zostanie wyświetlony widok ListPreference zawierający dzwonki zapisane w urządzeniu (rysunek 9.5). W widoku tym można wybrać dzwonek, a następnie kliknąć przycisk *OK* lub *Anuluj*. Po wybraniu opcji *OK* dane zostaną zapisane w magazynie preferencji. Zwróćmy uwagę, że w przypadku dzwonków wartością przechowywaną w magazynie preferencji jest identyfikator URI danego dzwonka — chyba że zostanie wybrana preferencja *Cichy*, która w miejsce przechowywanej wartości wstawia pusty ciąg znaków. Przykładowy identyfikator URI wygląda następująco:

```
<string name="ring_tone_pref">content://media/internal/audio/media/26</string>
```

**Uwaga!**

Jeżeli emulator jest skąpo wyposażony w dzwonki, można dodać je samemu. W tym celu wystarczy skopiować pliki muzyczne na kartę SD, następnie uruchomić aplikację Music Player, zaznaczyć plik muzyczny, po czym kliknąć przycisk *Menu* i wybrać opcję *Use as ringtone*. Mechanizm kopiowania plików na kartę SD został omówiony w rozdziale 19.

Przedstawiony na listingu 9.7 widok RingtonePreference korzysta z takiego samego algorytmu jak pozostałe preferencje. Różnica polega na tym, że ustawiamy kilka różnych atrybutów, w tym *showSilent* i *ringtoneType*. Atrybut *showSilent* służy do wstawienia wyciszzonego dzwonka na listę, a *ringtoneType* do ograniczania typów wyświetlanych dzwonków. Dla tej właściwości dostępne są wartości *ringtone*, *notification*, *alarm* i *all*.

## Organizowanie preferencji

Struktura preferencji pozwala na organizowanie preferencji w kategorie. Jeśli nasza aplikacja pozwala na ustawienie wielu preferencji, możemy zbudować widok przedstawiający ich wysoko-poziomowe kategorie. Użytkownicy mogliby wtedy wyświetlać każdą z tych kategorii w celu przeglądania preferencji umieszczonych w danej grupie i zarządzania nimi.

Istnieją dwa sposoby implementacji takiej struktury. Możemy albo wprowadzić zagnieżdżone elementy PreferenceScreen wewnątrz nadrzednej klasy PreferenceScreen, albo w podobny sposób wykorzystać elementy PreferenceCategory. Pierwszy z wymienionych sposobów, polegający na grupowaniu preferencji za pomocą zagnieżdżonych elementów PreferenceScreen, został zaprezentowany na rysunku 9.6 i listingu 9.8.



Rysunek 9.6. Tworzenie grup preferencji poprzez zagnieżdżenie elementów PreferenceScreen

**Listing 9.8.** Zagnieżdżanie elementów PreferenceScreen umożliwiające organizowanie preferencji

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="using_categories_in_root_screen"
    android:title="Kategorie"
    android:summary="Stosowanie kategorii preferencji">

    <PreferenceScreen
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:key="meats_screen"
        android:title="Mięso"
        android:summary="Preferencje odnoszące się do mięsa">

        <CheckBoxPreference
            android:key="fish_selection_pref"
            android:title="Ryba"
            android:summary="Ryby są bardzo zdrowe" />
        <CheckBoxPreference
            android:key="chicken_selection_pref"
            android:title="Kurczak"
            android:summary="Popularny gatunek drobiu" />
        <CheckBoxPreference
            android:key="lamb_selection_pref"
            android:title="Jagnię"
            android:summary="Jagnię jest młodą owcą" />

    </PreferenceScreen>
<PreferenceScreen>
```

```

    xmlns:android="http://schemas.android.com/apk/res/android"
        android:key="vegi_screen"
        android:title="Warzywa"
        android:summary=" Preferencje odnoszące się do warzyw">
<CheckBoxPreference
        android:key="tomato_selection_pref"
        android:title="Pomidor "
        android:summary="W rzeczywistości jest to owoc" />
<CheckBoxPreference
        android:key="potato_selection_pref"
        android:title="Ziemniak"
        android:summary="Moje ulubione warzywo" />
</PreferenceScreen>
</PreferenceScreen>

```

Widok z lewej strony na rysunku 9.6 prezentuje dwa ekranы preferencji, jeden zatytułowany *Mięso*, a drugi noszący nazwę *Warzywa*. Kliknięcie danej grupy spowoduje wyświetlenie jej elementów. Na listingu 9.8 pokazaliśmy, w jaki sposób tworzy się zagnieżdżone ekranы.

Grupy pokazane na rysunku 9.6 zostały utworzone poprzez zagnieżdżenie elementów PreferenceScreen wewnątrz nadzędnej klasy PreferenceScreen. Organizowanie w ten sposób preferencji jest korzystne, w przypadku gdy istnieje wiele preferencji, a nie chcemy, aby użytkownicy musieli przewijać ekran w celu znalezienia jednej z nich. Jeżeli w aplikacji nie ma zbyt wielu preferencji, a mimo to należy utworzyć ich wysokopoziomowe kategorie, można zastosować drugą metodę, związaną z obiektem PreferenceCategory. Szczegóły zostały zaprezentowane na rysunku 9.7 i listingu 9.9.



**Rysunek 9.7.** Zastosowanie obiektu PreferenceCategory do organizowania preferencji

#### **Listing 9.9.** Tworzenie kategorii preferencji

```

<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:key="using_categories_in_root_screen"
        android:title="Kategorie"

```

```
        android:summary="Zastosowanie kategorii preferencji">

<PreferenceCategory
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:key="meats_category"
        android:title="Mięso"
        android:summary="Preferencje odnoszące się do mięsa">

    <CheckBoxPreference
        android:key="fish_selection_pref"
        android:title="Ryba"
        android:summary="Ryby są bardzo zdrowe" />
    <CheckBoxPreference
        android:key="chicken_selection_pref"
        android:title="Kurczak"
        android:summary="Popularny gatunek drobiu" />
    <CheckBoxPreference
        android:key="lamb_selection_pref"
        android:title="Jagnię"
        android:summary="Jagnię jest młodą owcą" />

</PreferenceCategory>
<PreferenceCategory
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:key="vegi_category"
        android:title="Warzywa"
        android:summary=" Preferencje odnoszące się do warzyw">
    <CheckBoxPreference
        android:key="tomato_selection_pref"
        android:title="Pomidor "
        android:summary="W rzeczywistości jest to owoc" />
    <CheckBoxPreference
        android:key="potato_selection_pref"
        android:title="Ziemniak"
        android:summary="Moje ulubione warzywo" />

</PreferenceCategory>
</PreferenceScreen>
```

Na rysunku 9.7 pokazano grupy wykorzystane również w poprzednim przykładzie. Teraz jednak grupy te są zorganizowane w kategorie preferencji. Jedyna różnica pomiędzy kodem XML z listingu 9.9 a kodem XML z listingu 9.8 polega na utworzeniu obiektu `PreferenceCategory` dla zagnieżdżonych ekranów zamiast zagnieżdżenia elementów `PreferenceScreen`.

## Programowe zarządzanie preferencjami

Nie trzeba tłumaczyć, że może zaistnieć potrzeba uzyskania dostępu do rzeczywistych kontrolek preferencji w sposób programowy. Przykładowo należy zapewnić wprowadzanie parametrów `entry` i `entryValue` do klasy `ListPreference` w trakcie działania aplikacji. Możemy definiować kontrolki preferencji i uzyskiwać do nich dostęp w podobny sposób jak w przypadku plików układu graficznego i aktywności. Aby na przykład uzyskać dostęp do listy preferencji zdefiniowanej na listingu 9.1, musielibyśmy wywołać metodę `findPreference()` aktywności `PreferenceActivity`,

przekazując wartość właściwości key (zwróćmy uwagę na podobieństwo do metody `find→ViewById()`). Następnie można by oddać kontrolę obiekowi `ListPreference` i zająć się pracą z kontrolką. Jeśli na przykład chcemy ustawić wpisy widoku `ListPreference`, wywołujemy metodę `setEntries()` i tak dalej. Na listingu 9.10 pokazujemy przykładową implementację tego mechanizmu za pomocą kodu konfigurującego preferencje.

**Listing 9.10.** Konfigurowanie wartości widoku `ListPreference` w sposób programistyczny

---

```
public class FlightPreferenceActivity extends PreferenceActivity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.flightoptions);

        ListPreference listpref = (ListPreference) findPreference(
            "selected_flight_sort_option");

        listpref.setEntryValues(new String[] {"0","1","2"});
        listpref.setEntries(new String[] {"Jedzenie", "Poczekalnia",
                                         "Program Frequent Flyer"});
    }
}
```

---

## Zapisywanie stanu za pomocą preferencji

Preferencje w znakomity sposób umożliwiają użytkownikom dostosowywanie aplikacji do swoich potrzeb, jednak struktura preferencji pozwala jeszcze na inne zastosowania. Jeśli aplikacja ma zapamiętywać wartości danych pomiędzy jej wywołaniami, preferencje stanowią jedno ze stosowanych w tym celu rozwiązań. Omawialiśmy wcześniej rolę dostawców treści w utrzymywaniu danych. Moglibyśmy również wykorzystać w podobny sposób niestandardowe pliki przechowywane na karcie SD. Możemy także wprowadzić pliki i kod preferencji.

W klasie `Activity` znajduje się metoda `getPreferences(int mode)`. W rzeczywistości wywołuje ona po prostu metodę `getSharedPreferences()`, której argumentami są nazwa aktywności oraz tryb uprawnień. W wyniku jej działania otrzymujemy plik preferencji przeznaczony dla danej aktywności, w którym możemy przechowywać dane o tej aktywności pomiędzy jej wywołaniami. Prosty przykład zastosowania w taki sposób preferencji możemy prześledzić na listingu 9.11.

**Listing 9.11.** Stosowanie preferencji do przechowywania stanu aktywności

---

```
final String INITIALIZED = "initialized";
SharedPreferences myPrefs = getPreferences(MODE_PRIVATE);

boolean hasPreferences = myPrefs.getBoolean(INITIALIZED, false);
if(hasPreferences) {
    Log.v("Preferences", "Byliśmy już wcześniej wywołane");
    // Odczytuje w razie potrzeby inne wartości z pliku preferencji...
    someString = myPrefs.getString("someString", "");
}
```

---

```
else {
    Log.v("Preferences", "Wywołane po raz pierwszy");
    // Ustanawia początkowe wartości dla danych,
    // które znajdują się w pliku preferencji
    someString = "jakaś domyślna wartość";
}
// Gdy wartości będą już gotowe do wypisania
Editor editor = myPrefs.edit();
editor.putBoolean(INITIALIZED, true);
editor.putString("someString", someString);
// W razie potrzeby zapisuje inne wartości
editor.commit();
```

---

Powyższy kod otrzymuje dostęp do odniesienia do preferencji należących do danej aktywności i sprawdza istnienie dwuwartościowej „preferencji”, nazwanej `initialized`. Piszemy „preferencja” w cudzysłowie, ponieważ użytkownik nie zobaczy ani nie zmodyfikuje jej wartości; jest to jedynie wartość, którą chcemy przechować w pliku preferencji do następnego wywołania aplikacji. Jeżeli otrzymamy tę wartość, oznacza to, że plik preferencji istnieje, więc nasza aplikacja musiała zostać wcześniej wywołana. Możemy wtedy odczytać pozostałe wartości znajdujące się w tym pliku.

Aby zapisać wartości w pliku preferencji, musimy najpierw uzyskać ich klasę `Editor`. Możemy następnie wstawić wartości do preferencji i zapisać zmiany po zakończeniu ich wprowadzania. Zwróćmy uwagę, że poza wzrokiem użytkownika Android zarządza obiektem `SharedPreferences`, który jest w istocie współdzielony (ang. *shared*). W idealnym przypadku w jednej chwili powinna być aktywna tylko jedna klasa `Editor`. Niezwykle istotne jest jednak, aby wywoływać metodę `commit()` w celu ciągłego aktualizowania interfejsu `Shared Preferences` i pliku XML.

W każdej chwili możemy uzyskiwać dostęp do wartości, zapisywać i przypisywać je w pliku preferencji. Wśród możliwych zastosowań są takie, jak zapisywanie wyników gry lub rejestrowanie daty ostatniego uruchomienia aplikacji. Możemy również wywoływać metodę `getShared Preferences()` zawierającą inne nazwy w celu zarządzania osobnymi zestawami preferencji, wszystko w zakresie jednej aplikacji, a nawet aktywności.

Do tej pory korzystaliśmy w naszych przykładach z trybu `MODE_PRIVATE`. Pozostałe dostępne tryby to `MODE_WORLD_READABLE` oraz `MODE_WORLD_WRITEABLE`. Za pomocą tych trybów ustawiamy odpowiednie uprawnienia dostępu w trakcie tworzenia pliku preferencji. Ponieważ pliki preferencji są przechowywane wewnętrz katalogu danych aplikacji, a zatem nie są dostępne dla innych aplikacji, możemy tu zastosować jedynie tryb `MODE_PRIVATE`.

## Odbośniki

Poniższy odnośnik może się przydać Czytelnikom, którzy zechcą zapoznać się lepiej z omówioną w tym rozdziale tematyką:

- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — z tego adresu możemy pobrać projekty utworzone z myślą o niniejszej książce. Plik ten zawiera wszystkie przykłady omówione w tym rozdziale, umieszczone w oddzielnich katalogach. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R09\_Preferencje*. Dostępny jest tu także plik *Czytaj.TXT*, stanowiący dokładną instrukcję importowania projektów do środowiska Eclipse.

## **Podsumowanie**

W tym rozdziale omówiliśmy zarządzanie preferencjami w Androidzie. Pokazaliśmy, w jaki sposób można korzystać z widoków `ListPreference`, `CheckBoxPreference`, `EditTextPreference` i `RingtonePreference`. Omówiliśmy także techniki organizowania preferencji w grupy oraz programowy sposób modyfikowania preferencji. Na koniec wyjaśniliśmy, w jaki sposób można wykorzystać, poprzez przywołania, strukturę preferencji w procesie zapisywania i odczytywania informacji z aktywności.



# Analiza zabezpieczeń i uprawnień

W niniejszym rozdziale zajmiemy się modelem zabezpieczeń aplikacji. Kwestie bezpieczeństwa w systemie Android mają fundamentalne znaczenie — są brane pod uwagę na wszystkich etapach cyklu życia aplikacji, począwszy od rozoważań na temat polityki czasu projektowania aplikacji, na testowaniu aplikacji w środowisku uruchomieniowym skończywszy. Czytelnicy poznają architekturę zabezpieczeń i dowiedzą się, w jaki sposób tworzyć bezpieczne aplikacje.

Zapoznajmy się z modelem zabezpieczeń w Androidzie.

## Model zabezpieczeń w Androidzie

W pierwszym podrozdziale przyjrzymy się zabezpieczeniom na poszczególnych etapach wdrażania oraz uruchamiania aplikacji. Na etapie wdrażania aplikacje muszą zostać podpisane za pomocą cyfrowego certyfikatu, zanim zostaną zainstalowane na urządzeniu. Na etapie uruchamiania każda aplikacja działa wewnątrz oddzielnego procesu, który posiada unikalny i stały identyfikator użytkownika (przydzielony podczas instalacji). Zostaje w ten sposób utworzona granica wokół procesu, uniemożliwiająca uzyskanie bezpośredniego dostępu przez jedną aplikację do danych innej aplikacji. Ponadto w Androidzie zdefiniowano deklaracyjny model uprawnień, służący do ochrony wrażliwych funkcji (na przykład listy kontaktów).

Omówimy te zagadnienia w kilku następnych podrozdziałach. Zanim jednak do nich przejdziemy, musimy nakreślić pewne pojęcia dotyczące zabezpieczeń, do których będziemy się później odnosić.

## Przegląd pojęć dotyczących zabezpieczeń

Android wymaga, żeby aplikacje były podpisywane certyfikatami cyfrowymi. Jedną z zalet takiej polityki jest możliwość zaktualizowania aplikacji jedynie do wersji opublikowanej przez oryginalnego autora. Na przykład jeżeli my — autorzy tej

książki — opublikujemy aplikację, Czytelnik nie będzie mógł jej zaktualizować swoją własną wersją (chyba że w jakiś sposób uzyska nasz certyfikat). A zatem, co to znaczy, że aplikacja musi zostać podpisana? I jak wygląda proces jej podpisywania?

Dokonuje się tego za pomocą certyfikatu cyfrowego. **Certyfikat cyfrowy** jest artefaktem zawierającym informacje o producencie, takie jak nazwa firmy, adres i tak dalej. Wśród najważniejszych pojęć związanych z certyfikatem cyfrowym można wskazać podpis oraz klucz publiczny i prywatny. Klucz publiczny i prywatny jest również nazywany **parą kluczy**. Zauważmy, że chociaż używamy tu kluczy do podpisywania plików *.apk*, są one również wykorzystywane w innych celach (na przykład w komunikacji szyfrowanej). Certyfikat cyfrowy można otrzymać od zaufanego wydawcy certyfikatów (ang. *certificate authority* — CA), istnieje także możliwość samodzielnego wygenerowania własnego certyfikatu za pomocą takiego narzędzia, jak omówiona w dalszej części rozdziału aplikacja *keytool*. Certyfikaty cyfrowe są przechowywane w **magazynach kluczy**. Magazyn kluczy zawiera listę certyfikatów cyfrowych, z których każdy posiada alias, służący jako odniesienie do tego certyfikatu.

Podpisanie aplikacji w systemie Android wymaga trzech elementów: certyfikatu cyfrowego, pliku *.apk* oraz aplikacji, która zastosuje podpis cyfrowy z certyfikatu dla pliku *.apk*. Jak się wkrótce okaże, taką aplikacją może być bezpłatne narzędzie *jarsigner*, dostępne w zestawie Java Development Kit. Jest to program wiersza poleceń, który potrafi podpisać plik *.jar* za pomocą certyfikatu cyfrowego.

Przejdźmy teraz do tematu podpisywania pliku *.apk* za pomocą certyfikatu cyfrowego.

## Podpiswanie wdrażanych aplikacji

Żeby zainstalować aplikację systemu Android w urządzeniu, musimy najpierw podpisać plik pakietu Android (*.apk*) za pomocą certyfikatu cyfrowego. Jednak certyfikat można wygenerować samodzielnie — nie ma konieczności zakupu certyfikatu od wydawcy, takiego jak firma VeriSign.

Podpiswanie wdrażanej aplikacji obejmuje trzy etapy. Pierwszym krokiem jest wygenerowanie certyfikatu za pomocą aplikacji *keytool* (lub podobnego narzędzia). W drugim etapie wykorzystujemy na przykład narzędzie *jarsigner* do podpisania pliku *.apk* wygenerowanym certyfikatem. W ostatnim etapie następuje przypisanie fragmentów aplikacji do segmentów pamięci, dzięki czemu podczas działania programu pamięć jest wykorzystywana efektywniej. Warto zwrócić uwagę, że podczas etapu projektowania wtyczka ADT automatycznie obsługuje wszystkie wymienione czynności: podpisuje plik *.apk* i przydziela pamięć jeszcze przed wdrożeniem aplikacji na emulatorze.

### Wygenerowanie samoistnie podписанego certyfikatu za pomocą narzędzia *keytool*

Aplikacja *keytool* zarządza bazą danych kluczy prywatnych oraz powiązanych z nimi certyfikatów X.509 (standard certyfikatów cyfrowych). Jest dostępna w zestawie JDK, dokładniej w jego katalogu *bin*. Po wykonaniu omówionych w rozdziale 2. czynności modyfikowania zmiennej środowiskowej PATH katalog *bin* zestawu JDK powinien być częścią tej zmiennej.

W niniejszym podrozdziale pokażemy, w jaki sposób można wygenerować magazyn kluczy zawierający jeden wpis służący do podpisania pliku *.apk*. Żeby utworzyć taki wpis, należy wykonać następujące czynności:

1. Utwórz folder, w którym będzie przechowywany magazyn kluczy, na przykład *c:\android\release\*.

2. Otwórz okno narzędzi i uruchom narzędzie keytool wraz z parametrami pokazanymi na listingu 10.1 (w rozdziale 2. wyjaśniliśmy, co mamy na myśli, używając terminu „okno narzędzi”).

**Listing 10.1.** Generowanie wpisu magazynu kluczy za pomocą narzędzia keytool

---

```
keytool -genkey -v -keystore "c:\android\release\release.keystore" -alias
→androidbook -storepass paxxword -keypass paxxword -keyalg RSA -validity 14000
```

---

Wszystkie argumenty przekazane do narzędzia keytool zostały opisane w tabeli 10.1.

**Tabela 10.1.** Argumenty przekazane aplikacji keytool

Argument	Opis
genkey	Powoduje, że aplikacja keytool generuje parę kluczy: publiczny i prywatny.
v	Powoduje, że aplikacja keytool wyświetla dane wyjściowe w formie opisowej podczas generowania klucza.
keystore	Ścieżka do bazy danych magazynu kluczy (w naszym przypadku do pliku). W razie potrzeby plik zostanie utworzony.
alias	Niepowtarzalna nazwa wpisu magazynu kluczy. Alias będzie zastosowany jako odniesienie do wpisu.
storepass	Hasło magazynu kluczy.
keypass	Hasło dostępu do klucza prywatnego.
keyalg	Algorytm.
validity	Okres ważności.

Aplikacja keytool wyświetli monit o podanie haseł wymienionych w tabeli 10.1, jeżeli nie zostaną zdefiniowane w wierszu polecen. Jeżeli komputer jest współużytkowany przez wiele osób, bezpieczniej będzie nie określać parametrów –storepass i –keypass w wierszu polecen, lecz zdefiniować je, gdy aplikacja keytool tego zażąda. Polecenie przedstawione na listingu 10.1 wygeneruje bazodany plik magazynu kluczy w utworzonym przez nas folderze. Baza danych będzie plikiem noszącym nazwę *release.keystore*. Okres ważności wpisu (parametr validity) wynosi 14 000 dni (w przybliżeniu 38 lat) — co stanowi długi czas. Ważne jest zrozumienie powodu ustanowienia tak długiego okresu ważności. W dokumentacji Androida znalazło się zalecenie, aby definiować wystarczająco długie okresy ważności, tak by przewyższała całkowity okres istnienia aplikacji, wliczając w to jej wielokrotne aktualizacje. Zalecanym okresem ważności jest 25 lat. Co więcej, jeżeli aplikacja ma zostać umieszczona w serwisie Android Market (<http://www.android.com/market/>), certyfikat musi być ważny przynajmniej do dnia 22 października 2033 roku. Serwis ten sprawdza każdą umieszczoną aplikację pod kątem takiego okresu ważności. Ponieważ podpisy cyfrowe aktualizacji muszą bezwzględnie pasować do podpisu pierwszej wersji aplikacji, plik magazynu kluczy musi być zabezpieczony za wszelką cenę! Jeżeli Czytelnik go utraci bez możliwości odtworzenia, nie będzie mógł aktualizować aplikacji, a jej usprawnianie stanie się nagle olbrzymim problemem.

Wracając do aplikacji keytool, argument alias jest niepowtarzalną nazwą, przydzielaną każdemu wpisowi magazynu kluczy; można jej później używać jako odniesienia do tego wpisu. Po uruchomieniu pokazanego na listingu 10.1 polecenia keytool aplikacja wyświetli kilka pytań (rysunek 10.1), a następnie wygeneruje bazę danych oraz jej wpis.

```
C:\Program Files\Java\jre1.6.0_07\bin>keytool -genkey -v -keystore c:\android\release\keystore -alias androidbook -storepass paxword -keypass paxword -keyalg RSA -validity 14000
What is your first and last name?
[Unknown]: sayed
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]: sayedhashimi
What is the name of your City or Locality?
[Unknown]: Jacksonville
What is the name of your State or Province?
[Unknown]: FL
What is the two-letter country code for this unit?
[Unknown]: US
Is CN=sayed, OU=IT, O=sayedhashimi, L=Jacksonville, ST=FL, C=US correct?
[no]: yes

Generating 1,024 bit RSA key pair and self-signed certificate <SHA1withRSA> with
a validity of 14,000 days
[Unknown]: 04-04-2017, O=sayedhashimi, L=Jacksonville, ST=FL, C=US
[Storing c:\android\release\keystore]

C:\Program Files\Java\jre1.6.0_07\bin>
```

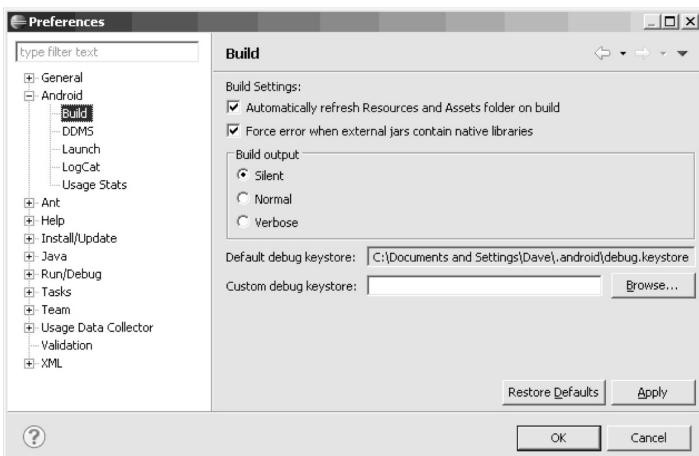
Rysunek 10.1. Dodatkowe pytania wyświetlane przez aplikację keytool

Po utworzeniu pliku magazynu kluczy możemy dodawać do niego kolejne certyfikaty. Wystarczy uruchomić ponownie narzędzie keytool i wprowadzić ścieżkę do istniejącego magazynu kluczy.

## Magazyn kluczy debugowania i certyfikat programistyczny

Wspomnieliśmy, że wtyczka ADT tworzy magazyn certyfikatu programistycznego. Jednak domyślny certyfikat, utworzony w trakcie programowania aplikacji, nie może być wykorzystany w przypadku aplikacji wdrażanej na rynek. Wynika to częściowo z faktu, że certyfikat ten jest ważny jedynie przez 365 dni, co oczywiście nie spełnia wymagania dotyczącego terminu jego ważności trwającego do 22 października 2033 roku. Co się zatem stanie po upłynięciu roku tworzenia aplikacji? Pojawi się błąd komplikacji. Istniejące aplikacje będą działały, jednak w celu utworzenia nowej wersji danego programu musimy wygenerować nowy certyfikat. Najprostszym rozwiązaniem jest usunięcie pliku *debug.keystore*, a w razie potrzeby narzędzie ADT wygeneruje kolejny plik magazynu, a tym samym certyfikat ważny przez następne 365 dni.

Aby znaleźć plik *debug.store*, otwieramy w środowisku Eclipse okno *Preferences* i przechodzimy do menu *Android/Build*. W polu *Default debug keystore* zostanie wyświetlona ścieżka do certyfikatu debugowania, co zostało pokazane na rysunku 10.2 (w rozdziale 2. znajdziemy informacje, jak dotrzeć do okna *Preferences*).



Rysunek 10.2. Położenie certyfikatu debugowania

Naturalnie, skoro posiadamy nowy certyfikat programistyczny, nie możemy za jego pomocą aktualizować bieżących wersji aplikacji obecnych w emulatorach AVD lub fizycznych urządzeniach. Środowisko Eclipse będzie wyświetlało w konsoli monit o odinstalowanie istniejącej aplikacji za pomocą narzędzia adb, co oczywiście powinniśmy uczynić. Jeżeli mamy zainstalowanych wiele aplikacji w urządzeniu AVD, prostszym sposobem może się okazać utworzenie nowego egzemplarza, dzięki czemu będziemy mogli pracować na nim od nowa. Żeby nie mieć problemu z terminem ważności certyfikatu, możemy utworzyć własny plik *debug.keystore* z dowolnie zdefiniowanym okresem ważności certyfikatu. Musi on oczywiście posiadać tę samą nazwę pliku oraz znajdować się w tym samym miejscu co plik, który został utworzony przez narzędzie ADT. Alias certyfikatu posiada wartość *androiddebugkey*, a obydwa hasła *storepass* i *keypass* brzmiają tak samo: *android*. Narzędzie ADT utworzy nazwę certyfikatu *Android Debug*, ustanowi jego jednostkę organizacyjną *Android* oraz dwuliterowy kod kraju US. Wartości parametrów organizacji, miasta oraz stanu mogą pozostać nieznane (*Unknown*).

Jeżeli Czytelnik otrzymał klucz *map-api* od firmy Google, korzystając z przestarzałego certyfikatu, będzie musiał uzyskać nowy klucz powiązany z nowym certyfikatem. Klucze *map-api* zostaną omówione w rozdziale 17.

Skoro wiemy już, jak zdobyć certyfikat cyfrowy pozwalający nam na podpiswanie plików *.apk*, warto się dowiedzieć, w jaki sposób wykorzystać narzędzie *jarsigner* do samego procesu podpisywania. Poniżej omawiamy, w jaki sposób możemy tego dokonać.

## Zastosowanie narzędzia *jarsigner* do podpisania pliku *.apk*

Narzędzie *keytool*, omówione we wcześniejszym podrozdziale, wygenerowało certyfikat cyfrowy, który stanowi jeden z parametrów aplikacji *jarsigner*. Jego drugim parametrem jest rzeczywisty pakiet Androida, który ma zostać podpisany. Żeby utworzyć pakiet Androida, musimy użyć narzędzia *Export Unsigned Application Package* dostępnego we wtyczce ADT środowiska Eclipse. Dostęp do niego uzyskuje się poprzez kliknięcie prawym przyciskiem myszy węzła projektu w aplikacji Eclipse, kliknięcie opcji *Android Tools*, a następnie wybranie *Export Unsigned Application Package*. W ten sposób zostanie wygenerowany plik *.apk* niepodpisany przez certyfikat testowy. Żeby sprawdzić działanie tego mechanizmu, zastosujmy narzędzie *Export Unsigned Application Package* wobec jednego z projektów i zapiszmy gdzieś wygenerowany plik *.apk*. W tym przykładzie użyjemy utworzonego uprzednio folderu magazynu kluczy i wygenerujemy plik *.apk* — *c:\android\release\myappraw.apk*.

Po utworzeniu pliku *.apk* oraz wpisu w magazynie kluczy uruchamiamy aplikację *jarsigner* w celu podpisania tego pliku (listing 10.2). Wpisujemy pełną ścieżkę do pliku magazynu kluczy oraz pliku *.apk*.

**Listing 10.2.** Zastosowanie aplikacji *jarsigner* do podpisania pliku *.apk*

---

```
jarsigner -keystore "ŚCIEŻKA DO PLIKU release.keystore" -storepass
→paxxword -keypass paxxword "ŚCIEŻKA DO NIEPRZETWORZONEGO PLIKU APK" androidbook
```

---

Żeby podpisać plik *.apk*, podajemy lokalizację magazynu kluczy, hasło do tego magazynu, hasło do klucza prywatnego, ścieżkę do pliku *.apk* oraz alias wpisu magazynu kluczy. Aplikacja *jarsigner* następnie podpisze plik *.apk* za pomocą sygnatury pochodzącej z wpisu magazynu kluczy. Aby uruchomić narzędzie *jarsigner*, należy otworzyć okno narzędzi (rozdział 2.) albo wiersz poleceń bądź okno terminala. Następnie trzeba przejść do katalogu *bin* zestawu JDK lub upewnić się, że katalog ten znajduje się w zmiennej systemowej PATH. Ze względów bezpieczeństwa

będzie lepiej, jeśli pominiemy w poleceniu argumenty stanowiące hasła i pozwolimy aplikacji jarsigner na wyświetlenie zapytania o nie. Na rysunku 10.3 widzimy wywołanie narzędzia jarsigner.

```
cmd C:\Windows\system32\cmd.exe
2011-06-22 13:04 1 204 weightgravityraw.apk
    1 plików> 1 204 bajtów
    2 katalogów> 13 129 572 352 bajtów wolnych
c:\android\release>jarsigner -keystore "release.keystore" weightgravityraw.apk a
ndroidbook
Enter Passphrase for keystore:
c:\android\release>
```

Rysunek 10.3. Użycie narzędzia jarsigner

Jak już wcześniej stwierdziliśmy, Android wymaga cyfrowego podpisu aplikacji, żeby uniemożliwić złośliwemu programiście aktualizowanie programu utworzonego przez kogoś innego do własnej wersji. Wynika z tego wniosek, że aktualizacja aplikacji musi być podpisana za pomocą tej samej sygnatury co jej pierwotna wersja. Jeżeli aktualizacja zostanie podpisana za pomocą innej sygnatury, zostanie potraktowana jak odrębna aplikacja. Przypominamy więc ponownie, że należy zachować szczególną ostrożność z plikiem magazynu kluczy, który musi być dostępny podczas aktualizowania aplikacji.

## Optymalizacja aplikacji za pomocą narzędzia zipalign

Jest rzeczą pożądaną, żeby podczas działania na urządzeniu aplikacja jak najwydajniej korzystała z pamięci. Jeżeli w trakcie pracy używa ona nieskompresowanych danych (na przykład niektórych rodzajów obrazów lub plików danych), Android może je odwzorować bezpośrednio w pamięci za pomocą wywołania `mmap()`. Jednak aby to było możliwe, dane muszą zostać przydzielone do czterobajtowego bloku pamięci (obrazowo rzecz ujmując, „wyrównane” do bloku pamięci). Jednostki przetwarzające w urządzeniach obsługujących system Android stanowią procesory 32-bitowe, a 32 bity są równoważne 4 bajtom. Wywołanie `mmap()` sprawia, że dane umieszczone w pliku `.apk` stają się odwzorowaniem pamięci. Jeżeli jednak to czterobajtowe (blokowe) przydzielanie danych nie zostało przeprowadzone, nie można też dokonać procesu odwzorowania i w czasie działania aplikacji będzie wykonywane dodatkowe kopianie danych na poziomie jednostki przetwarzającej (np. procesora lub emulatora). Dostępne w katalogu pakietu Android SDK narzędzie `zipalign` analizuje daną aplikację i w sposób transparentny z punktu widzenia aplikacji przenosi wszelkie nieskompresowane dane do czterobajtowych bloków. W wyniku tego procesu rozmiar aplikacji może się nieznacznie zwiększyć, nie będą to jednak duże zmiany. Żeby przeprowadzić ten proces na naszym pliku `.apk`, należy użyć poniższego polecenia w oknie narzędzi (również rysunek 10.4):

```
zipalign -v 4 infile.apk outfile.apk
```

```
cmd C:\Windows\system32\cmd.exe
C:\android\release>zipalign -v 4 weightgravity.apk weightgravityaligned.apk
Verifying alignment of weightgravityaligned.apk <4>...
 50 META-INF/MANIFEST.MF <OK - compressed>
 400 META-INF/ANDROID.RSA <OK - compressed>
 818 META-INF/ANDROID.BSF <OK - compressed>
1536 res/drawable/icon.png <OK>
499 res/layout/main.xml <OK - compressed>
5399 AndroidManifest.xml <OK - compressed>
6008 resources.arsc <OK>
7093 classes.dex <OK - compressed>
Verification successful
C:\android\release>
```

Rysunek 10.4. Zastosowanie aplikacji zipalign

Zauważmy, że aplikacja zipalign nie modyfikuje pliku wejściowego, dlatego stosujemy człon `raw` (ang. *nieprzetworzony*) w nazwie pliku podczas jego eksportowania ze środowiska Eclipse. W wyniku tego plik wyjściowy posiada nazwę odpowiednią do przeprowadzenia wdrażania. W przypadku potrzeby nadpisania istniejącego pliku wyjściowego `outfile.apk` można użyć opcji `-f`. Zwróćmy ponadto uwagę, że aplikacja zipalign weryfikuje sposób odwzorowania („wyrównania”) danych w pliku po jego przetworzeniu. Żeby zweryfikować poprawność „wyrównania” pliku, można użyć aplikacji zipalign w poniższy sposób:

```
zipalign -c -v 4 filename.apk
```

Istotne jest, żeby omawiany proces został przeprowadzony *po* procesie podpisywania, w przeciwnym razie podpisanie aplikacji może zniwecczyć efekt zoptymalizowania plików. Nie oznacza to wcale, że aplikacja będzie się zawieszać, po prostu może zajmować więcej pamięci, niż potrzeba.

W środowisku Eclipse w zakładce *Android Tools* można natrafić na opcję *Export Signed Application Package*. Powoduje ona uruchomienie tak zwanego kreatora eksportu, za pomocą którego można przeprowadzić wszystkie omówione powyżej etapy. Jednymi danymi, jakie należy wprowadzić, są ścieżka do magazynu kluczy, alias klucza, a także hasła i nazwa pliku `output.apk`. Kreator w razie konieczności wygeneruje nawet nowy magazyn kluczy lub sam klucz. Niektórym wygodniej będzie korzystać z kreatora, innym — przeprowadzać po kolej poszczególne etapy na eksportowanym, niepodpisanym pakiecie aplikacji. Skoro już Czytelnik wie, jak działają obydwa rozwiązania, będzie mógł wybrać wygodniejsze rozwiązanie.

Po podpisaniu i „wyrównaniu” zawartości pliku `.apk` można ręcznie zainstalować aplikację na emulatorze za pomocą narzędzia `adb`. W celach ćwiczeniowych zachęcamy teraz do uruchomienia emulatorka. Jednym ze sposobów uruchomienia emulatorka, o którym jeszcze nie wspomnialiśmy, jest kliknięcie menu *Window* w środowisku Eclipse i wybranie opcji *Android SDK and AVD Manager*. Zobaczmy wyświetcone okno z listą posiadanych urządzeń AVD. Wybierzmy jedno z nich i kliknijmy przycisk *Start....* Emulator uruchomi się bez funkcji kopiowania jakichkolwiek projektów ze środowiska Eclipse. Otwórzmy okno narzędzi iłączmy narzędzie `adb` wraz z polecienniem `install`:

```
adb install "ŚCIEŻKA DO PLIKU APK"
```

Proces może zakończyć się niepowodzeniem z kilku powodów. Najczęściej jednak jego przyczyną jest wcześniejsze zainstalowanie na emulatorze wersji testowej aplikacji, a to powoduje konflikt certyfikatów i wyświetlenie informacji o błędzie. Możliwe też, że wcześniej zainstalowano gotową wersję aplikacji, co spowoduje wyświetlenie komunikatu, że dana aplikacja jest już zainstalowana na urządzeniu. W pierwszym przypadku można odinstalować aplikację testową za pomocą polecenia:

```
adb uninstall packagename
```

Zwróćmy uwagę, że argumentem jest tutaj nazwa pakietu, a nie nazwa pliku `.apk`. Nazwa pakietu jest zdefiniowana w pliku *AndroidManifest.xml* zainstalowanej aplikacji.

W przypadku wystąpienia błędu drugiego rodzaju można wpisać poniższe polecenie, w którym parametr `-r` oznacza ponowne zainstalowanie aplikacji z zachowaniem danych na urządzeniu (w emulatorze):

```
adb install -r "ŚCIEŻKA DO PLIKU APK"
```

Prześledźmy teraz, w jaki sposób podpiswanie aplikacji wpływa na proces jej aktualizowania.

## Instalowanie aktualizacji aplikacji a podpisywanie

Wspomnieliśmy wcześniej, że certyfikat posiada okres ważności oraz że firma Google zaleca ustawienie bardzo długiego terminu wygaśnięcia certyfikatu na wypadek dużej liczby aktualizacji. Co się zatem dzieje z aplikacją po wygaśnięciu certyfikatu? Czy będzie ona nadal działać? Na szczęście tak — Android sprawdza certyfikat jedynie w czasie instalacji programu. Po zainstalowaniu aplikacji będzie ona działała nawet po wygaśnięciu ważności certyfikatu.

A co z aktualizacjami? Niestety, po wygaśnięciu ważności certyfikatu nie będzie możliwości aktualizowania aplikacji. Innymi słowy, zgodnie z radą firmy Google należy ustanowić wystarczająco długi termin ważności certyfikatu dla aplikacji, żeby objął jej cały cykl życia. Jeżeli ważność certyfikatu wygaśnie, Android nie będzie instalował aktualizacji aplikacji. Jedynym wyjątkiem pozostanie wtedy utworzenie kolejnej aplikacji — posiadającej inną nazwę pakietu — i podpisanie jej za pomocą nowego certyfikatu. Jak więc widać, podstawową kwestią jest dobranie terminu ważności certyfikatu w momencie jego utworzenia.

Skoro Czytelnik posiada już wiedzę na temat zabezpieczeń związanych z wdrażaniem oraz instalacją aplikacji, przejdźmy do omówienia zabezpieczeń środowiska wykonawczego w Androidzie.

## Przeprowadzanie testów zabezpieczeń środowiska wykonawczego

Zabezpieczenia środowiska wykonawczego w Androidzie zostały zaimplementowane na poziomie procesu oraz na poziomie operacyjnym. Na poziomie procesu Android zapobiega użytkiwaniu bezpośredniego dostępu do danych jednej aplikacji przez inną aplikację. W Androidzie można to było osiągnąć poprzez uruchomienie każdej aplikacji w odrębnym procesie oraz przez przydzielenie każdej z nich unikatowego i trwałego identyfikatora użytkownika. Na poziomie operacyjnym istnieje zdefiniowana lista chronionych funkcji i zasobów. Żeby aplikacja mogła uzyskać dostęp do tych informacji, należy dodać przynajmniej jedno żądanie uprawnień w pliku *AndroidManifest.xml*. Można także zdefiniować niestandardowe uprawnienie dla aplikacji.

W następnych podrozdziałach zajmiemy się tematyką zabezpieczeń granicy procesu oraz sposobami deklarowania i stosowania predefiniowanych uprawnień. Omówimy także proces tworzenia niestandardowych uprawnień i ustawiania ich dla aplikacji. Zaczniemy od analizy zabezpieczeń na granicach procesu.

## Zabezpieczenia na granicach procesu

W przeciwieństwie do środowisk komputerów biurkowych, w których większość aplikacji korzysta z tego samego identyfikatora użytkownika, niemal każda aplikacja w Androidzie posiada swój własny identyfikator. W ten sposób zostaje utworzona granica izolująca procesy od siebie. Żadna aplikacja nie może uzyskać bezpośredniego dostępu do danych innej aplikacji.

Chociaż każdy proces jest oddzielony od innych, współdzielenie danych pomiędzy aplikacjami jest oczywiście możliwe, musi jednak zostać jawnie zadeklarowane. Innymi słowy, żebytrzymać dane z innej aplikacji, należy podjąć interakcję z jej składnikami. Można na przykład wysłać zapytanie do dostawcy treści innej aplikacji, wywołać aktywność w innej aplikacji lub — jak się przekonamy w rozdziale 11. — nawiązać łączność z usługą innej aplikacji. Dla każdego z wymienionych sposobów określono metody umożliwiające wymianę informacji pomiędzy aplikacjami, dokonuje się tego jednak w jawnym sposób, ponieważ wymiana ta nie polega na bezpośrednim dostępie do właściwej bazy danych, plików i tak dalej.

W Androidzie zabezpieczenia na poziomie granicy procesu są proste i zrozumiałe. Sprawy stają się jednak bardziej interesujące, gdy zwróciśmy uwagę na ochronę zasobów (na przykład danych kontaktowych), funkcji (na przykład aparatu fotograficznego) oraz naszych własnych składników. W celu zapewnienia odpowiedniej ochrony Android definiuje schemat uprawnień. To właśnie nim zajmiemy się teraz.

## Deklarowanie oraz stosowanie uprawnień

Android zapewnia schemat uprawnień służący do ochrony zasobów i funkcji urządzenia. Na przykład domyślnie aplikacje nie mogą uzyskiwać dostępu do listy kontaktów, umożliwiać wykonywania połączeń telefonicznych i tak dalej. Aby chronić użytkownika przed złośliwym oprogramowaniem, Android wymusza na aplikacjach żądanie uprawnień, gdy do ich działania jest niezbędny dostęp do chronionego zasobu lub funkcji. Jak niebawem wyjaśnimy, żądania uprawnień są umieszczane w pliku manifeście. W trakcie instalacji instalator plików APK przydziela lub odrzuca żądane uprawnienia w zależności od sygnatury pliku *.apk* oraz (lub) decyzji użytkownika. Jeżeli uprawnienie nie zostanie przyznane, każda próba wykonania działania lub uzyskania dostępu do danej funkcji zakończy się niepowodzeniem.

W tabeli 10.2 zaprezentowano niektóre powszechnie stosowane funkcje oraz wymagane do nich uprawnienia. Chociaż większość wymienionych funkcji nie została jeszcze omówiona, zajmiemy się nimi w dalszej części książki (w dalszej części tego rozdziału oraz w następnych rozdziałach).

Kompletną listę uprawnień można znaleźć pod adresem:

<http://developer.android.com/reference/android/Manifest.permission.html>

Programiści mogą żądać uprawnień poprzez dodawanie wpisów w pliku *AndroidManifest.xml*. Na przykład na listingu 10.3 zostało utworzone żądanie uzyskania dostępu do aparatu fotograficznego i odczytania listy kontaktów oraz danych kalendarza.

**Listing 10.3.** Uprawnienia w pliku *AndroidManifest.xml*

---

```
<manifest ... >
  <application>
    ...
  </application>
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.READ_CALENDAR" />
</manifest>
```

---

Warto wiedzieć, że żądanie uprawnień można wprowadzać do pliku *AndroidManifest.xml* ręcznie albo za pomocą edytora manifestu. Jest on gotowy do uruchomienia tuż po otwarciu (dwukrotnym kliknięciu) pliku manifestu. W edytorze tym jest dostępna rozwijana lista wszystkich uprawnień, dzięki czemu można uniknąć popełnienia błędu. Jak widać na rysunku 10.5, można uzyskać dostęp do listy uprawnień poprzez wybranie zakładki *Permissions* w edytorze manifestu.

Wiemy już, w jaki sposób jest zdefiniowany w Androidzie zestaw uprawnień służących do ochrony funkcji i zasobów. W podobny sposób możemy definiować i wymuszać niestandardowe uprawnienia dla danej aplikacji. Zobaczmy, jak to działa.

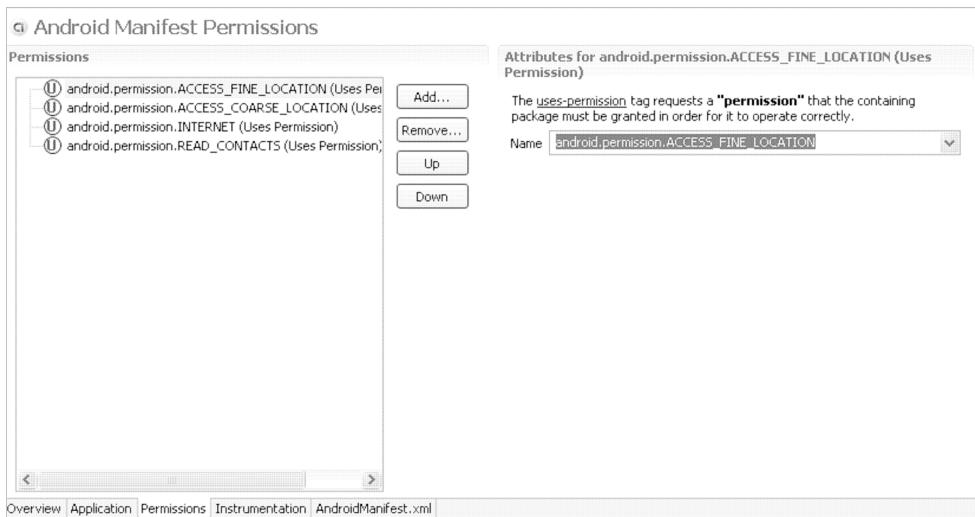
**Tabela 10.2.** Funkcje, zasoby oraz wymagane do nich uprawnienia

Funkcja/zasób	Wymagane uprawnienie	Opis
Aparat fotograficzny	android.permission. CAMERA	Udziela dostępu do aparatu fotograficznego urządzenia.
Internet	android.permission. INTERNET	Umożliwia połączenie sieciowe.
Dane kontaktów użytkownika	android.permission. READ_CONTACTS android.permission. WRITE_CONTACTS	Pozwala na odczytywanie lub zapisywanie danych kontaktów użytkownika.
Dane kalendarza użytkownika	android.permission. READ_CALENDAR android.permission. WRITE_CALENDAR	Pozwala na odczytywanie lub zapisywanie danych kalendarza użytkownika.
Dyktafon	android.permission. RECORD_AUDIO	Umożliwia nagrywanie dźwięku.
Informacje położenia geograficznego GPS	android.permission. ACCESS_FINE_LOCATION	Pozwala na uzyskanie dokładnych danych dotyczących położenia geograficznego. Obejmuje informacje lokalizacyjne GPS. Jest ono również wystarczające dla sieci Wi-Fi oraz wież komórkowych.
Informacje położenia geograficznego o sieci Wi-Fi	android.permission. ACCESS_COARSE_LOCATION	Pozwala na uzyskanie zgrubnych danych dotyczących położenia geograficznego. Obejmuje informacje lokalizacyjne sieci Wi-Fi oraz uzyskiwane z wież komórkowych.
Informacje o stanie baterii	android.permission. BATTERY_STATS	Umożliwia uzyskanie informacje o stanie baterii.
Bluetooth	android.permission. BLUETOOTH	Pozwala na połączenie ze sparowanym urządzeniem Bluetooth.

## Stosowanie niestandardowych uprawnień

Android pozwala na zdefiniowanie własnych uprawnień wobec danej aplikacji. Jeżeli na przykład chcemy uniemożliwić określonym użytkownikom uruchamianie jednej z aktywności w aplikacji, możemy tego dokonać za pomocą uprawnienia niestandardowego. Aby korzystać z własnych uprawnień, należy je najpierw zadeklarować w pliku *AndroidManifest.xml*. Po zdefiniowaniu uprawnienia można się do niego odnosić jak do pozostałych składowych definicji. Zademonstrujemy, jak to działa.

Zbudujmy zatem aplikację, której aktywność nie będzie dostępna dla każdego użytkownika. Żeby tego dokonać, będzie potrzebował specjalnego uprawnienia. Po utworzeniu aplikacji zawierającej taką uprzywilejowaną aktywność będziemy mogli napisać klienta zdolnego do wywołania tej aktywności.



**Rysunek 10.5.** Narzędzie edytora manifestu w środowisku Eclipse

#### Uwaga!

Na końcu tego rozdziału podaliśmy adres URL, pod którym można znaleźć projekty utworzone specjalnie na potrzeby tego rozdziału. Projekty te można zaimportować bezpośrednio do środowiska Eclipse.

Najpierw należy utworzyć projekt zawierający niestandardowe uprawnienie i aktywność. Dokonujemy tego poprzez uruchomienie środowiska Eclipse i kliknięcie opcji *New/New Project/Android Project*. Otworzy się okno dialogowe *New Android Project*. Jako nazwę projektu wpisujemy *CustomPermission*, wybieramy opcję *Create new project in workspace* i zaznaczamy *Use default location*. Nazwa aplikacji może brzmieć *Niestandardowe uprawnienie*, nazwa pakietu *com.cust.perm*, a nazwa aktywności *CustPermMainActivity*. Powinniśmy również wpisać dowolną wersję docelowej platformy w polu *Build Target*. Żeby utworzyć projekt, należy kliknąć przycisk *Finish*. Wygenerowany projekt będzie zawierał dopiero co utworzoną aktywność, pełniącą rolę domyślnej (głównej) aktywności. Stwórzmy także **aktywność uprzywilejowaną** — aktywność, dla której wymagane jest specjalne uprawnienie. W aplikacji Eclipse należy przejść do pakietu *com.cust.perm*, utworzyć klasę *PrivActivity*, dla której superklasą jest *android.app.Activity*, a następnie przepisać kod umieszczony na listingu 10.4.

#### Listing 10.4. Klasa PrivActivity

```
package com.cust.perm;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup.LayoutParams;
import android.widget.LinearLayout;
import android.widget.TextView;

public class PrivActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```

super.onCreate(savedInstanceState);
LinearLayout view = new LinearLayout(this);

view.setLayoutParams(new LayoutParams(
    LayoutParams.FILL_PARENT, LayoutParams.WRAP_CONTENT));
view.setOrientation(LinearLayout.HORIZONTAL);

TextView nameLbl = new TextView(this);

nameLbl.setText("Pozdrowienia z aktywności PrivActivity");
view.addView(nameLbl);

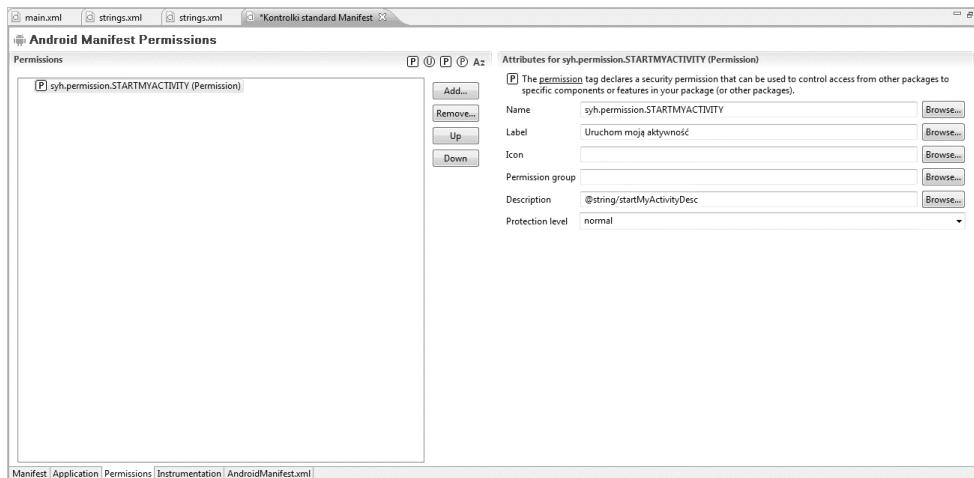
setContentView(view);

}
}

```

Jak widać, aktywność `PrivActivity` nie ma żadnych nadzwyczajnych funkcji. Chcemy jedynie pokazać, w jaki sposób można chronić ją za pomocą uprawnienia i wywołać ją za pomocą klienta. Jeżeli klient zostanie właściwie zaimplementowany, na ekranie pojawi się wiadomość `Pozdrowienia z aktywności PrivActivity`. Po wygenerowaniu tej chronionej aktywności można utworzyć uprawnienie do niej.

Żeby stworzyć niestandardowe uprawnienie, należy je zdefiniować w pliku `AndroidManifest.xml`. Najłatwiej to zrobić za pomocą edytora manifestu. Wystarczy dwukrotnie kliknąć plik `AndroidManifest.xml` i wybrać zakładkę `Permissions`. W oknie `Permissions` klikamy przycisk `Add`, wybieramy opcję `Permission` i wciskamy przycisk `OK`. Zostanie wygenerowane puste uprawnienie. Należy wypełnić je atrybutami, tak jak zilustrowano na rysunku 10.6. Wypełniamy pola z prawej strony, a jeżeli etykietka po prawej stronie wciąż będzie nosiła nazwę `Permission`, klikamy ją, dzięki czemu nazwa uprawnienia powinna zostać zaktualizowana.



**Rysunek 10.6.** Deklarowanie niestandardowego uprawnienia za pomocą edytora manifestu

Jak widać na rysunku 10.6, uprawnienie posiada nazwę, etykietę, ikonę, grupę uprawnienia, opis i poziom ochrony. W tabeli 10.3 opisaliśmy wymienione parametry.

**Tabela 10.3.** Atrybuty uprawnienia

Atrybut	Wymagany?	Opis
android:name	Tak	Nazwa uprawnienia. Zazwyczaj należy przestrzegać konwencji nazewnictwa Androida (*.permission.*).
android:protectionLevel	Tak	Definiuje „potencjał zagrożenia” związany z uprawnieniem. Należy wybrać jedną spośród następujących wartości:  normal dangerous signature signatureOrSystem
		W zależności od poziomu ochrony system podejmuje inne działanie podczas określania, czy należy przydzielić dostęp użytkownikowi, czy nie. Wartość normal oznacza, że uprawnienie stanowi niewielkie zagrożenie i nie stanowi niebezpieczeństwa dla systemu, użytkownika lub innych aplikacji. Poziom dangerous oznacza, że istnieje duże ryzyko oraz że system najprawdopodobniej będzie potrzebował zgody użytkownika przed akceptacją uprawnienia. Wartość signature przydziela uprawnienie jedynie aplikacjom posiadającym taką samą sygnaturę co aplikacja deklarująca to uprawnienie. Wartość signatureOrSystem pozwala przydziełać uprawnienie aplikacjom posiadającym taką samą sygnaturę jak aplikacja deklarująca to uprawnienie lub klasom pakietu Android. Ten poziom ochrony jest stosowany w szczególnych przypadkach, w których wielu producentów musi współdzielić funkcje poprzez obraz systemu.
android:permissionGroup	Nie	Uprawnienia można grupować, jednak w przypadku uprawnień niestandardowych powinno się pomijać tę właściwość. Jeżeli rzeczywiście trzeba utworzyć grupę uprawnień, lepiej skorzystać z tego atrybutu:  android.permission-group.SYSTEM_TOOLS
android:label	Nie	Ta właściwość również nie jest wymagana. Służy do stworzenia krótkiego opisu uprawnienia.
android:description	Nie	Ta właściwość także nie jest wymagana. Można tu umieścić dokładniejsze informacje na temat przeznaczenia uprawnienia.
android:icon	Nie	Uprawnienia można powiązać z ikonami umieszczonymi w zasobach (na przykład @drawable/mojaikona).

Po utworzeniu niestandardowego uprawnienia trzeba sprawić, by aktywność `PrivActivity` mogła być uruchamiana jedynie przez aplikacje posiadające uprawnienie `syh.permission.STARTMYACTIVITY`. Można dołączyć wymagane uprawnienie do aktywności poprzez dodanie atrybutu `android:permission` do definicji aktywności w pliku `AndroidManifest.xml`. Żeby uruchomić aktywność, będzie konieczne również dodanie do niej filtru intencji. Zaktualizujmy plik `AndroidManifest.xml` kodem zawartym na listingu 10.5.

**Listing 10.5.** Plik `AndroidManifest.xml` projektu zawierającego niestandardowe uprawnienie

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.cust.perm"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".CustPermMainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="PrivActivity"
            android:permission="syh.permission.STARTMYACTIVITY">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
    </application>

    <permission
        android:protectionLevel="normal"
        android:label="Uruchom moją aktywność"
        android:description="@string/startMyActivityDesc"
        android:name="syh.permission.STARTMYACTIVITY"></permission>

    <uses-sdk android:minSdkVersion="4" />
</manifest>
```

---

Jak widać na listingu 10.5, musimy dodać stałą w postaci ciągu znaków `startMyActivityDesc` do zasobów typu `string`. Żeby komplikacja kodu przebiegła bezbłędnie, dodajmy następujący ciąg znaków do pliku `res/values/strings.xml`:

```
<string name="startMyActivityDesc">Umożliwia uruchomienie mojej aktywności</string>
```

Włączmy teraz projekt na emulatorze. Chociaż główna aktywność nie wykonuje żadnej czynności, musimy zainstalować aplikację na emulatorze, zanim będzie można napisać klienta dla uprzywilejowanej aktywności.

Napiszmy więc takiego klienta. W środowisku Eclipse należy kliknąć opcję `New/Project/Android Project`. Jako nazwę projektu wpiszmy `ClientOfCustomPermission`, wybierzmy opcję `Create new project in workspace` i zaznaczmy opcję `Use default location`. Nazwa aplikacji może brzmieć `Klient niestandardowego uprawnienia`, nazwa pakietu `com.client.cust.perm`, a nazwa aktywności `ClientCustPermMainActivity`. W polu `Build Target` wpisujemy nazwę dowolnej wersji platformy. Aby utworzyć projekt, należy kliknąć przycisk `OK`.

Teraz napiszemy aktywność zawierającą przycisk, którego kliknięcie spowoduje wywołanie uprzywilejowanej aktywności. Przepiszmy kod układu graficznego z listingu 10.6 do pliku *main.xml*, znajdującego się w naszym nowym projekcie.

#### **Listing 10.6.** Plik main.xml w projekcie klienta

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:id="@+id	btn"
        android:text="Uruchom PrivActivity"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="doClick" />
</LinearLayout>
```

---

Jak widać, plik XML układu graficznego definiuje pojedynczy przycisk nazwany *Uruchom PrivActivity*. Napiszmy teraz aktywność, która będzie generowała zdarzenie wywołane kliknięciem i uruchamiała uprzywilejowaną aktywność. Kod z listingu 10.7 należy umieścić w klasie *ClientCustPermMainActivity*.

#### **Listing 10.7.** Zmodyfikowana klasa ClientCustPermMainActivity

---

```
package com.client.cust.perm;
// To jest plik ClientCustPermMainActivity.java

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ClientCustPermMainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void doClick(View view) {
        Intent intent = new Intent();
        intent.setClassName("com.cust.perm", "com.cust.perm.PrivActivity");
        startActivity(intent);
    }
}
```

---

Na listingu 10.7 widać, że po wywołaniu przycisku zostaje utworzona nowa intencja, a następnie ustanowiona nazwa klasy uruchamianej aktywności. W naszym przypadku chcemy uruchomić aktywność *com.cust.perm.PrivActivity* z pakietu *com.cust.perm*.

W tym momencie jedynym brakującym elementem jest wpis uses-permission, który jest dodawany do pliku manifestu po to, aby poinformować Androida, że należy skorzystać z uprawnienia syh.permission.STARTMYACTIVITY. Zamieńmy kod manifestu z projektu klienta na zawarty na listingu 10.8.

#### **Listing 10.8.** Plik manifest klienta

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.client.cust.perm"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".ClientCustPermMainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

    <uses-permission android:name="syh.permission.STARTMYACTIVITY">
        <uses-sdk android:minSdkVersion="4" />
    </manifest>
```

---

Na listingu 10.8 dodaliśmy wpis uses-permission, żądający niestandardowego uprawnienia, które jest wymagane do uruchomienia aktywności *PrivActivity* zaimplementowanej w naszym projekcie.

Teraz można zainstalować projekt klienta na emulowanym urządzeniu i kliknąć przycisk *Uruchom PrivActivity*. Po przetworzeniu zdarzenia kliknięcia zostanie wyświetlony napis *Pozdrowienia z aktywności PrivActivity*.

Po udanym wywołaniu uprzywilejowanej aktywności można usunąć wpis uses-permission z pliku manifestu klienta i ponownie wdrożyć projekt. Po kliknięciu przycisku wywołania aktywności pojawi się komunikat o błędzie. Zauważmy, że aplikacja *LogCat* wyświetli informację o wyjątku odmowy dostępu.

Wiemy już, w jaki sposób działają niestandardowe uprawnienia w Androidzie. Oczywiście nie są one ograniczone wyłącznie do aktywności. Można stosować uprawnienia zarówno predefiniowane, jak i niestandardowe również wobec innych rodzajów składników Androida. Zajmiemy się teraz kolejnym rodzajem uprawnień — uprawnieniami identyfikatorów URI.

## **Stosowanie uprawnień identyfikatorów URI**

W przypadku dostawców treści (omówionych w rozdziale 3.) całkowite przydzielanie dostępu lub jego całkowite blokowanie często nie wchodzi w grę. Na szczęście w Androidzie została zaimplementowana odpowiednia technologia. Dobrym przykładem są załączniki do wiadomości e-mail. Żeby załącznik został wyświetlony, musi zostać odczytany przez inną aktywność. Ale ta aktywność nie powinna mieć dostępu do wszystkich danych poczty e-mail, a nawet do pozostałych załączników. W tym momencie można wykorzystać uprawnienia do identyfikatorów URI.

## Przekazywanie uprawnień do identyfikatorów URI w intencjach

Po wywołaniu innej aktywności oraz przekazaniu jej identyfikatora URI aplikacja może określić, że przydziela uprawnienia jedynie do przekazywanych identyfikatorów URI. Najpierw jednak dany program sam musi otrzymać uprawnienia do identyfikatora URI. Z kolei dostawca tego identyfikatora musi współpracować z tym programem oraz umożliwiać przydział uprawnień do innej aktywności. Na listingu 10.9 zaprezentowano kod umożliwiający wywołanie aktywności wraz z przydzielaniem uprawnień. Kod ten w rzeczywistości stanowi wycinek programu Android Email, w którym służy do uruchamiania aplikacji pozwalającej na przeglądanie załączników.

**Listing 10.9.** Kod uruchamiający aktywność wraz z przydzielaniem uprawnień

---

```
try {
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(contentUri);
    intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    startActivity(intent);
} catch (ActivityNotFoundException e) {
    mHandler.attachmentViewError();
    // Do zrobienia: dodanie w następnej wersji odpowiedniego ostrzeżenia (oraz mnóstwa
    // mechanizmów czyszczących, zapobiegających jego pojawieniu się).
}
```

---

Rodzaj załącznika określa obiekt ContentUri. Zwróćmy uwagę, że utworzenie intencji jest efektem działania Intent.ACTION\_VIEW, a metoda setData() powoduje ustawienie danych. Widać też flagę przydzielającą uprawnienia do odczytu załącznika dowolnej aktywności dopasowanej do intencji. W tym miejscu do gry wkracza dostawca treści. To, że aktywność odczytała uprawnienia do treści, wcale nie oznacza, że będzie przekazywać te uprawnienia innej aktywności, która jeszcze nie posiada uprawnień. Musi na to pozwolić również dostawca treści. Po znalezieniu w aktywności pasującego filtra intencji Android sprawdza dostawcę treści w celu upewnienia się, czy można przydzielić dane uprawnienia. W istocie dostawca treści otrzymuje żądanie umożliwienia dostępu do tej nowej aktywności, dokładniej do treści określonej w identyfikatorze URI. Jeżeli dostawca odmówi, zostanie wyświetlony komunikat o wyjątku SecurityException, a cała operacja zakończy się niepowodzeniem. Na listingu 10.9 ta konkretna aplikacja nie sprawdza wyjątku SecurityException, ponieważ programista nie spodziewa się żadnej odmowy udzielenia uprawnień. Wynika to z faktu, że dostawca załącznika jest częścią aplikacji Email! Istnieje jednak możliwość, że nie będzie żadnej aktywności obsługującej załącznik, dlatego więc mamy do czynienia tylko z tym wyjątkiem. W przypadku gdy aktywność przetwarzająca identyfikator URI uzyska do niego uprawnienie, dostawca treści nie może go odmówić. To znaczy, że wywołującą aktywność może uzyskać uprawnienie, a jeżeli aktywność po stronie odbiorczej intencji posiada już wymagane uprawnienia do obiektu contentURI, wywołana aktywność będzie mogła kontynuować działanie bez najmniejszego problemu.

Oprócz flagi Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION istnieje jeszcze flaga dająca uprawnienia do zapisu — Intent.FLAG\_GRANT\_WRITE\_URI\_PERMISSION. Istnieje możliwość ustalenia każdej z tych flag w obiekcie klasy Intent. Poza tym znajdują one zastosowanie również w klasach typu Service, Broadcast Receiver, a także Activity, ponieważ aktywności także mogą odbierać intencje.

## Definiowanie uprawnień identyfikatorów URI w dostawcach treści

W jaki zatem sposób dostawca treści definiuje uprawnienia identyfikatorów URI? Dokonuje tego w pliku *AndroidManifest.xml* na jeden z dwóch sposobów:

- Pierwszy sposób polega na przypisaniu wartości atrybutowi `android:grantUriPermissions` wewnątrz znacznika `<provider>`. Jeżeli wstawimy wartość `true`, dostęp do treści tego dostawcy będzie nieograniczony. W przypadku wprowadzenia wartości `false` może się pojawić drugi sposób określania uprawnień identyfikatorów URI, ewentualnie dostawca treści może nie udzielić żadnych uprawnień.
- Drugie rozwiązanie polega na definiowaniu tych uprawnień w węzłach potomnych znacznika `<provider>`. Taki potomny znacznik nosi nazwę `<grant-uri-permission>` i możemy wstawić ich kilka w jednym węźle nadziedzonym. Znacznik `<grant-uri-permission>` posiada trzy atrybuty:
  - Za pomocą atrybutu `android:path` możemy określić pełną ścieżkę, która będzie mogła uzyskiwać wszelkie uprawnienia.
  - W analogiczny sposób atrybut `android:pathPrefix` definiuje początek ścieżki identyfikatora URI.
  - Atrybut `android:pathPattern` dopuszcza wyrażenia wieloznaczne (np. `*`) w definiowaniu ścieżki.

Jak już wcześniej stwierdziliśmy, jednostka przyznająca musi sama posiadać odpowiednie uprawnienia, zanim będzie mogła je przekazać innej jednostce. Dostawcy treści posiadają dodatkowe sposoby kontrolowania dostępu do swoich zasobów poprzez atrybut `android:readPermission`, a także atrybuty `android:writePermission` oraz `android:permission` (wygodny sposób określania uprawnień zapisu oraz odczytu za pomocą jednej wartości) w znaczniku `<provider>`. Wartość każdego z tych trzech atrybutów przybiera postać ciągu znaków, reprezentującego uprawnienie, które musi posiadać jednostka wywołująca w celu przeprowadzania operacji zapisu/odczytu na tym dostawcy treści. Zanim aktywność będzie mogła przydzielić uprawnienie odczytu identyfikatora URI innej jednostce, sama musi posiadać uprawnienie odczytu, co zostaje zapewnione dzięki obecności atrybutów `android:readPermission` lub `android:Permission`. Jednostka żądająca uprawnień może zadeklarować je w pliku manifeście, dokładniej w znaczniku `<uses-permissions>`.

## Odbośniki

Poniżej zamieszczamy przydatne łącza do materiałów, z którymi warto się dokładniej zapoznać:

- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — znajdziemy tu pełen zestaw projektów bezpośrednio związanych z tą książką. Projekt ukazujący aspekty omówione w tym rozdziale został umieszczony w katalogu *ProAndroid3\_R10\_Zabezpieczenia*. Umieściliśmy w nim również plik *Czytaj.TXT* wyjaśniający szczegółowo proces importowania projektów do środowiska Eclipse.
- <http://developer.android.com/guide/topics/security/security.html> — pod tym adresem znajdziemy podrozdział *Security and Permissions*, będący częścią dokumentacji *Android Developer's Guide*. Stanowi on ogólne omówienie zagadnienia wraz z odnośnikami do wielu dalszych materiałów.

- <http://developer.android.com/guide/publishing/app-signing.html> — adres kierujący nas do podrozdziału *Sigining Your Application* z wyżej wspomnianego dokumentu *Android Developer's Guide*.
- [http://android.git.kernel.org/?p=platform/packages/apps>Email.git;a=blob\\_plain;f=src/com/android/email/activity/MessageView.java](http://android.git.kernel.org/?p=platform/packages/apps>Email.git;a=blob_plain;f=src/com/android/email/activity/MessageView.java) — pod tym adresem<sup>1</sup> znajdziemy kod źródłowy podstawowej aplikacji Email, w którym została użyta flaga *FLAG\_GRANT\_READ\_URI\_PERMISSION*. Przeglądając ten kod źródłowy, możemy się przekonać, w jaki sposób zespół twórców Androida implementuje uprawnienia identyfikatorów URI.

## Podsumowanie

W niniejszym rozdziale wyjaśniliśmy, że Android wymaga cyfrowych certyfikatów od wszystkich aplikacji. Opisaliśmy sposoby zapewnienia bezpieczeństwa podczas projektowania aplikacji na emulatorze oraz w środowisku Eclipse, a także metody podpisywania końcowych wersji pakietu Android. Przedstawiliśmy także kwestię zabezpieczeń środowiska wykonawczego — można się było dowiedzieć, że instalator w systemie Android wymaga od aplikacji uprawnień podczas jej instalowania. Pokazaliśmy również, w jaki sposób definiować uprawnienia wymagane przez aplikację, a także jak utworzyć własne, niestandardowe uprawnienia. Na koniec wyjaśniliśmy, jak dostawcy treści kontrolują dostęp do swoich zasobów, a także jak pozwalają jednym jednostkom przekazywać uprawnienia dla innych jednostek, które mogą zostać wywołane w celu prowadzenia określonych operacji na danych z dostawcy treści, bez konieczności nadawania takiej pomocniczej jednostce uprawnień do *wszystkich* informacji zawartych w dostawcy.

W następnym rozdziale zajmiemy się tworzeniem oraz użytkowaniem usług w Androidzie.

---

<sup>1</sup> W trakcie tłumaczenia książki strona <http://android.git.kernel.org> została zaatakowana przez nieznanych sprawców i od tego czasu wszelkie dostępne na niej zasoby są niedostępne dla użytkowników. Prawdopodobnie strona ta zostanie ponownie oddana do użytku w niedalekiej przyszłości, do tego czasu osoby pragnące przejrzeć kod źródłowy Androida muszą skorzystać z alternatywnego źródła. Na stronie <http://source.android.com/source/downloading.html> znajdziemy instrukcję korzystania z narzędzia *Repo*, pozwalającego na pobieranie i przeglądanie wspomnianego kodu źródłowego — przyp. tłum.



# Tworzenie i użytkowanie usług

System Android zawiera kompletny stos programowy. Oznacza to, że otrzymujemy system operacyjny oraz oprogramowanie integracyjne (ang. *middleware*), a także działające aplikacje (na przykład program do obsługi telefonu). Poza tymi składnikami otrzymujemy dostęp do środowiska SDK, dzięki któremu możemy pisać aplikacje dla tego systemu. Dotychczas zajmowaliśmy się aplikacjami, które w bezpośredni sposób współpracują z użytkownikiem poprzez interfejs UI. Nie omawialiśmy jednak usług działających w tle oraz możliwości tworzenia składników przetwarzanych w tle.

W niniejszym rozdziale skupimy się na tworzeniu i użytkowaniu usług w Androidzie. Na początku omówimy użytkowanie usług HTTP, następnie pokażemy przyjemny sposób przeprowadzania prostych zadań w tle, a w dalszej kolejności zajmiemy się komunikacją międzyprocesową — czyli komunikacją pomiędzy aplikacjami znajdującymi się w obrębie jednego urządzenia. Na końcu pojedziemy jeszcze jeden krok naprzód i utworzymy działającą przykładową aplikację, integrującą się z interfejsem Tłumacza Google.

## Użycanie usług HTTP

Aplikacje w Androidzie, a ogólnie aplikacje dla urządzeń mobilnych, są małymi programami oferującymi dużą liczbę funkcji. Jednym ze sposobów zapewnienia tak dużej funkcjonalności aplikacji w tak małym urządzeniu jest uzyskiwanie przez nie informacji z różnych źródeł. Na przykład większość smartfonów zawiera aplikację Mapy, która jest wyposażona w rozbudowane funkcje przetwarzania map. Wiemy już jednak, że program ten jest zintegrowany z serwerem Mapy Google oraz innymi usługami, dzięki którym uzyskuje on wspomnianą złożoność.

Skoro o tym mowa, jest całkiem prawdopodobne, że tworzone przez nas aplikacje będą również wykorzystywać informacje z innych aplikacji i interfejsów API. Standardową strategią integracji jest stosowanie protokołu HTTP. Na przykład możemy przewidzieć udostępnianie w internecie serwetu Java, który będzie dostarczał usługi uzyskiwane poprzez jedną z aplikacji Androida. Jak tego dokonać

w Androidzie? Interesujący jest fakt, że zestaw Android SDK jest zaopatrzony w powszechnie stosowany w środowisku J2EE moduł `HttpClient`, w wersji zaprojektowanej przez organizację Apache (<http://hc.apache.org/httpclient-3.x/>). W środowisku Android SDK wersja tego modułu została dopasowana do Androida, ale interfejsy API zostały niemal niezmienione w stosunku do wersji dla środowiska J2EE.

Moduł `HttpClient` jest rozbudowanym klientem HTTP. Chociaż posiada pełną obsługę protokołu HTTP, najczęściej będziemy wykorzystywać jedynie wywołania metod GET i POST. W niniejszym podrozdziale przedstawimy zatem wspomniane wywołania GET i POST modułu `HttpClient`.

## Wykorzystanie modułu HttpClient do żądań wywołania GET

Poniżej przedstawiamy jeden z ogólnych algorytmów stosowania modułu `HttpClient`:

1. Utwórz moduł `HttpClient` (lub skorzystaj z istniejącego odniesienia).
2. Utwórz nową metodę HTTP, na przykład `PostMethod` lub `GetMethod`.
3. Skonfiguruj pary nazwa – wartość dla parametrów protokołu HTTP.
4. Wykonaj wywołanie HTTP za pomocą modułu `HttpClient`.
5. Przetwórz odpowiedź protokołu HTTP.

Na listingu 11.1 został pokazany sposób przeprowadzenia wywołania GET za pomocą modułu `HttpClient`.

**Uwaga!**

Na końcu tego rozdziału zamieszczamy adres, pod którym Czytelnik znajdzie projekty pozwalające zrozumieć koncepcje omówione w rozdziale. Istnieje możliwość ich bezpośredniego zimportowania do środowiska Eclipse. Ponadto, ponieważ kod będzie próbował uzyskać dostęp do internetu, podczas wywoływania protokołu HTTP za pomocą modułu `HttpClient` musimy dodać w pliku manifeście upoważnienie `android.permission.INTERNET`.

---

**Listing 11.1.** Stosowanie klas `HttpClient` i `HttpGet` — plik `HttpGetDemo.java`

---

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;
import android.app.Activity;
import android.os.Bundle;

public class HttpGetDemo extends Activity {
    /** Wywoływanie podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        BufferedReader in = null;
        try {
```

```
HttpClient client = new DefaultHttpClient();
HttpGet request = new HttpGet("http://code.google.com/android/");
HttpResponse response = client.execute(request);

in = new BufferedReader(
    new InputStreamReader(
        response.getEntity().getContent()));

StringBuffer sb = new StringBuffer("");
String line = "";
String NL = System.getProperty("line.separator");
while ((line = in.readLine()) != null) {
    sb.append(line + NL);
}
in.close();

String page = sb.toString();
System.out.println(page);
} catch (Exception e) {
    e.printStackTrace();
} finally {
    if (in != null) {
        try {
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}
```

Moduł `HttpClient` zawiera abstrakcje różnych typów żądania protokołu HTTP, na przykład `HttpGet`, `HttpPost` i tak dalej. Na listingu 11.1 moduł `HttpClient` służy do pobrania zawartości z adresu `http://code.google.com/android/`. Właściwe żądanie protokołu HTTP jest przeprowadzane poprzez wywołanie metody `client.execute()`. Po wykonaniu żądania kod przekształca całą odpowiedź w pojedynczy obiekt typu `string`. Zauważmy, że obiekt `BufferedReader` jest zamknięty w bloku `finally`, który jednocześnie zamyka podstawowe połączenie HTTP.

W powyższym przykładzie umieściliśmy logikę protokołu HTTP wewnętrz aktywności, ale nie musimy stosować modułu `HttpClient` z poziomu kontekstu aktywności. Możemy go wykorzystywać wewnątrz kontekstu składnika Androida lub stosować go jako część samodzielnej klasy. W rzeczywistości nie powinniśmy wykorzystywać klasy `HttpClient` bezpośrednio wewnątrz aktywności, ponieważ wywołanie sieci może trwać pewien czas i spowodować wymuszone zamknięcie aktywności. Zajmiemy się tym zagadnieniem w dalszej części rozdziału. Na razie posłużymy się niewielkim uproszczeniem, gdyż chcemy skupić się na mechanizmie wywoływanego klasy `HttpClient`.

Kod widoczny na listingu 11.1 wykonuje żądanie protokołu HTTP bez przekazywania jakichkolwiek parametrów serwerowi. Możemy przekazywać parametry (parę nazwa – wartość) jako część żądania poprzez dodanie tych par do adresu URL, podobnie jak ma to miejsce na listingu 11.2.

**Listing 11.2.** Dodawanie parametrów do żądania metody GET protokołu HTTP

```
HttpGet request = new HttpGet("http://somehost/WS2/Upload.aspx?one=valueGoesHere");
client.execute(request);
```

Podczas wykonywania żądania GET parametry (nazwy i wartości) tego żądania są przekazywane w postaci części adresu URL. Przekazywanie parametrów w ten sposób ma pewne ograniczenia. Gwoli ścisłości, długość adresu URL nie powinna przekraczać 2048 znaków. Jeżeli chcemy wysłać większą ilość danych, powinniśmy skorzystać z żądania metody HTTP POST. Metoda POST jest elastyczniejsza i przekazuje parametry w formie części treści żądania.

## **Wykorzystanie modułu HttpClient do żądań wywołania POST (przykład wieloczęściowy)**

Wykonanie wywołania POST jest bardzo podobne do wywołania GET (listing 11.3).

**Listing 11.3.** Żądanie metody HTTP POST za pomocą modułu HttpClient

```
HttpClient client = new DefaultHttpClient();
HttpPost request = new HttpPost(
    "http://192.165.13.37/uslugi/robCos.do");
List<NameValuePair> postParameters = new ArrayList<NameValuePair>();
postParameters.add(new BasicNameValuePair("first", "param value one"));
postParameters.add(new BasicNameValuePair("issuenum", "10317"));
postParameters.add(new BasicNameValuePair("username", "dave"));
UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(
    postParameters);
request.setEntity(formEntity);
HttpResponse response = client.execute(request);
```

Kod widoczny na listingu 11.3 zastępuje trzy wiersze z listingu 11.1, w miejscu, gdzie jest wykorzystywana metoda `HttpGet`. Cała reszta pozostaje bez zmian. Aby żądać wywołania metody POST za pomocą modułu `HttpClient`, musimy wywołać jego metodę `execute()` wobec instancji `HttpPost`. Podczas przeprowadzania wywołań metody POST zazwyczaj przekazujemy parametry nazwa – wartość zakodowane w postaci adresu URL jako część żądania protokołu HTTP. W celu wykonania tego za pomocą modułu `HttpClient` musimy utworzyć listę zawierającą wystąpienia obiektów `NameValuePair` i umieścić ją w klasie `UrlEncodedFormEntity`. Obiekt `NameValuePair` zawiera kombinację parametrów nazwa – wartość, natomiast klasa `UrlEncodedFormEntity` potrafi przetłumaczyć listę tych obiektów na język zrozumiały dla wywołań protokołu HTTP (ogółem wywołać metody POST). Po utworzeniu klasy `UrlEncodedFormEntity` można ustanowić typ obiektu funkcji `HttpPost`, a następnie odpowiedzieć na żądanie.

W kodzie pokazanym na listingu 11.3 stworzyliśmy moduł `HttpClient`, a następnie wywołaliśmy funkcję `HttpPost` posiadającą adres URL punktu końcowego protokołu HTTP. Wygenerowaliśmy listę obiektów `NameValuePair` i umieściliśmy w niej pojedynczy parametr nazwa – wartość. Następnie utworzyliśmy wystąpienie obiektu `UrlEncodedFormEntity` i przekazaliśmy jego konstruktorowi listę obiektów `NameValuePair`. Na koniec wywołaliśmy metodę `setEntity()` żądania metody POST i spełniliśmy to żądanie za pomocą instancji `HttpClient`.

W rzeczywistości metoda HTTP POST jest o wiele potężniejszym narzędziem. Dzięki niej możemy równie łatwo przekazywać pojedyncze parametry nazwa – wartość, co zostało przedstawione na listingu 11.3, jak również złożone obiekty, takie jak pliki. Metoda HTTP POST obsługuje inny format treści żądania, znany jako **wieloczęściowa metoda POST** (ang. *multipart POST*). Dzięki temu typowi metody POST możemy wraz z parametrami nazwa – wartość wysyłać własne pliki. Niestety, wersja modułu HttpClient dostępna w Androidzie nie obsługuje wieloczęściowych metod POST w sposób bezpośredni. Aby można było przeprowadzać wieloczęściowe wywołania metody POST, potrzebne jest dodatkowe oprogramowanie. Chodzi o trzy posiadające jawną kod źródłowy projekty firmy Apache: Apache Commons IO, Mime4J i HttpMime, które można pobrać z następujących witryn:

- **Commons IO** — <http://commons.apache.org/io/>,
- **Mime4J** — <http://james.apache.org/mime4j/>,
- **HttpMime** — <http://hc.apache.org/downloads.cgi> (wewnątrz wersji HttpClient).

Na listingu 11.4 zostało przedstawione zastosowanie wieloczęściowej metody POST w Androidzie.

**Listing 11.4.** Tworzenie wywołania wieloczęściowej metody POST

```
import java.io.ByteArrayInputStream;
import java.io.InputStream;
import org.apache.commons.io.IOUtils;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.mime.MultipartEntity;
import org.apache.http.entity.mime.content.InputStreamBody;
import org.apache.http.entity.mime.content.StringBody;
import org.apache.http.impl.client.DefaultHttpClient;

import android.app.Activity;

public class TestMultipartPost extends Activity
{
    public void executeMultipartPost() throws Exception
    {
        try {
            InputStream is = this.getAssets().open("data.xml");
            HttpClient httpClient = new DefaultHttpClient();
            HttpPost postRequest =
                new HttpPost("http://jakisserverwersieciowy.com/uslugi/robCos.do");

            byte[] data = IOUtils.toByteArray(is);

            InputStreamBody isb = new InputStreamBody(new
ByteArrayListInputStream(data), "uploadedFile");
            StringBody sb1 = new StringBody("Wpisujemy Jakiś Tekst");
            StringBody sb2 = new StringBody("Również Wpisujemy Jakiś Tekst");

            MultipartEntity multipartContent = new MultipartEntity();
            multipartContent.addPart("uploadedFile", isb);
            multipartContent.addPart("one", sb1);
            multipartContent.addPart("two", sb2);

            postRequest.setEntity(multipartContent);
        }
    }
}
```

```

        HttpResponse response =httpClient.execute(postRequest);
        response.getEntity().getContent().close();
    } catch (Throwable e)
    {
        // tutaj obsługuje wyjątki
    }
}
}

```

**Uwaga!**

W powyższym przykładowym kodzie, gdzie zaprezentowano sposób korzystania z wieloczęściowej metody, użyto kilku plików *.jar*, które nie są dostępne w środowisku Android. Aby pliki te zostały dołączone do pakietu *.apk*, musimy dodać je jako zewnętrzne pliki *.jar* w środowisku Eclipse. W tym celu należy kliknąć prawym przyciskiem myszy nazwę projektu w programie Eclipse, wybrać opcję *Properties*, następnie *Java Class Path*, przejść do zakładki *Libraries* i zaznaczyć opcję *Add External JARs*.

Wykonanie tych czynności sprawi, że pliki te będą dostępne zarówno w trakcie komplikacji, jak i działania aplikacji.

Żeby aktywować wieloczęściową metodę POST, musimy wywołać moduł *HttpPost* i uruchomić jego metodę *setEntity()* wraz z instancją *MultiPartEntity* (zamiast obiektu *UrlEncodedFormEntity*, który tworzyliśmy dla pojedynczego parametru nazwa – wartość). Element *MultiPartEntity* reprezentuje treść żądania wywołania wieloczęściowej metody POST. Jak widać, najpierw tworzymy wystąpienie *MultiPartEntity*, a następnie wywołujemy metodę *addPart()* dla każdej części. Na listingu 11.4 dodano do żądania trzy części: dwa ciągi znaków oraz plik XML.

Jeżeli tworzymy aplikację wymagającą przekazania wieloczęściowej metody POST do zasobu sieciowego, prawdopodobnie zechemy sprawdzić nasze rozwiązanie pod kątem błędów, korzystając z pozoowanej implementacji usługi na lokalnej stacji roboczej. Podczas pracy z aplikacjami na stacji roboczej uzyskujemy dostęp do komputera lokalnego za pomocą polecenia *localhost* lub adresu IP 127.0.0.1. Jednak w przypadku aplikacji dla Androida nie będziemy korzystać z polecenia *localhost* (ani z adresu 127.0.0.1), ponieważ emulator będzie swoim własnym lokalnym hostem. Nie chcemy wskazywać tego klienta usługodawcy znajdującej się w urządzeniu obsługującym Androida, tylko nakierować na stację roboczą. Żeby uzyskać dostęp do swojej projektowej stacji roboczej z poziomu emulatora, musimy użyć jej adresu IP (w rozdziale 2. omówiliśmy sposób określenia adresu IP stacji roboczej). Na listingu 11.4 należy podstawić w miejsce widocznego adresu IP adres IP stacji roboczej Czytelnika.

## Parsery SOAP, JSON i XML

Co z protokołem SOAP? W internecie istnieje wiele usług sieciowych bazujących na protokole SOAP, ale do tej pory firma Google nie wprowadziła w Androide bezpośredniej obsługi wywoływanego tych usług. Preferowane są raczej usługi oparte na architekturze REST, co na pierwszy rzut oka powoduje zredukowanie ilości obliczeń w urządzeniu klienckim. Jednak z drugiej strony projektant musi poświęcić więcej pracy na wysyłanie danych oraz analizowanie otrzymanych informacji. Idealnym rozwiązaniem byłoby posiadanie opcji umożliwiających wybór sposobu interakcji z usługami sieciowymi. Niektórzy twórcy korzystają z zestawu projektowego kSOAP2 do tworzenia klientów opartych na protokole SOAP dla systemu Android. Nie będziemy się zajmować tą technologią, jednak jest ona dostępna dla zainteresowanych osób.

**Uwaga!**

Oryginalny kod zestawu kSOAP2 znajdziemy na stronie <http://ksoap2.sourceforge.net/>. Na szczęście społeczność ludzi piszących projekty o otwartym źródle nie śpi i utworzyła wersję zestawu kSOAP2 dla systemu Android. Więcej informacji na ten temat znajdziemy pod adresem: <http://code.google.com/p/ksoap2-android/>.

Jednym z najskuteczniejszych rozwiązań jest implementacja w internecie własnych usług, umożliwiających komunikowanie się protokołu SOAP (lub innego) z docelową usługą. Wtedy dana aplikacja musi kontaktować się tylko z określonymi usługami, a programista uzyskuje pełną kontrolę. Jeżeli usługi docelowe ulegają zmianom, możemy sobie z tym poradzić bez konieczności aktualizowania aplikacji i wydawania jej nowej wersji. Wystarczy, że zaktualizujemy usługi na serwerze. Dodatkową korzyścią jest fakt, że możemy w łatwy sposób wprowadzić model płatnej subskrypcji dla aplikacji. Jeżeli upłynie termin subskrypcji danego użytkownika, możemy go odłączyć z poziomu serwera.

Android obsługuje specyfikację JSON (ang. *JavaScript Object Notation* — notacja obiektów JavaScript). Jest to całkiem popularna metoda kompresji danych przesyłanych pomiędzy serwerem sieciowym a klientem. Klasy parserów JSON bardzo ułatwiają rozpakowywanie danych otrzymywanych w odpowiedzi, dzięki czemu nasza aplikacja może je przetwarzać. W dalszej części rozdziału, w trakcie opisywania interfejsu Tłumacza Google, zaprezentujemy kod zawierający notację JSON.

Android posiada także szereg parserów XML, za pomocą których możemy interpretować odpowiedzi uzyskiwane z wywołań HTTP. Główny parser (`XmlPullParser`) został omówiony w rozdziale 3.

## Obsługa wyjątków

Obsługa wyjątków jest częścią każdego programu, jednak podczas tworzenia oprogramowania wykorzystującego usługi zewnętrzne (na przykład usługi HTTP) trzeba zwracać szczególną uwagę na wyjątki, ponieważ prawdopodobieństwo występowania błędów zostaje zwielokrotnione. Istnieje kilka rodzajów wyjątków, jakich można się spodziewać podczas korzystania z usług HTTP. Do nich zalicza się wyjątki transportowe, wyjątki protokołowe oraz przekroczenia limitu czasu. Należy wiedzieć, kiedy te wyjątki mogą się pojawić.

Wyjątki transportowe mogą wystąpić z wielu różnych powodów, jednak w przypadku urządzeń mobilnych najczęstszym scenariuszem jest niska jakość połączenia sieciowego. Wyjątki protokołowe występują na poziomie warstwy protokołu HTTP. Należą do nich błędy uwierzytelniania, niewłaściwe pliki cookies i tak dalej. Możemy spodziewać się wyjątku protokołowego na przykład wtedy, gdy mamy dostarczyć poświadczenie tożsamości za pomocą identyfikatora użytkownika, a tego nie uczynimy. Pod kątem wywołań protokołu HTTP przekroczenia limitu czasu dzielimy na dwie kategorie: przekroczenie limitu czasu połączenia oraz przekroczenie limitu czasu gniazda. Przekroczenie limitu czasu połączenia następuje wtedy, gdy moduł `HttpClient` nie może ustawić połączenia z serwerem — na przykład jeśli serwer nie odpowiada. Przekroczenie limitu czasu gniazda występuje, gdy moduł `HttpClient` nie otrzyma odpowiedzi w określonym czasie. Innymi słowy, moduł ten zdołał się połączyć z serwerem, lecz serwer nie przekazał odpowiedzi w wyznaczonym limicie czasowym.

Skoro już znamy rodzaje pojawiających się wyjątków, to jak sobie z nimi poradzić? Na szczęście moduł `HttpClient` jest solidną strukturą, zdejmującą większość obowiązków z barków projektanta. Tak naprawdę jedynymi wyjątkami, jakich obsługę powinniśmy przewidzieć, są te, którymi najłatwiej zarządzać. Moduł `HttpClient` obsługuje wyjątki transportowe poprzez

wykrywanie problemów z przesyłaniem danych i ponawianie żądań (w przypadku tego typu wyjątków opisany sposób jest bardzo skuteczny). Wyjątki protokołowe są zazwyczaj usuwane w procesie projektowania. Prawdopodobnie najczęściej będziemy się zajmować przekroczeniami limitu czasu. Prostym i skutecznym sposobem radzenia sobie z obydwooma rodzajami przekroczeń limitu czasu — przekroczeniami czasu połączenia i przekroczeniami czasu gniazda — jest umieszczenie metody execute() żądania HTTP wewnętrz instrukcji try/catch i sprawdzenie, czy znowu wystąpi niepowodzenie. Zostało to zaprezentowane na listingu 11.5.

---

**Listing 11.5.** Implementacja prostej techniki ponawiania prób w przypadku przekroczeń limitu czasu

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URI;

import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;

public class TestHttpGet {

    public String executeHttpGetWithRetry() throws Exception {
        int retry = 3;

        int count = 0;
        while (count < retry) {
            count += 1;
            try {
                String response = executeHttpGet();
                /**
                 * Jeśli tutaj trafiemy, to znaczy, że próba przebiegła pomyślnie
                 * i możemy zakończyć.
                 */
                return response;
            } catch (Exception e) {
                /**
                 * Jeśli wyczerpalismy limit powtórzeń.
                 */
                if (count < retry) {
                    /**
                     * Mamy jeszcze powtórzenia, zatem wyświetlamy wiadomość
                     * i ponawiamy próbę.
                     */
                    System.out.println(e.getMessage());
                } else {
                    System.out.println("wszystkie próby nieudane... ");
                    throw e;
                }
            }
        }
        return null;
    }
}
```

```

}

public String executeHttpGet() throws Exception {
    BufferedReader in = null;
    try {
        HttpClient client = new DefaultHttpClient();
        HttpGet request = new HttpGet();
        request.setURI(new URI("http://code.google.com/android/"));
        HttpResponse response = client.execute(request);
        in = new BufferedReader(new InputStreamReader(response.getEntity()
            .getContent()));

        StringBuffer sb = new StringBuffer("");
        String line = "";
        String NL = System.getProperty("line.separator");
        while ((line = in.readLine()) != null) {
            sb.append(line + NL);
        }
        in.close();

        String result = sb.toString();
        return result;
    } finally {
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}

```

Kod na listingu 11.5 jest ilustracją możliwości zaimplementowania prostej techniki ponawiania prób w celu wywołania protokołu HTTP po przekroczeniu limitu czasu. Na listingu zostały ukazane dwie metody: pierwsza wykonuje funkcję HTTP GET (`executeHttpGet()`), druga natomiast umieszcza pierwszą metodę w algorytmie ponawiania prób (`executeHttpGetWithRetry()`). Algorytm ten jest niezmiernie prosty. Liczba powtórzeń prób przypisujemy wartość 3, a następnie wprowadzamy pętlę `while`. We wnętrzu tej pętli egzekwujemy żądanie. Zauważmy, że jest ono umieszczone w bloku `try/catch`, a w bloku `catch` sprawdzamy, czy nie wyczerpałyśmy limitu powtórzeń.

Podczas korzystania z modułu `HttpClient` jako części zwykłej aplikacji musimy poświęcić baczniejszą uwagę problemom wielowątkowości, które mogą się pojawić. Zajmiemy się nimi teraz.

## Problemy z wielowątkowością

W dotychczas ukazanych przykładach dla każdego żądania tworzyliśmy nowy moduł `HttpClient`. W praktyce jednak powinniśmy tworzyć jeden moduł `HttpClient` dla całej aplikacji i używać go podczas wszelkiej komunikacji poprzez protokół HTTP. Jeżeli jeden moduł `HttpClient` przetwarza wszystkie żądania protokołu HTTP, należy również uważać na problemy związane z wielowątkowością, które mogą się pojawić w trakcie jednoczesnego przetwarzania wielu

żądań przez ten moduł. Na szczęście posiada on funkcje ułatwiające to zadanie — wystarczy utworzyć obiekt DefaultHttpClient za pomocą klasy ThreadSafeClientConnManager, tak jak zostało to pokazane na listingu 11.6.

**Listing 11.6.** Utworzenie wielowątkowego modułu HttpClient

```
import org.apache.http.HttpVersion;
import org.apache.http.client.HttpClient;
import org.apache.http.conn.ClientConnectionManager;
import org.apache.http.conn.params.ConnManagerParams;
import org.apache.http.conn.scheme.PlainSocketFactory;
import org.apache.http.conn.scheme.Scheme;
import org.apache.http.conn.scheme.SchemeRegistry;
import org.apache.http.conn.ssl.SSLSocketFactory;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.impl.conn.tsccm.ThreadSafeClientConnManager;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpConnectionParams;
import org.apache.http.params.HttpParams;
import org.apache.http.params.HttpProtocolParams;
import org.apache.http.protocol.HTTP;

public class CustomHttpClient {
    private static HttpClient customHttpClient;

    /** Prywatny konstruktor zapobiega tworzeniu wystąpień */
    private CustomHttpClient() {
    }

    public static synchronized HttpClient getHttpClient() {
        if (customHttpClient == null) {
            HttpParams params = new BasicHttpParams();
            HttpProtocolParams.setVersion(params, HttpVersion.HTTP_1_1);
            HttpProtocolParams.setContentCharset(params,
                HTTP.DEFAULT_CONTENT_CHARSET);
            HttpProtocolParams.setUseExpectContinue(params, true);
            HttpProtocolParams.setUserAgent(params,
                "Mozilla/5.0 (Linux; U; Android 2.2.1; en-us; Nexus One Build/FRG83) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533.1");
        }

        ConnManagerParams.setTimeout(params, 1000);

        HttpConnectionParams.setConnectionTimeout(params, 5000);
        HttpConnectionParams.setSoTimeout(params, 10000);

        SchemeRegistry schReg = new SchemeRegistry();
        schReg.register(new Scheme("http",
            PlainSocketFactory.getSocketFactory(), 80));
        schReg.register(new Scheme("https",
            SSLSocketFactory.getSocketFactory(), 443));
        ClientConnectionManager conMgr = new
            ThreadSafeClientConnManager(params,schReg);

        customHttpClient = new DefaultHttpClient(conMgr, params);
    }
}
```

---

```

        return customHttpClient;
    }

    public Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }
}

```

---

Jeżeli aplikacja musi wykonywać więcej wywołań protokołu HTTP, należy utworzyć moduł `HttpClient` obsługujący wszystkie te żądania. Najprościej możemy tego dokonać poprzez utworzenie klasy singletonowej, do której dostęp będzie uzyskiwany z dowolnego miejsca aplikacji, tak jak pokazaliśmy powyżej. Jest to standardowy wzorzec w środowisku Java, za pomocą którego synchronizujemy dostęp do metody pobierania, a ta z kolei przekazuje tylko i wyłącznie jeden obiekt `HttpClient` dla klasy singletonowej (i w razie konieczności tworzy go za pierwszym razem).

Przyjrzyjmy się teraz metodzie `getHttpClient()` klasy `CustomHttpClient`. Jest ona odpowiedzialna za tworzenie naszego singletonowego modułu `HttpClient`. Wprowadziliśmy pewne podstawowe parametry, wartości przekroczenia czasu, a także schematy obsługiwane przez naszą klasę `HttpClient` (na przykład HTTP i HTTPS). Odnotujmy fakt, że podczas tworzenia metody `DefaultHttpClient()` przekazujemy obiekt klasy `ClientConnectionManager`. Jest ona odpowiedzialna za zarządzanie połączeniami HTTP modułu `HttpClient`. Ponieważ chcemy, żeby wszystkie połączenia HTTP przetwarzają pojedynczy moduł `HttpClient` (a to dlatego, że żądania mogą się nakładać na siebie w przypadku korzystania z wątku), tworzymy klasę `ThreadSafeClientConnManager`.

Prezentujemy także prostszy sposób uzyskiwania odpowiedzi z żądania HTTP za pomocą klasy `BasicResponseHandler`. Kod naszej aktywności wykorzystującej klasę `CustomHttpClient` został ukazany na listingu 11.7.

#### **Listing 11.7.** Korzystanie z klasy `CustomHttpClient` — plik `HttpActivity.java`

---

```

import java.io.IOException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.BasicResponseHandler;
import org.apache.http.params.HttpConnectionParams;
import org.apache.http.params.HttpParams;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class HttpActivity extends Activity
{
    private HttpClient httpClient;
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        httpClient = CustomHttpClient.getHttpClient();
        getHttpContent();
    }
}

```

```
}

public void getHttpContent()
{
    try {
        HttpGet request = new HttpGet("http://www.google.com/");
        String page = httpClient.execute(request,
            new BasicResponseHandler());
        System.out.println(page);
    } catch (IOException e) {
        // obejmuję:
        // ClientProtocolException
        // ConnectTimeoutException
        // ConnectionPoolTimeoutException
        // SocketTimeoutException
        e.printStackTrace();
    }
}
```

W tej przykładowej aplikacji przeprowadzamy proste pobranie strony głównej wyszukiwarki Google poprzez HTTP. Wykorzystujemy również obiekt `BasicResponseHandler` do przetwarzania strony w jeden olbrzymi ciąg znaków, który następnie zostaje wyświetlony w oknie `LogCat`. Jak widać, dodanie obiektu `BasicResponseHandler` do metody `execute()` jest banalnie prostą czynnością.

Być może poczujemy pukusę wykorzystania faktu, że każda aplikacja Androida posiada powiązany z nią obiekt `Application`. Jeżeli nie zdefiniujemy własnego obiektu aplikacji, Android domyślnie będzie korzystał z obiektu `android.app.Application`. Mała ciekawostka dotycząca obiektu aplikacji: dla danej aplikacji zawsze będzie istniał tylko jeden obiekt `Application` i wszystkie jej składniki mogą uzyskiwać do niego dostęp (za pomocą obiektu globalnego kontekstu). Istnieje możliwość rozszerzenia klasy `Application` oraz dodania nowych elementów, na przykład naszej klasy `CustomHttpClient`. W naszym przypadku jednak nie ma potrzeby, aby dokonywać tego w samej klasie `Application` i najlepiej będzie jej nie modyfikować, gdy możemy w łatwy i przyjemny sposób utworzyć oddzielną klasę singletonową, co rozwiąże nasze problemy.

## Zabawa z przekroczeniami limitu czasu

Istnieje również mnóstwo innych zalet wynikających z umieszczenia pojedynczego obiektu klasy `HttpClient` w naszej aplikacji. Możemy modyfikować jej właściwości w jednym miejscu i każdy może na tym skorzystać. Jeżeli na przykład chcemy skonfigurować wspólne wartości przekroczenia limitu czasu dla wywołań HTTP, możemy tego dokonać w trakcie tworzenia obiektu `HttpClient` poprzez wywołanie odpowiednich metod ustawiania wobec obiektu `HttpParams`. Widzimy to na listingu 11.6 i w metodzie `getHttpClient()`. Zwrócmy uwagę, że mamy wpływ na trzy rodzaje przekroczenia limitu czasu. Pierwszy z nich to przekroczenie limitu czasu w menedżerze połączeń — definiujemy tutaj czas oczekiwania, zanim połączenie zostanie usunięte z puli połączeń zarządzanych przez ten menedżer. W naszym przykładzie ustawiamy wartość przekroczenia czasu równą 1 sekundzie. Jedynym przypadkiem, w którym będziemy musieli czekać, jest moment, gdy wszystkie połączenia z puli są zajęte. Druga wartość przekroczenia limitu czasu określa, jak długo powinniśmy czekać, zanim

połączenie z serwerem zostanie zakończone. Daliśmy w tym przypadku 2 sekundy. Na końcu zdefiniowaliśmy czterosekundową wartość przekroczenia limitu czasu dla gniazda, co oznacza czas oczekiwania na otrzymanie żądanego danych.

W związku z trzema rodzajami przekroczenia limitu czasu opisanymi w poprzednim akapicie możemy wprowadzić trzy wyjątki: `ConnectionPoolTimeoutException`, `ConnectTimeoutException` albo `SocketTimeoutException`. Wszystkie trzy wyjątki są podelementami klasy `IIOException`, którą wykorzystaliśmy w klasie `HttpActivity`, dzięki czemu obyło się bez oddzielnego wprowadzania każdej podklasy wyjątku.

Jeżeli przeanalizujemy każdą klasę wykorzystaną w metodzie `getHttpClient()`, pozwalającą na konfigurowanie ustawień, bardzo możliwe, że znajdziemy tam o wiele więcej interesujących parametrów.

Opisaliśmy sposób konfigurowania wspólnej puli połączeń HTTP, wykorzystywanej przez całą aplikację. Możemy wywnioskować, że za każdym razem, gdy jest potrzebne połączenie, wymagania te zostaną zaspokojone dzięki różnorodnym ustawieniom. A co jeśli w przypadku konkretnego komunikatu potrzebne są odmienne ustawienia? Na szczęście i na to znajdzie się sposób. Pokazaliśmy mechanizm wykorzystania obiektów `HttpGet` oraz `HttpPost` do przesyłania żądania przez sieć. W podobny sposób, jak wykorzystujemy obiekt `HttpParams` wobec obiektu `HttpClient`, możemy go ustanowić również dla obiektów `HttpGet` oraz `HttpPost`. Ustawienia wprowadzane na poziomie komunikatu będą przesyłane ustawienia dostępne w obiekcie `HttpClient`, które jednak nie zostaną zmienione. Listing 11.8 przedstawia sytuację, w której chcielibyśmy zmienić wartość przekroczenia limitu czasu na gnieździe z 4 sekund na jedną minutę dla jednego, konkretnego żądania. Poniższy fragment kodu powinien zostać wprowadzony na miejsce wierszy znajdujących się w bloku `try` metody `getHttpContent()`, widocznej na listingu 11.7.

**Listing 11.8.** Przesłanianie na poziomie żądania wartości przekroczenia czasu na gnieździe

---

```
HttpGet request = new HttpGet("http://www.google.com/");
HttpParams params = request.getParams();
HttpConnectionParams.setSoTimeout(params, 60000); // 1 minuta
request.setParams(params);
String page = httpClient.execute(request,
        new BasicResponseHandler());
System.out.println(page);
```

---

## Stosowanie klasy `HttpURLConnection`

W Androidzie znajdziemy jeszcze jeden sposób radzenia sobie z usługami HTTP — korzystanie z klasy `java.net.HttpURLConnection`. Jest ona dość podobna do prezentowanych wcześniej klas `HttpClient`, jednak wymaga większej liczby instrukcji do działania. Wybór, z której klasy skorzystać, zależy od programisty.

## Używanie klasy `AndroidHttpClient`

W wersji 2.2 Androida wprowadzono nowy element potomny klasy `HttpClient`, noszący nazwę `AndroidHttpClient`. Zadaniem tej klasy jest ułatwienie pracy programistów aplikacji dla Androida poprzez zapewnienie domyślnych wartości i logiki zoptymalizowanych pod kątem

tego systemu. Na przykład wartości przekroczenia limitu czasu dla połączenia i gniazda (przykładowo operacji) posiadają domyślną wartość po 20 sekund. Menedżer połączenia używa wartości domyślne klasy ThreadSafeClientConnManager. W większości przypadków klasa ta może być stosowana wymiennie z pokazaną w poprzednich przykładach klasą HttpClient. Istnieje jednak kilka różnic, o których powinniśmy wiedzieć.

- Aby utworzyć klasę `AndroidHttpClient`, wywołujemy statyczną metodę `newInstance()` tejże klasy, na przykład w następujący sposób:

```
AndroidHttpClient httpClient = AndroidHttpClient.newInstance
    ("moj-ciag-znakow-agenta=http");

```

- Zwróciły uwagę, że parametrem metody `newInstance()` jest ciąg znaków agenta HTTP. W domyślnej przeglądarce Android ciąg ten wygląda następująco, ale możemy również dobrze skorzystać z odmiennych wartości:

*Mozilla/5.0 (Linux; U; Android 2.1; en-us; ADR6200 Build/ERD79) AppleWebKit/530.17 (KHTML, like Gecko) Version/ 4.0 Mobile Safari/530.17*

- Wywoływanie metody `execute()` wobec tego klienta musi następować z poziomu wątku różnego od głównego wątku interfejsu użytkownika. Oznacza to, że jeśli spróbujemy po prostu podmienić klasę `HttpClient` klasą `AndroidHttpClient`, wystąpi wyjątek. Przeprowadzanie wywołań HTTP w głównym wątku jest bardzo złym nawykiem, dlatego klasa `AndroidHttpClient` na to nie pozwala. W kolejnym podrozdziale będziemy zajmować się zagadnieniem wątków.
- Po zakończeniu używania wystąpienia klasy `AndroidHttpClient` musimy wywołać metodę `close()`. W ten sposób pamięć zostanie poprawnie zwolniona.
- Dostępnych jest kilka przydatnych metod, przetwarzających skompresowane odpowiedzi ze strony serwera, wśród nich takie jak:
  - `modifyRequestToAcceptGzipResponse(HttpRequest request)`,
  - `getCompressedEntity(byte[] data, ContentResolver resolver)`,
  - `getUngzippedContent(HttpEntity entity)`.

Po uzyskaniu wystąpienia klasy `AndroidHttpClient` nie możemy w nim zmienić ani dodać żadnego parametru (na przykład wersji protokołu HTTP). Nasze opcje mają służyć do przesyłania ustawień wewnętrz obiektu `HttpGet` (co pokazaliśmy już wcześniej) albo nie powinniśmy korzystać z klasy `AndroidHttpClient`.

Na tym zakończymy dyskusję dotyczącą przetwarzania usług HTTP za pomocą modułu `HttpClient`. W kolejnych podrozdziałach przedstawimy kolejny interesujący element Androida: tworzenie usług przetwarzanych w tle i nastawionych na długie działanie. Chociaż początkowo nie jest to oczywiste, procesy tworzenia wywołań protokołu HTTP i tworzenia usług w Androïdzie są ze sobą powiązane w taki sposób, że często będziemy musieli integrować wiele elementów za pomocą usług Androïda. Weźmy na przykład prostą aplikację pocztową. W urządzeniu obsługującym system Android taka aplikacja będzie się prawdopodobnie składała z dwóch części: jedna będzie dostarczać interfejs użytkownika, druga natomiast będzie sprawdzała, czy są dostępne nowe wiadomości. Proces sprawdzania będzie najprawdopodobniej przeprowadzany w tle. Składnik odpowiedzialny za sprawdzanie będzie usługą Androïda, czego skutkiem z kolei będzie wykorzystywanie w tym celu modułu `HttpClient`.

#### Uwaga!

Na stronie firmy Apache <http://hc.apache.org/httpcomponents-client-ga/tutorial/html/> znajdziemy świetny samouczek dotyczący klasy `HttpClient` oraz pozostałych koncepcji.

## Stosowanie wątków drugoplanowych (AsyncTask)

Do tej pory wykorzystywaliśmy w przykładach główny wątek aktywności do wywołań protokołu HTTP. Chociaż szczęśliwie można otrzymywać za każdym razem błyskawiczne odpowiedzi na wywołania, połączenia sieciowe i internetowe nie muszą być wcale takie szybkie. Ponieważ główny wątek aktywności przede wszystkim ma przetwarzanie zdarzenia generowane przez użytkownika (np. kliknięcia) oraz aktualizować interfejs użytkownika, do przeprowadzania operacji mogących zajmować trochę więcej czasu powinniśmy używać wątków drugoplanowych. Android niejako wymusza na programistach takie rozwiązanie, ponieważ jeżeli główny wątek zatrzyma swoje działanie na czas pięciu sekund, zostanie uruchomiony warunek ANR (ang. *Application Not Responding* — aplikacja nie odpowiada), który ostatecznie zakończy działanie programu, gdy pojawi się brzydkie okno dialogowe zawierające monit o potwierdzenie zamknięcia aplikacji (jest to tak zwane wymuszone zamknięcie). W rozdziale 13. przyjrzymy się dokładniej głównemu wątkowi, na razie jednak wystarczy nam wiedzieć, że nie możemy zajmować głównego wątku jedną czynnością przez dłuższy czas.

Jeżeli jedynym zadaniem w danej chwili jest dokonywanie obliczeń bez konieczności aktualizowania interfejsu użytkownika, moglibyśmy zwyczajnie wykorzystać prosty obiekt Thread, który odciążyłby nieco główny wątek. Technika ta jednak nie jest odpowiednia, jeśli trzeba aktualizować interfejs użytkownika. Dlatego właśnie zestaw narzędziowy interfejsu użytkownika w Androidzie nie jest zabezpieczony przed wątkami. A zatem powinien być on zawsze aktualizowany wyłącznie z poziomu głównego wątku.

Jeżeli zamierzamy aktualizować interfejs użytkownika w jakikolwiek sposób w wyniku działania wątku drugoplanowego, powinniśmy się poważnie zastanowić nad implementacją klasy AsyncTask. Umożliwia ona wygodne przenoszenie do drugiego planu pewnych operacji, które wymagają aktualizowania interfejsu użytkownika. Klasa AsyncTask zapewnia utworzenie wątku drugoplanowego, w którym będą przeprowadzane dane operacje, a także działających w głównym wątku wywołań zwrotnych, dających łatwy dostęp do elementów interfejsu użytkownika (np. widoków). Wywołania te mogą zostać uruchomione przed, w trakcie lub po wyłączeniu wątku drugoplanowego.

Rozważmy na przykład problem pobierania obrazu z serwera sieciowego, który będzie wyświetlane w naszej aplikacji. Być może obraz ten będzie musiał być rekonstruowany w locie. Nie możemy zagwarantować, po jakim czasie obraz ten zostanie wyświetlony użytkownikowi, więc w tym celu rzeczywiście trzeba zastosować wątek drugoplanowy.

Listing 11.9 przedstawia prostą implementację klasy AsyncTask, która wykona za nas całą brudną robotę. Najpierw ją omówimy, a następnie zaprezentujemy plik układu graficznego oraz kod Java aktywności wywołująccej tę funkcję.

**Listing 11.9.** Klasa AsyncTask odpowiedzialna za pobieranie obrazu — plik DownloadImageTask.java

---

```
import java.io.IOException;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpConnectionParams;
import org.apache.http.params.HttpParams;
import org.apache.http.util.EntityUtils;
import android.app.Activity;
import android.content.Context;
```

```
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.util.Log;
import android.widget.ImageView;
import android.widget.TextView;

public class DownloadImageTask extends AsyncTask<String, Integer, Bitmap> {
    private Context mContext;

    DownloadImageTask(Context context) {
        mContext = context;
    }

    protected void onPreExecute() {
        // Moglibyśmy tutaj wprowadzić jakieś czynności konfiguracyjne przed uruchomieniem
        // metody doInBackground()
    }

    protected Bitmap doInBackground(String... urls) {
        Log.v("doInBackground", "pobieranie obrazu");
        return downloadImage(urls);
    }

    protected void onProgressUpdate(Integer... progress) {
        TextView mText = (TextView)
            ((Activity) mContext).findViewById(R.id.text);
        mText.setText("Dotychczasowy postęp: " + progress[0]);
    }

    protected void onPostExecute(Bitmap result) {
        if(result != null) {
            ImageView mImage = (ImageView)
                ((Activity) mContext).findViewById(R.id.image);
            mImage.setImageBitmap(result);
        }
        else {
            TextView errorMsg = (TextView)
                ((Activity) mContext).findViewById(R.id.errorMsg);
            errorMsg.setText("Problem z pobraniem obrazu. Prosimy spróbować później.");
        }
    }
}

private Bitmap downloadImage(String... urls)
{
    HttpClient httpClient = CustomHttpClient.getHttpClient();
    try {
        HttpGet request = new HttpGet(urls[0]);
        HttpParams params = new BasicHttpParams();
        HttpConnectionParams.setSoTimeout(params, 60000); // 1 minuta
        request.setParams(params);

        publishProgress(25);

        HttpResponse response = httpClient.execute(request);

        publishProgress(50);
```

```

byte[] image = EntityUtils.toByteArray(response.getEntity());

publishProgress(75);

Bitmap mBitmap = BitmapFactory.decodeByteArray(
    image, 0, image.length);

publishProgress(100);

return mBitmap;
} catch (IOException e) {
// Obejmuje:
// ClientProtocolException
// ConnectTimeoutException
// ConnectionPoolTimeoutException
// SocketTimeoutException
e.printStackTrace();
}
return null;
}
}

```

Ponieważ AsyncTask jest klasą abstrakcyjną, musimy ją dostosować za pomocą rozszerzania, w czym pomocna okazuje się klasa DownloadImageTask. Wykorzystamy tu konstruktor pobierający odniesienie do wywołującego kontekstu, który w naszym przypadku będzie wywołującą aktywnością. Kontekst ten posłuży nam do uzyskania dostępu do widoków aktywności. Używamy tu również znanej z wcześniejszych przykładów klasy CustomHttpClient.

Przygotowanie funkcji AsyncTask do pracy składa się z czterech etapów:

1. Wszelkie działania konfiguracyjne przeprowadzamy w metodzie `onPreExecute()`. Jest ona wykonywana w głównym wątku.
2. Za pomocą metody `doInBackground()` uruchamiamy wątek drugoplanowy. Samo tworzenie wątku nie wymaga naszego udziału. Kod ten jest wykonywany w oddzielnym wątku drugoplanowym.
3. Aktualizujemy postępy za pomocą metod `publishProgress()` i `onProgressUpdate()`. Ta pierwsza metoda zostaje wywołana z wnętrza kodu metody `doInBackground()`, podczas gdy ta druga jest wykonywana w głównym wątku jako wynik wywołania metody `publishProgress()`. Za pomocą tych dwóch metod uruchomiony wątek drugoplanowy może komunikować się z wątkiem głównym, zatem w interfejsie użytkownika mogą być dokonywane aktualizacje stanu, jeszcze zanim wątek drugoplanowy zakończy swoje zadanie.
4. Interfejs użytkownika zostaje zaktualizowany o wyniki za pomocą metody `onPostExecute()`. Metoda ta jest wykonywana w głównym wątku.

Etapy 1. i 3. są opcjonalne. W naszym przykładzie nie przeprowadziliśmy procesu inicjalizacji w metodzie `onPreExecute()`, jednak zastosowaliśmy mechanizm aktualizacji postępów omówiony w punkcie 3. Zasadnicza praca wątku drugoplanowego jest wykonywana przez metodę `downloadImage()` wywoływaną z poziomu metody `doInBackground()`. Metoda `downloadImage()` przyjmuje adres URL i wykorzystuje klasę `HttpClient` do wykonania żądania metody `HttpGet` oraz uzyskania odpowiedzi. Zwrócmy uwagę, że czas wygaśnięcia ustawiamy na 60 sekund bez obawy o wystąpienie warunku ANR. Widzimy w kodzie, że postępy są aktualizowane na etapach konfigurowania połączenia `HttpClient`, wykonywania żądania HTTP,

konwertowania odpowiedzi na tablicę bajtów i tworzenia z niej obiektu klasy `Bitmap`. Kiedy metoda `downloadImage()` powraca do metody `doInBackground()`, która z kolei również powraca, Android zajmuje się pobraniem otrzymanej wartości i przekazaniem jej do metody `onPostExecute()`. Gdy zostanie do niej przekazany obiekt klasy `Bitmap`, można za jego pomocą bezpiecznie zaktualizować widok `ImageView`, ponieważ metoda ta działa w głównym wątku aktywności. Co jednak w przypadku wystąpienia jakiegoś wyjątku na etapie pobierania? Jeżeli nie otrzymamy obrazu z wywołania HTTP, a zamiast niego pojawi się wyjątek, obiekt `Bitmap` uzyska wartość `null`. Wykrywamy ten fakt w metodzie `onPostExecute()` i wyświetlamy komunikat o błędzie, nie próbujemy natomiast przydzielić obiektu klasy `Bitmap` do widoku `ImageView`. Oczywiście, możemy również wykonać jakąś inną czynność, jeśli wiemy, że proces pobierania został zakończony niepowodzeniem.

Musimy pamiętać, że jedynie kod umieszczony w metodzie `doInBackground()` jest wykonywany poza głównym wątkiem. Nie możemy więc pracować z interfejsem użytkownika we wnętrzu tej metody, ponieważ w przeciwnym wypadku możemy spowodować problemy. Na przykład nie należy wywoływać z jej wnętrza metod modyfikujących elementy interfejsu użytkownika. Elementy interfejsu użytkownika możemy modyfikować jedynie w metodach `onPreExecute()`, `onProgressUpdate()` oraz `onPostExecute()`.

Uzupełnijmy nasz najnowszy przykład o plik układu graficznego oraz kod Java aktywności, które zostaną umieszczone, odpowiednio, na listingach 11.10 i 11.11.

---

**Listing 11.10.** Układ graficzny służący do wywoływania klasy `AsyncTask` — plik `/res/layout/main.xml`

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical"
    >
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    >
<Button android:id="@+id/button" android:text="Pobierz obraz"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="doClick"
    />
<TextView android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />
</LinearLayout>
<TextView android:id="@+id/errorMsg" android:textColor="#ff0000"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />
<ImageView android:id="@+id/image"
    android:layout_width="fill_parent" android:layout_height="0dp"
    android:layout_weight="1" />
</LinearLayout>
```

---

**Listing 11.11.** Aktywność wywołująca klasę AsyncTask — plik HttpActivity.java

```
import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class HttpActivity extends Activity {
    private DownloadImageTask diTask;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

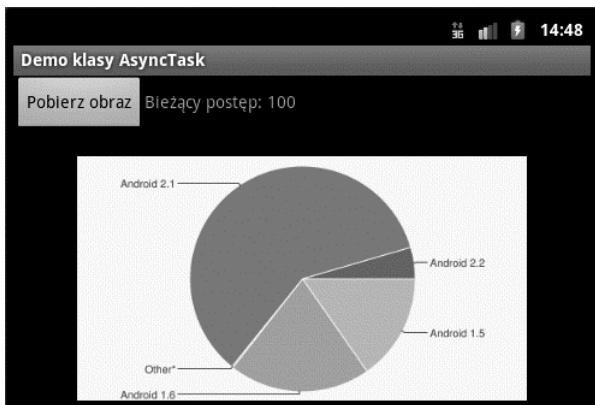
    public void doClick(View view) {
        if(diTask != null) {
            AsyncTask.Status diStatus = diTask.getStatus();
            Log.v("doClick", "status diTask wynosi" + diStatus);
            if(diStatus != AsyncTask.Status.FINISHED) {
                Log.v("doClick", "...nie ma potrzeby, aby uruchomić nowe zadanie");
                return;
            }
            // Ponieważ diStatus musi posiadać wartość FINISHED, możemy spróbować ponownie.
        }
        diTask = new DownloadImageTask(this);

        diTask.execute("http://chart.apis.google.com/chart?&cht=p&chs=460x250&chd=t:15.3,
        ↪20.3,0.
        2,59.7,4.5&chl=Android%201.5%7CAndroid%201.6%7Cother*%7CAndroid%202.1%7CAndroid%
        ↪202.2&ch
        co=c4df9b,6fad0c");
    }
}
```

Po uruchomieniu aplikacji i naciśnięciu przycisku ujrzymy ekran przedstawiony na rysunku 11.1.

Układ graficzny nie jest skomplikowany. Widzimy przycisk oraz obok niego komunikat tekstowy. Tekst ten będzie naszym wskaźnikiem postępów. Pod spodem mamy miejsce na komunikat o błędzie, który zostanie wyświetlony czerwoną czcionką. Na końcu znajduje się obszar przeznaczony dla rysunku.

Wewnątrz metody zwrotnej przycisku `doClick()` musimy utworzyć nowe wystąpienie naszej skonfigurowanej klasy `AsyncTask` oraz wywołać metodę `execute()`. Jest to bardzo powszechny wzorzec. Tworzymy wystąpienie rozszerzenia klasy `AsyncTask` i wywołujemy metodę `execute()`. W naszym przykładzie wywołujemy usługę tworzenia wykresów firmy Google, która przyjmuje wartości danych, nazwy etykiet i tworzy z nich wykres, który jest prezentowany w postaci obrazu PNG. Zanim jednak uruchomimy nasze zadanie, powinniśmy starannie sprawdzić, czy nie zostało już uruchomione. Jeżeli użytkownik kliknie przycisk dwukrotnie, może się okazać, że w tle są przetwarzane dwa zadania. Na szczęście klasa `AsyncTask` pozwala nam sprawdzić stan zadania. Jeżeli wartość zmiennej `diTask` nie wynosi `null`, istnieje prawdopodobieństwo, że



**Rysunek 11.1.** Wykorzystanie klasy AsyncTask do pobrania obrazu (wykres dostępności poszczególnych wersji Androida na rynku na dzień 2 sierpnia 2010 roku)

jakieś zadanie jest już uruchomione. Sprawdzamy więc stan klasy AsyncTask. Jeżeli ma on jakąkolwiek inną wartość niż FINISHED, to znaczy, że zadanie jest uruchomione (RUNNING) lub oczekujące (PENDING) i gotowe do uruchomienia. A zatem jeśli dla danego zadania klasa AsyncTask pozostaje w stanie FINISHED, możemy je uznać za zakończone i utworzyć nową klasę AsyncTask. Oczywiście, jeśli poprzednie wystąpienie klasy AsyncTask mogło z powodzeniem pobrać obraz, możemy nie chcieć pobierać go ponownie. Jednak w naszym przykładzie pozwalamy na jego ponowne pobranie.

W trakcie działania naszej przykładowej aplikacji po wcisnięciu przycisku pojawi się aktualizowany komunikat o postępach pracy, a po chwili rysunek. Przycisk przechodzi ze stanu kliknięcia w stan normalny, jeszcze *zanim* komunikat o postępach zacznie się aktualizować. Jest to bardzo ważna obserwacja, ponieważ oznacza ona, że główny wątek powrócił do zarządzania interfejsem użytkownika w trakcie pobierania obrazu.

Z czystej ciekawości przejdźmy do adresu URL kierującego aplikację na stronę z wykresami i wprowadźmy jakąś zmianę, która „zepsuje” ten ciąg znaków. Włączmy teraz ponownie aplikację. Wynik powinien być podobny do przedstawionego na rysunku 11.2.



**Rysunek 11.2.** Komunikat o wyjątku przekazany do interfejsu użytkownika

Warto dowiedzieć się jeszcze kilku rzeczy na temat klasy AsyncTask. Po utworzeniu rozszerzenia klasy AsyncTask i uruchomieniu metody execute() nasz główny wątek wraca do przetwarzania kodu. Ciągle jednak posiadamy odniesienie do zadania i możemy na nim działać z poziomu głównego wątku. Na przykład możemy wywołać metodę cancel(), aby zakończyć to zadanie. Za pomocą metody isCancelled() sprawdzamy, czy zostało ono anulowane. Możemy chcieć zmodyfikować logikę w metodzie onPostExecute() pod kątem anulowania zadania. Z kolei klasa AsyncTask zawiera dwie odmiany metody get(), w których wynik otrzymujemy z metody doInBackground(), bez konieczności wykorzystywania metody

`onPostExecute()`. Jedna z tych odmian metody `get()` blokuje korzystanie z metody `onPostExecute()`, podczas gdy druga korzysta z wartości przekroczenia limitu czasu — zapobiega w ten sposób zbyt długiemu oczekiwaniu przez wątek wywołujący.

Klasa `AsyncTask` może być uruchomiona tylko jednokrotnie. Jeśli więc przechowujemy odniesienie do tej klasy, nie wywołujmy metody `execute()` więcej niż raz, w przeciwnym wypadku zostanie wyświetlona informacja o wyjątku. Nic nie stoi na przeszkodzie, aby utworzyć nowe wystąpienia klasy `AsyncTask`, jednak każda z tych instancji będzie uruchomiona tylko jednokrotnie. Dlatego za każdym razem tworzymy nowe wystąpienie klasy `DownloadImageTask`, gdy jest nam ono potrzebne.

## Obsługa zmian konfiguracji za pomocą klasy `AsyncTask`

Istnieje jednak jeszcze jedna niezwykle ważna cecha klasy `AsyncTask`, o której musimy wiedzieć. Mianowicie nie zadziała ona, jeżeli uruchamiająca ją aktywność zostanie zakończona lub ponownie utworzona. To naprawdę istotna rzecz. Oczywiście, jeżeli wywołanie zwrotne metody `onPostExecute()` zostanie przeprowadzone wobec pierwotnej aktywności, która jednak w aplikacji została zastąpiona przez nową aktywność, klasa `AsyncTask` będzie aktualizowała widoki niewidoczne dla użytkownika. Jak to możliwe, że aktywność zostanie zakończona i ponownie odtworzona? Tak naprawdę procesy te zachodzą przez cały czas. Po każdej modyfikacji konfiguracji urządzenia, na przykład po zmianie trybu wyświetlania obrazu z pionowego na poziomy, aktywność zostanie zakończona i odtworzona.

Warto pamiętać, że w czasie tworzenia interfejsu użytkownika Android określa, za pomocą konfiguracji urządzenia, które układy graficzne i zasoby mają zostać wykorzystane. Jest to dość skomplikowany mechanizm, więc najłatwiejszym i najszybszym rozwiązaniem dla systemu jest zakończenie bieżącej aktywności i odtworzenie jej za pomocą nowej konfiguracji. Na szczęście nie wszystkie dotychczas utworzone obiekty zostają utracone, zakończeniu nie ulega działanie m.in. klasy `AsyncTask`. Ponieważ klasa ta jest przetwarzana w osobnym wątku aplikacji, ciągle jest dostępna w trakcie ponownego tworzenia aktywności. Naszym zadaniem jest utworzenie połączenia pomiędzy nimi, tak aby klasa `AsyncTask` odnalazła widoki mieszczące się w nowej aktywności. Aktywność posiada, odpowiedzialne za to zadanie, funkcję zwrotną oraz metody. Są to, odpowiednio, `onRetainNonConfigurationInstance()` oraz `getLastNonConfigurationInstance()`. Zasadniczo obiekty te zapewniają przekazywanie obiektu ze starej aktywności do nowej (listing 11.12).

**Listing 11.12.** Nowa wersja pliku `HttpActivity.java`. Plik obsługuje proces rekonfiguracji

---

```
import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class HttpActivity extends Activity {
    private DownloadImageTask diTask;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
```

```
// Poniższy fragment sprawdza, czy uruchamiamy ponownie aktywność
// w obecności klasy AsyncTask. Jeśli tak, zostaje ponownie ustanowione połaczenie.
// Poza tym w przypadku zakończenia działania klasy AsyncTask obraz nie został
// zachowany w trakcie cyklu kończenia/tworzenia aktywności, należy więc
// pobrać ten obraz z klasy AsyncTask.
if( (diTask =
    (DownloadImageTask)getLastNonConfigurationInstance() )
    != null)
{
    diTask.setContext(this); // Daje klasie AsyncTask nowe
                           // odniesienie do aktywności.
    if(diTask.getStatus() == AsyncTask.Status.FINISHED)
        diTask.setImageInView();
}
}

public void doClick(View view) {
    if(diTask != null) {
        AsyncTask.Status diStatus = diTask.getStatus();
        Log.v("doClick", "Stan diTask wynosi " + diStatus);
        if(diStatus != AsyncTask.Status.FINISHED) {
            Log.v("doClick", "...nie trzeba uruchamiać nowego zadania");
            return;
        }
        // Ponieważ diStatus musi posiadać wartość FINISHED, możemy spróbować ponownie.
    }
    diTask = new DownloadImageTask(this);

diTask.execute("http://chart.apis.google.com/chart?&cht=p&chs=460x250&chd=t:15.3,
    ↳20.3,0.
2,59.7,4.5&chl=Android%201.5%7CAndroid%201.6%7C0ther*%7CAndroid%202.1%7CAndroid%
    ↳202.2&ch
co=c4df9b,6fad0c");
}

// Metoda ta zostaje wywołana przed metodą onDestroy(). Chcemy zawczasu przekazać
// odniesienie do klasy AsyncTask.
@Override
public Object onRetainNonConfigurationInstance() {
    return diTask;
}
}
```

---

Powyższy przykładowy kod jest bardzo podobny do aktywności `HttpActivity` widocznej na listingu 11.11. Różnica między nimi polega na wywołaniu metody `getLastNonConfigurationInstance()` w celu sprawdzenia, czy w starym wystąpieniu tej aktywności znajduje się obiekt `DownloadImageTask`. Jeżeli istnieje taki obiekt, musimy przekazać mu odniesienie do nowej aktywności i w ten sposób dać mu dostęp do nowych widoków. Na listingu 11.13 ujrzymy zmodyfikowany kod klasy `DownloadImageTask`. Po ustanowieniu nowego kontekstu w klasie `DownloadImageTask` sprawdzamy, czy zadanie zostało zakończone, ponieważ mogło się tak stać w trakcie odtwarzania klasy `HttpActivity`. Jeśli stwierdzimy zakończenie zadania, stosujemy metodę `setImageInView()` do zaktualizowania obrazu — więcej informacji na ten temat znajdzie się poniżej.

**Listing 11.13.** Zmodyfikowany plik DownloadImageTask.java, obsługujący zmianę konfiguracji urządzenia

```
import java.io.IOException;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.params.BasicHttpParams;
import org.apache.http.params.HttpConnectionParams;
import org.apache.http.params.HttpParams;
import org.apache.http.util.EntityUtils;
import android.app.Activity;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.util.Log;
import android.widget.ImageView;
import android.widget.TextView;

public class DownloadImageTask extends AsyncTask<String, Integer, Bitmap> {
    private Context mContext; // Odniesienie do wywołującej aktywności
    int progress = -1;
    Bitmap downloadedImage = null;

    DownloadImageTask(Context context) {
        mContext = context;
    }

    // Wywoływanie z głównego wątku w celu ponownego połączenia
    protected void setContext(Context context) {
        mContext = context;
        if(progress >= 0) {
            publishProgress(this.progress);
        }
    }

    protected void onPreExecute() {
        progress = 0;
        // Możemy tutaj wprowadzić inne ustawienia konfiguracyjne,
        // zanim zostanie uruchomiona metoda doInBackground()
    }

    protected Bitmap doInBackground(String... urls) {
        Log.v("doInBackground", "pobieranie obrazu...");
        return downloadImage(urls);
    }

    protected void onProgressUpdate(Integer... progress) {
        TextView mText = (TextView)
            ((Activity) mContext).findViewById(R.id.text);
        mText.setText("Dotychczasowy postęp: " + progress[0]);
    }

    protected void onPostExecute(Bitmap result) {
        if(result != null) {
```

```
        downloadedImage = result;
        setImageInView();
    }
    else {
        TextView errorMsg = (TextView)
            ((Activity) mContext).findViewById(R.id.errorMsg);
        errorMsg.setText("Wystąpił problem z pobieraniem pliku. Prosimy spróbować
                        później.");
    }
}

public Bitmap downloadImage(String... urls)
{
HttpClient httpClient = CustomHttpClient.getHttpClient();
try {
    HttpGet request = new HttpGet(urls[0]);
    HttpParams params = new BasicHttpParams();
    HttpConnectionParams.setSoTimeout(params, 60000); // 1 minuta
    request.setParams(params);

    setProgress(25);

    HttpResponse response = httpClient.execute(request);

    setProgress(50);

    sleepFor(5000); // 5 sekund hibernacji

    byte[] image = EntityUtils.toByteArray(response.getEntity());

    setProgress(75);

    Bitmap mBitmap = BitmapFactory.decodeByteArray(image, 0,
                                                image.length);

    setProgress(100);

    return mBitmap;
} catch (IOException e) {
    // Obejmuje:
    // ClientProtocolException
    // ConnectTimeoutException
    // ConnectionPoolTimeoutException
    // SocketTimeoutException
    e.printStackTrace();
}
return null;
}

private void setProgress(int progress) {
    this.progress = progress;
    publishProgress(this.progress);
}

protected void setImageInView() {
    if(downloadedImage != null) {
        ImageView mImage = (ImageView)
```

```

        ((Activity) mContext).findViewById(R.id.image);
        mImage.setImageBitmap(downloadedImage);
    }
}

private void sleepFor(long msecs) {
    try {
        Thread.sleep(msecs);
    } catch (InterruptedException e) {
        Log.v("sleep", "przerwano");
    }
}
}

```

---

Powinniśmy zwrócić uwagę, że procedura obsługi przycisku `onClick()` pozostaje niezmieniona. Teraz jednak posiadamy implementację opartą na funkcji `onRetainNonConfiguration→Instance()`, której jedynym zadaniem jest przekazywanie zwracanego obiektu. W naszym przypadku zależy nam wyłącznie na obiekcie `DownloadImageTask`, więc tylko ten obiekt musimy przekazać. Gdybyśmy chcieli przekazać więcej elementów, musielibyśmy utworzyć obiekt przechowujący je wszystkie, a następnie przekazać ten obiekt. Mamy do czynienia z podstawowymi obiektami środowiska Java, nie musimy więc przejmować się serializacją ani obiektami typu `Parcel` (w dalszej części rozdziału zajmiemy się tymi obiektami). Klasę `AsyncTask` przekazujemy wyłącznie po to, aby ją wykorzystać nieco później. Możemy więc ponownie nawiązać z nią połączenie.

Zachęcamy teraz do uruchomienia naszej przykładowej aplikacji. Dodaliśmy pięciosekundowe opóźnienie do klasy `AsyncTask`, aby dać Czytelnikowi czas na obrócenie ekranu w trakcie działania drugoplanowego zadania. Aby zasymulować obrót ekranu urządzenia za pomocą emulatora, używamy skrótu klawiaturowego `Ctrl+F11`. Interfejs użytkownika powinien się obrócić i odpowiednio dostosować do położenia wyświetlacza. Z każdym obrotem pobrany obraz znika (jeśli był widoczny) i pojawia się ponownie w trakcie przetwarzania kodu. Warto obracać ekran na różnych etapach działania aplikacji, aby sprawdzić efekty. Możemy nawet wrócić do poprzedniego przykładu, dodać opóźnienie i przekonać się, że w trakcie obracania *nie* zachowuje się on zgodnie z życzeniem użytkownika. Przyjrzymy się teraz nowemu kodowi, aby zrozumieć mechanizm jego działania.

Tym razem klasa rozszerzająca klasę `AsyncTask` jest nieco inna.

W trakcie zmiany konfiguracji dzieje się kilka rzeczy, z którymi musimy sobie poradzić. Przede wszystkim trzeba wiedzieć, że klasa `AsyncTask` musi otrzymać nowe odniesienie do aktywności, aby móc aktualizować odpowiednie widoki zarówno w metodzie `onProgressUpdate()`, jak i `onPostExecute()`. Po zakończeniu starej aktywności odniesienie do niej staje się bezużyteczne i potrzebujemy nowego. Gdybyśmy zmienili jakiś element w interfejsie użytkownika umieszczonym w metodzie `onPreExecute()`, musielibyśmy również w niej zamieścić odniesienie do nowej aktywności. Teraz można wykorzystać metodę nazwaną `setContext()`, dzięki której kontekst jest aktualizowany wraz z pozostałymi obiektami, więc w razie potrzeby nie powinniśmy mieć problemów ze znalezieniem widoków.

Poza tym nieco inaczej zapewniamy obsługę aktualizacji postępów. W dalszym ciągu korzystamy z obiektu postępów, do którego możemy się odnosić w metodzie `setContext()`, a także w metodzie `setProgress()`. Wywołujemy teraz metodę `setProgress()` w odpowiednich

miejscach metody `downloadImage()`. Gdy w końcu połączymy się z nową aktywnością, chcemy natychmiast wyświetlić aktualny postęp pobierania obrazu, zatem ustanawiamy metodę `publishProgress()` w metodzie `setContext()`.

Wreszcie, obrazy nie są utrzymywane w trakcie cyklu zakańczania – odtwarzania aktywności. Jeżeli aktywność zostanie odtworzona przed zakończeniem działania klasy `AsyncTask`, wszystko będzie w porządku, ponieważ metoda `onPostExecute()` ustawi nową bitmapę. Jeśli jednak funkcja `AsyncTask` została zakończona jakiś czas temu, a my obróciśmy wyświetlacz, aktywność zostanie odtworzona, jednak obraz nie zostanie wyświetlony. Moglibyśmy ponownie pobrać obraz z serwera, jednak w naszym przykładzie zatrzymaliśmy tę bitmapę za pomocą nowego członka `downloadedImage` oraz wprowadziliśmy chronioną metodę `setImageInView()`, która łączyła rysunek do obiektu `ImageView`. Jak już wcześniej powiedzieliśmy, nie chcemy przechowywać elementu interfejsu użytkownika (na przykład klasy `View`) wewnątrz klasy `AsyncTask`. Dlatego właśnie przechowujemy bitmapę, a nie widok `ImageView`. Nie chcemy, żeby nastąpił wyciek pamięci poprzez odniesienia do widoków utworzonych w starej aktywności.

## Pobieranie plików za pomocą klasy `DownloadManager`

W pewnych warunkach aplikacja może pobierać większy plik na urządzenie. Ponieważ ten proces może zająć nieco czasu, a kod, który go obsługuje, mógł zostać znormalizowany, w wersji 2.3 Androida wprowadzono specjalną klasę, której jedynym zadaniem jest pobieranie dużych plików — `DownloadManager`. `DownloadManager` pobiera duże pliki na lokalny nośnik urządzenia, działając w wątku drugoplanowym. Istnieje możliwość skonfigurowania tej klasy w taki sposób, żeby wyświetlała użytkownikowi powiadomienie o procesie pobierania.

W kolejnej prezentowanej aplikacji wykorzystujemy klasę `DownloadManager` do pobrania jednego z plików zestawu Android SDK. Ten przykładowy projekt składa się z następujących plików:

- `res/layout/main.xml` (listing 11.14),
- `MainActivity.java` (listing 11.15),
- `AndroidManifest.xml` (listing 11.16).

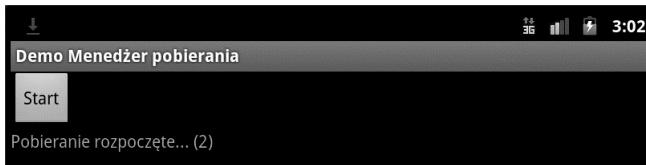
**Listing 11.14.** Korzystanie z klasy `DownloadManager` — plik `/res/layout/main.xml`

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <Button android:onClick="doClick" android:text="Start"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView android:id="@+id/tv"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

---

Na układ graficzny projektu składa się jeden przycisk i jedno pole tekstowe. Wciśnięcie przycisku spowoduje rozpoczęcie pobierania, a w polu tekstowym będą wyświetlane informacje dotyczące rozpoczęcia oraz zakończenia tego procesu. Interfejs użytkownika został zaprezentowany na rysunku 11.3.



**Rysunek 11.3.** Interfejs użytkownika w aplikacji DownloadManagerDemo

Z kolei listing 11.15 przedstawia kod Java naszej aplikacji.

**Listing 11.15.** Korzystanie z klasy DownloadManager — plik MainActivity.java

```

import android.app.Activity;
import android.app.DownloadManager;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class MainActivity extends Activity {
    protected static final String TAG = "DownloadMgr";
    private DownloadManager dMgr;
    private TextView tv;
    private long downloadId;

    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        tv = (TextView) findViewById(R.id.tv);
    }

    @Override
    protected void onResume() {
        super.onResume();
        dMgr = (DownloadManager) getSystemService(DOWNLOAD_SERVICE);
    }

    public void doClick(View view) {
        DownloadManager.Request dmReq = new DownloadManager.Request(
            Uri.parse(
                "http://dl-ssl.google.com/android/repository/" +
                "platform-tools_r01-linux.zip"));
        dmReq.setTitle("Narzędzia dla platformy");
        dmReq.setDescription("Wersja dla systemu Linux");
        dmReq.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_MOBILE);

        IntentFilter filter = new
        IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);
    }
}

```

```
registerReceiver(mReceiver, filter);

downloadId = dMgr.enqueue(dmReq);

tv.setText("Pobieranie rozpoczęte... (" + downloadId + ")");

}

public BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        Bundle extras = intent.getExtras();
        long doneDownloadId =
            extras.getLong(DownloadManager.EXTRA_DOWNLOAD_ID);
        tv.setText(tv.getText() + "\nPobieranie zakończone (" +
            doneDownloadId + ")");
        if(downloadId == doneDownloadId)
            Log.v(TAG, "Nasze pobieranie zostało ukończone.");
    }
};

@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mReceiver);
    dMgr = null;
}
}
```

---

Zrozumienie tego kodu nie powinno nastręczać większych trudności. Najpierw inicjalizujemy główny widok i otrzymujemy odniesienie do pola tekstowego. Wewnątrz metody onResume() uzyskujemy odniesienie do usługi DOWNLOAD\_SERVICE. Zauważmy, że usuwamy pośredniość z tej usługi w metodzie onPause(). Metoda obsługująca kliknięcie onClick() generuje nowe żądanie DownloadManager.Request za pomocą ścieżki do pliku, który chcemy pobrać. Ponadto definiujemy tytuł, opis i rodzaj preferowanego typu sieci. Istnieje jeszcze kilka dodatkowych opcji do wyboru. Zostały one szczegółowo opisane w dokumentacji.

W celach demonstracyjnych jako typ połączenia wybraliśmy sieć komórkową, możemy jednak zdefiniować sieć Wi-Fi (za pomocą wartości NETWORK\_WIFI zamiast NETWORK\_MOBILE), możemy też połączyć obydwie wartości za pomocą operatora OR. Domyslnie obydwa rodzaje sieci mogą pobierać dane, co w przypadku naszej aplikacji oznacza, że będziemy korzystać z sieci komórkowej, nawet jeśli będzie dostępna sieć Wi-Fi.

Po zdefiniowaniu żądanego obiektu utworzyliśmy i zarejestrowaliśmy filtr dla odbiorcy komunikatów. Wkrótce przedstawimy kod tego odbiorcy. Dzięki jego zarejestrowaniu użytkownik zostanie poinformowany o zakończeniu pobierania każdego pliku. Oznacza to, że musimy śledzić identyfikator żądania, odsyłany po wywołaniu metody enqueue() wobec klasy DownloadManager. Na koniec aktualizujemy w interfejsie użytkownika informację o stanie, tym samym użytkownik będzie informowany o rozpoczętym pobieraniu.

Aby aplikacja zadziałała, musimy zdefiniować kilka uprawnień w pliku *AndroidManifest.xml*, co zostało ukazane na listingu 11.16. Wśród wymaganych uprawnień musimy zadeklarować dostęp do internetu oraz możliwość zapisywania plików na karcie SD. Co dziwne, jeżeli w wersji 2.3 Androida nie wprowadzimy tych uprawnień, pojawi się komunikat o błędzie, informujący o braku uprawnień ACCESS\_ALL\_DOWNLOADS, mimo że uprawnienie to nie jest naszej przykładowej aplikacji potrzebne.

**Listing 11.16.** Korzystanie z klasy DownloadManager — plik AndroidManifest.xml

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.services.download"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="9" />

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
</manifest>

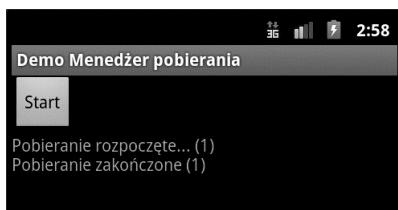
```

Po uruchomieniu aplikacji powinniśmy ujrzeć przycisk. Po jego kliknięciu Android rozpocznie pobieranie i wyświetli komunikat widoczny na rysunku 11.3. Zauważmy, że widoczna jest ikona pobierania w pasku powiadomień (lewy górny róg ekranu). Gdyby umieścić wskaźnik myszy na tej ikonie, zostałoby wyświetcone okno powiadomień, widoczne na rysunku 11.4.



**Rysunek 11.4.** Informacja o pobieraniu widoczna na liście powiadomień

Powiadomieniem w tym przypadku jest wyświetlenie paska postępu pobierania pliku. Po zakończeniu pobierania element ten zostanie wyczyszczony, a w oknie aplikacji pojawią się dodatkowe informacje, widoczne na rysunku 11.5.



**Rysunek 11.5.** Okno aplikacji informujące o zakończeniu pobierania

Teraz należy przeanalizować intencję w odbiorcy komunikatów. W ten sposób można się przekonać, czy zakończone pobieranie było częścią naszej aplikacji. Jeśli tak, trzeba jeszcze tylko zaktualizować komunikat o stanie w interfejsie użytkownika. Nie zapominajmy, że nie można zbyt mocno rozbudowywać odbiorcy komunikatów, gdyż musimy szybko wrócić z metody `onReceive()`. Możemy na przykład zamiast tego przywołać usługę przetwarzającą pobrany plik. W jej wnętrzu moglibyśmy wstawić fragment kodu zaprezentowany na listingu 11.17, umożliwiający przejrzenie zawartości pliku.

#### **Listing 11.17. Odczytywanie pobranego pliku**

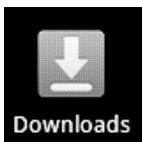
---

```
try {
    ParcelFileDescriptor pfd = dMgr.openDownloadedFile(doneDownloadId);
    // Posiadamy teraz uchwyty w trybie odczytu, przeznaczony do pobranego pliku
    // W dalszej części odczytujemy plik...
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

---

Jednym ze sposobów zlokalizowania pobranego pliku jest wykorzystanie usługi klasy `DownloadManager`. Klasa ta wymaga zdefiniowanego identyfikatora pobierania. Widzieliśmy to na listingu 11.17. Klasa `DownloadManager` zapewnia rozpoznanie pobranego pliku po identyfikatorze pobierania. Chociaż nasza aplikacja umieściła ten plik w publicznym obszarze karty SD, możemy w rzeczywistości wskazać, że zostanie umieszczony w rejonie prywatnych danych aplikacji za pomocą jednej z metod typu `setDestination*`() obiektu `DownloadManager.Request`.

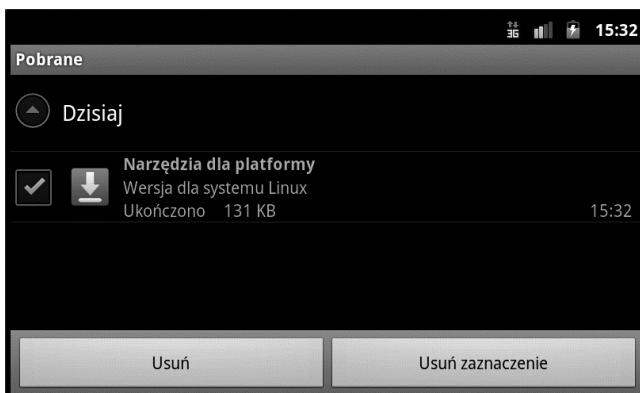
Klasa `DownloadManager` posiada własną aplikację, dzięki której można przejrzeć listę pobranych plików. Na poziomie menu aplikacji na emulatorze lub w urządzeniu poszukajmy ikony przedstawionej na rysunku 11.6.



**Rysunek 11.6.** Ikona aplikacji Downloads

Dzięki tej aplikacji możemy również uzyskać dostęp do tych plików. Spróbujmy już teraz. Po uruchomieniu programu `Downloads` ujrzymy okno zaprezentowane na rysunku 11.7. Menu widoczne w dolnej części ekranu pojawi się dopiero w momencie zaznaczenia pola wyboru znajdującego się przy nazwie pobranego pliku, co zrobiliśmy tuż przed wykonaniem zrzutu ekranu.

Klasa `DownloadManager` zawiera dostawcę treści przechowującego informacje o pobranym pliku. Aplikacja `Downloads` po prostu uzyskuje dostęp do tego dostawcy w celu wygenerowania listy plików dostępnych dla użytkownika. Oznacza to, że my również możemy przebadać tego dostawcę wewnętrz naszej aplikacji, aby zdobyć dane o pobranych plikach. W tym celu wykorzystujemy zapytanie `DownloadManager.Query` oraz metodę `query()`. Nie mamy tu jednak do dyspozycji zbyt wielu opcji wyszukiwania. Możemy wyszukiwać pliki za pomocą identyfikatorów pobierania (jednego bądź kilku) lub pod kątem stanu pobierania. Wynikiem metody `query()` jest obiekt `Cursor`, który może zostać wykorzystany do sprawdzenia wierszy w dostawcy treści klasy `DownloadManager`. Listę dostępnych kolumn znajdziemy w dokumentacji tej klasy, a niektóre



**Rysunek 11.7.** Aplikacja Downloads

z nich to lokalny identyfikator Uri pobranego pliku, rozmiar pliku w bajtach, rodzaj pliku, status pobierania oraz kilka innych parametrów. Gdy uzyskamy w ten sposób dostęp do dostawcy treści, musimy dodać w pliku *AndroidManifest.xml* uprawnienie `ACCESS_ALL_DOWNLOADS`.

Możemy również wykorzystać metodę `remove()` do anulowania procesu pobierania, chociaż znajdujący się na karcie fragment pliku nie zostanie automatycznie usunięty.

Pokazaliśmy, w jaki sposób można korzystać z usług opartych na protokole HTTP, a także jak zarządzać interfejsem tych usług za pomocą wyspecjalizowanej klasy `AsyncTask`. Typowo stojimy ją w sytuacji, w której jakaś operacja trwa przez określony, niezbyt długi czas oraz której wyniki bezpośrednio wpływają w jakiś sposób na interfejs użytkownika. Niekiedy jednak trzeba uruchomić jakiś proces przetwarzania danych trwający przez dłuższy czas albo przywołać jakąś niezwiązaną z interfejsem użytkownika funkcję, istniejącą w osobnej aplikacji. Do takich celów można wykorzystać usługi systemu Android. Zajmiemy się teraz ich omówieniem.

## Stosowanie usług w Androidzie

W Androidzie zdefiniowano pojęcie usług. Przez usługi rozumiemy składniki działające w tle, nieposiadające interfejsu użytkownika. Przypominają one usługi systemu Windows albo demony systemu Unix. Podobnie jak w przypadku wymienionych usług, usługi w Androidzie są zawsze dostępne, lecz nie muszą być bez przerwy aktywne. Co ważniejsze, cykle życia usług nie pokrywają się z cyklami życia aktywności. Podczas wstrzymywania, zatrzymywania lub zamykania aktywności możemy mieć jeszcze do czynienia z przetwarzaniem danych, które musi być kontynuowane. Do tego celu znakomicie nadają się właśnie usługi.

Istnieją dwa rodzaje usług w Androidzie: usługi lokalne i usługi zdalne. **Usługa lokalna** nie jest dostępna z poziomu innych aplikacji uruchomionych w urządzeniu. Zasadniczo usługa tego typu jest dostępna jedynie dla aplikacji, która ją wywołała, a dla innych programów uruchomionych w urządzeniu — już nie. W przypadku **usług zdalnych** dostęp do nich jest uzyskiwany z poziomu aplikacji, które je wywołyły, jak również z poziomu innych programów. Usługi zdalne są definiowane wobec klientów za pomocą języka AIDL (ang. *Android Interface Definition Language* — język definicji interfejsu systemu Android). Zapoznamy się z obydwoema rodzajami usług. W kilku następnych rozdziałach zagłębiimy się w zagadnienia dotyczące usług lokalnych, zatem wstępnie je tutaj opiszymy. Nie będziemy jednak wdawać się w szczegóły. W tym rozdziale dokładniej przedstawimy usługi zdalne.

## Usługi w Androidzie

Klasa Service stanowi osłonę dla kodu wykazującego zachowanie charakterystyczne dla usług. W przeciwieństwie do omawianej wcześniej klasy AsyncTask, obiekt klasy Service nie generuje automatycznie własnych wątków. Niezbędna jest ingerencja programisty, aby obiekt Service mógł korzystać z wątków. Oznacza to, że w przypadku braku funkcji wątkowania w usłudze jej kod będzie przetwarzany w głównym wątku. Jeśli usługa prowadzi czynności zajmujące niewiele czasu, nie powinno to stanowić problemu. Wątkowanie staje się niezbędne w przypadku operacji zajmujących więcej czasu. Pamiętajmy, że nie ma żadnych przeciwskań co do wykorzystania klasy AsyncTask do wątkowania wewnętrz usług.

Android wykorzystuje koncepcję usług z dwóch powodów:

- Po pierwsze, ma to na celu ułatwienie implementacji zadań przetwarzanych w tle.
- Po drugie, aby zapewnić komunikację międzyprocesową dla aplikacji uruchomionych na jednym urządzeniu.

Te dwa powody odpowiadają dwóm konkretnym rodzajom usług w Androidzie: usługom lokalnym i usługom zdalnym. W pierwszym przypadku jako przykład można wskazać usługę lokalną, zaimplementowaną jako część aplikacji pocztowej. Usługa ta zajmowałaby się wysyłaniem wiadomości na serwer pocztowy wraz z załącznikami i ponowieniami. Ponieważ proces ten może zabrać trochę czasu, usługa stanowi dobre rozwiązanie. Pozwala na osłonięcie funkcji, dzięki czemu można je przenieść z głównego wątku, a następnie wrócić do obsługi żądań użytkownika. W dodatku, nawet jeśli aktywność aplikacji pocztowej zostanie zakończona, wiadomości wciąż muszą zostać dostarczone. Jak się przekonamy, przykładem usługi wykorzystanej z drugiego powodu jest omówiona w dalszej części rozdziału aplikacja do tłumaczenia tekstu. Założymy, że na urządzeniu uruchomiono kilka aplikacji korzystających z jednego tekstu. W tym czasie, gdy aplikacje te pracują, potrzebujemy usługi, która przyjmuje tekst do przetłumaczenia z jednego języka na drugi. Zamiast umieszczać odpowiedni kod w każdej aplikacji, możemy napisać zdalną usługę tłumaczącą i sprawić, żeby aplikacje się z nią komunikowały.

Istnieją pewne istotne różnice pomiędzy usługami lokalnymi a zdalnymi. Ścisłej mówiąc, jeżeli usługa jest używana wyłącznie przez składniki w obrębie jednego procesu, klient musi ją uruchomić za pomocą wywołania metody `Context.startService()`. Jest to usługa lokalna, ponieważ jej celem jest, zasadniczo, uruchamianie zadań przetwarzanych w tle dla aplikacji, która uruchomiła tę usługę. Jeżeli usługa obsługuje metodę `onBind()`, mamy do czynienia z jej wersją zdalną, którą można wywołać za pomocą komunikacji międzyprocesowej (`Context.bindService()`). Usługi zdalne są również nazywane **usługami obsługującymi język AIDL**, ponieważ klienci porozumiewają się z nimi za pomocą języka AIDL.

Chociaż interfejs klasy `android.app.Service` obsługuje zarówno usługi lokalne, jak i zdalne, wdrożenie jednej implementacji usługi, która obsługiwałaby obydwa rodzaje usług, nie jest dobrym pomysłem. Wynika to z faktu, że każdy typ usługi posiada predefiniowany cykl życia; połączenie obydwu rodzajów, chociaż jest dopuszczalne, może powodować błędy.

Możemy teraz przeprowadzić szczegółową analizę obydwu kategorii usług. Zaczniemy od omówienia usług lokalnych, a następnie przejdziemy do usług zdalnych (usług obsługujących język AIDL). Jak już wspomnieliśmy, usługa lokalna jest wywoływana jedynie przez zarządzającąnią aplikację. Usługi zdalne obsługują mechanizm wywołania RPC (ang. *Remote Procedure Call* — zdalne wywołanie procedury). Usługa tego typu umożliwia zewnętrznym klientom, istniejącym na tym samym urządzeniu, podłączenie do niej i korzystanie z jej funkcji.

**Uwaga!**

Drugi rodzaj usługi nosi w Androidzie różne nazwy: usługa zdalna, usługa obsługująca język AIDL, usługa AIDL, usługa zewnętrzna oraz usługa RPC. Nazwy te dotyczą tego samego typu usługi — pozwalającej na uzyskanie do niej dostępu zdalnego przez aplikacje uruchomione na urządzeniu.

## Usługi lokalne

Usługi lokalne są usługami uruchamianymi za pomocą metody `Context.startService()`. Ten typ usług będzie działał od chwili uruchomienia do momentu wywołania przez klienta metody `Context.stopService()` wobec usługi lub do czasu, gdy usługa sama nie wywoła metody `stopSelf()`. Zwróćmy uwagę, że podczas wywołania metody `Context.startService()`, w przypadku gdy usługa nie została jeszcze utworzona, system ją utworzy i wywoła jej metodę `onStartCommand()`. Musimy pamiętać, że ponowne wywołanie metody `Context.startService()` nie spowoduje utworzenia nowej instancji usługi (jeżeli już istnieje), lecz przywoła ponownie metodę `onStartCommand()` już istniejącej usługi. Poniżej przedstawiamy przykłady usług lokalnych:

- Usługa monitorująca odczyty z czujników urządzenia oraz przeprowadzająca analizę danych, a po spełnieniu określonych warunków wyświetlającą alert. Usługa ta może działać nieprzerwanie.
- Usługa wykonująca zadania, umożliwiająca aktywnościom aplikacji zgłaszanie czynności do wykonania oraz kolejującą je. Usługa ta może działać wyłącznie przez okres zgłaszania tych czynności.

Na listingu 11.18 przedstawiono przykład usługi lokalnej, stanowiącej implementację usługi wykonującej zadania w tle. Efektem naszej pracy są cztery artefakty niezbędne do utworzenia oraz użytkowania usługi: plik `BackgroundService.java` (sama usługa), plik `MainActivity.java` (tworzy klasę aktywności wywołującej usługę), plik `main.xml` (układ graficzny aktywności) oraz plik `AndroidManifest.xml`. Na listingu 11.18 umieszczono wyłącznie kod z pliku `BackgroundService.java`. Najpierw przeanalizujemy tę klasę, a następnie zajmiemy się pozostałymi trzema plikami. Poniższa implementacja wymaga co najmniej wersji 2.0 Androida.

**Listing 11.18.** Implementowanie usługi lokalnej — plik `BackgroundService.java`

```
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class BackgroundService extends Service
{
    private static final String TAG = "BackgroundService";
    private NotificationManager notificationMgr;
    private ThreadGroup myThreads = new ThreadGroup("ServiceWorker");

    @Override
    public void onCreate() {
        super.onCreate();
```

```
Log.v(TAG, "w onCreate()");
notificationMgr =(NotificationManager) getSystemService(
    NOTIFICATION_SERVICE);
displayNotificationMessage("Usługa drugoplanowa została uruchomiona");
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    super.onStartCommand(intent, flags, startId);

    int counter = intent.getExtras().getInt("counter");
    Log.v(TAG, "w onStartCommand(), counter = " + licznik +
        ", startId = " + startId);

    new Thread(myThreads, new ServiceWorker(counter),
        "BackgroundService")
        .start();

    return START_STICKY;
}

class ServiceWorker implements Runnable
{
    private int counter = -1;
    public ServiceWorker(int counter) {
        this.counter = counter;
    }

    public void run() {
        final String TAG2 = "ServiceWorker:" +
            Thread.currentThread().getId();
        // w tym miejscu znajdują się operacje wykonywane w tle... reszta odpoczywa...
        try {
            Log.v(TAG2, "hibernacja przez 10 sekund. licznik = " +
                counter);
            Thread.sleep(10000);
            Log.v(TAG2, "...wybudzanie");
        } catch (InterruptedException e) {
            Log.v(TAG2, "...hibernacja przerwana");
        }
    }
}

@Override
public void onDestroy()
{
    Log.v(TAG, "w onDestroy(). Przerywanie wątków i anulowanie powiadomien.");
    myThreads.interrupt();
    notificationMgr.cancelAll();
    super.onDestroy();
}

@Override
public IBinder onBind(Intent intent) {
    Log.v(TAG, "w onBind()");
    return null;
}
```

```
}

private void displayNotificationMessage(String message)
{
    Notification notification =
        new Notification(R.drawable.emo_im_winking,
                         message, System.currentTimeMillis());

    notification.flags = Notification.FLAG_NO_CLEAR;

    PendingIntent contentIntent =
        PendingIntent.getActivity(this, 0,
                               new Intent(this, MainActivity.class), 0);

    notification.setLatestEventInfo(this, TAG, message,
                                   contentIntent);

    notificationMgr.notify(0, notification);
}
}
```

Struktura obiektu Service jest nieco podobna do architektury aktywności. Mamy do dyspozycji metodę `onCreate()` służącą do inicjalizacji, a także metodę `onDestroy()` odpowiedzialną za zakończenie działania usługi. W wersjach Androida starszych od 2.0 usługi posiadały metodę `onStart()`, która — począwszy od wersji 2.0 — została przemianowana na `onStartCommand()`. Różnica pomiędzy tymi metodami polega na wstawieniu parametru flagi w tym drugim przypadku, dzięki któremu usługa otrzymuje informacje o ponownym dostarczeniu intencji lub wskazanie, czy usługa powinna zostać ponownie uruchomiona. W naszym przykładzie wykorzystujemy metodę `onStartCommand()`. Usługi nie są wstrzymywane ani wznawiane w taki sposób jak aktywności, zatem nie ujrzymy w nich metod `onPause()` ani `onResume()`. Ponieważ mamy do czynienia z usługą lokalną, nie będziemy wprowadzali żadnego mechanizmu wiązania, jednak klasa Service wymaga implementacji metody  `onBind()`, więc wprowadzimy jedną metodę odsyłającą po prostu wartość `null`.

Wracając do metody `onCreate()` — nie musimy robić zbyt wiele, wystarczy powiadomić użytkownika o utworzeniu usługi. Dokonujemy tego za pomocą klasy `NotificationManager`. Prawdopodobnie Czytelnik zauważył już pasek powiadomień, znajdujący się w lewym górnym rogu ekranu Androida. Po jego rozcięgnięciu użytkownik ujrzy różnorodne komunikaty, a po kliknięciu powiadomienia może je uruchamiać, co zazwyczaj oznacza powrót do aktywności związanej z danym powiadomieniem. Ponieważ usługi działają (a przynajmniej istnieją) w tle, bez żadnej widocznej aktywności, musi istnieć jakiś sposób uzyskania z nimi kontaktu, chociażby po to, aby je wyłączyć. Tworzymy więc obiekt `Notification`, uzupełniamy go intencją oczekującą, dzięki której powróćmy do aktywności sterującej, i ją tam umieszczaemy. Wszystkie te czynności odbywają się w metodzie `displayNotificationMessage()`. Jeszcze jednym bardzo ważnym zadaniem jest ustawienie flagi na obiekcie `Notification`, dzięki czemu użytkownik nie będzie mógł usunąć go z listy. Istnienie tego obiektu w trakcie trwania usługi jest naprawdę niezbędne, zatem wprowadzamy atrybut `Notification.FLAG_NO_CLEAR`, aby przechowywać ten obiekt na liście, dopóki nie zostanie usunięty za pomocą metody `onDestroy()`. Dokładniej mówiąc, usuwanie powiadomień z listy obsługuje znajdująca się wewnątrz `onDestroy()` metoda `cancelAll()` użyta wobec klasy `NotificationManager`.

Musimy jeszcze zapewnić jedną rzecz, aby nasza przykładowa aplikacja działała. Musimy utworzyć obiekt klasy `Drawable`, nazwany `emo_im_winking`, i umieścić go w katalogu `drawable` naszego projektu. W celach demonstracyjnych dobrym źródłem obiektów klasy `Drawable` jest katalog zestawu Androida `SDK/platforms/<wersja>/data/res/drawable`, gdzie `<wersja>` oznacza aktualnie wykorzystywaną wersję systemu. Niestety, nie możemy komunikować się z systemowymi obiekttami klasy `Drawable` w taki sam sposób, jak ma to miejsce w przypadku układów graficznych, musimy więc skopiować wszystkie wymagane obiekty do katalogu `drawable`. Jeżeli chcemy korzystać z innego obiektu typu `Drawable`, wystarczy zmienić identyfikator zasobu w konstruktorze klasy `Notification`.

Po wysłaniu intencji do usługi za pomocą metody `startService()` w razie konieczności zostaje również wywołana metoda `onCreate()`, a także — w celu otrzymania intencji — metoda `onStartCommand()`. W naszym przypadku nie będziemy niczego szczególnego robić, może poza rozpakowaniem licznika oraz rozpoczęciem działania wątku drugoplanowego. W usłudze ustanowionej na potrzeby rzeczywistej aplikacji spodziewalibyśmy się przekazania jakichkolwiek danych za pomocą intencji, dajmy na to identyfikatorów URI. Warto zauważyć, że do utworzenia wątku używamy klasy `ThreadGroup`. Później, gdy będziemy chcieli pozbyć się drugoplanowych wątków, okaże się to przydatne. Zwrócić również uwagę na parametr `startId`. Zostaje on ustanowiony przez system i stanowi unikatowy identyfikator wywołań usługi od samego momentu jej utworzenia.

Klasa `ServiceWorker` jest typową klasą uruchamialną i to właśnie w niej jest wykonywana cała praca. W naszym przypadku zapisujemy pewne informacje w dzienniku oraz wstrzymujemy na chwilę działanie usługi. Na pewno nie manipulujemy interfejsem użytkownika; nie aktualizujemy na przykład widoków. Ponieważ nie działamy już w poziomie głównego wątku, nie możemy bezpośrednio wpływać na interfejs użytkownika. Klasa `ServiceWorker` posiada pewne mechanizmy umożliwiające wprowadzanie zmian w interfejsie użytkownika; wróćmy do ich omawiania w kilku najbliższych rozdziałach.

Ostatnim obiektem, na który warto zwrócić uwagę, omawiając klasę `BackgroundService`, jest metoda `onDestroy()`. To za jej pomocą kończymy działanie usługi. W naszym przykładzie pokażemy, jak pozbyć się uprzednio utworzonych wątków, jeżeli jeszcze jakieś będą istniały. Jeżeli tego nie zrobimy, mogą pozostać w tle i zajmować zasoby pamięci. Poza tym należy usunąć powiadomienie. Jeśli usługa zostanie zamknięta, użytkownik nie musi uruchamiać aktywności, aby samodzielnie wyłączyć tę usługę. W standardowej aplikacji możemy jednak zechcieć, aby niektóre wątki normalnie pracowały dalej. Jeżeli nasza usługa wysyła wiadomości e-mail, zwykłe zamykanie wątków na pewno nam nie pomoże. Być może pokazujemy nieco zbyt uproszczony przykład, ponieważ Czytelnik może się zasugerować, że za pomocą metody `interrupt()` można zamykać drugoplanowe wątki. W rzeczywistości jednak ta metoda pozwala najwyżej na ich przerywanie, a to nie jest jednoznaczne z zamknięciem wątku. Istnieją pewne przestarzałe metody, służące do zamykania wątków, nie powinniśmy ich jednak stosować. Mogą one powodować problemy ze stabilnością i pamięcią. W omawianym przykładowym kodzie sprawdzają się przerwania, ponieważ korzystamy z hibernacji, którą także można przerywać.

Warto poświęcić chwilę na zapoznanie się z klasą `ThreadGroup`, ponieważ zawiera ona mechanizmy umożliwiające uzyskanie dostępu do wątków. Utworzyliśmy w naszej usłudze jeden obiekt `ThreadGroup`, który następnie był używany podczas generowania poszczególnych wątków. Wewnątrz metody `onDestroy()` stosujemy po prostu metodę `interrupt()` wobec obiektu `ThreadGroup` i tym samym przerywamy każdy wątek stanowiący część tego obiektu.

Znamy więc już podstawy tworzenia prostej usługi lokalnej. Zanim ukażemy kod aktywności, zapoznajmy się najpierw z zawartością pliku układu graficznego, zaprezentowaną na listingu 11.19.

#### **Listing. 11.19. Implementacja usługi lokalnej — plik main.xml**

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<Button android:id="@+id/startBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Rozpocznij usługę" android:onClick="doClick" />
<Button android:id="@+id/stopBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Zakończ usługę" android:onClick="doClick" />
</LinearLayout>
```

---

Interfejs użytkownika składa się z dwóch przycisków. Wciśnięcie jednego powoduje wywołanie metody `startService()`, a drugiego — metody `stopService()`. Moglibyśmy zamiast nich wprowadzić kontrolkę `ToggleButton`, wtedy jednak nie moglibyśmy wywołać metody `startService()` kilka razy pod rząd. Jest to ważne spostrzeżenie. Nie istnieje bezpośrednia relacja pomiędzy metodami `startService()` a `stopService()`. Po wywołaniu metody `stopService()` usługa zostanie zakończona, tak samo jak wszystkie wątki utworzone za pomocą wszystkich wystąpień metody `startService()`. Nasza przykładowa aplikacja wymaga wartości 5 parametru `minSdkVersion`, ponieważ stosujemy nowszą metodę `onStartCommand()` w miejscu starszej `onStart()`. Zatem możemy również skorzystać z atrybutu `android:onClick` znacznika `Button` w naszym pliku układu graficznego. Przyjrzyjmy się teraz kodowi aktywności, zaprezentowanemu na listingu 11.20.

#### **Listing 11.20. Implementacja usługi lokalnej — plik MainActivity.java**

---

```
// MainActivity.java
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.View;

public class MainActivity extends Activity
{
    private static final String TAG = "MainActivity";
    private int counter = 1;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
```

---

```
        setContentView(R.layout.main);  
    }  
  
    public void doClick(View view) {  
        switch(view.getId()) {  
        case R.id.startBtn:  
            Log.v(TAG, "Uruchamianie usługi... licznik = " + counter);  
            Intent intent = new Intent(MainActivity.this,  
                BackgroundService.class);  
            intent.putExtra("counter", counter++);  
            startService(intent);  
            break;  
        case R.id.stopBtn:  
            stopService();  
        }  
    }  
  
    private void stopService() {  
        Log.v(TAG, "Zatrzymywanie usługi...");  
        if(stopService(new Intent(MainActivity.this,  
            BackgroundService.class)))  
            Log.v(TAG, "metoda stopService zakończona sukcesem");  
        else  
            Log.v(TAG, "metoda stopService zakończona niepowodzeniem");  
    }  
  
    @Override  
    public void onDestroy()  
{  
    stopService();  
    super.onDestroy();  
}  
}
```

---

Nasza aktywność `MainActivity` nie różni się zbytnio od innych aktywności, jakie wcześniej tworzyliśmy. Istnieje prosta metoda `onCreate()`, służąca do konfigurowania interfejsu użytkownika za pomocą pliku układu graficznego `main.xml`. Mamy również do dyspozycji metodę `doClick()`, obsługującą wywołanie zwrotne przycisku. W naszym przykładzie wywołujemy metodę `startService()` po wcisnięciu przycisku *Rozpocznij usługę* oraz metodę `stopService()` po wcisnięciu przycisku *Zakończ usługę*. W momencie uruchomienia usługi chcemy przekazać jej pewne dane, czego dokonujemy za pomocą intencji. Wybraliśmy przekazanie tych danych w pakiecie Extras, ale gdybyśmy posiadali identyfikator URI, moglibyśmy tego dokonać za pomocą metody `setData()`. W czasie zatrzymywania usługi sprawdzamy otrzymane wyniki. W normalnych warunkach powinniśmy otrzymać wartość `true`, jeśli jednak usługa nie była uruchomiona, może zostać przekazana wartość `false`. Na koniec, jeśli zamkniemy aktywność, chcemy również zatrzymać usługę, zatem dokonujemy tego w metodzie `onDestroy()`. Pozostał nam jeszcze jeden plik do omówienia — `AndroidManifest.xml`, który widzimy na listingu 11.21.

---

**Listing 11.21.** Implementacja usługi lokalnej — plik `AndroidManifest.xml`

---

```
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.androidbook.services.simplelocal"  
    android:versionCode="1"
```

```

        android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".MainActivity"
            android:label="@string/app_name"
            android:launchMode="singleTop" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name="BackgroundService"/>
    </application>
<uses-sdk android:minSdkVersion="5" />

</manifest>
```

---

Poza standardowymi znacznikami `<activity>` widzimy także znacznik `<service>`. Ponieważ mamy do czynienia z usługą lokalną, wywoływaną jawnie za pomocą nazwy klasy, nie musimy umieszczać zbyt wielu informacji w tym węźle. Wymagana jest jedynie nazwa usługi. Jest jednak jeszcze jedna istotna kwestia dotycząca tego pliku manifestu. Nasza usługa tworzy powiadomienie, dzięki któremu użytkownik może powrócić do aktywności `MainActivity`, na przykład w przypadku wcisnięcia przycisku startowego bez uprzedniego zatrzymania usługi.

Aktywność `MainActivity` jest wciąż obecna, lecz niewidoczna. Jednym ze sposobów jej przywrócenia jest kliknięcie powiadomienia utworzonego przez naszą usługę. Na pewno nie chcemy, aby obok już istniejącego, niewidocznego wystąpienia aktywności `MainActivity` zostało utworzone nowe jej wystąpienie. Aby tak się nie stało, wprowadzamy w pliku *AndroidManifest.xml* atrybut `android:launchMode` i nadajemy mu wartość `singleTop`. W ten sposób zapewniamy wysunięcie istniejącej, niewidzialnej aktywności `MainActivity` na pierwszy plan i wyświetlenie jej na ekranie zamiast tworzenia jej nowego wystąpienia.

Po uruchomieniu aplikacji ujrzymy dwa przyciski. Po kliknięciu przycisku *Rozpocznij usługę* zostanie utworzony obiekt usługi oraz wywołana metoda `onStartCommand()`. Nasz kod generuje kilka informacji w oknie *LogCat*, zatem możemy śledzić jego pracę. Teraz można kilkakrotnie kliknąć przycisk *Rozpocznij usługę* — można próbować kliknąć szybko. Zobaczmy tworzone wątki, obsługujące oddzielnie każde żądanie. Zauważymy także, że wartość licznika jest przekazywana poprzez wszystkie wątki obiektu `ServiceWorker`. Po wcisnięciu przycisku *Zatrzymaj usługę* nasza usługa zostanie zakończona i w oknie *LogCat* ujrzymy komunikaty pochodzące z metod `stopService()` aktywności `MainActivity`, `onDestroy()` usługi `BackgroundService` oraz prawdopodobnie z wątków `ServiceWorker` (jeśli zostały przerwane).

Powinniśmy również ujrzeć powiadomienie po uruchomieniu usługi. Gdy już będzie działała, wcisnijmy przycisk cofania. Zauważymy, że po zamknięciu aktywności zniknie powiadomienie. Oznacza to, że zakończyliśmy również usługę. Aby ponownie uruchomić aktywność `MainActivity`, wystarczy kliknąć przycisk *Uruchom usługę*, co spowoduje również włączenie usługi. Wcisnijmy teraz przycisk ekranu startowego. Aktywność zniknie z ekranu, ale powiadomienie pozostaje, co oznacza, że usługa wciąż istnieje. Kliknijmy je, a aktywność zostanie ponownie wyświetlona na ekranie.

Zwróćmy uwagę, że w tym przykładzie wykorzystujemy aktywność do komunikowania się z usługą, ale każdy składnik aplikacji może również korzystać z tej usługi. Dotyczy to innych usług, aktywności, ogólnych klas itd. Odnotujmy także fakt, że nasza usługa nie zatrzyma się

samoistne; jest pod tym względem zależna od aktywności. Istnieją jednak pewne metody pozwalające usłudze na samoistne zatrzymanie, chociażby takie jak `stopSelf()` czy `stopSelfResult()`.

`BackgroundService` jest typowym przykładem usługi wykorzystywanej przez składniki aplikacji przechowującą tę usługę. Inaczej mówiąc, aplikacja przechowująca tę usługę jest jednocześnie jej jedynym konsumentem. Ponieważ usługa nie obsługuje klientów znajdujących się poza jej procesem, jest ona lokalna. Z tego wynika, że w przeciwieństwie do usługi zdalnej, odsyła wartość `null` w metodzie `bind()`. Zatem jedynym sposobem powiązania obiektu z tą usługą jest wywołanie metody `Context.startService()`. Najważniejszymi metodami usługi lokalnej są `onCreate()`, `onStartCommand()`, `stop*`() oraz `onDestroy()`.

W przypadku usługi lokalnej mamy do dyspozycji jeszcze jedną opcję, wykorzystywaną w przypadku uruchomienia tylko jednego jej wystąpienia wraz z tylko jednym wątkiem. W takim przypadku w metodzie `onCreate()` tej usługi możemy utworzyć wątek przetwarzający wszystkie operacje usługi. Możemy go umieścić w metodzie `onCreate()` zamiast w `onStartCommand()`. Jest to dopuszczalne, ponieważ metoda `onCreate()` jest wywoływana tylko raz, a my potrzebujemy tylko jednego wątku istniejącego przez czas trwania usługi. Nie otrzymalibyśmy jednak w metodzie `onCreate()` treści intencji przekazywanej przez metodę `startService()`. Gdybyśmy jej potrzebowali, wystarczy skorzystać z opisanego wcześniej algorytmu i wywołać metodę `onStartCommand()` tylko raz.

Na tym zakończymy część dotyczącą usług lokalnych. Zostaną one dokładniej opisane w następnych rozdziałach. Zbadajmy teraz usługi AIDL — posiadające bardziej złożoną strukturę.

## Usługi AIDL

W poprzednim podrozdziale pokazaliśmy, jak należy pisać usługę użytkowaną przez aplikację, która uruchomiła tę usługę. Zademonstrujemy teraz technikę tworzenia usługi wykorzystywanej przez inne procesy poprzez wywołanie RPC. Podobnie jak w przypadku innych rozwiązań opartych na wywołaniach RPC, tak i teraz musimy korzystać w Androidzie z języka IDL (ang. *Interface Definition Language* — język definiowania interfejsu) do definiowania interfejsu dostępnego dla klientów. W świecie Androida język ten nosi nazwę AIDL. Aby utworzyć usługę zdальną, należy wykonać poniższe działania:

1. Napisz plik AIDL, który definiuje interfejs dla klientów. Plik AIDL wykorzystuje składnię języka Java i posiada rozszerzenie `.aidl`. Nazwa pakietu powinna być taka sama jak nazwa pakietu w projekcie Androida.
2. Dodaj plik AIDL do projektu w środowisku Eclipse do katalogu `src`. Wtyczka ADT wywoła kompilator języka AIDL, dzięki któremu z pliku AIDL zostanie wygenerowany interfejs Java (kompilator AIDL jest wywoływany podczas procesu budowania aplikacji).
3. Zaimplementuj usługę i przekaż interfejs z metodą `onBind()`.
4. Dodaj konfigurację usługi do pliku `AndroidManifest.xml`.

W kolejnych podrozdziałach omówiliśmy poszczególne czynności.

## Definiowanie interfejsu usługi w języku AIDL

W celu zademonstrowania usługi zdalnej napiszemy usługę przedstawiającą notowania giełdowe. Będzie ona zawierała metodę przyjmującą symbol notowanej firmy i odsyającą wartość notowań tej firmy. Pierwszym krokiem tworzenia usługi zdalnej w Androidzie jest zaprojektowanie

definicji interfejsu tej usługi w pliku AIDL. Na listingu 11.22 została ukazana definicja usługi IStockQuoteService w języku AIDL. Plik ten należy umieścić wraz ze wszystkimi innymi plikami Java związanymi z projektem StockQuoteService.

#### **Listing 11.22.** Definicja usługi notowań giełdowych w języku AIDL

---

```
// Jest to plik IStockQuoteService.aidl
package com.androidbook.services.stockquoteservice;
interface IStockQuoteService
{
    double getQuote(String ticker);
}
```

---

Usługa IStockQuoteService przyjmuje symbol notowanej firmy i przekazuje bieżącą wartość (typu double) notowań tej firmy. Podczas tworzenia pliku AIDL wtyczka ADT środowiska Eclipse uruchamia kompilator języka AIDL przetwarzający ten plik (podczas procesu komplikacji kodu całej aplikacji). W wyniku bezbłędnej komplikacji pliku AIDL zostanie utworzony interfejs Java zdolny do komunikacji typu RPC. Zwróćmy uwagę, że wygenerowany plik zostanie umieszczony w pakiecie noszącym nazwę pliku AIDL — w naszym przypadku będzie to *com.androidbook.services.stockquoteservice*.

Na listingu 11.23 przedstawiamy wygenerowany plik Java naszego interfejsu IStockQuoteService. Wygenerowany plik zostanie umieszczony w folderze *gen* naszego projektu.

#### **Listing 11.23.** Wygenerowany przez kompilator plik Java

---

```
/*
 * Ten plik został automatycznie wygenerowany. NIE NALEŻY GO MODYFIKOWAĆ.
 * Oryginalny plik: C:\android\StockQuoteService\src\com\androidbook\
services\stockquoteservice\IStockQuoteService.aidl
 */

package com.androidbook.services.stockquoteservice;
import java.lang.String;
import android.os.RemoteException;
import android.os.IBinder;
import android.os.IInterface;
import android.os.Binder;
import android.os.Parcel;
public interface IStockQuoteService extends android.os.IInterface
{
    /** Implementacja IPC lokalnej klasy pośredniczącej. */
    public static abstract class Stub extends android.os.Binder implements
com.androidbook.services.stockquoteservice.IStockQuoteService
    {
        private static final java.lang.String DESCRIPTOR =
"com.androidbook.services.stockquoteservice.IStockQuoteService";
        /** Tworzy procedurę pośredniczącą, przyłączaną do interfejsu. */
        public Stub()
        {
            this.attachInterface(this, DESCRIPTOR);
        }
    }
}
```

---

```
* Umieszcza obiekt IBinder wewnątrz interfejsu IStockQuoteService,  
* w razie potrzeby tworzy pośrednika.  
*/  
public static com.androidbook.services.stockquoteservice.IStockQuoteService  
asInterface(android.os.IBinder obj)  
{  
    if ((obj==null)) {  
        return null;  
    }  
    android.os.IInterface iin = (android.os.IInterface)obj.queryLocalInterface(DESCRIPTOR);  
    if (((iin!=null)&&(iin instanceof  
com.androidbook..services.stockquoteservice.IStockQuoteService))) {  
        return ((com.androidbook.services.stockquoteservice.IStockQuoteService)iin);  
    }  
    return ((com.androidbook.services.stockquoteservice.IStockQuoteService)iin);  
}  
return new com.androidbook.services.stockquoteservice.  
→IStockQuoteService.Stub.Proxy(obj);  
}  
public android.os.IBinder asBinder()  
{  
    return this;  
}  
@Override public boolean onTransact(int code, android.os.Parcel data,  
    android.os.Parcel reply, int flags) throws android.os.RemoteException  
{  
    switch (code)  
    {  
        case INTERFACE_TRANSACTION:  
        {  
            reply.writeString(DESCRIPTOR);  
            return true;  
        }  
        case TRANSACTION_getQuote:  
        {  
            data.enforceInterface(DESCRIPTOR);  
            java.lang.String _arg0;  
            _arg0 = data.readString();  
            double _result = this.getQuote(_arg0);  
            reply.writeNoException();  
            reply.writeDouble(_result);  
            return true;  
        }  
    }  
    return super.onTransact(code, data, reply, flags);  
}  
private static class Proxy implements  
com.androidbook.services.stockquoteservice.IStockQuoteService  
{  
    private android.os.IBinder mRemote;  
    Proxy(android.os.IBinder remote)  
    {  
        mRemote = remote;  
    }  
    public android.os.IBinder asBinder()  
    {  
        return mRemote;
```

```

}
public java.lang.String getInterfaceDescriptor()
{
    return DESCRIPTOR;
}
public double getQuote(java.lang.String ticker) throws android.os.RemoteException
{
    android.os.Parcel _data = android.os.Parcel.obtain();
    android.os.Parcel _reply = android.os.Parcel.obtain();
    double _result;
    try {
        _data.writeInterfaceToken(DESCRIPTOR);
        _data.writeString(ticker);
        mRemote.transact(Stub.TRANSACTION_getQuote, _data, _reply, 0);
        _reply.readException();
        _result = _reply.readDouble();
    }
    finally {
        _reply.recycle();
        _data.recycle();
    }
    return _result;
}
static final int TRANSACTION_getQuote = (IBinder.FIRST_CALL_TRANSACTION + 0);
}
public double getQuote(java.lang.String ticker) throws android.os.RemoteException;
}

```

Poniżej przedstawiamy kilka istotnych wniosków dotyczących wygenerowanych klas:

- Interfejs zdefiniowany w pliku AIDL został zaimplementowany w wygenerowanym kodzie (to znaczy, że istnieje interfejs `IStockQuoteService`).
- Abstrakcyjna klasa `static final` nosząca nazwę `stub` rozszerza klasę `android.os.Binder` i implementuje interfejs `IStockQuoteService`. Zwróćmy uwagę, że jest to abstrakcyjna klasa.
- Wewnętrzna klasa `Proxy` implementuje interfejs `IStockQuoteService`, który jest pośrednikiem dla klasy `Stub`.
- Plik AIDL musi się znaleźć w tym samym pakiecie, w którym będą umieszczone wygenerowane pliki (co zostało określone w deklaracji pakietu pliku AIDL).

Zaimplementujmy teraz interfejs AIDL w klasie usługi.

## Implementowanie interfejsu AIDL

W poprzednim podrozdziale zdefiniowaliśmy plik AIDL dla usługi prezentującej notowania giełdowe i wygenerowaliśmy wiążący plik. Teraz wprowadzimy implementację tej usługi. Żeby zaimplementować jej interfejs, musimy napisać klasę rozszerzającą klasę `android.app.Service` i rozszerzającą interfejs `IStockQuoteService`. Utworzoną przez nas klasę nazwiemy `StockQuoteService`. Aby można było wyeksponować usługę klientom, w usłudze `StockQuoteService` zaimplementujemy metodę `onBind()`. Dodamy też pewne informacje do pliku `AndroidManifest.xml`. Listing 11.24 przedstawia implementację interfejsu `IStockQuoteService`. Również ten plik umieszczamy w folderze `src` projektu `StockQuoteService`.

**Listing 11.24.** Implementacja usługi IStockQuoteService

```
// StockQuoteService.java
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;

public class StockQuoteService extends Service
{
    private static final String TAG = "StockQuoteService";
    public class StockQuoteServiceImpl extends IStockQuoteService.Stub
    {
        @Override
        public double getQuote(String ticker) throws RemoteException
        {
            Log.v(TAG, "Metoda getQuote() wywołana dla " + ticker);
            return 20.0;
        }
    }

    @Override
    public void onCreate()
    {
        super.onCreate();
        Log.v(TAG, "Wywołana metoda onCreate()");
    }

    @Override
    public void onDestroy()
    {
        super.onDestroy();
        Log.v(TAG, "Wywołana metoda onDestroy()");
    }

    @Override
    public IBinder onBind(Intent intent)
    {
        Log.v(TAG, "Wywołana metoda onBind()");
        return new StockQuoteServiceImpl();
    }
}
```

---

Klasa StockQuoteService.java przytoczona na listingu 11.24 przypomina omówioną wcześniej usługę lokalną BackgroundService, pozbawiona jest jednak klasy NotificationManager. Zasadnicza różnica polega na zaimplementowaniu w tym przypadku metody onBind(). Przypominamy, że w pliku AIDL wygenerowaliśmy abstrakcyjną klasę Stub, która implementowała interfejs IStockQuoteService. W naszej implementacji usługi zawieramy wewnętrzną klasę StockQuoteServiceImpl, która rozszerza klasę Stub. Klasa ta służy nam jako implementacja usługi zdalnej, a instancja tej klasy jest przekazywana z metody onBind(). W ten sposób otrzymujemy działającą usługę AIDL, chociaż zewnętrzne klienci nie mogą się z nią jeszcze połączyć.

Aby wyeksponować usługę klientom, musimy dodać deklarację usługi w pliku *AndroidManifest.xml*, tym razem jednak potrzebujemy filtru intencji do jej odsłonięcia. Na listingu 11.25 została pokazana deklaracja usługi StockQuoteService. Znacznik <service> jest podzielony wobec znacznika <application>.

**Listing 11.25.** Deklaracja usługi IStockQuoteService w pliku manifeście

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.services.stockquoteservice"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <service android:name="StockQuoteService">
            <intent-filter>
                <action
                    android:name="com.androidbook.services.stockquoteservice.IStockQuoteService" />
            </intent-filter>
        </service>
    </application>
    <uses-sdk android:minSdkVersion="4" />
</manifest>
```

Podobnie jak to ma miejsce w przypadku wszystkich usług, poprzez znacznik `<service>` definiujemy usługę, którą chcemy wyeksponować. W przypadku usługi AIDL musimy dodać również węzeł `<intent-filter>` z wpisem `<action>` dla interfejsu usługi, którą chcemy wyeksponować.

Po umieszczeniu tego wpisu będziemy posiadać już wszystkie elementy wymagane do wdrożenia usługi. Gdy będziemy już gotowi na wdrożenie usługi z poziomu środowiska Eclipse, wybierzmy po prostu opcję *Run As*, podobnie jak ma to miejsce w przypadku zwykłej aplikacji. Ujrzymy w konsoli komunikat informujący nas, że aplikacja nie posiada programu wywołującego, ale i tak zostanie wdrożona, co jest zgodne z naszymi oczekiwaniami. Przyjrzyjmy się teraz, w jaki sposób można wywołać usługę z poziomu innej aplikacji (zainstalowanej, oczywiście, na tym samym urządzeniu).

## Wywoływanie usługi z poziomu aplikacji klienckiej

Komunikacja pomiędzy klientem a usługą jest możliwa pod warunkiem ustanowienia protokołu lub kontraktu pomiędzy tymi dwoma obiektami. W Androidzie kontrakt jest umieszczony w pliku AIDL. Zatem pierwszą czynnością prowadzącą do skorzystania z usługi jest skopiowanie jej pliku AIDL i wklejenie go do projektu klienckiego. Podczas kopiowania tego pliku kompilator tworzy identyczny plik definicji interfejsu jak w przypadku implementacji usługi (w projekcie implementacji usługi). W ten sposób zostają wyeksponowane klientowi wszystkie metody, parametry oraz przekazywane typy usługi. Stwórzmy nowy projekt i skopijmy do niego plik AIDL.

1. Utwórz nowy projekt Androida o nazwie *StockQuoteClient*. Dla pakietu użyj innej nazwy, na przykład *com.androidbook.stockquoteclient*. Wpisz klasę *MainActivity* w polu *Create Activity*.
2. Utwórz nowy pakiet Java w katalogu *src* tego projektu i nazwij go *com.androidbook.services.stockquoteservice*.
3. Skopuj do tego pakietu plik *IStockQuoteService.aidl* z projektu *StockQuoteService*. Zwróć uwagę, że po skopiowaniu tego pliku do projektu kompilator języka AIDL wygeneruje powiązany z nim plik Java.

Odtwarzany interfejs usługi posłuży nam jako kontrakt pomiędzy klientem a usługą. Kolejnym krokiem jest uzyskanie odniesienia do usługi w celu wywołania metody `getQuote()`. W przypadku usług zdalnych musimy zamiast metody `startService()` wywołać metodę `bindService()`. Na listingu 11.26 umieściliśmy klasę aktywności zachowującą się jak klient w stosunku do usługi `IStockQuoteService`. Z kolei na listingu 11.27 umieszczono plik układu graficznego tej aktywności.

Na listingu 11.26 przedstawiono plik `MainActivity.java`. Nazwa pakietu aktywności klienckiej nie jest tak istotna — możemy umieścić aktywność w dowolnym pakiecie. Jednak należy pamiętać, że tworzone przez nas artefakty języka AIDL rozpoznają nazwę pakietu, ponieważ kompilator generuje kod z zawartością pliku AIDL.

---

**Listing 11.26.** Klient usługi `IStockQuoteService`

```
// Jest to plik MainActivity.java
import com.androidbook.services.stockquoteservice.IStockQuoteService;
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;
import android.widget.ToggleButton;

public class MainActivity extends Activity {
    private static final String TAG = "StockQuoteClient";
    private IStockQuoteService stockService = null;
    private ToggleButton bindBtn;
    private Button callBtn;

    /** Wywoływanie podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        bindBtn = (ToggleButton)findViewById(R.id.bindBtn);
        callBtn = (Button)findViewById(R.id.callBtn);
    }

    public void doClick(View view) {
        switch(view.getId()) {
            case R.id.bindBtn:
                if(((ToggleButton) view).isChecked()) {
                    bindService(new Intent(
                        IStockQuoteService.class.getName(),
                        serConn, Context.BIND_AUTO_CREATE));
                }
                else {

```

```

        unbindService(servConn);
        callBtn.setEnabled(false);
    }
    break;
case R.id.callBtn:
    callService();
    break;
}
}

private void callService() {
    try {
        double val = stockService.getQuote("ANDROID");
        Toast.makeText(MainActivity.this,
                    "Wartość z usługi wynosi " + val,
                    Toast.LENGTH_SHORT).show();
    } catch (RemoteException ee) {
        Log.e("MainActivity", ee.getMessage(), ee);
    }
}

private ServiceConnection servConn = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName name,
        IBinder service)
    {
        Log.v(TAG, "wywołana metoda onServiceConnected()");
        stockService = IStockQuoteService.Stub.asInterface(service);
        bindBtn.setChecked(true);
        callBtn.setEnabled(true);
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
        Log.v(TAG, "wywołana metoda onServiceDisconnected()");
        bindBtn.setChecked(false);
        callBtn.setEnabled(false);
        stockService = null;
    }
};

protected void onDestroy() {
    Log.v(TAG, "wywołana metoda onDestroy()");
    if(callBtn.isEnabled())
        unbindService(servConn);
    super.onDestroy();
}
}

```

Aktywność wyświetla nasz układ graficzny i pobiera odniesienie do przycisku *Wywołaj usługę*, dzięki czemu możemy go poprawnie uruchomić w trakcie działania usługi oraz wyłączyć, gdy usługa jest zatrzymana. Po kliknięciu przycisku *Podłącz* zostanie wywołana metoda `bindService()`. W analogiczny sposób po wybraniu przycisku *Odlacz* zostanie wywołana

metoda `unbindService()`. Zwróćmy uwagę, że metodzie `bindService()` są przekazywane trzy parametry: nazwa usługi AIDL, instancja obiektu `ServiceConnection` oraz flaga automatycznego utworzenia usługi.

#### **Listing 11.27. Układ graficzny klienta usługi `IStockQuoteService`**

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

<ToggleButton android:id="@+id/bindBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOff="Podłącz" android:textOn="Odłącz"
    android:onClick="doClick" />

<Button android:id="@+id/callBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Wywołaj usługę" android:enabled="false"
    android:onClick="doClick" />
</LinearLayout>
```

---

W przypadku usługi AIDL musimy wprowadzić implementację interfejsu `ServiceConnection`. Interfejs ten definiuje dwie metody: jedna jest wywoływana przez system po ustanowieniu połączenia z usługą, a druga jest wywoływana po przerwaniu takiego połączenia. W naszej implementacji aktywności definiujemy prywatnego, anonimowego członka, implementującego interfejs `ServiceConnection` dla usługi `IStockQuoteService`. Podczas wywoływanego metody `bindService()` przekazujemy jej odniesienie do tego członka. Po ustanowieniu połączenia z usługą zostaje przywołana metoda zwrotna `onServiceConnected()`, a następnie uzyskujemy odniesienie do obiektu `IStockQuoteService` za pomocą klasy `Stub`, po czym aktywujemy przycisk *Wywołaj usługę*.

Zauważmy, że wywołanie `bindService()` jest asynchroniczne. Ta asynchroniczność wynika z faktu, że proces lub usługa mogą nie być uruchomione, zatem należy je utworzyć lub uruchomić. Główny wątek nie może oczekiwać na uruchomienie usługi. Wobec tego platforma posiada wywołanie zwrotne `ServiceConnection`, dzięki któremu wiadomo, kiedy usługa jest uruchomiona lub kiedy przestaje być dostępna.

Zwróćmy uwagę na metodę zwrotną `onServiceDisconnected()`. *Nie* zostaje ona przywołana podczas odłączania się od usługi. Zostaje ona wywołana jedynie w przypadku, gdy usługa przestanie działać. Jeżeli tak się stanie, nie musimy się zastanawiać nad tym, czy połączenie z tą usługą jest wciąż aktywne, możemy za to chcieć ponownie wywołać metodę `onBind()`. Z tego właśnie powodu zmieniamy status przycisków po wywołaniu tej metody. Zauważmy jednak, że napisaliśmy „możemy chcieć ponownie wywołać metodę `onBind()`”. Android może samoczynnie uruchomić usługę i wywołać metodę `onServiceConnected()`.

Możemy sami to sprawdzić, uruchamiając klienta, podłączając się do usługi, a potem — za pomocą narzędzi DDMS — zatrzymując usługę.

Po uruchomieniu przykładowej aplikacji obserwujmy komunikaty dziennika w oknie *LogCat*, aby prześledzić tok działania aplikacji.

Teraz wiemy, w jaki sposób tworzyć i użytkować usługi AIDL. Zanim wprowadzimy dodatkowe, komplikujące temat szczegóły, powtórzmy, co należy zrobić, aby utworzyć prostą usługę lokalną oraz usługę AIDL. Usługa lokalna nie obsługuje metody `onBind()` — przekazywaną wartością jest `null`. Ten rodzaj usługi jest dostępny wyłącznie dla składników aplikacji, która wywołała tę usługę. Usługi lokalne są wywoływane za pomocą metody `startService()`.

Z drugiej strony istnieje usługa AIDL, która może być użytkowana zarówno przez składniki będące częścią tego samego obiektu, jak i składniki innych aplikacji. W przypadku tego typu usługi należy zdefiniować kontrakt pomiędzy tą usługą a klientem za pomocą pliku AIDL. Usługa implementuje kontrakt AIDL, a klient łączy się z definicją języka AIDL. Usługa dokonuje implementacji kontraktu poprzez przekazanie implementacji interfejsu AIDL z metody `onBind()`. Klienci łączą się z usługą AIDL poprzez wywołanie metody `bindService()`, a rozłączają się z nią dzięki wywołaniu metody `unbindService()`.

W naszych dotychczasowych przykładach usług przekazywaliśmy wyłącznie proste typy danych języka Java. Usługi w Androidzie w rzeczywistości obsługują również przekazywanie złożonych typów danych. Jest to bardzo przydatne, zwłaszcza w przypadku usług AIDL, ponieważ możemy określić dowolną liczbę parametrów, które należy przekazać usłudze, a przekazywanie ich wszystkich w formie prostych typów danych nie jest rozsądne. Bardziej sensowne jest upakowanie ich w formie złożonych typów danych i przekazanie ich usłudze w tej postaci.

Zobaczmy, w jaki sposób można przekazywać usługom złożone typy danych.

## Przekazywanie usługom złożonych typów danych

Przekazywanie do usług i z usług złożonych typów danych wymaga więcej pracy niż przekazywanie prostych typów danych. Zanim zagłębiimy się w ten temat, powinniśmy przedstawić podstawowe koncepcje obsługi złożonych typów danych w języku AIDL:

- Język AIDL obsługuje typy `String` i `CharSequence`.
- Istnieje możliwość przekazywania innych interfejsów AIDL, jednak dla każdego interfejsu, do którego tworzono jest odniesienie, wymagana jest instrukcja `import` (nawet jeśli ten interfejs znajduje się w tym samym pakiecie).
- Istnieje możliwość przekazywania złożonych typów danych, implementujących interfejs `android.os.Parcelable`. Wymagana jest instrukcja `import` w pliku AIDL dla tych typów danych.
- Język AIDL obsługuje w ograniczonym stopniu interfejsy `java.util.List` i `java.util.Map`. Dopuszczalne typy danych dla elementów w zbiorze obejmują proste typy danych Java, `String`, `CharSequence` lub `android.os.Parcelable`. Instrukcje `import` nie są wymagane dla interfejsów `List` lub `Map`, ale są potrzebne dla interfejsu `Parcelable`.
- Złożone typy danych — poza typem `String` — wymagają zdefiniowania wskaźnika kierunkowego. Zaliczane są do nich parametry `in`, `out` oraz `inout`. Parametr `in` oznacza, że wartość jest definiowana przez klienta; dzięki parametrowi `out` wartość zostaje określona przez usługę; w przypadku parametru `inout` wartość określa ją zarówno klient, jak i usługa.

Interfejs Parcelable przekazuje Androidowi informacje, w jaki sposób obiekty mają być serializowane oraz deserializowane w procesie szeregowania lub rozszeregowania. Na listingu 11.28 została pokazana klasa Person implementująca interfejs Parcelable.

**Listing 11.28.** Implementowanie interfejsu Parcelable

```
// Jest to plik Person.java
package com.androidbook.services.stock2;
import android.os.Parcel;
import android.os.Parcelable;

public class Person implements Parcelable {
    private int age;
    private String name;
    public static final Parcelable.Creator<Person> CREATOR =
new Parcelable.Creator<Person>() {
        public Person createFromParcel(Parcel in) {
            return new Person(in);
        }

        public Person[] newArray(int size) {
            return new Person[size];
        }
    };

    public Person() {
    }

    private Person(Parcel in) {
        readFromParcel(in);
    }

    @Override
    public int describeContents() {
        return 0;
    }

    @Override
    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(age);
        out.writeString(name);
    }

    public void readFromParcel(Parcel in) {
        age = in.readInt();
        name = in.readString();
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
}

```

Aby zaimplementować ten kod, utwórzmy nowy projekt Androida nazwany StockQuote →Service2. Przypiszmy polu *Create Activity* aktywność o nazwie *MainActivity* i skorzystajmy z pakietu *com.androidbook.services.stock2*. Następnie dodajmy powyższy plik *Person.java* do pakietu *com.androidbook.services.stock2* w naszym nowym projekcie.

Interfejs *Parcelable* definiuje kontrakt odpowiedzialny za dołączanie lub odłączanie obiektów w trakcie procesu szeregowania lub rozszeregowania. Podstawą interfejsu *Parcelable* jest pojemnik *Parcel*. Klasa *Parcel* jest szybkim mechanizmem serializowania i deserializowania, zaprojektowanym specjalnie dla komunikacji międzyprocesowej w Androidzie. W klasie tej są zawarte metody pozwalające na rozmieszczanie członków klasy w pojemniku oraz uzyskiwanie do nich dostępu. Żeby zaimplementować obiekt komunikacji międzyprocesowej we właściwy sposób, należy postępować zgodnie z następującym algorytmem:

1. Zaimplementuj interfejs *Parcelable*. Oznacza to konieczność implementacji metod *writeToParcel()* i *readFromParcel()*. Pierwsza metoda służy do zapisania obiektu w paczce, druga natomiast pozwala odczytywać obiekty umieszczone w paczce. Pamiętajmy, że kolejność odczytywania właściwości musi być taka sama jak kolejność ich zapisywania.
2. Dodaj właściwość *static final* o nazwie *CREATOR* do klasy. Właściwość ta wymaga implementacji interfejsu *android.os.Parcelable.Creator<T>*.
3. Przygotuj dla interfejsu *Parcelable* konstruktor, który będzie tworzył obiekty klasy *Parcel*.
4. Zdefiniuj klasę *Parcelable* w pliku *.aidl* odpowiadającym plikowi *.java*, w którym zawarty jest złożony typ danych. Kompilator AIDL będzie szukał tego pliku podczas komplikowania plików AIDL. Na listingu 11.29 umieściliśmy przykładowy plik *Person.aidl*. Plik ten powinien się znajdować w tym samym miejscu co plik *Person.java*.

#### Uwaga!

W przypadku interfejsu *Parcelable* możerodzić się pytanie, dlaczego w Androidzie nie został wykorzystany wbudowany w środowisko Java mechanizm serializacji. Okazuje się, że twórcy Androida uznali proces serializacji w środowisku Java za zbyt powolny, aby spełnić wymogi komunikacji międzyprocesowej w Androidzie. Zostało zatem utworzone rozwiązanie w postaci interfejsu *Parcelable*. Należy w nim jawnie serializować członków klasy, jednak w zamian cały proces przebiega o wiele szybciej.

Należy również uświadomić sobie, że istnieją w Androidzie dwa procesy umożliwiające przekazywanie danych do innego procesu. Pierwszy z nich polega na przekazaniu danych do aktywności za pomocą intencji, a drugi na przesłaniu interfejsu *Parcelable* do usługi. Te dwa mechanizmy nie są stosowane wymiennie i nie należy ich ze sobą mylić. Oznacza to, że interfejs *Parcelable* nie jest przekazywany do aktywności. Jeżeli chcemy uruchomić aktywność i przekazać jej dane, powinniśmy wykorzystać w tym celu intencję. Interfejs *Parcelable* jest przeznaczony wyłącznie do użytku jako część definicji AIDL.

**Listing 11.29.** Przykładowy plik Person.aidl

```
// Jest to plik Person.aidl
package com.androidbook.services.stock2;
parcelable Person;
```

---

Będziemy potrzebować pliku *.aidl* dla każdego interfejsu *Parcelable* w projekcie. W tym przypadku posiadamy tylko jeden interfejs *Parcelable* — *Person*. Warto zauważyc, że nie został utworzony plik *Person.java* w katalogu *gen*. Należało się tego spodziewać. Plik ten utworzyliśmy już wcześniej.

Zastosujmy teraz klasę *Person* w usłudze zdalnej. Żeby nie komplikować sprawy, zmodyfikujemy nasz obiekt *IStockQuoteService*, dzięki czemu będzie pobierał parametr wejściowy typu danych klasy *Person*. Pomyśl jest taki, aby klienci przekazywały klasę *Person* do usługi w celu powiadomienia, jaka aktywność żąda wyniku notowania. Nowy plik *IStockQuoteService.aidl* został zaprezentowany na listingu 11.30.

**Listing 11.30.** Przekazywanie usługom plików parcelowanych

```
// Jest to plik IStockQuoteService.aidl
package com.androidbook.services.stock2;
import com.androidbook.services.stock2.Person;

interface IStockQuoteService
{
    String getQuote(in String ticker,in Person requester);
}
```

---

Metoda *getQuote()* przyjmuje obecnie dwa parametry: symbol notowanej firmy i obiekt *Person* określający, jaki obiekt wysyła żądanie. Zwróćmy uwagę, że umieściliśmy wskaźniki kierunkowe dla tych parametrów, ponieważ typy danych tych parametrów nie są proste, oraz że wprowadziliśmy instrukcję *import* wobec klasy *Person*. Klasa *Person* znajduje się w tym samym pakiecie co definicja usługi (*com.androidbook.services.stock2*).

Implementacja usługi wygląda teraz tak jak na listingu 11.31, a jej układ graficzny został umieszczony na listingu 11.32.

**Listing 11.31.** Implementacja usługi StockQuoteService2

```
package com.androidbook.services.stock2;
// Jest to plik StockQuoteService2.java

import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.os.RemoteException;

public class StockQuoteService2 extends Service
{
    private NotificationManager notificationMgr;
```

```

public class StockQuoteServiceImpl extends IStockQuoteService.Stub
{
    @Override
    public String getQuote(String ticker, Person requester)
        throws RemoteException {
        return "Witaj "+requester.getName()+"! Notowanie dla "+ticker+" wynosi 20.0";
    }

}

@Override
public void onCreate() {
    super.onCreate();

    notificationMgr =
(NotificationManager) getSystemService(NOTIFICATION_SERVICE);

    displayNotificationMessage("Metoda onCreate() wywołana w StockQuoteService2");
}
@Override
public void onDestroy()
{
    displayNotificationMessage("Metoda onDestroy() wywołana w StockQuoteService2");
    // Usuwa wszelkie powiadomienia z tej usługi
    notificationMgr.cancelAll();
    super.onDestroy();
}

@Override
public IBinder onBind(Intent intent)
{
    displayNotificationMessage("Metoda onBind() wywołana w StockQuoteService2");
    return new StockQuoteServiceImpl();
}

private void displayNotificationMessage(String message)
{
    Notification notification = new Notification(R.drawable.emo_im_happy,
message,System.currentTimeMillis());

    PendingIntent contentIntent =
PendingIntent.getActivity(this, 0, new Intent(this, MainActivity.class), 0);

    notification.setLatestEventInfo(this, "StockQuoteService2", message,
contentIntent);
    notification.flags = Notification.FLAG_NO_CLEAR;
    notificationMgr.notify(R.id.app_notification_id, notification);
}
}

```

**Listing 11.32.** Układ graficzny usługi StockQuoteService2

---

```

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

```

```
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Tutaj usługa poprosi o pomoc."
    />
</LinearLayout>
```

---

Różnica pomiędzy obecną a poprzednią implementacją polega na tym, że teraz ponownie wprowadzamy powiadomienia oraz przypisujemy notowaniu wartość `string`, a nie `double`. Przekazywany użytkownikowi ciąg znaków zawiera uzyskane z obiektu `Person` informacje o elemencie żądającym wyniku notowania, co ma na celu zademonstrowanie poprawnego odczytania wartości wysłanej przez klienta oraz poprawnego przekazania usłudze obiektu `Person`.

Musimy wykonać jeszcze kilka czynności, aby nasz przykładowy kod działał:

1. Znajdź plik `emo_im_happy.png` z katalogu `Android SDK/platforms/android-2.1/data/res/drawable-mdpi` i skopiuj go do folderu `/res/drawable` naszego projektu. Możemy również zmienić w kodzie nazwę zasobu i wstawić dowolny obraz w katalogu `drawable`.
2. Dodaj nowy znacznik `<item type="id" name="app_notification_id"/>` w pliku `/res/values/strings.xml`.
3. Musimy zmodyfikować kod aplikacji w pliku `AndroidManifest.xml`, tak jak pokazano na listingu 11.33.

**Listing 11.33.** Zmodyfikowany węzeł `<application>` w pliku `AndroidManifest.xml` usługi `StockQuoteService2`

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.services.stock2"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".MainActivity"
            android:label="@string/app_name"
            android:launchMode="singleTop" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
        <service android:name="StockQuoteService2">
            <intent-filter>
                <action android:name="com.androidbook.services.
                    stock2.IStockQuoteService" />
            </intent-filter>
        </service>
    </application>
    <uses-sdk android:minSdkVersion="7" />
</manifest>
```

---

Podczas gdy można wprowadzać skróconą notację kropkową w atrybucie `android:name=>=".MainActivity"`, takie postępowanie nie jest już poprawne w przypadku notacji używanej w ramach znacznika `<action>` umieszczonego w środku znacznika `<intent-filter>` usługi. Musimy wprowadzić pełną notację, w przeciwnym wypadku klient nie odnajdzie specyfikacji usługi.

Na koniec wykorzystamy domyślny plik `MainActivity.java`, który wyświetla prosty układ graficzny z nieskomplikowaną wiadomością. Pokazaliśmy wcześniej, w jaki sposób można uruchomić aktywność za pomocą powiadomienia. Aktywność spełniałaby również taką rolę w standardowej aplikacji, w naszym przykładzie jednak nie będziemy niczego komplikować. Gdy już posiadamy implementację naszej usługi, utworzmy nowy projekt Androida, nazwany `StockQuoteClient2`. Pakiet niech nosi nazwę `com.dave`, a aktywność — `MainActivity`. Aby zaimplementować klienta zdolnego do przekazywania obiektu `Person` usłudze, musimy skopiować do jego projektu wszystkie elementy projektu usługi wymagane przez klienta. W poprzednim przykładzie potrzebowaliśmy jedynie pliku `IStockQuoteService.aidl`. Teraz musimy skopiować również pliki `Person.java` i `Person.aidl`, ponieważ obiekt `Person` jest częścią interfejsu. Po skopiowaniu tych trzech plików do projektu klienta musimy zmodyfikować pliki `main.xml` tak jak na listingu 11.34 i `MainActivity.java` zgodnie z listkiem 11.35. Ewentualnie możemy po prostu zimportować cały projekt z pliku umieszczonego na oficjalnej stronie książki.

**Listing 11.34.** Zaktualizowany plik main.xml aplikacji StockQuoteClient2

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

<ToggleButton android:id="@+id/bindBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textOff="Podłącz" android:textOn="Odłącz"
    android:onClick="doClick" />

<Button android:id="@+id/callBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Wywołaj ponownie" android:enabled="false"
    android:onClick="doClick" />
</LinearLayout>
```

---

**Listing 11.35.** Wywoływanie usługi za pomocą obiektu Parcelable

---

```
package com.dave;
// Jest to plik MainActivity.java
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
```

```
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.Toast;
import android.widget.ToggleButton;

import com.androidbook.services.stock2.IStockQuoteService;
import com.androidbook.services.stock2.Person;

public class MainActivity extends Activity {

    protected static final String TAG = "StockQuoteClient2";
    private IStockQuoteService stockService = null;
    private ToggleButton bindBtn;
    private Button callBtn;

    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        bindBtn = (ToggleButton)findViewById(R.id.bindBtn);
        callBtn = (Button)findViewById(R.id.callBtn);
    }

    public void doClick(View view) {
        switch(view.getId()) {
            case R.id.bindBtn:
                if(((ToggleButton) view).isChecked()) {
                    bindService(new Intent(
                        IStockQuoteService.class.getName(),
                        serConn, Context.BIND_AUTO_CREATE));
                } else {
                    unbindService(serConn);
                    callBtn.setEnabled(false);
                }
                break;
            case R.id.callBtn:
                callService();
                break;
        }
    }

    private void callService() {
        try {
            Person person = new Person();
            person.setAge(47);
            person.setName("Dave");
            String response = stockService.getQuote("ANDROID", person);
            Toast.makeText(MainActivity.this,
                "Wartość z usługi wynosi "+response,

```

```

        Toast.LENGTH_SHORT).show();
    } catch (RemoteException ee) {
        Log.e("MainActivity", ee.getMessage(), ee);
    }
}

private ServiceConnection serConn = new ServiceConnection() {

    @Override
    public void onServiceConnected(ComponentName name,
                                   IBinder service)
    {
        Log.v(TAG, "wywolana metoda onServiceConnected()");
        stockService = IStockQuoteService.Stub.asInterface(service);
        bindBtn.setChecked(true);
        callBtn.setEnabled(true);
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {
        Log.v(TAG, "wywolana metoda onServiceDisconnected()");
        bindBtn.setChecked(false);
        callBtn.setEnabled(false);
        stockService = null;
    }
};

protected void onDestroy() {
    if(callBtn.isEnabled())
        unbindService(serConn);
    super.onDestroy();
}
}

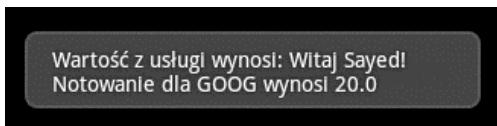
```

Nasza usługa jest już gotowa do uruchomienia. Należy jeszcze przesłać ją do emulatora, zanim uruchomimy klienta. Interfejs użytkownika powinien wyglądać tak jak na rysunku 11.8.



**Rysunek 11.8.** Interfejs użytkownika w aplikacji StockQuoteClient2

Przyjrzyjmy się temu, co otrzymaliśmy. Podobnie jak poprzednio, tworzymy powiązanie z usługą, a następnie możemy wywołać jej metodę. Dzięki metodzie `onServiceConnected()` dowiadujemy się, że nasza usługa jest uruchomiona, możemy więc aktywować przycisk *Wywołaj ponownie*, dzięki któremu wywołujemy metodę `callService()`. Jak widać, tworzymy nowy obiekt `Person` i konfigurujemy jego właściwości `Age` i `Name`. Następnie uruchamiamy usługę i otrzymujemy wyświetlony wynik jej wywołania. Został on zilustrowany na rysunku 11.9.



Rysunek 11.9. Wynik wywołania usługi za pomocą interfejsu Parcelable

Zauważmy, że podczas wywoływanego usługi zostaje wyświetlone powiadomienie w pasku stanu. Pochodzi ono z samej usługi. We wcześniejszej części rozdziału poruszyliśmy temat powiadomień jako sposobu komunikacji usługi z użytkownikiem. Usługi pozostają w tle i nie wyświetlają interfejsu użytkownika w żadnej postaci. Ale co w przypadku, gdy usługa musi nawiązać komunikację z użytkownikiem? Chociaż kusząca jest myśl, żeby usługa wywołała aktywność, nie powinna nigdy tej czynności wykonywać bezpośrednio. Zamiast tego usługa powinna utworzyć powiadomienie, które informuje użytkownika, w jaki sposób może się dostać do żądanej aktywności. Zostało to zademonstrowane w powyższym przykładzie. Zdefiniowaliśmy prosty układ graficzny i implementację aktywności dla naszej usługi. Kiedy utworzyliśmy powiadomienie w usłudze, skonfigurowaliśmy również aktywność dla niego. Użytkownik może kliknąć to powiadomienie i wtedy uruchomi aktywność będącą częścią usługi. W ten sposób będzie mógł komunikować się z usługą.

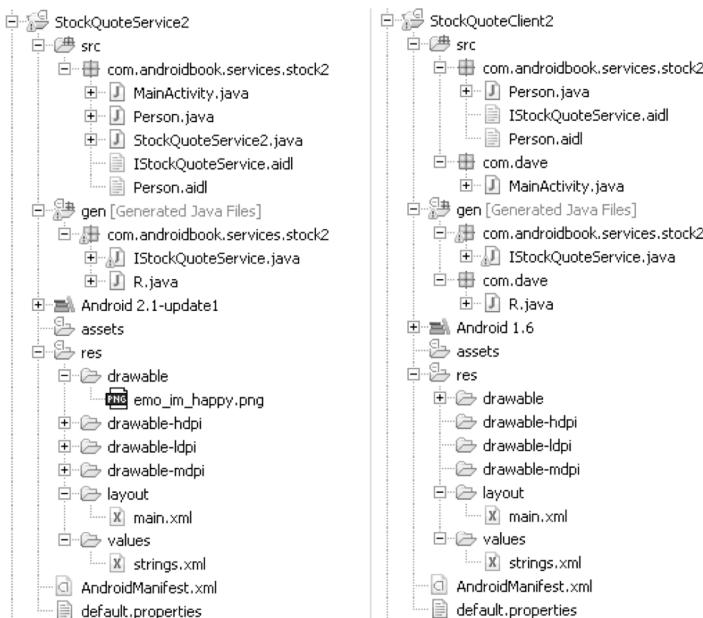
Powiadomienia są zapisywane, zatem możemy je odczytać, klikając przycisk *Menu* na ekranie startowym Androida i wybierając opcję *Notifications*. Użytkownik może również przeciągnąć ikonę powiadomień z paska stanu, aby przejrzeć powiadomienia. Zwrócmy uwagę na sposób wykorzystania wywołania metody `setLatestEventInfo()` oraz na fakt, że dla każdej wiadomości używamy tego samego identyfikatora. Dzięki temu aktualizujemy przez cały czas tylko jedno powiadomienie, zamiast tworzyć nowe wpisy. Zatem jeśli po kilkakrotnym kliknięciu przycisków *Przylacz*, *Wywołaj ponownie* i *Odlacz* przejdziemy do ekranu *Notifications*, ujrzymy tylko jedno powiadomienie — ostatnie wysłane przez usługę *BackgroundService*. Gdybyśmy używali kilku różnych identyfikatorów, moglibyśmy utworzyć kilka wpisów powiadomień i każdy z nich mógłby być aktualizowany oddzielnie. Do powiadomień można dodać również dodatkowe „zachęty” użytkownika, takie jak dźwięk, zmiana obrazu oraz (lub) vibracja.

Warto również przejrzeć artefakty projektu usługi oraz projektu wywołującego ją programu (rysunek 11.10).

Na rysunku 11.10 zostały ukazane artefakty projektu w środowisku Eclipse dla usługi (z lewej strony) oraz dla klienta (z prawej strony). Zauważmy, że na kontrakt pomiędzy klientem a usługą składają się pliki języka AIDL oraz obiekty `Parcelable`, skopiowane pomiędzy obydwooma projektami. Dlatego po obydwu stronach widzimy pliki *IStockQuoteService.aidl*, *Person.java* i *Person.aidl*. Ponieważ kompilator AIDL generuje z artefaktów języka AIDL interfejs Java, klasy pośredniczące i tak dalej, podczas kopiowania tych artefaktów do projektu klienta tworzony jest plik *IStockQuoteService.java* po jego stronie.

Wiemy już, w jaki sposób wymieniać złożone typy danych pomiędzy usługami a klientami. Zajmijmy się pokrótkę innym istotnym aspektem wywoływania usług: różnicami pomiędzy synchronicznym a asynchronicznym wywoływaniem usługi.

Wszystkie wywołania usług są przeprowadzane w sposób synchroniczny. Nasuwa się oczywiste pytanie, czy musimy implementować wszystkie wywołania usług w wątku roboczym. Niekończenie. Dla większości innych platform standardem jest wykorzystywanie przez klienta usługi, która jest dla niego zupełnie nieznanym obiektem, a zatem musi on przedsięwziąć odpowiednie



Rysunek 11.10. Artefakty usługi i klienta

środki ostrożności przed wywołaniem usługi. W przypadku Androida najprawdopodobniej będziemy znać usługi (często sami jesteśmy ich twórcami), zatem możemy podjąć przemyślaną decyzję. Jeśli wiemy, że wywoływana metoda będzie wymagała dużej mocy obliczeniowej, powinniśmy się zastanowić nad umieszczeniem jej w wątku pobocznym. Jeśli zaś jesteśmy przekonani o tym, że metoda nie będzie powodowała problemów z wydajnością, możemy spokojnie umieścić ją w wątku interfejsu użytkownika. W przypadku gdy stwierdzimy, że wywołanie usługi najlepiej umieścić w wątku roboczym, możemy utworzyć wątek, a następnie wywołać usługę. Następnie można przekazywać komunikaty do wątku interfejsu UI.

## Przykład aplikacji użytkowej korzystającej z usług

Prezentowaliśmy dotychczas różnorodne sposoby wywoływanego usług HTTP oraz implementowania usług systemu Android. W tym podrozdziale pokażemy, w jaki sposób można przetłumaczyć tekst z jednego języka na inny za pomocą tych usług oraz interfejsu Tłumacz Google, który stanowi usługę internetową opartą na protokole HTTP. Najpierw jednak poświęcimy nieco czasu na omówienie tego interfejsu.

### Interfejs Tłumacz Google

Urządzenia mobilne nie do końca się nadają do tłumaczenia tekstu. Język polski składa się z setek tysięcy, może nawet z ponad miliona wyrazów (w zależności od tego, jak definiujemy słowo „język polski”). Wczytanie wszystkich wyrazów i reguł danego języka do pamięci urządzenia mobilnego w celu zapewnienia poprawnego przetłumaczenia tekstu z jednego języka na drugi jest na razie nieosiągalne.

Firma Google umieściła w internecie interfejs API służący do tłumaczenia tekstu. Pobiera on ciąg znaków tekstowych oraz dwie specyfikacje języków: jedną dla języka źródłowego i drugą dla języka docelowego. Istnieje jednak pewien haczyk. Początkowo intencja tej usługi miała być wywoływana z poziomu stron WWW, a nie urządzeń mobilnych. Warunki korzystania z interfejsu Google AJAX Language API (taka jest jego oficjalna nazwa) nie zawierają informacji na temat urządzeń używających Androida, inaczej niż to ma miejsce w przypadku warunków korzystania z interfejsu API Google Maps. Poniższy odnośnik skieruje nas do informacji o warunkach korzystania z interfejsu AJAX Language API:

<http://code.google.com/apis/ajaxlanguage/terms.html>

Chociaż nie jest do końca jasne, czy projektanci aplikacji dla systemu Android mogą korzystać z tego interfejsu, to jednak prezentację tego interfejsu na konferencji Google I/O w maju 2009 roku przeprowadzono na programie uruchomionym w Androidzie! Być może do chwili wydania tej książki firma Google określi oddzielne warunki korzystania z interfejsu AJAX Language API dla systemów Android lub opublikuje aktualizację tych warunków, w której sprecyzowane zostaną dopuszczalne formy stosowania tego interfejsu w Androidzie. Istnieje również wersja 2 tego interfejsu, stworzona przez firmę Google Labs, warto więc również zwrócić uwagę na jej rozwój<sup>1</sup>. Tymczasem mamy kilka możliwości. Po pierwsze, możemy bezpośrednio zastosować interfejs AJAX Language API z poziomu naszej aplikacji, co też wkrótce zaprezentujemy. Po drugie, możemy uzyskać dostęp do tego interfejsu za pomocą kontrolowanego przez siebie serwera sieciowego, na przykład serwera będącego pośrednikiem dla interfejsu AJAX Language API. Nasza aplikacja będzie połączona z serwerem sieciowym, który będzie wywoływał ten interfejs. Jeżeli możemy wykorzystać własny serwer sieciowy, będzie o wiele łatwiej wyłączyć dostęp do interfejsu AJAX Language API z poziomu aplikacji, ponieważ pilnujemy punktu kontrolnego znajdującego się pomiędzy nimi. Oczywiście, na niewiele zda się kontrola dostępu aplikacji do interfejsu, jeżeli nie będzie można dłużej wykorzystywać tej usługi. Powinniśmy zapewnić chociaż jakąś formę odpowiedzi ze strony aplikacji w przypadku utraty łączności z usługą firmy Google, aby powiadomić o tym użytkownika. Jeżeli nastąpi sytuacja, w której firma Google zażąda od nas zaprzestania korzystania z interfejsu AJAX Language API, tak naprawdę nie będziemy mieć wielkiego wyboru; aplikacja została zainstalowana na wielu urządzeniach i będzie próbowała się połączyć z serwerem, chyba że wprowadziliśmy do niej jakieś rozwiązanie zatrzymujące korzystanie z tego interfejsu.

Firma Google ma prawo zabronić dostępu do tej usługi, jednak nie można tego wykonać w prosty sposób. W warunkach korzystania z interfejsu AJAX Language API nie określono konieczności stosowania klucza API, chociaż w dokumentacji dla programistów (<http://code.google.com/apis/ajaxlanguage/documentation/>) znalazło się stwierdzenie, że *należy* stosować atrybut *REFERER* oraz że *powinniśmy* używać klucza API. Bez tych elementów żądania pochodzące z urządzeń użytkowników będą pojawiały się anonimowo i firma Google nie będzie posiadała możliwości skontaktowania się z programistą w przypadku wystąpienia jakichś problemów związanych ze stosowaniem tego interfejsu API. W poniższym przykładzie widać, że wprowadziliśmy wartość nagłówka *REFERER* (w kodzie *Translator.java*), lecz ominąliśmy fragment związany z kluczem API. Jeżeli chcemy przesyłać wartość klucza API do interfejsu AJAX Language, musimy najpierw go otrzymać od firmy Google. Zwracamy uwagę, że do omawianego interfejsu nie należy wprowadzać stosowanych wcześniej kluczy API związanych z aplikacją Google Maps. Aby zare-

---

<sup>1</sup> Przewidywania autorów okazały się mylne; w momencie wydania polskiej wersji książki cała rodzina interfejsów Google Language została zdeprecjonowana i przestanie być obsługiwana w grudniu 2011 roku. Wyjątkiem jest wersja 2 tego interfejsu, która stała się płatną usługą — *przyp. tłum.*

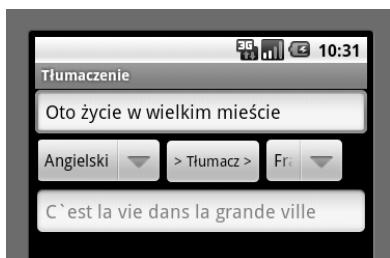
jestrować klucz API AJAX, wystarczy wysłać adres URL swojej strony WWW (dokładnie taki sam jak wprowadzony w nagłówku REFERER) i zaakceptować warunki korzystania z usług. Otrzymawszy klucz API, należy dodać go do adresu URL interfejsu API AJAX w następujący sposób:

```
&key=Your_API_key_goes_here_with_no_quotation_marks
```

Jeżeli zdecydujemy się przekazać interfejsowi API AJAX klucz API, wartość atrybutu REFERER musi być adresem URL — lub jakimś jego podelementem — za pomocą którego został utworzony ten klucz. W przeciwnym wypadku nie otrzymamy wyników.

## Stosowanie interfejsu Tłumacz Google

W pozostałej części rozdziału pokażemy, w jaki sposób utworzyć aplikację wywołującą bezpośrednio interfejs Google AJAX Language API. Do tej pory zaprezentowaliśmy pojedynczo wszystkie elementy potrzebne do tłumaczenia. Teraz połączymy je ze sobą. W poniższym przykładzie zbudujemy program zawierający edytowalne pole *EditText*, kontrolki *Spinner* pozwalające na wybór języka źródłowego i docelowego, a także drugie pole *EditText* (tylko do odczytu) wyświetlające przetłumaczony wynik. Usługę będziemy wywoływać poprzez internet. W celu odizolowania interfejsu UI od procesu przetwarzania, który może być dosyć czasochłonny, wykorzystamy usługę. Jednym z zamieszczonych przez nas dodatków w tej aplikacji jest projekt Jakarta Commons Lang, umożliwiający przede wszystkim wykorzystanie funkcji unescape do wyświetlania kodów znaków XML w systemie Unicode. Omówimy również ten temat. Na rysunku 11.11 widzimy układ graficzny tej aplikacji.



Rysunek 11.11. Interfejs użytkownika aplikacji demonstracyjnej służącej do tłumaczeń

Na kod naszej aplikacji będą się składać następujące pliki:

- */res/layout/main.xml* (listing 11.36),
- */res/values/strings.xml* (listing 11.37),
- */res/values/arrays.xml* (listing 11.38),
- *ITranslate.aidl* w katalogu */src* (listing 11.39),
- *MainActivity.java* (listing 11.40),
- *TranslateService.java* (listing 11.41), zapewniający semantykę usługi,
- *Translator.java* (listing 11.42), zawierający wywołanie właściwej usługi firmy Google,
- *AndroidManifest.xml* (listing 11.43).

W przypadku tej aplikacji zastosowaliśmy klasę *HttpURLConnection*, a nie *HttpClient*, zatem Czytelnik może się przekonać, w jaki sposób jest ona wykorzystywana przez standardowy program użytkowy.

**Listing 11.36.** Układ graficzny XML służący do zaimplementowania wersji demonstracyjnej aplikacji tłumaczącej

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent">

    <EditText android:id="@+id/input"
        android:hint="@string/input"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent" />

    <Spinner android:id="@+id/from"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/input"
        android:prompt="@string/prompt" />

    <Button android:id="@+id/translateBtn"
        android:text="@string/translateBtn"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/input"
        android:layout_toRightOf="@id/from"
        android:enabled="false" />

    <Spinner android:id="@+id/to"
        android:layout_weight="1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/input"
        android:layout_toRightOf="@+id/translateBtn"
        android:prompt="@string/prompt" />

    <EditText android:id="@+id/translation"
        android:hint="@string/translation"
        android:layout_height="wrap_content"
        android:layout_width="fill_parent"
        android:editable="false"
        android:layout_below="@+id/from" />

    <TextView android:id="@+id/poweredBy"
        android:text="powered by Google"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true" />

</RelativeLayout>
```

---

**Listing 11.37.** Plik zasobów zawierający ciągi znaków

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/values/strings.xml -->
<resources>
    <string name="translateBtn">Tłumacz</string>
    <string name="input">Wprowadź tekst do przetłumaczenia</string>
    <string name="translation">Tutaj pojawi się tłumaczenie tekstu</string>
    <string name="prompt">Wybierz język</string>
</resources>
```

---

**Listing 11.38.** Plik zasobów zawierający tablice

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/values/arrays.xml -->
<resources>
    <string-array name="languages">
        <item>Chiński</item>
        <item>Polski</item>
        <item>Francuski</item>
        <item>Niemiecki</item>
        <item>Japoński</item>
        <item>Hiszpański</item>
    </string-array>
    <string-array name="language_values">
        <item>zh</item>
        <item>pl</item>
        <item>fr</item>
        <item>de</item>
        <item>ja</item>
        <item>es</item>
    </string-array>
</resources>
```

---

**Listing 11.39.** Plik AIDL usługi tłumacza

---

```
// Jest to plik ITranslate.aidl umieszczony w katalogu /src
interface ITranslate {
    String translate(in String text, in String from, in String to);
}
```

---

**Listing 11.40.** Główny plik aplikacji — MainActivity.java

---

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.Handler;
import android.os.IBinder;
```

```
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Spinner;
import android.widget.TextView;

public class MainActivity extends Activity implements OnClickListener {
    static final String TAG = "Translator";
    private EditText inputText = null;
    private TextView outputText = null;
    private Spinner fromLang = null;
    private Spinner toLang = null;
    private Button translateBtn = null;
    private String[] langShortNames = null;
    private Handler mHandler = new Handler();

    private ITranslate mTranslateService;

    private ServiceConnection mTranslateConn = new ServiceConnection() {
        public void onServiceConnected(ComponentName name, IBinder service) {
            mTranslateService = ITranslate.Stub.asInterface(service);
            if (mTranslateService != null) {
                translateBtn.setEnabled(true);
            } else {
                translateBtn.setEnabled(false);
                Log.e(TAG, "Nie można znaleźć usługi TranslateService");
            }
        }

        public void onServiceDisconnected(ComponentName name) {
            translateBtn.setEnabled(false);
            mTranslateService = null;
        }
    };

    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        setContentView(R.layout.main);
        inputText = (EditText) findViewById(R.id.input);
        outputText = (EditText) findViewById(R.id.translation);
        fromLang = (Spinner) findViewById(R.id.from);
        toLang = (Spinner) findViewById(R.id.to);

        langShortNames = getResources().getStringArray(R.array.language_values);

        translateBtn = (Button) findViewById(R.id.translateBtn);
        translateBtn.setOnClickListener(this);

        ArrayAdapter<?> fromAdapter = ArrayAdapter.createFromResource(this,
            R.array.languages, android.R.layout.simple_spinner_item);

        fromAdapter.setDropDownViewResource(android.R.layout.simple_dropdown_item_1line);
```

```
fromLang.setAdapter(fromAdapter);
fromLang.setSelection(1); // Polski

ArrayAdapter<?> toAdapter = ArrayAdapter.createFromResource(this,
    R.array.languages, android.R.layout.simple_spinner_item);
toAdapter.setDropDownViewResource(android.R.layout.simple_dropdown_item_1line);
toLang.setAdapter(toAdapter);
toLang.setSelection(3); // Niemiecki

inputText.selectAll();

Intent intent = new Intent(Intent.ACTION_VIEW);
bindService(intent, mTranslateConn, Context.BIND_AUTO_CREATE);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    unbindService(mTranslateConn);
}

public void onClick(View v) {
    if (inputText.getText().length() > 0) {
        doTranslate();
    }
}

private void doTranslate() {
    mHandler.post(new Runnable() {
        public void run() {
            String result = "";
            try {
                int fromPosition = fromLang.getSelectedItemPosition();
                int toPosition = toLang.getSelectedItemPosition();
                String input = inputText.getText().toString();
                if(input.length() > 5000)
                    input = input.substring(0,5000);
                Log.v(TAG,"Tłumaczenie z " + langShortNames[fromPosition] + " na " +
                      langShortNames[toPosition]);
                result = mTranslateService.translate(input,
                                                    langShortNames[fromPosition],
                                                    langShortNames[toPosition]);
                if (result == null) {
                    throw new Exception("Proces tłumaczenia zakończony niepowodzeniem");
                }
                outputText.setText(result);
                inputText.selectAll();
            } catch (Exception e) {
                Log.e(TAG, "Błąd: " + e.getMessage());
            }
        }
    });
}
});
```

**Listing 11.41.** Plik usługi tłumaczenia — TranslateService.java

```
// Jest to plik TranslateService.java
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class TranslateService extends Service {
    public static final String TAG = "TranslateService";

    private final ITranslate.Stub mBinder = new ITranslate.Stub() {
        public String translate(String text, String from, String to) {
            try {
                return Translator.translate(text, from, to);
            } catch (Exception e) {
                Log.e(TAG, "Nie udało się przetłumaczyć: " + e.getMessage());
                return null;
            }
        }
    };

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }
}
```

---

**Listing 11.42.** Plik zawierający funkcję tłumaczenia

```
// Jest to plik Translator.java
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import java.net.URLEncoder;

import org.apache.commons.lang.StringEscapeUtils;
import org.json.JSONObject;
import android.util.Log;

public class Translator {
    private static final String ENCODING = "UTF-8";
    private static final String URL_BASE =
    "http://ajax.googleapis.com/ajax/services/language/translate?v=1.0&langpair=";
    private static final String INPUT_TEXT = "&q=";
    private static final String MY_SITE = "http://my.website.com";
    private static final String TAG = "Translator";

    public static String translate(String text, String from, String to) throws Exception
    {
        try {
            StringBuilder url = new StringBuilder();
            url.append(URL_BASE).append(from).append("%7C").append(to);
```

```

url.append(INPUT_TEXT).append(URLEncoder.encode(text, ENCODING));

HttpURLConnection conn = (HttpURLConnection) new URL(url.toString())
    .openConnection();
conn.setRequestProperty("REFERER", MY_SITE);
conn.setDoInput(true);
conn.setDoOutput(true);
try {
    InputStream is= conn.getInputStream();
    String rawResult = makeResult(is);

    JSONObject json = new JSONObject(rawResult);
    String result = ((JSONObject)json.get("responseData"))
        .getString("translatedText");
    return (StringEscapeUtils.unescapeXml(result));
} finally {
    conn.getInputStream().close();
    if(conn.getErrorStream() != null)
        conn.getErrorStream().close();
}
} catch (Exception ex) {
    throw ex;
}
}

private static String makeResult(InputStream inputStream) throws Exception {
    StringBuilder outputString = new StringBuilder();
    try {
        String string;
        if (inputStream != null) {
            BufferedReader reader =
                new BufferedReader(new InputStreamReader(inputStream, ENCODING));
            while (null != (string = reader.readLine())) {
                outputString.append(string).append('\n');
            }
        }
    } catch (Exception ex) {
        Log.e(TAG, "Błąd podczas odczytu strumienia tłumaczenia.", ex);
    }
    return outputString.toString();
}
}

```

---

**Listing 11.43.** Plik AndroidManifest.xml

```

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.translation"
    android:versionName="1.0"
    android:versionCode="1" >

    <application android:label="Tłumaczenie"
        android:icon="@drawable/icon">

        <activity android:name="MainActivity" android:label="Translate">

```

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

<service android:name="TranslateService" android:label="Translate">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</service>
</application>
<uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

---

Przed poprawnym skompilowaniem przykładu musimy wprowadzić klasę pomocniczą. W projekcie Jakarta Commons Lang znajduje się klasa `StringEscapeUtils`, którą wykorzystamy do konwersji wynikowego ciągu znaków z interfejsu AJAX Language API na tekst zrozumiały dla użytkownika. Interfejs ten odsyła nam obiekty XML reprezentujące określone znaki specjalne. Na przykład odpowiednikiem apostrofu jest tu wartość ‚. Te znaki specjalne muszą być wyświetlane w sposób zrozumiały dla użytkownika. W tym celu zastosujemy projekt Jakarta Commons Lang. Można go znaleźć pod adresem:

<http://commons.apache.org/lang/>

Należy otworzyć stronę projektu Jakarta Commons Lang i pobrać plik `commons-lang.zip` lub `commons-lang.tar`, w którym zawarte są pliki `.jar`. Następnie należy je rozpakować. W środowisku Eclipse wybieramy projekt, klikamy jego nazwę prawym przyciskiem myszy i wskazujemy opcje *Build Path/Configure Build Path*. Klikamy zakładkę *Libraries* i wybieramy opcję *Add External JARs*. Wyszukujemy pobrany plik `commons-lang` i dodajemy go. Aby zakończyć proces dodawania pliku, klikamy przycisk *OK*. Cała aplikacja powinna zostać bezbłędnie zbudowana. Nic nie stoi na przeszkodzie, żeby ją teraz wypróbować. Jeżeli nie wygląda ona zbyt dobrze w orientacji pionowej, możemy wypróbować skrót klawiaturowy *Ctrl+F11*, aby przełączyć emulator w tryb orientacji poziomej. Jeżeli wątpimy w poprawność generowanych wyników, możemy je porównać z tłumaczeniem dostępnym na serwerze Google:

<http://www.google.com/uds/samples/language/translate.html>

Chcielibyśmy uczuścić Czytelnika na kilka spraw. Z powodu określonych zapisów w warunkach korzystania z usługi Google nasza przykładowa aplikacja zawiera w interfejsie użytkownika ciąg znaków *powered by Google*. Te same warunki określają maksymalny limit 5000 wprowadzanych znaków, zatem po przekroczeniu tej liczby nadmiar znaków jest usuwany. Prawdopodobnie chcemy zaprojektować tu nieco inny model, na przykład umożliwiający dzielenie tekstu na edytowalne fragmenty, które są następnie przesyłane do interfejsu API. Celowo stworzyliśmy krótką listę dostępnych języków, aby nasza aplikacja była łatwiejsza do zarządzania, można jednak bez problemu zamieścić dowolną liczbę języków w tablicy ciągów znaków. Musimy mieć jednak świadomość, że czcionki Droid mogą nie posiadać kompletnego znaków dla niektórych języków, które są dostępne w tłumaczu. Jeżeli tłumaczenie wynikowe wygląda podejrzanie, prawdopodobnie mamy problem z czcionką. Można temu zapobiec poprzez wprowadzenie dodatkowych czcionek, nie jest to jednak tematem tego rozdziału. Odpowiedzi interfejsu API przybierają postać formatu JSON. Zatem będziemy za pomocą tego formatu poddawać analizie

składniowej zwracane wynikowe ciągi znaków. Format JSON stanowi część struktury Androida, zatem nie musielibyśmy go pobierać jako osobnego pliku *.jar*.

Jedną z cech interfejsu AJAX Language API jest brak konieczności wskazania języka źródłowego. Interfejs ten spróbuje samodzielnie ustalić, jaki język jest używany. Jeżeli chcemy skorzystać z takiego rozwiązania, nie zamieszczamy wartości języka źródłowego w przekazywanym adresie URL, lecz zamiast tego w atrybucie `langpair=` zamieszczamy wartość `%7C`. Jest to przydatna funkcja, jeśli nie jesteśmy pewni, jaki język źródłowy został użyty; jednak jeżeli ilość wprowadzonego tekstu jest zbyt mała, interfejs API może nie rozpoznać języka.

## Odbośniki

Poniżej prezentujemy przydatne odnośniki, pomagające zapoznać się dokładniej z omawianymi tematami:

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów utworzonych specjalnie na potrzeby książki. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R11\_Uslugi*. Dostępny jest tu także plik *Czytaj.TXT*, stanowiący dokładną instrukcję importowania projektów do środowiska Eclipse.
- <http://hc.apache.org/httpcomponents-client-ga/tutorial/html/> — strona ta zawiera znakomite samouczki dotyczące klas `HttpClient`, w tym również informacje o uwierzytelnianiu i korzystaniu z plików cookies.

## Podsumowanie

Cały niniejszy rozdział został poświęcony usługom. Omówiliśmy sposób użytkowania zewnętrznych usług HTTP za pomocą modułu `HttpClient` firmy Apache, a także metody pisania usług przetwarzanych w tle. Pod kątem modułu `HttpClient` zademonstrowaliśmy, w jaki sposób można wywoływać metody HTTP `GET` oraz HTTP `POST`. Pokazaliśmy także zastosowanie wieloczęściowej metody `POST`.

Druga część rozdziału dotyczyła pisania usług dla systemu Android. W szczególności zajęliśmy się tworzeniem usług lokalnych i usług zdalnych. Stwierdziliśmy, że usługa lokalna jest użytkowana przez składniki (na przykład aktywności) tego samego procesu, w którym znajduje się ta usługa. Klienci usług zdalnych znajdują się poza procesami, które wywołały te usługi.



# Analiza pakietów

We wszystkich dotychczasowych rozdziałach zajmowaliśmy się podstawowymi składnikami systemu Android. Chcemy zauważyć, że to była łatwiejsza część podróży przez Androida. Począwszy od niniejszego, w kilku następnych rozdziałach (12., 13., 14. i 15.) przyjrzymy się dokładniej kolejnemu poziomowi organizacji Androida.

Badanie rozpoczniemy od zjazdzenia w głąb pakietów, procesu ich podpisywania, współdzielenia danych pomiędzy nimi oraz zapoznania się z projektami bibliotek. Zrozumiemy także kontekst linuksowego procesu, w którym jest uruchomiony plik *.apk*. Dowiemy się też, w jaki sposób wiele plików *.apk* współdzieli dane i zasoby za pomocą tego kontekstu.

Chociaż w rozdziale 10. dowiedzieliśmy się co nieco na temat podpisywania pakietów w Androidzie, dopiero teraz poznamy znaczenie, implikacje oraz zastosowania podpisanych plików JAR. W kontekście współdzielenia plików zainteresujemy się również projektami bibliotek Androida, aby zrozumieć, w jaki sposób działają oraz czy można ich używać do współdzielenia zasobów i kodu.

Rozpoczniemy od przypomnienia podstawowych informacji na temat pliku *.apk*, gdyż to stanowi bazę dalszych rozważań na temat procesów Androida.

## Pakiety i procesy

Jak już widzieliśmy w poprzednich rozdziałach, proces tworzenia aplikacji kończy się utworzeniem pliku *.apk*, który zostaje następnie podpisany i wdrożony do użytkowania w urządzeniu. Zobaczmy, czego jeszcze możemy się dowiedzieć o pakietach systemu Android.

### Szczegółowa specyfikacja pakietu

Każdy plik *.apk* posiada swój własny, niepowtarzalny identyfikator, oparty na nazwie pakietu, który zostaje zdefiniowany w pliku manifeście. Poniżej prezentujemy przykładową definicję, która będzie wykorzystywana w tym rozdziale (nazwa pakietu została oznaczona pogrubionym drukiem):

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.library.testlibraryapp"
    ...>
    ...pozostałe węzły xml
</manifest>
```

Jeżeli twórca tego pakietu podpisał go i zainstalował, nikt inny nie będzie mógł go aktualizować. Nazwa pakietu jest ściśle związana z sygnaturą, za pomocą której został podpisany. W wyniku tego programista posiadający inną sygnaturę nie może podpisywać i instalować pakietu przy użyciu nazwy wykorzystywanej już przez kogoś innego.

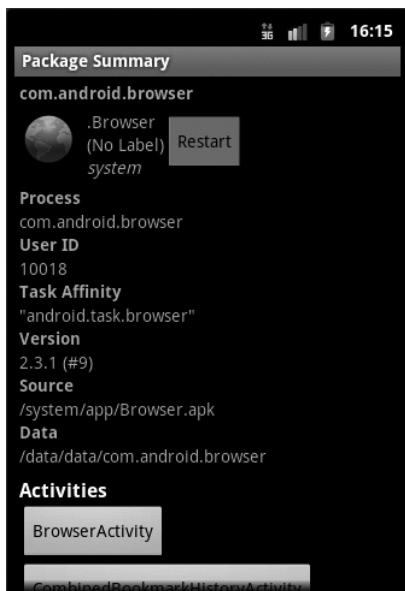
## Przekształcanie nazwy pakietu w nazwę procesu

Android wykorzystuje nazwę pakietu do utworzenia procesu, w którym będą działały składniki tego pakietu. Zostaje również przydzielony unikatowy identyfikator użytkownika, określający ten uruchomiony proces. Identyfikator ten jest w istocie wykorzystywany przez system operacyjny Linux, który stanowi podstawę Androida. Informacje te znajdziemy podczas przeglądania szczegółów na temat zainstalowanego pakietu.

## Tworzenie listy zainstalowanych pakietów

Aby przejrzeć listę pakietów zainstalowanych w emulatorze, należy uruchomić ekran startowy i otworzyć *Dev Tools/Package Browser*. Warto zauważyć, że możemy (choć nie musimy) posiadać podobną przeglądarkę pakietów na urządzeniu fizycznym. Może się to również zmienić w zależności od wersji Androida.

Po otwarciu takiej listy możemy zaznaczyć pakiet jakieś aplikacji, dajmy na to — przeglądarki, i kliknąć jego nazwę. Zobaczmy okno zawierające szczegółowe informacje o pakiecie, takie jak to widoczne na rysunku 12.1.



Rysunek 12.1. Szczegółowe informacje dotyczące pakietu Androida

Na rysunku 12.1 widać nazwę procesu, zdefiniowaną przez nazwę pakietu Java w pliku *AndroidManifest.xml*, oraz niepowtarzalny identyfikator użytkownika przydzielony do tego pakietu. W przypadku aplikacji przeglądarkowej nazwa pakietu wskazywana w pliku manifeście to `com.android.browser` (na rysunku 12.1 jako wartość atrybutu *Process*).

Wszelkie zasoby utworzone przez ten proces lub pakiet zostaną zabezpieczone za pomocą tego linuksowego identyfikatora. Na tej liście wyszczególniono również poszczególne składniki pakietu. Przykładowymi składnikami są aktywności, usługi oraz odbiorcy komunikatów.

## Usuwanie pakietu za pomocą aplikacji Package Browser

Skoro zajmujemy się tematyką przeglądarki pakietów, należałoby zauważyc, że możemy również usunąć pakiet z emulatora. W tym celu:

1. Zaznacz pakiet.
2. Wciśnij przycisk *Menu*.
3. Wybierz opcję *Delete package*, aby usunąć pakiet.

Wspomnialiśmy wcześniej, że proces jest związany z nazwą pakietu, z kolei nazwa pakietu jest uzależniona od podpisu cyfrowego. Podpis cyfrowy jest elementem mechanizmu zabezpieczającego dane należące do pakietu. Aby w pełni zrozumieć implikacje tej zależności, prześledźmy naturę procesu podpisywania aplikacji.

## Jeszcze raz o procesie podpisywania pakietów

W rozdziale 10. Czytelnik miał okazję zapoznać się z mechanizmem podpisywania aplikacji. Podkreśliśmy, że jest to wymagana czynność przed zainstalowaniem aplikacji na urządzeniu. Jednak nie wyjaśniliśmy, dlaczego podpiswanie aplikacji jest konieczne oraz jakie są tego implikacje.

Nie musimy na przykład podpisywać aplikacji podczas jej pobierania i instalowania w systemie Windows lub jakimś innym. Dlaczego więc podpiswanie programów jest w przypadku Androida czynnością obligatoryjną? Co ten proces tak naprawdę oznacza? Co otrzymujemy dzięki niemu? Czy istnieją jakieś analogiczne przykłady, do których możemy go szybko porównać? W niniejszym podrozdziale odpowiademy na powyższe pytania.

W trakcie instalowania pakietów w urządzeniu niezbędne jest zapewnienie unikatowej lub oddzielnej nazwy środowiska Java każdemu nowemu pakietowi. Jeżeli spróbujemy wdrożyć aplikację o takiej samej nazwie co już jakaś zainstalowana, urządzenie nie pozwoli na to, dopóki nie usuniemy tego pierwszego pakietu. Jeżeli chcemy umożliwić aktualizowanie aplikacji, nowy pakiet musi być powiązany z tym samym wydawcą programów co stary. Do tego celu służą właśnie podpisy cyfrowe. W czasie czytania następnych podrozdziałów Czytelnik przekona się, że za pomocą procesu podpisywania gwarantujemy sobie zarezerwowanie nazwy pakietu.

Przyjrzyjmy się kilku scenariuszom, dzięki którym w pełni zrozumiemy i przyswoimy sobie wiedzę na temat podpisów cyfrowych.

## Zrozumienie koncepcji podpisów cyfrowych — scenariusz 1.

Wyobraźmy sobie, że jesteśmy kolekcjonerami win mieszkającymi w bardzo nieprzyjaznym dla wina miejscu, powiedzmy, że na Saharze. Co więcej, winiarze z całego świata przesyłają nam na przechowanie lub na sprzedaż beczułki z winem.

Jako profesjonalni kolekcjonerzy mamy pełną świadomość, że każda beczułka zawiera wino posiadające specyficzny bukiet oraz barwę, niepowtarzalne wśród pozostałych rodzajów win. Po dokładniejszym sprawdzeniu stwierdzimy, że jeśli nawet w dwóch beczułkach znajdą się wina, które mają taki sam bukiet, to zawsze pochodzą od tego samego producenta. Przyglądając się tej sprawie jeszcze uważniej, dowiemy się, że każdy winiarz posiada sekretny przepis na swój niepowtarzalny bukiet, pilnie strzeżony i nieuwajniany nikomu. W ten sposób staje się zrozumiałe, dlaczego każde wino jest inne oraz dlaczego wina o tym samym bukciecie *muszą* pochodzić od tego samego producenta. Oczywiście, tego rodzaju identyfikacja nie zdradza tożsamości winiarza — upewnia nas jedynie, że jest on jedyny w swoim rodzaju.

Bukiet staje się sygnaturą tego winiarza, podobnie jak pieczęć rodowa, a receptura pozwalająca na jego otrzymanie staje się najskrzetniej skrywaną tajemnicą.

Ważnym spostrzeżeniem jest fakt, że jako kolekcjonerzy nie mamy możliwości dowiedzieć się, który winiarz wysłał nam daną dostawę wina — do sygnatury nie dołącza się nazwiska ani adresu. A nawet jeśli takie dane byłyby dołączone, całkiem możliwe, że w rzeczywistości winiarz mógł je wysłać z innego miejsca. W takim przypadku możemy założyć, że dwie beczułki wina przysłane z tego samego adresu, lecz zawierające wina o różnym bukciecie, pochodzą od dwóch różnych winiarzy przebywających w tym samym miejscu.

## Zrozumienie koncepcji podpisów cyfrowych — scenariusz 2.

Zastanówmy się nad innym, bardziej praktycznym przykładem. Gdy przebywamy za granicą i włączymy radio, usłyszmy wiele piosenek. Usłyszmy wielu wokalistów i będziemy mogli ich rozróżnić, nie będziemy jednak nawet znali ich nazwisk. Mamy więc do czynienia z podpisaniem utworu — barwa głosu każdego człowieka jest niepowtarzalna. Jeśli ktoś poda nam nazwisko piosenkarza i powiążemy je z danym utworem, będzie to analogiczne do podpisywania przez niezależnego wydawcę.

Jeden wokalista może imitować drugiego, odpowiednio modulując głos, aby zaciekać lub oszukać słuchacza. W przypadku podpisów cyfrowych takie oszustwo jest o wiele trudniejsze do prowadzenia z powodu algorytmów matematycznych stosowanych do szyfrowania podpisu.

## Wydanie koncepcji podpisów cyfrowych

Gdy mówimy o podpisywaniu pliku JAR, uzyskuje on osobny „bukiet” i staje się rozróżnialny w zbiorze innych plików JAR. Jednak nie ma stu procentowej możliwości zidentyfikowania programisty lub firmy. Są to tak zwane samopodpisane pliki JAR.

Aby znać źródło — miejsce pochodzenia wina, kolekcjoner musi uzyskać informacje od kogoś zaufanego, że dany bukiet pochodzi od Firma1. Jeśli teraz kolekcjoner zobaczy wino o takim bukciecie, będzie wiedział, że jego producentem jest Firma1. Mamy tu do czynienia z plikami JAR podpisanymi przez trzecią stronę. Dane te są wykorzystywane przez przeglądarki, kiedy wyświetlają informacje, że pobieramy plik od Firma1 lub instalujemy aplikację napisaną przez tę firmę (autorytatywnie).

## Jak zatem tworzymy cyfrowy podpis

Podpisy cyfrowe, wykorzystujące semantykę opisaną w powyższych scenariuszach, są implementowane za pomocą tak zwanego szyfrowania parą kluczy: publicznym i prywatnym. Stosowane są tutaj algorytmy matematyczne, w wyniku których powstają dwie liczby. Jedna z nich służy do szyfrowania podpisu (klucz prywatny), a jedynie za pomocą drugiej da się tak zaszyfrowany plik (wiadomość) rozkodować (klucz publiczny). Są to klucze asymetryczne. Nawet jeśli wszyscy znają klucz publiczny, nie ma możliwości zaszyfrowania pliku za jego pomocą. Można tego dokonać jedynie za pomocą klucza prywatnego, powiązanego z tym kluczem publicznym.

Rozważmy koncepcję kluczy publicznych i prywatnych na naszym przykładzie z winami.

Winiarz, który pragnie rozpoznawać wina nie za pomocą bukietów, lecz podpisów cyfrowych, za pomocą klucza prywatnego generuje kod (bukiet) przeznaczony dla określonej beczułki. Ponieważ do utworzenia tego kodu został użyty klucz prywatny, można go rozszyfrować jedynie za pomocą klucza publicznego.

Producent wina odważnie teraz zapisuje na beczułce nazwę klucza publicznego oraz zaszyfrowany kod, ewentualnie powierza klucz publiczny kurierowi.

Teraz my, jako kolekcjonerzy win, po odebraniu tego klucza i skutecznym rozszyfrowaniu kodu wiemy już, że klucz publiczny jest poprawny i jedynie producent tego wina mógł go podpisać. Jeżeli w tym scenariuszu jakiś oszust skopiuje taki klucz publiczny i umieści go na beczce ze swoim winem, nie będzie mógł napisać ukrytej wiadomości odblokowywanej przez ten klucz.

Klucz publiczny staje się w istocie podpisem winiarza. Nawet jeśli ktoś inny uzyska do niego dostęp, nie będzie mógł zaszyfrować za jego pomocą żadnych informacji.

Dzięki porównaniu sygnatur stosowanych w fizycznym świecie z podpisami cyfrowymi łatwiej nam teraz będzie uchwycić i zrozumieć koncepcję podpisywania plików. Przypomnijmy jeszcze, że w rozdziale 10. omówiliśmy już technikę stosowania poleceń keytool i jarsigner w procesie podpisywania pliku aplikacji.

## Implikacje wynikające z podpisywania plików

Teraz już rozumiemy, dlaczego nie możemy posiadać dwóch różniących się podpisów dla jednej nazwy pakietu. Podpisy tego typu są czasami nazywane certyfikatami infrastruktury klucza publicznego (ang. *Public Key Infrastructure* —PKI). Ścisłe rzecz biorąc, za pomocą certyfikatu PKI podpisujemy pakiety, pliki JAR, biblioteki DLL lub aplikacje.

Certyfikat PKI jest powiązany z nazwą pakietu w taki sposób, że instalacja dwóch pakietów posiadających taką samą nazwę, wydanych przez różnych producentów, jest niemożliwa. Jednak można używać tego samego certyfikatu w przypadku wielu osobnych pakietów. Inaczej mówiąc, *jeden* certyfikat PKI obsługuje *wiele* pakietów. Jest to relacja typu „*jeden do wielu*”. Jednak *po-jedynczy* pakiet uzyskuje *tylko i wyłącznie jeden* certyfikat za pośrednictwem infrastruktury PKI. Twórca zabezpiecza następnie klucz prywatny za pomocą hasła.

Powyższe informacje są bardzo ważne nie tylko ze względu na możliwość aktualizowania aplikacji, lecz również dla procesu współdzielenia danych pomiędzy pakietami, które są podpisane za pomocą tej samej sygnatury.

## Współdzielanie danych pomiędzy pakietami

W poprzednich rozdziałach ustaliliśmy, że każdy pakiet jest uruchomiony w osobnym procesie. Wszystkie dodatkowe składniki, które zostały zainstalowane lub utworzone w pakiecie, należą do użytkownika, którego identyfikator jest przydzielony do tego pakietu. Wiemy także, że Android przydziela unikatowy, linuksowy identyfikator, pozwalający na uruchomienie tego pakietu. Na rysunku 12.1 widzimy konstrukcję takiego identyfikatora. Zgodnie z dokumentacją pakietu SDK:

*Ten identyfikator użytkownika zostaje przypisany w momencie instalowania aplikacji i pozostaje niezmienny przez cały okres istnienia aplikacji w urządzeniu. Wszelkie dane zachowane przez aplikację otrzymają taki identyfikator użytkownika, a nie identyfikator dostępny dla pozostałych pakietów. Podczas tworzenia nowego pliku za pomocą metod getSharedPreferences(String, int), openFileOutput(String, int) lub openOrCreateDatabase(String, int, SQLiteDatabase.CursorFactory) możemy wykorzystać flagi MODE\_WORLD\_READABLE oraz (lub) MODE\_WORLD\_WRITEABLE do umożliwienia odczytywania lub zapisywania pliku przez inne pakiety. Po ustanowieniu tych flag plik ciągle należy do aplikacji, jednak uprawnienia globalnego odczytu lub zapisu zostały właściwie wprowadzone, więc każda inna aplikacja może z nich korzystać.*

Jeżeli mamy zamiar umożliwić współpracę pomiędzy aplikacjami korzystającymi ze wspólnego zestawu danych, możemy w sposób jawny zdefiniować unikatowy identyfikator użytkownika, który będzie spełniał nasze wymagania. Taki współdzielony identyfikator należy zdefiniować w pliku manifeście. Przypomina on nieco definicję nazwy pakietu. Przykład został ukazany na listingu 12.1.

**Listing 12.1.** Deklarowanie współdzielonego identyfikatora użytkownika

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.jakispakiet"
    sharedUserId="com.androidbook.mojwspoldzielonyduzytktownika"
    ...
    >
    ...pozostale węzły xml
</manifest>
```

## Natura współdzielonych identyfikatorów użytkownika

Wiele aplikacji może posiadać ten sam współdzielony identyfikator użytkownika, pod warunkiem że zawierają one tę samą sygnaturę (podpisana za pomocą certyfikatu PKI). Posiadanie takiego identyfikatora pozwala aplikacjom na współdzielanie danych, a nawet na działanie w obrębie tego samego procesu. Jeżeli nie chcemy zduplikować współdzielonego identyfikatora użytkownika, stosujmy konwencję nazewnictwa analogiczną do nazewnictwa klas Java. Poniżej prezentujemy dwa przykładowe współdzielone identyfikatory użytkownika spotykane w Androidzie:

```
android.uid.system
android.uid.phone
```

### Uwaga!

Natknęliśmy się na informację, że taki współdzielony identyfikator musi zostać zdefiniowany jako nieprzetworzony ciąg znaków, a nie jako zasób typu String.

Chcielibyśmy w tym miejscu ostrzec Czytelnika: jeżeli zechce korzystać ze współdzielonego identyfikatora, zalecamy jego używanie od samego początku procesu pisania aplikacji. W przeciwnym wypadku mogą się pojawiać problemy, jeśli aplikacja w wersji nieposiadającej identyfikatora zostanie zaktualizowana do wersji zawierającej ten identyfikator. Wynika to z tego, że w takim przypadku Android wykona polecenie chown na starych zasobach z powodu zmiany identyfikatora użytkownika. Dlatego usilnie zalecamy:

- W razie potrzeby należy stosować identyfikator użytkownika od samego początku budowy aplikacji.
- Nie należy zmieniać już ustanowionego identyfikatora użytkownika.

## Schemat kodu wykorzystywanego przy współdzieleniu danych

W tym podrozdziale zajmiemy się możliwościami płynącymi ze współdzielenia zasobów i danych pomiędzy dwiema aplikacjami. Jak wiemy, zasoby i dane umieszczone w każdym pakiecie są chronione i należą do kontekstu tego pakietu w trakcie jego działania. Nie powinno zatem nas zaskoczyć, że potrzebujemy dostępu do kontekstu zawierającego dane i zasoby, które chcemy współdzielić.

Do tego celu pomocny będzie interfejs API `createPackageContext()`. Możemy wykorzystać go wobec dowolnego istniejącego obiektu kontekstu (na przykład aktywności), aby uzyskać odniesienie do docelowego kontekstu, z którym chcemy nawiązać współpracę. Na listingu 12.2 został umieszczony przykładowy kod (jest jedynie poglądowy, nie jest przeznaczony do komplikacji).

**Listing 12.2.** Zastosowanie interfejsu `createPackageContext()`

```
//Identyfikuje pakiet, który chcemy wykorzystać
String targetPackageName="com.androidbook.samplepackage1";

//Określamy odpowiednią flagę kontekstu
int flag=Context.CONTEXT_RESTRICTED;

//Za pomocą jednej z aktywności pobieramy kontekst
Activity myContext = ....;
Context targetContext =
    myContext.createPackageContext(targetPackageName, flag);

//Wykorzystujemy kontekst do określenia ścieżek do plików
Resources res = targetContext.getResources();
File path = targetContext.getFilesDir();
```

Zwróćmy uwagę, w jaki sposób możemy uzyskać odniesienie do kontekstu danego pakietu, na przykład `com.androidbook.samplepackage1`. Obiekt `targetContext` widoczny na listingu 12.2 jest taki sam jak kontekst przekazywany docelowej aplikacji w momencie jej uruchomienia. Jak sama nazwa metody wskazuje (przedrostek „`create`”), każde wywołanie odsyła nowy obiekt kontekstu. Jednak w dokumentacji znalazło się zapewnienie, że mechanizm ten został zaprojektowany w taki sposób, aby jak najmniej obciążać system.

Interfejs ten jest dostępny bez względu na to, czy korzystamy ze współdzielonego interfejsu użytkownika, czy nie. Jeżeli stosujemy ten identyfikator, to bardzo dobrze. W przeciwnym wypadku w docelowej aplikacji musi się znaleźć deklaracja zasobów jako dostępnych dla zewnętrznych użytkowników.

Interfejs `createPackageContext()` wykorzystuje jedną z trzech następujących flag:

- W przypadku flagi `CONTEXT_INCLUDE_CODE` Android pozwala załadować kod docelowej aplikacji do bieżącego procesu. Kod ten będzie wtedy działał jak nasz własny. Działa to jedynie wtedy, gdy obydwa pakiety będą posiadały wspólną sygnaturę oraz identyfikator użytkownika. Jeżeli identyfikatory użytkownika będą się różniły, wprowadzenie tej flagi będzie skutkowało wystąpieniem wyjątku zabezpieczeń.
- Flaga `CONTEXT_RESTRICTED` oznacza, że wciąż istnieje możliwość uzyskania dostępu do ścieżek zasobów i nie zachodzi skrajny przypadek, jakim jest żądanie wczytania kodu.
- Dzięki flagie `CONTEXT_IGNORE_SECURITY` certyfikaty są ignorowane i kod zostanie wczytany, ale będzie działał pod naszym identyfikatorem użytkownika. W dokumentacji sugerowana jest wyjątkowa ostrożność podczas korzystania z tej flagi.

Wiemy już teraz, w jaki sposób pakiety, sygnatury i współdzielone identyfikatory użytkownika współpracują ze sobą w procesie kontrolowania dostępu do elementów przechowywanych i tworzonych przez aplikację.

## Projekty bibliotek

Podczas omawiania koncepcji współdzielenia kodu i zasobów nasuwa się jedno zasadnicze pytanie: czy pomocna okaże się idea projektu bibliotek? Aby się tego dowiedzieć, najpierw musimy zrozumieć, czym są te projekty, jak są tworzone oraz w jaki sposób są używane.

### Czym jest projekt bibliotek?

Począwszy od wersji 0.9.7 wtyczki ADT, Android posiada funkcję obsługi projektów bibliotek (ang. *library projects*). Projekt bibliotek stanowi zbiór kodów Java oraz zasobów przypominających architekturą zwyczajny projekt, ale nazwa pliku, w którym są zawarte, nigdy nie kończy się rozszerzeniem `.apk`. Zamiast tego zawartość tego pliku może zostać włączona do innego projektu oraz skompilowana w głównym pliku `.apk` aplikacji.

### Twierdzenia dotyczące projektów bibliotek

Poniżej prezentujemy pewne fakty związane z projektami bibliotek:

- Projekt bibliotek może posiadać własną nazwę pakietu.
- Projekt bibliotek nie zostanie skompilowany do osobnego pliku `.apk`, a jedynie może być włączony do pliku `.apk` projektu, który wykorzystuje dane zawarte w tym projekcie bibliotek.
- Projekt bibliotek może korzystać z innych plików JAR.
- Nie można przekształcić samego projektu bibliotek w plik JAR.
- Wtyczka ADT połączy projekt bibliotek z głównym projektem i skompiluje je jako część głównego projektu.

- Zarówno projekt bibliotek, jak i główny projekt mogą uzyskać dostęp do zasobów przechowywanych w tym pierwszym za pomocą odpowiednich plików *R.java*.
- Możemy posiadać zduplikowane identyfikatory zasobów pomiędzy projektem głównym a bibliotek. Identyfikatory zasobów w projekcie głównym mają wyższy priorytet.
- Jeżeli chcemy rozróżniać identyfikatory zasobów dwóch projektów, możemy wprowadzić odmienne przedrostki, na przykład *lib\_* dla zasobów projektu bibliotek.
- Główny projekt może odnosić się do dowolnej liczby projektów bibliotek.
- Możemy ustanowić pierwszeństwo projektów bibliotek, aby się przekonać, które zasoby są ważniejsze.
- Takie składniki projektu bibliotek jak aktywność należy zadeklarować w pliku manifeście głównego projektu. Nazwa składnika z pakietu bibliotek musi być całkowicie zgodna z nazwą pakietu bibliotek.
- Nie ma potrzeby definiowania składników w pliku manifeście projektu bibliotek, chociaż może się to okazać przydatne do szybkiego rozpoznawania obsługiwanych przez niego składników.
- Tworzenie projektu bibliotek rozpoczyna się od utworzenia standardowego projektu Androida oraz ustawienia flagi *Is Library* w oknie właściwości.
- Możemy również w oknie właściwości projektu powiązać główny projekt z projektami bibliotek.
- Do wielu różnych głównych projektów można dołączać projekty bibliotek.
- Funkcja projektów bibliotek została wprowadzona w wersji 0.9.7 narzędzi ADT, zestawie SDL w wersji 6 lub wyższej oraz od wersji 2.1 systemu Android.
- W aktualnej wersji środowiska jeden projekt bibliotek nie może odnosić się do innego projektu bibliotek, chociaż w przyszłych wersjach może się pojawić taka możliwość.
- Projekty bibliotek nie obsługują plików AIDL.
- Projekt bibliotek nie obsługuje katalogu ze współdzielonymi plikami dodatkowymi.

Przekonajmy się, do czego służą projekty bibliotek, poprzez utworzenie jednego z nich oraz projektu głównego. Poniżej prezentujemy cele naszego przykładowego projektu:

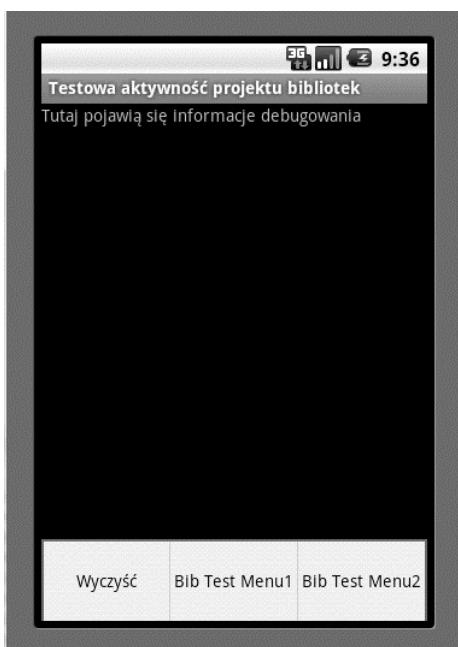
1. Utworzenie prostej aktywności w projekcie bibliotek.
2. Utworzenie menu dla aktywności z punktu 1. poprzez zdefiniowanie pewnych zasobów menu.
3. Utworzenie w głównym projekcie aktywności, która będzie korzystała z projektu bibliotek.
4. Utworzenie aktywności w głównym projekcie, wygenerowanym w punkcie 3.
5. Utworzenie menu dla głównej aktywności z punktu 4.
6. Przywołanie aktywności projektu bibliotek za pomocą elementu menu głównej aktywności.

Po utworzeniu obydwu projektów ujrzymy ekran głównej aktywności (punkt 4.), zaprezentowany na rysunku 12.2.

Po kliknięciu elementu *lib*, widocznego w aktywności głównego projektu, zostanie wyświetlona aktywność przedstawiona na rysunku 12.3, pochodząca z projektu bibliotek.



**Rysunek 12.2.** Przykładowa aktywność projektu głównego zawierająca elementy menu



**Rysunek 12.3.** Przykładowa aktywność projektu bibliotek

Menu widoczne w aktywności projektu bibliotek pochodzą z zasobów tego projektu. Efektem kliknięcia poszczególnych opcji menu jest wyświetlony na ekranie komunikat o kliknięciu danego elementu. Rozpocznijmy ćwiczenie od utworzenia projektu bibliotek.

## Utworzenie projektu bibliotek

Nasz projekt będzie się składał z następujących plików:

- *TestLibActivity.java* (listing 12.3),
- *layout/lib\_main.xml* (listing 12.4),
- *menu/lib\_main\_menu.xml* (listing 12.5),
- *AndroidManifest.xml* (listing 12.6).

Te pliki, ukazane na poniższych listingach, powinny całkowicie wystarczyć do utworzenia przykładowego projektu bibliotek.

### Uwaga!

Na końcu rozdziału zamieszczamy adres URL, pod którym są dostępne wszystkie omawiane tu projekty. W ten sposób możemy je zimportować bezpośrednio do środowiska Eclipse.

### **Listing 12.3.** Przykładowa aktywność projektu bibliotek — plik TestLibActivity.java

```
package com.androidbook.library.testlibrary;

//...miejsce na podstawowe instrukcje importu
//po wcisnięciu kombinacji CTRL+SHIFT+O środowisko Eclipse
//wygeneruje niezbędne instrukcje importu

public class TestLibActivity extends Activity
{
    public static final String tag="TestLibActivity";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.lib_main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater(); //z aktywności
        inflater.inflate(R.menu.lib_main_menu, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        appendMenuItemText(item);
        if (item.getItemId() == R.id.menu_clear){
            this.emptyText();
            return true;
        }
        return true;
    }
    private TextView getTextView(){
```

```
        return (TextView)this.findViewById(R.id.text1);
    }
    public void appendText(String abc){
        TextView tv = getTextView();
        tv.setText(tv.getText() + "\n" + abc);
    }
    private void appendMenuItemText(MenuItem menuItem){
        String title = menuItem.getTitle().toString();
        TextView tv = getTextView();
        tv.setText(tv.getText() + "\n" + title);
    }
    private void emptyText(){
        TextView tv = getTextView();
        tv.setText("");
    }
}
```

---

Na listingu 12.4 widzimy plik układu graficznego obsługujący powyższą aktywność — prosty widok tekstowy stosowany do wyświetlenia nazwy klikniętego elementu.

**Listing 12.4.** Przykładowy plik układu graficznego w projekcie bibliotek — plik layout/lib\_main.xml

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Tutaj pojawią się informacje debugowania. "
    />
</LinearLayout>
```

---

Listing 12.5 prezentuje nam zawartość pliku opcji menu, które są widoczne w aktywności projektu bibliotek na rysunku 12.3.

**Listing 12.5.** Plik menu projektu bibliotek — plik menu/lib\_main\_menu.xml

---

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_clear"
            android:title="wyczyść" />
        <item android:id="@+id/menu_testlib_1"
            android:title="Bib Test Menu1" />
        <item android:id="@+id/menu_testlib_2"
            android:title="Bib Test Menu2" />
    </group>
</menu>
```

---

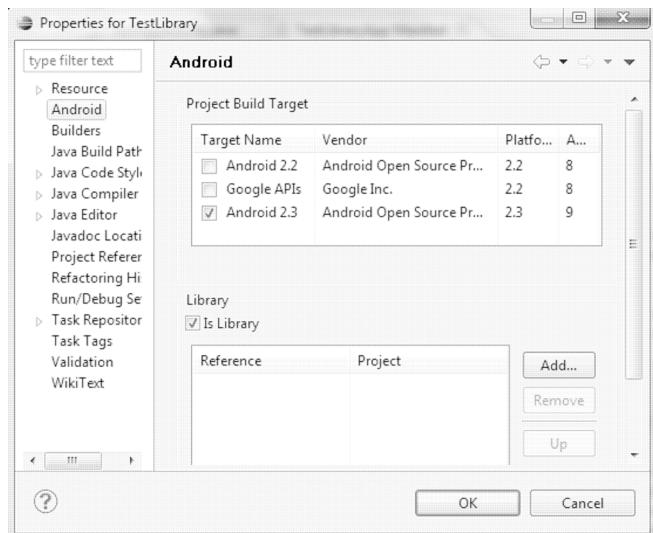
Natomiast plik manifest projektu bibliotek został zaprezentowany na listingu 12.6.

#### **Listing 12.6.** Plik manifest projektu bibliotek — AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.library.testlibrary"
    android:versionCode="1"
    android:versionName="1.0.0">
    <uses-sdk android:minSdkVersion="3" />
    <application android:icon="@drawable/icon"
        android:label="Testowy projekt bibliotek">
        <activity android:name=".TestLibActivity"
            android:label="Testowa aktywność projektu bibliotek">
        </activity>
    </application>
</manifest>
```

Jak już stwierdziliśmy w punkcie, w którym zaprezentowaliśmy twierdzenia dotyczące projektów bibliotek, definicja aktywności w pliku manifeście projektu bibliotek pojawia się wyłącznie w celach dokumentacyjnych oraz jest opcjonalna pod kątem mechanizmów działania kodu.

Po zapoznaniu się z plikami możemy utworzyć standardowy projekt Androida. Po skonfigurowaniu projektu klikamy prawym przyciskiem myszy nazwę projektu, a następnie menu kontekstowe właściwości, dzięki czemu pojawi się okno dialogowe właściwości, w którym możemy ustanowić projekt bibliotek. Omawiane okno dialogowe widzimy na rysunku 12.4 (w zależności od wersji zestawu SDK widoczne na rysunku wersje Androida mogą być inne). Wystarczy zaznaczyć opcję *Is Library*, aby przetworzyć bieżący projekt w projekt bibliotek.



**Rysunek 12.4.** Ustanawianie projektu bibliotek

W ten sposób zakończyliśmy proces tworzenia projektu bibliotek. Dowiemy się teraz, w jaki sposób utworzyć projekt aplikacji wykorzystujący wygenerowany przed chwilą projekt bibliotek.

## Tworzenie projektu testowego wykorzystującego projekt bibliotek

Do zbudowania aplikacji wykorzystamy podobny zestaw plików, a następnie dołączymy napisany powyżej projekt bibliotek. Wygenerujemy teraz następujące pliki:

- *TestAppActivity.java* (listing 12.7),
- *layout/main.xml* (listing 12.8),
- *menu/main\_menu.xml* (listing 12.9),
- *AndroidManifest.xml* (listing 12.10).

Na listingu 12.7 pokazaliśmy plik *TestAppActivity.java*.

**Listing 12.7.** Kod aktywności głównego projektu — plik *TestAppActivity.java*

---

```
package com.androidbook.library.testlibraryapp;
import com.androidbook.library.testlibrary.*;
//...inne instrukcje importu

public class TestAppActivity extends Activity
{
    public static final String tag="TestAppActivity";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater(); //z aktywności
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        appendMenuItemText(item);
        if (item.getItemId() == R.id.menu_clear)
        {
            this.emptyText();
            return true;
        }
        if (item.getItemId() == R.id.menu_library_activity){
            this.invokeLibActivity(item.getItemId());
            return true;
        }
        return true;
    }
    private void invokeLibActivity(int mid)
    {
        Intent intent = new Intent(this,TestLibActivity.class);
        intent.putExtra("com.ai.menuid", mid);
        startActivity(intent);
    }
}
```

---

```

private TextView getTextView(){
    return (TextView)this.findViewById(R.id.text1);
}
public void appendText(String abc){
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + abc);
}
private void appendMenuItemText(MenuItem menuItem){
    String title = menuItem.getTitle().toString();
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + title);
}
private void emptyText(){
    TextView tv = getTextView();
    tv.setText("");
}
}

```

---

Zwróćmy uwagę, że po utworzeniu tego pliku może się pojawić błąd komplikacji związany z odniesieniem do aktywności umieszczonej w projekcie bibliotek. Nie pozbędziemy się go, dopóki nie przeczytamy dalszej części rozdziału i nie odkryjemy, w jaki sposób zdefiniować projekt bibliotek jako obiekt zależny od głównego projektu.

Kod pliku układu graficznego obsługującego powyższą aktywność jest widoczny na listingu 12.8.

#### **Listing 12.8.** Układ graficzny głównego projektu — plik layout/main.xml

---

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Tutaj pojawią się informacje debugowania"
    />
</LinearLayout>

```

---

Kod Java występujący w aktywności głównego projektu (listing 12.7) wykorzystuje element menu *R.id.menu\_library\_activity* do wywołania aktywności *TestLibActivity*. Poniżej przedstawiliśmy fragment kodu z pliku Java (pokazanego wcześniej na listingu 12.7):

```

private void invokeLibActivity(int mid)
{
    Intent intent = new Intent(this,TestLibActivity.class);
    //Przekazuje identyfikator menu w postaci dodatkowej intencji
    //na wypadek, gdyby wymagała tego aktywność bibliotek.
    intent.putExtra("com.androidbook.library.menuid", mid);
    startActivityForResult(intent);
}

```

Zwróćmy uwagę, że klasa `TestLibActivity.class` jest wykorzystywana lokalnie, mimo że importowaliśmy klasy Java z pakietu bibliotek:

```
import com.androidbook.library.testlibrary.*;
```

Z kolei na listingu 12.9 widzimy kod menu.

---

**Listing 12.9.** Plik menu głównego projektu — `menu/main_menu.xml`

---

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_clear"
            android:title="wyczyść" />
        <item android:id="@+id/menu_library_activity"
            android:title="invoke lib" />
    </group>
</menu>
```

---

Do zakończenia procesu tworzenia projektu potrzebujemy jeszcze pliku manifestu, zamieszczonego na listingu 12.10.

---

**Listing 12.10.** Plik manifest głównego projektu — `AndroidManifest.xml`

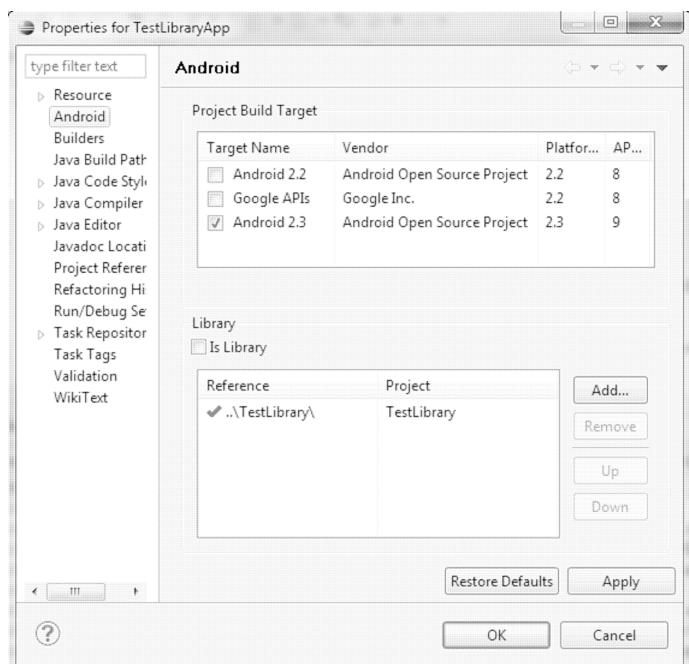
---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.library.testlibraryapp"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="Aplikacja
    testująca bibl.">
        <activity android:name=".TestAppActivity"
            android:label="Aplikacja testująca bibl.">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=
"com.androidbook.library.testlibrary.TestLibActivity"
            android:label="Aktywność testująca bibl."/>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

---

Zwróćmy uwagę, w jaki sposób w tym pliku manifeście głównej aplikacji zdefiniowano aktywność `TestLibActivity`, pochodząą z projektu bibliotek. Podczas definiowania aktywności zastosowaliśmy również pełną nazwę pakietu. Zauważmy też, że nazwy pakietu dla projektu bibliotek mogą się różnić od tych zawartych w głównej aplikacji.

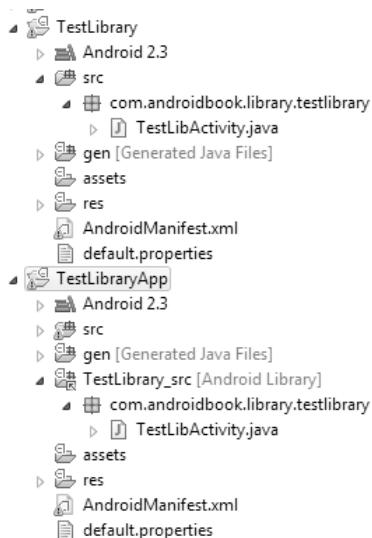
Po zapełnieniu projektu tymi plikami należy otworzyć okno dialogowe właściwości projektu (rysunek 12.5), aby zaznaczyć, że nasz główny projekt zależy od utworzonego wcześniej projektu bibliotek.



Rysunek 12.5. Deklarowanie zależności aplikacji od projektu bibliotek

Widzimy w tym oknie dialogowym przycisk *Add*. Możemy za jego pomocą dodać odniesienie do projektu bibliotek, pokazanego na rysunku 12.5. Inne czynności są niepotrzebne.

Po dodaniu projektu bibliotek ukazuje się on zazwyczaj w postaci dodatkowego węzła drzewa głównej aplikacji (a także pozostaje jednocześnie osobnym projektem bibliotek). Ilustruje to rysunek 12.6.

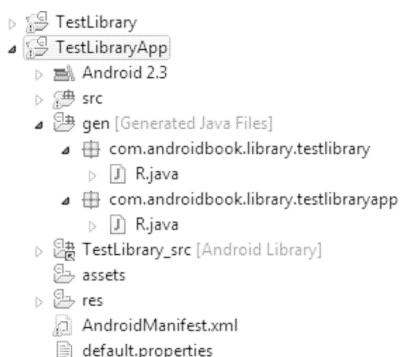


Rysunek 12.6. Dołączony projekt bibliotek w widoku głównego projektu

Zwróćmy uwagę na węzeł z dopiskiem *[Android Library]* oraz na powielone (przekierowane) pliki źródłowe Java. Przyjrzyjmy się także strukturze tego węzła. Jego nazwa powstaje poprzez połączenie nazwy projektu bibliotek, podkreśnika oraz nazwy powiązanego katalogu źródłowego umieszczonego w projekcie bibliotek. Taki schemat nazewnictwa pozwala nam na umieszczanie dowolnej liczby własnych katalogów źródłowych w projekcie bibliotek. Jest to główna różnica pomiędzy wersją 0.9.8 narzędzia ADT a nowszymi.

Jeżeli zamierzamy w głównym projekcie modyfikować pliki źródłowe należące do projektu bibliotek, będą one zmieniane również w samym projekcie bibliotek. Czasami nie widać tego węzła. W takim wypadku warto uruchomić ponownie środowisko Eclipse. W każdym razie, jeśli widzimy ten węzeł, oznacza to, że nasz projekt działa.

Wspomnijmy jeszcze, że Android traktuje pliki *R.java* w ciekawy sposób. Spójrzmy na rysunek 12.7.



**Rysunek 12.7.** Powielone zasoby w pliku *R.java*

Najpierw zostaje wygenerowany jeden plik *R.java* w projekcie bibliotek, odpowiedzialny za zasoby przechowywane w tym projekcie. Ponadto system tworzy również plik *R.java* przechowujący informacje o zasobach znajdujących się w głównym projekcie. Można się było tego spodziewać — dwóm projektom odpowiadają dwa pliki *R.java*.

Jednak, co ciekawe, Android generuje identyfikatory zasobów mieszczących się w projekcie bibliotek również w pliku *R.java* głównej aplikacji. Oznacza to, że programista może stosować składnię *R.id.* związaną z identyfikatorami znajdującymi się w pliku *R.java*, który stanowi część głównej aplikacji (nie zapominajmy, że plik *R.java* zostaje automatycznie wygenerowany, więc wartości z listingu 12.11, takie jak 0x7f02000, mogą być zupełnie inne).

**Listing 12.11.** Ponownie zdefiniowane identyfikatory współdzielonych zasobów, przechowywane w pliku *R.java* głównego projektu

---

```
public final class R {  
    public static final class attr {  
    }  
    public static final class drawable {  
        public static final int icon=0x7f020000;  
        public static final int robot=0x7f020001;  
    }  
    public static final class id {  
        public static final int menuGroup_Main=0x7f060001;
```

```

    public static final int menu_clear=0x7f060002;
    public static final int menu_library_activity=0x7f060005;
    public static final int menu_testlib_1=0x7f060003;
    public static final int menu_testlib_2=0x7f060004;
    public static final int text1=0x7f060000;
}
public static final class layout {
    public static final int lib_main=0x7f030000;
    public static final int main=0x7f030001;
}
public static final class menu {
    public static final int lib_main_menu=0x7f050000;
    public static final int main_menu=0x7f050001;
}
public static final class string {
    public static final int app_name=0x7f040001;
    public static final int hello=0x7f040000;
}
}

```

---

Zauważmy, że w pliku *R.java* głównej aplikacji zostały zdefiniowane również zasoby rozpoczęjące się od przedrostka *lib\_*. Oznacza to, że projekt bibliotek będzie posiadał swoje stałe dla zasobów *lib\_*, a główny projekt będzie posiadał inne stałe dla tych samych zasobów.

Obydwa projekty mogą się odnosić do tego samego zasobu za pomocą składni *R.jakis -> identyfikator*. Wartość tej stałej może być identyczna, jednak identyfikator tego zasobu będzie dostępny w obydwu przestrzeniach nazw Java: w przestrzeni nazw projektu bibliotek oraz w przestrzeni nazw głównej aplikacji.

Uważajmy także na nazwy kontrolek menu: *lib\_main\_menu* oraz *main\_menu*. Mogłyby się to okazać kłopotliwe, gdyby w aplikacji znalazły się dwa menu wypełnione różnymi elementami, posiadające jednak taką samą nazwę zasobu. Reasumując, zasoby zostają zebrane i udostępnione w jednym miejscu przeznaczonym dla głównej aplikacji. Musimy być szczególnie ostrożni w przypadku zasobów zlokalizowanych na poziomie pliku, na przykład kontrolki menu lub układów graficznych, a także identyfikatorów generowanych z tych obiektów dla wewnętrznych elementów.

Skoro wiemy, czym są projekty bibliotek, czy potrafimy już odpowiedzieć na jakiekolwiek wcześniej postawione pytanie na temat współdzielonych danych?

Jak widać, projekty bibliotek są konstruktami czasu komplikacji. Jasne staje się, że wszelkie zasoby należące do tego projektu zostają wchłonięte i przyłączone do głównego projektu. Nie możemy zadać pytania o współdzielenie w czasie działania aplikacji, ponieważ istnieje tylko jeden plik pakietu zawierający nazwę głównego pakietu. Jedną z często wymienianych propozycji jest możliwość utworzenia wersji darmowych i płatnych aplikacji, współdzielących jeden projekt bibliotek.

## Odbośni

Dzięki poniższym odnośnikom Czytelnik łatwiej zrozumie koncepcje zawarte w tym rozdziale:

- <http://developer.android.com/guide/publishing/app-signing.html> — bardzo przydatne informacje na temat podpisywania plików *.apk*.

- <http://java.sun.com/j2se/1.3/docs/tooldocs/win32/keytool.html> — znakomita dokumentacja dotycząca narzędzi keytool, jarsigner oraz samego procesu podpisywania.
- <http://www.androidbook.com/item/3493> — notatki autorów, włącznie z modelem pojęciowym, wyjaśniającym znaczenie podpisywania plików JAR.
- <http://www.androidbook.com/item/3279> — na tej stronie zebraliśmy wszystkie dane badawcze związane z pakietami Androida. Zawarte są tutaj informacje na temat podpisywania plików .apk, odnośniki do artykułów opisujących proces współdzielienia danych pomiędzy pakietami, dalsze wiadomości na temat współdzielonych identyfikatorów użytkownika, a także instrukcje instalowania oraz odinstalowywania pakietów.
- <http://developer.android.com/guide/developing/projects/projects-eclipse.html> — znajdziemy tu między innymi informacje dotyczące projektów bibliotek.
- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — znajdziemy tu pełen zestaw projektów do pobrania, opracowanych na podstawie informacji zawartych w książce. Projekty związane z tym rozdziałem zostały umieszczone w katalogu *ProAndroid3\_R12\_Biblioteki*.

## Podsumowanie

Rozdział ten poświęciliśmy zagadnieniom dotyczącym pracy z pakietami i procesami, współdzielenia kodu i danych pomiędzy pakietami, a także tworzenia projektów bibliotek Androida. Dowiedzieliśmy się, że proces podpisywania odgrywa istotną rolę w zabezpieczeniach, zwłaszcza na etapie przydzielania uprawnień pakietom.

Niniejszy rozdział stanowi wprowadzenie do następnego, w którym przeanalizujemy składniki przechowywane w procesie pakietu oraz (przede wszystkim) przebiegające w jego głównym wątku. Dowiemy się, w jaki sposób można optymalnie dostosować główny wątek za pomocą procedur obsługi oraz wątków podrzędnych, co pozwala na zapewnienie płynnego działania aplikacji.

# Analiza procedur obsługi

W rozdziale 12. stwierdziliśmy, że każdy pakiet jest przetwarzany w osobnym procesie. Teraz zajmiemy się organizacją wątków we wnętrzu procesu. W ten sposób odpowiemy sobie na pytanie, do czego są nam potrzebne procedury obsługi.

Większa część kodu aplikacji w Androidzie jest przetwarzana w kontekście takiego składnika, jak aktywność lub usługa. Zastanowimy się, w jaki sposób te składniki aplikacji oddziałują z wątkami. Przez większość czasu działania aplikacji uruchomiony jest tylko jeden wątek wewnętrzny procesu, znany jako wątek główny. Wyjaśnimy skutki współdzielenia takiego głównego wątku przez różne składniki. Przede wszystkim prowadzi to do wyświetlanego komunikatu ANR (ang. *Application Not Responding* — aplikacja nie odpowiada; podkreślamy tylko, że „A” jest skrótem od „aplikacja”, a nie od „annoying”, czyli „irytująca”). Pokażemy, w jaki sposób możemy stosować procedury obsługi, komunikaty oraz wątki do uniezależnienia się od głównego wątku w przypadku konieczności uruchomienia operacji trwających dłuższy czas.

Rozpoczniemy ten rozdział od przyjrzenia się składnikom aplikacji oraz kontekstowi wątku, w którym są one przetwarzane.

## Składniki Androida i wątkowanie

Po przeczytaniu wcześniejszych rozdziałów mogliśmy zdążyć już wywnioskować, że proces w Androidzie zawiera cztery podstawowe elementy. Są to klasy:

- **Activity** (czyli aktywność),
- **Service** (usługa),
- **ContentProvider** (często zwana po prostu dostawcą),
- **BroadcastReceiver** (w skrócie odbiorca).

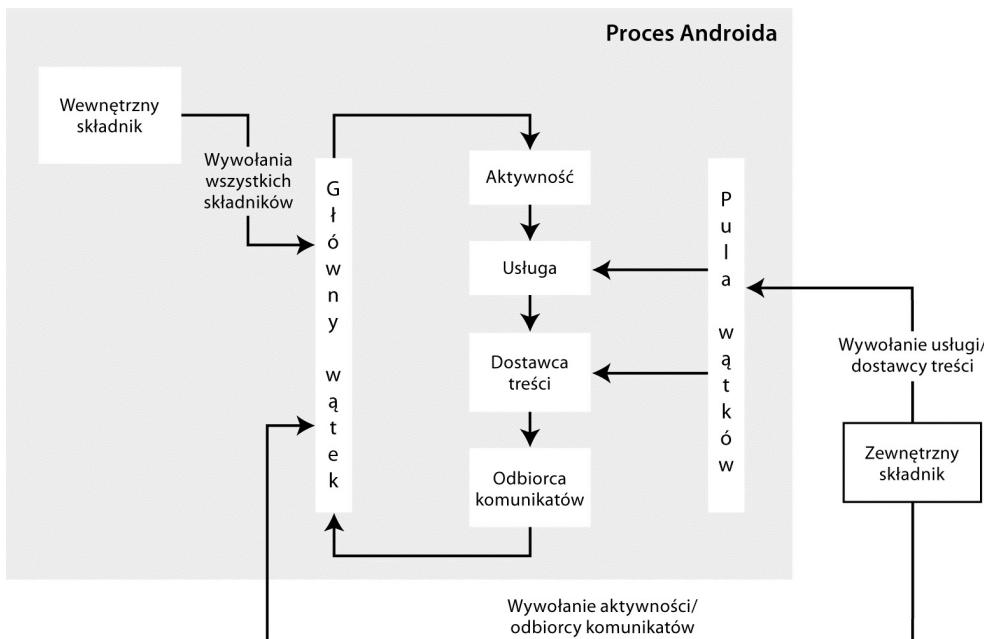
Większość kodu tworzonej aplikacji jest częścią jednego z wymienionych składników lub jest wywoływana przez jeden z nich. Każdy komponent tego typu posiada własny znacznik XML w specyfikacji aplikacji, zdefiniowanej w pliku manifeście projektu. Poniżej przypominamy, jak wyglądają te węzły:

```
<application>
    <activity/>
```

```
<service/>
<receiver/>
<provider/>
</application>
```

Nie licząc pewnych wyjątków, Android wykorzystuje ten sam wątek do przetwarzania (lub przechowywania) kodu z tych składników. Mamy tu do czynienia z głównym wątkiem aplikacji. Wywołanie tych składników może być synchroniczne, na przykład w przypadku żądania danych od dostawcy treści, lub opóźnione za pomocą kolejkowania wiadomości, jak choćby podczas przywoływanego funkcji za pomocą uruchomienia usługi.

Na rysunku 13.1 przedstawiono związki pomiędzy wątkami a tymi czterema składnikami. Celem tego diagramu jest ukazanie, w jaki sposób wątki krążą po strukturze Androida oraz jej składnikach. Zagadnienia z tym związane omówiliśmy w kolejnych kilku podrozdziałach.



Rysunek 13.1. Struktura składników i wątkowania w Androidzie

## Aktywności działają w głównym wątku

Jak widzimy na rysunku 13.1, wszystkie operacje są wykonywane w głównym wątku. Główny wątek oddziałuje ze wszystkimi składnikami. Należy dodać, że dzieje się to poprzez kolejkowanie wiadomości. Na przykład podczas zaznaczania elementów menu lub przycisków na ekranie urządzenia przez użytkownika działania te zostaną przekształcone w komunikaty i przekazane do głównego wątku aktywnego procesu. Główny wątek jest zapętlony i przetwarza każdy nadesłany komunikat. Jeżeli przetwarzanie jakiegoś komunikatu trwa dłużej niż 5 sekund, zostanie wyświetlony komunikat ANR.

## Odbiorcy komunikatów działają w głównym wątku

W podobny sposób, jeżeli odpowiedzią na kliknięcie elementu menu ma być przywołanie dostawcy komunikatów, Android umieszcza komunikat w głównej kolejce procesu, z którego ma zostać wywołany zarejestrowany odbiorca. Główny wątek natrafi w pewnym momencie na ten komunikat i wywoła odbiorcę. Wątek ten może również obsłużyć odbiorcę komunikatów. Jeżeli główny wątek jest zajęty odpowiadaniem na działanie związane z menu, odbiorca komunikatów będzie musiał poczekać na zakończenie tej czynności.

## Usługi działają w głównym wątku

Taka sama zasada dotyczy usług. Jeśli po kliknięciu elementu menu nastąpi uruchomienie usługi lokalnej za pomocą metody `startService`, odpowiedni komunikat zostanie umieszczony w głównej kolejce, a główny wątek przetworzy go za pomocą kodu usługi.

## Dostawcy treści działają w głównym wątku

Wywołanie lokalnego dostawcy treści przebiega na nieco odmiennych zasadach. Dostawca treści również działa w obrębie głównego wątku, ale wywołanie tego dostawcy jest synchroniczne i nie zachodzi z wykorzystaniem kolejek wiadomości.

## Skutki posiadania pojedynczego głównego wątku

Być może Czytelnik zastanawia się, dlaczego poświęcamy tyle uwagi temu, czy większość kodu aplikacji działa w głównym wątku, czy też nie. Odpowiedź na to pytanie brzmi: ponieważ główny wątek ma powracać do kolejki wiadomości w celu zapewnienia ciągłości odpowiedzi na działania w interfejsie użytkownika. W konsekwencji nie należy przetrzymywać głównego wątku. Jeżeli wiemy, że jakaś czynność będzie trwała ponad pięć sekund, powinniśmy ją umieścić w osobnym wątku lub opóźnić jej wykonanie, programując powrót wątku głównego do niej po wykonaniu innych operacji. Okazuje się jednak, że przeprowadzanie operacji w oddzielnym wątku nie jest takie łatwe, jak by się mogło na początku wydawać. Wróćmy do tego zagadnienia w dalszej części rozdziału, a także w następnym rozdziale, teraz jednak przyjrzyjmy się puli wątków przedstawionej na rysunku 13.1.

## Pule wątków, dostawcy treści, składniki zewnętrznych usług

Kiedy zewnętrzne klienty lub składniki spoza procesu żądają danych od dostawcy treści, żądanie to jest przydzielane do wątku z puli wątków. To samo dotyczy zewnętrznych klientów łączących się z usługami.

## Narzędzia wątkowania — poznaj swój wątek

Po tym dość długim omówieniu tematyki wątków głównych i roboczych warto byłoby pokazać, w jaki sposób wykorzystać poniższą klasę, zaprezentowaną na listingu 13.1, służącą do określania, który wątek obsługuje dany fragment kodu. Następnie, zaglądając do okna `LogCat`, możemy porównać naszą wiedzę teoretyczną z rzeczywistością poprzez analizowanie wyświetlanych identyfikatorów wątków.

**Listing 13.1.** Narzędzia wątkowania

```
//utils.java
public class Utils
{
    public static long getThreadId() {
        Thread t = Thread.currentThread();
        return t.getId();
    }

    public static String getThreadSignature(){
        Thread t = Thread.currentThread();
        long l = t.getId();
        String name = t.getName();
        long p = t.getPriority();
        String gname = t.getThreadGroup().getName();
        return (name
            + ":(id)" + l
            + ":(priorytet)" + p
            + ":(grupa)" + gname);
    }

    public static void logThreadSignature(){
        Log.d("ThreadUtils", getThreadSignature());
    }

    public static void sleepForInSecs(int secs){
        try{
            Thread.sleep(secs * 1000);
        } catch(InterruptedException x){
            throw new RuntimeException("przerwano",x);
        }
    }

    //Poniższe dwie metody są używane przez wątki robocze,
    //które zostaną omówione później.
    public static Bundle getStringAsABundle(String message){
        Bundle b = new Bundle();
        b.putString("message", message);
        return b;
    }

    public static String getStringFromABundle(Bundle b){
        return b.getString("message");
    }
}
```

---

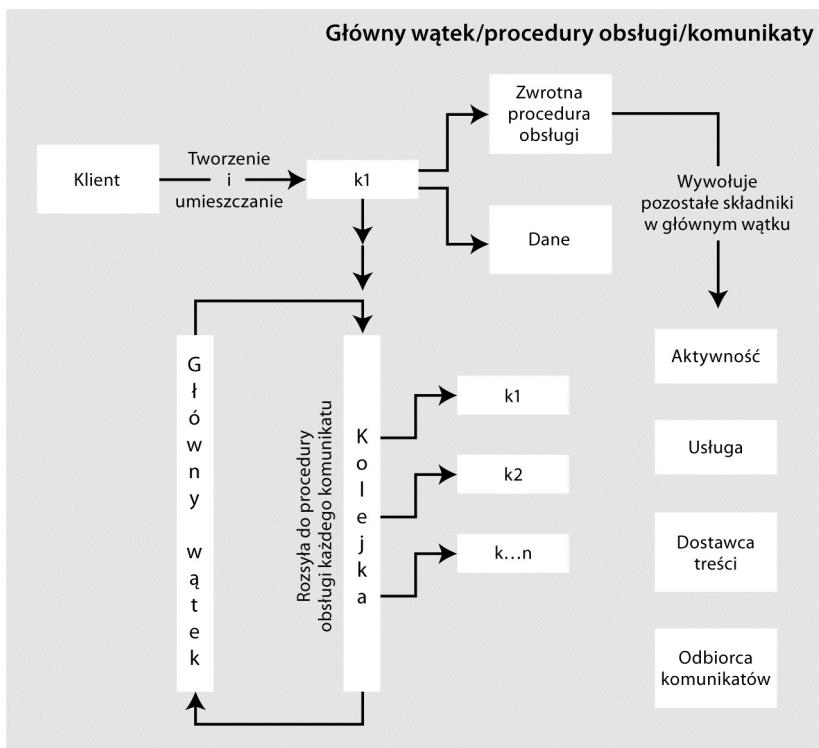
Dzięki wykorzystaniu metody `logThreadSignature()` dowiemy się, w którym wątku jest wykonywany kod. Możemy również sprawdzić, co się stanie po wstrzymaniu głównego wątku za pomocą metody `sleep()`, w wyniku czego przestaną on przetwarzać kolejkę komunikatów.

Wspomnialiśmy o możliwości opóźnienia operacji w głównym wątku, jeśli zachodzi taka konieczność. Dokonujemy tego za pomocą procedur obsługi. Są one powszechnie wykorzystywane w całym systemie Android po to, aby główny wątek interfejsu użytkownika nie był wstrzymywany. Odgrywają one również istotną rolę w komunikacji pomiędzy głównym wątkiem a innymi wątkami roboczymi. W następnym podrozdziale przyjrzymy się budowie i funkcjom procedur obsługi.

## Procedury obsługi

**Procedura obsługi** (ang. *handler*) jest mechanizmem służącym do umieszczania komunikatu w głównej kolejce (dokładniej mówiąc, w kolejce dołączonej do wątku, dla którego ta procedura została utworzona), dzięki czemu komunikat ten zostanie później przetworzony przez główny wątek. Taki komunikat zawiera wewnętrzne odniesienie wskazujące na procedurę obsługi, która go umieściła.

Gdy główny wątek przechodzi do przetwarzania tego komunikatu, przywołuje odpowiedzialną za jego umieszczenie procedurę obsługi za pomocą metody zwrotnej na obiekcie tej procedury. Metoda ta nosi nazwę `handleMessage`. Na rysunku 13.2 zostały ukazane powiązania pomiędzy procedurami usług, komunikatami i głównym wątkiem.



**Rysunek 13.2.** Związek pomiędzy procedurą obsługi, komunikatem i kolejką komunikatów

Podczas omawiania procedur obsługi przydatny okaże się schemat pokazany na rysunku 13.2, na którym przedstawiono podstawowe współpracujące ze sobą elementy. Tymi elementami są:

- główny wątek,
- kolejka głównego wątka,
- procedura obsługi,
- komunikat.

Spośród tych czterech elementów nie posiadamy bezpośredniego dostępu do wątku ani do kolejki. Możemy bezpośrednio oddziaływać przede wszystkim na obiekty typu Handler i Message. Spośród tych dwóch elementów to właśnie ten pierwszy koordynuje większość działań.

Pomimo znaczenia obiektu Handler w tym oddziaływaniu, powinniśmy pamiętać, że chociaż procedura obsługi pozwala nam na umieszczenie komunikatu w kolejce, to w rzeczywistości właśnie obiekt Message przechowuje odwołanie do tej procedury. Obiekt ten przechowuje również strukturę danych przekazywanych z powrotem do procedury obsługi. Na rysunku 13.2 powiązanie to jest symbolizowane za pomocą odniesienia do obiektu Dane.

Z powodu tego pozornie odwróconego powiązania pomiędzy procedurą obsługi a komunikatem, a także dlatego, że główny wątek i kolejka są ukryte przed wzrokiem programisty, najłatwiej nam będzie zrozumieć pojęcie procedury obsługi na przykładzie.

Przykład ten będzie reprezentowany przez element menu aktywujący funkcję, która z kolei będzie wykonywała jakąś operację pięciokrotnie, co sekundę, i za każdym razem będzie się zgłaszała do wywołującej ją aktywności.

## Skutki przetrzymywania głównego wątku

Jeśli nie mamy nic przeciwko wstrzymywaniu głównego wątku, moglibyśmy zaprojektować wcześniej przedstawiony scenariusz za pomocą pseudokodu widocznego na listingu 13.2.

---

### **Listing 13.2. Wstrzymywanie głównego wątku za pomocą metody sleep()**

---

```
public class JakasAktywnosc
{
    ....inne metody

    void respondToMenuItem()
    {
        //Dowód na to, że mamy do czynienia z głównym wątkiem
        Utils.logThreadSignature();

        for (int i=0;i<5;i++)
        {
            sleepFor(1000); // główny wątek zostaje wstrzymany na 1 sekundę
            robcos();
            JakiśWidokTekstowy.setText("coś zrobiła");
        }
    }
}
```

---

Taka konstrukcja spełni nasze wymogi w omawianym przypadku. Jednak w rzeczywistości w ten sposób wstrzymujemy główny wątek i gwarantujemy sobie wystąpienie komunikatu ANR.

## **Zastosowanie procedury obsługi do opóźnienia operacji w wątku głównym**

Aby uniknąć wyświetlenia komunikatu ANR, nieuniknionego w efekcie wykonania czynności pokazanych w poprzednim przykładzie, możemy wykorzystać procedurę obsługi. Odpowiedzialny za to pseudokod jest widoczny na listingu 13.3.

**Listing 13.3.** Utworzenie procedury obsługi z poziomu głównego wątku

---

```
void respondToMenuItem()
{
    JakasProceduraObslugiPochodzaca0dInnejProceduryObslugi myHandler =
        new JakasProceduraObslugiPochodzaca0dInnejProceduryObslugi ();
    myHandler.wykonuj0poznionaPrace(); //przywołuje funkcję w jednosekundowych odstępach
}
```

---

Metoda `respondToMenuItem()` umożliwia głównemu wątkowi powrót do wykonywania pętli. Utworzona procedura obsługi została przywołana w głównym wątku i podłącza się do kolejki. Metoda `wykonuj0poznionaPrace()` zapewni kontrolę zaplanowanych zadań, dzięki czemu główny wątek w dogodnym momencie powróci do tej operacji. Teraz warto się zastanowić, w jaki sposób ta funkcja działa. Poniżej przedstawiamy etapy jej implementowania:

1. Utwórz obiekt komunikatu, który zostanie umieszczony w kolejce.
2. Wyślij ten obiekt do kolejki w taki sposób, aby mógł co sekundę wywoływać metodę zwrotną.
3. Odpowiedz z poziomu głównego wątku zwrotnym wywołaniem metody `handleMessage()`.

Aby zrozumieć cały ten protokół, przyjrzyjmy się kodowi źródłowemu rzeczywistej procedury obsługi. Został on umieszczony na listingu 13.4 i nosi nazwę `DeferWorkHandler`.

W pseudokodzie z listingu 13.3 procedura obsługi `JakasProceduraObslugiPochodzaca0dInnejProceduryObslugi` jest odpowiednikiem procedury `DeferWorkHandler`. W kodzie z listingu 13.4 zaimplementowano także metodę analogiczną do `wykonuj0poznionaPrace()`.

## Przykładowy kod źródłowy procedury obsługi opóźniającej przeprowadzanie operacji

Zanim omówimy etapy wyszczególnione w poprzednim podrozdziale, przyjrzymy się najpierw kodowi procedury `DeferWorkHandler` na listingu 13.4. Pamiętajmy, że kod źródłowy głównej aktywności wywołującej tę procedurę obsługi został umieszczony w dalszej części rozdziału.

Na listingu 13.4 ta nadrzędna aktywność jest symbolizowana przez zmienną `parentActivity`. Zmienna ta nie jest niezbędna do zrozumienia działania kodu i służy przede wszystkim do informowania nas o stanie pracy wykonywanej w procedurze obsługi.

**Listing 13.4.** Kod źródłowy procedury `DeferWorkHandler`


---

```
public class DeferWorkHandler extends Handler
{
    public static final String tag = "DeferWorkHandler";

    //Zlicza wysłane wiadomości
    private int count = 0;

    //Nadrzędna aktywność, którą możemy wykorzystać
    //do informowania nas o stanie.
    private TestHandlersDriverActivity parentActivity = null;
```

---

```
//W trakcie konstruowania zgłasamy do
//nadrzędnnej aktywności.
public DeferWorkHandler(TestHandlersDriverActivity inParentActivity){
    parentActivity = inParentActivity;
}
@Override
public void handleMessage(Message msg)
{
    String pm = new String(
        "komunikat wywołany:" + count + ":" +
        msg.getData().getString("message"));

    Log.d(tag,pm);
    this.printMessage(pm);

    if (count > 5)
    {
        return;
    }
    count++;
    sendTestMessage(1);
}
public void sendTestMessage(long interval)
{
    Message m = this.obtainMessage();
    prepareMessage(m);
    this.sendMessageDelayed(m, interval * 1000);
}
public void doDeferredWork()
{
    count = 0;
    sendTestMessage(1);
}
public void prepareMessage(Message m)
{
    Bundle b = new Bundle();
    b.putString("message", "Witaj, świecie!");
    m.setData(b);
    return ;
}
//Ta metoda wyświetla jedynie komunikat
//w polu tekstowym nadrzednej aktywnosci.
//Metoda ta została umieszczona na listingu 13.9
private void printMessage(String xyz)
{
    parentActivity.appendText(xyz);
}
```

---

Przyjrzymy się podstawowym aspektom powyższego kodu źródłowego.

## Konstruowanie odpowiedniego obiektu Message

Jak już wspominaliśmy wcześniej, po utworzeniu procedury DeferWorkHandler potrafi ona samostannie podłączyć się do głównego wątku, ponieważ odziedziczyła taką właściwość z podstawowej klasy Handler. Podstawowa procedura obsługi zawiera zestaw metod służących do wysyłania do kolejki komunikatów, które posiadają zdolność do późniejszej odpowiedzi.

Dwoma przykładami takich metod są `sendMessage()` i `sendMessageDelayed()`. Wykorzystywana w naszym przykładzie metoda `sendMessageDelayed()` pozwala nam na umieszczenie komunikatu w głównej kolejce przy z góry założonym opóźnieniu.

Podczas wywoływania metod `sendMessage()` lub `sendMessageDelayed()` potrzebne jest wystąpienie obiektu `Message`. Najlepiej uzyskać ten obiekt z procedury dostępu, ponieważ po jego odesłaniu procedura ta zostaje ukryta w komunikacie. W ten sposób główny wątek, dzięki samej zawartości wiadomości, uzyska informację, która procedura obsługi ma zostać wywołana.

Na listingu 13.4 uzyskujemy komunikat za pomocą następującego wiersza:

```
Message m = this.obtainMessage();
```

Zmienna `this` odnosi się do instancji obiektu `Handler`. Jak sama nazwa wskazuje, metoda ta nie powoduje utworzenia nowego komunikatu, lecz pobiera go z globalnej puli komunikatów. Nieco później, już po przetwarzaniu komunikatu, będzie on ponownie wykorzystywany. Na listingu 13.5 zaprezentowaliśmy różne odmiany metody `obtainMessage()`.

### **Listing 13.5.** Tworzenie komunikatu za pomocą procedury obsługi

---

```
obtainMessage();
obtainMessage(int what);
obtainMessage(int what, Object object);
obtainMessage(int what, int arg1, int arg2)
obtainMessage(int what, int arg1, int arg2, Object obj);
```

---

Każda odmiana tej metody ustanawia odpowiednie pola w obiekcie komunikatu. Istnieją pewne ograniczenia związane z argumentem `Object object`, łączące się z przekraczaniem granicy procesu przez komunikat. W takich przypadkach musi to być obiekt typu `Parcelable`. O wiele bezpieczniej i wygodniej jest stosować jawnie metodę `setData()` wobec obiektu komunikatu, do czego wymagany jest typ `bundle`. Na listingu 13.4 zastosowaliśmy właśnie metodę `setData()`. Zachęcamy również do korzystania z argumentów `arg1` oraz `arg2`, jeżeli chcemy przekazywać proste wskaźniki, które mogą być dostosowane do wartości typu `int`.

Argument `what` pozwala na usunięcie komunikatu z kolejki lub wysłanie zapytania o obecność komunikatów tego typu w kolejce. Więcej szczegółów poznamy po zapoznaniu się z operacjami klasy `Handler`. W podrozdziale „Odbońniki” umieszczonym na końcu tego rozdziału znajdziemy adres URL dokumentacji klasy `Handler`.

## Wysyłanie obiektów Message do kolejki

Gdy już uzyskamy komunikat od procedury obsługi, możemy opcjonalnie zmodyfikować treść danych znajdujących się w tym komunikacie. W naszym przykładzie wykorzystaliśmy funkcję `setData()`, przekazując jej obiekt `bundle`. Po sklasyfikowaniu lub zidentyfikowaniu danych zawartych w komunikacie możemy go przesyłać do kolejki za pomocą metod `sendMessage()` lub `sendMessageDelayed()`. Po wywołaniu tych metod główny wątek powróci do obsługi kolejki.

## Odpowiedź na metodę zwrotną handleMessage

Klasa `DeferWorkHandler` wywodzi się od klasy `Handler`. Po dostarczeniu komunikatów do kolejki procedura obsługi siada i czeka (mówiąc w przenośni) na ich odczytanie przez główny wątek i wywołanie metody `handleMessage()` tej procedury.

Jeżeli chcemy lepiej przyjrzeć się oddziaływaniu wątku głównego z procedurą obsługi, możemy zaprogramować wyświetlanie się wiadomości dziennika `LogCat` w momencie wysyłania komunikatu oraz w chwili wywoływania zwrotnego metody `handleMessage()`. Znaczniki czasowe będą się nieco różniły, ponieważ główny wątek potrzebuje kilku dodatkowych milisekund na powrót do metody `handleMessage()`.

Jest to również dobry sposób na sprawdzenie, czy obydwie metody — `sendMessage()` oraz `handleMessage()` — są uruchomione w głównym wątku. Możemy tego dokonać za pomocą metody `Utils.logThreadSignature()` (listing 13.1).

W naszym przykładzie każda metoda `handleMessage()` po przetworzeniu jednego komunikatu wysyła kolejny komunikat do kolejki, dzięki czemu może zostać ponownie wywołana. Czynność ta jest wykonywana pięciokrotnie, a następnie proces wysyłania komunikatów do kolejki zostaje przerwany.

Jak już wcześniej wspominaliśmy, procedura `DeferWorkHandler` również pobiera nadziedną aktywność jako dane wejściowe, dzięki czemu może zwracać wszelkie informacje za pomocą metod dostarczanych przez tę aktywność.

## Stosowanie wątków roboczych

Kiedy korzystamy z procedury obsługi, takiej jak omówiona w poprzednim podrozdziale, kod jest ciągle przetwarzany w głównym wątku. Każde wywołanie metody `handleMessage()` musi mieścić się w zastrzeżonym czasie głównego wątku (inaczej mówiąc, każde wywołanie komunikatu powinno zostać wykonane w ciągu maksymalnie pięciu sekund, aby wiadomość ANR nie została wyświetlona). Jeżeli naszym zadaniem jest wydłużenie tego czasu przeznaczonego na wykonanie operacji, będziemy musieli rozpocząć oddzielny wątek, utrzymywać go do czasu zakończenia czynności oraz pozwolić takiemu poboczniemu wątkowi na zgłoszenie się do głównej aktywności, uruchomionej w głównym wątku. Taki rodzaj wątku jest często nazywany **wątkiem roboczym**.

Rozpoczęcie osobnego wątku podczas odpowiadania na naciśnięcie elementu menu nie jest wielkim wyzwaniem. Odrobiny wysiłku wymaga jednak umożliwienie wątkowi roboczemu wstawienia komunikatu w kolejce wątku głównego. Komunikat ten miałby na celu wskazanie, że coś się dzieje oraz że wątek główny powinien to obsłużyć, gdy tylko napotka tę wiadomość.

Rozsądne rozwiązanie, polegające na wykorzystaniu wątku roboczego, może wyglądać następująco:

1. Utwórz procedurę obsługi w głównym wątku podczas odpowiadania na element menu. Trzymaj ją pod ręką. W przeciwnieństwie do sytuacji z poprzedniego podrozdziału, tutaj nie będziemy jej używać do wysyłania komunikatów opóźniających działanie.
2. Utwórz osobny wątek (wątek roboczy), który będzie wykonywał daną pracę. Przekaż procedurę obsługi z punktu 1. do tego wątku.
3. Wątek roboczy może teraz przetwarzać operacje trwające dłużej niż 5 sekund, a w międzyczasie może wysyłać wiadomości o stanie, umożliwiające komunikację z wątkiem głównym.

4. Komunikaty o stanie są teraz przetwarzane przez główny wątek, ponieważ procedura obsługi należała do głównego wątku. Główny wątek może przetwarzać te komunikaty, podczas gdy wątek roboczy wykonuje dalej swoją pracę.

Zajmiemy się teraz przykładowym kodem obsługi elementu menu, który uruchamia proces dla wątku roboczego.

## Przywoływanie wątku roboczego z poziomu menu

Kod na listingu 13.6 przedstawia funkcję nazwaną `testThread()`, która może zostać przywołana w odpowiedzi na naciśnięcie elementu menu w wątku głównym.

**Listing 13.6.** Tworzenie wątku pobocznego z poziomu wątku głównego

```
//Przechowujemy kilka zmiennych lokalnych,  
//dzięki czemu nie zostaną one odtworzone za każdym razem, gdy  
//menu zostanie kliknięte w aktywności.  
  
//Przechowuje wskaźnik do procedury obsługi.  
Handler statusBackHandler = null;  
  
//Wystąpienie wątku.  
Thread workerThread = null;  
  
//Ta metoda zostanie przywołana przez menu.  
private void testThread()  
{  
    if (statusBackHandler == null)  
    {  
        //Element menu nie był wcześniej kliknięty.  
        //Widoczne tu klasy zostaną omówione w dalszej części rozdziału.  
        statusBackHandler = new ReportStatusHandler(this);  
        workerThread = new Thread(new WorkerThreadRunnable(statusBackHandler));  
        workerThread.start();  
        return;  
    }  
  
    //Wątek już tu jest.  
    if (workerThread.getState() != Thread.State.TERMINATED)  
    {  
        Log.d(tag, "watek jest nowy albo juz istniejacy, ale niezakonczony");  
    }  
    else  
    {  
        Log.d(tag, "watek jest prawdopodobnie zakonczony. uruchamianie");  
        //Musimy utworzyć nowy wątek.  
        //Nie można odtworzyć zakończonego wątku.  
        workerThread = new Thread(new WorkerThreadRunnable(statusBackHandler));  
        workerThread.start();  
    }  
}
```

Powyższy kod wydaje się nieco złożony, jednak jego sedno mieści się w następujących wierszach:

```
statusBackHandler = new ReportStatusHandler(this);
workerThread = new Thread(new WorkerThreadRunnable(statusBackHandler));
workerThread.start();
```

Zasadniczo utworzyliśmy procedurę obsługi (odpowiedzialną za przesyłanie raportów o stanie), przekazaliśmy ją do wątku roboczego i uruchomiliśmy ten wątek. Dodatkowy kod widoczny na listingu 13.6 jest po to, aby w przypadku dwukrotnego lub trzykrotnego kliknięcia elementu menu podczas działania wątku nie został utworzony nowy wątek ani procedura obsługi.

## Komunikacja pomiędzy wątkami głównym i roboczym

Omówimy teraz klasy ReportStatusHandler i WorkerThreadRunnable. Nie prezentowaliśmy ich wcześniej, ponieważ chcieliśmy przedstawić zasadę działania procedur obsługi oraz wątków, począwszy od ogólnego objaśnienia wymagań, a następnie przejść do szczegółowego omówienia każdego z wykorzystywanych pojęć.

### Implementacja klasy WorkerThreadRunnable

Zobaczmy teraz, jak działa wątek roboczy w klasie WorkerThreadRunnable. Kod źródłowy tej klasy został zamieszczony na listingu 13.7. Wystarczy szybkie spojrzenie na ten kod, zwłaszcza na zawarte w nim komentarze, aby mniej więcej zrozumieć, do czego może służyć. Poniżej wyjaśniamy również podstawowe koncepcje dotyczące tego kodu.

---

#### **Listing 13.7.** Implementacja wątku roboczego

---

```
//Podstawowe zadania
//1. Wykonywanie operacji
//2. Informowanie nadziednej aktywnosci
public class WorkerThreadRunnable implements Runnable
{
    //Procedura obslugi sluzaca do komunikowania sie z glownym wątkiem
    //Ustanawiana w konstruktorze
    Handler statusBackMainThreadHandler = null;

    public WorkerThreadRunnable(Handler h)
    {
        statusBackMainThreadHandler = h;
    }

    //Standardowy znacznik debugowania
    public static String tag = "WorkerThreadRunnable";
    public void run()
    {
        Log.d(tag, "rozpoczecie wykonywania");
        //Sprawdza, który wątek jest uruchomiony w kodzie
        //Poniższa metoda pochodzi z listingu 13.1
        //Wyświetla identyfikator i nazwę wątku
        Utils.logThreadSignature();

        //Informuje wątek nadziedny, że wątek roboczy
        //rozpoczal dzialanie
```

```

    informStart();
    for(int i=1;i <= 5;i++)
    {
        //W rzeczywistej aplikacji byłaby tu wykonywana praca
        //zamiast wstrzymywania
        Utils.sleepForInSecs(1);
        //Informuje o postępach pracy
        informMiddle(i);
    }
    informFinish();
}

public void informMiddle(int count)
{
    Message m = this.statusBackMainThreadHandler.obtainMessage();
    m.setData(Utils.getStringAsABundle("zrobiono:" + count));
    this.statusBackMainThreadHandler.sendMessage(m);
}

public void informStart()
{
    Message m = this.statusBackMainThreadHandler.obtainMessage();
    m.setData(Utils.getStringAsABundle("Przebieg rozpoczęty"));
    this.mainThreadHandler.sendMessage(m);
}
public void informFinish()
{
    Message m = this.statusBackMainThreadHandler.obtainMessage();
    m.setData(Utils.getStringAsABundle("Przebieg kończący"));
    this.statusBackMainThreadHandler.sendMessage(m);
}
}

```

Na listingu 13.7 widzimy dwie istotne rzeczy. W metodzie `run()` wstrzymujemy działanie wątku na 1 sekundę i wywołujemy metody informujące główny wątek o stanie postępów wątku roboczego: czy jest na początku, w środku, czy pod koniec procesu przetwarzania.

Dodatekliśmy również wywołanie metody `Utils.logThreadSignature()`, pozwalającej na zidentyfikowanie wątku.

Jednak w standardowej aplikacji zamiast metody `sleep()` powyższy kod wywoływałby jakąś przydatną funkcję, aby działała, dopóki będzie potrzebna. Możemy uznać tę metodę za symulowanie jakieś części operacji, która zajmuje dokładnie tyle samo sekund.

## Implementacja klasy ReportStatusHandler

Wszystkie metody informacyjne z listingu 13.7 generują odpowiednie komunikaty i wysyłają je do głównego wątku za pomocą widocznej na listingu 13.8 klasy `ReportStatusHandler`.

### **Listing 13.8.** Wysyłanie informacji o stanie do głównego wątku

```

public class ReportStatusHandler extends Handler
{
    public static final String tag = "ReportStatusHandler";

```

```
//Zapamiętuje nadzczną aktywność, dzięki czemu
//możemy ją informować o postępach.
private TestHandlersDriverActivity
    parentTestHandlersDriverActivity = null;

public ReportStatusHandler(
    TestHandlersDriverActivity inParentActivity){
    parentTestHandlersDriverActivity = inParentActivity;
}

@Override
public void handleMessage(Message msg)
{
    //Pobiera ciągi znaków z komunikatu.
    String pm = Utils.getStringFromABundle(msg.getData());
    Log.d(tag,pm);
    //Powiadamia nadzczną aktywność, że coś się stało.
    this.printMessage(pm);
    //Potwierdza, że jest uruchomione w głównym wątku.
    Utils.logThreadSignature();
}

private void printMessage(String xyz){
    parentTestHandlersDriverActivity.appendText(xyz);
}
}
```

---

Kod zawarty w tej klasie jest oczywisty. Kiedy procedura obsługi otrzymuje metodę `handleMessage()`, informuje nadzczną aktywność, że wątek roboczy przesłał komunikat o stanie za pomocą metody `appendText()`. Z kolei nadzonna aktywność po otrzymaniu komunikatu wykonuje odpowiednią operację. W naszym przykładzie zostaje jedynie wyświetlona wiadomość na ekranie aktywności.

Do tej pory, posługując się odpowiednimi przykładami, przedstawiliśmy dość istotne zagadnienia:

- Za pomocą procedury `DeferWorkHandler` pokazaliśmy, że główny wątek może określić moment przetworzenia komunikatu (czy komunikatów), może też opóźnić jego (ich) przetworzenie. Ta sama technika może być wykorzystywana do powtarzania tej samej czynności bez konieczności korzystania z czasomierza lub menedżera alarmów.
- Za pomocą metod `ReportStatusHandler` i `WorkerThread` udowodniliśmy, że możemy rozpoczęć osobny wątek roboczy i umożliwić mu komunikację z interfejsem użytkownika za pomocą procedury obsługi.

## Szybki przegląd — jak działa wątek?

Skoro w odpowiedzi na kliknięcie elementu menu rozpoczęliśmy wątek, naturalną konsekwencją staje się konieczność jego zakończenia. Wątek zostaje automatycznie zatrzymany, gdy metoda `run()` zakończy działanie. W rzeczywistości zaleca się, żeby nie zatrzymywać z zewnątrz działającego wątku, ponieważ w ten sposób możemy przerwać operację w trakcie przetwarzania. Dobrym rozwiązaniem jest ustanowienie flagi, dzięki czemu wątek ją rozpozna i elegancko opuści metodę `run()`.

Warto również zwrócić uwagę na różne stany wątku, aby dobrze zrozumieć jego zachowanie. Wątek może znajdować się w jednym z następujących stanów:

- New thread — został utworzony (`alive=false`);
- Runnable — został uruchomiony (`alive=true`);
- Not runnable — wstrzymany, zawieszony, oczekujący, wywołany lub zablokowany na wejściu-wyjściu (`alive=true`);
- Dead — gdy zostaje wywołana metoda `stop()` lub nastąpi wyjście z metody `run()` (`alive=false`).

Metoda `isAlive()` w wątku informuje nas, czy wątek został uruchomiony, ale nie zatrzymany. Oznacza to, że wątek może znajdować się w stanie `runnable` lub `not-runnable`. Jeżeli zostanie zwrócona wartość `false`, to mamy do czynienia z nowym albo zakończonym wątkiem.

W trakcie pracy z wątkami powinniśmy pamiętać o ich stanach.

## Klasy przykładowego sterownika procedury obsługi

Do tej pory zaprezentowaliśmy kod źródłowy następujących klas:

- `DeferWorkHandler.java` — posiada możliwość opóźniania przetwarzania funkcji (listing 13.4),
- `ReportStatusHandler.java` — jest to nośnik komunikacyjny dla wątku roboczego (listing 13.8),
- `WorkerThreadRunnable.java` — implementacja wątku roboczego (listing 13.7),
- `Utils.java` — umieszczone tu kilka narzędzi wątkowania (listing 13.1).

Nadszedł czas, aby zaprezentować pełny kod źródłowy aktywności sterującej, odpowiadającej na kliknięcie elementu menu i przywołującej omawiane funkcje. Zaprezentujemy również kod źródłowy zasobów menu i plików manifestów, dzięki czemu Czytelnik będzie miał wszystkie klasy wymagane do utworzenia projektu i przetestowania omawianych koncepcji.

Warto zauważyc, że w listingach brakuje nazwy pakietów lub instrukcji importowania. Te drugie łatwo odtworzyć za pomocą środowiska Eclipse. Gdy plik źródłowy jest otwarty, wystarczy użyć skrótu klawiaturowego `Ctrl+Shift+O`, aby Eclipse wstawiło wymagane instrukcje.

Jeśli chodzi o nazwę pakietu, możemy zahrzeć do pliku manifestu i dowiedzieć się, jak ona wygląda dla naszego przykładu. Będziemy musieli umieścić tę nazwę na samym szczycie hierarchii plików źródłowych Java. Ponieważ wszystkie omawiane pliki powinny się znajdująć w obrębie tego samego pakietu, możemy również dostosować jego nazwę do swoich potrzeb, a następnie wstawić ją do pliku manifestu.

### Uwaga!

Można również pobrać gotowy projekt, do którego adres został umieszczony w podrozdziale „Odnośniki” na końcu rozdziału. Nazwa pliku, w którym jest zawarty projekt, to `ProAndroid3_R13_ProceduryObslugi.zip`. Aby utworzyć projekt, należy wypakować zawartość tego pliku oraz zaimportować ją do środowiska ADT.

Poniżej przedstawiamy z koleją listę dodatkowych plików, które będą wymagane do skompilowania projektu:

- *TestHandlersDriverActivity.java* — główna aktywność sterująca (listing 13.9),
- *layout/main.xml* — plik układu graficznego dla klasy *TestHandlersDriverActivity* (listing 13.10),
- *res/menu/main\_menu.xml* — menu służące do przywołania procedur obsługi (listing 13.11),
- *AndroidManifest.xml* — standardowy plik manifest (listing 13.12).

W następnych podrozdziałach zaprezentujemy te pliki jeden po drugim.

## Plik aktywności sterującej

Poniżej prezentujemy pierwszy z wymienionych plików — *TestHandlersDriverActivity.java*. Mamy tu do czynienia z prostą aktywnością zawierającą widok tekstowy. W widoku tym będą umieszczone elementy menu. Za pomocą jednego elementu menu będziemy testować opóźnioną procedurę obsługi, a za pomocą drugiego — wątek roboczy. W umieszczonym tu polu tekstowym będą wyświetlane komunikaty pochodzące z wątku roboczego.

Klasa ta zawiera również metody cyklu życia aktywności aż do jej zakończenia. Chcemy w ten sposób sprawdzić zachowanie głównego wątku i kolejki w odniesieniu do cyklu życia aktywności. Kod tej aktywności znajdziemy na listingu 13.9.

---

### Listing 13.9. Aktywność służąca do testowania procedur obsługi oraz wątków roboczych

---

```
public class TestHandlersDriverActivity extends Activity
{
    public static final String tag="TestHandlersDriverActivity";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater(); //z aktywności
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        appendMenuItemText(item);
        if (item.getItemId() == R.id.menu_clear)
        {
            this.emptyText();
            return true;
        }
        if (item.getItemId() == R.id.menu_test_thread)
        {
            this.testThread();
        }
    }
}
```

```
        return true;
    }
    if (item.getItemId() == R.id.menu_test_defered_handler)
    {
        this.testDeferedHandler();
        return true;
    }
    return true;
}

private TextView getTextView(){
    return (TextView)this.findViewById(R.id.text1);
}
public void appendText(String abc){
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + abc);
}
private void appendMenuItemText(MenuItem menuItem){
    String title = menuItem.getTitle().toString();
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + title);
}
private void emptyText(){
    TextView tv = getTextView();
    tv.setText("");
}

private DeferWorkHandler th = null;
private void testDeferedHandler()
{
    if (th == null)
    {
        th = new DeferWorkHandler(this);
        this.appendText("Tworzenie nowej procedury obsługi");
    }
    this.appendText(
        "Rozpoczęcie wykonywania opóźnionej pracy poprzez wysyłanie komunikatów");
    th.doDeferredWork();
}

Handler statusBackHandler = null;
Thread workerThread = null;
private void testThread()
{
    if (statusBackHandler == null)
    {
        statusBackHandler = new ReportStatusHandler(this);
        workerThread =
            new Thread(
                new WorkerThreadRunnable(statusBackHandler));
    }
    if (workerThread.getState() != Thread.State.TERMINATED)
    {
        Log.d(tag, "watek jest nowy lub istniejacy, ale niezakonczony");
    }
    else
    {
```

```
        Log.d(tag, "watek jest prawdopodobnie zakończony. uruchamianie");
        //Musimy utworzyć nowy wątek.
        //Nie można uruchomić zakończonego wątku.
        workerThread =
            new Thread(
                new WorkerThreadRunnable(statusBackHandler));
        workerThread.start();
    }
}

//Poniższe metody cyklu życia zostały dołączone w celu obserwacji zachowania
//opóźnionych komunikatów oraz natury wątku roboczego w trakcie
//przechodzenia aktywności przez różne etapy cyklu życia.

@Override
protected void onPause() {
    Log.d(tag,"onpause. Mogę być częściowo lub całkowicie niewidoczna");
    this.appendText("onpause");
    super.onPause();
}
@Override
protected void onStop() {
    Log.d(tag,"onstop. Jestem w pełni niewidoczna");
    this.appendText("onstop");
    super.onStop();
}
@Override
protected void onDestroy() {
    Log.d(tag,"ondestroy. Chwile przed usunięciem");
    super.onDestroy();
}
@Override
protected void onRestart() {
    Log.d(tag,"onRestart. Są tu kontrolki interfejsu użytkownika");
    super.onRestart();
}
@Override
protected void onStart() {
    Log.d(tag,"onStart. Interfejs użytkownika może być częściowo niewidoczny");
    super.onStart();
}
@Override
protected void onResume() {
    Log.d(tag,"onResume. Interfejs użytkownika całkowicie widoczny");
    super.onResume();
}
}
```

---

## Plik układu graficznego

Listing 13.10 ukazuje plik układu graficznego (*layout/main.xml*). Mamy tu do czynienia z prostym układem graficznym obsługującym aktywność zaprezentowaną na listingu 13.9. Jak już wspomnieliśmy, zawarty jest w nim pojedynczy widok tekstowy oraz informacja, że kliknięcie elementu menu spowoduje uruchomienie procedury obsługi wątku roboczego.

**Listing 13.10.** Plik układu graficznego

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Kliknij menu, aby wyświetlić dostępne opcje"
    />
</LinearLayout>
```

---

**Plik menu**

Wykorzystywany na potrzeby tego przykładu kod pliku menu, *menu/main\_menu.xml*, został umieszczony na listingu 13.11. Jest to menu obsługujące aktywność zdefiniowaną na listingu 13.9. Jak wynika z aktywności, menu to zawiera trzy elementy. Jeden służy do czyszczenia widoku tekstowego podczas pracy z opcjami menu. Dalej mamy dwa główne elementy menu: *menu\_test\_defered\_handler* przywołuje procedurę obsługi *DeferWorkHandler*, natomiast *menu\_test\_thread* tworzy wątek roboczy i wykorzystuje procedurę *ReportStatusHandler*.

**Listing 13.11.** Elementy menu wywołujące kod procedury obsługi i wątku pobocznego

---

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_clear"
            android:title="Wyczyść" />

        <item android:id="@+id/menu_test_thread"
            android:title="Testuj wątek roboczy" />

        <item android:id="@+id/menu_test_defered_handler"
            android:title="Opóźniona procedura obsługi" />
    </group>
</menu>
```

---

**Plik manifest**

Listing 13.12 prezentuje plik manifest (*AndroidManifest.xml*), który zamyka listę plików źródłowych. Jego zawartość jest bardzo nieskomplikowana, gdyż wskazuje tylko pojedynczą aktywność, widoczną na listingu 13.9 (główna aktywność sterująca).

**Listing 13.12.** Plik AndroidManifest.xml

---

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.handlers"
    android:versionCode="1"
```

```
    android:versionName="1.0.0">
<application android:icon="@drawable/icon" android:label="Testowe
↳procedury obsługi">
    <activity android:name=".TestHandlersDriverActivity"
        android:label="Testowe procedury obsługi">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
<uses-sdk android:minSdkVersion="3" />
</manifest>
```

---

Plik manifest zawiera odniesienie do ikony aplikacji. Możemy wykorzystać plik ZIP projektu, do którego odnośnik znajduje się na końcu rozdziału, albo możemy umieścić inną ikonę z dowolnego innego projektu.

## Czas życia składnika i procesu

Jeżeli przyjrzelismy się uważnie testowej aktywności `TestHandlersDriverActivity` (listing 13.9), na pewno zauważymy, że zostały w niej zamieszczone różne metody dotyczące poszczególnych etapów cyklu życia aktywności. Wprowadziliśmy je po to, aby pokazać, co się stanie, gdy aktywność zostanie ukryta oraz ponownie wyświetcona. Co się stanie z oczekującymi komunikatami w głównej kolejce? Co się dzieje z przetwarzanym wątkiem roboczym?

Wyjaśnimy zachodzące zjawiska poprzez omówienie cyklu życia każdego składnika Androida.

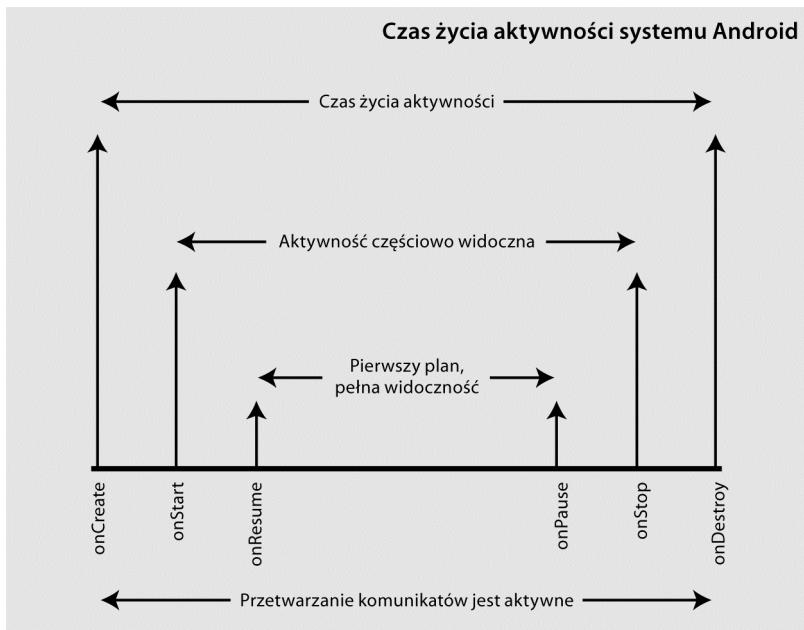
Chociaż omówimy zaraz cykle życia składników, przypominamy, że nie zajmujemy się tu nimi szczegółowo. Cykl życia aktywności został dokładnie omówiony (wraz z diagramem) w rozdziale 2. Z kolei dyskusję na temat cyklu życia usługi odnajdziemy w rozdziale 11. Informacje przedstawione w dalszej części rozdziału dotyczą wyłącznie aspektów wpływających na przetwarzanie komunikatów oraz wątki robocze.

## Cykł życia aktywności

Rozpoczniemy od składnika `Activity`. Rysunek 13.3 prezentuje cykl życia aktywności z uwzględnieniem jej widoczności oraz czasu życia (zmiany stanu aktywności w zależności od jej metod cyklu życia) zostały omówione w rozdziale 2.).

Po utworzeniu aktywności (w momencie uruchomienia aplikacji) może być ona całkowicie widoczna, częściowo widoczna lub zupełnie niewidoczna. Możemy wykryć każdą taką granicę widoczności za pomocą metod zwrotnych.

Aktywność wywołuje metodę `onPause`, gdy przechodzi do stanu częściowej widoczności. Z tego miejsca może wywołać metodę `onStop`, dzięki której staje się zupełnie niewidoczna. Ostatecznie, po zakończeniu procesu, zostaje wywołana metoda `onDestroy`, co jest jednoznaczne z całkowitym usunięciem stanu widoczności. Przedtem stan ten jest ciągle dostępny.



**Rysunek 13.3.** Cykl życia aktywności

Gdy aktywność przechodzi do stanu pełnej widoczności, zostaje wywołana metoda onResume. Jeżeli przechodzi ze stanu zupełnej niewidoczności, najpierw zostaje wywołana metoda onStart, a następnie dopiero onResume (albo onStop, jeśli aktywność ma znowu stać się niewidoczna). Pomiędzy wywołaniami metod onResume i onPause aktywność jest w pełni widoczna.

Chociaż aplikacja może być częściowo lub całkowicie niewidoczna, kolejka komunikatów będzie niezmiennie aktywna, tak samo jak wątek roboczy. Zauważmy to, jeśli będziemy obserwować metody cyklu życia aktywności pokazane na listingu 13.9. Stwierdzimy, że komunikaty z wątku roboczego oraz z procedury obsługi są ciągle aktywne po wywołaniu metod onPause i onStop.

Mogliśmy przetestować tę hipotezę, klikając przycisk ekranu startowego w trakcie przebywania w aktywności. W ten sposób przeniesiemy aktywność do tła i wywołamy metody onPause, onStop, a może nawet onDestroy. Przez cały czas będą generowane komunikaty, dopóki nie zostanie wywołana metoda onDestroy (pod warunkiem że wysłaliśmy tyle komunikatów).

Jeżeli proces nie jest aktywny w czasie żądania wywołania aktywności, zostanie ona uruchomiona i zwrócona na pierwszy plan. W przypadku ograniczenia pamięci lub wtedy, gdy aktywność jest ukryta i nic się nie dzieje w danym procesie, zostanie on usunięty przez system.

#### Uwaga!

Najważniejsza jest świadomość, że jeżeli aktywność zostanie zatrzymana z któregoś z wymienionych wyżej powodów, nie zostanie później automatycznie przywrócona. Użytkownik musi jawnie przywołać aktywność — albo poprzez kliknięcie elementu menu, albo w jakiś inny, pośredni sposób, na przykład uruchomienie aktywności, której działanie przywoła tę zatrzymaną aktywność. Jedynym przypadkiem, w którym aktywność zostanie zatrzymana i ponownie uruchomiona, jest zmiana konfiguracji urządzenia (na przykład zmiana orientacji ekranu). Mogliśmy sobie wyobrazić, że takie obracanie telefonu może występować dość często.

## Cykl życia usługi

Usługa zachowuje się inaczej od aktywności pod jednym podstawowym względem — jest ona zasadniczo trwała. Android gwarantuje nieprzerwane działanie usługi, jeśli to tylko jest możliwe. Nawet jeśli proces usługi musi zostać zatrzymany z powodu ograniczeń pamięci, zostanie uruchomiony ponownie, jeśli będą się w nim znajdować komunikaty oczekujące na przetworzenie. Zajmiemy się tym zagadnieniem o wiele dokładniej w następnym rozdziale, w trakcie omawiania odbiorców komunikatów oraz usług długoterminowych.

Jednak wspólną cechą aktywności i usługi jest możliwość ich zamknięcia z powodu braku pamięci. System będzie „się starał” utrzymać usługę w stanie działania, nie ma jednak żadnych gwarancji, że uda się utrzymać ją do samego końca.

### Uwaga!

Aktywności i usługi powinny być pisane w taki sposób, aby umożliwiać ich eleganckie zatrzymywanie po wywołaniu metody `onDestroy`, nawet jeśli przypisano im wątki robocze, w których zachodzi ich przetwarzanie.

## Cykl życia odbiorców komunikatów

Odbiorcy komunikatów działają zgodnie z zasadą „wywołaj i znikaj”. Proces związany z odbiorcą będzie dostępny wyłącznie przez czas trwania tego odbiorcy i ani chwili dłużej. Poza tym odbiorca komunikatów działa w głównym wątku i ma bezwzględnie ustaloną, dziesięciosekundową ramę czasową na przeprowadzenie operacji. Aby w odbiorcy komunikatów przeprowadzać bardziej skomplikowane, dłużej trwające zadania, musimy postępować zgodnie z dość złożonym protokołem. To będzie, w rzeczy samej, tematem następnego rozdziału. W skrócie jednak, jeśli dany odbiorca komunikatów będzie pracował dłużej niż dziesięć sekund, musimy zastosować następujący protokół:

1. Ustaw blokadę przechodzenia urządzenia w stan uśpienia (ang. *wakelock*) w kodzie odbiorcy (nie później), dzięki czemu urządzenie będzie przynajmniej częściowo aktywne.
2. Wyślij wywołanie metody `startService()`, dzięki czemu proces zostanie oznaczony jako trwały oraz, w razie potrzeby, ponownie uruchamialny. Dzięki temu proces będzie łatwiej dostępny.
3. Zwróć uwagę, że nie możesz wykonywać operacji bezpośrednio na usłudze, bo zajmie to więcej czasu niż dziesięć sekund, a to spowodowałoby wstrzymanie głównego wątku. Wynika to z faktu, że usługa również działa w głównym wątku.
4. Rozpocznij wątek roboczy z poziomu usługi.
5. Zaprogramuj wątek roboczy, aby zamieszczał komunikaty za pomocą procedury obsługi w usłudze, albo wywołaj na usłudze metodę `stopService()`.

Zgodnie z wcześniejszą obietnicą w następnym rozdziale przeanalizujemy o wiele dokładniej powyższy protokół. W rzeczywistości jest on silnie zależny od procedur obsługi. Omawiane koncepcje będziemy wyjaśniać za pomocą sporej ilości przykładowego kodu.

## Cykl życia dostawcy treści

Z dostawcą treści jest zupełnie inna historia. Zarówno wewnętrzne, jak i zewnętrzne klienci oddziałują synchronicznie na dostawcę treści. W przypadku klientów zewnętrznych jest to możliwe dzięki puli wątków. Podobnie jak odbiorcy komunikatów, dostawcy treści nie posiadają

jakiegoś szczególnego cyklu życia. Od momentu uruchomienia istnieją, dopóki trwa proces. Nawet jeśli wykazują synchroniczność wobec zewnętrznych klientów, nie będą przetwarzani w głównym wątku, lecz w puli wątków przechowującego ich procesu, podobnie jak ma to miejsce w przypadku związku klient sieciowy – serwer sieciowy. Wątek kliencki będzie oczekiwany, dopóki nie nadaje się wywołanie zwrotne. Jeżeli nie ma żadnych klientów, proces zostaje odzyskany zgodnie z regułami odzyskiwania, w zależności od tego, jakie inne składniki zostały zdefiniowane oraz aktywnie działają w tym procesie.

## Instrukcje dotyczące kompilowania kodu

W tym rozdziale utworzyliśmy osiem głównych plików projektu. Zalecamy pobranie pliku ZIP dostępnego pod adresem umieszczonym na końcu rozdziału, w podrozdziale „Odbońniki”, chociaż możemy również skompilować projekt za pomocą listingów zamieszczonych w książce.

### Tworzenie projektu za pomocą pliku ZIP

Aby utworzyć projekt za pomocą pliku ZIP, trzeba wykonać następujące czynności:

1. Pobierz plik ZIP.
2. Wybierz opcje *File/Import* z menu środowiska Eclipse.
3. Następnie kliknij opcje *General/Existing Project into Workspace*.
4. Wybierz teraz opcję *Select Root Directory*.
5. Zaznacz opcję *Copy projects into workspace*.
6. Być może zaistnieje potrzeba wyboru odpowiedniego poziomu interfejsu API po zamieszczeniu projektu poprzez wybór opcji *Project properties/Android* oraz zaznaczenie właściwego interfejsu.

### Tworzenie projektu za pomocą listingów

Można ewentualnie skonstruować projekt z listingów umieszczonych w tym rozdziale. Potrzebne pliki zostały wymienione w podrozdziale „Klasy przykładowego sterownika procedury obsługi”; poniżej prezentujemy wymagane czynności:

1. Utwórz nowy projekt, wybierając opcje *File/New Project/Android/Android Project*.
2. Wybierz nazwę i zaznacz opcję *Create new project in workspace*.
3. Nazwij aplikację *Testowe procedury obsługi*.
4. Wybierz poziom interfejsu API.
5. Wykorzystaj nazwę pakietu, na przykład `com.androidbook.handlers`.
6. Wybierz wartość 3 parametru `min SDK version`.
7. Wprowadź nazwę aktywności `TestHandlersDriverActivity` i kliknij przycisk *Finish*.
8. Android wygeneruje pliki zasobów oraz najprawdopodobniej jeden plik źródłowy (w zależności od wersji środowiska).
9. Utwórz lub zaktualizuj pliki na podstawie listingów zamieszczonych w tym rozdziale.
10. W przypadku plików Java umieść nazwę pakietu na samym początku listingu, zanim go skopiujesz. Następnie, po skopiowaniu i wklejeniu kodu, wcisnij kombinację klawiszy `Ctrl+Shift+O`, aby uzupełnić plik o instrukcje importu.

Należy pamiętać, że w czasie tworzenia projektu trzeba dostosować przedstawione tu pliki i wprowadzić pewne brakujące składniki, aby projekt został skompilowany. Wszelkie braki łatwo uzupełnić, korzystając z pliku ZIP.

## Odbośniki

W czasie zapoznawania się z tematami omawianymi w tym rozdziale Czytelnik może zechcieć zdobyć więcej informacji. Są one dostępne pod poniższymi adresami URL; przy każdym odnośniku umieściliśmy również krótką notatkę na temat prezentowanych zasobów.

- <http://developer.android.com/reference/android/os/Handler.html> — znajdziemy tu odniesienie do interfejsu API procedur obsługi. Zostały tu omówione metody pozwalające na konstruowanie procedur obsługi, uzyskiwanie komunikatu, przesłanie metod `handleMessage()` oraz `sendMessage()` i tak dalej.
- <http://developer.android.com/reference/android/os/Message.html> — pod tym adresem zdobędziemy informacje na temat interfejsu API komunikatów. Chociaż interfejs ten jest stosunkowo rzadko stosowany, ponieważ równoważne funkcje dostępne są w interfejsie API procedury obsługi, warto poznać fundamenty obiektu `Message` i dowiedzieć się co nieco na temat tego interfejsu. Zalecamy zapoznanie się z wiadomościami umieszczonymi pod tym adresem.
- <http://developer.android.com/guide/topics/fundamentals.html#lcycles> — szczegółowe dane na temat cykłów życia. Został tu położony nacisk przede wszystkim na cykle życia aktywności i usług, wspomniano także o cyklach życia odbiorców komunikatów. Właściwie nie znajdziemy tu informacji na temat dostawców treści.
- <http://www.science.uva.nl/ict/ossdocs/java/tutorial/java/threads/states.html> — bardzo merytoryczny i niezbędny do przeczytania artykuł wprowadzający w tematykę wątków.
- <http://www.netmite.com/android/mydroid/1.6/frameworks/base/core/java/android/app/IntentService.java> — znakomity przykład wykorzystania procedur obsługi przez bazowy kod systemu Android w implementacji klasy `IntentService`. Niniejszy adres jest odniesieniem do kodu źródłowego zawartego w pliku `IntentService.java`. Bardzo zalecamy, by po przeczytaniu tego rozdziału, w celu utrwalenia wiadomości dotyczących wątków, przejrzeć kod źródłowy klasy `IntentService`.
- <http://www.androidbook.com/item/3514> — notatki jednego z autorów dotyczące usług o dłuższym czasie trwania.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów utworzonych na potrzeby książki. Właściwy plik znajdziesz w katalogu o nazwie *ProAndroid3\_R13\_ProceduryObslugi*.

## Podsumowanie

W tym rozdziale przeanalizowaliśmy różne składniki procesu Androida, a także sposób koordynowania ich działania przez główny wątek. Pokazaliśmy, w jaki sposób procedury obsługi i wątki mogą być wykorzystywane do poszerzania zasięgu głównego wątku, a także dlaczego główny wątek musi powracać do działania w przeciągu pięciu sekund, zanim pojawi się komunikat ANR. Taka sama zasada dotyczy odbiorców komunikatów, przy czym w ich przypadku rama czasowa zostaje zwiększena do dziesięciu sekund.

Omówiliśmy cykle życia poszczególnych składników oraz ich wpływ na wątek główny i wątki poboczne. Wiedza ta jest niezbędna do zrozumienia zawiłości tych składników oraz czynności potrzebnych do wykonywania długoterminowych operacji.

Następny rozdział jest poświęcony pracy z odbiorcami komunikatów oraz przeprowadzaniu długoterminowych działań. Informacje zawarte w tym rozdziale pomogą nam zrozumieć pojęcia przedstawione w następnym.



# Odbiorcy komunikatów i usługi długoterminowe

W poprzednich rozdziałach zajmowaliśmy się w przeważającej części aktywnościąmi, dostawcami treści i usługami. Niezbyt dokładnie przedstawiliśmy zagadnienia dotyczące odbiorców komunikatów, dlatego teraz przyjrzymy im się uważniej.

Zaproponujemy najpierw sposób przywołania prostego odbiorcy komunikatów, a następnie rozszerzymy ten mechanizm na procedurę wywołania wielu takich obiektów. Pokażemy również, w jaki sposób odbiorcy komunikatów mogą się znajdować w zewnętrznych procesach. Zademonstrujemy także sposób, w jaki odbiorcy komunikatów wysyłają powiadomienia poprzez menedżer powiadomień.

Przeanalizujemy dziesięciosekundowy limit, w którym musi się zmieścić odbiorca komunikatów, aby nie została wyświetlona informacja ANR (aplikacja nie odpowiada), a także sposoby pominięcia tego ograniczenia czasowego. Utworzmy specjalną strukturę, pozwalającą na obserwowanie długoterminowej usługi, będącej specjalną wersją odbiorcy komunikatów, na końcu natomiast omówimy blokady przechodzenia urządzenia w stan zatrzymania (ang. *wakelock*) w kontekście długoterminowych usług.

Rozpoczniemy od obszernej analizy odbiorców komunikatów na przykładzie utworzenia prostego obiektu tego typu.

## Odbiorcy komunikatów

W rozdziale 13. zdefiniowaliśmy odbiorcę komunikatów (ang. *broadcast receiver*) jako jeden ze składników procesu (innymi składnikami są: aktywność, usługa oraz dostawca treści). Jak sama nazwa wskazuje, zadaniem odbiorcy komunikatów jest odpowiadanie na wiadomości wysypane przez klienta. Sam taki komunikat jest intencją, która może być odbierana przez wielu odbiorców.

Taki składnik, jak aktywność lub usługa (albo inny komponent implementujący klasę `Context`), próbujący wysłać zdarzenie (intencję), korzysta z metody `sendBroadcast()` dostępnej w klasie `Context`. Argumentem tej metody jest wysyłana intencja.

Składniki odbierające przesyłaną intencję muszą dziedziczyć po klasie Receiver, dostępnej w zestawie Android SDK. Te odbierające składniki (odbiorcy komunikatów) muszą zostać następnie zarejestrowane w pliku manifeście jako obiekt receiver, który reaguje na nadawaną intencję.

**Uwaga!**

Możemy również rejestrować odbiorców treści w trakcie działania kodu, bez konieczności ich rejestrowania w pliku manifeście. Zwróćmy uwagę, że nie omawiamy tu tego rozwiązania, zalecamy natomiast przejrzenie dokumentacji interfejsu API, do której adres można znaleźć w podrozdziale „Odnośniki”.

## Wysyłanie komunikatu

Listing 14.1 przedstawia przykładowy kod będący częścią klasy aktywności, służący do przesyłania komunikatu. Za pomocą tego kodu tworzymy intencję zawierającą unikatowe, specyficzne dla niej działanie, następnie wstawiamy do niej dodatkowy komunikat oraz wywołujemy metodę sendBroadcast(). Wstawienie dodatkowego komunikatu jest opcjonalną czynnością; często samo otrzymanie intencji całkowicie wystarczy odbiorcy, a dodatkowy komunikat okazuje się zbędny.

---

**Listing 14.1.** Nadawanie intencji

```
private void testSendBroadcast(Activity activity)
{
    //Tworzy intencję zawierającą określone działanie
    String uniqueActionString = "com.androidbook.intents.testbc";
    Intent broadcastIntent = new Intent(uniqueActionString);
    broadcastIntent.putExtra("message", "Witaj, świecie!");
    activity.sendBroadcast(broadcastIntent);
}
```

---

W kodzie z listingu 14.1 działaniem jest niestandardowy identyfikator, dostosowany do naszych potrzeb. Żeby ten ciąg znaków działania był niepowtarzalny, możemy wprowadzić przestrzeń nazw podobną do stosowanej w klasie Java. Spójrzmy teraz, w jaki sposób możemy odpowiedzieć na tak nadaną intencję.

## Tworzenie prostego odbiorcy — przykładowy kod

Na listingu 14.2 został umieszczony kod odbiorcy odpowiadający na nadalaną intencję, utworzoną z kodu na listingu 14.1.

---

**Listing 14.2.** Przykładowy kod odbiorcy

```
public class TestReceiver extends BroadcastReceiver
{
    private static final String tag = "TestReceiver";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Utils.logThreadSignature(tag);
        Log.d("TestReceiver", "intencja=" + intent);
        String message = intent.getStringExtra("message");
    }
}
```

---

```

        Log.d(tag, message);
    }
}

```

---

Utworzenie odbiorcy komunikatów jest dość proste. Rozszerzamy po prostu klasę Broadcast Receiver i przesyłamy metodę onReceive(). Możemy obejrzeć intencję wewnątrz tego odbiorcy i odczytać z niej komunikat. Jeżeli nadawana intencja nie posiada dodatkowego komunikatu, nazwanego tutaj message, zostanie zwrócona wartość null. Ponieważ wiemy, że w naszym przykładzie ten komunikat został wstawiony do intencji, nie przeprowadziliśmy testu na wartość null. Po odczytaniu tego komunikatu wypisujemy go w oknie dziennika.

Umieściliśmy w naszym testowym odbiorcy metodę narzędziową, pozwalającą na zapisywanie w dzienniku sygnatury wątku, w którym jest przetwarzany kod odbiorcy. Ponieważ w tym rozdziale będziemy często korzystać z klasy Utils, na listingu 14.3 zamieszczamy kod źródłowy pliku *Utils.java*.

#### **Listing 14.3.** Definicja klasy Utils

---

```

public class Utils
{
    public static long getThreadId()
    {
        Thread t = Thread.currentThread();
        return t.getId();
    }
    public static String getThreadSignature()
    {
        Thread t = Thread.currentThread();
        long l = t.getId();
        String name = t.getName();
        long p = t.getPriority();
        String gname = t.getThreadGroup().getName();
        return (name + ":(id)" + l + ":(priorytet)" + p
            + ":(grupa)" + gname);
    }
    public static void logThreadSignature(String tag)
    {
        Log.d(tag, getThreadSignature());
    }
    public static void sleepForInSecs(int secs)
    {
        try
        {
            Thread.sleep(secs * 1000);
        }
        catch(InterruptedException x)
        {
            throw new RuntimeException("przerwano",x);
        }
    }
}

```

---

Po utworzeniu odbiorcy komunikatów z listingu 14.2 musimy zarejestrować go w pliku manifeście.

## Rejestrowanie odbiorcy komunikatów w pliku manifeście

Na listingu 14.4 widać, w jaki sposób można zadeklarować odbiorcę jako adresata intencji, której działaniem jest identyfikator `com.androidbook.intents.testbc`.

**Listing 14.4.** Definicja odbiorcy w pliku manifeście

---

```
<manifest>
<application>
...
<activity ....>
...
<receiver android:name=".TestReceiver">
    <intent-filter>
        <action android:name="com.androidbook.intents.testbc"/>
    </intent-filter>
</receiver>
...
</application>
</manifest>
```

---

Podobnie jak w przypadku pozostałych rodzajów składników, obiekt `receiver` jest węzłem potomnym elementu `application`. To nam wystarczy do przetestowania odbiorcy. W następnym punkcie zamieszczamy listę plików wymaganych do utworzenia projektu testowego.

Zanim zaczniemy z zapalem kopiować (albo, co gorsza, przepisywać) kody zamieszczone na listingach, pamiętajmy, że w podrozdziale „Odnośniki” został umieszczony adres URL, pod którym znajdziemy listę wszystkich projektów — możemy je pobrać na dysk i zimportować do środowiska Eclipse.

## Wysyłanie komunikatu testowego

Poniżej przedstawiamy listę plików wymaganych do utworzenia projektu testowego wraz z odniesieniami do odpowiednich listingów:

- `TestBCRActivity.java` — przykładowa aktywność uruchamiająca odbiorcę wiadomości (w skrócie BCR; listing 14.5),
- `layout/main.xml` — prosty, tekstowy układ graficzny wyświetlający komunikaty; układ ten zostanie wykorzystany w aktywności `TestBCRActivity` (listing 14.6),
- `menu/main_menu.xml` — menu pozwalające na ponowną transmisję komunikatu, wykorzystywane przez klasę `TestBCRActivity` (listing 14.7),
- `TestReceiver.java` — przykładowy odbiorca komunikatów (zaprezentowany na listingu 14.2),
- `Utils.java` — kilka narzędzi wątkowania (plik zaprezentowany na listingu 14.3),
- `AndroidManifest.xml` — plik manifest, w którym zostały zdefiniowane aktywność oraz odbiorca komunikatów (listing 14.8).

Wcześniej już zaprezentowaliśmy część plików wchodzących w skład tego projektu, zamieścimy więc teraz pozostałe kody. Na listingu 14.5 widzimy aktywność `TestBCRActivity` przywołującą menu, za pomocą którego możemy nadać komunikat. Wywołanie menu zostało oznaczone pogrubioną czcionką.

**Listing 14.5.** Aktywność klienta nadawczego

```

public class TestBCRActivity extends Activity
{
    public static final String tag="TestBCRActivity";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu){
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater(); //z aktywności
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item){
        appendMenuItemText(item);
        if (item.getItemId() == R.id.menu_clear){
            this.emptyText();
            return true;
        }
        if (item.getItemId() == R.id.menu_send_broadcast){
            this.testSendBroadcast();
            return true;
        }
        return true;
    }
    private TextView getTextView(){
        return (TextView)this.findViewById(R.id.text1);
    }
    private void appendMenuItemText(MenuItem menuItem){
        String title = menuItem.getTitle().toString();
        TextView tv = getTextView();
        tv.setText(tv.getText() + "\n" + title);
    }
    private void emptyText(){
        TextView tv = getTextView();
        tv.setText("");
    }
    private void testSendBroadcast()
    {
        //Wyświetla identyfikator działającego wątku
        Utils.logThreadSignature(tag);

        //Tworzy intencję zawierającą działanie
        Intent broadcastIntent = new Intent("com.androidbook.intents.testbc");

        //Zapisuje w intencji komunikat,
        //który chcemy nadać
        broadcastIntent.putExtra("message", "Witaj, świecie!");

        //Wysyła komunikat
    }
}

```

```
//Może go odczytać wielu odbiorców  
this.sendBroadcast(broadcastIntent);  
  
//Wyświetla komunikat po wysłaniu go do odbiorcy  
//Powinien on pojawić się najpierw w pliku dziennika  
//jeszcze przed komunikatami pochodząymi z transmisji,  
//ponieważ działają one w tym samym wątku  
Log.d(tag,"po wysłaniu komunikatu z poziomu głównego menu");  
}  
}
```

---

Układ graficzny obsługujący klasę `TestBCRActivity` został zaprezentowany na listingu 14.6, a generowany z niego widok został pokazany na rysunku 14.1.

#### **Listing 14.6.** Plik układu graficznego

---

```
<!-- layout/main.xml -->  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    >  
<TextView  
    android:id="@+id/text1"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="Tutaj będą wyświetlane komunikaty debugowania"  
    />  
</LinearLayout>
```

---

Poniżej na listingu 14.7 przedstawiamy plik menu.

#### **Listing 14.7.** Plik zasobów menu

---

```
<!-- menu/main_menu.xml -->  
<menu xmlns:android="http://schemas.android.com/apk/res/android">  
    <group android:id="@+id/menuGroup_Main">  
        <item android:id="@+id/menu_clear"  
            android:title="Wyczyść" />  
        <item android:id="@+id/menu_send_broadcast"  
            android:title="Transmituj" />  
    </group>  
</menu>
```

---

**Uwaga!**

Pełny kod źródłowy pliku `TestReceiver.java` znajdziemy na listingu 14.2, natomiast pliku `Utils.java` — na listingu 14.3.

Listing 14.8 zawiera kod źródłowy pliku manifestu.

**Listing 14.8.** Plik AndroidManifest.xml

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.bcr"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="Testowy odbiorca kom.">
        <activity android:name=".TestBCRActivity"
            android:label="Testowy odbiorca komunikatów">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".TestReceiver">
            <intent-filter>
                <action android:name="com.androidbook.intents.testbc"/>
            </intent-filter>
        </receiver>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>

```

Po skompilowaniu i uruchomieniu tego projektu ujrzymy aktywność i menu przypominające ekran widoczny na rysunku 14.1.



**Rysunek 14.1.** Przykładowa aktywność zawierająca menu, pozwalające na przetestowanie odbiorcy komunikatów

Po kliknięciu przycisku *Transmituj* zostanie wywołany odbiorca *TestReceiver*, zdefiniowany na listingu 14.2, a w oknie *LogCat* pojawi się komunikat *Witaj, świecie!*. Komunikat ten został wczytany do wysyłanej intencji przez aktywność.

## Wprowadzanie wielu odbiorców komunikatów

Jednym z kluczowych punktów koncepcji nadawania komunikatów jest możliwość ich odczytywania przez wielu odbiorców. Spróbujmy więc powieść klasę TestReceiver (z listingu 14.2) jako TestReceiver2 i sprawdzić, czy obydwie zostaną przywołane. Kod odbiorcy TestReceiver2 został zaprezentowany na listingu 14.9.

**Listing 14.9.** Powielony odbiorca komunikatów

---

```
public class TestReceiver2 extends BroadcastReceiver
{
    private static final String tag = "TestReceiver2";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Utils.logThreadSignature(tag);
        Log.d(tag, "intencja=" + intent);
        String message = intent.getStringExtra("message");
        Log.d(tag, message);
    }
}
```

---

Po utworzeniu powyższego kodu możemy dodać tego odbiorcę do pliku manifestu (listing 14.8), zgodnie z poniższą definicją (listing 14.10).

**Listing 14.10.** Definicja odbiorcy TestReceiver2 w pliku manifeście

---

```
<receiver android:name=".TestReceiver2">
    <intent-filter>
        <action android:name="com.androidbook.intents.testbc"/>
    </intent-filter>
</receiver>
```

---

Jeżeli teraz ponownie przywołamy element menu widoczny na rysunku 14.1, w oknie *LogCat* ujrzymy komunikat  *Witaj, świecie!* pochodzący z obydwu odbiorców.

Stwierdzimy również, że odbiorcy są wywoływani zgodnie z kolejnością ich definiowania w pliku manifeście. Możemy także sprawdzić, w jakim wątku ci odbiorcy są uruchomieni. Wywołanie metody `Utils.logThreadSignature(tag)` spowoduje wyświetlenie sygnatury działającego wątku. Okaże się, że w istocie odbiorcy są przetwarzani w głównym wątku.

Zauważymy ponadto, że komunikaty dziennika umieszczone przed wywołaniem metody `sendBroadcast()` w metodzie `testSendBroadcast()` oraz po jej wywołaniu (listing 14.5) będą wyświetlane w oknie dziennika przed wiadomościami z odbiorców komunikatów oraz posiadają tę samą sygnaturę wątku.

Mamy więc dowód, że główny wątek pracuje przez cały czas i po wyjściu z kolejki wiadomości przetwarza odbiorców komunikatów. Widać wyraźnie, że metoda `sendBroadcast()` jest asynchroniczną wiadomością, umożliwiającą głównemu wątkowi powrót do kolejki komunikatów.

Jeżeli chcemy otrzymać mocniejsze dowody, możemy przetrzymać główny wątek trochę dłużej, tak aby znaczniki czasowe zostały wyraźnie zaznaczone. Utwórzmy kolejnego odbiorcę komunikatów, który tym razem będzie nieco opóźniał wątek główny za pomocą wstrzymania. Kod źródłowy takiego opóźniającego odbiorcy został zaprezentowany na listingu 14.11.

#### **Listing 14.11.** Odbiorca generujący opóźnienie czasowe

```
/*
 * Odbiorca ten został wprowadzony w celu ukazania,
 * w jaki sposób główny wątek ustala kolejność przetwarzania odbiorców komunikatów
 *
 * Przykład ten pomaga odpowiedzieć na takie pytania, jak:
 * 1. Czy są one przywoływane w kolejności ich definiowania w pliku manifeście?
 * 2. Czy są one przetwarzane jeden po drugim, czy też równolegle?
 *
 * Widzimy, że za pomocą opóźnienia czasowego wprowadzamy główny wątek
 * w stan wstrzymania na tyle właśnie sekund. Widać to
 * w pliku wynikowym Log.d
 */
public class TestTimeDelayReceiver extends BroadcastReceiver
{
    private static final String tag = "TestTimeDelayReceiver";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Utils.logThreadSignature(tag);
        Log.d(tag, "intencja=" + intent);
        Log.d(tag, "przechodzi w stan wstrzymania na 2 sekundy");
        Utils.sleepForInSecs(2);
        Log.d(tag, "wybudzanie");
        String message = intent.getStringExtra("message");
        Log.d(tag, message);
    }
}
```

Jeżeli teraz wstawimy definicję tego odbiorcy pomiędzy definicje dwóch poprzednio utworzonych, zauważymy, że główny wątek jest przeglądany przez podstawową logikę oraz logikę odbiorcy komunikatu. Obserwując okno *LogCat*, stwierdzimy, że najpierw jest przetwarzany pierwszy odbiorca. Następnie zostaje przywołyany drugi odbiorca, zaś główny wątek zostaje wtedy wstrzymany na dwie sekundy i przechodzi do trzeciego odbiorcy. Poza tym zauważymy, że wszyscy odbiorcy będą przywoływani dopiero po powrocie metody *sendBroadcast()*.

Aby przetestować odbiorcę komunikatów z opóźnieniem czasowym, dodajmy jego definicję z listingu 14.12 do pliku manifestu widocznego na listingu 14.8.

#### **Listing 14.12.** Definicja odbiorcy treści z opóźnieniem czasowym w pliku manifeście

```
<receiver android:name=".TestTimeDelayReceiver">
    <intent-filter>
        <action android:name="com.androidbook.intents.testbc"/>
    </intent-filter>
</receiver>
```

## Projekt wykorzystujący odbiorców pozaprocesowych

Sednem całej koncepcji nadawania komunikatu jest umożliwienie jego odczytania i przetwarzania obcemu procesowi, który nie jest tożsamy z procesem klienckim. Wobec tego spróbujmy utworzyć kolejny plik *.apk* oraz zarejestrować w tym pakiecie odbiorcę, tak aby mógł on zareagować na komunikat o zdarzeniu, nadawany przez kod z [listingu 14.1](#).

Poniżej zamieszczamy listę plików wymaganych do utworzenia takiego niezależnego projektu; jak zwykle możemy skorzystać z adresu URL zamieszczonego na końcu rozdziału i pobrać niezbędne pliki:

- *StandaloneReceiver.java* — prosty odbiorca ([listing 14.13](#)),
- *AndroidManifest.xml* — plik manifest ([listing 14.14](#)),
- *Utils.java* — ten sam plik był wykorzystywany w poprzednim projekcie ([listing 14.4](#)).

Jest to okrojony projekt, niekomunikujący się z żadną aktywnością, w wyniku czego jest on dość zrozumiały i nie wymaga wprowadzania aktywności ani układu graficznego. Na [listingu 14.13](#) widzimy przykładowego odbiorcę stanowiącego część tego niezależnego projektu. Nazwiemy go *StandaloneReceiver*.

**Listing 14.13.** Przykład odbiorcy przebywającego we własnym procesie

---

```
public class StandaloneReceiver extends BroadcastReceiver
{
    private static final String tag = "Niezależny odbiorca";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Utils.logThreadSignature(tag);
        Log.d(tag, "intencja=" + intent);
        String message = intent.getStringExtra("message");
        Log.d(tag, message);
    }
}
```

---

Nie ma tutaj niczego wyjątkowego, utworzyliśmy jedynie standardowego odbiorcę. Plik manifest rejestrujący go jest widoczny na [listingu 14.14](#).

**Listing 14.14.** Plik *AndroidManifest.xml*, w którym został umieszczony jedynie odbiorca komunikatów

---

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.salbcr"
    android:versionCode="1"
    android:versionName="1.0.0">

    <application android:icon="@drawable/icon"
        android:label="Niezależny odbiorca komunikatów">

        <receiver android:name=".StandaloneReceiver">
            <intent-filter>
                <action android:name="com.androidbook.intents.testbc"/>
            </intent-filter>
        </receiver>
    </application>

```

```
</application>
<uses-sdk android:minSdkVersion="3" />
</manifest>
```

Wraz z omówionym przy okazji poprzedniego projektu plikiem *Utils.java* możemy utworzyć i wdrożyć nowy projekt systemu Android. Jeżeli teraz przejdziemy do ekranu widocznego na rysunku 14.1 i klikniemy element menu *Transmituj*, przekonamy się, że nasz niezależny odbiorca wyświetli komunikat w oknie dziennika podobnie do pozostałych odbiorców.

## Używanie powiadomień pochodzących od odbiorcy komunikatów

Odbiorcy komunikatów muszą czasami powiadomić użytkownika o jakimś wydarzeniu lub o bieżącym stanie, a w takim przypadku najlepszym rozwiązańiem jest wykorzystanie ikony powiadomień dostępnej w systemowym pasku powiadomień. W tym podrozdziale wyjaśnimy, jak można utworzyć powiadomienie za pomocą odbiorcy komunikatów, wysłać je i przeglądać poprzez menedżer powiadomień.

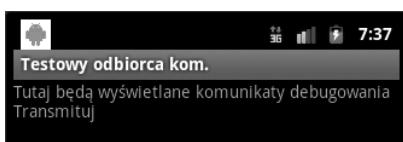
### Monitorowanie powiadomień za pomocą menedżera powiadomień

Ikony powiadomień w Androidzie są wyświetlane jako alerty umieszczone w obszarze powiadomień. Obszar powiadomień jest wąskim paskiem mieszkającym się w szczytowej części wyświetlacza. Został pokazany na rysunku 14.2. Wygląd oraz położenie obszaru powiadomień może się różnić w zależności od tego, czy urządzenie jest tabletem, czy telefonem, a także czasami zależy od wersji Androida.



Rysunek 14.2. Pasek powiadomień w Androidzie

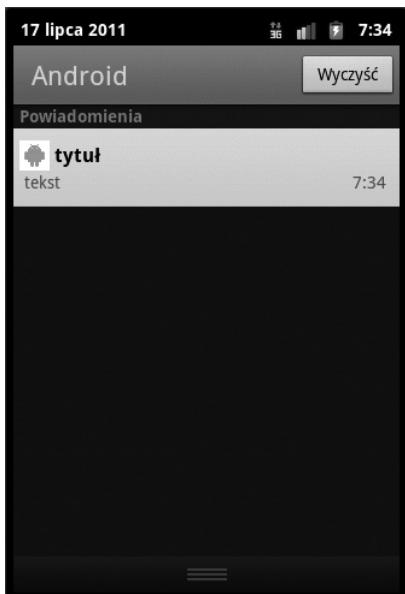
Po wprowadzeniu powiadomienia będzie ono wyświetlane w postaci ikony w obszarze zaprezentowanym na rysunku 14.2. Ikona ta została ukazana na rysunku 14.3.



Rysunek 14.3. Pasek stanu z widoczną ikoną powiadomień

Na rysunku 14.3 widzimy obszar powiadomień, aktywność, a także ikonę powiadomienia. W tym przypadku naszą aktywnością jest aplikacja rozsyłająca komunikaty. Może to być również dobrze inna aktywność, a nawet ekran startowy.

Ikona powiadomień informuje użytkownika, że dzieje się coś, na co należy zwrócić uwagę. Aby ujrzeć właściwe powiadomienie, przytrzymujemy palec na ikonie powiadomienia i rozsuwamy pasek powiadomień (widoczny na rysunku 14.2) jak kurtynę. Zostanie rozwinięty obszar powiadomień, widoczny na rysunku 14.4.



Rysunek 14.4. Rozwinięty widok powiadomień

W rozwiniętym ekranie powiadomień, widocznym na rysunku 14.4, odnajdziemy szczegółowe informacje danego powiadomienia. Możemy również kliknąć dane powiadomienie, aby uruchomić intencję przywołującą aplikację, której to powiadomienie dotyczy. W kolejnym przykładzie wykorzystamy intencję uruchamiającą przeglądarkę.

Na rysunku 14.4 widzimy również, że w tym widoku możemy usuwać powiadomienia.

Ten sam widok szczegółów powiadomień możemy uruchomić z poziomu menu ekranu startowego. Rysunek 14.5 prezentuje menu ekranu startowego, dostępne na emulatorze. W zależności od urządzenia oraz wersji Androida menu ekranu startowego może wyglądać inaczej.

Kliknięcie ikony *Powiadomienia*, widocznej na rysunku 14.5, spowoduje wyświetlenie ekranu powiadomień, znanego z rysunku 14.4.

Przekonajmy się teraz, w jaki sposób możemy wygenerować widoczną na rysunkach 14.3 i 14.4 ikonę powiadomień.

## Wysyłanie powiadomienia

Zaczynajmy. Proces wysyłania powiadomienia składa się z trzech następujących etapów:

1. Utworzenie odpowiedniego powiadomienia.
2. Uzyskanie dostępu do menedżera powiadomień.
3. Przesłanie powiadomienia do menedżera powiadomień.



**Rysunek 14.5.** Element menu Powiadomienia, dostępny z poziomu ekranu startowego

Tworzone powiadomienie musi zawierać następujące elementy:

- wyświetlaną ikonę,
- główny tekst, na przykład „Witaj, świecie!”,
- czas, w którym ma zostać dostarczone.

Po zapełnieniu obiektu powiadomienia tymi elementami uzyskujemy dostęp do kontekstu poprzez zażądanie od kontekstu dostarczenia usługi systemowej, noszącej nazwę `Context`. →`NOTIFICATION_SERVICE`. Po uzyskaniu dostępu do menedżera powiadomień wywołujemy odpowiednią metodę wobec tego obiektu, aby wysłać komunikat.

Na listingu 14.15 został zamieszczony kod odbiorcy komunikatów, przesyłającego powiadomienie widoczne na rysunkach 14.4 i 14.5.

#### **Listing 14.15.** Odbiorca przesyłający powiadomienia

```
public class NotificationReceiver extends BroadcastReceiver
{
    private static final String tag = "Odbiorca powiadomień";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Utils.logThreadSignature(tag);
        Log.d(tag, "intencja=" + intent);
        String message = intent.getStringExtra("message");
        Log.d(tag, message);
        this.sendNotification(context, message);
    }
    private void sendNotification(Context ctx, String message)
    {
```

```
//Uzyskuje dostęp do menedżera powiadomień
String ns = Context.NOTIFICATION_SERVICE;
NotificationManager nm =
    (NotificationManager)ctx.getSystemService(ns);

//Tworzy obiekt powiadomienia
int icon = R.drawable.robot;
CharSequence tickerText = "Witaj";
long when = System.currentTimeMillis();

Notification notification =
    new Notification(icon, tickerText, when);

//Ustanawia widok ContentView za pomocą metody setLatestEventInfo
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.google.com"));
PendingIntent pi = PendingIntent.getActivity(ctx, 0, intent, 0);
notification.setLatestEventInfo(ctx, "tytuł", "tekst", pi);

//Wysyła powiadomienie.
//Pierwszym argumentem jest unikatowy identyfikator tego powiadomienia.
//Identyfikator ten pozwala na późniejsze anulowanie powiadomienia.
nm.notify(1, notification);
}

}
```

W kodzie źródłowym z listingu 14.15 wprowadziliśmy odniesienie do ikony alertu nazwanej *R.drawable.robot*. Możemy utworzyć własną ikonę alertu i wstawić ją do podkatalogu *res/drawable*, przy czym musimy nazwać ją *robot* i wstawić odpowiednie rozszerzenie pliku. Ewentualnie możemy również skorzystać z pliku ZIP zawierającego gotowy projekt (adres URL został podany w podrozdziale „Odnośniki”).

Podczas tworzenia powiadomienia zawierającego podstawowe parametry (ikona, tekst, czas) oraz wysyłania go do menedżera powiadomień możemy odnieść wrażenie, że są one niewystarczające (pierwszy segment listingu 14.15 dotyczy tworzenia powiadomienia). Musimy także zdefiniować dla powiadomienia element zwany widokiem treści za pomocą metody:

*setLatestEventInfo(...)*

Widok treści powiadomienia jest wyświetlany po rozwinięciu paska powiadomień. Jest on widoczny na rysunku 14.4. Przeważnie musi on być obiektem *RemoteViews*. Nie przekazujemy jednak tego widoku bezpośrednio do metody *setLatestEventInfo*. Metoda jest wykorzystywana jako skrót pozwalający na zdefiniowanie standardowego, domyślnego widoku treści, wyświetlającego tytuł i tekst.

Metoda *setLatestEventInfo()* pobiera także argument w postaci oczekującej intencji (zwanej intencją treści), która zostaje uruchomiona po rozwinięciu widoku powiadomień. Spójrzmy jeszcze raz na listing 14.15, aby zobaczyć, jakie parametry zostały przekazane tej metodzie.

Możemy także sami stworzyć zdalny widok i przetworzyć go na widok treści, bez konieczności stosowania metody *setLatestEventInfo()*.

Aby przetworzyć zdalny widok na widok treści powiadomienia, należy wykonać następujące czynności:

1. Utwórz plik układu graficznego.
2. Utwórz obiekt `RemoteViews` za pomocą nazwy pakietu oraz identyfikatora pliku układu graficznego.
3. Wywołaj wobec tego obiektu metody ustanawiające tekst, ikony itd.
4. Wywołaj metodę `setContentView()` na obiekcie powiadomienia przed jego wysłaniem do menedżera powiadomień.

Nie zapominajmy, że w przypadku wersji 2.2 Androida zdalny widok posiada ograniczony zestaw kontrolek:

- `FrameLayout`,
- `LinearLayout`,
- `RelativeLayout`,
- `AnalogClock`,
- `Button`,
- `Chronometer`,
- `ImageButton`,
- `ImageView`,
- `ProgressBar`,
- `TextView`.

W rozdziale 22. dokładnie omówiliśmy proces tworzenia takich zdalnych widoków, ponieważ widżety ekranu startowego zasadniczo są widokami tego typu. Z kolei rozdział 31. zawiera listę obiektów `RemoteViews` zaktualizowaną dla wersji 2.3 i 3.0 Androida.

Kod z listingu 14.15 generuje powiadomienie i wykorzystuje metodę `setLatestEventInfo()` ustanowienia niejawnego widoku treści (za pomocą tytułu i tekstu) oraz uruchamianej intencji (w naszym przypadku jest to intencja przeglądarki).

## Długoterminowi odbiorcy komunikatów i usługi

Dotychczas rozważaliśmy optymistyczną wersję wydarzeń, wedle której przetwarzanie odbiorców komunikatów zajmuje nie dłużej niż dziesięć sekund. Okazuje się, że zadanie nieco się komplikuje, gdy chcemy wykonywać zadania zajmujące więcej czasu.

Aby zrozumieć, dlaczego tak się dzieje, przyjrzyjmy się pokrótko kilku faktom dotyczącym odbiorców komunikatów:

- Podobnie jak pozostałe składniki procesu w Androidzie, odbiorcy komunikatów działają w głównym wątku.
- Wstrzymywanie kodu w odbiorcy komunikatów powoduje również wstrzymanie głównego wątku i wystąpienie komunikatu ANR.
- Limit czasu w odbiorcy komunikatów wynosi 10 sekund w porównaniu do limitu wynoszącego 5 sekund dla aktywności. Restrykcje są, jak widać, nieco łagodniejsze, co nie zmienia faktu, że nadal mamy do czynienia z ramami czasowymi.

- Proces przechowujący odbiorcę komunikatów będzie istniał jedynie przez czas działania tego odbiorcy. Inaczej mówiąc, proces nie będzie już obecny po powrocie metody `onReceive()` odbiorcy komunikatów. Oczywiście, zakładamy w tym momencie, że proces będzie zawierał tylko odbiorcę komunikatów. Jeżeli zawiera on również inne uruchomione składniki, takie jak aktywności lub usługi, brane są również pod uwagę ich cykle życia.
- W przeciwnieństwie do procesu usługi, proces odbiorcy komunikatów nie jest ponownie uruchamiany.
- Jeżeli odbiorca komunikatów rozpoczął oddzielny wątek i wraca do wątku głównego, Android automatycznie określi, że operacja została zakończona i zamknie proces, nawet jeśli znajdują się tam inne działające wątki, które w konsekwencji zostaną gwałtownie zamknięte.
- Podczas przywoływania usługi wysyłającej komunikat Android wchodzi częściowo w stan blokady przechodzenia w stan zatrzymania i wychodzi z tej blokady w momencie zakończenia usługi wysyłającej komunikat w wątku głównym. Blokada przechodzenia w stan zatrzymania (ang. *wakelock*) jest mechanizmem oraz interfejsem API uniemożliwiającym urządzeniu przechodzenie w stan wstrzymania (uśpienia) lub — jeśli urządzenie znajduje się w stanie wstrzymania — przechodzenie w stan aktywności (wybudzenia).

Znając już te fakty, zastanówmy się, w jaki sposób można doprowadzić do wykonywania dłuższych operacji w odpowiedzi na nadany komunikat.

## Protokół długoterminowego odbiorcy komunikatów

Odpowiedź leży w rozwiązyaniu następujących problemów:

- Będziemy koniecznie musieli uruchomić oddzielny wątek, aby wątek główny mógł działać bez ryzyka wyświetlenia wiadomości ANR.
- Aby uniemożliwić systemowi zamknięcie procesu, a w konsekwencji wątku roboczego, musimy przekazać Androidowi informację, że proces ten zawiera składnik posiadający cykl życia, na przykład usługę. Musimy więc ją utworzyć i uruchomić. Sama usługa nie może wykonywać operacji trwających dłużej niż 5 sekund, ponieważ będzie się znajdowała w głównym wątku, a zatem będzie musiała uruchomić wątek roboczy i pozwolić wątkowi głównemu dalej działać.
- Przez okres przetwarzania wątku roboczego musimy wykorzystać częściową blokadę przechodzenia w stan uśpienia, aby urządzenie nie przeszło w stan wstrzymania. Taka częściowa blokada umożliwia urządzeniu przetwarzanie kodu bez konieczności włączania wyświetlacza itd., co pozwala na oszczędność energii baterii.
- Wspomniana częściowa blokada musi zostać umieszczona w głównej części kodu odbiorcy; w przeciwnym wypadku nie zacznie działać w odpowiednim momencie. Na przykład nie możemy tego kodu umieścić w usłudze, ponieważ czas pomiędzy wywołaniem metody `startService()` przez odbiorcę komunikatów a wywołaniem metody `onStartCommand()` usługi rozpoczynającej działanie może się okazać zbyt krótki.
- Ponieważ tworzymy usługę, z powodu ograniczeń pamięci może ona zostać zamknięta i ponownie przywrócona. Jeżeli tak się stanie, będzie trzeba znowu wprowadzić blokadę przechodzenia w stan zatrzymania.

- Kiedy wątek roboczy rozpoczęty przez metodę `onStartCommand()` wykona swoją pracę, musi wymusić zatrzymanie usługi w taki sposób, żeby została zamknięta przez system i nie musiała już się pojawiać.
- Możliwe jest również wystąpienie większej liczby nadawanych komunikatów. Mając to na uwadze, musimy ostrożnie kontrolować liczbę powstających wątków roboczych.

W zgodzie z powyższymi informacjami zalecany protokół przedłużenia czasu trwania odbiorcy komunikatów wygląda następująco:

1. Wprowadź (statyczną) częściową blokadę przechodzenia w stan uśpienia w metodzie `onReceive()` odbiorcy komunikatów. Musi być ona statyczna, gdyż w ten sposób umożliwia komunikację pomiędzy odbiorcą a usługą. Nie ma innej możliwości przekazania usłudze odniesienia do tej blokady, ponieważ jest ona przywoływana za pomocą domyślnego konstruktora, który nie przyjmuje żadnych parametrów.
2. Uruchom usługę lokalną, dzięki czemu proces nie zostanie zamknięty.
3. Aktywuj w usłudze wątek roboczy, aby wykonywał całą pracę. Nie wykonuj operacji w metodzie `onStart()` usługi. Jeżeli tak zrobisz, po prostu wstrzymasz wątek główny.
4. Po zakończeniu działania wątku roboczego zatrzymaj usługę — albo bezpośrednio, albo za pomocą procedury obsługi.
5. Zapewnij uruchomienie przez usługę statycznej blokady przechodzenia w stan zatrzymania. Przypominamy, że ta statyczna blokada stanowi jedyny sposób komunikowania się usługi z obiektem ją wywołującym, w tym przypadku usługą nadającą komunikaty, ponieważ nie ma innej możliwości przekazania usłudze odniesienia do blokady.

## Klasa IntentService

Aby umożliwić tworzenie usług, które nie będą zatrzymywały głównego wątku, system Android zapewnia implementację lokalnej usługi zwanej `IntentService`. Usługa ta przenosi operacje do wątku roboczego, dzięki czemu główny wątek zostaje odciążony po określeniu zakresu zadań wątku pobocznego. Po wywołaniu metody `startService()` wobec usługi `IntentService` klasa ta kolejkuje żądanie w wątku pobocznym za pomocą zapętlenia oraz procedury obsługi, dzięki czemu rzeczywistą pracę wykonuje metoda wywodząca się z tej usługi.

Dokumentacja interfejsu API definiuje klasę `IntentService` w następujący sposób:

*IntentService jest bazową klasą usług przetwarzających zapytania asynchroniczne (wyrażane w postaci intencji) wysypane na żądanie. Klienci wysyłają zapytania za pomocą wywołań metody `startService(Intent)`. W razie potrzeby zostaje uruchomiona usługa, która z kolei przetwarza każdą intencję w wątku roboczym. Gdy ten wątek zakończy działanie, usługa zostaje zatrzymana. Taki wzorzec „procesora kolejki operacji” jest powszechnie stosowany do odciążenia głównego wątku aplikacji z różnorodnych zadań. Klasa `IntentService` została stworzona w celu uproszczenia tego wzorca.*

*Zapewnia usługę wykorzystywanych w tym przypadku mechanizmów. Aby z niej skorzystać, należy rozszerzyć klasę `IntentService` i zaimplementować metodę `onHandleIntent()`. Klasa ta będzie otrzymywała intencje, uruchomi wątek roboczy oraz w odpowiednim momencie zatrzyma usługę. Wszystkie żądania są przetwarzane w pojedynczym wątku roboczym — mogą one trwać dowolnie długo (i nie będą blokować głównej pętli aplikacji), ale w danym momencie będzie obsługiwane tylko jedno żądanie.*

Możemy w prosty sposób zademonstrować koncepcję klasy IntentService za pomocą przykładu widocznego na listingu 14.16. Rozszerzamy klasę IntentService i wprowadzamy wszelkie wymagane zmiany w metodzie onHandleIntent().

#### **Listing 14.16.** Stosowanie klasy IntentService

---

```
public class MyService extends IntentService
{
    protected abstract void onHandleIntent(Intent intent)
    {
        Utils.logThreadSignature("MyService");
        //wykonuje pracę w tym wątku pobocznym
        //i powraca
    }
}
```

---

Po utworzeniu usługi tego typu możemy ją zarejestrować w pliku manifeście i wykorzystać kod klienta do przywołania usługi w postaci context.startService(new Intent(MyService.class)). Tego typu przywołanie poskutkuje wywołaniem metody onHandleIntent(), widocznej na listingu 14.16.

Warto zauważyc, że metoda logThreadSignature() wyświetli identyfikator wątku roboczego, a nie głównego (pamiętajmy, że mamy tu do czynienia wyłącznie z pseudokodem; wkrótce jednak zaprezentujemy rzeczywisty przykład).

### **Kod źródłowy klasy IntentService**

W rozdziale 13. zajmowaliśmy się zagadnieniami wątku głównego i procedur obsługi. W tym kontekście warto przestudiować kod źródłowy klasy IntentService, aby zrozumieć, w jaki sposób wspomniane elementy współdziałają wraz z długoterminową usługą wykorzystującą wątek roboczy. Przyjrzymy się teraz kodowi źródłowemu klasy IntentService (powielonemu z kodu źródłowego Androida), zamieszczonemu na listingu 14.17.

#### **Listing 14.17.** Kod źródłowy klasy IntentService

---

```
public abstract class IntentService extends Service {
    private volatile Looper mServiceLooper;
    private volatile ServiceHandler mServiceHandler;
    private String mName;

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
        @Override
        public void handleMessage(Message msg) {
            onHandleIntent((Intent)msg.obj);
            stopSelf(msg.arg1);
        }
    }

    public IntentService(String name) {
        super();
    }
}
```

```

        mName = name;
    }
    @Override
    public void onCreate() {
        super.onCreate();
        HandlerThread thread =
            new HandlerThread("IntentService[" + mName + "]");
        thread.start();

        mServiceLooper = thread.getLooper();
        mServiceHandler = new ServiceHandler(mServiceLooper);
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        msg.obj = intent;
        mServiceHandler.sendMessage(msg);
    }
    @Override
    public void onDestroy() {
        mServiceLooper.quit();
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    protected abstract void onHandleIntent(Intent intent);
}

```

---

Poniżej objaśniliśmy działanie powyższego kodu:

1. Tworzymy osobny wątek roboczy w metodzie `onCreate()` usługi. Zazwyczaj uruchamiamy wątki robocze w metodzie `onStartCommand()` usługi. W wyniku tego powstaloby jednak wiele wątków roboczych, po jednym dla każdej metody `startService`. W przypadku klasy `IntentService` wystarczy jeden wątek roboczy obsługujący wszystkie wywołania metody `startService`, zatem konfigurujemy wątek roboczy w metodzie `onCreate`, która jest wywoływana tylko raz.
2. Konfigurujemy zapętlenie (a tym samym kolejkę otrzymywanych i wysyłanych komunikatów) w wątku roboczym. W ten sposób jeden wątek roboczy może odpowiadać na wiele komunikatów jeden po drugim, nie musimy więc tworzyć osobnego wątku dla każdego żądania.
3. Ustanawiamy uchwyt wobec wątku roboczego, dzięki czemu główny wątek usługi może przekazać komunikat za pomocą procedury obsługi. Wątek roboczy jest nam potrzebny, ponieważ za każdym razem, gdy klient stosuje metodę `startService()`, jej wywołanie przechodzi do głównego wątku klasy `IntentService`, a nie należy przetrzymywać głównego wątku tej klasy. Potrzebny jest nam mechanizm kolejkowania tych żądań, dzięki czemu wątek roboczy będzie mógł je przetwarzać, gdy tylko będzie miał taką sposobność. Możemy tego dokonać poprzez umieszczenie w głównym wątku procedury obsługi dla wątku roboczego. Zwrócić uwagę na metodę `onStart()` uruchomioną w głównym wątku. Jeżeli chcemy to sprawdzić, wystarczy przesłonić tę

metodę i wywołać jej nadziedną klasę przy jednoczesnym wyświetlaniu w dzienniku sygnatury wątku. Przekonamy się, że metoda `onStart()` działa w głównym wątku, natomiast metoda `onHandleMessage()` — w drugoplanowym wątku roboczym.

4. W końcu, po powrocie metody `onHandleIntent()`, procedura obsługi wywołuje metodę `stopSelf()` wobec usługi. W przypadku braku oczekujących komunikatów metoda ta skutecznie zatrzyma usługę. Metoda `stopSelf()` jest zliczana referencyjnie. Oznacza to, że jeśli nawet wywołamy ją wielokrotnie, musi istnieć taka sama liczba wywołań metody `startService`. Dlatego właśnie możemy wywoływać metodę `stopSelf()` za każdym razem, gdy zostanie obsłużone wywołanie metody `startService`.

## Rozszerzanie klasy IntentService na odbiorcę komunikatów

Z perspektywy odbiorcy komunikatów klasa `IntentService` stanowi świetne rozwiązanie. Pozwala nam uruchamiać długo wykonujący się kod bez obawy wstrzymywania głównego wątku. Czy możemy więc wykorzystywać klasę `IntentService` na potrzeby takich dłużej trwających operacji? I tak, i nie.

Tak, ponieważ klasa `IntentService` wykonuje dwie czynności: po pierwsze, przechowuje działający proces, ponieważ jest usługą. Po drugie, pozwala działać głównemu wątkowi, dzięki czemu unikamy komunikatu ANR.

Aby zrozumieć odpowiedź „nie”, musimy dokładniej zastanowić się nad pojęciem blokady przechodzenia w stan zatrzymania. Przy przywołaniu odbiorcy komunikatów, zwłaszcza za pomocą menedżera alarmów, urządzenie nie musi być włączone. Zatem menedżer ten częściowo włącza urządzenie (jedynie na tyle, aby móc przetworzyć kod bez użycia interfejsu użytkownika) poprzez wywołanie menedżera zasilania i zażądanie blokady. Blokada ta zostaje zwolniona w momencie powrotu odbiorcy komunikatów.

W ten sposób wywołanie usługi `IntentService` pozostaje bez blokady przechodzenia w stan zatrzymania, więc urządzenie może przejść w stan uśpienia jeszcze przed uruchomieniem właściwego kodu. Jednak klasa `IntentService`, będąc ogólnym rozszerzeniem usługi, nie otrzymuje dostępu do blokady.

Potrzebujemy więc dodatkowego rozszerzenia możliwości klasy `IntentService`. Potrzebujemy abstrakcji.

Mark Murphy stworzył odmianę klasy `IntentService`, nazwaną `WakefulIntentService`, która utrzymując semantykę działania pierwotnej wersji usługi, dodatkowo otrzymuje dostęp do blokady przechodzenia w stan zatrzymania i poprawnie ją zwalnia po spełnieniu pewnych warunków. Implementację tej klasy znajdziemy pod adresem <http://github.com/commonsguy/cwac-wakeful>.

## Abstrakcja długoterminowej usługi wysyłającej komunikaty

Usługa `WakefulIntentService` jest świetną klasą abstrakcyjną. Chcemy jednak pójść krok dalej i sprawić, żeby klasa ta była równie rozszerzalna jak usługa `IntentService` na listingu 14.14 oraz żeby posiadała wszystkie funkcje tej usługi, ale dodatkowo miała następujące zalety:

1. Uzyskiwanie i zwalnianie blokady przechodzenia w stan zatrzymania (podobnie jak w przypadku standardowej klasy `WakefulIntentService`).

2. Przekazanie oryginalnej intencji (która była pierwotnie przekazana odbiorcy komunikatów) przesłoniętej metodzie `onHandleIntent`. Pozwalałoby nam to w dużym stopniu ukryć dostawcę komunikatów.
3. Przetworzenie ponownie uruchomionej usługi.
4. Ujednolicony sposób przetwarzania blokady przechodzenia w stan zatrzymania w przypadku wielu odbiorców komunikatów oraz wielu usług w tym samym procesie

Nazwiemy tę abstrakcyjną klasę `ALongRunningNonStickyBroadcastService`. Jak sama nazwa wskazuje, chcemy, aby ta usługa pozwalała na przetwarzanie dłużej trwających operacji. Będzie ona również specyficznie dostosowana do odbiorców komunikatów. Usługa ta będzie także nietrwała (omówimy tę koncepcję później, w skrócie jednak pojęcie to oznacza, że system nie uruchomi usługi, jeśli nie będzie żadnego komunikatu w kolejce). Usługa ta rozszerzy klasę `IntentService`, aby uzyskać jej własności, a także przesłoni metodę `onHandleIntent`.

Połączysz powyższe wymagania, kod abstrakcyjnej usługi `ALongRunningNonStickyBroadcastService` przybierze wygląd zaprezentowany na listingu 14.18.

#### **Listing 14.18.** Koncepcja abstrakcyjnej, długoterminowej usługi

---

```
public abstract class ALongRunningNonStickyBroadcastService
extends IntentService
{
    ...inne szczegóły implementacji
    protected abstract void
        handleBroadcastIntent(Intent broadcastIntent);
    ...inne szczegóły implementacji
}
```

---

Szczegóły implementacyjne dotyczące tej usługi są dość zaawansowane i zajmiemy się nimi w dalszej części rozdziału, najpierw musimy jednak wyjaśnić, dlaczego wybraliśmy taki rodzaj usługi. Chcemy najpierw udowodnić użyteczność i prostotę tego rozwiązania.

Po utworzeniu tej abstrakcyjnej klasy można pozmieniać przykładowy kod usługi `MyService` z listingu 14.16 i uzyskać kod przedstawiony na listingu 14.19.

#### **Listing 14.19.** Przykładowe zastosowanie długoterminowej usługi

---

```
public class MyService extends ALongRunningNonStickyBroadcastService
{
    protected abstract void handleBroadcastIntent(Intent broadcastIntent)
    {
        Utils.logThreadSignature("MyService");
        //tutaj są przeprowadzane operacje
        //i następuje powrót
    }
}
```

---

Jak widać, możemy rozszerzyć tę nową, długoterminową klasę usługi (podobnie jak klasy `IntentService` oraz `WakefulIntentService`) oraz przesłonić pojedynczą metodę, a także nic nie zmieniać lub wprowadzać bardzo niewielkie zmiany w odbiorcy komunikatów. Wszystkie zadania zostaną wykonane w wątku roboczym (dzięki klasie `IntentService`) bez obawy o zablokowanie głównego wątku.

Listing 14.19 stanowi prosty przykład demonstrujący omawianą koncepcję. Przejdźmy teraz do bardziej złożonej implementacji, uwzględniającej długoterminową usługę, która działa przez 60 sekund w odpowiedzi na nadawany komunikat (chcemy dać dowód, że usługa może działać dłużej niż 10 sekund bez wywoływanego komunikatu ANR). Nazwiemy tę usługę `Test60SecBCRService` (BCR jest skrótem wyrażenia *Broadcast Receiver*, czyli odbiorca komunikatów), a jej implementacja jest widoczna na listingu 14.20.

**Listing 14.20.** Usługa `Test60SecBCRService`

---

```
public class Test60SecBCRService
extends ALongRunningNonStickyBroadcastService
{
    public static String tag = "Test60SecBCRService";
    //Wymagane przez klasę IntentService do przekazania nazwy klasy.
    public Test60SecBCRService(){
        super("com.androidbook.service.Test60SecBCRService");
    }

    /*
     * Wykonuje długoterminowe operacje wewnętrz tej metody.
     * Są one przeprowadzane w oddzielnym wątku.
     */
    @Override
    protected void handleBroadcastIntent(Intent broadcastIntent)
    {
        Utils.logThreadSignature(tag);
        Log.d(tag, "Wstrzymanie na 60 sekund");
        Utils.sleepForInSecs(60);
        String message =
            broadcastIntent.getStringExtra("message");
        Log.d(tag, "Praca zakończona");
        Log.d(tag, message);
    }
}
```

---

Jak widać, powyższy kod skutecznie symuluje pracę wykonywaną przez 60 sekund i jednocześnie nie powoduje wyświetlenia komunikatu ANR. Być może Czytelnik się teraz zastanawia, dlaczego nie możemy skompilować tej klasy i dlaczego nie pokazaliśmy jeszcze implementacji abstrakcyjnej klasy długoterminowej usługi. Warto jednak poczekać i najpierw dokładnie zrozumieć wszystkie elementy tego przykładu. W trakcie objaśniania zaprezentujemy kod implementacji wszystkich niezbędnych klas. Poza tym w dalszej części rozdziału umieściliśmy dokładne informacje dotyczące komplikacji tego przykładu, a na jego końcu zamieściliśmy adres URL, z którego możemy pobrać cały projekt.

## Długoterminowy odbiorca komunikatów

Po utworzeniu długoterminowej usługi widocznej na listingu 14.20 niezbędne okaże się wprowadzenie możliwości jej przywołania z poziomu odbiorcy komunikatów.

Pierwszym celem długoterminowego odbiorcy komunikatów jest przekazanie pracy długoterminowej usłudze. W tym celu odbiorca ten potrzebuje nazwy klasy tej usługi, aby mógł ją przywołać.

Drugim celem długoterminowego odbiorcy jest wprowadzenie blokady przechodzenia w stan zatrzymania, gdyż w ten sposób zapewniamy bezustanne przetwarzanie kodu po powrocie odbiorcy.

Trzecim celem jest przeniesienie do usługi oryginalnej intencji, przywołanej dla tego odbiorcy. Dokonamy tego poprzez zdefiniowanie oryginalnej intencji jako typ `Parcelable` w jej dodatkowej wartości. Wartość ta będzie nosić nazwę `original_intent`. Następnie długoterminowa usługa wydobywa wartość `original_intent` i przekazuje ją swojej przesłoniętej metodzie (zobaczmy to później, podczas implementacji długoterminowej usługi). Dzięki takiemu rozwiązaniu odnosimy więc wrażenie, że długoterminowa usługa jest istotnie rozszerzeniem odbiorcy komunikatów.

Chociaż możemy zaprogramować każdego długoterminowego odbiorcę, żeby za każdym razem wykonywał te czynności, będzie lepiej, jeśli je wyodrębnimy i utworzymy klasę bazową. Taki abstrakcyjny, długoterminowy odbiorca będzie wykorzystywał wtedy pochodną klasę do zapewnienia nazwy długoterminowej usługi (ang. *Long-Running Service* — LRS) poprzez abstrakcyjną metodę, nazwaną `getLRSClass()`.

Zanim przejdziemy do implementacji tej abstrakcyjnej klasy, musimy przez chwilę zastanowić się nad obranym kierunkiem w przypadku blokad przechodzenia w stan zatrzymania. Blokady te muszą zostać skoordynowane pomiędzy odbiorcą komunikatów a odpowiadającą mu usługą, która jest przez nie wywoływana. Chociaż sam pomysł jest banalny, w rzeczywistej implementacji musimy zatroszczyć się o wiele obszarów oraz warunków, które muszą zostać spełnione, aby ten proces nastąpił. Wyodrębniliśmy więc blokadę za pomocą pojęcia nazwanego `LightedGreenRoom`. Zaprezentujemy później tę klasę, na razie jednak możemy ją traktować jako blokadę przechodzenia w stan zatrzymania, którą możemy włączać i wyłączać.

Po uwzględnieniu tych wszystkich uwarunkowań kod źródłowy implementacji abstrakcyjnej klasy `ALongRunningReceiver` będzie wyglądał tak jak przedstawiono na listingu 14.21.

#### **Listing 14.21.** Klasa `ALongRunningReceiver`

```
public abstract class ALongRunningReceiver
extends BroadcastReceiver
{
    private static final String tag = "ALongRunningReceiver";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Log.d(tag, "Odbiorca uruchomiony");
        //Klasa LightedGreenRoom wyodrębnia blokadę
        //przechodzenia w stan zatrzymania w Androidzie,
        //aby zatrzymywać urządzenie częściowo uruchomione.
        //W skrócie — mamy tu do czynienia z czynnością równoważną włączeniu
        //lub nabyciu blokady.
        LightedGreenRoom.setup(context);
        startService(context,intent);
        Log.d(tag, "Odbiorca zakonczony");
    }
    private void startService(Context context, Intent intent)
    {
        Intent serviceIntent = new Intent(context,getLRSClass());
        serviceIntent.putExtra("original_intent", intent);
    }
}
```

```
        context.startService(serviceIntent);
    }
    /*
     * Przesłaniamy tę metodę, aby powrócić do
     * obiektu „klasy” należącego do
     * nietrwałej usługi.
     */
    public abstract Class getLRSClass();
}
```

---

Po utworzeniu tej abstrakcyjnej klasy potrzebny będzie jeszcze odbiorca ściśle współpracujący z długoterminową (60-sekundową) usługą, widoczną na listingu 14.16. Kod takiego typu odbiorcy jest widoczny na listingu 14.22.

#### **Listing 14.22.** Przykładowy długoterminowy odbiorca komunikatów, nazwany Test60SecBCR

---

```
public class Test60SecBCR
extends ALongRunningReceiver
{
    @Override
    public Class getLRSClass()
    {
        Utils.logThreadSignature("Test60SecBCR");
        return Test60SecBCRService.class;
    }
}
```

---

Podobnie jak abstrakcja usługi z listingów 14.19 i 14.20, kod na listingu 14.22 stanowi klasę abstrakcyjną pozwalającą na utworzenie odbiorcy komunikatów. Klasa ta rozpoczyna usługę wskazywaną przez wartość zwracaną w metodzie `getLRSClass()`.

Do tej pory wyjaśniliśmy, dlaczego potrzebne są nam dwie abstrakcyjne klasy do zaimplementowania długoterminowych usług przywoływanych przez odbiorcę komunikatów, mianowicie:

- `ALongRunningNonStickyBroadcastService`,
- `ALongRunningReceiver`.

Odkładaliśmy jednak moment zademonstrowania implementacji każdej z tych klas z powodu ich złożoności. Nie pokazaliśmy też jeszcze implementacji wspólnej klasy, `LightedGreenRoom`, wykorzystywanej przez obydwie wymienione klasy. Dotarliśmy w końcu do miejsca, z którego możemy zaprezentować kody źródłowe tych dwóch wspomnianych klas. Rozpoczniemy jednak od ich wspólnej klasy — `LightedGreenRoom`.

## **Wyodrębnianie blokady przechodzenia w stan zatrzymania za pomocą klasy LightedGreenRoom**

Jak już wcześniej wspomnieliśmy, głównym zadaniem abstrakcyjnej klasy `LightedGreenRoom` jest uproszczenie interakcji z blokadą przechodzenia w stan zatrzymania. Blokada ta z kolei pozwala na przetrzymywanie urządzenia w stanie aktywności. Na listingu 14.23 pokazujemy, w jaki sposób — zgodnie z zestawem SDK — jest zazwyczaj wykorzystywana typowa blokada przechodzenia w stan zatrzymania.

**Listing 14.23.** Interfejs API blokady przechodzenia w stan zatrzymania

```
//Uzyskuje dostęp do usługi menedżera zasilania
PowerManager pm =
    (PowerManager)inCtx.getSystemService(Context.POWER_SERVICE);

//Kontaktuje się z blokadą
PowerManager.WakeLock wl =
    pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, tag);

//Uzyskuje blokadę
wl.acquire();

//Wykonuje jakieś operacje
//W trakcie wykonywania operacji urządzenie będzie częściowo włączone

//Zwala blokadę
wl.release();
```

Za pomocą tego rodzaju oddziaływania odbiorca komunikatów powinien mieć możliwość uzyskania blokady, a po zakończeniu działania długoterminowej usługi blokada ta musi zostać zwolniona. Jednak nie istnieje skuteczna metoda przekazania usłudze tej zmiennej blokady przechodzenia w stan zatrzymania z poziomu odbiorcy komunikatów. Jedynym rozwiązaniem, dzięki któremu usługa ta uzyska informację o obecności blokady budzenia, jest użycie zmiennej statycznej lub zmiennej na poziomie aplikacji.

Kolejną trudnością pojawiającą się podczas uzyskiwania i zwalniania blokady przechodzenia w stan zatrzymania jest zliczanie referencji. Jeśli więc odbiorca komunikatów zostanie kilkakrotnie przywołany, to jeżeli te wywołania będą się na siebie nakładaly, pojawi się jednocześnie kilka wywołań żądających nałożenia blokady. Analogicznie — pojawi się też kilka wywołań żądających zwolnienia tej blokady. Jeżeli liczba wywołań nakładania i zwalniania blokady nie będzie taka sama, skończy się na tym, że w najgorszym wypadku urządzenie będzie włączone o wiele dłużej, niż potrzeba. Ponadto, jeżeli usługa nie będzie już nam potrzebna i będzie uruchomiony proces oczyszczania pamięci, to w przypadku różnic w liczbie blokad nastąpi wyświetlenie wyjątku wykonawczego w oknie *LogCat*.

Problemy te zachęciły nas do możliwie jak najskuteczniejszego wyodrębnienia blokady przechodzenia w stan zatrzymania w celu zapewnienia jej właściwego działania.

**Uwaga!**

Skoro już jesteśmy świadomi problemów oraz potrzeby posiadania takich blokad, zachęcamy do eksperymentowania z klasą `LightedGreenRoom` oraz zastępowania jej inną klasą, jeśli takie rozwiązanie okaże się prostsze. Powyższe zadanie ma nas upewnić, że klasa `LightedGreenRoom` nie jest magicznym bytem i w swej istocie jest bardzo nieskomplikowana.

Wyjaśnimy teraz, co nas skłoniło do uznania klasy `LightedGreenRoom` za odpowiednią blokadę przechodzenia w stan zatrzymania.

## Oświetlony zielony pokój<sup>1</sup>

Rozpoczniemy od zielonego pokoju, który może być odwiedzany przez gości. Na początku w pokoju jest ciemno, a pierwszy gość, który wejdzie, zapala światło. Kolejni goście nie mają już wpływu na światło, jeśli zostało zapalone. Ostatni gość przy wyjściu zgasi światło. Pokój jest nazywany zielonym, ponieważ w wydajny sposób gospodaruje energią. Metody pozwalające na wchodzenie i wychodzenie z pokoju muszą być zsynchronizowane, aby kontrolować swoje stany, ponieważ mogą być wywoływane pomiędzy wieloma wątkami.

Czym więc jest oświetlony zielony pokój? W przeciwnieństwie do zwykłego zielonego pokoju, w którym początkowo światła są zgaszone, w oświetlonym pokoju od początku jest jasno, jeszcze przed przybyciem pierwszego gościa. Możemy założyć, że gdyby światło było zgaszone, odwiedzająca osoba nie mogłaby odnaleźć drogi do pokoju. Jest to związane z faktem, że w wyłączonym urządzeniu żadna usługa nie może zostać uruchomiona. Ciągle jednak ostatni wychodzący gość będzie gasił światło przy wyjściu. Taki mechanizm okazuje się przydatny dla odbiorcy komunikatów, ponieważ musi najpierw zapalić światło, a następnie przenieść usługę.

Uruchomienie usługi jest równoważne wejściu pierwszego gościa. Zatrzymanie usługi to wyjście gościa z pomieszczenia. Zwróćmy uwagę, że musimy rozróżnić pojęcia **utworzenia** usługi od jej **uruchomienia**. Utworzenie i zamknięcie usługi występuje tylko raz w jej cyklu życia, natomiast jej uruchamianie i zatrzymywanie może występować wielokrotnie.

Może się pojawić, i zazwyczaj się pojawia, opóźnienie pomiędzy konfiguracją blokady przechodzenia urządzenia w stan zatrzymania (oświetlonego zielonego pokoju) w odbiorcy a uruchomieniem usługi, a dokładniej wywołaniem metody `onStartCommand` (wkroczeniem pierwszego gościa do pokoju).

Ponieważ obiekt `wakeLock` jest zliczany referencyjnie, w przypadku gdy usługa zostanie zamknięta z powodu ograniczenia pamięci, chcielibyśmy jawnie zwolnić blokady. Gdybyśmy chcieli wykorzystać ten sam oświetlony zielony pokój do obsługi wielu usług, możemy zechcieć wy śledzić ostatnią usługę, która zostanie zamknięta, i zwolnić blokady tuż po jej zakończeniu.

W tym celu utworzymy klienta. Każda usługa będzie rejestrowana z oświetlonym zielonym pokojem jako jej klient, dzięki czemu będzie działała metoda jej zamiany.

A przede wszystkim musimy śledzić „wejścia” i „wyjścia” każdej metody `startService`.

## Implementacja oświetlonego zielonego pokoju

Po połączeniu wszystkich koncepcji wymienionych w poprzednim punkcie implementacja oświetlonego zielonego pokoju będzie wyglądała jak na listingu 14.24. Pragniemy zauważyc, że spisywała się ona dobrze w naszym ograniczonym środowisku testowym. Warto jednak trochę eksperymentować i dostosować do własnych potrzeb, ponieważ nie potrafimy przewidzieć każdej okoliczności, jaka może wystąpić w środowisku projektowym Czytelnika (inaczej mówiąc, uznajmy ten przykład za eksperymentalny).

<sup>1</sup> Tytuł podrozdziału odnosi się do nazwy omawianej klasy, która w dosłownym tłumaczeniu oznacza właśnie oświetlony zielony pokój — przyp. tłum.

**Listing 14.24.** Implementacja oświetlonego zielonego pomieszczenia

```

public class LightedGreenRoom
{
    //Znacznik debugowania
    private static String tag="LightedGreenRoom";

    //Zlicza liczbę gości, aby wiedzieć, który jest ostatni.
    //Podczas zamykania usługi zeruje licznik w celu wyczyszczenia pokoju.
    private int count;

    //Potrzebna do utworzenia blokady budzenia
    private Context ctx = null;

    //Nasz przełącznik
    PowerManager.WakeLock wl = null;

    //Obsługa wielu klientów
    private int clientCount = 0;

    /*
     * Oczekujemy, że będzie to klasa singletonowa.
     * Potencjalnie można zrobić prywatny
     * konstruktor.
     */
    public LightedGreenRoom(Context inCtx)
    {
        ctx = inCtx;
        wl = this.createWakeLock(inCtx);
    }

    /*
     * Konfigurowanie zielonego pokoju za pomocą statycznej metody.
     * Musi być wywołana przed wywołaniem innych metod.
     * Jej zadania:
     * 1. Tworzenie wystąpienia obiektu.
     * 2. Wprowadzenie blokady włączającej światła.
     * Założenie:
     * Nie musi być synchronizowana,
     * ponieważ będzie wywoływana z głównego wątku.
     * (Może być błędne. Należy to sprawdzić!!)
     */
    private static LightedGreenRoom s_self = null;

    public static void setup(Context inCtx)
    {
        if (s_self == null)
        {
            Log.d(LightedGreenRoom.tag,"Tworzenie zielonego pokoju i oświetlanie go");
            s_self = new LightedGreenRoom(inCtx);
            s_self.turnOnLights();
        }
    }
}

```

```
        public static boolean isSetup()
{
    return (_self != null) ? true: false;
}

/*
 * Spodziewamy się, że metody „wchodzenia” i „wychodzenia”
 * będą wspólnie wywoływanie.
 *
 * Przy „wejściu” licznik jest zwiększany.
 *
 * Nie włączamy ani nie wyłączamy światła,
 * ponieważ są one już zapalone.
 *
 * Zwiększamy wartość licznika tylko po to,
 * aby wiedzieć, kiedy wyjdzie ostatni gość.
 *
 * Jest to synchronizowana metoda, ponieważ
 * będzie wchodziło i wychodziło wiele wątków.
 *
 */
synchronized public int enter()
{
    count++;
    Log.d(tag, "Nowy gosc: licznik:" + count);
    return count;
}
/*
 * Spodziewamy się, że metody „wchodzenia” i „wychodzenia”
 * będą wspólnie wywoływanie.
 *
 * Przy „wyjściu” zmniejszamy licznik.
 *
 * Jeżeli licznik osiągnie wartość 0, gasimy światła.
 *
 * Jest to metoda synchroniczna, ponieważ
 * wiele wątków będzie wchodziło i wychodziło.
 *
 */
synchronized public int leave()
{
    Log.d(tag, "Opuszczanie pokoju: licznik w momencie wywołania:" + count);
    //Jeżeli wartość licznika już wynosi 0,
    //po prostu wychodzimy.
    if (count == 0)
    {
        Log.w(tag,"Wartosc licznika wynosi zero.");
        return count;
    }
    count--;
    if (count == 0)
```

```
{  
    //Ostatni gość  
    //gasi światła.  
    turnOffLights();  
}  
return count;  
}  
synchronized public int getCount()  
{  
    return count;  
}  
  
/*  
 * Wprowadzamy blokadę, aby zapalić światła.  
 * Od innych synchronizowanych metod zależy, czy zostanie ona  
 * wywołana we właściwym momencie.  
 */  
private void turnOnLights()  
{  
    Log.d(tag, "Zapalanie swiatla. Licznik:" + count);  
    this.wl.acquire();  
}  
  
/*  
 * Zwalnia blokadę, aby zgasić światła.  
 * Od innych synchronizowanych metod zależy, czy zostanie ona  
 * wywołana we właściwym momencie.  
 */  
private void turnOffLights()  
{  
    if (this.wl.isHeld())  
    {  
        Log.d(tag,"Zwalnianie blokady. Nie ma juz gosci");  
        this.wl.release();  
    }  
}  
/*  
 * Standardowy kod służący do utworzenia częściowej blokady budzenia.  
 */  
private PowerManager.WakeLock createWakeLock(Context inCtx)  
{  
    PowerManager pm =  
        (PowerManager)inCtx.getSystemService(Context.POWER_SERVICE);  
  
    PowerManager.WakeLock wl = pm.newWakeLock  
        (PowerManager.PARTIAL_WAKE_LOCK, tag);  
    return wl;  
}  
  
private int registerClient()  
{  
    Utils.logThreadSignature(tag);  
    this.clientCount++;  
    Log.d(tag,"rejestrowanie nowego klienta:licznik:" + clientCount);  
}
```

```
        return clientCount;
    }

private int unRegisterClient()
{
    Utils.logThreadSignature(tag);
    Log.d(tag, "wyrejestrowanie nowego klienta:licznik:" + clientCount);
    if (clientCount == 0)
    {
        Log.w(tag, "Brak Klientow do wyrejestrowania.");
        return 0;
    }
    //Wartość clientCount nie jest równa 0.
    clientCount--;
    if (clientCount == 0)
    {
        emptyTheRoom();
    }
    return clientCount;
}
synchronized public void emptyTheRoom()
{
    Log.d(tag, "Wywoluje do wyczyszczenia pokoju");
    count = 0;
    this.turnOffLights();
}
//*****
//Statyczni członkowie: same metody pomocnicze
//Delegowanie do ukrytego obiektu singletonowego
//*****

public static int s_enter()
{
    assertSetup();
    return s_self.enter();
}
public static int s_leave()
{
    assertSetup();
    return s_self.leave();
}
//Nie wywołujmy tej metody bezpośrednio,
//prawdopodobnie stanie się przestarzała.
//Zamiast tego wywołujmy klienckie metody rejestrowania/wyrejestrowania.
public static void ds_emptyTheRoom()
{
    assertSetup();
    s_self.emptyTheRoom();
    return;
}
public static void s_registerClient()
{
    assertSetup();
    s_self.registerClient();
    return;
}
```

```

public static void s_unregisterClient()
{
    assertSetup();
    s_self.unregisterClient();
    return;
}
private static void assertSetup()
{
    if (LightedGreenRoom.s_self == null)
    {
        Log.w(LightedGreenRoom.tag, "Musimy najpierw wywolac konfigurator");
        throw new RuntimeException("Musimy najpierw skonfigurowac obiekt GreenRoom");
    }
}

```

---

Rozsądnym rozwiążaniem w celu umożliwiania komunikacji pomiędzy odbiorcą komunikatów i usługą jest zastosowanie zmiennej statycznej. W naszym przykładowym kodzie obiekt `wakelock` nie jest statyczny, ale statyczne jest całe wystąpienie klasy `LightedGreenRoom`, jednak każda inna zmienna umieszczona wewnątrz klasy `LightedGreenRoom` pozostaje lokalna i niestatyczna.

Każda publiczna metoda klasy `LightedGreenRoom` dla naszej wygody staje się również statyczna. Możemy zamiast tego pozbyć się statycznych metod i bezpośrednio wywołać pojedyncze wystąpienie klasy `LightedGreenRoom`.

## Implementacja długoterminowej usługi

Po zaimplementowaniu klasy `LightedGreenRoom` jesteśmy niemal gotowi do zaprezentowania abstrakcyjnej, długoterminowej usługi. Musimy jednak najpierw wyjaśnić zagadnienia czasu życia usługi oraz jego związek z implementacją metody `onStartCommand`. To właśnie ta metoda ostatecznie obsługuje rozpoczęcie wątku roboczego i semantykę usługi.

Wiemy, że odbiorca komunikatów przywołuje usługę za pomocą metody `startService`, w wyniku której z kolei zostaje wywołana metoda `onStartCommand` usługi. Czas życia usługi jest zależny od wartości zwracanych przez tę metodę.

Aby zrozumieć, co się dzieje we wnętrzu tej metody, musimy znać szczegółowe informacje dotyczące natury usług lokalnych. W rozdziale 11. podaliśmy podstawowe informacje na temat usług lokalnych, obecnie musimy przyjrzeć się im nieco uważniej.

Aby uruchomić usługę, musi ona zostać najpierw utworzona, a następnie należy wywołać jej metodę `onStartCommand`. Android posiada wystarczająco wiele rezerw, aby przechowywać ten proces w pamięci, dzięki czemu usługa może obsługiwać przychodzące żądania klienta.

Istnieje różnica pomiędzy procesem usługi pozostającej w pamięci i uruchomionej. Usługa zostaje uruchomiona jedynie w odpowiedzi na metodę `startService`, która z kolei wywołuje metodę `onStartCommand`. To, że metoda nie jest wykonywana, wcale nie oznacza, że proces usługi nie znajduje się w pamięci. Czasami programiści odnoszą się do niej jak do uruchomionej usługi, nawet jeżeli ona jedynie istnieje w pamięci, pochłania zasoby, ale nie wykonuje żadnych innych działań. Zazwyczaj to ma się na myśli, gdy się twierdzi, że system przechowuje uruchomioną usługę.

W rzeczywistości w wyniku wywołania metody `startService` zostanie wywołana metoda `onStartCommand` i jeśli zajmie to więcej czasu niż 5 – 10 sekund, może to spowodować wystąpienie ostrzeżenia ANR i zamknięcie procesu przechowującego usługę. Usługa nie może bez wątku roboczego działać dłużej niż 10 sekund. Powinniśmy zatem odróżniać usługi dostępne od usług uruchomionych.

Android utrzymuje usługę dostępną w pamięci, jeśli jest to tylko możliwe. Jednak w przypadku rygorystycznych ograniczeń pamięci system może zadecydować o odzyskaniu procesu i wywołaniu metody `onDestroy()` wobec usługi. Dzieje się tak wtedy, gdy nie są wywoływane metody `onCreate()`, `onStart()` lub `onDestroy()` usługi.

W przeciwnieństwie jednak do zamkniętej aktywności, usługa może zostać ponownie uruchomiona, gdy znów zasoby staną się dostępne oraz jeśli w kolejce znajdują się oczekujące intencje `startService`. System uruchomi usługę ponownie, a następnie za pomocą metody `onStartCommand()` dostarczy do niej intencję. Oczywiście, w momencie przywracania usługi zostanie wywołana metoda `onCreate()`. Ponieważ usługi są bez przerwy ponownie uruchamiane, rozsądnie jest uznać, że w przeciwnieństwie do aktywności i innych składników, są one **fundamentalnie trwałymi składnikami**.

## Szczegółowe informacje na temat usługi nietrwałe

Czym więc jest nietrwała usługa?

Zastanówmy się nad sytuacją, w której usługa nie jest automatycznie uruchamiana ponownie. Po wywołaniu metody `startService` przez klienta zostaje utworzona usługa, a następnie jest wywołana metoda `onStartCommand`. Usługa taka nie zostanie ponownie uruchomiona, jeżeli klient jawnie wywoła metodę `stopService`.

Metoda ta, w zależności od liczby stale podłączonych klientów, może przenosić usługę w stan zatrzymania, w czasie którego zostaje wywołana metoda `onDestroy` usługi, i cykl życia usługi dobiera końca. Po zatrzymaniu w taki sposób usługi przez jej ostatniego klienta nie zostanie ona ponownie uruchomiona.

Protokół ten świetnie sprawdza się w przypadku, gdy wszystko przebiega zgodnie z planem, gdy metody rozpoczęcia i zatrzymania są wywoływane i przetwarzane we właściwej kolejności, bez żadnych uchybień.

Zanim wydano wersję 2.0 Androida, bardzo często urządzenia przechowywały w pamięci mnóstwo uruchomionych usług, które pochłaniały zasoby, chociaż nie miały żadnych zadań do wykonania. Oznaczało to, że system przywracał je pomimo braku komunikatów w kolejce. Tak się mogło dziać w przypadku braku wywołania usługi `stopService` — albo z powodu wystąpienia wyjątku, albo ponieważ proces został zamknięty pomiędzy wywołaniami metod `onStartCommand` i `stopService`.

W wersji 2.0 Androida wprowadzono rozwiązanie, dzięki któremu możemy zadecydować, że przy braku oczekujących intencji usługi nie będą uruchamiane ponownie. Jest to dobre rozwiązanie, ponieważ każdy obiekt, który uruchomił usługę, na przykład menedżer alarmu, wywoła ją ponownie. Dokonujemy tego poprzez zwrot flagi nietrwałości (`Service.START_NOT_STICKY`) z metody `onStartCommand`.

Jednak nietrwała usługa wcale nie jest taka nietrwała. Pamiętajmy, że nawet jeśli oznaczymy usługę jako nietrwałą, w przypadku obecności oczekujących intencji system uruchomi ją ponownie. Flagi okazują się przydatne jedynie wtedy, gdy intencje oczekujące są niedostępne.

## Informacje dotyczące trwałej usługi

Zatem co to znaczy, że usługa jest trwała?

Flaga trwałości (`Service.START_STICKY`) oznacza, że Android powinien ponownie uruchomić usługę, nawet jeśli nie ma oczekujących intencji. Podczas ponownego uruchamiania usługi system wywołuje metody `onCreate` i `onStartCommand`, zawierające argument w postaci pustej intencji. W razie potrzeby usługa uzyska w ten sposób możliwość wywołania metody `stopSelf`, jeśli takie rozwiązanie okaże się właściwe. Wynika z tego wniosek, że trwała usługa podczas ponownego uruchamiania musi przetwarzanie puste intencje.

## Odmiana nietrwałej usługi — ponownie dostarczane intencje

Usługi lokalne pracują przede wszystkim zgodnie z wzorcem, wedle którego metody `onStart` i `stopSelf` są wywoływane parami. Klient wywołuje metodę `onStart`; usługa, po zakończeniu zadania, sama wywołuje metodę `stopSelf`. Widzimy to wyraźnie w implementacji klasy `IntentService`, omawianej przy okazji listingu 14.15.

Jeżeli usługa wymaga, powiedzmy, 30 minut na wypełnienie zadania, przez ten czas nie wywoła metody `stopSelf`. W międzyczasie jest ona odzyskiwana. Jeżeli wprowadzimy flagę nietrwałości, usługa nie zostanie przywrócona i metoda `stopSelf` nigdy nie zostanie wywołana.

W wielu przypadkach taka sytuacja nikomu nie przeszkaździ. Jeżeli jednak chcemy upewnić się, że pojawią się wywołania obydwu wspomnianych metod, Android nie powinien pozbywać się z kolejki zdarzenia `start`, dopóki nie zostanie wywołana metoda `stopSelf`. W ten sposób, gdy usługa zostanie odzyskana, będzie zawsze istniało oczekujące zdarzenie, chyba że zostanie wywołana metoda `stopSelf`. Jest to tak zwany tryb ponownego dostarczania (`redeliver`) i możemy go zdefiniować w metodzie `onStartCommand` za pomocą flagi `Service.START_REDELIVER`.

## Definiowanie flag usługi w metodzie `onStartCommand`

Co ciekawe, trwałość usługi jest powiązana z metodą `onStartCommand`, a nie `onCreate`. Jest to nieco dziwne, ponieważ dotychczas mówiliśmy o usługach znajdujących się w trybach `sticky`, `nonsticky` lub `redeliver`, jakby były atrybutami umieszczanymi na poziomie usługi. Jednak takie określanie natury usługi jest oparte na wartości otrzymywanej z metody `onStartCommand`. Jaki jest w tym cel? Sami się zastanawiamy, ponieważ w przypadku tego samego wystąpienia danej usługi metoda `onStartCommand` jest wywoływana wielokrotnie, osobno dla każdej metody `startService`. Co zatem stanie się w przypadku, gdy metoda zwraca różne flagi wskazujące na różniące się tryby działania usługi? Najprawdopodobniej ostatnia zwracana wartość jest decydująca.

## Wybieranie odpowiedniego trybu usługi

Skoro znamy już różne tryby zachowania usługi, który z nich będzie nadawał się do długoterminowego odbiorcy komunikatów? Uważamy, że powinna wystarczyć prosta, nietrwała usługa, która zostanie zatrzymana w przypadku braku oczekujących komunikatów w kolejce. Trudno nam sobie wyobrazić zastosowanie trwałych, długoterminowych odbiorców komunikatów, zwłaszcza jeśli chcemy wprowadzić obiekt `IntentService`, zatrzymujący usługę, jeśli nie będzie żadnych oczekujących intencji.

Rezultat ujrzymy w implementacji naszej długoterminowej, abstrakcyjnej usługi, zamieszczonej na listingu 14.19, w której otrzymaliśmy flagę nietrwałości.

## Kontrolowanie blokady przechodzenia w stan zatrzymania z dwóch miejsc jednocześnie

Zanim zaprezentujemy kod źródłowy długoterminowej usługi, zastanówmy się nad zadaniami usługi w zakresie utrzymywania urządzenia w stanie aktywności.

Po uruchomieniu kodu usługi należy założyć częściową blokadę przechodzenia w stan zatrzymania. W tym celu na etapie generowania usługi musimy wprowadzić tę blokadę poprzez utworzenie klasy LightedGreenRoom. Stwierdzamy, że dokonuje tego odbiorca komunikatów, i to będzie prawda. Jednak usługa może zostać uruchomiona sama z siebie, co spowodowałoby ominięcie etapu budowania oświetlonego pokoju. Musimy więc w obydwu tych miejscach kontrolować blokadę przechodzenia w stan zatrzymania.

Kod długoterminowego odbiorcy komunikatów z listingu 14.18 uruchamia blokadę budzenia za pomocą metody LightedGreenRoom.setup(). Taką samą czynność wykonamy w wywoaniu metody tworzącej usługę.

Oprócz konfigurowania oświetlonego zielonego pokoju nasza usługa musi zostać zarejestrowana jako jego klient. Pozwoli to na wyczyszczenie pamięci po zamknięciu usługi za pomocą metody onDestroy().

## Implementacja długoterminowej usługi

Skoro wiemy już co nieco na temat klasy IntentService, flag trybu usługi oraz oświetlonego zielonego pokoju, jesteśmy gotowi przyjrzeć się długoterminowej usłudze, zamieszczonej na listingu 14.25.

### Listing 14.25. Długoterminowa usługa

---

```
public abstract class ALongRunningNonStickyBroadcastService
extends IntentService
{
    public static String tag = "ALongRunningBroadcastService";
    protected abstract void
        handleBroadcastIntent(Intent broadcastIntent);

    public ALongRunningNonStickyBroadcastService(String name){
        super(name);
    }
    /*
     * Metoda ta może zostać wywołana w dwóch przypadkach:
     * 1. Gdy odbiorca komunikatów dostarcza metodę startService.
     * 2. Gdy Android ponownie ją uruchamia z powodu oczekujących intencji.
     *
     * W pierwszym przypadku odbiorca komunikatów zdążył
     * już skonfigurować „oświetlony zielony pokój”.
     *
     * W drugim przypadku musimy zrobić to samo.
     */
}
```

```
@Override
public void onCreate()
{
    super.onCreate();

    //Konfiguruje zielony pokój.
    //Konfigurator ten można wywoływać wiele razy.
    LightedGreenRoom.setup(this.getApplicationContext());

    //Istnieje spore prawdopodobieństwo, że istnieje więcej uruchomionych
    //usług tego typu.
    //Znajomość ich liczby pozwoli nam usunąć blokady
    //w trakcie wywoływania metody onDestroy.
    LightedGreenRoom.s_registerClient();
}

@Override
public int onStartCommand(Intent intent, int flag, int startId)
{
    //Wywołuje metodę onStart klasy IntentService.
    super.onStart(intent, startId);

    //Informuje zielony pokój o obecności gościa.
    LightedGreenRoom.s_enter();

    //Zaznaczamy usługę jako nietrwałą.
    //Znaczenie: usługa nie jest ponownie uruchamiana, jeśli nie ma
    //oczekujących intencji.
    return Service.START_NOT_STICKY;
}
/*
 * Zwróćmy uwagę, że wywołanie tej metody przebiega
 * w wątku drugoplanowym, skonfigurowanym przez klasę IntentService.
 *
 * Przesłaniamy tę metodę z poziomu klasy IntentService.
 * Odczytujemy oryginalną, nadawaną intencję.
 * Wywołujemy pochodną klasę, zajmującą się intencją nadawanego komunikatu.
 * Na końcu oświetlony zielony pokój zostaje poinformowany, że gość wychodzi.
 * W przypadku ostatniego gościa blokada
 * zostanie zwolniona.
 */
@Override
final protected void onHandleIntent(Intent intent)
{
    try
    {
        Intent broadcastIntent
            = intent.getParcelableExtra("original_intent");
        handleBroadcastIntent(broadcastIntent);
    }
    finally
    {
        LightedGreenRoom.s_leave();
    }
}
```

```
}

/*
 * Jeżeli Android odzyska proces,
 * metoda ta zwolni blokadę
 * niezależnie od liczby obecnych gości.
 */
@Override
public void onDestroy() {
    super.onDestroy();
    LightedGreenRoom.s_unregisterClient();
}
}
```

---

Widać wyraźnie, że rozszerzamy tutaj klasę IntentService i wykorzystujemy wszystkie zalety wątku roboczego, konfigurowanego przez tę klasę. Dodatkowo klasa ta staje się jeszcze bardziej wyspecjalizowana, dzięki czemu zostaje skonfigurowana jako nietrwała usługa. Z perspektywy programisty główną metodą, na jakiej należy się skupić, jest handleBroadcastIntent().

## Testowanie długoterminowych usług

Aby sprawdzić ten kod w akcji, musimy dodać do projektu następujące pliki:

- *LightedGreenRoom.java* (listing 14.24),
- *ALongRunningNonStickyBroadcastService.java* (listing 14.25),
- *ALongRunningReceiver.java* (listing 14.21),
- *Test60SecBCR.java* (listing 14.22),
- *Test60SecBCRService.java* (listing 14.20),
- zaktualizowany plik manifest, zawierający 60-sekundowego odbiorcę oraz usługę (listing 14.14).

Kody źródłowe wymienionych plików znajdziemy we wcześniejszej części rozdziału, obecnie natomiast zapoznamy się z dodatkowymi wpisami pliku manifestu, które znajdziemy na listingu 14.26.

**Listing 14.26.** Definicja długoterminowych odbiorcy i usługi

---

```
<manifest...>
.....
<application....>
    <receiver android:name=".Test60SecBCR">
        <intent-filter>
            <action android:name="com.androidbook.intents.testbc"/>
        </intent-filter>
    </receiver>
    <service android:name=".Test60SecBCRService"/>
</application>
...
<uses-permission android:name="android.permission.WAKE_LOCK"/>
</manifest>
```

---

Zwróćmy również uwagę, że do uruchomienia tego projektu będą jeszcze potrzebne uprawnienia do korzystania z blokady przechodzenia w stan zatrzymania.

## Instrukcje dotyczące kompilowania kodu

W niniejszym rozdziale zawarliśmy dwa projekty. Jeden z nich pozwala nam na testowanie odbiorców komunikatów (nazwijmy go TestBCR), w drugim natomiast badamy samodzielnego odbiorcę, w tym także długoterminowych: odbiorcę i usługę (StandaloneBCR). Obydwa projekty są skompresowane i dostępne do pobrania; adres URL można znaleźć w podrozdziale „Odbońniki”. Proponujemy pobrać ten plik ZIP i rozpakować go, aby przejrzeć każdy z tych projektów.

### Utworzenie projektów za pomocą pliku ZIP

Aby utworzyć projekty za pomocą danych umieszczonych w pliku ZIP, należy wykonać poniższe czynności:

1. Pobierz plik ZIP.
2. Rozpakuj ten plik; powinny się ukazać dwa katalogi główne, po jednym dla każdego projektu. Dla każdego z tych projektów:
  - a) W środowisku Eclipse wybierz z menu *File/Import* opcje *General/Existing Project into Workspace*.
  - b) Wybierz ścieżkę w polu *Select Root Directory*.
  - c) Wybierz opcję *Copy Projects Into workspace*.
  - d) Być może zaistnieje potrzeba wybrania właściwego poziomu interfejsu API, już po wstawieniu projektu poprzez wybranie opcji *Project Properties/Android* i wybór właściwej wartości.

Po skompilowaniu projektów wdrażamy je do emulatora. Samodzielny projekt zawiera wyłącznie odbiorców i usługi. W projekcie TestBCR zamieściliśmy prostą aktywność, uruchamiającą pojedynczą intencję nadawania komunikatu, oddziałującą na odbiorców uwzględnionych w projekcie TestBCR, a także na odbiorców pochodzących ze wspomnianego, samodzielnego projektu.

### Utworzenie projektów za pomocą listingów

Zamieściliśmy tutaj listę plików wymaganych do utworzenia każdego projektu oraz informacje dotyczące sposobu przekształcenia listingów dostępnych w tym rozdziale w działające projekty.

#### Pliki projektu TestBCR

Oto pliki, które będą nam potrzebne do utworzenia projektu TestBCR:

- *TestBCRActivity.java* (listing 14.5),
- *TestReceiver.java* (listing 14.2),
- *TestReceiver2.java* (listing 14.9),
- *TestTimeDelayReceiver.java* (listing 14.11),
- *Utils.java* (listing 14.3),

- */res/layout/main.xml* (listing 14.6),
- */res/menu/main\_menu.xml* (listing 14.7),
- *AndroidManifest.xml* (listing 14.8).

Jeżeli będą potrzebne jakieś inne pliki, możemy zjrzeć do projektu umieszczonego w pliku *.zip* lub sami je utworzyć. Mogą to być tak proste elementy, jak domyślne ikony lub wartości w postaci ciągów znaków. Gdy już utworzymy odbiorców, będziemy musieli zarejestrować ich w pliku manifeście, widocznym na listingu 14.8.

Aby z powyższej listy plików utworzyć działający projekt, należy wykonać poniższe czynności:

1. Utwórz nowy projekt poprzez wybór opcji *File/New Project/Android/Android Project*.
2. Wybierz nazwę, a następnie zaznacz opcję *Create New Project in Workspace*.
3. Wprowadź nazwę aplikacji, na przykład *TestBCR*. Nazwa aplikacji nie ma większego znaczenia, o wiele ważniejsza jest nazwa pakietu.
4. Wybierz poziom interfejsu API.
5. Wprowadź nazwę pakietu *com.androidbook.bcr*.
6. Wybierz dowolną, minimalną wersję pakietu SDK, np. 3.
7. Wybierz aktywność o nazwie *TestBCRActivity* i kliknij przycisk *Finish*.
8. Android utworzy sporą liczbę plików zasobów oraz, prawdopodobnie (w zależności od wersji systemu), pojedynczy plik źródłowy.
9. Utwórz, zaktualizuj lub usuń te pliki na podstawie listingów 14.2 – 14.11.
10. W przypadku plików Java podczas kopiowania treści listingów zamieść na samej górze pliku nazwę pakietu. Następnie wciśnij skrót klawiaturowy *Ctrl+Shift+O*, aby instrukcje importu zostały automatycznie wprowadzone.

Należy zwrócić uwagę, że w trakcie przeprowadzania tego procesu będzie trzeba zmodyfikować kod, aby mógł zostać skompilowany oraz aby uzupełnić brakujące fragmenty. Wszelkie brakujące elementy możemy skopiować z pliku ZIP zawierającego projekt.

## Pliki projektu zawierającego samodzielnego odbiorcę komunikatów

Poniżej wymieniliśmy pliki będące częścią projektu samodzielnego odbiorcy komunikatów:

- *ALongRunningNonStickyBroadcastService.java* (listing 14.25),
- *ALongRunningReceiver.java* (listing 14.21),
- *LightedGreenRoom.java* (listing 14.24),
- *NotificationReceiver.java* (listing 14.15),
- *StandaloneReceiver.java* (listing 14.13),
- *Test60SecBCR.java* (listing 14.22),
- *Test60SecBCRService.java* (listing 14.20),
- *Utils.java* (listing 14.3),
- *AndroidManifest.xml* (listingi 14.14, 14.26).

Ponieważ mamy tu do czynienia z okrejonym projektem, nie będzie potrzeby tworzenia pliku układu graficznego ani pliku menu. Możemy z powyższej listy plików utworzyć działający projekt w następujący sposób:

1. Utwórz nowy projekt poprzez wybór opcji *File/New Project/Android/Android Project*.
2. Wybierz nazwę, a następnie zaznacz opcję *Create New Project in Workspace*.
3. Wprowadź nazwę aplikacji, na przykład *TestStandaloneBCR*. Nazwa aplikacji nie ma większego znaczenia, o wiele ważniejsza jest nazwa pakietu.
4. Wybierz poziom interfejsu API.
5. Wprowadź nazwę pakietu *com.androidbook.salbcr*.
6. Wybierz dowolną, minimalną wersję pakietu SDK, np. 3.
7. Nie wybieraj żadnej aktywności.
8. Android stworzy sporą liczbę plików zasobów, ale, prawdopodobnie (w zależności od wersji systemu) wszelkie pliki źródłowe zostaną pominięte. Zostanie natomiast utworzony pakiet Java.
9. Utwórz, zaktualizuj lub usuń te pliki, opierając się na listingach wymienionych na początku punktu.
10. W przypadku plików Java podczas kopiowania treści listingów zamieść na samej górze pliku nazwę pakietu. Następnie wciśnij skrót klawiaturowy *Ctrl+Shift+O*, aby zostały automatycznie wprowadzone instrukcje importu.

Należy zwrócić uwagę, że w trakcie przeprowadzania tego procesu będzie trzeba zmodyfikować kod, aby go skompilować oraz aby uzupełnić brakujące fragmenty. Wszelkie brakujące elementy możemy skopiować z pliku ZIP zawierającego projekt.

## Odbośniki

Poniżej prezentujemy pomocne odnośniki dla Czytelników, którzy zechcą poszerzyć wiedzę zawartą w tym rozdziale o nowe informacje:

- <http://developer.android.com/reference/android/content/BroadcastReceiver.html> — jest to odnośnik kierujący do interfejsu *BroadcastReceiver*. W niniejszym rozdziale omówiliśmy najbardziej podstawowy rodzaj odbiorcy komunikatów. Pod tym adresem znajdziemy informacje na temat zamawianych komunikatów oraz nieco więcej informacji na temat ich cyklu życia.
- <http://developer.android.com/reference/android/app/Service.html> — łączy to pozwala uzyskać informacje na temat interfejsu *Service*. Jest to szczególnie przydatne źródło podczas pracy z długoterminowymi usługami.
- <http://developer.android.com/reference/android/app/NotificationManager.html> — odnośnik do interfejsu menedżera powiadomień.
- <http://developer.android.com/reference/android/app/Notification.html> — adres URL kierujący nas do interfejsu *Notification*. Poznamy tu różnorodne opcje dostępne podczas korzystania z powiadomień, na przykład takie jak widoki treści oraz efekty dźwiękowe.
- <http://developer.android.com/reference/android/widget/RemoteViews.html> — tu znajdziemy informacje o interfejsie *RemoteViews*. Obiekty tego typu są wykorzystywane do tworzenia własnych, szczególnych widoków powiadomień.
- <http://www.androidbook.com/item/3514> — znajdziemy tutaj notatki autorów dotyczące badań nad długoterminowymi usługami.

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów do pobrania, stworzonych na potrzeby książki. Katalog przechowujący projekty z tego rozdziału, nosi nazwę *ProAndroid3\_Ch14\_TestReceivers*.

## Podsumowanie

W tym rozdziale omówiliśmy bardzo ważne zagadnienia: odbiorców komunikatów, powiadomienia, blokady przechodzenia w stan zatrzymania oraz długoterminowe usługi. Zebraliśmy tu również najistotniejsze zagadnienia omówione w rozdziałach 12. i 13.

Zademonstrowaliśmy podstawy stosowania odbiorcy komunikatów, jego czas życia oraz sposób działania zarówno w procesie, jak i poza nim. Pokazaliśmy, jak można dołączyć do niego usługi, dzięki czemu czas życia odbiorcy komunikatów zostaje przedłużony. Na koniec poeksperymentowaliśmy z klasą `IntentService` i pokazaliśmy, w jaki sposób można ją dalej dostosowywać do własnych potrzeb przy okazji używania długoterminowych usług.

W rozdziale 15. dowiemy się, w jaki sposób można skorzystać z menedżera alarmów do przywołania odbiorcy komunikatów.

# Badanie menedżera alarmów

Za pomocą menedżera alarmów można uruchamiać zdarzenia w Androidzie. Zdarzenia te mogą występować o określonej porze lub w regularnych odstępach czasowych. Rozpoczniemy ten rozdział od omówienia podstaw menedżera alarmów, mianowicie od skonfigurowania prostego alarmu. Następnie zwrócimy uwagę na sposób konfiguracji powtarzalnego alarmu, anulowania alarmu, roli intencji oczekujących (zwłaszcza roli, jaką odgrywa ich unikatowość) oraz ustanawiania wielu alarmów naraz. Po ukończeniu lektury rozdziału Czytelnik zdąży zapoznać się z podstawami menedżera alarmów w Androidzie oraz jego praktycznymi zastosowaniami.

## Podstawy menedżera alarmów — konfiguracja prostego alarmu

Rozpoczniemy od skonfigurowania alarmu uruchamianego o określonej porze i wywołującego odbiorcę komunikatów. Po wywołaniu odbiorcy możemy wykorzystać informacje zawarte w rozdziale 14. do przeprowadzania w nim zarówno krótkich, jak i dłużej trwających operacji.

Aby wykonać ćwiczenie, należy wykonać następujące czynności:

1. Uzyskaj dostęp do menedżera alarmu.
2. Skonfiguruj czas uruchomienia alarmu.
3. Utwórz odbiorcę, który zostanie później wywołany.
4. Utwórz oczekującą intencję, aby później przekazać ją menedżerowi alarmu w celu przywołania odbiorcy.
5. Ustaw alarm z wykorzystaniem czasu określonego w punkcie 2. oraz oczekującej intencji z punktu 4.
6. Obserwuj okno *LogCat*, w którym pojawią się komunikaty wywołanego odbiorcy, utworzonego w punkcie 3.

## Uzyskanie dostępu do menedżera alarmów

Dostęp do menedżera alarmów uzyskujemy w prosty sposób, uwidoczniony na listingu 15.1.

### **Listing 15.1.** Uzyskanie menedżera alarmów

---

```
AlarmManager am =  
    (AlarmManager)  
    mContext.getSystemService(Context.ALARM_SERVICE);
```

---

Na listingu 15.1 zmienna `mContext` odnosi się do obiektu kontekstu. Jeśli na przykład przywołamy ten kod z poziomu menu aktywności, zmienna kontekstu jest tą aktywnością.

## Definiowanie czasu uruchomienia alarmu

Aby ustawić alarm na określona datę i godzinę, będzie nam potrzebna instancja obiektu `Calendar`. Na listingu 15.2 widzimy plik Java (będziemy go potrzebować do utworzenia projektu), w którym wprowadziliśmy pewne funkcje współpracujące z obiektem `Calendar`.

### **Listing 15.2.** Kilka przydatnych funkcji kalendarza

---

```
public class Utils {  
    public static Calendar getTimeAfterInSecs(int secs) {  
        Calendar cal = Calendar.getInstance();  
        cal.add(Calendar.SECOND,secs);  
        return cal;  
    }  
    public static Calendar getCurrentTime(){  
        Calendar cal = Calendar.getInstance();  
        return cal;  
    }  
    public static Calendar getTodayAt(int hours){  
        Calendar today = Calendar.getInstance();  
        Calendar cal = Calendar.getInstance();  
        cal.clear();  
  
        int year = today.get(Calendar.YEAR);  
        int month = today.get(Calendar.MONTH);  
        //reprezentuje dzień miesiąca  
        int day = today.get(Calendar.DATE);  
        cal.set(year,month,day,hours,0,0);  
        return cal;  
    }  
    public static String getDateTimeString(Calendar cal){  
        SimpleDateFormat df = new SimpleDateFormat("MM/dd/yyyy hh:mm:ss");  
        df.setLenient(false);  
        String s = df.format(cal.getTime());  
        return s;  
    }  
}
```

---

Z zestawu funkcji pokazanych w powyższym kodzie, będziemy korzystać z funkcji `getTimeAfterInSecs()`, widocznej na listingu 15.3, aby określić zdarzenie, które odbędzie się 30 sekund później.

#### **Listing 15.3.** Uzyskiwanie wystąpienia czasu

---

```
Calendar cal = Utils.getTimeAfterInSecs(30);
```

---

## Konfigurowanie odbiorcy dla alarmu

Potrzebny jest nam teraz odbiorca umożliwiający odpowiedź na alarm. Przykładowy prosty odbiorca został zaprezentowany na listingu 15.4.

#### **Listing 15.4.** Testowy odbiorca pozwalający na analizę komunikatów alarmu

---

```
public class TestReceiver extends BroadcastReceiver
{
    private static final String tag = "TestReceiver";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Log.d("TestReceiver", "intencja=" + intent);
        String message = intent.getStringExtra("message");
        Log.d(tag, message);
    }
}
```

---

Musimy zarejestrować tego odbiorcę w pliku manifeście za pomocą odpowiedniego znacznika `<receiver>`, co zostało pokazane na listingu 15.5.

#### **Listing 15.5.** Rejestrowanie odbiorcy komunikatów

---

```
<receiver android:name=".TestReceiver"/>
```

---

## Utworzenie oczekującej intencji dostosowanej do alarmu

Po utworzeniu odbiorcy można utworzyć intencję `PendingIntent`, która jest niezbędna do skonfigurowania alarmu. Rozpoczniemy od utworzenia intencji wywołującej odbiorcę `TestReceiver` zdefiniowanego na listingu 15.4. Proces tworzenia tej intencji został ukazany na listingu 15.6.

#### **Listing 15.6.** Utworzenie intencji wskazującej odbiorcę `TestReceiver`

---

```
Intent intent =
    new Intent(mContext, TestReceiver.class);
intent.putExtra("message", "Jednokrotny alarm");
```

---

Zmienna `mContext` stanowi kontekst aktywności, z którego poziomu będziemy wywoływać omawianą funkcję. Skorzystaliśmy bezpośrednio z klasy `TestReceiver` (odmienną metodą było zastosowanie filtra wobec działania intencji, co pokazaliśmy w rozdziale 14.). Mamy również możliwość utworzenia intencji zawierającej dodatkowe dane.

Po utworzeniu standardowej intencji wskazującej danego odbiorcę musimy utworzyć intencję oczekującą, którą trzeba przekazać menedżerowi intencji. Na listingu 15.7 widzimy przykładową intencję oczekującą PendingIntent, utworzoną na podstawie intencji z listingu 15.6.

#### **Listing 15.7.** Utworzenie intencji oczekującej

---

```
PendingIntent pi =  
    PendingIntent.getBroadcast(  
        mContext, //kontekst  
        1, //identyfikator żądania, stosowany do odróżnienia tej intencji od innych  
        intent, //dostarczana intencja  
        0); //flagi oczekującej intencji
```

---

Zwrócmy uwagę, że klasa PendingIntent skonstruuje oczekującą intencję, która ma być w jawnym sposobie dostosowana do nadawanego komunikatu. Mamy również do dyspozycji następujące odmiany tej intencji:

```
PendingIntent.getActivity() //przydatna do rozpoczęcia aktywności  
PendingIntent.getService() //przydatna do rozpoczęcia usługi
```

W dalszej części rozdziału omówimy dokładniej argument identyfikatora żądania, posiadający w tym przypadku wartość 1. W skrócie — służy on do rozróżniania dwóch podobnych do siebie obiektów intencji.

Flagi oczekujących intencji właściwie nie posiadają wpływu na menedżer alarmów. Zalecamy, aby w ogóle z nich nie korzystać i ustawiać je na wartość 0. Flagi te znajdują się przeważnie zastosowanie w kontroli czasu życia oczekującej intencji. Jednak w tym przypadku jest on zarządzany przez menedżer alarmów. Jeżeli chcemy na przykład zakończyć intencję oczekującą, dokonujemy tego za pomocą menedżera.

## **Ustawianie alarmu**

Po zdefiniowaniu czasu w postaci obiektu Calendar (czas jest wyrażany w milisekundach) oraz oczekującej intencji wskazującej na odbiorcę możemy ustawić alarm za pomocą metody set() menedżera alarmów, co zostało zaprezentowane na listingu 15.8.

#### **Listing 15.8.** Metoda służąca do ustanawiania menedżera alarmów

---

```
alarmManager.set(AlarmManager.RTC_WAKEUP,  
    calendarObject.getTimeInMillis(),  
    pendingIntent);
```

---

Jeżeli wprowadzimy atrybut RTC\_WAKEUP, alarm spowoduje wyjście urządzenia ze stanu wstrzymania. Możemy zamiast tego ustawienia wprowadzić atrybut RTC, który spowoduje dostarczenie intencji w momencie wyjścia urządzenia z tego stanu.

Czas zdefiniowany przez drugi argument jest wyrażony w sposób właściwy dla obiektu calendar → object utworzonego na listingu 15.3. Jest to czas liczony w milisekundach, zliczany od 1970 roku. Pokrywa się to również z domyślnymi ustawieniami obiektu Java Calendar.

Po wywołaniu tej metody menedżer alarmów przywoła odbiorcę TestReceiver (listing 15.4) po upływie 30 sekund.

## Projekt testowy

Utwórzmy teraz projekt testowy, dzięki któremu dokładniej przeanalizujemy działanie dotychczas omówionego kodu.

**Uwaga!** Na końcu rozdziału zamieściliśmy adres URL, z którego można pobrać projekty utworzone na potrzeby książki oraz zainportować je do środowiska Eclipse.

Do utworzenia projektu będziemy potrzebować następujących plików:

- *TestAlarmsDriverActivity.java* — aktywność służąca do ustanawiania alarmów (listing 15.12).
- *SendAlarmOnceTester.java* — jest to główna klasa, służąca do testowania funkcji jednokrotnego wysłania alarmu. Zaprezentujemy również podobne klasy, pozwalające Czytelnikowi na przetestowanie innych funkcjonalności (listing 15.11).
- *BaseTester.java* — baza klasowa umożliwiająca klasom testującym, takim jak *SendAlarmOnceTester.java*, odsyłanie wyników za pomocą interfejsu *IReportBack* (listing 15.10).
- *IReportBack.java* — ten niewielki pomocniczy interfejs, stworzony dla klasy *BaseTester.java*, zbiera komunikaty debugowania i przesyła je do aktywności sterującej (listing 15.9).
- *TestReceiver.java* — jest to klasa przywoływana w momencie uruchamiania alarmu. Została ona zaprezentowana na listingu 15.4.
- *Utils.java* — narzędzia pozwalające na zarządzanie datą, godziną czy kalendarzem, zaprezentowane na listingu 15.2.
- */res/menu/main\_menu.xml* — plik menu aktywności sterującej (listing 15.13).
- */res/layout/main.xml* — plik układu graficznego aktywności sterującej (listing 15.14).
- *AndroidManifest.xml* — znany nam doskonale plik manifest, wymagany przez każdą aplikację piszącą dla Androida (listing 15.15).

Zaprezentujemy po kolejny każdy z wymienionych plików, począwszy od klas bazowych, dzięki którym skoordynujemy działania pomiędzy aktywnością sterującą a różnorodnymi klasami sterującymi, pozwalającymi na analizę poszczególnych właściwości alarmu. Pierwsza z klas bazowych, *IReportBack*, została umieszczona na listingu 15.9.

### Listing 15.9. IReportBack.java

---

```
//IReportBack.java
package com.androidbook.alarms;

/*
 * Interfejs, zazwyczaj implementowany przez aktywność,
 * za pomocą której klasa robocza może przekazać informacje
 * na temat zachodzących zdarzeń.
 */
public interface IReportBack
{
    public void reportBack(String tag, String message);
}
```

---

Jak zostało wspomniane w komentarzach, interfejs ten jest wykorzystywany przez klasę testującą do przekazywania komunikatów aktywności sterującej. Zobaczmy to wyraźnie podczas omawiania kodów klas podrzędnych, na przykład *SendAlarmOnceTester.java* (listing 15.11).

Wszystkie klasy testujące, takie jak *SendAlarmOnceTester*, wywodzą się z klasy *BaseTester*. Kod źródłowy pliku *BaseTester.java* jest dostępny na listingu 15.10.

---

**Listing 15.10.** *BaseTester.java*

---

```
//BaseTester.java
package com.androidbook.alarms;
import android.content.Context;public class BaseTester
{
    protected IReportBack mReportTo;
    protected Context mContext;
    public BaseTester(Context ctx, IReportBack target)
    {
        mReportTo = target;
        mContext = ctx;
    }
}
```

---

Jest to prosta klasa pomocnicza, zapewniająca dwa elementy wywodzącym się od niej klasom testującym, takim jak *SendAlarmOnceTester*: kontekst, który w miarę potrzeby będzie wykorzystywany przez ich metody, oraz aktywność implementującą interfejs *IReportBack*, dzięki czemu komunikaty będą zapisywane w dzienniku.

Po utworzeniu interfejsu *IReportBack* oraz *SendAlarmOnceTester* będziemy mogli zacząć wprowadzanie kodu klasy *SendAlarmOnceTester.java*, testującej wysyłanie pojedynczego alarmu (listing 15.11).

---

**Listing 15.11.** Plik klasy pozwalającej na jednorazowe wysłanie alarmu

---

```
// SendAlarmOnceTester.java
package com.androidbook.alarms;
import java.util.Calendar;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;

public class SendAlarmOnceTester extends BaseTester
{
    private static String tag = "SendAlarmOnceTester";
    SendAlarmOnceTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }

    /*
     * Alarm może przywoływać żądanie nadania komunikatu
     * o określonej porze.
     * Nazwa odbiorcy komunikatu jest jawnie
    
```

```

* zdefiniowana w intencji.
*/
public void sendAlarmOnce()
{
    //Generuje wystąpienie w czasie równym 30 sekundom od bieżącej chwili.
    Calendar cal = Utils.getTimeAfterInSecs(30);

    //Jeżeli chcemy wywołać alarm dzisiaj o godzinie 11.
    //Calendar cal = Utils.getTodayAt(11);
    //Wyświetla w widoku debugowania informację o fakcie.
    //Ustanawiamy alarm na określona godzinę.
    String s = Utils.getDateTimeString(cal);
    mReportTo.reportBack(tag, "Ustanawianie alarmu na: " + s);

    //Pobiera intencję pozwalającą na przywołanie odbiorcy.
    //TestReceiver
    Intent intent =
        new Intent(mContext, TestReceiver.class);
    intent.putExtra("message", "Alarm jednorazowy");

    PendingIntent pi =
        PendingIntent.getBroadcast(
            mContext, //kontekst
            1, //identyfikator żądania, pozwalający na rozróżnianie intencji
            intent, //dostarczana intencja
            PendingIntent.FLAG_ONE_SHOT); //flagi intencji oczekującej

    //Tworzy harmonogram alarmu!
    AlarmManager am =
        (AlarmManager)
            mContext.getSystemService(Context.ALARM_SERVICE);

    am.set(AlarmManager.RTC_WAKEUP,
        cal.getTimeInMillis(),
        pi);
}
}

```

Intencja klasy SendAlarmOnceTester wysyła pojedynczy alarm, co powoduje wywołanie odbiorcy komunikatów. Możemy się o tym przekonać, przyglądając się metodzie sendAlarmOnce(), pokazanej na listingu 15.11. Odbiorca TestReceiver, będący celem alarmu, został przedstawiony na listingu 15.4, zatem każdy aspekt tej metody został już wcześniej omówiony. Listing 15.11 stanowi jedynie złożenie opisanych powyżej fragmentów kodu.

Przeanalizujmy teraz aktywność sterującą, wywołującą metodę sendAlarmOnce(). Jej kod źródłowy prezentujemy na listingu 15.12. Ta główna aktywność projektu testowego przywołuje obiekty menu, za pomocą których będziemy sprawdzać różne rodzaje alarmów (zarówno już przedstawionych, jak i tych, które dopiero zostaną omówione). Na razie jednak dysponujemy wyłącznie fragmentem kodu pozwalającym na wywołanie elementu menu, dzięki któremu uruchomimy omówiony powyżej mechanizm. W dalszej części rozdziału poznamy fragmenty kodu uruchamiane po wcisnięciu pozostałych opcji menu.

Metoda onCreate() klasy TestAlarmsDriverActivity (listing 15.12) tworzy wystąpienie klasy SendAlarmOnceTester, do której zostaną przeniesione działania menu. Zwróćmy uwagę, że wspomniana aktywność przenosi samoistnie siebie samą, jak również zmienne IReportBack i Context do konstruktora klasy SendAlarmOnceTester. Implementuje ona również interfejs IReportBack oraz aktualizuje widok debugera za pomocą przekazanego tekstu (zapisana grubioną czcionką metoda reportBack na listingu 15.12).

**Listing 15.12.** Przykładowa aktywność pozwalająca na testowanie różnych ustawień alarmów

---

```
// TestAlarmsDriverActivity.java
package com.androidbook.alarms;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.TextView;

public class TestAlarmsDriverActivity extends Activity
implements IReportBack
{
    public static final String tag="TestAlarmsDriverActivity";
    private SendAlarmOnceTester alarmTester = null;
    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        alarmTester = new SendAlarmOnceTester(this,this);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        //Wywołuje klasę nadziedną w celu doliczenia dowolnych menu systemowych
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater(); //z aktywności
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        appendMenuItemText(item);
        if (item.getItemId() == R.id.menu_clear)
        {
            this.emptyText();
            return true;
        }
        if (item.getItemId() == R.id.menu_alarm_once)
        {
            alarmTester.sendAlarmOnce();
            return true;
        }
    }
}
```

```

    }
    //Później dodamy tutaj więcej opcji menu
    return true;
}
//Funkcja odziedziczona z interfejsu IReportBack
public void reportBack(String tag, String message)
{
    this.appendText(tag + ":" + message);
    Log.d(tag,message);
}

//Proste funkcje służące do pracy z widokiem debuggera
//tej aktywności
private TextView getTextView() {
    return (TextView)this.findViewById(R.id.text1);
}
private void appendMenuItemText(MenuItem menuItem){
    String title = menuItem.getTitle().toString();
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + title);
}
private void emptyText(){
    TextView tv = getTextView();
    tv.setText("");
}
private void appendText(String s){
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + s);
    Log.d(tag,s);
}
}

```

---

Jak widać, aktywność `TestAlarmsDriverActivity` reaguje na kilka elementów menu. Zdefiniowany dla niej plik `menu.xml` został zaprezentowany na listingu 15.13. Zamieściliśmy w nim od razu wszystkie dodatkowe scenariusze testowe, którymi będziemy się zajmować przez resztę rozdziału. Ponieważ obecność tych dodatkowych opcji nie będzie nam przeszkadzać w skompilowaniu projektu, postanowiliśmy umieścić je wszystkie za jednym zamachem.

#### **Listing 15.13.** Elementy menu służące do testowania różnorodnych scenariuszy menedżera alarmów

---

```

<!-- /res/menu/main_menu.xml -->
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_alarm_once"
            android:title="Pojedynczy alarm" />
        <item android:id="@+id/menu_alarm_repeated"
            android:title="Alarm powtarzalny" />
        <item android:id="@+id/menu_alarm_cancel"
            android:title="Anulowanie alarmów" />
        <item android:id="@+id/menu_alarm_multiple"
            android:title="Wiele alarmów" />
        <item android:id="@+id/menu_alarm_distinct_intents"
            android:title="Oddzielne intencje" />
    
```

```
<item android:id="@+id/menu_alarm_intent_primacy"
      android:title="Pierwszeństwo intencji" />
<item android:id="@+id/menu_clear"
      android:title="Wyczyszczać" />
</group>
</menu>
```

---

Listing 15.14 zawiera plik układu graficznego obsługujący aktywność sterującą TestAlarms →DriverActivity (listing 15.12). Plik ten znajduje się w katalogu */res/layout/main.xml*.

**Listing 15.14.** Układ graficzny aktywności TestAlarmsDriverActivity

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

---

Na listingu 15.15 umieściliśmy plik manifest tego projektu.

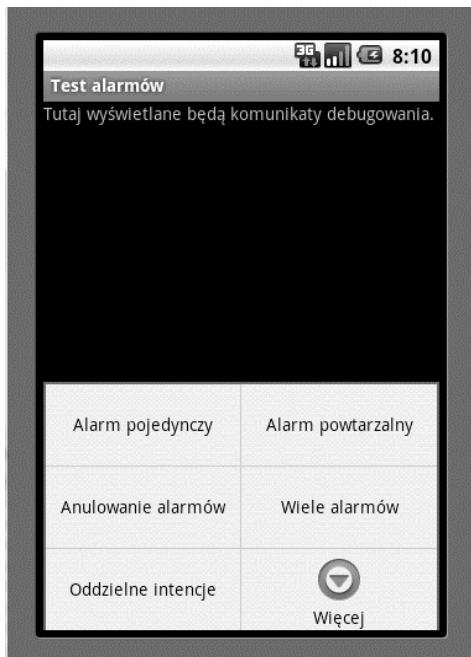
**Listing 15.15.** Plik manifest programu testującego menedżer alarmów

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.alarm"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="Test alarmów">
        <activity android:name=".TestAlarmsDriverActivity"
            android:label="Test alarmów">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".TestReceiver">
            <intent-filter>
                <action android:name="com.androidbook.intents.testbc"/>
            </intent-filter>
        </receiver>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

---

Poza definicją odbiorcy nie są potrzebne inne wpisy w manifeście, żeby korzystać z menedżera alarmów. Definicja odbiorcy jest na powyższym listingu zaznaczona pogrubionym drukiem. Po skompilowaniu i uruchomieniu projektu powinniśmy ujrzeć aktywność i strukturę menu przypominającą interfejs użytkownika na rysunkach 15.1 i 15.2.



**Rysunek 15.1.** Przykładowa aktywność pozwalająca na przetestowanie menedżera alarmów

Na rysunku 15.1 widzimy część opcji dostępnych w menu. Aby ujrzeć pozostałe elementy menu, musimy kliknąć ikonę *Więcej*. Opcje te zostały ukazane na rysunku 15.2.

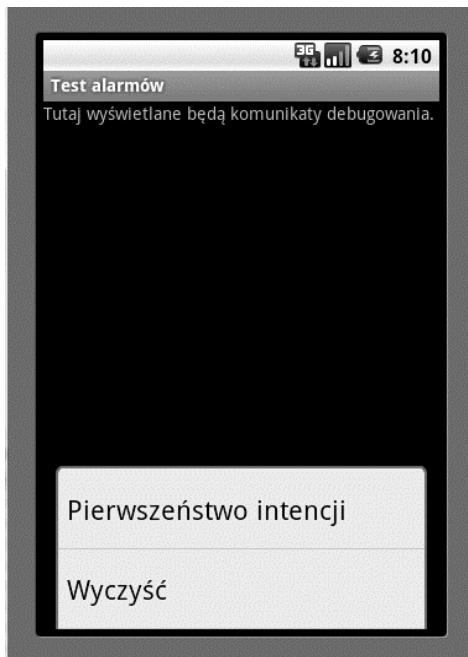
Jeżeli teraz wybierzemy widoczną na rysunku 15.1 opcję *Alarm pojedynczy*, uruchomimy kod zawarty w metodzie `sendAlarmOnce()` (listing 15.11). W tym momencie zostanie zdefiniowany alarm, który uruchomi się po 30 sekundach. Po tym czasie odbiorca `TestReceiver` zacznie umieszczać komunikaty w oknie *LogCat*.

## Analiza alternatywnych wersji menedżera alarmów

Po wyjaśnieniu podstawowych zasad dotyczących konfigurowania alarmu możemy się zająć kilkoma dodatkowymi scenariuszami, na przykład ustanowieniem powtarzającego się alarmu lub anulowaniem alarmów. Przyjrzymy się także nietypowym sytuacjom, na które możemy natrafić podczas pracy z menedżerem alarmów.

### Konfigurowanie powtarzalnego alarmu

Omówiliśmy już sposób utworzenia jednorazowego alarmu, czas zatem zastanowić się, jak możemy uzyskać alarm, którego wywoływanie może następować w sposób powtarzalny.



Rysunek 15.2. Rozszerzone menu naszej przykładowej aktywności

Aby zrozumieć zasadę postępowania w takim przypadku, spójrzmy na kod na listingu 15.16. Mamy tu do czynienia ze środowiskiem testowym, zawierającym metodę `SendOnceAlarm→Tester()`, w którym zaimplementowano również metodę `sendRepeatingAlarm()`. W ten sposób umożliwiono sprawdzanie powtarzalnego alarmu.

---

**Listing 15.16.** Konfigurowanie powtarzalnego alarmu

---

```
// SendRepeatingAlarmTester.java
package com.androidbook.alarms;
import java.util.Calendar;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;

public class SendRepeatingAlarmTester
extends SendAlarmOnceTester
{
    private static String tag = "SendRepeatingAlarmTester";
    SendRepeatingAlarmTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }

    /*
     * Alarm może wywołać żądanie nadawania komunikatu
     * o określonej porze oraz
     * w regularnych odstępach.
    }
```

```

*
* Wykorzystuje tę samą intencję co powyżej,
* lecz inny identyfikator żądania w celu uniknięcia konfliktów
* z wcześniejszym utworzonym alarmem jednokrotnym.
*
* Wykorzystuje metodę getDistinctPendingIntent().
*/
public void sendRepeatingAlarm()
{
    Calendar cal = Utils.getTimeAfterInSecs(30);
    //Calendar testcal = Utils.getTodayAt(11);
    String s = Utils.getDateTimeString(cal);
    this.mReportTo.reportBack(tag,
        "Ustanowienie powtarzalnego alarmu w odstępach 5 s, poczawszy od: " + s);

    //Pobiera intencję wywołującą odbiorcę
    Intent intent =
        new Intent(this.mContext, TestReceiver.class);
    intent.putExtra("message", "Alarm powtarzalny");

    PendingIntent pi = this.getDistinctPendingIntent(intent, 2);
    // Tworzy harmonogram alarmu!
    AlarmManager am =
        (AlarmManager)
            this.mContext.getSystemService(Context.ALARM_SERVICE);

    am.setRepeating(AlarmManager.RTC_WAKEUP,
        cal.getTimeInMillis(),
        5*1000, //5 sekund
        pi);
}

protected PendingIntent getDistinctPendingIntent
    (Intent intent, int requestId)
{
    PendingIntent pi =
        PendingIntent.getBroadcast(
            mContext, //kontekst
            requestId, //identyfikator żądania
            intent, //dostarczana intencja
            0);
    return pi;
}
}

```

Kluczowe elementy na listingu 15.16 zostały zaznaczone pogrubionym drukiem. Powtarzalny alarm zostaje wywołany za pomocą metody `setRepeating()` obiektu menedżera alarmów. Jedną z danych wejściowych tej metody jest intencja oczekująca, wskazująca odbiorcę komunikatów. Wykorzystaliśmy tu tę samą intencję co w przypadku klasy `SendAlarmOnceTester`.

Jednak w trakcie odczytywania intencji oczekującej wykorzystaliśmy inny identyfikator żądania — o wartości 2. Jeśli pozostawilibyśmy stary identyfikator, nasz program mógłby zachować się w dziwny sposób. Powiedzmy, że najpierw klikamy obiekt menu odpowiedzialny za uruchomienie

powtarzalnego alarmu. W ten sposób włączymy alarm oraz wywołamy odbiorcę `TestReceiver`. Założmy, że ten powtarzalny alarm zostanie uruchomiony 30 sekund później. Przechodzimy teraz dalej i klikamy element menu *Alarm pojedynczy*. Alarm ten uruchomi się tylko jeden raz po upływie 30 sekund i wywoła tego samego odbiorcę `TestReceiver`.

Gdyby obydwa elementy menu działały w ten sposób, zostałyby uruchomione obydwa rodzaje alarmów. Warto jednak zauważyc, że alarm odezwie się tylko jeden raz. Aby wszystko działało jak należy, musimy wprowadzić różne argumenty `requestCode` w oczekującej intencji. Wyjaśnienie tej sytuacji znajdziemy w podrozdziale „Pierwszeństwo intencji w uruchamianiu alarmów”.

## Kompilowanie kodu omawianego przykładu

Aby przetestować opisany fragment kodu, będziemy musieli zmienić zawartość kilku plików w projekcie.

Najpierw musimy dodać klasę widoczną na listingu 15.16 w postaci nowego pliku źródłowego o nazwie `SendRepeatingAlarmTester.java`.

Następnie musimy zmienić w kilku miejscach aktywność sterującą `TestAlarmsDriverActivity`, zdefiniowaną na listingu 15.12.

Zastąpmy poniższe wiersze:

```
private SendAlarmOnceTester alarmTester = null;  
"  
alarmTester = new SendAlarmOnceTester(this, this);
```

następującymi:

```
private SendRepeatingAlarmTester alarmTester = null;  
"  
alarmTester = new SendRepeatingAlarmTester(this, this);
```

Dodajmy poniższy kod, umożliwiający odpowiedź na zdarzenia menu:

```
if (item.getItemId() == R.id.menu_alarm_repeated)  
{  
    alarmTester.sendRepeatingAlarm();  
    return true;  
}
```

Po wprowadzeniu tych zmian możemy wykorzystać interfejs widoczny na rysunku 15.1 do przywołania obiektu menu *Alarm powtarzalny*. Wyniki testu ujrzymy w oknie *LogCat*. Dowiemy się teraz, w jaki sposób możemy anulować powtarzalny alarm.

## Anulowanie alarmu

Aby zrozumieć, jak przebiega anulowanie alarmów, wprowadzimy kolejną klasę testową, nazwaną `CancelRepeatingAlarmTester` (listing 15.17).

---

### Listing 15.17. Anulowanie powtarzalnego alarmu

---

```
// CancelRepeatingAlarmTester.java  
package com.androidbook.alarms;  
import android.app.AlarmManager;
```

```

import android.app.PendingIntent;
import android.content.Context;
import android.content.Intent;

public class CancelRepeatingAlarmTester
extends SendRepeatingAlarmTester
{
    private static String tag = "CancelRepeatingAlarmTester";
    CancelRepeatingAlarmTester(Context ctx, IReportBack target) {
        super(ctx, target);
    }
    /*
     * Alarm może zostać zatrzymany poprzez anulowanie intencji.
     * Do anulowania intencji będzie nam potrzebna
     * jej kopia.
     *
     * Intencja ta musi posiadać tę samą sygnaturę
     * oraz identyfikator żądania.
     */
    public void cancelRepeatingAlarm()
    {
        //Pobiera intencję do przywołania
        //klasy TestReceiver.
        Intent intent =
            new Intent(this.mContext, TestReceiver.class);

        //Aby anulować intencje, nie trzeba wypełniać argumentu extra
        //intent.putExtra("message", "Powtarzalny alarm");

        PendingIntent pi = this.getDistinctPendingIntent(intent, 2);

        //Ustanawia harmonogram alarmu!
        AlarmManager am =
            (AlarmManager)
                this.mContext.getSystemService(Context.ALARM_SERVICE);
        am.cancel(pi);
        this.mReportTo.reportBack(tag, "Nie powinnismy mieć juz do czynienia z alarmem");
    }
}

```

Aby anulować alarm, musimy najpierw skonstruować intencję oczekującą, a następnie przekazać ją menedżerowi alarmów jako argument metody `cancel()`.

Musimy jednak mieć pewność, że obiekt `PendingIntent` jest skonstruowany dokładnie w taki sam sposób, jak to miało miejsce podczas konfigurowania alarmu, łącznie z identyfikatorem żądania i docelowym odbiorcą. Przyjrzyjmy się kodowi źródłowemu metody `getDistinct PendingIntent()` z listingu 15.16, aby zrozumieć, w jaki sposób kod żądania jest wykorzystywany wraz z metodą `PendingIntent.getBroadcast()` — możemy zignorować dodatkowe dane intencji z listingu 15.17, ponieważ nie odgrywają one roli w procesie jej anulowania.

## Kompilowanie kodu omawianego przykładu

Aby przetestować ten fragment kodu, będziemy musieli zmienić zawartość kilku plików w projekcie.

Najpierw musimy dodać klasę widoczną na listingu 15.17 w postaci nowego pliku źródłowego o nazwie *CancelRepeatingAlarmTester.java*.

Następnie musimy zmienić w kilku miejscach aktywność sterującą *TestAlarmsDriverActivity*, zdefiniowaną na listingu 15.12.

Zastąpmy poniższe wiersze:

```
private SendAlarmOnceTester alarmTester = null;  
"  
alarmTester = new SendAlarmOnceTester(this, this);
```

następującymi:

```
private CancelRepeatingAlarmTester alarmTester = null;  
"  
alarmTester = new CancelRepeatingAlarmTester(this, this);
```

Dodajmy poniższy kod, umożliwiający odpowiedź na zdarzenia menu:

```
if (item.getItemId() == R.id.menu_alarm_cancel)  
{  
    alarmTester.cancelRepeatingAlarm();  
    return true;  
}
```

Możemy przetestować tę funkcję, klikając najpierw obiekt menu *Alarm powtarzalny* (rysunek 15.1). W wyniku tego widok *LogCat* zacznie być aktualizowany co 5 sekund. Jeżeli wybierzemy teraz opcję *Anuluj alarmy*, komunikaty przestaną napływać.

## Praca z wieloma alarmami jednocześnie

Naszym zdaniem podczas konfigurowania wielu menedżerów alarmów dla tego samego odbiorcy komunikatów zachowanie menedżerów alarmów jest nieco nieintuicyjne — jeżeli kilkakrotnie przywołamy alarm wskazujący danego odbiorcę, poskutkuje jedynie ostatnie wywołanie.

W celu wyjaśnienia tego zachowania przyjrzyjmy się najpierw przygotowanej na listingu 15.18 klasie testującej. Znajdziemy tutaj dwie metody. Pierwsza z nich, *scheduleSameIntentMultipleTimes()*, ustanawia wielokrotnie harmonogram dla tej samej intencji. Druga funkcja, *scheduleDistinctIntents()*, ma takie samo zadanie, rozróżnia ona jednak intencje po identyfikatorze żądania.

---

### Listing 15.18. Praca z wieloma alarmami

---

```
//ScheduleIntentMultipleTimesTester.java  
package com.androidbook.alarms;  
import java.util.Calendar;  
import android.app.AlarmManager;  
import android.app.PendingIntent;  
import android.content.Context;  
import android.content.Intent;
```

```
public class ScheduleIntentMultipleTimesTester  
extends CancelRepeatingAlarmTester  
{  
    private static String tag = "ScheduleIntentMultipleTimesTester";  
    ScheduleIntentMultipleTimesTester(Context ctx, IReportBack target){  
        super(ctx, target);  
    }  
    /*  
     * Nie można wielokrotnie ustanawiać harmonogramu tej samej intencji.  
     * Jeżeli tak zrobimy, tylko ostatni będzie wykonywany.  
     *  
     * Zwróćmy uwagę, że wykorzystujemy ten sam identyfikator żądania.  
     */  
    public void scheduleSameIntentMultipleTimes()  
    {  
        //Pobiera wiele wystąpień czasu.  
        Calendar cal = Utils.getTimeAfterInSecs(30);  
        Calendar cal2 = Utils.getTimeAfterInSecs(35);  
        Calendar cal3 = Utils.getTimeAfterInSecs(40);  
        Calendar cal4 = Utils.getTimeAfterInSecs(45);  
  
        //Wyświetla w widoku debugowania informację, że  
        //alarmy zostaną uruchomione o określonej porze.  
        String s = Utils.getDateTimeString(cal);  
        mReportTo.reportBack(tag, "Nastawianie alarmu na: " + s);  
  
        //Pobiera intencję służącą do wywołania odbiorcy.  
        Intent intent =  
            new Intent(mContext, TestReceiver.class);  
        intent.putExtra("message", "Ta sama intencja wiele razy");  
  
        PendingIntent pi = this.getDistinctPendingIntent(intent, 1);  
  
        // Ustanawia wiele harmonogramów tej samej intencji.  
        AlarmManager am =  
            (AlarmManager)  
            mContext.getSystemService(Context.ALARM_SERVICE);  
  
        am.set(AlarmManager.RTC_WAKEUP,  
              cal.getTimeInMillis(),  
              pi);  
        am.set(AlarmManager.RTC_WAKEUP,  
              cal2.getTimeInMillis(),  
              pi);  
        am.set(AlarmManager.RTC_WAKEUP,  
              cal3.getTimeInMillis(),  
              pi);  
        am.set(AlarmManager.RTC_WAKEUP,  
              cal4.getTimeInMillis(),  
              pi);  
    }  
    /*  
     * Można wielokrotnie wykorzystywać tę samą intencję,  
     * jeżeli zmienimy identyfikator żądania w oczekującej intencji.  
     */
```

```
* Identyfikator ten odróżnia daną intencję od innych.  
*/  
public void scheduleDistinctIntents()  
{  
    //Wystąpienie zostanie pobrane po 30 sekundach  
    //od bieżącego momentu.  
    Calendar cal = Utils.getTimeAfterInSecs(30);  
    Calendar cal2 = Utils.getTimeAfterInSecs(35);  
    Calendar cal3 = Utils.getTimeAfterInSecs(40);  
    Calendar cal4 = Utils.getTimeAfterInSecs(45);  
  
    //W przypadku gdy chcemy uruchomić alarm dzisiaj o godzinie 11.  
    //Calendar cal = Utils.getTodayAt(11);  
  
    //Wyświetla informację w widoku debugowania, że  
    //ustanawiamy alarm na określoną godzinę.  
    String s = Utils.getDateTimeString(cal);  
    mReportTo.reportBack(tag, "Ustanawianie alarmu na: " + s);  
  
    //Pobiera intencję, aby przywołać  
    //klasę TestReceiver.  
    Intent intent =  
        new Intent(mContext, TestReceiver.class);  
    intent.putExtra("message", "Ustanawia oddzielne alarmy");  
  
    //Tworzy harmonogram tej samej intencji, ale za pomocą różnych id. żądań.  
    AlarmManager am =  
        (AlarmManager)  
            mContext.getSystemService(Context.ALARM_SERVICE);  
  
    am.set(AlarmManager.RTC_WAKEUP,  
        cal.getTimeInMillis(),  
        getDistinctPendingIntent(intent,1));  
  
    am.set(AlarmManager.RTC_WAKEUP,  
        cal2.getTimeInMillis(),  
        getDistinctPendingIntent(intent,2));  
    am.set(AlarmManager.RTC_WAKEUP,  
        cal3.getTimeInMillis(),  
        getDistinctPendingIntent(intent,3));  
    am.set(AlarmManager.RTC_WAKEUP,  
        cal4.getTimeInMillis(),  
        getDistinctPendingIntent(intent,4));  
}  
}
```

---

W kodzie metody `scheduleSameIntentMultipleTimes()` wykorzystaliśmy tę samą intencję i za jej pomocą utworzyliśmy cztery alarmy. Przekonamy się o tym, wybierając element menu *Wiele alarmów*; został uruchomiony jedynie ostatni alarm, pozostałe nie zadziały.

Zalecanym rozwiązaniem jest taka zmiana kodu, aby każda intencja oczekująca posiadała inny identyfikator żądania. Dlatego właśnie korzystamy z funkcji `getDistinctPendingIntent()`, która szybko tworzy oczekujące intencje na podstawie identyfikatora żądania. Na listingu 15.16 widzimy kod źródłowy tej funkcji.

Spójrzcie na metodę `scheduleDistinctIntents()` z listingu 15.18 pomoże nam rozwiązać problem duplikujących się intencji. Widzimy tu zróżnicowane identyfikatory żądań, dzięki czemu odbiorca `TestReceiver` zostanie wywołany wiele razy, czego dowód znajdziemy w widoku `LogCat`.

Twórcy Androida usilnie zalecają, aby pamiętać o następujących zasadach podczas tworzenia intencji oczekujących:

Starajmy się nie tworzyć wielu różnorodnych intencji oczekujących. Musimy być ostrożni, w przypadku gdy tworzymy wiele unikatowych intencji oczekujących, różniących się identyfikatorem żądania oraz pozostałymi aspektami intencji.

Spodziewamy się po oczekującej intencji, że będzie szybko odtwarzana przez obiekt ją wysyłający po to, aby mogła zostać anulowana. Wynika z tego pewien naturalny porządek tworzenia intencji oczekujących. W idealnym przypadku parametry wykorzystywane do jej tworzenia powinny być niepowtarzalne. Jeżeli tak nie jest, a my musimy wykorzystywać identyfikatory żądania do odróżnienia intencji od innych, powinniśmy zapamiętać wartości tych identyfikatorów. Będą nam one potrzebne w momencie, gdy będziemy chcieli anulować te intencje oczekujące.

W przypadku gdy nie mamy identyfikatorów żądania, dwie intencje oczekujące wskazują tę samą intencję, jeżeli ich kluczowe atrybuty są takie same. W tym przypadku nie są tu brane pod uwagę dodatkowe dane intencji.

W przypadku intencji oczekujących metody pobierające zazwyczaj lokalizują już istniejącą intencję, a nie tworzą nowej.

Standardowo intencje oczekujące powinny wskazywać określona klasę lub składnik.

## Kompilowanie kodu omawianego przykładu

Aby przetestować ten fragment kodu, będziemy musieli zmienić zawartość kilku plików w projekcie.

Najpierw musimy dodać klasę widoczną na listingu 15.18 w postaci nowego pliku źródłowego o nazwie `CancelRepeatingAlarmTester.java`.

Następnie musimy zmienić w kilku miejscach aktywność sterującą `TestAlarmsDriverActivity`, zdefiniowaną na listingu 15.12.

Zastąpmy poniższe wiersze:

```
private SendAlarmOnceTester alarmTester = null;
...
alarmTester = new SendAlarmOnceTester(this, this);
```

następującymi:

```
private ScheduleIntentMultipleTimesTester alarmTester = null;
...
alarmTester = new ScheduleIntentMultipleTimesTester (this, this);
```

Dodajmy poniższy kod, umożliwiający odpowiedź na dwa elementy menu:

```
if (item.getItemId() == R.id.menu_alarm_multiple)
{
    alarmTester.scheduleSameIntentMultipleTimes();
    return true;
}
if (item.getItemId() == R.id.menu_alarm_distinct_intents)
```

```
{  
    alarmTester.scheduleDistinctIntents();  
    return true;  
}
```

Po wprowadzeniu tych zmian możemy przetestować funkcje dostępne w tym przykładzie poprzez kliknięcie dwóch elementów menu: *Wiele alarmów* i *Oddzielne intencje*. Wyniki testu ujrzymy w oknie *LogCat*.

## Pierwszeństwo intencji w uruchamianiu alarmów

Wielokrotnie już wspominaliśmy, że w przypadku skonfigurowania kilku alarmów wobec tego samego typu intencji zostanie uruchomiony tylko ostatni utworzony alarm. Spróbujmy znaleźć wyjaśnienie, dlaczego tak się dzieje. Przyglądając się naszym przykładowym kodom, możemy uznać, że konfiguruujemy alarm w menedżerze alarmów. Przynajmniej takie wrażenie sprawia interfejs API, w którym znajdziemy następującą metodę:

```
alarmManager.set(time, intent);
```

Załóżmy jednak następującą sytuację:

```
alarmManager.set(time1, intent1);  
alarmManager.setRepeated(time2, interval, intent1);
```

Moglibyśmy się spodziewać, że obiekt `intent1` może być wyłącznie pasywnym odbiorcą, który jest odbierany przez obydwa alarmy. Jednak w praktyce okazuje się, że liczy się wyłącznie ostatnia ustawiona metoda. Jest to tak, jakbyśmy konfigurowali intencję, tak jak w poniższym przykładzie:

```
intent1.set(...)  
intent1.setRepeated(...)
```

W tym przypadku prawdopodobnie nabiera sensu ustwienie tylko jednego obiektu intencji oraz związanego z nim alarmu, a także stwierdzenie, że jeśli ustanowimy ten alarm wiele razy, każdy poprzedni alarm zostanie zresetowany, podobnie jak ma to miejsce w przypadku zwykłego budzika.

Tę koncepcję możemy sprawdzić za pomocą kodu umieszczonego na listingu 15.19. Interesującą nas tutaj metodą jest `alarmIntentPrimacy()`.

---

**Listing 15.19.** Kod służący do przetestowania pierwszeństwa intencji

---

```
//AlarmIntentPrimacyTester.java  
package com.androidbook.alarms;  
import java.util.Calendar;  
import android.app.AlarmManager;  
import android.app.PendingIntent;  
import android.content.Context;  
import android.content.Intent;  
  
public class AlarmIntentPrimacyTester  
extends ScheduleIntentMultipleTimesTester  
{  
    private static String tag = "AlarmIntentPrimacyTester";  
    AlarmIntentPrimacyTester(Context ctx, IReportBack target){  
        super(ctx, target);
```

```

}

/*
 * Tutaj nie alarm ma znaczenie,
 * lecz oczekująca intencja.
 * Nawet jeśli zdefiniujemy powtarzalny alarm,
 * w przypadku wykorzystywania tej samej intencji za jednym razem
 * tylko ta późniejsza intencja odniesie skutek.
 *
 * Możemy to porównać do wielokrotnego konfigurowania
 * alarmu wobec istniejącej intencji,
 * a nie okrężnym sposobem.
 */
public void alarmIntentPrimacy()
{
    Calendar cal = Utils.getTimeAfterInSecs(30);
    String s = Utils.getDateTimeString(cal);
    this.mReportTo.reportBack(tag,
        "Ustanawianie powtarzalnego alarmu o interwale 5 s, począwszy od: " + s);

    //Pobiera intencję służącą do przywołania
    //klasy TestReceiver.
    Intent intent =
        new Intent(this.mContext, TestReceiver.class);
    intent.putExtra("message", "Powtarzalny alarm");

    PendingIntent pi = getDistinctPendingIntent(intent,0);
    AlarmManager am =
        (AlarmManager)
            this.mContext.getSystemService(Context.ALARM_SERVICE);

    this.mReportTo.reportBack(tag,"Konfig. powtarzalnego alarmu, interwał - 5 s");
    am.setRepeating(AlarmManager.RTC_WAKEUP,
        cal.getTimeInMillis(),
        5*1000, //5 sekund
        pi);

    this.mReportTo.reportBack(tag,"Konfig. jednorazowego alarmu dla tej samej
        intencji");
    am.set(AlarmManager.RTC_WAKEUP,
        cal.getTimeInMillis(),
        pi);
    this.mReportTo.reportBack(tag,
        "Późniejszy alarm, jednorazowy, posiada pierwszeństwo");
}
}

```

## Kompilowanie kodu omawianego przykładu

Aby przetestować ten fragment kodu, będziemy musieli zmienić zawartość kilku plików w projekcie.

Najpierw musimy dodać klasę widoczną na listingu 15.19 w postaci nowego pliku źródłowego o nazwie *AlarmIntentPrimacyTester.java*.

Następnie musimy zmienić w kilku miejscach aktywność sterującą `TestAlarmsDriverActivity`, zdefiniowaną na listingu 15.12.

Zastąpmy poniższe wiersze:

```
private SendAlarmOnceTester alarmTester = null;  
"  
alarmTester = new SendAlarmOnceTester(this, this);  
następującymi:
```

```
private AlarmIntentPrimacyTester alarmTester = null;  
"  
alarmTester = new AlarmIntentPrimacyTester (this, this)
```

Dodajmy poniższy kod, umożliwiający odpowiedź na zdarzenie menu:

```
if (item.getItemId() == R.id.menu_alarm_intent_primacy)  
{  
    alarmTester.alarmIntentPrimacy();  
    return true;  
}
```

Po wprowadzeniu tych zmian możemy przetestować funkcje dostępne w tym przykładzie poprzez kliknięcie elementu menu *Pierwszeństwo intencji*. Wyniki testu ujrzymy w oknie *LogCat*, w którym będą również wyświetlane nadpisywanie starszych alarmów przez alarmy nowsze.

Dlaczego starszy alarm jest zastępowany nowszym w przypadku korzystania z tej samej intencji?

Wiele osób spośród zespołu twórców Androida zauważa, że dwie intencje są tak naprawdę wystąpieniem tego samego obiektu `PendingIntent`, jeżeli wartości ich atrybutów są takie same. Ustanawianie takich intencji jako celów dla wielu alarmów jest tożsame z ustanawianiem alarmów dla jednej i tej samej intencji.

Jednak właściwy mechanizm zrozumiemy dopiero po przyjrzeniu się kodowi źródłowemu usługi `AlarmManagerService` (jest to implementacja interfejsu `IAlarmManager`). Na listingu 15.20 został zamieszczony fragment kodu odpowiedzialny za ustanawianie alarmu (wszystkie metody ustawiające ostatecznie przechodzą przez ten fragment).

---

**Listing 15.20.** Implementacja klasy `AlarmManagerService`, wyciąg z kodu źródłowego Androida

```
160     public void setRepeating(int type, long triggerAtTime, long interval,  
161                         PendingIntent operation) {  
162         if (operation == null) {  
163             Slog.w(TAG, "set/setRepeating ignored because there is no intent");  
164             return;  
165         }  
166         synchronized (mLock) {  
167             Alarm alarm = new Alarm();  
168             alarm.type = type;  
169             alarm.when = triggerAtTime;  
170             alarm.repeatInterval = interval;  
171             alarm.operation = operation;  
172  
173             // Usuwa ten alarm, jeżeli już został wstawiony do harmonogramu  
174             removeLocked(operation);  
175         }
```

```

176         if (localLOGV) Slog.v(TAG, "set: " + alarm);
177
178         int index = addAlarmLocked(alarm);
179         if (index == 0) {
180             setLocked(alarm);
181         }
182     }
183 }
```

Zwróćmy uwagę, że w samym środku metody ustawiania kod wywołuje metodę `removeLocked(→(operation))`, gdzie argumentem `operation` jest obiekt `PendingIntent`. Ta metoda jest właśnie odpowiedzialna za usunięcie wcześniej występującego alarmu. W istocie, gdy wywołujemy metodę `cancel(pendingIntent)`, ostatecznie zostaje również wywołana ta sama metoda `removeLocked(pendingIntent)`.

Mówiąc krótko, zestaw SDK postanowił anulować wszystkie wcześniejsze alarmy i dla tej konkretnej intencji oczekującej zostawić tylko najnowszy alarm. Jeżeli chcemy, aby stało się inaczej, musimy ustawić dla tej intencji identyfikator żądania. Staje się to również jasne, gdy zapoznamy się z interfejsem API metody `cancel()`, która pobiera wyłącznie argument w postaci obiektu `PendingIntent`. Gdyby związek pomiędzy alarmem a obiektem `PendingIntent` nie miał niepowtarzalnego charakteru, jakie znaczenie miałyby anulowanie alarmu wyłącznie na podstawie obiektu `PendingIntent`?

Oczywiście, możemy wykorzystać ten mechanizm na naszą korzyść, jeżeli naszym celem będzie anulowanie wszelkich wcześniejszych alarmów i ustanowienie nowego dla danego odbiorcy.

## Trwałość alarmów

Czytelnikowi należy się jeszcze jedna informacja dotycząca alarmów: nie są one zachowywane po ponownym uruchomieniu urządzenia. Oznacza to, że będziemy musieli utrzymywać ustawienia alarmu oraz intencji oczekującej w pamięci trwalej oraz ponownie je rejestrować na podstawie komunikatów nadawanych podczas ponownego uruchamiania urządzenia oraz, ewentualnie, komunikatów związanych ze zmianą czasu (na przykład `android.intent.action.→BOOT_COMPLETED`, `ACTION_TIME_CHANGED`, `ACTION_TIMEZONE_CHANGED`).

## Twierdzenia dotyczące menedżera alarmów

Podsumujmy ten rozdział skrótownym wypisaniem faktów dotyczących alarmów, intencji oczekujących oraz menedżera alarmów:

- Intencje oczekujące są przechowywane w puli oraz mogą być wielokrotnie używane. Tak naprawdę nie możemy utworzyć nowej intencji oczekującej. W rzeczywistości lokalizujemy taką intencję za pomocą opcji pozwalających na jej ponowne użycie, aktualizowanie itp.
- Intencja jest niepowtarzalnie odróżnialna od innych poprzez jej argumenty działania, identyfikator URI danych oraz kategorię. Szczegóły tej niepowtarzalności zostały zdefiniowane w interfejsie API `filterEquals()` klasy intencji.
- Intencja oczekująca jest kwalifikowana za pomocą kodu żądania (w dodatku do bazowej intencji, od której jest uzależniona).

- Alarms and intents waiting (in addition to all other intents) are not guaranteed. Awaiting intent can not be associated with many alarms. The latest alarm will ignore all others.
- Alarms are not stored during a device's re-boot. Any alarms created by the alarm manager will be lost at the moment of the device's re-boot.
- We must keep the alarm parameters if we want to reuse them. We must listen to notifications about the device's re-boot and changes in time to reset the alarms.
- If we use the cancel() method, we must also store the intent so that we can cancel the alarm later.

## Odbośniki

Poniższe odnośniki stanowią dodatkowy materiał do informacji omawianych w niniejszym rozdziale. Warto zwłaszcza zwrócić uwagę na ostatni adres URL, gdyż prowadzi do strony z projektami, które możemy pobrać i zainportować w środowisku Eclipse.

- <http://developer.android.com/reference/android/app/AlarmManager.html> — znajdziemy tu interfejs menedżera alarmów. Zostały tu omówione takie metody, jak set(), setRepeating() czy cancel().
- <http://developer.android.com/reference/android/app/PendingIntent.html> — na tej stronie został wyjaśniony mechanizm konstruowania intencji oczekujących. Nie zwracajmy zbytnio uwagi na flagi intencji oczekujących; nie są one niezbędne w przypadku menedżera alarmów.
- <http://www.androidbook.com/item/1040> — kilka krótkich przykładów oraz dalsze odniesienia do informacji o klasach związanych z datą i czasem.
- [http://download.oracle.com/docs/cd/E17476\\_01/javase/1.4.2/docs/api/java/util/Calendar.html](http://download.oracle.com/docs/cd/E17476_01/javase/1.4.2/docs/api/java/util/Calendar.html) — dzięki zawartym tu zasobom lepiej zrozumiemy zasady pracy z obiektem Calendar.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów wykorzystywanych w tej książce. Katalog zawierający projekty z tego rozdziału nosi nazwę ProAndroid3\_R15\_MenedżerAlarmów.

## Podsumowanie

W tym rozdziale pokazaliśmy, jak wykorzystać menedżer alarmów do uruchomienia kodu o określonej porze oraz we wskazanych przedziałach czasowych. Jest to bardzo ważna funkcja, wykorzystywana do aktualizacji widżetów ekranu startowego oraz w innych czynnościach zależnych od czasu. Wymieniliśmy również pewne specyficzne cechy związane z tym menedżerem i pokazaliśmy sposoby rozwiązywania wynikających z nich problemów.

# Analiza animacji dwuwymiarowej

Animacja jest procesem pozwalającym wyświetlanemu na ekranie obiektyowi na zmianę koloru, pozycji, rozmiaru lub orientacji w określonym przedziale czasowym. Animacje, które można wykorzystać w Androidzie, są bardzo praktyczne, zabawne, proste oraz są często wykorzystywane.

W wersji 2.3 i wcześniejszych Androida dostępne są trzy rodzaje animacji: animacja poklatkowa, która polega na rysowaniu serii klatek jedna po drugiej w regularnych odstępach czasu; animacja układu graficznego, w której są przetwarzane widoki osadzone w pojemniku, na przykład w tabeli na liście; a także animacja widoku, polegająca na animowaniu dowolnego widoku ogólnego przeznaczenia. Dwa ostatnie rodzaje należą do kategorii animacji klatek kluczowych (ang. *tweening*), która polega na interpolowaniu przez komputer klatek pośrednich pomiędzy klatkami kluczowymi.

**Uwaga!**

W wersji 3.0 Androida zmodernizowano mechanizm animacji, gdyż wprowadzono możliwość animowania elementów interfejsu użytkownika. Niektóre z nowych koncepcji, zwłaszcza dotyczące fragmentów, zostały omówione w rozdziale 29. Niniejszy rozdział ukończylismy przed wydaniem wersji 3.0 Androida, więc z powodu ograniczeń czasowych zajęliśmy się w nim jedynie elementami dostępnymi do wersji 2.3 systemu. W rozdziale 29. opisaliśmy kilka funkcji animacji dostępnych od wersji 3.0 Androida.

Animację klatek kluczowych można wyjaśnić również w taki sposób, że *nie* wymaga ona rysowania klatki po klatce. Jeżeli możemy animować obiekt bez konieczności nakładania i powtarzania kolejnych klatek, to mamy do czynienia z techniką klatek kluczowych. Jeżeli na przykład dany obiekt znajduje się w punkcie A, a za 4 sekundy znajdzie się w punkcie B, możemy zmieniać jego położenie co sekundę i za każdym razem od nowa go rysować. Będziemy odnosić wrażenie, że obiekt ten porusza się z punktu A do punktu B.

Koncepcja jest taka, że znajomość początkowych i końcowych stanów animowanego obiektu pozwala grafikowi na zmianę pewnych aspektów tego obiektu w trakcie procesu animowania. Takim aspektem może być kolor, pozycja, rozmiar albo jakiś inny element. W komputerach jest to osiągane poprzez zmianę średnich wartości w regularnych odstępach czasu oraz ponowne rysowanie powierzchni.

W tym rozdziale zajmiemy się zagadnieniami animacji poklatkowej, układu graficznego oraz widoku — zaprezentujemy je z wykorzystaniem działających przykładów oraz poddamy je do głębszej analizie.

**Uwaga!**

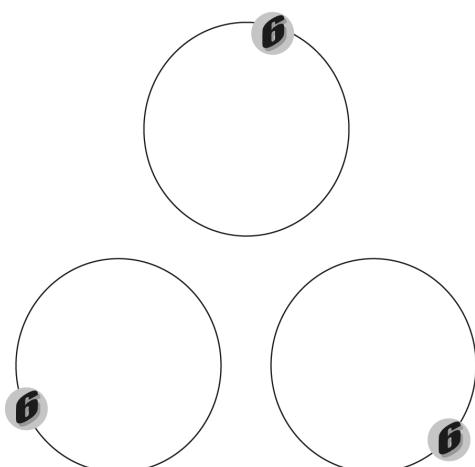
Na końcu rozdziału zamieściliśmy adres URL, z którego możemy pobrać projekty utworzone na potrzeby tego rozdziału i zimportować je do środowiska Eclipse.

## Animacja poklatkowa

Animacja jest prostym procesem polegającym na wyświetlaniu serii obrazów następujących po sobie w krótkich odstępach czasu, w wyniku czego powstaje wrażenie poruszającego lub zmieniającego się obiektu. W taki sposób działają projektory filmowe. Pokażemy przykładowy projekt, w którym zaprojektujemy obraz i zapiszemy go w formie serii oddzielnych klatek, różniących się od siebie w niewielkim stopniu. Następnie umieścimy ten zbiór obrazów w przykładowym kodzie umożliwiającym uruchomienie animacji.

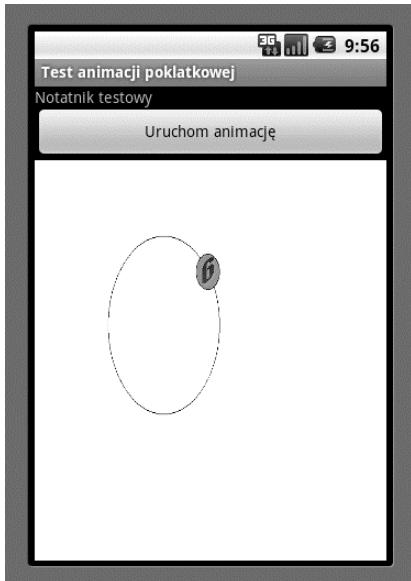
## Zaplanowanie animacji poklatkowej

Przed rozpoczęciem pisania kodu należy najpierw zaplanować sekwencję animacji za pomocą rozrysowania jej na papierze. Przykład takiego planowania został zilustrowany na rysunku 16.1, przedstawiającym zbiór równowymiarowych okręgów, na których obwodach zostały umieszczone w różnych pozycjach kolorowe kółka. Można stworzyć zbiór takich rysunków przedstawiających okrąg oraz kółko umieszczone w różnych miejscach na obwodzie tego okręgu. Po zachowaniu siedmiu lub ośmiu takich klatek utworzymy animację symulującą ruch kółka po okręgu.



Rysunek 16.1. Etap projektowania animacji

Określmy sobie podstawowy człon nazwy takiego rysunku, na przykład colored-ball, a następnie zachowajmy utworzone rysunki w podkatalogu /res/drawable, żeby w przyszłości można było uzyskać do nich dostęp za pomocą identyfikatorów zasobów. Nazwa każdego pliku powinna zostać utworzona za pomocą wzoru colored-ballN, gdzie N jest numerem porządkowym klatki. Po utworzeniu animacji powinna ona wyglądać tak jak na rysunku 16.2.



**Rysunek 16.2.** Środowisko testowe animacji poklatkowej

Główny obszar aktywności jest wykorzystywany przez widok animacji. Wstawiliśmy przycisk uruchamiania i zatrzymywania animacji w celu obserwacji jej zachowania. W górnej części ekranu umieściliśmy również notatnik testowy, w którym można zapisywać wszelkie ważne zdarzenia podczas eksperymentowania z programem. Zobaczmy, w jaki sposób można utworzyć układ graficzny takiej aktywności.

## Utworzenie aktywności

Rozpoczniemy od utworzenia prostego pliku XML układu graficznego w podkatalogu /res/layout (listing 16.1).

**Listing 16.1.** Plik XML układu graficznego do przykładu animacji poklatkowej

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- nazwa pliku: /res/layout/frame_animations_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView android:id="@+id/textViewId1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
    android:text="Notatnik testowy"
  />
<Button
  android:id="@+id/startFAButtonId"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="Uruchom animację"
/>
<ImageView
  android:id="@+id/animationImage"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  />
</LinearLayout>
```

---

Pierwszą kontrolką jest kontrolka notatnika testowego stanowiąca prosty widok `TextView`. Następnie dodajemy przycisk uruchamiania i zatrzymywania animacji. Ostatni jest widok `ImageView`, w którym będzie odtwarzana animacja. Po skonstruowaniu układu graficznego należy utworzyć aktywność wczytującą ten widok (listing 16.2).

#### **Listing 16.2. Aktywność wczytująca widok ImageView**

---

```
public class FrameAnimationActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.frame_animations_layout);
    }
}
```

---

Taką aktywność będzie można uruchomić za pomocą dowolnego elementu menu, dostępnego w bieżącej aplikacji, poprzez wykonanie następującego kodu:

```
Intent intent = new Intent(inActivity,FrameAnimationActivity.class);
inActivity.startActivity(intent);
```

W tym momencie aktywność powinna wyglądać tak jak na rysunku 16.3.

## **Dodawanie animacji do aktywności**

Po utworzeniu aktywności oraz układu graficznego pokażemy, w jaki sposób można do nich dodawać animację. W Androidzie animacja poklatkowa jest tworzona poprzez klasę `AnimationDrawable` z pakietu graficznego. Można stwierdzić po nazwie, że będzie się ona zachowywać jak każdy inny obiekt rysowany, mogący stanowić tło dla dowolnego widoku (na przykład mapy bitowe ta są reprezentowane jako elementy `Drawable`). Klasa `AnimationDrawable` poza tym, że należy do kategorii `Drawable`, może pobierać listę innych obiektów tego typu (na przykład obrazów) i wyświetlać je w określonych interwałach czasowych. W rzeczywistości klasa ta jest cienką osłoną wokół obsługi animacji zapewnianej przez bazową klasę `Drawable`.



**Rysunek 16.3.** Aktywność animacji poklatkowej

**Wskazówka**

Klasa `Drawable` uruchamia animację w ten sposób, że pojemnik lub widok wywołuje klasę `Runnable`, która w istocie przerysowuje obiekt `Drawable` za pomocą innego zestawu parametrów. Zwróćmy uwagę, że nie musimy znać takich szczegółów wewnętrznej implementacji, żeby korzystać z klasy `AnimationDrawable`. Jednak w przypadku bardziej złożonych wymagań można zatrzymać się do kodu źródłowego klasy `AnimationDrawable`, aby znaleźć wskazówki do napisania własnych protokołów animacji.

Żeby móc skorzystać z klasy `AnimationDrawable`, należy najpierw umieścić zestaw zasobów typu `Drawable` (na przykład zbiór obrazów) w podkatalogu `/res/drawable`. Gwoli ściśliwości, umieścimy tam osiem podobnych, lecz nie identycznych obrazów omówionych w punkcie „Zaplanowanie animacji poklatkowej”. Następnie utworzymy plik XML definiujący listę klatek (listing 16.3). Także ten plik musi zostać umieszczony w podkatalogu `/res/drawable`.

**Listing 16.3.** Plik XML definiujący listę animowanych klatek

---

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
    <item android:drawable="@drawable/colored_ball1" android:duration="50" />
    <item android:drawable="@drawable/colored_ball2" android:duration="50" />
    <item android:drawable="@drawable/colored_ball3" android:duration="50" />
    <item android:drawable="@drawable/colored_ball4" android:duration="50" />
    <item android:drawable="@drawable/colored_ball5" android:duration="50" />
    <item android:drawable="@drawable/colored_ball6" android:duration="50" />
    <item android:drawable="@drawable/colored_ball7" android:duration="50" />
    <item android:drawable="@drawable/colored_ball8" android:duration="50" />
</animation-list>
```

---

**Uwaga!**

Podczas przygotowywania listy obrazów musimy pamiętać o pewnych ograniczeniach klasy AnimationDrawable. Przed rozpoczęciem animacji klasa ta wczytuje wszystkie obrazy do pamięci. Podczas testowania przykładowego projektu na emulatorze wyposażonym w wersję systemu 2.3 liczba klatek większa od 6 przekraczała pojemność pamięci przydzielonej dla aplikacji. W zależności od środowiska testowego być może będziemy musieli ograniczyć liczbę klatek. Aby rozwiązać ten problem, musimy bezpośrednio skorzystać z funkcji animacyjnych klasy Drawable i wprowadzić własny mechanizm. Niestety, klasa Drawable nie została szczegółowo omówiona w tym wydaniu książki. Proponujemy wizytę na stronie [www.androidbook.com](http://www.androidbook.com), gdzie planujemy zaktualizować jej zawartość w niedługim czasie.

Każda klatka wskazuje na jeden z rysunków określony przez jego identyfikator zasobu. Znacznik `animation-list` zostaje przekształcony do obiektu `AnimationDrawable`, reprezentującego zbiór obrazów. Musimy teraz umieścić klasę `Drawable` jako zasób tła dla widoku `ImageView`. Zakładając, że nazwaliśmy ten plik `frame_animation.xml` i umieściliśmy go w podkatalogu `/res/drawable`, możemy zastosować poniższy kod do ustanowienia klasy `AnimationDrawable` jako tła widoku `ImageView`:

```
view.setBackgroundResource(Resource.drawable.frame_animation);
```

Dzięki tej linii kodu Android rozpoznaje identyfikator zasobu `Resource.drawable.frame_animation` jako zasób XML i zgodnie z nim tworzy odpowiedni obiekt Java `AnimationDrawable`, zanim ustawi go jako tło. Gdy już będziemy mieli tło, możemy uzyskać dostęp do tego obiektu `AnimationDrawable` poprzez wprowadzenie instrukcji `get` do widoku `View` w następujący sposób:

```
Object backgroundObject = view.getBackground();
AnimationDrawable ad = (AnimationDrawable)backgroundObject;
```

Po umieszczeniu obiektu klasy `AnimationDrawable` możemy wprowadzić metody `start()` i `stop()` służące do uruchamiania i zatrzymywania animacji. Poniżej zaprezentowaliśmy dwie inne istotne metody tego obiektu:

```
setOneShot();
addFrame(drawable, duration);
```

Metoda `setOneShot()` odtwarza animację jeden raz i potem ją zatrzymuje. Metoda `addFrame()` dodaje nową klatkę za pomocą obiektu `Drawable` i konfiguruje czas jej wyświetlania. Działanie tej metody przypomina funkcję znacznika XML `android:drawable`.

Teraz musimy złożyć wszystkie fragmenty kodu w całość, aby otrzymać środowisko testowe animacji poklatkowej (listing 16.4).

#### **Listing 16.4.** Pełny kod środowiska testowego animacji poklatkowej

```
public class FrameAnimationActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.frame_animations_layout);
        this.setupButton();
    }
    private void setupButton()
    {
```

```

Button b = (Button)this.findViewById(R.id.startFAButtonId);
b.setOnClickListener(
    new Button.OnClickListener(){
        public void onClick(View v)
        {
            parentButtonClicked(v);
        }
    });
}

private void parentButtonClicked(View v)
{
    animate();
}

private void animate()
{
    ImageView imgView = (ImageView) findViewById(R.id.animationImage);
    imgView.setVisibility(ImageView.VISIBLE);
    imgView.setBackgroundDrawable(R.drawable.frame_animation);

    AnimationDrawable frameAnimation =
        (AnimationDrawable) imgView.getBackground();

    if (frameAnimation.isRunning())
    {
        frameAnimation.stop();
    }
    else
    {
        frameAnimation.stop();
        frameAnimation.start();
    }
}
}

//eof-class

```

Metoda `animate()` lokalizuje widok `ImageView` w bieżącej aktywności i przypisuje mu tło `AnimationDrawable` rozpoznane przez zasób `R.drawable.frame_animation`. Następnie kod odczytuje ten obiekt i przeprowadza proces animowania. Przycisk uruchamiania i zatrzymywania jest skonfigurowany w ten sposób, że jego naciśnięcie w trakcie odtwarzania animacji zatrzyma ją; jeżeli animacja jest zatrzymana, jego naciśnięcie spowoduje jej uruchomienie.

Zauważmy, że jeśli przypiszemy parametrowi listy animacji `OneShot` wartość `true`, animacja wykona tylko jeden cykl. Jednak nie można dokładnie przewidzieć, kiedy się to stanie. Chociaż animacja zostaje zakończona po wyświetleniu ostatniego obrazu, nie otrzymamy żadnego informującego o tym komunikatu. Z tego powodu nie istnieje żaden bezpośredni sposób wywołania kolejnej czynności w odpowiedzi na zakończenie animacji.

Pomimo tej niedogodności można uzyskać wspaniałe efekty wizualne poprzez wyświetlanie po kolei serii obrazów w prostym procesie animacji poklatkowej.

## Animacja układu graficznego

Podobnie jak w przypadku animacji poklatkowej, animacja układu graficznego jest bardzo prosta. Jak sama nazwa wskazuje, animacja tego typu jest przeznaczona dla pewnego rodzaju widoków,

ułożonych w określony sposób. Stosowana jest ona na przykład w przypadku widoków ListView oraz GridView, które są dwiema powszechnie implementowanymi kontrolkami w systemie Android. W szczególności animacja układu graficznego jest używana do dodawania efektów graficznych, zmieniających sposób wyświetlania elementów umieszczonych w wymienionych widokach. Tak naprawdę może być ona stosowana wobec wszystkich kontrolek wywodzących się z klasy ViewGroup.

W przeciwieństwie do animacji poklatkowej, animacja układu graficznego nie jest generowana poprzez powtarzanie klatek. Zamiast tego są zmieniane w czasie różne właściwości widoku. Każdy widok w Androidzie zawiera macierz transformacji, która odwzorowuje widok wyświetlony na ekranie. Poprzez zmianę takiej macierzy na różne sposoby można przeprowadzić procesy skalowania, obracania i przemieszczania (translacji) tego widoku. Na przykład poprzez zmianę przezroczystości widoku w skali od 0 do 1 otrzymujemy tak zwaną animację typu alfa.

## Podstawowe typy animacji klatek kluczowych

Poniżej prezentujemy nieco bardziej szczegółowo podstawowe rodzaje animacji klatek kluczowych (ang. *tweening*):

- **Animacja skali.** Ten typ animacji umożliwia powiększanie lub zmniejszanie widoku w osi *x* oraz w osi *y*. Można także określić punkt zwrotny, wokół którego będzie odtwarzana animacja.
- **Animacja rotacyjna.** Dzięki niej można obracać widok wokół punktu zwrotnego o określony kąt.
- **Animacja translacyjna.** Służy ona do przesuwania widoku wzdłuż osi *x* lub *y*.
- **Animacja typu alfa.** Służy do zmieniania przezroczystości widoku.

Animacje tego typu są definiowane w postaci plików XML umieszczonych w podkatalogu /res/anim. Listing 16.5 prezentuje krótki przykład, pomagający zrozumieć, w jaki sposób te animacje są definiowane.

**Listing 16.5.** Animacja skali zdefiniowana w pliku XML, umieszczonym w podkatalogu /res/anim/scale.xml

---

```
<set xmlns:android="http://schemas.android.com/apk/res/android"  
      android:interpolator="@android:anim/accelerate_interpolator">  
    <scale  
        android:fromXScale="1"  
        android:toXScale="1"  
        android:fromYScale="0.1"  
        android:toYScale="1.0"  
        android:duration="500"  
        android:pivotX="50%"  
        android:pivotY="50%"  
        android:startOffset="100" />  
</set>
```

---

Wszystkie wartości parametrów w tym pliku animacji zostają określone „od – do”, ponieważ musimy określić wartości początkowe i końcowe animacji.

Każda z animacji dopuszcza również możliwość korzystania z interpolatorów czasu w postaci argumentów. Interpolatory zostaną omówione na końcu podrozdziału związanego z animacją układu graficznego, teraz jednak wystarczy wiedzieć, że są one odpowiedzialne za szybkość zmian argumentów w trakcie przetwarzania animacji.

Po utworzeniu tego pliku deklarowania animacji możemy powiązać animację z układem graficznym, dzięki czemu elementy składowe układu graficznego będą animowane.

### Uwaga!

W tym miejscu warto wspomnieć, że każda z tych animacji jest reprezentowana jako klasa Java w pakiecie `android.view.animation`. Dokumentacja każdej z tych klas nie tylko opisuje jej metody języka Java, lecz również dopuszczalne argumenty XML dla każdego typu animacji.

Skoro już naszkicowaliśmy zarys rodzajów animacji układu graficznego wystarczający do ich chociażby podstawowego zrozumienia, zajmijmy się projektowaniem przykładu.

## Zaplanowanie środowiska testowego animacji układu graficznego

Za pomocą prostego zestawu `ListView` w aktywności można przetestować wszystkie omówione przez nas koncepcje animacji układu graficznego. Po utworzeniu widoku `ListView` można do niego dołączyć animację, co spowoduje jej przetworzenie wobec każdego elementu tego widoku.

Załóżmy, że chcemy utworzyć animację skali, która powiększa widok od zera do oryginalnego rozmiaru w osi *y*. Możemy to sobie wyobrazić wizualnie jako linijkę tekstu, która najpierw przypomina poziomą linię, a następnie zostaje powiększona do właściwego rozmiaru czcionki.

Można tę animację dołączyć do widoku `ListView`. Kiedy to zrobimy, każdy element tej listy będzie wyświetlany za pomocą tej animacji.

Możemy dodać kilka parametrów, które urozmaicą podstawową animację, na przykład animowanie listy od góry do dołu lub odwrotnie. Parametry te są definiowane w klasie pośredniej, zachowującej się jak mediator pomiędzy konkretnym plikiem XML animacji a widokiem listy.

Istnieje możliwość zdefiniowania zarówno animacji, jak i mediatora w pliku XML umieszczonem w podkatalogu `/res/anim`. Gdy już utworzymy taki pośredniczący plik XML, możemy go wykorzystać w postaci danych wejściowych dla widoku `ListView` w jego własnym pliku definicji XML. Gdy ta podstawowa konfiguracja będzie już działać, będziemy zmieniać animacje, żeby przekonać się, w jaki sposób wpływają one na wyświetlanie elementów widoku `ListView`.

Zanim rozpoczniemy ćwiczenie, przyjrzyjmy się, jak widok `ListView` będzie wyglądał po zakończeniu animacji (rysunek 16.4).

## Utworzenie aktywności oraz widoku `ListView`

Rozpoczniemy od utworzenia układu graficznego XML dla widoku `ListView` przedstawionego na rysunku 16.4, dzięki czemu możliwe będzie wczytanie tego układu graficznego w prostej aktywności. Na listingu 16.6 został umieszczony taki nieskomplikowany układ graficzny z zaimplementowanym widokiem `ListView`. Taki plik należy umieścić w podkatalogu `/res/layout`. Zakładając, że nazwa pliku brzmi `list_layout.xml`, kompletna ścieżka do niego będzie wyglądała następująco: `/res/layout/list_layout.xml`.



Rysunek 16.4. Animowana lista ListView

---

**Listing 16.6.** Plik XML układu graficznego definiujący widok ListView

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- nazwa pliku: /res/layout/list_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <ListView
        android:id="@+id/list_view_id"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        />
</LinearLayout>
```

---

Listing 16.6 przedstawia prosty menedżer `LinearLayout` z umieszczonym wewnątrz niego prostym widokiem `ListView`. Powinniśmy jednak skorzystać z okazji i wyjaśnić pewną rzecz dotyczącą definicji widoku `ListView`, która jest dość marginalnie powiązana z treścią rozdziału. Jeżeli Czytelnik będzie pracował na aplikacji Notepad lub innych przykładowych programach, zauważ zapewne, że identyfikator widoku `ListView` jest przeważnie określany jako `@android:id/list`. Zgodnie z informacjami z rozdziału 3. odniesienie `@android:id/list` wskazuje na identyfikator predefiniowany w przestrzeni nazw `android`. Pytanie brzmi: kiedy należy stosować odniesienie `android:id`, a kiedy nasz własny identyfikator, na przykład `@+id/list_view_id`?

Identyfikatora `@android:id/list` używamy jedynie w przypadku, gdy aktywnością jest `ListActivity`. W przypadku tej aktywności zakłada się, że widok `ListView`, określony przez ten predefiniowany identyfikator, jest dostępny do wczytania. W tym wypadku używamy raczej

aktywności ogólnego przeznaczenia, a nie `ListActivity`, i musimy własnoręcznie zapełnić w jawnym sposób widok `ListView`. W związku z tym nie ma żadnych ograniczeń co do rodzaju identyfikatora, który ma reprezentować tę listę. Jednak można także wykorzystać odniesienie `@android:id/list`, ponieważ nie stwarza to żadnego konfliktu z powodu braku aktywności `ListActivity`.

To taka mała dygresja, warto jednak o niej pamiętać podczas tworzenia własnych widoków `ListView` poza aktywnością `ListActivity`. Gdy już posiadamy układ graficzny wymagany dla aktywności, możemy napisać kod odpowiedzialny za wczytanie tego pliku układu graficznego, dzięki czemu zostanie wygenerowany interfejs użytkownika (listing 16.7).

#### **Listing 16.7.** Kod aktywności odpowiedzialnej za animację układu graficznego

---

```
public class LayoutAnimationActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list_layout);
        setupListView();
    }
    private void setupListView()
    {
        String[] listItems = new String[] {
            " Element 1", " Element 2", " Element 3",
            " Element 4", " Element 5", " Element 6",
        };
        ArrayAdapter listItemAdapter =
            new ArrayAdapter(this
                , android.R.layout.simple_list_item_1
                , listItems);
        ListView lv = (ListView)this.findViewById(R.id.list_view_id);
        lv.setAdapter(listItemAdapter);
    }
}
```

---

Niektóre fragmenty kodu widocznego na listingu 16.7 są oczywiste, ale inne nie. Pierwsza część kodu w zwykły sposób wczytuje widok na podstawie wygenerowanego identyfikatora układu graficznego `R.layout.list_layout`. Naszym zadaniem jest zapełnienie widoku `ListView` z tego układu graficznego sześcioma elementami. Te elementy tekstowe zostały wczytane do tablicy. Musimy ustanowić adapter danych wobec widoku `ListView`, żeby te elementy mogły zostać wyświetlane.

Aby utworzyć wymagany adapter, musimy określić, w jaki sposób każdy z elementów będzie wstawiany podczas wyświetlania listy na ekranie. Układ graficzny określamy za pomocą predefiniowanego układu, znajdującego się w strukturze Androida. W naszym przykładzie układ graficzny wyznaczono następująco:

`android.R.layout.simple_list_item_1`

Innymi dostępnymi układami graficznymi widoku dla tych elementów są:

```
simple_list_item_2  
simple_list_item_checked  
simple_list_item_multiple_choice  
simple_list_item_single_choice
```

Można zatrzymać się do dokumentacji Androida, aby się dowiedzieć, jak te uklady graficzne wyglądają i jak się zachowują. Teraz możemy wywołać tę aktywność za pomocą dowolnego przycisku menu w aplikacji po wstawieniu następującego kodu:

```
Intent intent = new Intent(inActivity, LayoutAnimationActivity.class);  
inActivity.startActivity(intent);
```

Jednak — podobnie jak w przypadku wywołań innych aktywności — musimy zarejestrować aktywność *LayoutAnimationActivity* w pliku *AndroidManifest.xml*, jeżeli powyższe wywołanie aktywności ma zadziałać. Poniżej umieściliśmy potrzebny do tego kod:

```
<activity android:name=".LayoutAnimationActivity"  
        android:label="Testowa aktywność widoku animacji"/>
```

## Animowanie widoku ListView

Po przygotowaniu środowiska testowego (listingi 16.6 i 16.7) Czytelnik dowie się, w jaki sposób wstawiać animację skali do widoku *ListView*. Spójrzmy, jak animacja ta zostaje zdefiniowana w pliku XML (listing 16.8).

**Listing 16.8.** Definiowanie animacji skali w pliku XML

---

```
<set xmlns:android="http://schemas.android.com/apk/res/android"  
      android:interpolator="@android:anim/accelerate_interpolator">  
    <scale  
      android:fromXScale="1"  
      android:toXScale="1"  
      android:fromYScale="0.1"  
      android:toYScale="1.0"  
      android:duration="500"  
      android:pivotX="50%"  
      android:pivotY="50%"  
      android:startOffset="100" />  
  </set>
```

---

Jak już wcześniej wspomnieliśmy, pliki definiujące animacje są przechowywane w podkatalogu */res/anim*.

Przetłumaczmy te atrybuty XML na język polski.

Wagi *from* i *to* są wskaźnikami początku oraz zakończenia procesu powiększania. W naszym wypadku powiększanie rozpoczyna się od wartości 1 i takie pozostaje dla osi *x*. Oznacza to, że element nie będzie powiększany ani zmniejszany w tej osi.

Jednak w przypadku osi *y* powiększanie rozpoczyna się od wartości 0.1 i dąży do 1.0. Innymi słowy, na początku animacji rozmiar obiektu stanowi jedną dziesiątą jego naturalnego rozmiaru, do którego dąży w czasie trwania animacji.

Cała operacja skalowania zajmie 500 milisekund.

Środek działania znajduje się w połowie drogi obydwu osi (50%).

Wartość `startOffset` odnosi się do czasu (wyrażonego w milisekundach), po którym animacja zostanie uruchomiona.

Węzeł nadzędny animacji skali wskazuje na zestaw animacji, który dopuszcza wprowadzenie większej liczby animacji. Omówimy również tego rodzaju przykład. Na razie jednak mamy do dyspozycji tylko jedną animację w zestawie.

Nazwijmy ten plik `scale.xml` i umieścmy go w podkatalogu `/res/anim`. Nie jesteśmy na razie gotowi, żeby wstawić ten plik XML animacji jako argument w widoku `ListView`; widok ten wymaga jeszcze jednego pliku XML, który będzie zachowywał się jak pośrednik pomiędzy widokiem a zestawem animacji. Kod pliku XML, w którym zaimplementowane jest takie połączanie, został pokazany na listingu 16.9.

#### **Listing 16.9.** Definicja dla pliku XML stanowiącego kontroler układu graficznego

---

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
    android:delay="30%"
    android:animationOrder="reverse"
    android:animation="@anim/scale" />
```

---

Również ten plik należy umieścić w podkatalogu `/res/anim`. W naszym przykładzie zakładamy, że plik nosi nazwę `list_layout_controller`. Po przyjrzeniu się definicji pliku pośredniczącego zrozumiemy, dlaczego jest on niezbędny.

W pliku tym zostaje określone, że animacja tej listy powinna przebiegać w odwróconym porządku oraz że animacja każdego elementu będzie opóźniona o 30% względem całkowitego czasu trwania animacji. Znajduje się tu również odniesienie do pliku animacji — `scale.xml`. Zauważmy również, że w kodzie jest użyte odniesienie do tego pliku `@anim/scale` zamiast jego nazwy.

Gdy już posiadamy wymagane pliki XML z danymi wejściowymi, pokażemy, w jaki sposób należy zaktualizować definicję XML widoku `ListView`, żeby obejmowała ona animację XML jako argument. Najpierw przejrzyjmy dotychczas utworzone pliki XML:

```
// pojedyncza animacja skali
/res/anim/scale.xml

// plik pośredniczący
/res/anim/list_layout_controller.xml

// plik układu graficznego widoku aktywności
/res/layout/list_layout.xml
```

Gdy te pliki są gotowe, musimy zmodyfikować plik XML układu graficznego `list_layout.xml` w taki sposób, żeby widok `ListView` wskazywał plik `list_layout_controller.xml` (listing 16.10).

#### **Listing 16.10.** Zaktualizowany kod pliku `List_Layout.xml`

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <ListView
```

```
    android:id="@+id/list_view_id"
    android:persistentDrawingCache="animation|scrolling"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layoutAnimation="@anim/list_layout_controller" />
  />
</LinearLayout>
```

---

Zmienione wiersze zostały wyróżnione pogrubioną czcionką. Kluczowym znacznikiem jest `android:layoutAnimation`, który wskazuje pośredniczący plik XML definiujący kontroler układu graficznego za pomocą znacznika `layoutAnimation` (listing 16.9). Z kolei znacznik `layoutAnimation` odnosi się do animacji, w naszym wypadku animacji skali zdefiniowanej w pliku `scale.xml`.

Android zaleca także wstawienie znacznika `persistentDrawingCache`, który optymalizuje animację i przesuwanie. Więcej informacji na jego temat można znaleźć w dokumentacji środowiska Android SDK.

Po aktualizowaniu pliku `list_layout.xml` zgodnie z listingiem 16.10 wtyczka ADT środowiska Eclipse automatycznie przekompiluje pakiet, uwzględniając wprowadzone zmiany. Gdybyśmy teraz uruchomili aplikację, zobaczylibyśmy, że animacja skali jest przeprowadzana na każdym elemencie. Zdefiniowaliśmy czas trwania animacji na 500 milisekund, zatem ujrzymy wyraźnie zmianę skali podczas rysowania obiektu.

Możemy już eksperymentować z innymi rodzajami animacji. Sprawdzimy teraz animację typu alfa. W tym celu utworzymy plik `/res/anim/alpha.xml` i umieścimy w nim treść listingu 16.11.

#### **Listing 16.11.** Plik alpha.xml do testowania animacji typu alfa

---

```
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
      android:interpolator="@android:anim/accelerate_interpolator"
      android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="1000" />
```

---

Animacja typu alfa jest odpowiedzialna za kontrolę zmiany nasycenia kolorów. W tym przykładzie w ciągu 1000 milisekund (1 sekundy) kolor z przezroczystego staje się w pełni nasycony. Dobrze jest ustawić czas trwania animacji na co najmniej 1 sekundę, w przeciwnym wypadku zmiana nasycenia będzie trudna do zaobserwowania.

W przypadku zmiany animacji pojedynczego elementu musimy zmienić również treść pliku pośredniczącego (listing 16.9), żeby wskazywała plik z nową animacją. Poniżej pokazaliśmy sposób wskazywania z animacji skali na animację typu alfa:

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
      android:delay="30%"
      android:animationOrder="reverse"
      android:animation="@anim/alpha" />
```

Zmieniony wiersz w tym kodzie wyróżniono pogrubioną czcionką. Spróbujmy teraz stworzyć animację łączącą zmianę położenia ze zmianą gradientu nasycenia koloru. Listing 16.12 przedstawia przykładowy kod takiej animacji.

**Listing 16.12.** Połączenie animacji translacyjnej z animacją typu alfa w zestawie animacji

---

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator">
    <translate android:fromYDelta="-100%" android:toYDelta="0"
        android:duration="500" />
    <alpha android:fromAlpha="0.0" android:toAlpha="1.0"
        android:duration="500" />
</set>
```

---

Zwróćmy uwagę, w jaki sposób określiliśmy dwie animacje w zestawie animacji. Animacja translacyjna będzie przesuwała tekst z góry na dół w wydzielonym dla niego obszarze wyświetlenia. Animacja typu alfa będzie powodować zmianę gradientu nasycenia koloru od przezroczystego do całkowicie nasyconego podczas przesuwania tekstu w dół. Wartość 500 czasu trwania animacji pozwoli użytkownikowi obserwować w wygodny sposób zmianę. Oczywiście znowu będzie trzeba zmienić plik pośredniczący layoutAnimation, tak żeby znalazło się w nim odniesienie do nowego pliku. Zakładając, że nazwą pliku zawierającego połączone animacje jest */res/anim/translate-alpha.xml*, plik *layoutAnimation* będzie wyglądał następująco:

```
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
    android:delay="30%"
    android:animationOrder="reverse"
    android:animation="@anim/translate-alpha" />
```

Zobaczmy, w jaki sposób można używać animacji rotacyjnej (listing 16.13).

**Listing 16.13.** Plik XML animacji rotacyjnej

---

```
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator"
    android:fromDegrees="0.0"
    android:toDegrees="360"
    android:pivotX="50%"
    android:pivotY="50%"
    android:duration="500" />
```

---

Kod z listingu 16.13 spowoduje wykonanie jednego pełnego obrotu przez każdy element tekstowy wokół środka tego elementu. Czas trwania 500 milisekund całkowicie wystarczy, żeby obserwator dostrzegł animację. Podobnie jak w poprzednich przypadkach, tak i teraz muszą zostać zmodyfikowane pliki XML kontrolera animacji oraz układu graficznego ListView, a aplikacja musi zostać ponownie uruchomiona, żeby animacja zadziałała.

Omówiliśmy już podstawowe pojęcia dotyczące animacji układu graficznego, począwszy od prostego pliku animacji, a skończywszy na powiązaniu go poprzez plik pośredniczący layout → Animation z widokiem ListView. Ta wiedza wystarczy, żeby ujrzeć animowane efekty. Musimy omówić jednak jeszcze jedno pojęcie dotyczące animacji układu graficznego — interpolatory.

## **Stosowanie interpolatorów**

Interpolatory określają, w jaki sposób dana właściwość, na przykład gradient koloru, zmienia się względem czasu. Czy będzie się ona zmieniała w sposób liniowy, czy w sposób wykładniczy? Czy rozpocznie się szybko, lecz będzie zwalniała z biegiem czasu? Zastanówmy się nad przykładem animacji typu alfa z listingu 16.11:

```
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@android:anim/accelerate_interpolator"
    android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="1000" />
```

Animacja rozpoznaje zastosowany interpolator — w tym przypadku `accelerate_interpolator`. Istnieje odpowiedni obiekt Java, służący do definiowania tego interpolatora. Poza tym zwróćmy uwagę, że określiliśmy ten interpolator jako odniesienie do zasobów. Oznacza to, że musi istnieć plik odpowiadający identyfikatorowi `anim/accelerate_interpolator`, w którym opisany jest ten obiekt języka Java oraz jego dodatkowe parametry. Tak jest w istocie. Przyjrzymy się definicji pliku XML, do którego odniesieniem jest identyfikator `@android:anim/accelerate_interpolator`:

```
<accelerateInterpolator
    xmlns:android="http://schemas.android.com/apk/res/android"
    factor="1" />
```

Plik ten można odnaleźć w następującym podkatalogu pakietu Android:

```
/res/anim/accelerate_interpolator.xml
```

Znacznik XML `accelerateInterpolator` odpowiada następującemu obiekowi środowiska Java:

```
android.view.animation.AccelerateInterpolator
```

W dokumentacji języka Java dotyczącej tej klasy można zobaczyć, jakie znaczniki XML są dla niej dostępne. Zadaniem tego interpolatora jest zapewnienie współczynnika powielania danego przedziału czasowego w oparciu o krzywą hiperboliczną. Widać to w kodzie źródłowym interpolatora:

```
public float getInterpolation(float input)
{
    if (mFactor == 1.0f)
    {
        return (float)(input * input);
    }
    else
    {
        return (float)Math.pow(input, 2 * mFactor);
    }
}
```

Każdy interpolator w inny sposób implementuje metodę `getInterpolation`. W naszym przypadku, jeśli interpolator zostanie skonfigurowany tak, że współczynnik będzie wynosił 1.0, zostanie zwrócony kwadrat tego współczynnika. W przeciwnym razie zostanie zwrócona potęga danych wejściowych, które będą nadal skalowane przez ten współczynnik. Jeżeli zaś wartość współczynnika będzie wynosiła 1.5, zamiast funkcji kwadratowej ujrzymy funkcję sześcienną.

Poniżej wypisaliśmy listę obsługiwanych interpolatorów:

```
AccelerateDecelerateInterpolator
AccelerateInterpolator
CycleInterpolator
DecelerateInterpolator
LinearInterpolator
AnticipateInterpolator
AnticipateOvershootInterpolator
BounceInterpolator
OvershootInterpolator
```

Żeby zaprezentować potencjalną elastyczność interpolatorów, przyjrzyjmy się pokróćce obiekowi `BounceInterpolator`, powodującemu „podskakiwanie” elementu (to znaczy jego naprzemienne ruch w górę i w dół) do samego końca poniższej animacji:

```
public class BounceInterpolator implements Interpolator {
    private static float bounce(float t) {
        return t * t * 8.0f;
    }

    public float getInterpolation(float t) {
        t *= 1.1226f;
        if (t < 0.3535f) return bounce(t);
        else if (t < 0.7408f) return bounce(t - 0.54719f) + 0.7f;
        else if (t < 0.9644f) return bounce(t - 0.8526f) + 0.9f;
        else return bounce(t - 1.0435f) + 0.95f;
    }
}
```

Zachowanie tych interpolatorów zostało omówione pod poniższym adresem:

<http://developer.android.com/reference/android/view/animation/package-summary.html>

W dokumentacji języka Java wymienione są również znaczniki XML, pozwalające na kontrolowanie każdej z tych klas. Jednak z dokumentacji trudno wywnioskować przeznaczenie każdego typu interpolatora. Najlepiej jest samemu wypróbować wszystkie interpolatory i sprawdzić skutki ich działania. Pod poniższym adresem można również przejrzeć kod źródłowy:

<http://android.git.kernel.org/?p=platform%2Fframeworks%2Fbase.git&a=search&h=HEAD&st=grep&s=BounceInterpolator>

Na tym zakończymy wywody poświęcone animacji układu graficznego. Przejdziemy teraz do trzeciej części animowania, poświęconej programowaniu animacji widoku.

## Animacja widoku

Skoro zapoznaliśmy się już z animacją poklatkową oraz animacją układu graficznego, możemy zająć się animacją widoku — najbardziej skomplikowanym rodzajem animacji. Stosowana jest w niej technika animowania dowolnego widoku poprzez kontrolowanie macierzy transformacji, służącej do wyświetlania widoku.

### Animacja widoku

Widok wyświetlany przez Androida przechodzi przez macierz transformacji. W aplikacjach graficznych macierze transformacji służą do przekształcenia w jakiś sposób widoku. Proces ten polega na przetłumaczeniu wejściowego zestawu współrzędnych pikseli i kombinacji kolorów na nowy zestaw. Po przeprowadzeniu transformacji ujrzymy obraz zmieniony pod względem rozmiaru, pozycji, orientacji lub koloru.

Te przekształcenia można przeprowadzić za pomocą aparatu matematycznego, mnożąc w określony sposób wejściowy zestaw współrzędnych przez wartości macierzy transformacji, dzięki czemu powstanie nowy zestaw współrzędnych. Poprzez zmianę macierzy transformacji wpływamy na wygląd widoku.

Macierz, która *nie* zmienia widoku podczas tego mnożenia, nazywana jest macierzą jednostkową. Transformację przeważnie rozpoczynamy od macierzy jednostkowej i kolejno wprowadzamy serie transformacji rozmiaru, pozycji i orientacji. Następnie za pomocą macierzy końcowej rysujemy widok.

Android odsłania taką macierz transformacji widoku poprzez umożliwienie zarejestrowania obiektu animacji wobec tego widoku. Obiekt animacji będzie posiadał procedurę wywołania, dzięki której uzyska dostęp do tej macierzy i w określony sposób zmieni jej wartości, co pociągnie za sobą zmianę wyświetlanego widoku. Zajmiemy się teraz tym procesem.

Rozpoczniemy tworzenie przykładowego projektu od zaplanowania animacji widoku. Na początek zapełnimy aktywność kilkoma elementami w widoku ListView, podobnie jak miało to miejsce w podrozdziale „Animacja układu graficznego”. Następnie w górnej części ekranu umieścimy przycisk powodujący uruchomienie animacji ListView (rysunek 16.5). Widoczne są zarówno lista elementów, jak i przycisk, żadna animacja nie została jednak jeszcze uruchomiona. Do tego będzie służył utworzony przycisk.



Rysunek 16.5. Aktywność animacji widoku

Po kliknięciu przycisku *Uruchom animację* powinien się pojawić mały widok pośrodku ekranu, który następnie stopniowo będzie się powiększał aż do wypełnienia zarezerwowanej dla niego przestrzeni. Zaprezentujemy kod, który nam to umożliwi. Na listingu 16.14 został pokazany kod pliku XML układu graficznego, nadający się do zastosowania w aktywności.

---

#### **Listing 16.14.** Plik XML układu graficznego dla aktywności animacji widoku

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Ten plik jest umieszczony w /res/layout/list_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
```

```

<Button
    android:id="@+id/btn_animate"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Uruchom animację"
/>
<ListView
    android:id="@+id/list_view_id"
    android:persistentDrawingCache="animation|scrolling"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
/>
</LinearLayout>

```

---

Pogrubiona czcionka ma zwrócić uwagę Czytelnika na lokalizację oraz nazwę pliku. Ten układ graficzny składa się z dwóch części: pierwsza z nich to przycisk *btn\_animate*, służący do uruchomienia animacji widoku; drugą jest widok *ListView*, w naszym przypadku nazwany *list\_view\_id*.

Skoro mamy już układ graficzny dla aktywności, możemy utworzyć samą aktywność, żeby wyświetlić widok i skonfigurować przycisk *Uruchom animację* (listing 16.15).

#### **Listing 16.15.** Kod dla aktywności animacji widoku przed rozpoczęciem animacji

---

```

public class ViewAnimationActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.list_layout);
        setupListView();
        this.setupButton();
    }
    private void setupListView()
    {
        String[] listItems = new String[] {
            "Element 1", "Element 2", "Element 3",
            "Element 4", "Element 5", "Element 6",
        };
        ArrayAdapter listItemAdapter =
            new ArrayAdapter(this
                , android.R.layout.simple_list_item_1
                , listItems);
        ListView lv = (ListView)this.findViewById(R.id.list_view_id);
        lv.setAdapter(listItemAdapter);
    }
    private void setupButton()
    {
        Button b = (Button)this.findViewById(R.id.btn_animate);
        b.setOnClickListener(
            new Button.OnClickListener(){
                public void onClick(View v)
                {

```

```
        //animateListView();  
    }  
});  
}  
}
```

---

Kod przedstawiony dla aktywności animacji widoku z listingu 16.15 bardzo przypomina kod aktywności animacji układu graficznego z listingu 16.7. W podobny sposób wczytaliśmy widok i wstawiliśmy sześć elementów tekstowych do widoku `ListView`. Skonfigurowaliśmy przycisk w taki sposób, żeby wywoływał metodę `animateListView()` po kliknięciu. Na razie jednak oznaczymy ten fragment komunikatem, dopóki nasz przykład nie zadziała.

Aktywność możemy wywołać tuż po jej zarejestrowaniu w pliku `AndroidManifest.xml`:

```
<activity android:name=".ViewAnimationActivity"  
        android:label="Aktywność testowa animacji widoku">
```

Po przeprowadzeniu procesu rejestracji możemy wywołać aktywność animacji widoku za pomocą dowolnego przycisku menu w aplikacji, korzystając z poniższego fragmentu kodu:

```
Intent intent = new Intent(this, ViewAnimationActivity.class);  
startActivity(intent);
```

Po uruchomieniu programu pojawi się ekran pokazany na rysunku 16.5.

## Dodawanie animacji

W tym ćwiczeniu naszym celem jest dodanie animacji do widoku `ListView`, widocznego na rysunku 16.5. W tym celu potrzebujemy klasy wywodzącej się z pakietu `android.view`.  
→`animation.Animation`. Następnie musimy przesłonić metodę `applyTransformation`, aby można było zmodyfikować macierz transformacji. Nazwijmy tę klasę `ViewAnimation`. Po jej utworzeniu możemy przeprowadzić w klasie `ListView` następującą czynność:

```
ListView lv = (ListView)this.findViewById(R.id.list_view_id);  
lv.startAnimation(new ViewAnimation());
```

Pójdzmy dalej. Przyjrzyjmy się kodowi źródłowemu klasy `ViewAnimation` i zastanówmy się, jaki rodzaj animacji chcemy otrzymać (listing 16.16).

**Listing 16.16.** Kod źródłowy klasy `ViewAnimation`

---

```
public class ViewAnimation extends Animation  
{  
  
    @Override  
    public void initialize(int width, int height, int parentWidth,  
                          int parentHeight)  
    {  
        super.initialize(width, height, parentWidth, parentHeight);  
        setDuration(2500);  
        setFillAfter(true);  
        setInterpolator(new LinearInterpolator());  
    }  
    @Override  
    protected void applyTransformation(float interpolatedTime, Transformation t)
```

```
{  
    final Matrix matrix = t.getMatrix();  
    matrix.setScale(interpolatedTime, interpolatedTime);  
}  
}
```

Metoda zwrotna `initialize` informuje nas o wymiarach widoku. W niej są również inicjalizowane wszelkie parametry animacji. W naszym przykładzie skonfigurowaliśmy czas trwania na 2500 milisekund (2,5 sekundy). Sprawimy także, że wynik końcowy animacji pozostanie niezmieniony po jej zakończeniu, a to za sprawą przypisania parametrowi `FillAfter` wartości `true`. W dodatku określiliśmy, że nasz interpolator jest liniowy, co oznacza, że animacja zmienia się stopniowo od początku do końca. Wszystkie wymienione właściwości pochodzą z bazowej klasy `android.view.animation.Animation`.

Część zasadnicza animacji jest przeprowadzana w metodzie `applyTransformation`. Szkielet Androida będzie ją bez przerwy wywoływał w celu symulowania animacji. Za każdym wywołaniem tej metody zmienia się wartość parametru `interpolatedTime`. Zmienia się ona w zakresie od 0 do 1 w zależności od tego, w jakim momencie się znajdujemy podczas 2,5-sekundowego cyklu animacji, ustawionego na etapie jej inicjalizacji. Kiedy wartość parametru `interpolatedTime` wynosi 1, znajdujemy się na końcu animacji.

Naszym kolejnym zadaniem jest zmiana macierzy transformacji, dostępnej poprzez obiekt transformacji `t`, umieszczony w metodzie `applyTransformation`. Najpierw należy uzyskać dostęp do macierzy i zmienić jej wartości. Po narysowaniu nowego widoku zadziała również zmodyfikowana macierz. W dokumentacji interfejsów API dotyczącej klasy `android.graphics.Matrix` można znaleźć opis wielu metod dostępnych w obiekcie `Matrix`:

<http://developer.android.com/reference/android/graphics/Matrix.html>

W kodzie z listingu 16.16 zmiana macierzy transformacji zajmuje się poniższy wiersz:

```
matrix.setScale(interpolatedTime, interpolatedTime);
```

Metoda `setScale` zawiera dwa parametry — są to współczynniki skali w osiach  $x$  oraz  $y$ . Ponieważ wartości parametru `interpolatedTime` mieszczą się w zakresie od 0 do 1, można go zastosować bezpośrednio w postaci współczynnika skali.

Zatem na początku animacji współczynnik ten wynosi 0 w obydwu kierunkach. W połowie przebiegu animacji osie  $x$  oraz  $y$  będą miały wartość 0,5. Po zakończeniu animacji widok będzie miał pełny rozmiar, ponieważ obydwa współczynniki skali będą miały wartości równe 1. W wyniku tego widok ListView jest na początku animacji niewielki i powiększa się do standardowego rozmiaru.

Na listingu 16.17 został zaprezentowany kompletny kod źródłowy aktywności `ViewAnimationActivity` zawierającej animacje.

**Listing 16.17.** Kod źródłowy aktywności animacji widoku wraz z animacją

```
public class ViewAnimationActivity extends Activity {  
  
    @Override  
    public void onCreate(Bundle savedInstanceState)  
    {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.list_layout);  
        setupListView();  
    }  
}
```

```
        this.setupButton();
    }
    private void setupListView()
    {
        String[] listItems = new String[] {
            "Element 1", "Element 2", "Element 3",
            "Element 4", "Element 5", "Element 6",
        };

        ArrayAdapter listItemAdapter =
            new ArrayAdapter(this
                , android.R.layout.simple_list_item_1
                , listItems);
        ListView lv = (ListView) this.findViewById(R.id.list_view_id);
        lv.setAdapter(listItemAdapter);
    }
    private void setupButton()
    {
        Button b = (Button) this.findViewById(R.id.btn_animate);
        b.setOnClickListener(
            new Button.OnClickListener(){
                public void onClick(View v)
                {
                    animateListView();
                }
            });
    }
    private void animateListView()
    {
        ListView lv = (ListView) this.findViewById(R.id.list_view_id);
        lv.startAnimation(new ViewAnimation());
    }
}
```

---

Po uruchomieniu kodu z listingu 16.17 Czytelnik zobaczy coś dziwnego. Widok `ListView`, zamiast równomiernie powiększać się od środka ekranu, rozrasta się od lewego górnego rogu. Wynika to z faktu, że operacje macierzy transformacji mają swój początek właśnie w lewym górnym rogu ekranu. Chcąc uzyskać zamierzony efekt, musimy najpierw przesunąć cały widok w taki sposób, żeby jego środek pokrywał się ze środkiem animacji (w lewym górnym rogu). Następnie wprowadzamy macierz i z powrotem przenosimy widok na właściwe miejsce.

Na listingu 16.18 wstawiliśmy przerobiony kod z listingu 16.16 i zaznaczyliśmy najistotniejsze elementy.

---

**Listing 16.18.** Animacja widoku wykorzystująca metody `preTranslate` i `postTranslate`

---

```
public class ViewAnimation extends Animation {
    float centerX, centerY;
    public ViewAnimation3(){}
    @Override
    public void initialize(int width, int height, int parentWidth, int parentHeight) {
        super.initialize(width, height, parentWidth, parentHeight);
        centerX = width/2.0f;
        centerY = height/2.0f;
```

```

        setDuration(2500);
        setFillAfter(true);
        setInterpolator(new LinearInterpolator());
    }
    @Override
    protected void applyTransformation(float interpolatedTime, Transformation t) {
        final Matrix matrix = t.getMatrix();
        matrix.setScale(interpolatedTime, interpolatedTime);
        matrix.preTranslate(-centerX, -centerY);
        matrix.postTranslate(centerX, centerY);
    }
}

```

---

Metody `preTranslate` oraz `postTranslate` konfigurują macierz przed operacją skalowania oraz po tej operacji. Jest to proces równoważny utworzeniu zespołu trzech macierzy transformacji. Następujący kod:

```

matrix.setScale(interpolatedTime, interpolatedTime);
matrix.preTranslate(-centerX, -centerY);
matrix.postTranslate(centerX, centerY);

```

jest równoważny instrukcjom:

```

przejdź do innego środka
skaluj
przejdź do oryginalnego środka

```

Taki wzorzec metod `pre` i `post` jest stosowany bardzo często. Podobne wyniki można osiągnąć za pomocą innych metod klasy `Matrix`, ta technika jest jednak najpopularniejsza — a do tego jest zwięzła. Pozostałe techniki również zostaną omówione pod koniec rozdziału.

Co ważniejsze, klasa `Matrix` umożliwia nie tylko skalowanie widoku, lecz również przenoszenie go za pomocą metod `translate` oraz zmianę jego orientacji za pomocą metod `rotate`. Można sprawdzić te metody i przekonać się, jak wyglądają ich efekty. W rzeczywistości wszystkie animacje omówione w podrozdziale „Animacja układu graficznego” są implementowane wewnętrznie za pomocą metod klasy `Matrix`.

## Zastosowanie klasy Camera do symulowania głębi w obrazie dwuwymiarowym

Pakiet graficzny w Androidzie zawiera jeszcze jedną klasę związaną z animacją — a dokładniej z transformacją — klasę `Camera`. Można ją wykorzystać do symulowania głębi poprzez rzutowanie obrazu dwuwymiarowego, poruszającego się w przestrzeni trójwymiarowej po płaszczyźnie. Możemy na przykład wysłać nasz widok `ListView` o 10 pikseli w głąb ekranu po osi `z` i obrócić ją o 30 stopni wokół osi `y`. Na listingu 16.19 podajemy przykład modyfikowania macierzy za pomocą klasy `Camera`:

**Listing 16.19.** Zastosowanie klasy Camera

```

...
public class ViewAnimation extends Animation {
    float centerX, centerY;
    Camera camera = new Camera();
    public ViewAnimation1(float cx, float cy){

```

```
        centerX = cx;
        centerY = cy;
    }
    @Override
    public void initialize(int width, int height, int parentWidth, int parentHeight) {
        super.initialize(width, height, parentWidth, parentHeight);
        setDuration(2500);
        setFillAfter(true);
        setInterpolator(new LinearInterpolator());
    }
    @Override
    protected void applyTransformation(float interpolatedTime, Transformation t) {
        applyTransformationNew(interpolatedTime, t);
    }
    protected void applyTransformationNew(float interpolatedTime, Transformation t)
    {
        final Matrix matrix = t.getMatrix();
        camera.save();
        camera.translate(0.0f, 0.0f, (1300 - 1300.0f * interpolatedTime));
        camera.rotateY(360 * interpolatedTime);
        camera.getMatrix(matrix);

        matrix.preTranslate(-centerX, -centerY);
        matrix.postTranslate(centerX, centerY);
        camera.restore();
    }
}
```

---

Animacja widoku ListView przebiega tu w następujący sposób: najpierw jest on umieszczony w odległości 1300 pikseli od ekranu po osi z, a następnie wraca do płaszczyzny, w której oszczędza wartość 0. W międzyczasie zostaje on również obrócony od 0 do 360 stopni wokół osi y. Zobaczmy, w jaki sposób w kodzie jest zdefiniowane to zachowanie, opisane w poniższej metodzie:

```
camera.translate(0.0f, 0.0f, (1300 - 1300.0f * interpolatedTime));
```

Metoda ta powoduje, że obiekt camera przesuwa się w taki sposób, iż przy wartości 0 parametru `interpolatedTime` (początek animacji) wartość z będzie wynosiła 1300. Podczas trwania animacji wartość z będzie systematycznie maleć aż do samego końca, gdy wartość parametru `interpolatedTime` wyniesie 1, a tym samym wartość parametru z będzie równa 0.

Metoda `camera.rotateY(360 * interpolatedTime)` wykorzystuje możliwość obracania bryły w trójwymiarze wokół wybranej osi przez obiekt camera. Na początku animacji jej wartość wynosi 0. Na końcu animacji przybierze wartość 360.

Metoda `camera.getMatrix(matrix)` pobiera operacje dotychczas wykonane na obiekcie Camera i narzuca je przekazanej macierzy transformacji. W tym momencie klasa `matrix` posiada wszystkie translacje potrzebne do uzyskania końcowego efektu, zapewnione przez klasę Camera. Teraz klasa Camera schodzi z widoku (niezamierzona gra słów), ponieważ w macierzy zostały zaimplementowane wszystkie niezbędne operacje. Wykonujemy teraz operacje `pre` i `post` w celu przesunięcia środka widoku i sprowadzenia go z powrotem. Na koniec przywracamy obiekt Camera do pierwotnego, uprzednio zachowanego stanu.

Po wstawieniu kodu do naszego przykładu zobaczymy kontrolkę ListView zbliżającą się ze środka widoku w stronę użytkownika, przy okazji wirującą, dokładnie tak jak zaplanowaliśmy.

Część naszej analizy wiążącej się z animacją widoku dotyczyła sposobu animowania dowolnego widoku poprzez rozszerzenie klasy `Animation` i zastosowanie jej wobec tego widoku. Poza modyfikowaniem matryc (bezpośrednio i za pomocą klasy `Camera`) klasa `Animation` umożliwia też wykrywanie poszczególnych etapów animacji. Właśnie tym się teraz zajmiemy.

## Analiza interfejsu AnimationListener

Android wykorzystuje interfejs nasłuchujący `AnimationListener` do monitorowania zdarzeń animacji (listing 16.20). Możemy nasłuchiwać tych zdarzeń poprzez zaimplementowanie interfejsu `AnimationListener` i skonfigurowanie tej implementacji wobec klasy `Animation`.

**Listing 16.20.** Implementacja interfejsu `AnimationListener`

---

```
public class ViewAnimationListener
implements Animation.AnimationListener {

    private ViewAnimationListener(){}

    public void onAnimationStart(Animation animation)
    {
        Log.d("Przykładowa animacja", "onAnimationStart");
    }
    public void onAnimationEnd(Animation animation)
    {
        Log.d("Przykładowa animacja", "onAnimationEnd");
    }
    public void onAnimationRepeat(Animation animation)
    {
        Log.d("Przykładowa animacja", "onAnimationRepeat");
    }
}
```

---

Klasa `ViewAnimationListener` służy jedynie do tworzenia dzienników komunikatów. Możemy zaktualizować metodę `animateListView` w naszym przykładzie animacji widoku (listing 16.17), żeby dołączyć obiekt nasłuchujący animacje:

```
private void animateListView()
{
    ListView lv = (ListView)this.findViewById(R.id.list_view_id);
    ViewAnimation animation = new ViewAnimation();
    animation.setAnimationListener(new ViewAnimationListener());
    lv.startAnimation(animation);
}
```

## Kilka uwag na temat macierzy transformacji

Jak pokazaliśmy w tym rozdziale, macierze stanowią podstawę przekształcania widoków i przetwarzania animacji. Teraz omówimy w skrócie niektóre kluczowe metody klasy `Matrix`. Poniżej zostały wymienione podstawowe operacje na macierzach:

```
matrix.reset();
matrix.setScale();
matrix.setTranslate()
matrix.setRotate();
matrix.setSkew();
```

Pierwsza operacja przekształca macierz do postaci macierzy jednostkowej, która po zastosowaniu nie wprowadza zmian w widoku. Operacja `setScale` jest odpowiedzialna za zmianę rozmiaru, `setTranslate` powoduje przesunięcie pozycji obiektu imitujące ruch, a `setRotate` służy do zmiany orientacji. Operacja `setSkew` pozwala na wykrzywienie widoku.

Można powiązać ze sobą macierze lub je wspólnie powielać, aby utworzyć efekt złożony z wielu transformacji. Rozpatrzmy następujący przykład, w którym `m1`, `m2` oraz `m3` są macierzami jednostkowymi:

```
m1.setScale();  
m2.setTranlate()  
m3.concat(m1,m2)
```

Transformacja widoku przez macierz `m1` i następująca po niej transformacja widoku przez macierz `m2` są tożsame z transformacją tego samego widoku przez macierz `m3`. Zwróćmy uwagę, że metody typu `set` zastępują poprzednie transformacje, natomiast `m3.concat(m1,m2)` nie jest tym samym co `m3.concat(m2,m1)`.

Pokazaliśmy już sposób postępowania podczas stosowania metod `preTranslate` oraz `postTranslate` wobec zmiany macierzy transformacji. W rzeczywistości metody `pre` i `post` nie są przeznaczone wyłącznie dla operacji `translate`, lecz tego typu odmiany są dostępne dla każdego rodzaju metod transformacji typu `set`. Ostatecznie metoda `preTranslate`, taka jak `m1.preTranslate(m2)`, jest równoważna operacji:

```
m1.concat(m2,m1)
```

W analogiczny sposób metoda `m1.postTranslate(m2)` jest tożsama operacji:

```
m1.concat(m1,m2)
```

Po rozszerzeniu ekwiwalentem poniższego kodu:

```
matrix.setScale(interpolatedTime, interpolatedTime);  
matrix.preTranslate(-centerX, -centerY);  
matrix.postTranslate(centerX, centerY);
```

jest:

```
Matrix matrixPreTranslate = new Matrix();  
matrixPreTranslate.setTranslate(-centerX, -centerY);  
  
Matrix matrixPostTranslate = new Matrix();  
matrixPostTranslate.setTranslate(cetnerX, centerY);  
  
matrix.concat(matrixPreTranslate,matrix);  
matrix.postTranslate(matrix,matrixPostTranslate);
```

## Odbośniki

Poniżej prezentujemy przydatne odnośniki do materiałów, dzięki którym jeszcze lepiej zrozumiemy koncepcje zawarte w tym rozdziale:

- <http://developer.android.com/reference/android/view/animation/package-summary.html>
  - znajdziemy tu różnorodne interfejsy związane z animacją, w tym również interpolatory.

- <http://developer.android.com/guide/topics/resources/animation-resource.html>  
— omówienie znaczników XML stosowanych w różnych odmianach animacji.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu projekt do pobrania, przygotowany specjalnie do tego rozdziału. Jest to plik umieszczony w katalogu *ProAndroid3\_R16\_Animacje*.

## Podsumowanie

W tym rozdziale zaprezentowaliśmy ciekawy sposób uatrakcyjnienia interfejsu użytkownika poprzez zastosowanie animacji. Omówiliśmy wszystkie podstawowe typy animacji obsługiwane w Androidzie: animację poklatkową, animację układu graficznego oraz animację widoku. Opisaliśmy także dodatkowe pojęcia dotyczące animacji, między innymi interpolatory i macierze transformacji.

Skoro Czytelnik poznał już podstawy, proponujemy przejrzeć przykładowe interfejsy API, udostępnione w zestawie Android SDK, aby przeanalizować pliki XML definiujące różne typy animacji. Poruszamy jeszcze temat animacji w rozdziale 20., poświęconym rysowaniu i animowaniu za pomocą technologii OpenGL. Natomiast w rozdziale 29. możemy się zapoznać z ogólnym omówieniem animacji opartej na właściwościach, stosowanej wraz z fragmentami.



# Analiza usług wykorzystujących mapy i dane o lokalizacji

W niniejszym rozdziale zajmiemy się usługami systemu Android wykorzystującymi mapy oraz dane o lokalizacji urządzenia. Są to jedne z najciekawszych elementów zestawu Android SDK. Ta część środowiska SDK zawiera interfejsy API umożliwiające projektantom aplikacji wyświetlanie map oraz manipulowanie nimi, a także uzyskiwanie informacji o lokalizacji urządzenia w czasie rzeczywistym. Interfejsy te posiadają wiele innych fascynujących funkcji.

Usługi wykorzystujące dane o lokalizacji urządzenia składają się z dwóch zasadniczych części: interfejsów API map oraz interfejsów API lokalizacji. Każdy z tych interfejsów jest umieszczony w oddzielnym pakiecie. I tak pakiet map to *com.google.android.maps*, natomiast pakiet lokalizacji nosi nazwę *android.location*. Interfejsy map zawierają narzędzia pozwalające na wyświetlanie map oraz manipulowanie nimi. Można na przykład przybliżać oraz przesuwać widok, zmieniać tryb wyświetlania mapy (z widoku satelitarnego na widok ulic), nakładać na mapę własne informacje i tak dalej. Z drugiej strony dostępne są dane systemu GPS (ang. *Global System Positioning* — globalny system ustalania położenia geograficznego) oraz informacje o położeniu geograficznym uzyskiwane w czasie rzeczywistym, które są dostarczane poprzez pakiet lokalizacji.

Interfejsy te często łączą się poprzez internet z usługami dostępnymi na serwerach firmy Google. Zatem jeżeli te usługi mają działać, zazwyczaj trzeba zapewnić dostęp do internetu. Ponadto należy zaakceptować warunki korzystania z usług, zanim będzie można rozpocząć projektowanie aplikacji używających interfejsów API tych usług. Należy uważnie przeczytać te warunki; firma Google ogranicza zastosowanie danych usług. Na przykład można udostępnić informacje o położeniu dla użytku osobistego użytkowników, jednak pewne zastosowania komercyjne, jak również aplikacje wspomagające zautomatyzowaną kontrolę pojazdów są zabronione. Warunki korzystania z usług zostaną wyświetcone podczas procesu rejestracji w celu uzyskania klucza interfejsu API.

W niniejszym rozdziale omówimy obydwa rodzaje pakietów. Zaczniemy od interfejsów map i pokażemy, w jaki sposób można wykorzystać mapy w aplikacji. Jak się przekonamy, praca z mapami w Androidzie ogranicza się do stosowania kontrolki interfejsu UI MapView oraz klasy MapActivity, wraz z interfejsami map API zintegrowanymi z serwerem Google Maps. Zademonstrujemy także, w jaki sposób można umieszczać własne informacje na wyświetlonej mapie oraz jak wyświetlać bieżące położenie urządzenia na mapie. W następnej części zajmiemy się usługami lokalizacji, które rozwijają koncepcje wykorzystania map w usługach. Zaprezentujemy możliwości klasy Geocoder oraz usługi LocationManager. Poruszmy także tematykę wątkowania — bardzo istotną w przypadku tych interfejsów API.

## Pakiet do pracy z mapami

Jak już wspomnieliśmy, interfejsy API map są jednym ze składników lokalizacyjnych usług Androida. Pakiet ten zawiera wszelkie elementy niezbędne do wyświetlenia mapy na ekranie, umożliwienia użytkownikowi dostosowywania jej (na przykład zmiany skali), wyświetlania własnych informacji w górnej części mapy i tak dalej. Pierwszym etapem pracy z tym pakietem jest wyświetlenie mapy. W tym celu wykorzystamy klasę widoku MapView. Zanim jednak będziemy mogli z nią pracować, musimy przeprowadzić odpowiednie przygotowania. W szczególności musimy uzyskać od firmy Google klucz interfejsu API mapy. **Klucz interfejsu API mapy** pozwala Androidowi na nawiązanie łączności z usługą Google Maps w celu uzyskania danych mapy. W kolejnym punkcie przedstawiamy sposób uzyskania takiego klucza.

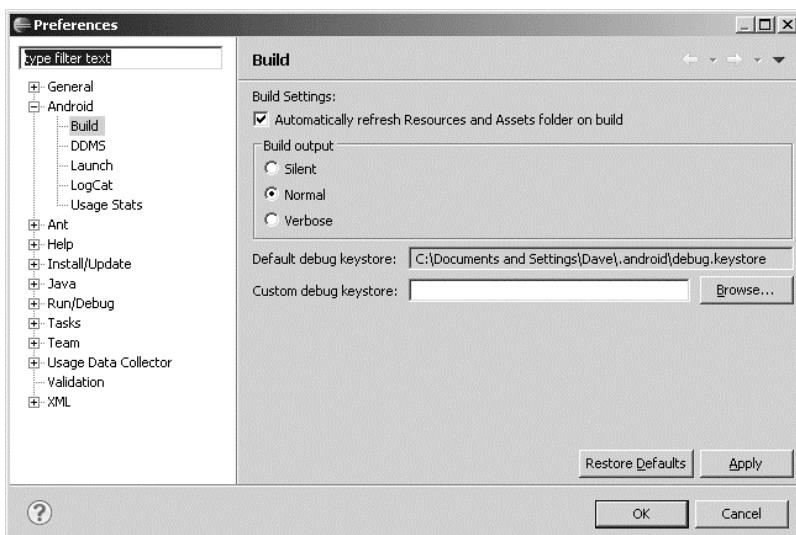
## Uzyskanie klucza interfejsu API mapy od firmy Google

Pierwszą ważną informacją dotyczącą klucza interfejsu API mapy jest to, że w rzeczywistości potrzebne będą dwa klucze: jeden do etapu projektowego na emulatorze, a drugi dla produktu końcowego (pracującego na urządzeniu fizycznym). Wynika to z faktu, iż certyfikat potrzebny do uzyskania klucza będzie inny dla wersji testowej i wersji końcowej (wyjaśniliśmy to w rozdziale 10.).

Na etapie projektowania wtyczka ADT generuje plik *.adt* i wdraża go na emulator. Ponieważ plik ten musi być podpisany za pomocą certyfikatu, na tym etapie zostaje wykorzystany certyfikat testowy. W momencie przygotowywania aplikacji do użytku naprawdopodobniej będziemy chcieli skorzystać z własnoręcznie utworzonego certyfikatu do podpisania pliku *.apk*. Dobra wiadomość jest taka, że otrzymuje się jeden klucz interfejsu API mapy przeznaczony do wersji testowej oraz drugi klucz dla wersji końcowej, a ich zamiana przed eksportowaniem wersji ostatecznej nie stanowi problemu.

Aby uzyskać taki klucz, potrzebny jest certyfikat służący do podpisania aplikacji (w przypadku emulatora certyfikat testowy). Uzyskujemy skrót MD5 certyfikatu, a następnie wprowadzamy go w witrynie Google, co spowoduje wygenerowanie odpowiadającego mu klucza.

Najpierw musimy znaleźć certyfikat testowy, wygenerowany i przechowywany przez środowisko Eclipse. Jego dokładną lokalizację możemy odszukać w programie Eclipse. Z menu *Preferences* przechodzimy do opcji *Android/Build*. Położenie certyfikatu testowego jest wyświetcone w polu *Default debug keystore*, zaprezentowanym na rysunku 17.1 (w rozdziale 2. zostały umieszczone informacje dotyczące lokalizacji menu *Preferences*).



**Rysunek 17.1.** Lokalizacja certyfikatu testowego

Żeby uzyskać skrót MD5, możemy uruchomić narzędzie keytool wraz z opcją -list, zgodnie z poniższym poleceniem:

```
keytool -list -alias androiddebugkey -keystore "PEŁNA ŚCIEŻKA PLIKU
→debug.keystore" – storepass android -keypass android
```

Warto zauważyć, że parametr alias testowego magazynu kluczy posiada wartość android →debugkey. Hasło do magazynu kluczy brzmi *android*, podobnie jak hasło klucza prywatnego. Po uruchomieniu powyższego polecenia aplikacja keytool wyświetli skrót (rysunek 17.2).

```
C:\> C:\WINDOWS\system32\cmd.exe
C:\>Program Files\Java\jdk1.6_0_16\bin>keytool -list -alias androiddebugkey -keystore "C:\Documents and Settings\Dave\.android\debug.keystore"
Enter keystore password:
androiddebugkey, Sep 6, 2009, PrivateKeyEntry,
Certificate fingerprint <MD5>: [REDACTED]
C:\>Program Files\Java\jdk1.6_0_16\bin>
```

**Rysunek 17.2.** Dane wyjściowe opcji list w narzędziu keytools (skrót MD5 został celowo zamazany)

Teraz wklejamy uzyskany skrót MD5 certyfikatu testowego w odpowiednie pole na stronie Google:  
<http://code.google.com/android/maps-api-signup.html>

Następnie trzeba przeczytać warunki korzystania z usług. Jeżeli są do zaakceptowania, należy wcisnąć przycisk *Generate API Key*, aby uzyskać klucz interfejsu API map dla usługi Google Maps, odpowiadający skrótwi MD5. Klucz ten jest od razu aktywny, zatem można już teraz za jego pomocą uzyskać dostęp do danych map z serwisu Google. Pamiętajmy, że do uzyskania klucza wymagane jest posiadanie konta Google — podczas próby wygenerowania klucza zostanie wyświetlony monit o zalogowanie się w serwisie Google.

Przypominamy informacje z rozdziału 10. Napisaliśmy tam, że jeśli certyfikat testowy utraci ważność, to samo spotka używany w tym czasie klucz interfejsu map. Jeżeli zmienimy certyfikat testowy, będziemy musieli powtórzyć powyższe kroki i za pomocą nowego certyfikatu uzyskać

kolejny klucz interfejsu map. Taki mechanizm stanowi dobrą motywację do definiowania w tworzonym certyfikacie testowym czasu ważności dłuższego niż domyślny rok. W rozdziale 10. znajdziemy instrukcje dotyczące tworzenia długotrwałego certyfikatu testowego.

Pobawmy się teraz mapami.

## Klasy MapView i MapActivity

Wiele elementów technologii wykorzystującej mapy opiera się na kontrolce interfejsu użytkownika `MapView` oraz na rozszerzeniu klasy `android.app.Activity` nazwanym `MapActivity`. Obydwie klasy wykonują złożone zadania podczas wyświetlania mapy w Androidzie oraz korzystania z niej. Podczas pracy z mapami należy pamiętać, że obydwie klasy muszą ze sobą współpracować. Dokładniej mówiąc, żeby móc korzystać z klasy `MapView`, należy utworzyć jej egzemplarz w klasie `MapActivity`. Aby z kolei ten proces się powiodł, potrzebny będzie klucz interfejsu API mapy.

W przypadku tworzenia widoku `MapView` za pomocą układu graficznego XML należy skonfigurować właściwość `android:apiKey`. Z kolei w celu utworzenia tej klasy w kodzie Java musimy przekazać klucz API mapy do konstruktora widoku `MapView`. Ponieważ dane mapy pochodzą z serwera Google Maps, aplikacja będzie wymagała dostępu do internetu. Oznacza to, że będzie potrzebne przynajmniej następujące żądanie uprawnienia w pliku `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Na listingu 17.1 wpisy pliku `AndroidManifest.xml` umożliwiające działanie aplikacji obsługującej mapy wyróżniono pogrubioną czcionką.

---

**Listing 17.1.** Znaczniki pliku `AndroidManifest.xml` wymagane dla aplikacji obsługującej mapy

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <uses-library android:name="com.google.android.maps" />
        <activity android:name=".MapViewDemoActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-sdk android:minSdkVersion="4" />
</manifest>
```

---

Powinniśmy wprowadzić jeszcze jedną modyfikację do pliku `AndroidManifest.xml`. Definicja aplikacji obsługującej mapy wymaga odniesienia do biblioteki map (odpowiedni wiersz na listingu 17.1 został również oznaczony pogrubioną czcionką). Przyjrzyjmy się rysunkowi 17.3.



**Rysunek 17.3.** Kontrolka MapView w trybie widoku ulic

Na rysunku 17.3 zaprezentowano aplikację wyświetlającą mapę w trybie widoku ulic. Widać także, w jaki sposób można zmieniać skalę mapy oraz jak zmieniać jej tryb wyświetlania. Układ graficzny XML tej aplikacji jest ukazany na listingu 17.2.

**Uwaga!**

Na końcu rozdziału zamieszczamy adres URL, pod którym możemy pobrać projekty przygotowane dla tego rozdziału. Będziemy mogli dzięki temu bezpośrednio zimportować te projekty do środowiska Eclipse.

**Listing 17.2.** Układ graficzny XML aplikacji demonstracyjnej MapViewDemo

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Plik ten znajduje się w /res/layout/mapview.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal" android:layout_width="fill_parent"
        android:layout_height="wrap_content">

        <Button android:id="@+id/zoomin" android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="+"
            android:onClick="myClickHandler" android:padding="12px" />

        <Button android:id="@+id/zoomout" android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="-"
            android:onClick="myClickHandler" android:padding="12px" />

        <Button android:id="@+id/sat" android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="Satelita"
            android:onClick="myClickHandler" android:padding="8px" />

    
```

```
<Button android:id="@+id/street" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Ulice"
        android:onClick="myClickHandler" android:padding="8px" />

<Button android:id="@+id/traffic" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Ruch"
        android:onClick="myClickHandler" android:padding="8px" />

<Button android:id="@+id/normal" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Normalny"
        android:onClick="myClickHandler" android:padding="8px" />

</LinearLayout>

<com.google.android.maps.MapView
    android:id="@+id/mapview" android:layout_width="fill_parent"
    android:layout_height="wrap_content" android:clickable="true"
    android:apiKey="TUTAJ UMIESZCZAMY KLUCZ INTERFEJSU API MAPY" />

</LinearLayout>
```

---

Jak widać na listingu 17.2, nadrzędny menedżer `LinearLayout` zawiera podrzędny menedżer `LinearLayout` oraz widok `MapView`. W podrzędnym menedżerze `LinearLayout` zostały umieszczone przyciski, widoczne u góry ekranu na rysunku 17.3. Należy również pamiętać, żeby aktualizować wartość parametru `android:apiKey` wartością swojego klucza interfejsu API mapy.

Kod naszej przykładowej aplikacji obsługującej mapy został umieszczony na listingu 17.3.

---

**Listing 17.3.** Klasa rozszerzająca `MapActivity`, wczytująca układ graficzny XML

---

```
// Jest to plik MapViewDemoActivity.java
import android.os.Bundle;
import android.view.View;

import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;

public class MapViewDemoActivity extends MapActivity
{
    private MapView mapView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.mapview);

        mapView = (MapView) findViewById(R.id.mapview);
    }

    public void myClickHandler(View target) {
        switch(target.getId()) {
            case R.id.zoomin:
                mapView.getController().zoomIn();
                break;
            case R.id.zoomout:
                mapView.getController().zoomOut();
```

```

        break;
    case R.id.sat:
        mapView.setSatellite(true);
        break;
    case R.id.street:
        mapView.setStreetView(true);
        break;
    case R.id.traffic:
        mapView.setTraffic(true);
        break;
    case R.id.normal:
        mapView.setSatellite(false);
        mapView.setStreetView(false);
        mapView.setTraffic(false);
        break;
    }
    // Poniższy wiersz nie powinien być wymagany, jest jednak inaczej,
    // przynajmniej aż do wersji Froyo (Android 2.2)
    mapView.postInvalidateDelayed(2000);
}

@Override
protected boolean isLocationDisplayed() {
    return false;
}

@Override
protected boolean isRouteDisplayed() {
    return false;
}
}

```

Na listingu 17.3 widać, że wyświetlanie widoku `MapView` za pomocą metody `onCreate()` nie różni się od wyświetlania innych typów kontrolek: konfigurujemy widok treści interfejsu użytkownika w pliku układu graficznego zawierającego widok `MapView`, a pozostałe zadania są wykonywane przez ten widok automatycznie. O dziwo, obsługa funkcji zmiany skali mapy jest również bardzo prosta. Aby zmienić skalę mapy wyświetlonej w widoku, używa się klasy `MapController` widoku `MapView`. Najpierw wywołujemy metodę `mapView.getController()`, a następnie stosujemy metodę `zoomIn()` dla zmniejszania skali i metodę `zoomOut()` dla jej zwiększania. Taki sposób pracy jest jednopoziomowy; użytkownik musi wielokrotnie powtarzać czynność, aby dalej zwiększać lub zmniejszać skalę oglądanej mapy.

Równie prosto można zmienić tryb wyświetlania mapy. Widok `MapView` obsługuje kilka trybów:

- Domyślny jest widok standardowy mapy.
- W trybie widoku ulic zostaje nałożona na mapę dodatkowa warstwa, dzięki której można oglądać zdjęcia ulic zaznaczonych niebieskim obrysem. Zdjęcia te zostały wykonane za pomocą kamer, które umieszczone na pojazdach poruszających się po tych ulicach. Należy jednak pamiętać, że sama kontrolka `MapView` nie wyświetla zdjęć w widoku ulic. Do tego jest potrzebna odrębna kontrolka widoku. Temat ten zostanie dokładniej omówiony w rozdziale 25.
- W trybie satelitarnym są wyświetlane rzeczywiste zdjęcia lotnicze na mapy, dzięki którym można obserwować dachy budynków, korony drzew, drogi i tak dalej.

- W trybie ruchu ulicznego zaznacza się na mapie kolorowymi liniami, które ulice są łatwo przejezdne, a które są zakorkowane. Tryb ten jest obsługiwany jedynie dla największych arterii<sup>1</sup>.

Aby zmieniać tryby, należy wywołać właściwą metodę ustawiającą, podając wartość `true`. W pewnych przypadkach ustawienie jednego trybu wyłączy inny tryb. Na przykład nie można jednocześnie ustawić trybu widoku ulic z trybem ruchu ulicznego — w takim przypadku po wybraniu trybu ruchu ulicznego tryb widoku ulic zostanie automatycznie wyłączony. Żeby wyłączyć dany tryb, należy przydzielić mu wartość `false`. Wkrótce zajmiemy się omawianiem klasy `Over1ay`, teraz jednak wystarczy wiedzieć, że tryb ruchu ulicznego oraz tryb widoku ulic *nie korzystają* z tych obiektów.

**Uwaga!**

Instrukcja `MapView.invalidateDelayed(2000)` umożliwia uniknięcie problemu wynikającego z korzystania z trybów widoku ulic oraz ruchu ulicznego. Problem ten wiąże się z wewnętrznym wykorzystywaniem wątków do pobierania danych pozwalających na wyświetlanie linii w tych dwóch trybach. Więcej informacji znajdziemy na stronie <http://code.google.com/p/android/issues/detail?id=10317>, dotyczącej problemu numer 10317.

Aby przesuwać mapę, należy dla widoku `MapView` w pliku XML skonfigurować atrybut `android:clickable="true"` — w przeciwnym wypadku będzie można wyłącznie zmniejszać i zwiększać skalę mapy. W kodzie Java można tego dokonać za pomocą wywołania metody `setClickable(true)` w widoku `MapView`.

Ostatnia rzecz, którą można powiedzieć o naszym przykładowym projekcie, dotyczy dwóch metod: `isLocationDisplayed()` i `isRouteDisplayed()`. W dokumentacji tych metod widnieje informacja, że są one wymagane przez warunki korzystania z usług firmy Google, chociaż w trakcie uzyskiwania klucza interfejsu map nie ma w zatwierdzanym dokumencie żadnej wzmianki na ich temat. Nie jesteśmy prawnikami, zalecamy jednak uwzględnienie tych metod. Aplikacja musi przekazywać serwerowi map wartości `true` lub `false` w celu określenia, czy położenie geograficzne urządzenia jest wyświetlane lub czy są wyświetlane informacje o trasie, na przykład wyznaczanie trasy samochodowej.

Przyznajemy, że w Androidzie ilość kodu potrzebnego do wyświetlenia mapy oraz zaimplementowania funkcji zmiany skali i trybów przeglądania jest minimalna (listing 17.3). Istnieje jednak jeszcze prostszy sposób wstawienia kontrolek służących do zmianiania skali mapy. Przyjrzyjmy się plikowi XML układu graficznego oraz kodowi na listingu 17.4.

---

**Listing 17.4. Prostsza zmiana skali wyświetlonej mapy**

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/mapview.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <com.google.android.maps.MapView android:id="@+id/mapview"
        android:layout_width="fill_parent"
```

---

<sup>1</sup> Tryb widoku ulicznego nie obejmuje Polski — przyp. tłum.

```
        android:layout_height="wrap_content"
        android:clickable="true"
        android:apiKey="TUTAJ UMIESZCZAMY KLUCZ INTERFEJSU API MAPY"
    />
</RelativeLayout>

public class MapViewDemoActivity extends MapActivity
{
    private MapView mapView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.mapview);

        mapView = (MapView)findViewById(R.id.mapview);
        mapView.setBuiltInZoomControls(true);
    }

    @Override
    protected boolean isLocationDisplayed() {
        return false;
    }

    @Override
    protected boolean isRouteDisplayed() {
        return false;
    }
}
```

Różnica pomiędzy listingami 17.4 a 17.3 polega na przekształceniu układu graficznego naszego widoku w taki sposób, żeby wykorzystywał menedżer RelativeLayout. Usunęliśmy wszystkie kontrolki odpowiedzialne za zmianę skali oraz trybu wyświetlania mapy. Cała magia znajduje się w kodzie Java, a nie w układzie graficznym. Kontrolka MapView zawiera już kontrolki umożliwiające zmianę skali oglądanej mapy. Wystarczy je włączyć za pomocą metody `setBuiltInZoomControls()`. Na rysunku 17.4 zostały pokazane domyślne kontrolki zmiany skali mapy w widoku MapView.

Teraz dowiedzmy się, w jaki sposób dodawać własne dane do mapy.

## Dodawanie znaczników za pomocą nakładek

Usługa Google Maps posiada funkcję pozwalającą na umieszczanie własnych informacji na mapie. Można ją przetestować, wyszukując na przykład pizzerie w swoim mieście: serwis Google Maps umieszcza ikony pinezki lub dymki informacyjne nad miejscami spełniającymi kryteria wyszukiwania. Jest to możliwe, ponieważ Google Maps dopuszcza nakładanie własnej warstwy na mapę. W Androidzie istnieje kilka klas usprawniających nakładanie takich warstw. Najważniejszą klasą tego typu jest `Overlay`, można jednak równie dobrze korzystać z rozszerzenia tej klasy, nazwanego `ItemizedOverlay`. Na listingu 17.5 został umieszczony przykład. Możemy w tym projekcie wykorzystać plik układu graficznego zaprezentowany na listingu 17.4.



Rysunek 17.4. Wbudowane kontrolki widoku MapView

**Listing 17.5.** Zaznaczanie punktów na mapie za pomocą klasy ItemizedOverlay

```
import java.util.ArrayList;
import java.util.List;

import android.graphics.Canvas;
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.widget.LinearLayout;

import com.google.android.maps.GeoPoint;
import com.google.android.maps.ItemizedOverlay;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;
import com.google.android.maps.OverlayItem;

public class MappingOverlayActivity extends MapActivity {
    private MapView mapView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.mapview);

        mapView = (MapView) findViewById(R.id.mapview);
        mapView.setBuiltInZoomControls(true);

        Drawable marker=getResources().getDrawable(R.drawable.mapmarker);
        marker.setBounds( (int) (-marker.getIntrinsicWidth()/2),
                         -marker.getIntrinsicHeight(),
                         (int) (marker.getIntrinsicWidth()/2),
                         0);
```

```

InterestingLocations funPlaces = new InterestingLocations(marker);
mapView.getOverlays().add(funPlaces);

GeoPoint pt = funPlaces.getCenterPt();
int latSpan = funPlaces.getLatSpanE6();
int lonSpan = funPlaces.getLonSpanE6();
Log.v("Overlays", "Rozpiętość szerokości geog. wynosi " + latSpan);
Log.v("Overlays", "Rozpiętość długości geog. wynosi " + lonSpan);

MapController mc = mapView.getController();
mc.setCenter(pt);
mc.zoomToSpan((int)(latSpan*1.5), (int)(lonSpan*1.5));
}

@Override
protected boolean isLocationDisplayed() {
    return false;
}

@Override
protected boolean isRouteDisplayed() {
    return false;
}

class InterestingLocations extends ItemizedOverlay {
    private ArrayList<OverlayItem> locations = new ArrayList<OverlayItem>();
    private GeoPoint center = null;
    public InterestingLocations(Drawable marker)
    {
        super(marker);

        // tworzy zaznaczenia miejsc godnych uwagi
        GeoPoint disneyMagicKingdom = new
GeoPoint((int)(28.418971*1000000),(int)(-81.581436*1000000));
        GeoPoint disneySevenLagoon = new
GeoPoint((int)(28.410067*1000000),(int)(-81.583699*1000000));

        locations.add(new OverlayItem(disneyMagicKingdom ,
"Magiczne Królestwo", "Magiczne Królestwo"));
        locations.add(new OverlayItem(disneySevenLagoon ,
"Laguna Siedmiu Mór", "Laguna Siedmiu Mór"));

        populate();
    }

    // Dodaliśmy tę metodę w celu odnalezienia środka klastra.
    // Rozpoczyna od każdej krawędzi po przeciwniej stronie i porusza się wraz
    // z każdym punktem. Górną krawędź posiada wartość +90, dolna — -90,
    // zachodnia — -180, a wschodnia — +180
    public GeoPoint getCenterPt() {
        if(center == null) {
            int northEdge = -90000000; // np. -90E6 mikrostopni
            int southEdge = 90000000;
            int eastEdge = -180000000;

```

```
int westEdge = 180000000;
Iterator<OverlayItem> iter = locations.iterator();
while(iter.hasNext()) {
    GeoPoint pt = iter.next().getPoint();
    if(pt.getLatitudeE6() > northEdge)
        northEdge = pt.getLatitudeE6();
    if(pt.getLatitudeE6() < southEdge)
        southEdge = pt.getLatitudeE6();
    if(pt.getLongitudeE6() > eastEdge)
        eastEdge = pt.getLongitudeE6();
    if(pt.getLongitudeE6() < westEdge)
        westEdge = pt.getLongitudeE6();
}
center = new GeoPoint((int)((northEdge +southEdge)/2),
                      (int)((westEdge + eastEdge)/2));
}
return center;
}

@Override
public void draw(Canvas canvas, MapView mapView, boolean shadow) {
    //Ukrywa cień poprzez ustanowienie wartości false w argumencie shadow
    super.draw(canvas, mapView, shadow);
}

@Override
protected OverlayItem createItem(int i) {
    return locations.get(i);
}

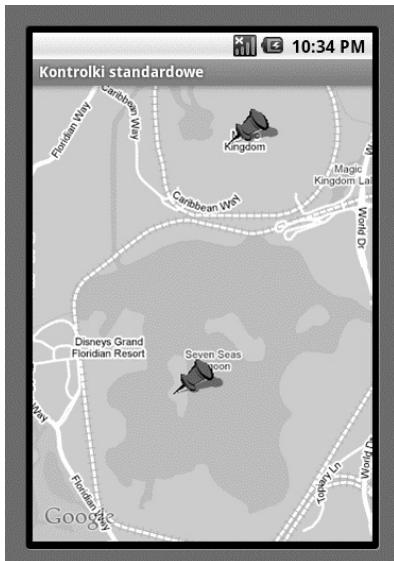
@Override
public int size() {
    return locations.size();
}
}
```

---

Na listingu 17.5 pokazujemy, w jaki sposób można nakładać znaczniki na mapę. W przykładzie tym zaznaczyliśmy dwa miejsca: Magiczne Królestwo Disneya (ang. *Magic Kingdom*) oraz Lagunę Siedmiu Mór (ang. *Seven Seas Lagoon*). Obydwa miejsca znajdują się w pobliżu miasta Orlando na Florydzie (rysunek 17.5).

**Uwaga!**

Aby nasza aplikacja demonstracyjna zadziałała, musimy wprowadzić obiekt rysowany, który posłuży nam za znacznik. Obraz ten musi zostać zachowany w folderze */res/drawable* w taki sposób, żeby identyfikator tego zasobu w metodzie *getDrawable()* odpowiadał nazwie pliku obrazu. W razie możliwości postarajmy się, aby obszar otaczający znacznik był przezroczysty. Kilka przykładowych znaczników znajdziemy w zasobach projektu utworzonego na potrzeby tego rozdziału.



**Rysunek 17.5.** Widok MapView wraz ze znacznikami

Aby dodawać znaczniki do mapy, należy dla niej utworzyć i dodać do niej rozszerzenie klasy `com.google.android.maps.Overlay`. Nie można utworzyć samej klasy `Overlay`, zatem należy ją rozszerzyć lub skorzystać z jednego z rozszerzeń. W naszym przykładzie zaimplementowaliśmy klasę `InterestingLocations` rozszerzającą klasę `ItemizedOverlay`, która z kolei rozszerza klasę `Overlay`. Klasa `Overlay` definiuje kontrakt dla nakładki, a klasa `ItemizedOverlay` stanowi jej przydatną implementację, pozwalającą na łatwe utworzenie listy lokalizacji, które mogą zostać następnie zaznaczone na mapie.

Standardowym algorytmem działania jest rozszerzenie klasy `ItemizedOverlay` oraz dodanie elementów — miejsc wartych odwiedzenia — do konstruktora. Po utworzeniu takich miejsc można wywołać metodę `populate()` w klasie `ItemizedOverlay`. Metoda `populate()` zapisuje w pamięci podręcznej element bądź elementy `OverlayItem`. Klasa `ItemizedOverlay` wywołuje wewnętrznie metodę `size()` służącą do określenia liczby nakładanych elementów, a następnie wchodzi w pętlę, wywołując metodę `createItem(i)` wobec każdego elementu. Metoda `createItem` przekazuje gotowy element, utworzony na podstawie indeksu tablicy.

Jak widać na listingu 17.5, tworzymy po prostu punkty i wywołujemy metodę `populate()`, aby wyświetlić znaczniki na mapie. Kontrakt klasy `Overlay` wykonuje pozostałą część pracy. Aby to zadziałało, metoda `onCreate()` aktywności tworzy wystąpienie klasy `InterestingLocations` i przekazuje obiekt rysowany, który będzie wyświetlany jako znaczniki. Następnie metoda `onCreate()` dodaje wystąpienie `InterestingLocations` do zbioru nakładek (`mapView.getOverlays().add()`).

Wybrany przez nas obiekt klasy `Drawable` musi być przystosowany do korzystania z nakładki `ItemizedOverlay`. Interfejs API map musi otrzymać współrzędne punktu (0,0) obiektu `Drawable`. Punkt ten będzie definiował dokładne miejsce na mapie, reprezentowane przez znacznik. Możemy tego sami dokonać za pomocą metody `setBounds()` klasy `Drawable`, która została przedstawiona w powyższym przykładzie. Argumenty reprezentują współrzędne lewej, prawej, górnej i dolnej krawędzi; można również wykorzystać metody `getIntrinsicHeight()` oraz `getIntrinsicWidth()` do określenia, jak wysoki i szeroki jest obiekt `Drawable`.

W naszym przykładzie punkt (0,0) znajduje się pośrodku dolnej krawędzi. Pamiętajmy, że w systemie współrzędnych wartości punktów rosną od lewej do prawej strony i z góry do dołu. Z tego właśnie powodu współrzędna górnej krawędzi musi przybrać wartość ujemną.

Klasa `ItemizedOverlay` posiada kilka metod ułatwiających definiowanie krawędzi w obiektach typu `Drawable`. Należą do nich metody `boundCenterBottom()` oraz `boundCenter()`. Pierwsza z wymienionych metod pozwala na dokładne taki efekt, jaki uzyskaliśmy w naszym przykładzie, czyli umieszczenie punktu o współrzędnych (0,0) dokładnie pośrodku dolnej krawędzi obiektu. Za pomocą drugiej metody umieszczamy ten punkt dokładnie w środku obiektu. Standardowym rozwiązaniem jest wywołanie jednej z tych metod w konstruktorze. Moglibyśmy tego dokonać, nie korzystając z metody `setBounds()`:

```
public InterestingLocations(Drawable marker)
{
    super(boundCenterBottom(marker));
    [ ... ]
```

Zauważmy także, że możemy korzystać z obiektów `Drawable` o dowolnym rozmiarze i kształcie. Świetnym zabiegem kosmetycznym jest zastosowanie przezroczystego tła dookoła wybranego kształtu. Dymki wykorzystywane w usłudze Mapy Google nie są kwadratowe, a ponieważ ich tło jest przezroczyste, widzimy pod jego spodem fragment mapy. Jest to dobry解决方案, ponieważ interfejs map rysuje również cień obiektu rzucany na mapę, a lepiej by wyglądało, gdyby cień ten przybrał zarys narysowanego obiektu, a nie miał prostokątnego kształtu (no dobrze,ściślej: kształtu równoległoboku).

A jeśli Czytelnik nie chce tego cienia? I temu można zaradzić. Wystarczy, że przesłonimy metodę `draw()` naszej rozszerzonej klasy `ItemizedOverlay` i nadamy argumentowi `shadow` wartość `false` w czasie wywoływania metody `draw()` w nadrzednej klasie. Spójrzmy na metodę `draw()` w naszym przykładowym kodzie. Wspomnieliśmy, że obiekt `Drawable`, którym posłużyliśmy się do utworzenia nakładki `ItemizedOverlay`, jest naszym domyślnym znacznikiem. Każdy obiekt `OverlayItem` może posiadać unikatowy znacznik ustawiany za pomocą metody `setMarker()` w innym obiekcie `Drawable`. Możemy ustanawiać te znaczniki podczas tworzenia wystąpień obiektów `OverlayItem` albo możemy zrobić to później. Przyjrzymy się jeszcze znacznikom w rozdziale 25., w trakcie omawiania ekranów dotykowych, zademonstrujemy jeszcze także ich inne, ciekawe możliwości.

Gdy nakładka jest już powiązana z mapą, musimy jeszcze zdefiniować właściwą pozycję na mapie, żeby znaczniki były widoczne na ekranie. W tym celu trzeba wyznaczyć interesujący nas punkt jako środek ekranu. Metoda `getCenter()` nakładki przekazuje współrzędne pierwszego punktu, a nie punktu środkowego, jak można by się spodziewać. Nakładka `ItemizedOverlay` sortuje określone wcześniej punkty i wskazuje pierwszy z nich. Zatem w celu odnalezienia punktu środkowego implementujemy własną metodę `getCenterPt()`, która będzie iterowała przez punkty i wyszukiwała punkt środkowy. Metoda `setCenter()` kontrolera widoku mapy określa środek wyświetlanego elementu, trzeba tylko przekazać jej obliczony punkt środkowy.

Dzięki metodzie `setZoom()` klasy `MapController` określamy skalę oglądanej mapy. Metoda ta przyjmuje wartości od 1 do 21, gdzie 21 oznacza najmniejszą możliwą skalę, a 1 — największą. Ponieważ jednak nie wiemy dokładnie, jaka skala będzie potrzebna, aby wyświetlić cały interesujący użytkownika zakres mapy na ekranie, wprowadzamy metodę `zoomToSpan()` klasy `MapController`. Trzeba wprowadzić wysokość i szerokość prostokąta, w którym zostanie wyświetlona mapa. Na szczęście nakładka `ItemizedOverlay` posiada dwie metody pozwalające nam na określenie tych wymiarów: `getLatSpanE6()` przekazuje zakres szerokości geograficz-

nnych, a `getLonSpanE6()` — definiuje zakres długości geograficznych. Można tu wykorzystać wartości otrzymane w metodzie `zoomToSpan()`. Zwróćmy uwagę, że rozszerzyliśmy nasz prostokąt o współczynnik 1,5, czyli interesujące nas punkty nie znajdują się już dokładnie na krawędziach mapy podczas wyświetlania.

Kolejnym interesującym aspektem, widocznym na listingu 17.5, jest tworzenie elementu (elementów) `OverlayItem`. Aby utworzyć element `OverlayItem`, wymagany jest obiekt typu `GeoPoint`. Klasa `GeoPoint` reprezentuje położenie geograficzne poprzez wyznaczenie długości i szerokości geograficznej w mikrostopniach. W naszym przykładzie uzyskaliśmy współrzędne geograficzne Magicznego Królestwa i Laguny Siedmiu Mór za pomocą internetowych witryn geokodujących (jak się wkrótce okaże, geokodowanie może posłużyć do przekształcania adresu na parę współrzędnych — szerokość geograficzna). Następnie przekonwertowaliśmy współrzędne geograficzne na mikrostopnie — interfejsy API akceptują te jednostki — mnożąc wynik przez 1 000 000 i otrzymując liczbę przekształcającą na liczbę całkowitą.

Do tej pory pokazaliśmy, w jaki sposób umieszczać znaczniki na mapie. Możliwości nakładek nie są jednak ograniczone wyłącznie do wyświetlania pinezek i chmurek informacyjnych. Można za ich pomocą przeprowadzać inne czynności. Możemy na przykład wyświetlać animacje wędrujące po mapie lub pokazywać symbole, na przykład frontów atmosferycznych lub burz.

Podsumowując, Czytelnik mógłby się zgodzić, że umieszczanie znaczników na mapie nie może być prostsze. Ale może jednak mógłby? Nie posiadamy bazy danych współrzędnych geograficznych, domyślamy się jednak, że moglibyśmy utworzyć co najmniej jeden obiekt `GeoPoint`, korzystając ze zwykłego adresu. Rzeczywiście tak jest. Do tego służy klasa `Geocoder`, którą zajmiemy się w następnych podrozdziałach.

## Pakiet do obsługi danych o położeniu geograficznym

Pakiet `android.location` zawiera funkcje usług umożliwiających pracę na danych o położeniu geograficznym. W tym podrozdziale zajmiemy się dwoma istotnymi elementami tego pakietu: klasą `Geocoder` oraz usługą `LocationManager`. Rozpoczniemy od klasy `Geocoder`.

### Geokodowanie w Androidzie

Jeżeli mapy mają być wykorzystywane w jakiś praktyczny sposób, prawdopodobnie należy przekonwertować adres (lub lokację) do współrzędnych geograficznych. Pojęcie to jest znane jako **geolokalizacja**, a klasa `android.location.Geocoder` posiada tę funkcję. W rzeczywistości klasa `Geocoder` umożliwia konwersję w obydwie strony — może przekształcić adres do współrzędnych geograficznych oraz przetłumaczyć parę szerokość – długość geograficzna na listę adresów. Klasa ta posiada następujące metody:

- `List<Address> getFromLocation(double latitude, double longitude, int maxResults),`
- `List<Address> getFromLocationName(String locationName, int maxResults, double lowerLeftLatitude, double lowerLeftLongitude, double upperRightLatitude, double upperRightLongitude),`
- `List<Address> getFromLocationName(String locationName, int maxResults).`

Okazuje się, że przetwarzanie adresu nie zawsze zachodzi tak samo. Na przykład metody `getFromLocationName()` akceptują nazwę miejsca, adres fizyczny, kod lotniska lub zwyczajową nazwę. Zatem metody przekazują nie jeden adres, a całą ich listę. Z tego powodu istnieje możliwość ograniczenia listy wyników poprzez ustawienie wartości `maxResults` w zakresie od 1 do 5. Przyjrzyjmy się przykładowi.

Listing 17.6 przedstawia układ graficzny XML oraz odpowiadający mu kod Java interfejsu użytkownika z rysunku 17.6. Żeby ten przykładowy kod zadziałał, należy umieścić we właściwym miejscu własny klucz interfejsu API mapy.

---

**Listing 17.6. Praca z klasą Geocoder**

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/geocode.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <LinearLayout android:layout_width="fill_parent"
        android:layout_alignParentBottom="true"
        android:layout_height="wrap_content" android:orientation="vertical" >

        <EditText android:layout_width="fill_parent" android:id="@+id/location"
            android:layout_height="wrap_content" android:text="White House"/>

        <Button android:id="@+id/geocodeBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:text="Znajdź lokację"/>
    </LinearLayout>

    <com.google.android.maps.MapView
        android:id="@+id/geoMap" android:clickable="true"
        android:layout_width="fill_parent"
        android:layout_height="320px"
        android:apiKey="WSTAWIAMY TUTAJ KLUCZ INTERFEJSU API MAPY"
        />

</RelativeLayout>

package com.androidbook.maps.geocoding;

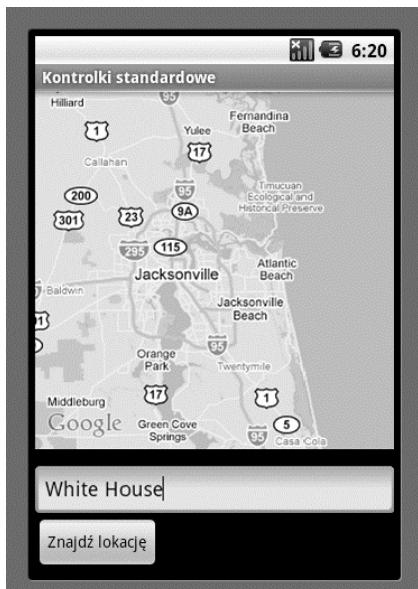
import java.io.IOException;
import java.util.List;

import android.location.Address;
import android.location.Geocoder;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;

public class GeocodingDemoActivity extends MapActivity
```

```
{  
    Geocoder geocoder = null;  
    MapView mapView = null;  
  
    @Override  
    protected boolean isLocationDisplayed() {  
        return false;  
    }  
    @Override  
    protected boolean isRouteDisplayed() {  
        return false;  
    }  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState)  
    {  
        super.onCreate(savedInstanceState);  
  
        setContentView(R.layout.geocode);  
        mapView = (MapView) findViewById(R.id.geoMap);  
        mapView.setBuiltInZoomControls(true);  
  
        // Szerokość/długość geograficzna miasta Jacksonville na Florydzie  
        int lat = (int)(30.334954*1000000);  
        int lng = (int)(-81.5625*1000000);  
        GeoPoint pt = new GeoPoint(lat,lng);  
        mapView.getController().setZoom(10);  
        mapView.getController().setCenter(pt);  
  
        geocoder = new Geocoder(this);  
  
        public void onClick(View arg0) {  
            try {  
                EditText loc = (EditText) findViewById(R.id.location);  
                String locationName = loc.getText().toString();  
  
                List<Address> addressList =  
geocoder.getFromLocationName(locationName, 5);  
                if(addressList!=null && addressList.size()>0)  
                {  
                    int lat = (int)(addressList.get(0).getLatitude()*1000000);  
                    int lng = (int)(addressList.get(0).getLongitude()*1000000);  
  
                    GeoPoint pt = new GeoPoint(lat,lng);  
                    mapView.getController().setZoom(15);  
                    mapView.getController().setCenter(pt);  
                }  
  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



**Rysunek 17.6.** Geokodowanie miejsca przy znanej lokalizacji

Aby sprawdzić zastosowanie geokodowania w Androidzie, wpiszmy nazwę lokacji lub jej adres w polu *EditText*, a następnie kliknijmy przycisk *Znajdź lokację*. Żeby odnaleźć adres lokacji, wywołujemy metodę `getFromLocationName()` klasy `Geocoder`. Lokacja może być adresem lub znaną nazwą (w naszym przypadku *White House*, czyli Biały Dom). Geokodowanie może być czasochronnym procesem, dlatego proponujemy ograniczyć liczbę wyników do pięciu, zgodnie z sugestią dokumentacji Androida.

Po wywołaniu metody `getFromLocationName()` otrzymujemy listę adresów. Nasza przykładowa aplikacja pobiera listę adresów i, jeżeli zostanie jakiś znaleziony, przetwarza pierwszy z nich. Każdy adres posiada współrzędne szerokości i długości geograficznej, służące do utworzenia obiektu `GeoPoint`. Następnie otrzymujemy dostęp do kontrolera mapy i przenosimy się do wskazanego punktu. Wartość poziomu przybliżenia jest liczbą całkowitą w zakresie wartości od 1 do 21. Z każdym krokiem następuje dwukrotne przybliżenie. Moglibyśmy zaprezentować okno dialogowe wyświetlające jednocześnie wiele poszukiwanych lokacji, na razie jednak pokażemy tylko przypadek zawierający jedno znalezione miejsce.

W naszej przykładowej aplikacji odczytujemy jedynie długość i szerokość geograficzną zwracanego obiektu `Address`. W rzeczywistości możemy otrzymać ogrom danych powiązanych z obiektem tego typu, włącznie ze zwyczajową nazwą miejsca, adresem, miastem, prowincją, kodem pocztowym, państwem, a nawet numerem telefonu i adresem URL.

#### Uwaga!

W przeciwnieństwie do interfejsu map, usługi geolokalizacyjne nie korzystają z mikrostopni. Częstą przyczyną błędów jest pozostawianie nieskonwertowanych jednostek. Żeby przekazać wartości długości i szerokości geograficznej obiektu `Address` do metody interfejsu map, musimy je najpierw pomnożyć przez 1 000 000.

Należy zrozumieć kilka faktów związanych z geokodowaniem:

- Po pierwsze, nie zawsze otrzymujemy dokładny adres. Oczywiście, dokładność zwracanych adresów zależy od danych wejściowych, zatem należy starać się przekazywać klasie Geocoder jak najdokładniejszą nazwę szukanego obiektu.
- Po drugie, jeśli to możliwe, warto podawać parametr maksymalnej liczby wyników w zakresie od 1 do 5.
- Wreszcie, należy poważnie zastanowić się nad przeprowadzaniem operacji geokodowania w wątku innym niż wątek interfejsu użytkownika. Wpływają na to dwa czynniki. Pierwszy powód jest oczywisty: geokodowanie jest czasochłonnym procesem i nie chcemy, żeby interfejs UI trwał bezczynnie podczas wyszukiwania lokacji, co spowoduje w końcu zamknięcie aktywności. Drugi dotyczy słusznego założenia, że w przypadku urządzenia mobilnego połączenie sieciowe może zostać w każdej chwili zerwane oraz że takie połączenie jest słabe. Zatem należy odpowiednio zająć się wyjątkami I/O (ang. *Input/Output* — wejście-wyjście) oraz przekraczaniem limitów czasowych. Po przetworzeniu adresów można będzie przekazać wyniki do wątku interfejsu UI. Przyjrzyjmy się temu bliżej.

## Geokodowanie za pomocą wątków przebiegających w tle

Bardzo popularnym zwyczajem jest przeznaczanie wątków przebiegających w tle do przetwarzania czasochłonnych operacji. Ogólny algorytm dotyczy przetwarzania zdarzenia interfejsu UI (na przykład kliknięcia przycisku), które inicjalizuje czasochlonną operację. Za pomocą procedury obsługi zdarzeń tworzy się i uruchamia nowy wątek, przeznaczony do wykonania tej operacji. W międzyczasie wątek interfejsu UI wraca do tego interfejsu w celu zachowania interakcji z użytkownikiem podczas przetwarzania operacji przez wątek przebiegający w tle. Po zakończeniu operacji część interfejsu UI może zostać zaktualizowana lub użytkownik otrzyma powiadomienie. Wątek przebiegający w tle nie aktualizuje bezpośrednio interfejsu UI; informuje raczej wątek interfejsu UI o konieczności aktualizacji. Na listingu 17.7 został przedstawiony omówiony algorytm na przykładzie geokodowania. Użyjemy tego samego pliku *geocode.xml* co poprzednio. Nie ma również przeciwwskazań co do wykorzystania stosowanego wcześniej pliku *AndroidManifest.xml*.

**Listing 17.7.** Geokodowanie w oddzielnym wątku

```
package com.androidbook.maps.geocodingthreads;

import java.io.IOException;
import java.util.List;

import android.app.AlertDialog;
import android.app.Dialog;
import android.app.ProgressDialog;
import android.location.Address;
import android.location.Geocoder;
import android.os.Bundle;
import android.os.Handler;
import android.os.Message;
import android.view.View;
import android.widget.EditText;

import com.google.android.maps.GeoPoint;
```

```
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;

public class GeocodingDemoActivity extends MapActivity
{
    Geocoder geocoder = null;
    MapView mapView = null;
    ProgressDialog progDialog=null;
    List<Address> addressList=null;
    @Override
    protected boolean isRouteDisplayed() {
        return false;
    }

    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);

        setContentView(R.layout.geocode);
        mapView = (MapView)findViewById(R.id.geoMap);
        mapView.setBuiltInZoomControls(true);

        // szerokość/długość geograficzna miasta Jacksonville na Florydzie
        int lat = (int)(30.334954*1000000);
        int lng = (int)(-81.5625*1000000);
        GeoPoint pt = new GeoPoint(lat,lng);
        mapView.getController().setZoom(10);
        mapView.getController().setCenter(pt);

        geocoder = new Geocoder(this);

        public void doClick(View view) {
            EditText loc = (EditText)findViewById(R.id.location);
            String locationName = loc.getText().toString();

            progDialog =
            ProgressDialog.show(GeocodingDemoActivity.this,
            "Przetwarzanie...", "Szukanie lokacji...", true, false);

            findLocation(locationName);
        }

        private void findLocation(final String locationName)
        {
            Thread thrd = new Thread()
            {
                public void run()
                {
                    try {
                        // wykonuje pracę w tle
                        addressList = geocoder.getFromLocationName(locationName, 5);
                        // wysyła komunikat do procedury obsługi zdarzeń w celu przetworzenia
                        // wyników
                        uiCallback.sendEmptyMessage(0);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            };
            thrd.start();
        }
    }
}
```

```

        }
    }
}

// procedura obslugi metody zwrotnej wątku interfejsu ui
private Handler uiCallback = new Handler()
{
    @Override
    public void handleMessage(Message msg)
    {
        // usuwa okno dialogowe
        progDialog.dismiss();

        if(addressList!=null && addressList.size()>0)
        {
            int lat = (int)(addressList.get(0).getLatitude()*1000000);
            int lng = (int)(addressList.get(0).getLongitude()*1000000);
            GeoPoint pt = new GeoPoint(lat,lng);
            mapView.getController().setZoom(15);
            mapView.getController().animateTo(pt);
        }
        else
        {
            Dialog foundNothingDlg = new
AlertDialog.Builder(GeocodingDemoActivity.this)
                .setIcon(0)
                .setTitle("Wyszukiwanie lokacji zakończone niepowodzeniem")
                .setPositiveButton("OK", null)
                .setMessage("Nie znaleziono lokacji... ")
                .create();
            foundNothingDlg.show();
        }
    }
};
}

```

Kod na listingu 17.7 jest zmodyfikowaną wersją przykładu z listingu 17.6. Różnica polega na tym, że w metodzie `onClick()` wyświetlamy okno dialogowe postępów i wywołujemy metodę `findLocation()` (rysunek 17.7). Metoda `findLocation()` tworzy następnie nowy wątek i wywołuje metodę `start()`, czego ostatecznym wynikiem jest wywołanie metody `run()` tego wątku. W tej metodzie wykorzystujemy klasę `Geocoder` do znalezienia szukanej lokalizacji. Po za-kończeniu wyszukiwania musimy wysłać komunikat do obiektu, który może nawiązać interakcję z wątkiem interfejsu UI, ponieważ musimy zaktualizować mapę. Do tego celu służy klasa `android.os.Handler`. Z poziomu wątku przebiegającego w tle wywołujemy metodę `uiCallback.sendMessage(0)`, aby wątek interfejsu UI mógł przetwarzać wyniki wyszukiwania. W naszym przypadku nie musimy tak naprawdę przesyłać żadnej treści w komunikacie, ponieważ dane są współdzielone w obiekcie `addressList`. Kod wywołuje następnie procedurę metody zwrotnej, która usuwa okno dialogowe, a następnie sprawdza listę `addressList` zwróconą przez klasę `Geocoder`. Poprzez wywołanie zwrotne mapa zostaje następnie zaktualizowana o wyniki wyszukiwania lub zostaje wyświetlony alert informujący o braku wyników wyszukiwania. Interfejs użytkownika dla tego przykładu został zilustrowany na rysunku 17.7.



Rysunek 17.7. Wyświetlanie okna postępów w czasie przeprowadzania długich operacji

## Usługa LocationManager

Usługa *LocationManager* jest jedną z najważniejszych funkcji oferowanych przez pakiet `android.location`. Dzięki niej dostępne stają się dwie funkcje: mechanizm pozwalający uzyskać współrzędne geograficzne urządzenia oraz technologia informująca (poprzez intencję) o tym, że urządzenie znalazło się w określonym rejonie geograficznym.

W tym punkcie opiszymy tajniki działania usługi *ServiceManager*. Żeby korzystać z tej usługi, należy najpierw utworzyć do niej odniesienie. Na listingu 17.8 został ukazany wzorzec korzystania z tej usługi.

**Listing 17.8.** Korzystanie z usługi ServiceManager

```
package com.androidbook.maps.locationmanager;

import java.util.List;

import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationManager;
import android.os.Bundle;

public class LocationManagerDemoActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        LocationManager locMgr =

```

```

        (LocationManager)this.getSystemService(Context.LOCATION_SERVICE);
        Location loc = locMgr.getLastKnownLocation(LocationManager.GPS_PROVIDER);
        List<String> providerList = locMgr.getAllProviders();
    }
}

```

---

Usługa *LocationManager* jest usługą systemową. Usługi systemowe są uzyskiwane z kontekstu za pomocą ich nazw; nie są tworzone bezpośrednio. Klasa *android.app.Activity* zawiera metodę *getSystemService()*, dzięki której można uzyskać dostęp do usługi systemowej. Jak zostało pokazane na listingu 17.8, wywołujemy metodę *getSystemService()* i umieszczamy w niej nazwę pożądanej usługi — w tym przypadku *Context.LOCATION\_SERVICE*.

Usługa *LocationManager* dostarcza szczegóły o położeniu geograficznym poprzez dostawców lokalizacji. Obecnie dostępne są trzy typy tych dostawców:

- **Dostawcy systemu GPS** uzyskują dane o lokalizacji za pomocą Globalnego Systemu Pozycjonowania.
- **Dostawcy sieciowi** korzystają z wież operatorów sieci komórkowych lub z sieci bezprzewodowych Wi-Fi.
- **Dostawca pasywny** przypomina aplikację weszczącą, wyszukującą aktualizacje położenia i przekazującą je do aplikacji żądających tego typu informacji, nawet gdy sama aplikacja przechowująca pasywnego dostawcę nie wymaga tych danych. Oczywiście, jeżeli żadna aplikacja nie będzie potrzebowała tych danych, my również ich nie otrzymamy.

Klasa *LocationManager* może uzyskać informacje dotyczące ostatniej znanej lokalizacji urządzenia dzięki metodzie *getLastKnownLocation()*. Informacja o położeniu geograficznym jest uzyskiwana od dostawcy, zatem parametrem tej metody jest nazwa używanego dostawcy. Akceptowanymi wartościami tego parametru są *LocationManager.GPS\_PROVIDER*, *LocationManager.NETWORK\_PROVIDER* oraz *LocationManager.PASSIVE\_PROVIDER*. Aby nasza aplikacja skutecznie zdobyła informacje o lokalizacji urządzenia, będziemy musieli wstawić odpowiednie wpisy o uprawnieniach w pliku *AndroidManifest.xml*. Dla dostawców GPS i pasywnych wymagane będzie uprawnienie *android.permission.ACCESS\_FINE\_LOCATION*, natomiast dostawcy sieciowi mogą, zależnie od potrzeb, korzystać z uprawnień *android.permission.ACCESS\_COARSE\_LOCATION* lub *android.permission.ACCESS\_FINE\_LOCATION*. Założymy na przykład, że nasza aplikacja będzie korzystała z danych GPS lub sieciowych do aktualizowania położenia. Ponieważ dostawca GPS wymaga uprawnienia *android.permission.ACCESS\_FINE\_LOCATION*, spełnimy tym samym wymogi dostawcy sieciowego i nie będziemy musieli definiować uprawnienia *android.permission.ACCESS\_COARSE\_LOCATION*. Gdybyśmy korzystali tylko z dostawcy sieciowego, wystarczyłoby nam wprowadzenie uprawnienia *android.permission.ACCESS\_COARSE\_LOCATION* w pliku manifeście.

Dzięki wywołaniu metody *getLastKnownLocation()* otrzymujemy instancję klasy *android.location.Location* lub wartość null, jeśli lokalizacja jest niedostępna. Klasa *Location* uzyskuje takie informacje o położeniu geograficznym, jak szerokość i długość geograficzna, data ostatniej aktualizacji położenia, a także potencjalnie wysokość, prędkość oraz peleng. Obiekt *Location* może nas również poinformować za pomocą metody *getProvider()*, od którego dostawcy pochodzą; do wyboru mamy *GPS\_PROVIDER* lub *NETWORK\_PROVIDER*. Jeżeli otrzymujemy informacje o położeniu w trybie *PASSIVE\_PROVIDER*, w rzeczywistości jedynie wyszukujemy dane o aktualizacji położenia, zatem wszystkie informacje pochodzą ostatecznie od systemu GPS lub dostawcy sieciowego.

Ponieważ klasa `LocationManager` operuje na dostawcach, posiada interfejsy API implementujące tych dostawców. Na przykład można uzyskać wszystkich dostawców poprzez wywołanie metody `getAllProviders()`. Wybranego dostawcę uzyskujemy poprzez wywołanie metody `getProvider()` oraz przekazanie nazwy dostawcy w formie argumentu (na przykład `LocationManager.GPS_PROVIDER`). W tym przypadku musimy pamiętać, że metoda `getAllProviders()` będzie przekazywała również dostawców, do których nie mamy dostępu lub którzy są aktualnie nieaktywni. Na szczęście możemy określić stan dostawców za pomocą innych metod, na przykład `isProviderEnabled(String providerName)` lub `getProviders(boolean enabledOnly)`, w przypadku których wywołanie ich z wartością `true` spowoduje pobranie wyłącznie dostępnych dostawców.

Istnieje jeszcze inny sposób pobrania odpowiedniego dostawcy, mianowicie za pomocą metody `getProviders(Criteria criteria, boolean enabledOnly)` klasy `LocationManager`. Po przez określenie kryteriów aktualizacji położenia oraz wprowadzenie wartości `true` do atrybutu `enabledOnly` (dzięki czemu będą wynajdywani aktywni i przygotowani do działania dostawcy), otrzymamy listę z nazwami dostawców, lecz bez żadnych informacji na temat ich rodzajów. Takie rozwiązanie czasami bardziej się przydaje, ponieważ urządzenie może posiadać własnego dostawcę położenia, który spełnia nasze wymogi bez konieczności jego dokładnej znajomości. Obiekt `Criteria` może zawierać takie parametry, jak poziom dokładności, a także żądania informacji o szybkości, pelengu, wysokości, kosztach oraz zapotrzebowaniu energetycznym. Jeżeli żaden dostawca nie będzie spełniał naszych kryteriów, otrzymamy pustą listę, po czym będziemy mogli zrezygnować lub złagodzić nieco wymagania i spróbować ponownie.

## Jak uaktywnić dostawcę położenia?

Można by pomyśleć, że istnieje prosty interfejs API, pozwalający na aktywację dostawcy położenia (na przykład systemu GPS), w przypadku gdyby nie był aktywny, a aplikacja wymagałaby danych. Niestety, problem jest związany z czymś innym. Aby włączyć usługę lokacyjną, użytkownik musi to zrobić z poziomu aplikacji *Ustawienia*, dostępnej w urządzeniu. Aplikacja może znacznie ułatwić to zadanie, jeżeli pozwoli użytkownikowi wyświetlić okno ustawień bez konieczności jej ukrywania. Okno ustawień lokalizacyjnych jest w rzeczywistości aktywnością, która odpowiada na intencje. Jedyne zatem, co nasza aplikacja musi zrobić, to wywołać tę aktywność za pomocą właściwej intencji. Wymagany do tego kod może wyglądać następująco:

```
startActivityForResult(new Intent(  
    android.provider.Settings.ACTION_LOCATION_SOURCE_SETTINGS), 0);
```

Pamiętajmy, że w celu obsługi odpowiedzi musimy zaimplementować metodę zwrotną `onActivityResult()` wewnątrz naszej aktywności (zagadnienie to zostało omówione w rozdziale 5.). Nie zapominajmy również o tym, że chociaż pokładamy nadzieję w użytkowniku, iż uruchomimy jakiegoś dostawcę, na przykład system GPS, wcale nie musi tego zrobić. Nasza aplikacja będzie musiała sprawdzić, czy użytkownik uaktywnił dostawcę, a następnie w odpowiedzi na wynik podjąć jakieś działanie.

## Co możemy zrobić z położeniem?

Jak już wspomnieliśmy wcześniej, obiekty `Location` mogą nam podawać informacje o długości i szerokości geograficznej, godzinę przeprowadzenia obliczeń, nazwę dostawcy obliczającego współrzędne, a także, opcjonalnie, wysokość, szybkość, peleng oraz poziom dokładności. Mogą się w nich również znaleźć inne informacje, w zależności od dostawcy. Jeżeli na przykład obiekt `Location` pochodzi od dostawcy GPS, otrzymamy dane typu `Bundle`, określające liczbę sateli-

tów, które wzięto pod uwagę podczas obliczenia współrzędnych urządzenia. Takie dodatkowe informacje mogą, ale nie muszą być dostępne. Aby się dowiedzieć, czy obiekt klasy `Location` posiada jedną z takich wartości, w klasie tej należy ustawić metody typu `has...()`, które przekazują wartość logiczną. Przykładem może być metoda `hasAccuracy()`. Przed otrzymaniem wartości metody `getAccuracy()` rozsądnie byłoby najpierw wywołać metodę `hasAccuracy()`.

Klasa `Location` posiada również inne przydatne metody, na przykład metodę statyczną `distanceBetween()`, dzięki której możemy otrzymać najkrótszą odległość pomiędzy dwoma obiektami `Location`. Inną metodą związaną z odlegością jest `distanceTo()`, która będzie przekazywała najmniejszą wartość dystansu pomiędzy bieżącym obiektem `Location` a obiektem `Location` przekazanym metodzie. Zwróćmy uwagę, że odległość jest mierzona w metrach oraz że pod uwagę brana jest krzywizna Ziemi. Musimy też jednak mieć świadomość, że odległości podawane w wyniku działania omawianych metod nie są mierzone na przykład z perspektywy jazdy samochodem.

Jeżeli potrzebujemy wskazówek dla kierowców lub dystansu między dwoma punktami mierzonego jak dla jazdy samochodem, będą nam potrzebne początkowy i końcowy obiekt `Location`, jednak do przeprowadzenia wszystkich obliczeń prawdopodobnie będziemy musieli wykorzystać usługi Google Maps JavaScript API. Przydatny może się okazać na przykład interfejs Google Directions, podobny nieco do omówionego w rozdziale 11. interfejsu Google Translate. Umożliwia on aplikacji ukazanie całej trasy przejazdu — od punktu początkowego do punktu końcowego.

## Wysyłanie aktualizacji położenia do aplikacji podczas jej tworzenia

W trakcie testowania aplikacji usługa `LocationManager` wymaga informacji o położeniu geograficznym, a emulator nie ma dostępu do systemu GPS ani do wież operatorów sieci komórkowych. Aby przetestować usługę `LocationManager` w emulatorze, będziemy ręcznie przesyłali aktualizacje danych o położeniu z poziomu środowiska Eclipse. Na listingu 17.9 został zaprezentowany prosty przykład ilustrujący cele, jakie sobie wyznaczyliśmy.

### **Listing 17.9.** Rejestrowanie aktualizacji lokacji

---

```
package com.androidbook.location.update;

import android.app.Activity;
import android.content.Context;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.widget.Toast;

public class LocationUpdateDemoActivity extends Activity
{
    LocationManager locMgr = null;
    LocationListener locListener = null;

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        locMgr = (LocationManager)
```

```
getSystemService(Context.LOCATION_SERVICE);

    locListener = new LocationListener()
    {

        public void onLocationChanged(Location location)
        {
            if (location != null)
            {
                Toast.makeText(getApplicationContext(),
                    "Nowa lokacja: szerokość [" +
                location.getLatitude() +
                    "] długość [" + location.getLongitude()+"]",
                    Toast.LENGTH_SHORT).show();
            }
        }

        public void onProviderDisabled(String provider)
        {
        }

        public void onProviderEnabled(String provider)
        {
        }

        public void onStatusChanged(String provider,
        int status, Bundle extras)
        {
        }
    };

    @Override
    public void onResume() {
        super.onResume();

        locMgr.requestLocationUpdates(
            LocationManager.GPS_PROVIDER,
            0, // minTime w milisekundach
            0, // minDistance w metrach
            locListener);
    }

    @Override
    public void onPause() {
        super.onPause();
        locMgr.removeUpdates(locListener);
    }
}
```

---

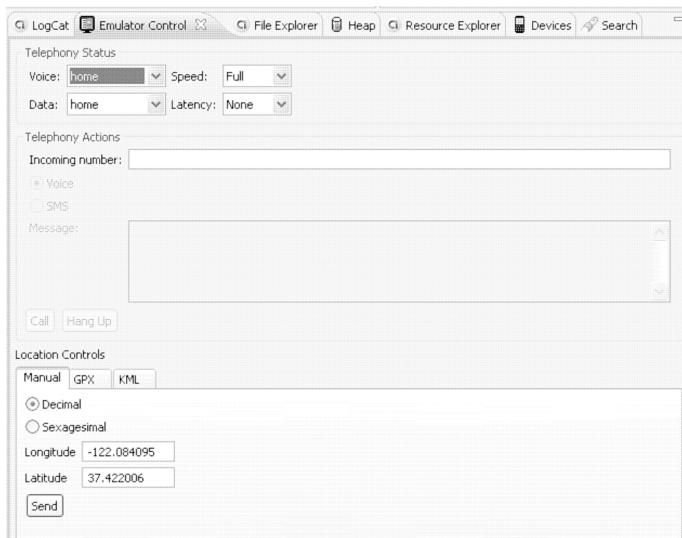
Nie zamieściliśmy tutaj kodu interfejsu użytkownika. Standardowy początkowy układ graficzny powinien nam całkowicie wystarczyć. Z tego samego powodu nie rozszerzamy klasy **MapActivity**, gdyż nie będziemy wyświetlać żadnej mapy.

Jednym z głównych zastosowań usługi *LocationManager* jest uzyskiwanie powiadomień o położeniu geograficznym urządzenia. Na listingu 17.9 pokazaliśmy, w jaki sposób można zarejestrować obiekt nasłuchujący odbierający zdarzenia aktualniania pozycji. W celu zarejestrowania obiektu nasłuchującego wywołujemy metodę `requestLocationUpdates()` i przekazujemy jej nazwę dostawcy jako parametr. Podczas zmiany położenia geograficznego usługa *LocationManager* wywołuje metodę `onLocationChanged()` obiektu nasłuchującego wraz z danymi o nowej lokalizacji. Bardzo ważne jest, aby usunąć we właściwym czasie wszystkie rejestracje aktualizacji położenia. W naszym przykładzie rejestrujemy metodę `onResume()` i usuwamy tę rejestrację w metodzie `onPause()`. Jeżeli nie będziemy mieli zamiaru korzystać z aktualizacji położenia, powinniśmy powiadomić dostawcę, żeby ich nie przesyłał. Istnieje również prawdopodobieństwo, że aktywność zostanie zakończona (na przykład w przypadku obrócenia urządzenia i ponownego uruchomienia aktywności), co oznacza, że wcześniejsza aktywność może ciągle istnieć, otrzymywać aktualizacje, wyświetlać je w obiekcie *Toast* oraz zajmować pamięć.

W naszym przykładzie ustalamy wartości minimalnego czasu oraz minimalnego dystansu na 0. W ten sposób usługa *LocationManager* będzie przesyłała informacje jak najczęściej. W rzeczywistej aplikacji nie byłoby to zalecane ustawienia, ale dzięki nim wersje testowe będą działały lepiej (jest niewskazane, aby fizyczne urządzenie zbyt często sprawdzało bieżące położenie geograficzne, ponieważ czynność ta wyczerpuje baterię). Należy te parametry ustawić w sposób odpowiedni do sytuacji, starając się zminimalizować liczbę powiadomień o zmianie położenia geograficznego.

Na listingu 17.9 zostało zaprezentowane nowe narzędzie — widżet *Toast*. Jest to przydatna aplikacja, pozwalająca na krótki czas wyświetlić mały widok użytkownikowi. Zaśnia on na moment widok bieżący, a następnie sam znika. Jego czas trwania można wydłużyć, stosując zamiast argumentu `LENGTH_SHORT` argument `LENGTH_LONG`.

Żeby przetestować nasz przykład na emulatorze, możemy skorzystać z umieszczonego we wtyczce ADT interfejsu DDMS (ang. *Dalvik Debug Monitor Service* — usługa monitora sprawdzania błędów Dalvik). Interfejs DDMS posiada ekran, za pomocą którego możliwe jest emulowanie zmieniającej się lokalizacji (rysunek 17.8).



**Rysunek 17.8.** Zastosowanie interfejsu DDMS w środowisku Eclipse do przesyłania emulatorowi danych o położeniu geograficznym

Aby otworzyć interfejs DDMS w środowisku Eclipse, klikamy opcje *Window/Open Perspective/DDMS*. Powinniśmy już wcześniej mieć do dyspozycji widok *Emulator Control*, jeżeli jednak nie jest widoczny w używanej perspektywie, przechodzimy do zakładki *Window*, a następnie klikamy *Show View/Other/Android/Emulator Control*. Być może będziemy musieli nieco przewinąć w dół ten panel, aby uzyskać dostęp do kontrolek lokalizacji. Jak widać na rysunku 17.8, w zakładce *Manual* interfejsu DDMS można wysyłać informacje GPS dotyczące nowego położenia geograficznego (parę wartości: szerokość i długość geograficzną) do emulatora. Wysłanie współrzędnych nowej lokalizacji spowoduje uruchomienie metody *onLocationChanged()* w obiekcie nasłuchującym, dzięki czemu użytkownik zostanie powiadomiony o zmianie położenia geograficznego.

Istnieje możliwość przesłania emulatorowi danych o nowej lokalizacji geograficznej za pomocą kilku innych technik, co widać w interfejsie DDMS (rysunek 17.8). Na przykład interfejs DDMS pozwala na przesyłanie pliku GPX (ang. *GPS Exchange Format* — format wymiany danych GPS) lub pliku KML (ang. *Keyhole Markup Language* — język znaczników formatu Key-hole). Przykładowe pliki GPX można pobrać z następujących adresów:

- [http://www.topografix.com/gpx\\_resources.asp](http://www.topografix.com/gpx_resources.asp),
- <http://trammer.co.nz/?view=gpxFiles>,
- <http://www.gpxchange.com/>.

W analogiczny sposób można wykorzystać poniższe zasoby do pobrania istniejących lub utworzenia nowych plików KML:

- <http://bbs.keyhole.com/>,
- [http://code.google.com/apis/kml/documentation/kml\\_tut.html](http://code.google.com/apis/kml/documentation/kml_tut.html).

#### Uwaga!

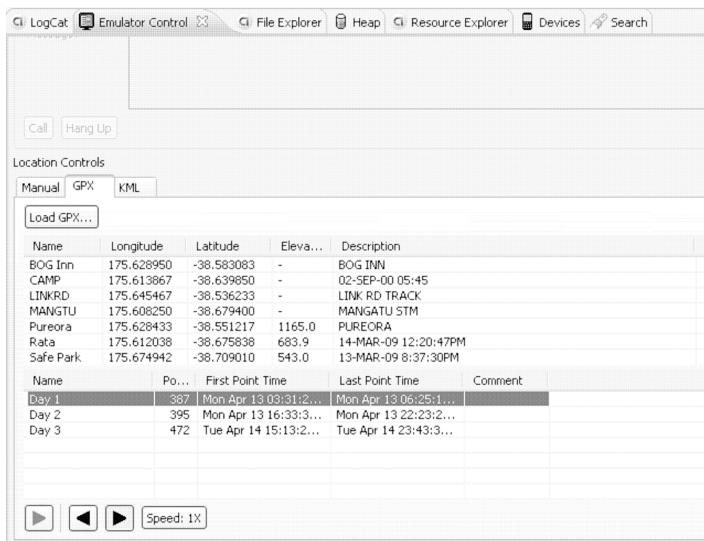
Na niektórych stronach umieszczone są pliki KMZ. Są to skompresowane pliki KML, zatem wystarczy je rozpakować, aby uzyskać pliki KML. Niektóre pliki KML wymagają zdefiniowania ich wartości przestrzeni nazw XML w celu uruchomienia ich w interfejsie DDMS. Jeżeli nie można uruchomić danego pliku KML, należy upewnić się, że posiada wpis `<kml xmlns="http://earth.google.com/kml/2.x">`.

Można wczytać plik GPX lub KML do emulatora oraz skonfigurować szybkość, z jaką będzie on odtwarzał te pliki (rysunek 17.9). W efekcie emulator będzie przesyłał aktualizacje lokalizacji do aplikacji, bazując na ustalonej szybkości symulowanego poruszania się. Jak widać na rysunku 17.9, plik GPX posiada w górnej części ekranu punkty, a w dolnej części ścieżki. Nie można odtwarzać punktów, jednak po kliknięciu jednego z nich zostanie on wysłany do emulatora. Po kliknięciu ścieżki stanie się dostępny przycisk *Play*, służący do odtwarzania punktów.

#### Uwaga!

Doszły do nas informacje, że nie wszystkie pliki GPX są odczytywane przez emulator. Jeżeli po wczytaniu pliku GPX nic się nie dzieje, należy spróbować wykorzystać inny plik z innego źródła.

Na listingu 17.9 znajduje się jeszcze kilka metod klasy *LocationManager*, o których do tej pory nie wspomnialiśmy. Są to metody zwrotne *onProviderDisabled()*, *onProviderEnabled()* oraz *onStatusChanged()*. W naszym przykładzie nie wykorzystaliśmy ich potencjału, jednak w prawdziwej aplikacji będziemy za ich pomocą powiadamiani o dostępności dostawcy położenia, na przykładzie systemu GPS, dla użytkownika lub o zmianach stanu dostawcy w czasie.



**Rysunek 17.9.** Wczytywanie plików GPX oraz KML do emulatora w celu odtworzenia danych

Wartości stanów dostawcy są następujące: OUT\_OF\_SERVICE, TEMPORARILY\_UNAVAILABLE i AVAILABLE. Nawet jeśli dostawca jest aktywny, nie oznacza to wcale, że będzie wysyłał aktualizacje położenia — dowiemy się tego właśnie za pomocą informacji o stanie. Zwróćmy uwagę, że metoda `onProviderDisabled()` zostanie wywołana natychmiast po wywoaniu metody `requestLocationUpdates()` wobec dostawcy położenia.

## Wysyłanie aktualizacji położenia z poziomu konsoli emulatora

Środowisko Eclipse zawiera łatwe w użyciu narzędzia, pozwalające na przesyłanie aktualizacji położenia do aplikacji, istnieje jednak jeszcze inny sposób. Przypomnijmy sobie informacje z rozdziału 2., że wykorzystujemy następujące polecenie w oknie narzędzi do uruchomienia konsoli emulatora:

```
telnet localhost emulator_numer_portu
```

gdzie `emulator_numer_portu` jest wartością powiązaną z wystąpieniem uruchomionego urządzenia AVD, która jest podana w pasku tytułowym emulatora. Po podłączeniu się do konsoli możemy wprowadzić polecenie `geo fix`, aby wysłać aktualizację położenia. W tym celu stosujemy następującą składnię (wysokość jest opcjonalna):

```
geo fix dlugosc szerokosc [ wysokosc ]
```

Na przykład poniższe polecenie spowoduje przesłanie współrzędnych geograficznych Jacksonville na Florydzie na wysokość 120 metrów nad poziomem morza:

```
geo fix -81.5625 30.334954 120
```

Powinniśmy zwracać szczególną uwagę na kolejność wprowadzania argumentów w poleceniu `geo fix`. Pierwszym argumentem jest długość, a szerokość — drugim.

## Alternatywne metody uzyskiwania aktualizacji położenia

Zaprezentowaliśmy wcześniej sposób uzyskania aktualizacji danych o położeniu, przesyłanej do aktywności za pomocą metody `requestLocationUpdates()` klasy `LocationManager`. W rzeczywistości istnieje kilka różnych sygnatur tej metody, włącznie z wersją wykorzystującą oczekujące intencje. Możemy w ten sposób kierować aktualizacje położenia do usług lub odbiorców komunikatów. Istnieje także możliwość przesyłania tych aktualizacji do wątków pobocznych zamiast do wątku głównego, dzięki czemu nasza aplikacja zyskuje na elastyczności. Należy tylko wspomnieć, że niektóre z tych metod są dostępne dopiero od wersji 2.3 Androida.

## Wyświetlanie informacji o położeniu za pomocą klasy `MyLocationOverlay`

Powszechnym zastosowaniem systemu GPS i map jest wskazywanie użytkownikowi jego położenia geograficznego. W Androidzie można tego łatwo dokonać poprzez zastosowanie specjalnej nakładki, noszącej nazwę `MyLocationOverlay`. Jeżeli dodamy tę nakładkę do widoku `MapView`, w prosty sposób będziemy mogli umieścić niebieską, migoczącą kropkę, która będzie wskazywała aktualne miejsce przebywania użytkownika, wyznaczone przez usługę `LocationManager`.

W naszym przykładowym projekcie połączymy kilka koncepcji w jednej aplikacji. Korzystając z listingu 17.10, możemy zmodyfikować poprzedni przykład poprzez aktualizację plików `main.xml` oraz `MyLocationDemoActivity.java`. Możemy ewentualnie utworzyć nowy projekt z istniejących kodów źródłowych, przygotowanych na potrzeby niniejszego rozdziału. Nie zapomnijmy umieścić klucza interfejsu map w pliku układu graficznego.

---

**Listing 17.10.** Zastosowanie klasy `MyLocationOverlay`

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <com.google.android.maps.MapView
        android:id="@+id/geoMap" android:clickable="true"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:apiKey="UMIESZCZAMY TUTAJ KLUCZ INTERFEJSU API MAPY"
    />

</RelativeLayout>

package com.androidbook.location.myoverlay;

import android.os.Bundle;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;
import com.google.android.maps.MyLocationOverlay;

public class MyLocationDemoActivity extends MapActivity {
```

```
MapView mapView = null;
MapController mapController = null;
MyLocationOverlay whereAmI = null;

@Override
protected boolean isLocationDisplayed() {
    return whereAmI.isMyLocationEnabled();
}

@Override
protected boolean isRouteDisplayed() {
    return false;
}

/** Wywoływane podczas pierwszego utworzenia aktywności. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    mapView = (MapView)findViewById(R.id.geoMap);
    mapView.setBuiltInZoomControls(true);

    mapController = mapView.getController();
    mapController.setZoom(15);

    whereAmI = new MyLocationOverlay(this, mapView);
    mapView.getOverlays().add(whereAmI);
    mapView.postInvalidate();
}

@Override
public void onResume()
{
    super.onResume();
    whereAmI.enableMyLocation();
    whereAmI.runOnFirstFix(new Runnable() {
        public void run() {
            mapController.setCenter(whereAmI.getLocation());
        }
    });
}

@Override
public void onPause()
{
    super.onPause();
    whereAmI.disableMyLocation();
}
}
```

Zwróćmy uwagę, że w tym przykładzie metoda `isLocationDisplayed()` będzie przekazywała wartość `true`, w przypadku gdy na mapie będzie wyświetlane położenie geograficzne urządzenia.

Po uruchomieniu aplikacji na emulatorze musimy zacząć wysyłać aktualizacje położenia geograficznego — dopiero wtedy nasz przykładowy projekt stanie się naprawdę interesujący. W tym celu należy otworzyć okno *Emulator Control* interfejsu DDMS, które omówiliśmy kilka stron wcześniej:

1. Znajdź w internecie przykładowy plik GPX. Wymienione wcześniej witryny zawierają ich olbrzymią liczbę. Wybierz jeden plik i skopiuj go na swój komputer.
2. Wczytaj pobrany plik w interfejsie DDMS, korzystając z przycisku *Load GPX...* w zakładce *GPX* interfejsu DDMS.
3. Zaznacz ścieżkę z listy na dole i klikaj przycisk odtwarzania (zielony trójkąt). Warto również pamiętać o przycisku *Speed*. Powinien rozpocząć się proces przesyłania do emulatora strumienia aktualizacji położenia geograficznego, które zostaną przekazane aplikacji.
4. Wciśnięcie przycisku *Speed* spowoduje częstszy przebieg aktualizacji.

Rysunek 17.10 pokazuje nam, jak może wyglądać ekran wynikowy.



Rysunek 17.10. Wyświetlanie bieżącego położenia za pomocą klasy MyLocationOverlay

Powyższy kod jest bardzo prosty. Po skonfigurowaniu podstawowych parametrów widoku *MapView*, ustawnieniu kontrolek zmiany rozmiaru mapy i wybraniu odpowiedniej skali mapy tworzymy nakładkę *MyLocationOverlay*. Dodajemy nową nakładkę do widoku *MapView*, a następnie wywołujemy metodę *postInvalidate()* w tym widoku, aby ta warstwa została nanieciona na mapę. Gdyby ta ostatnia metoda nie została umieszczona, nakładka zostałaaby utworzona, ale nie byłaby wyświetlana.

Pamiętajmy, że w naszej aplikacji metoda `onResume()` będzie wywoływana nawet podczas jej uruchamiania, jak również po wyjściu ze stanu wstrzymania. Trzeba zatem powiązać nawiązanie lokalizacji z metodą `onResume()`, a wyłączyć je po wywołaniu metody `onPause()`. Nie ma sensu marnowanie zapasu energii w baterii na żądania ustalenia położenia geograficznego, jeżeli nie będą potrzebne. Po włączeniu śledzenia położenia geograficznego podczas wywołania metody `onResume()` chcemy od razu uzyskać dane o aktualnym położeniu geograficznym. Klasa `myLocationOverlay` posiada pomocną w tym przypadku klasę — `runOnFirstFix()`. Dzięki tej metodzie możemy skonfigurować kod, który zostanie uruchomiony tuż po otrzymaniu współrzędnych geograficznych położenia urządzenia. Może to nastąpić natychmiast, ponieważ posiadamy współrzędne ostatniego znanego położenia, albo nieco później, po otrzymaniu informacji od dostawcy `GPS_PROVIDER`, `NETWORK_PROVIDER` lub `PASSIVE_PROVIDER`. Gdy uzyskamy odpowiednie dane, mapa zostanie zgodnie z nimi wyśrodkowana. Nie musimy robić nic więcej, ponieważ klasa `MyLocationOverlay` będzie uzyskiwała aktualnienia położenia geograficznego i będzie w tych miejscach umieszczała migoczącą, niebieską kropkę. Jeżeli kropka ta zbliży się zanadto do krawędzi mapy, mapa automatycznie zostanie wyśrodkowana w taki sposób, że migoczący punkt znajdzie się w centrum ekranu.

## Dostosowywanie klasy `MyLocationOverlay`

Mogliśmy zauważyc, że podczas aktualizowania położenia geograficznego możemy przybliżać i oddalać widok, a nawet przesuwać widok mapy. W zależności od punktu widzenia może to być zaletą lub wadą. Jeżeli przesuniemy mapę i nie będziemy pamiętać swej lokalizacji, może być trudno odnaleźć niebieską kropkę, chyba że zwiększymy skalę mapy w stopniu umożliwiającym wyświetlenie tego punktu. Automatyczne wyśrodkowanie mapy działa jedynie wtedy, gdy niebieska kropka stopniowo i bez naszego udziału zbliża się do krawędzi ekranu. Jeżeli samodzielnie przesuniemy widok tak bardzo, że nasz punkt zniknie, nie zostanie już automatycznie wyśrodkowany. Taka sytuacja może również nastąpić, jeżeli kropka znajdzie się poza krawędzią ekranu bez uprzedniego zbliżenia się do niej.

Aby bieżąca lokalizacja była zawsze wyświetlana w pobliżu środka ekranu, musi być cały czas animowana, co jest względnie prostym zadaniem. W następnej wersji naszego ćwiczenia wykorzystamy uprzednio utworzone elementy, wprowadzimy jednak niewielką zmianę w naszej aktywności, a także dodamy nową klasę do pakietu — rozszerzenie klasy `MyLocationOverlay`, dzięki czemu nieco usprawnimy działanie aplikacji. Nowe rozszerzenie klasy `MyLocationOverlay` zostało przedstawione na listingu 17.11.

**Listing 17.11.** Zastosowanie klasy `MyLocationOverlay` oraz centrowanie bieżącej lokalizacji

```
package com.androidbook.location.myoverlay;

import android.content.Context;
import android.location.Location;

import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapView;
import com.google.android.maps.MyLocationOverlay;

public class MyCustomLocationOverlay extends MyLocationOverlay {
    MapView mMapView = null;

    public MyCustomLocationOverlay(Context ctx, MapView mapView) {
        super(ctx, mapView);
```

```
    mMapView = mapView;
}

public void onLocationChanged(Location loc) {
    super.onLocationChanged(loc);
    GeoPoint newPt = new GeoPoint((int) (loc.getLatitude()*1E6),
        (int) (loc.getLongitude()*1E6));
    mMapView.getController().animateTo(newPt);
}
}
```

---

Jedyna zmiana w porównaniu z listingiem 17.10 polega na wykorzystaniu klasy `MyCustomLocationOverlay`, a nie `MyLocationOverlay` w metodzie `onCreate()`:

```
whereAmI = new MyCustomLocationOverlay(this, mapView);
```

Teraz można włączyć tę aplikację na emulatorze, a następnie przesyłać dane o nowej lokalizacji poprzez okno *Emulator Control*. Jeżeli wysyłamy strumień aktualizacji za pomocą pliku GPX, stwierdzimy, że niebieski punkt zawsze jest przesuwany na środek ekranu. Nawet jeśli całkowicie oddalimy widok, mapa zostanie wyśrodkowana na tym punkcie.

## Stosowanie alertów odległościowych

Wspomnialiśmy wcześniej, że klasa `LocationManager` może powiadomić użytkownika o znalezieniu się urządzenia w określonym obszarze geograficznym. Metodą służącą do implementacji tego mechanizmu jest `addProximityAlert()`, będącą częścią klasy `LocationManager`. W ogólnym założeniu klasa `LocationManager` uruchamia intencję w momencie przekroczenia okręgu o określonym promieniu, którego środek znajduje się w punkcie wyznaczonym przez współrzędne geograficzne. Intencja ta może wywołać usługę lub odbiorcę komunikatów, ewentualnie uruchomić aktywność. Opcjonalnie można również zdefiniować limit czasowy nakładany na ten alert, który może zostać przekroczony jeszcze przed uruchomieniem intencji.

Sam kod tej metody rejestruje obiekty nasłuchujące zarówno dla dostawców GPS, jak i sieciowych, a także ustanawia czas aktualizacji co jedną sekundę i minimalną odległość równą jednemu metrowi. Nie ma możliwości przesłonięcia tego zachowania ani modyfikowania parametrów. Zatem jeśli pozostawimy ten mechanizm włączony przez długi czas, baterie urządzenia zostaną bardzo szybko wyczerpane. Jeżeli urządzenie przejdzie w stan oczekiwania, alert odległościowy będzie sprawdzał położenie zaledwie co cztery minuty, jednak także i w tym przypadku nie możemy zmienić tego parametru.

Byłoby o wiele lepiej, gdybyśmy mogli samodzielnie zdefiniować promień obszaru za pomocą omówionych wcześniej technik. Jeśli na przykład utworzymy listę interesujących nas lokalizacji, możemy sprawdzać odległość od bieżącego miejsca do każdego z wyznaczonych punktów. W zależności od dystansu możemy chcieć poczekać chwilę, zanim znowu sprawdzimy bieżące położenie. Jeżeli najbliższe interesujące nas miejsce znajduje się 100 kilometrów od nas, a my chcemy wiedzieć, że się zbliżamy, gdy znajdziemy się w odległości 300 metrów, to jest oczywiste, że nie musimy w danej chwili sprawdzać położenia co sekundę.

Jeżeli jednak Czytelnik zechce wykorzystać tę metodę, zaprezentujemy, jak należy to zrobić. Na listingu 17.12 zamieściliśmy kod Java naszej głównej aktywności oraz klasy `BroadcastReceiver`, odbierającej komunikaty.

**Listing 17.12.** Konfigurowanie alertu odległościowego za pomocą odbiorcy komunikatów

```
// Jest to plik ProximityActivity.java
package com.androidbook.location.proximity;

import android.app.Activity;
import android.app.PendingIntent;
import android.content.Intent;
import android.content.IntentFilter;
import android.location.LocationManager;
import android.net.Uri;
import android.os.Bundle;

public class ProximityActivity extends Activity {
    private final String PROX_ALERT =
        "com.androidbook.intent.action.PROXIMITY_ALERT";
    private ProximityReceiver proxReceiver = null;
    private LocationManager locMgr = null;
    PendingIntent pIntent1 = null;
    PendingIntent pIntent2 = null;

    /** Wywołane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        locMgr = (LocationManager)
            this.getSystemService(LOCATION_SERVICE);

        double lat = 30.334954; // Współrzędne Jacksonville, Floryda
        double lon = -81.5625;
        float radius = 5.0f * 1609.0f; // 5 mil × 1609 metrów na jedną milę

        String geo = "geo:"+lat+","+lon;

        Intent intent = new Intent(PROX_ALERT, Uri.parse(geo));
        intent.putExtra("message", "Jacksonville, Floryda");

        pIntent1 = PendingIntent.getBroadcast(getApplicationContext(), 0,
            intent, PendingIntent.FLAG_CANCEL_CURRENT);

        locMgr.addProximityAlert(lat, lon, radius, -1L, pIntent1);

        lat = 28.54; // Współrzędne Orlando, Floryda
        lon = -81.38;
        geo = "geo:"+lat+","+lon;

        intent = new Intent(PROX_ALERT, Uri.parse(geo));
        intent.putExtra("message", "Orlando, Floryda");

        pIntent2 = PendingIntent.getBroadcast(getApplicationContext(), 0,
            intent, PendingIntent.FLAG_CANCEL_CURRENT);

        locMgr.addProximityAlert(lat, lon, radius, -1L, pIntent2);

        proxReceiver = new ProximityReceiver();
```

```
IntentFilter iFilter = new IntentFilter(PROX_ALERT);
iFilter.addDataScheme("geo");

registerReceiver(proxReceiver, iFilter);

}

protected void onDestroy() {
    super.onDestroy();
    unregisterReceiver(proxReceiver);
    locMgr.removeProximityAlert(pIntent1);
    locMgr.removeProximityAlert(pIntent2);
}
}
```

---

```
// Jest to plik ProximityReceiver.java
package com.androidbook.location.proximity;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.location.LocationManager;
import android.os.Bundle;
import android.util.Log;

public class ProximityReceiver extends BroadcastReceiver {

    private static final String TAG = "ProximityReceiver";

    @Override
    public void onReceive(Context arg0, Intent intent) {
        Log.v(TAG, "Intencja otrzymana");
        if(intent.getData() != null)
            Log.v(TAG, intent.getData().toString());
        Bundle extras = intent.getExtras();
        if(extras != null) {
            Log.v(TAG, "Komunikat: " + extras.getString("message"));
            Log.v(TAG, "Wkraczamy? " +
                extras.getBoolean(LocationManager.KEY_PROXIMITY_ENTERING));
        }
    }
}
```

---

Ponieważ w rzeczywistości nie wyświetlamy żadnej pozycji na mapie, nie korzystamy ani z klasy `MapActivity`, ani z bibliotek interfejsu Google Map, ani nie określamy żadnego celu. Musimy jednak dodać w pliku manifeście uprawnienie `android.permission.ACCESS_FINE_LOCATION`, ponieważ klasa `LocationManager` będzie próbowała korzystać z dostawcy GPS. W takim samym stopniu będzie korzystała z dostawcy sieciowego, ale skoro wprowadzamy uprawnienie `ACCESS_FINE_LOCATION`, wymagania dotyczące uprawnień zostają spełnione. Rejestrujemy odbiorcę `BroadcastReceiver` w metodzie `onCreate()`, nie musimy więc go rejestrować w pliku manifeście. Gdybyśmy umieścili odbiorcę komunikatów w oddzielnej aplikacji, wtedy *zaistniałaby* potrzeba jego zdefiniowania w manifeście. Definicja przykładowego kodu z listingu 17.12 została zaprezentowana na listingu 17.13.

**Listing 17.13.** Fragment pliku AndroidManifest.xml dotyczący klasy BroadcastReceiver alertu odległościowego

---

```
<application ... >

    <receiver android:name=".ProximityReceiver">
        <intent-filter>
            <action android:name="com.androidbook.android.intent.PROXIMITY_ALERT" />
            <data android:scheme="geo" />
        </intent-filter>
    </receiver>
</application>
```

---

Funkcja alertu odległościowego w Androidzie działa poprzez otrzymywanie obiektu Pending Intent, współrzędnych geograficznych interesującego nas punktu, promienia (w metrach) obszaru wokół tego punktu oraz czasu wymaganego do wykonywania aktualizacji. Wszystkie te argumenty są przekazywane za pomocą metody addProximityAlert() klasy LocationManager. Obiekt PendingIntent zawiera intencję, która zostanie uruchomiona w momencie przekroczenia (nieważne, w którą stronę) okręgu otaczającego interesujące nas miejsce. W naszym przykładzie postanowiliśmy skorzystać z intencji nadającej komunikaty, zatem wywołaliśmy metodę getBroadcast() klasy PendingIntent, przekazaliśmy naszej aplikacji kontekst wraz z intencją zawierającą działanie alertu oraz identyfikator URI obiektu Location. Jeżeli urządzenie przekroczy w jakiś sposób interesujący nas obszar, intencja będzie nadawała komunikaty do wszystkich zarejestrowanych odbiorców.

Postanowiliśmy nie wprowadzać wartości przekroczenia czasu alertów, określając czas ich trwania za pomocą wartości -1L. Jeżeli chcemy zaimplementować przekroczenie czasu, wartość ta musi definiować w milisekundach okres, po którego minięciu klasa LocationManager zaprzestanie prób i usunie oczekującą intencję. Nie zostaniemy powiadomieni o usunięciu tej intencji.

W naszym przykładzie uzyskujemy odniesienie do klasy LocationManager, tworzymy pierwsze obiekty Intent oraz PendingIntent, a następnie wywołujemy metodę addProximityAlert() konfigurującą nasz pierwszy alert. Następnie, już po uruchomieniu intencji, klasa Location Manager doda jeszcze tylko (w danych typu extra) wartość logiczną określającą, czy wkraczamy do interesującego nas okręgu, czy też go opuszczamy. Nie przesyła bieżących współrzędnych geograficznych urządzenia ani współrzędnych użytkowych podczas wywołania metody addProximityAlert(). Zatem aby wiedzieć, w pobliżu której lokalizacji znalazło się urządzenie, musielibyśmy dodać pewne informacje do intencji, mianowicie współrzędne geograficzne interesującego nas miejsca. Dodatkowo, dla urozmaicenia, wprowadziliśmy również komunikat (w danych typu extra) stanowiący opis tego miejsca. W razie potrzeby możemy dodać współrzędne typu double.

Po dodaniu pierwszego alertu dokładnie w taki sam sposób konfigurujemy drugi alert. Na końcu rejestrujemy obiekt BroadcastReceiver odbierający intencje nadawane przez klasę LocationManager. Filtr IntentFilter stanowi działanie w przypadku obydwu alertów, natomiast schematem jest zmienna geo. Obydwa te obiekty są wymagane do odbierania komunikatów, ponieważ komunikaty zawierają dane; moglibyśmy je odbierać bez udziału schematu, gdyby nie posiadały danych. Ostatnią czynnością, jaką musimy wykonać, jest wyczyszczenie pamięci po wywołaniu metody onDestroy() poprzez wyrejestrowanie odbiorcy oraz usunięcie alarmów odległościowych z klasy LocationManager za pomocą zachowanych intencji oczekujących. Dlatego właśnie przechowujemy odniesienia do obiektów PendingIntent — aby móc później usunąć alerty.

Nasza klasa `ProximityReceiver` jest bardzo prosta. Po odebraniu nadawanego komunikatu wyszukuje informacje, które będzie mogły wyświetlić w oknie `LogCat`. To tutaj znajdziemy wszelkie dodatkowe dane, wstawiane przez klasę `LocationManager`, informujące nas, czy wkraczamy, czy wychodzimy z interesującego nas obszaru.

Po uruchomieniu tej aplikacji na emulatorze ujrzymy pusty ekran, zawierający jedynie pasik z nazwą programu. Możemy teraz zacząć wysyłać aktualizacje położenia — albo za pomocą interfejsu DDMS, albo polecenia `geo fix` w konsoli emulatora. Gdy prześlemy współrzędne przekraczające granice obserwowanych okręgów (na przykład krawędzie obszarów o promieniu 5 mil wokół miast Jacksonville lub Orlando), w oknie `LogCat` powinny zacząć się pojawiać komunikaty pochodzące od odbiorcy komunikatów. Na rysunku 17.11 pokazujemy, jak może wyglądać okno `LogCat` po uaktywnieniu odbiorców komunikatów.

```

Log
tag      Message
ActivityManager
dalvikvm
ddm-heap
Gst feature list request
Constructor: service = android.location.ILocationManager$Stub$Proxy@43d07da8
addProximityAlert: latitude = 30.334954, longitude = -81.5625, radius = 8045.0, expiration = 2000
setMinTime 1000
startNavigating
addProximityAlert: latitude = 28.54, longitude = -81.38, radius = 8045.0, expiration = 60000, int
Displaying Activity com.androidbook.location.proximity/ ProximityActivity 2154 ms (total 2164 ms)
GC freed 540 objects / 46120 bytes in 126ms
GC freed 1860 objects / 108208 bytes in 147ms
TFFF: 17381
ProximityReceiver
Intencja otrzymana
geo:28.54,-81.38
Komunikat: Orlando, Floryda
Wkraczany? true
    
```

Rysunek 17.11. Okno LogCat zawierające komunikaty pochodzące od odbiorców komunikatów

Ponieważ mamy tu do czynienia z nadawanymi komunikatami, nie możemy polegać na kolejności, w jakiej są otrzymywane. Jeżeli na przykład urządzenie znajduje się wewnątrz okręgu otaczającego Orlando i nagle przemieści się do okręgu wokół Jacksonville, użytkownik może otrzymać komunikat, że znajduje się w Jacksonville, jeszcze zanim pojawią się informacje o jego obecności w Orlando.

Ponieważ zajmujemy się tu obiektami typu `Location`, naszym identyfikatorem URI jest schemat `geo`, który jest jednym z lepiej znanych schematów i doskonale się nadaje do przekazywania informacji o współrzędnych geograficznych. Zwróćmy uwagę, że struktura tego identyfikatora zawiera najpierw szerokość, a dopiero później długość geograficzną, gdy jednak korzystamy z polecenia `geo fix`, kolejność występowania tych współrzędnych jest odwrotna. Może to spowodować spore problemy, jeżeli nie będziemy ostrożni. W efekcie łatwo stracić mnóstwo czasu na znalezienie źródła problemu, podczas gdy wystarczy jedynie zamienić wartości współrzędnych służących do aktualizowania informacji o położeniu. Zawsze możemy również skorzystać z plików GPX lub KML, w których możemy wybierać lokalizacje do testowania, nakładające się na utworzone przez nas obszary zainteresowania.

Nasza próbna aplikacja jest bardzo prosta. W rzeczywistym programie odbiorca komunikatów może wysyłać powiadomienia lub uruchamiać usługi. W przypadku aktywności lub usługi (nawet znajdującej się w innej aplikacji) zamiast nadawanego komunikatu możemy zamieścić oczekującą intencję. Również zamiast aplikacji możemy korzystać ze wspomnianej już usługi.

## Odrośniki

Poniżej prezentujemy przydatny odnośnik do informacji rozszerzających informacje zawarte w tym rozdziale:

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów stworzonych specjalnie na potrzeby książki. Projekty dotyczące tego rozdziału zostały zawarte w katalogu *ProAndroid3\_R17\_Mapy*. W katalogu umieściliśmy także plik *Czytaj.TXT*, w którym zawarte są szczegółowe instrukcje dotyczące importowania projektów ze skompresowanych plików do środowiska Eclipse.

## Podsumowanie

W niniejszym rozdziale omówiliśmy usługi wykorzystujące mapy oraz dane o lokalizacji. Szczegółowo przedstawiliśmy zastosowanie kontrolki *MapView* oraz klasy *MapActivity*. Najpierw zajęliśmy się podstawowymi funkcjami mapy, a następnie pokazaliśmy, w jaki sposób można wykorzystać nakładki do umieszczania znaczników na mapie. Przeanalizowaliśmy nawet proces geokodowania i przetwarzania tego procesu w wątkach przeprowadzanych w tle. Podaliśmy informacje na temat klasy *LocationManager*, która uzyskuje szczegółowe informacje o położeniu geograficznym za pomocą dostawców oraz pozwala wyświetlić bieżące położenie urządzenia na mapie. Na koniec pokazaliśmy, w jaki sposób można wykorzystać alerty odległościowe.

W następnym rozdziale zajmiemy się usługami telefonicznymi w Androidzie.



# Używanie interfejsów telefonii

Wiele urządzeń obsługiwanych przez system Android to smartfony, dotychczas jednak nie zajmowaliśmy się kwestią pisania aplikacji wykorzystujących funkcje telefonii. W niniejszym rozdziale zapoznamy się ze sposobami wysyłania i odbierania wiadomości SMS. Poruszmy także temat kilku innych interesujących aspektów dotyczących interfejsów API telefonii w Androidzie, w tym protokołu SIP (ang. *Session Initiation Protocol* — protokół inicjalizacji sesji). Protokół SIP stanowi standard utworzony przez grupę IETF, służący do implementacji technologii VoIP (ang. *Voice over Internet Protocol* — protokół przesyłania głosu po internet), za pomocą której użytkownik może wykonywać poprzez sieć internetową połączenia przypominające telefoniczne. Za pomocą protokołu SIP można także przesyłać obraz wideo.

## Praca z wiadomościami SMS

Rozwinięciem skrótu SMS jest *Short Messaging Service*, czyli usługa wysyłania krótkich wiadomości tekstowych, powszechnie stosuje się jednak termin *wiadomości tekstowe*. Środowisko Android SDK pozwala na wysyłanie i odbieranie takich wiadomości. Przyjrzymy się najpierw różnym sposobom wysyłania wiadomości SMS za pomocą środowiska SDK.

## Wysyłanie wiadomości SMS

Aby wysłać wiadomość tekstową z poziomu aplikacji, musimy dodać uprawnienie `→android.permission.SEND_SMS` w pliku manifeście, a następnie skorzystać z klasy `android.telephony.SmsManager`. Na listingu 18.1 zamieściliśmy plik układu graficznego oraz kod Java przykładowej aplikacji. Jeżeli chcemy dowiedzieć się, gdzie należy wstawić uprawnienie w pliku manifeście, możemy zajrzeć do listingu 18.2.

### Uwaga!

Na końcu rozdziału zamieszczamy adres URL strony, z której możemy pobrać projekty utworzone w tym rozdziale. W ten sposób będziemy mogli bezpośrednio zaimportować cały kod źródłowy do środowiska Eclipse.

**Listing 18.1.** Wysyłanie wiadomości SMS (tekstowych)

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">

        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Adres docelowy:" />

        <EditText android:id="@+id/addrEditText"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:phoneNumber="true"
            android:text="9045551212" />

    </LinearLayout>

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">

        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Wiadomość:" />

        <EditText android:id="@+id/msgEditText"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Witaj, sms-ie" />

    </LinearLayout>

    <Button android:id="@+id/sendSmsBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Wyślij wiadomość"
        android:onClick="doSend"/>

</LinearLayout>

// Jest to plik TelephonyDemo.java
import android.app.Activity;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.view.View;
import android.widget.EditText;
```

```

import android.widget.Toast;

public class TelephonyDemo extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void doSend(View view) {
        EditText addrTxt =
            (EditText) findViewById(R.id.addrEditText);

        EditText msgTxt =
            (EditText) findViewById(R.id.msgEditText);

        try {
            sendSmsMessage(
                addrTxt.getText().toString(),msgTxt.getText().toString());
            Toast.makeText(this, "Wiadomość SMS wysłana",
                Toast.LENGTH_LONG).show();
        } catch (Exception e) {
            Toast.makeText(this, "Nie udało się wysłać wiad. SMS",
                Toast.LENGTH_LONG).show();
            e.printStackTrace();
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
    }

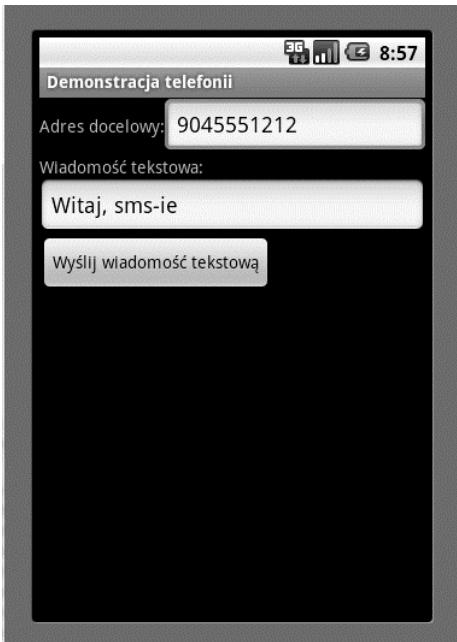
    private void sendSmsMessage(String address,String message) throws Exception
    {
        SmsManager smsMgr = SmsManager.getDefault();
        smsMgr.sendTextMessage(address, null, message, null, null);
    }
}

```

Na listingu 18.1 pokazano przykład wysyłania wiadomości SMS w środowisku SDK. Jeśli przyjrzymy się fragmentowi kodu układu graficznego, zauważymy dwa pola *EditText*: w jednym umieszczaćmy adres docelowy (numer telefonu) adresata, a w drugim wpisujemy treść wiadomości. Interfejs użytkownika jest także zaopatrzony w przycisk powodujący wysłanie wiadomości SMS, co zostało zilustrowane na rysunku 18.1.

Interesującą częścią kodu jest metoda *sendSmsMessage()*. Wykorzystuje ona metodę *sendTextMessage()* klasy *SmsManager* do wysyłania wiadomości SMS. Poniżej umieściliśmy sygnaturę metody *SmsManager.sendTextMessage()*:

```
sendTextMessage(String destinationAddress, String smscAddress, String textMsg,
PendingIntent sentIntent, PendingIntent deliveryIntent);
```



Rysunek 18.1. Przykładowy interfejs UI aplikacji służącej do wysyłania wiadomości SMS

W naszym przykładzie wypełniamy jedynie dane adresu docelowego oraz parametry wiadomości tekstowej. Można jednak dostosować metodę do własnych potrzeb, tak żeby nie korzystała z domyślnego centrum usługi SMS (adresu serwera sieci komórkowej przesyłającego wiadomość). Możemy także zaimplementować konfigurację pozwalającą na nadanie oczekujących intencji po wysłaniu (lub nieudanym wysłaniu) wiadomości i dostarczeniu powiadomienia.

Wysyłanie wiadomości SMS składa się z dwóch głównych etapów: wysłania wiadomości i jej dostarczenia. Jeżeli aplikacja to umożliwia, po osiągnięciu każdego etapu zostaje nadana intencja oczekująca. Możemy w niej umieścić cokolwiek zechcemy, na przykład jakieś działanie, ale kod wynikowy wysłany do odbiorcy komunikatów będzie niepowtarzalny dla etapu wysyłania i dostarczenia wiadomości SMS. Ponadto możemy otrzymywać dodatkowe dane związane z błędami transmisji lub raportami o stanie wiadomości, w zależności od sposobu implementacji wysyłania wiadomości.

Przy braku intencji oczekujących aplikacja nie może określić, czy wiadomość została pomyślnie wysłana. Dlatego musimy jak najwięcej testować. Jeżeli włączymy tę przykładową aplikację w emulatorze i uruchomimy kolejne wystąpienie emulatora (bez różnicy, czy z wiersza poleceń, czy za pomocą opcji *Window/Android SDK and AVD Manager* w środowisku Eclipse), możemy wykorzystać numer jego portu jako adres docelowy. Numer portu jest liczbą pojawiającą się w pasku tytułowym emulatora; zazwyczaj przybiera wartość 5554. Po kliknięciu przycisku *Wyślij wiadomość* w drugim emulטורze powinno się pojawić powiadomienie informujące, że została do niego dostarczona wiadomość.

Klasa `SmsManager` zawiera jeszcze dwie metody pozwalające na wysłanie wiadomości SMS:

- `sendDataMessage()` pobiera dodatkowy argument definiujący numer portu i zamiast ciągu znaków przesyła tablicę bajtów;

- `sendMultipartTextMessage()` umożliwia wysyłanie wiadomości tekstowej, gdy jest ona większa od dopuszczalnej wartości określonej w specyfikacji SMS. Metoda ta pobiera tablicę ciągu znaków, zwróciemy jednak uwagę, że jednocześnie może zostać pobrana opcjonalna tablica odpowiadających im intencji oczekujących, odpowiedzialnych za wysyłanie i dostarczanie wiadomości. Klasa `SmsManager` posiada metodę `divideMessage()`, ułatwiającą dzielenie dużych wiadomości na mniejsze części.

Podsumowując, obsługa wysyłania wiadomości SMS w Androidzie jest wyjątkowo prosta. Pamiętajmy, że emulator nie będzie w rzeczywistości wysyłał wiadomości tekstowych. Możemy jednak założyć, że implementacja przebiegała pomyślnie, jeśli metoda `sendTextMessage()` nie przekaże żadnego wyjątku. Jak widać na listingu 18.1, klasa `Toast` została wykorzystana do wyświetlenia w interfejsie użytkownika komunikatu o powodzeniu wysyłania wiadomości tekstowej.

Wysłanie wiadomości SMS jest zaledwie połową sukcesu. Zajmiemy się teraz monitorowaniem przychodzących wiadomości tekstowych.

## Monitorowanie przychodzących wiadomości tekstowych

Wykorzystamy utworzoną przed chwilą aplikację do wysyłania wiadomości SMS, a następnie zaimplementujemy odbiorcę `BroadcastReceiver` do nasłuchiwanego działania `android.permission.SMS_RECEIVED`. Działanie to jest nadawane przez Androida w momencie otrzymania przez urządzenie wiadomości SMS. Po zarejestrowaniu odbiorcy nasza aplikacja będzie powiadamiana o każdej przychodzącej wiadomości SMS. Pierwszym etapem monitorowania przychodzących wiadomości SMS jest żądanie uprawnień do ich otrzymywania. W tym celu musimy dodać uprawnienie `android.permission.RECEIVE_SMS` w pliku manifeście. Następnie zaimplementujemy monitor nasłuchujący. W celu zaimplementowania odbiorcy musimy napisać rozszerzenie klasy `android.content.BroadcastReceiver`, a następnie zarejestrować go w pliku manifeście. Na listingu 18.2 umieściliśmy zarówno plik `AndroidManifest.xml`, jak i klasę odbiorcy. Zwrócmy uwagę, że w manifeście są obecne obydwa uprawnienia, ponieważ musimy jeszcze wysłać uprawnienie do uprzednio utworzonej aktywności.

**Listing 18.2.** Monitorowanie wiadomości SMS

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.telephony" android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".TelephonyDemo"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name="MySMSMonitor">
            <intent-filter>
                <action
                    android:name="android.provider.Telephony.SMS_RECEIVED"/>
```

```
</intent-filter>
</receiver>

</application>
<uses-sdk android:minSdkVersion="4" />

<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>

</manifest>

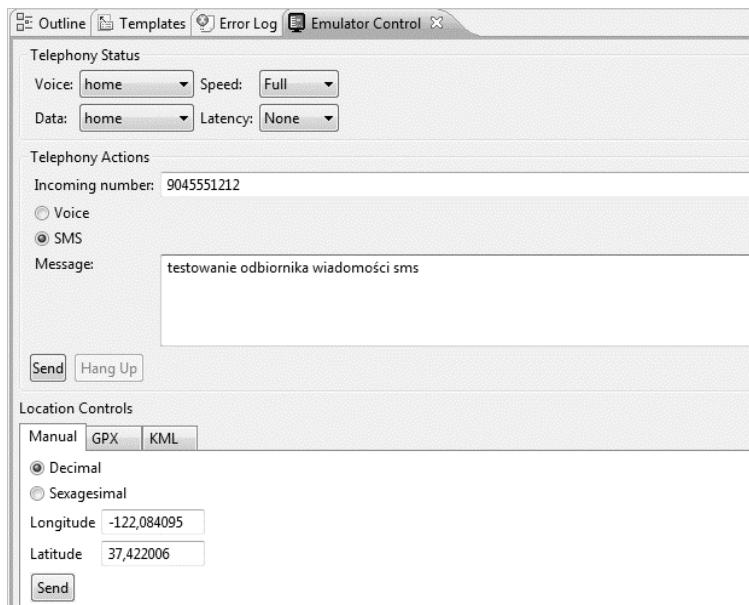
// Jeste to plik MySMSMonitor.java
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.telephony.SmsMessage;
import android.util.Log;

public class MySMSMonitor extends BroadcastReceiver
{
    private static final String ACTION =
            "android.provider.Telephony.SMS_RECEIVED";
    @Override
    public void onReceive(Context context, Intent intent)
    {
        if(intent!=null && intent.getAction()!=null &&
           ACTION.compareToIgnoreCase(intent.getAction())==0)
        {
            Object[] pduArray= (Object[]) intent.getExtras().get("pdus");
            SmsMessage[] messages = new SmsMessage[pduArray.length];
            for (int i = 0; i<pduArray.length; i++) {
                messages[i] = SmsMessage.createFromPdu(
                    (byte[])pduArray [i]);
                Log.d("MySMSMonitor", "Od: " +
                      messages[i].getOriginatingAddress());
                Log.d("MySMSMonitor", "Wiadomosc: " +
                      messages[i].getMessageBody());
            }
            Log.d("MySMSMonitor", "Wiadomosc SMS otrzymana.");
        }
    }
}
```

---

Pierwszą częścią listingu 18.2 jest definicja manifestu dla klasy `BroadcastReceiver` pozwalająca na odbieranie wiadomości SMS. Klasą monitorującą wiadomości SMS jest klasa `MySMSMonitor`. Zostaje w niej zaimplementowana abstrakcyjna metoda `onReceive()`, która jest wywoływana przez system podczas otrzymywania wiadomości SMS. Jednym ze sposobów przetestowania tej aplikacji jest skorzystanie z widoku Emulator Control w środowisku Eclipse. Uruchamiamy aplikację na emulatorze, a następnie przechodzimy do `Window>Show View/Other/Android/Emulator Control`. Interfejs użytkownika pozwala na przesyłanie do emulatora danych, które

imitują otrzymanie wiadomości SMS lub odebranie połączenia telefonicznego. Na rysunku 18.2 widać, że wiadomości SMS są wysyłane poprzez wypełnienie pola *Incoming number*, a następnie zaznaczenie opcji SMS. W dalszej kolejności wpisujemy treść wiadomości w polu *Message* i klikamy przycisk *Send*. W ten sposób zostaje wysłana wiadomość tekstowa do emulatora i wywołana metoda `onReceive()` klasy `BroadcastReceiver`.



**Rysunek 18.2.** Wykorzystanie interfejsu UI Emulator Control do wysyłania wiadomości SMS do emulatora

Metoda `onReceive()` posiada intencję służącą do transmisji danych, zawierającą obiekt `SmsMessage` wewnętrz właściwości `bundle`. Zawartość tego obiektu uzyskujemy poprzez wywołanie metody `intent.getStringExtra("pdus")`. Wywołanie to przekazuje tablicę obiektów zdefiniowanych w trybie PDU (ang. *Protocol Description Unit* — jednostka opisu protokołu) — standardzie przemysłowym, reprezentującym wiadomości SMS. Możemy następnie przekonwertować jednostki PDU na obiekty `SmsMessage` Androida, podobnie jak na listingu 18.2. Jak widać, jednostki PDU otrzymujemy z intencji w postaci tablicy obiektów. W końcu przeprowadzamy iteracje na tablicy PDU i tworzymy obiekty `SmsMessage` z jednostek PDU poprzez wywołanie metody `SmsMessage.createFromPdu()`. Czynności wykonywane po odczytaniu przychodzącej wiadomości są przeprowadzane bardzo szybko. Odbiorca komunikatów otrzymuje wysoki priorytet w systemie, lecz jego zadanie musi zostać szybko zakończone i nie można go wyświetlić na pierwszym planie do wglądu użytkownika. Z tego powodu nasze możliwości są dość ograniczone. Nie powinniśmy przeprowadzać bezpośrednio żadnych czynności na interfejsie użytkownika. Nie zaszkodzi wysłanie powiadomienia, ponieważ uruchamia usługę kontynuującą pracę. Po zakończeniu działania metody `onReceive()` jej główny proces może zostać zamknięty w dowolnym momencie. Można uruchomić usługę, ale nie należy wiązać jej z inną usługą, ponieważ może to oznaczać, że proces musi jeszcze przez chwilę istnieć, co nie zawsze jest możliwe. Szczegółowe informacje dotyczące odbiorców komunikatów znajdziemy w rozdziale 14.

Przyjrzyjmy się teraz, w jaki sposób możemy pracować z poszczególnymi folderami wiadomości SMS.

## Praca z folderami wiadomości SMS

Następnym powszechnym wymogiem jest uzyskanie dostępu do skrzynki odbiorczej wiadomości SMS. Rozpoczniemy od dodania uprawnienia odczytywania wiadomości SMS (`android.permission.READ_SMS`) w pliku manifeście. Po dodaniu tego uprawnienia będziemy mogli odczytać nadesłane wiadomości umieszczone w skrzynce odbiorczej.

Aby odczytywać wiadomości SMS, musimy przeprowadzić kwerendę wobec skrzynki odbiorczej, co zostało zaprezentowane na listingu 18.3.

**Listing 18.3.** Wyświetlanie wiadomości tekstowych dostępnych w skrzynce odbiorczej

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/sms_inbox.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView android:id="@+id/row"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"/>

</LinearLayout>

// Jest to plik SMSInboxDemo.java
import android.app.ListActivity;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

public class SMSInboxDemo extends ListActivity {

    private ListAdapter adapter;
    private static final Uri SMS_INBOX = Uri.parse("content://sms/inbox");

    @Override
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        Cursor c = getContentResolver()
            .query(SMS_INBOX, null, null, null, null);
        startManagingCursor(c);
        String[] columns = new String[] { "body" };
        int[] names = new int[] { R.id.row };
        adapter = new SimpleCursorAdapter(this, R.layout.sms_inbox, c, columns,
            names);
        setListAdapter(adapter);
    }
}
```

---

Kod z listingu 18.3 otwiera skrzynkę odbiorczą wiadomości SMS i tworzy listę elementów, z których każdy zawiera część treści wiadomości tekstowej. Fragment kodu dotyczący układu graficznego z listingu 18.3 zawiera prostą kontrolkę `TextView`, w której będzie przechowywana treść wiadomości każdego elementu listy. Aby uzyskać listę wiadomości SMS, tworzymy identyfikator URI wskazujący na skrzynkę wiadomości przychodzących (`content://sms/inbox`), a następnie wykonujemy prostą kwerendę. W kolejnym kroku filtrujemy treść wiadomości SMS i wyznaczamy adapter listy klasy `ListActivity`. Po wykonaniu kodu z listingu 18.3 ujrzymy listę wiadomości tekstowych, dostępnych w skrzynce odbiorczej. Zanim uruchomimy kod na emulatorze, utworzmy najpierw kilka wiadomości SMS za pomocą narzędzia Emulator Control.

Skoro potrafimy uzyskać dostęp do skrzynki odbiorczej wiadomości SMS, spodziewamy się, że będziemy mogli korzystać również z innych, powiązanych folderów, na przykład ze skrzynki nadawczej lub z folderu przechowującego wersje robocze wiadomości. Jedyną różnicą pomiędzy skrzynką odbiorczą a innymi folderami są ich identyfikatory URI. Na przykład możemy uzyskać dostęp do skrzynki nadawczej poprzez wykonanie kwerendy wobec adresu `content://sms/sent`. Poniżej znajduje się pełna lista folderów wiadomości SMS oraz definiujące je identyfikatory URI:

- **Wszystkie:** `content://sms/All`.
- **Odebrane:** `content://sms/inbox`.
- **Wysłane:** `content://sms/sent`.
- **Wersje robocze:** `content://sms/draft`.
- **Nieodebrane:** `content://sms/outbox`.
- **Niewysłane:** `content://sms/failed`.
- **Zakolejkowane:** `content://sms/queued`.
- **Niedostarczone:** `content://sms/undelivered`.
- **Konwersacje:** `content://sms/conversations`.

W Androidzie koncepcje wiadomości MMS i SMS są ze sobą połączone i można uzyskać dostęp jednocześnie do dostawców treści obydwu rodzajów za pomocą upoważnienia `mms-sms`. Zatem możemy korzystać z następującego identyfikatora URI:

`content://mms-sms/conversations`

## Wysyłanie wiadomości e-mail

Skoro wiemy już, w jaki sposób można wysyłać w Androidzie wiadomości SMS, możemy założyć, że za pomocą podobnych interfejsów API są wysyłane wiadomości e-mail. Niestety, Android nie został wyposażony w interfejsy API obsługujące wiadomości e-mail. Panuje powszechna opinia, że użytkownicy nie chcą aplikacji, która wysyłałyby wiadomości e-mail w ich imieniu bez ich wiedzy. Zamiast tego w celu wysłania wiadomości należy skorzystać z zarejestrowanego klienta pocztowego. Na przykład możemy wykorzystać działanie `ACTION_SEND` do uruchomienia takiej aplikacji, co zostało ukazane na listingu 18.4.

**Listing 18.4.** Uruchamianie aplikacji pocztowej za pomocą intencji

---

```
Intent emailIntent=new Intent(Intent.ACTION_SEND);

String subject = "Cześć!";
String body = "Pozdrowienia z Androida...";
```

```
String[] recipients = new String[]{"aaa@bbb.com"};
emailIntent.putExtra(Intent.EXTRA_EMAIL, recipients);

emailIntent.putExtra(Intent.EXTRA_SUBJECT, subject);
emailIntent.putExtra(Intent.EXTRA_TEXT, body);
emailIntent.setType("message/rfc822");

startActivity(emailIntent);
```

---

Powyższy kod uruchamia domyślną aplikację pocztową i pozwala zadecydować użytkownikowi o wysłaniu wiadomości e-mail. Dodatkowymi elementami, które można dodać do intencji wiadomości e-mail, są obiekty EXTRA\_CC i EXTRA\_BCC.

Załóżmy, że chcemy wysłać wiadomość e-mail wraz z załącznikiem. W tym celu musimy wprowadzić konstrukcję podobną do poniższej, gdzie za Uri wstawiamy odniesienie do załączanego pliku:

```
emailIntent.putExtra(Intent.EXTRA_STREAM,
Uri.fromFile(new File(myFileName)));
```

Przejdźmy teraz do menedżera telefonii.

## Praca z menedżerem telefonii

Wśród interfejsów API telefonii znajduje się także menedżer telefonii (`android.telephony.TelephonyManager`), dzięki któremu możemy uzyskać informacje o usługach telefonicznych dostępnych w urządzeniu i o subskrypcji, a także rejestrować zmiany stanu połączenia telefonicznego. Zastosowanie funkcji telefonicznych wymaga, aby aplikacje zachowywały się w odpowiedni sposób po wykryciu połączenia przychodzącego. Na przykład odtwarzacz muzyczny wstrzymuje działanie w trakcie połączenia przychodzącego i odtwarzanie zostaje wznowione po zakończeniu rozmowy. Najprostszym sposobem nasłuchiwania zmian stanu telefonu jest implementacja odbiorcy komunikatów wobec wartości `android.intent.action.PHONE_STATE`. Możemy wykorzystać w tym celu taki sam algorytm jak w przypadku omawianego wcześniej nasłuchiwanego przychodzących wiadomości SMS. Innym rozwiązaniem jest wprowadzenie klasy `TelephonyManager`. W tym podrozdziale pokażemy, w jaki sposób rejestrować zmiany stanu połączenia telefonicznego oraz jak wykrywać przychodzące połączenia telefoniczne. Szczegóły zostały przedstawione na listingu 18.4.

---

**Listing 18.4.** Zastosowanie menedżera telefonii

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<Button
    android:id="@+id/callBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Wykonaj połączenie"
```

```
    android:onClick="doClick"
  />
<TextView
  android:id="@+id/textView"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
</LinearLayout>

// Jest to plik PhoneCallActivity.java
package com.androidbook.phonecall.demo;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.telephony.PhoneStateListener;
import android.telephony.TelephonyManager;
import android.view.View;
import android.widget.TextView;

public class PhoneCallActivity extends Activity {
    private TelephonyManager teleMgr = null;
    private MyPhoneStateListener myListener = null;
    private String logText = "";
    private TextView tv;

    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        tv = (TextView) findViewById(R.id.textView);

        teleMgr =
            (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        myListener = new MyPhoneStateListener();
    }

    protected void onResume() {
        super.onResume();
        teleMgr.listen(myListener, PhoneStateListener.LISTEN_CALL_STATE);
    }

    protected void onPause() {
        super.onPause();
        teleMgr.listen(myListener, PhoneStateListener.LISTEN_NONE);
    }

    public void doClick(View target) {
        Intent intent = new Intent(Intent.ACTION_VIEW,
            Uri.parse("tel:5551212"));
    }
}
```

```
        startActivity(intent);
    }

    class MyPhoneStateListener extends PhoneStateListener
    {
        @Override
        public void onCallStateChanged(int state, String incomingNumber)
        {
            super.onCallStateChanged(state, incomingNumber);

            switch(state)
            {
                case TelephonyManager.CALL_STATE_IDLE:
                    logText = "wywolanie stanu spoczynku...przychodzacy numer to["+
                        incomingNumber + "]\n" + logText;
                    break;
                case TelephonyManager.CALL_STATE_RINGING:
                    logText = "wywolanie stanu dzwonienia...przychodzacy numer to["+
                        incomingNumber + "]\n" + logText;
                    break;
                case TelephonyManager.CALL_STATE_OFFHOOK:
                    logText = "wywolanie stanu Zajety...przychodzacy numer to["+
                        incomingNumber + "]\n" + logText;
                    break;
                default:
                    logText = "wywolanie stanu [" + state +
                        "]przychodzacy numer to[" +
                        incomingNumber + "]\n" + logText;
                    break;
            }
            tv.setText(logText);
        }
    }
}
```

Podczas pracy z menedżerem telefonii należy dodać uprawnienie `android.permission.READ_PHONE_STATE` do pliku manifestu, aby uzyskać dostęp do informacji na temat stanu telefonu. Na listingu 18.5 widać, że będziemy otrzymywać powiadomienia o zmianie stanu telefonu poprzez zaimplementowanie klasy `PhoneStateListener` i wywołanie metody `listen()` klasy `TelephonyManager`. Jeśli zostanie wykryte połączenie przychodzące lub stan telefonu ulegnie zmianie, system wywoła metodę `onCallStateChanged()` klasy `PhoneStateListener` wraz z nowym stanem. Gdy wyprobujemy działanie projektu, okaże się, że połączenia przychodzące są dostępne jedynie w stanie `CALL_STATE_RINGING`. W naszym przykładzie wyświetliśmy wiadomość na ekranie, jednak w tym miejscu można wstawić określone działanie aplikacji, na przykład wstrzymywanie odtwarzania muzyki lub wideo. Aby emulować przychodzące połączenia telefoniczne, możemy skorzystać z interfejsu UI Emulator Control — podobnie jak w przypadku wysyłania wiadomości SMS (rysunek 18.2), tym razem jednak zamiast opcji `SMS` wybieramy opcję `Voice`.

Zwróćmy uwagę, że poprzez metodę `onPause()` klasa `TelephonyManager` przerywa wysyłanie aktualizacji. Ważne jest, aby zawsze wyłączać wysyłanie komunikatów w stanie wstrzymania aktywności. W przeciwnym wypadku klasa `TelephonyManager` może utrzymywać odniesienie do naszego obiektu i uniemożliwić jego późniejsze usunięcie.

W powyższym przykładzie zajmujemy się tylko jednym ze stanów telefonów, które możemy nasłuchiwać. Warto przejrzeć dokumentację klasy PhoneStateListener, aby poznać inne stany, na przykład LISTEN\_MESSAGE\_WAITING\_INDICATOR. Podczas pracy ze zmianami stanu telefonu może być potrzebny również numer telefonu subskrybenta (użytkownika). Otrzymujemy go za pomocą metody TelephonyManager.getLine1Number().

Czytelnik być może zastanawia się, czy można odebrać połączenie za pomocą kodu. Niestety, na obecną chwilę zestaw Android SDK nie posiada takiej możliwości, chociaż w dokumentacji widnieje informacja, że można uruchomić intencję za pomocą działania ACTION\_ANSWER. W praktyce jednak rozwiązanie to nie działa, chociaż warto byłoby sprawdzić, czy problem ten nie został już rozwiązany.

Analogicznie, możemy chcieć umieścić połączenie wychodzące w kodzie. Tutaj na szczęście sprawy mają się prościej. Najłatwiejszym sposobem wprowadzenia połączenia wychodzącego jest przywołanie aplikacji Dialer za pomocą intencji, wykorzystując następujący kod:

```
Intent intent = new Intent(Intent.ACTION_CALL, Uri.parse("tel:5551212"));
startActivity(intent);
```

Pamiętajmy, że jeśli nasza aplikacja ma rzeczywiście wykonywać połączenia, będzie wymagała uprawnienia android.permission.CALL\_PHONE. W przeciwnym wypadku w momencie próby wywołania tej aplikacji aplikacja wyrzuci wyjątek zabezpieczeń. Aby móc nawiązywać połączenia bez konieczności korzystania z uprawnień, zmieniamy działanie intencji na Intent.ACTION\_VIEW, dzięki czemu aplikacja Dialer zostanie od razu wyświetlona z wprowadzonym numerem telefonu, użytkownik jednak będzie musiał wcisnąć przycisk *Wyślij*, aby zainicjować połączenie.

Podczas korzystania z funkcji telefonicznych należy jeszcze pamiętać, że inne aplikacje mogą bardzo aktywnie reagować na przychodzące połączenia i powodować wstrzymanie naszej aktywności. W takim przypadku przestaniemy otrzymywać powiadomienia, chociaż pojawi się komunikat dokładnie w momencie wywołania metody onResume() i klasa TelephonyManager zostanie ponownie zarejestrowana. Będźmy przygotowani na taką sytuację podczas dokonywania zmian w procedurze obsługi powiadomień związanych ze stanami telefonu.

## Protokół inicjalizacji sesji (SIP)

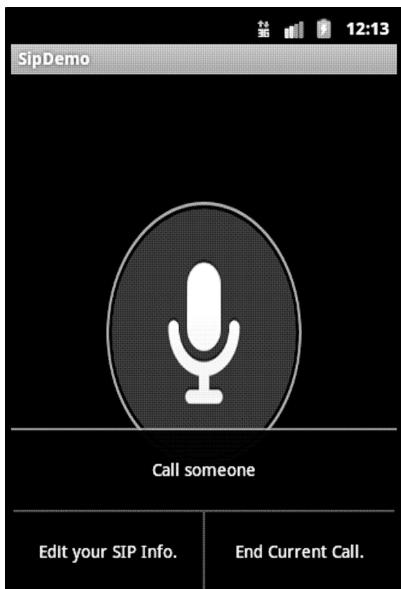
W wersji 2.3 Androida (Gingerbread) wprowadzono nowe funkcje, pozwalające na obsługę protokołu SIP, dokładniej w pakiecie android.net.sip. Protokół SIP stanowi utworzony przez zrzeszenie IETF (ang. *Internet Engineering Task Force*) standard przesyłania strumieni głosowych i wideo poprzez łącze internetowe, co ma umożliwić nawiązywanie komunikacji pomiędzy ludźmi. W całokształcie omawiana technologia jest często nazywaną VoIP (ang. *Voice over Internet Protocol*), pamiętajmy jednak, że istnieje wiele mechanizmów jej implementacji. Na przykład w aplikacji Skype wprowadzono opatentowany protokół, definiujący własną wersję technologii VoIP, który jest niezgodny z protokołem SIP. Omawiany protokół nie jest również tożsam z technologią Google Voice. Google Voice nie obsługuje (w momencie pisania książki) protokołu SIP, chociaż istnieją pewne sposoby integrowania obydwu tych technologii. Usługa Google Voice ustanawia dla nas nowy numer telefonu, dzięki któremu możemy następnie łączyć się z innymi telefonami, na przykład w domu czy w pracy, lub z innym telefonem komórkowym. Niektórzy dostawcy protokołu SIP generują numer telefonu, który może być następnie wykorzystany w usłudze Google Voice, jednak w tym przypadku usługa ta nie rozróżnia numerów utworzonych w koncie SIP. Możemy znaleźć w internecie kilku dostawców protokołu SIP, z których wielu nie pobiera wysokich opłat, a niektórzy oferują darmowe usługi.

Należy zauważyć, że standard SIP nie bierze bezpośredniego udziału w przesyłaniu danych audio i wideo poprzez sieć. Jest on wykorzystywany wyłącznie do ustanawiania i przerywania bezpośredniego połączenia pomiędzy urządzeniami, za pomocą którego można przesyłać dane. Z protokołu SIP korzystają komputery klienckie, a także kodeki audio i wideo oraz różne biblioteki do konfigurowania wywołań pomiędzy użytkownikami. Innymi standardami powiązanymi z wywołaniami protokołu SIP są na przykład: protokół RTP (ang. *Real-time Transport Protocol* — protokół przesyłania w czasie rzeczywistym), RTSP (ang. *Real-time Streaming Protocol* — protokół przesyłania strumienia w czasie rzeczywistym) czy SDP (ang. *Session Description Protocol* — protokół opisu sesji).

Użytkownicy mogą wykonywać połączenia SIP bez ponoszenia opłat za odległość. Program komputerowy może być równie dobrze uruchomiony w urządzeniu przenośnym, na przykład w smartfonie lub tablecie wyposażonym w system Android. Aplikacje wykorzystujące protokół SIP często są nazywane „telefonami programowymi”. Rzeczywista przewaga telefonu programowego zainstalowanego na urządzeniu przenośnym uwidacznia się, gdy urządzenie jest podłączone do internetu w sieci Wi-Fi, dzięki czemu użytkownik nie ponosi żadnych kosztów, mimo że może wykonywać i odbierać połączenia. W przypadku odbierania połączeń telefon programowy musi zarejestrować swoje położenie oraz dostępne funkcje w dostawcy protokołu SIP, dzięki czemu serwer SIP będzie mógł tworzyć bezpośrednie połączenia. Jeżeli telefon programowy adresata jest niedostępny, serwer SIP może skierować wychodzące połączenie na przykład na konto poczty głosowej.

Firma Google udostępnia aplikację demonstracyjną, ukazującą możliwości protokołu SIP, nazywaną SipDemo. Chcielibyśmy przyjrzeć się jej teraz uważniej i wyjaśnić zasadę działania. Dla osób dopiero zapoznających się z protokołem SIP pewne pojęcia mogą na początku sprawiać trudności. Jeżeli będziemy chcieli poeksperymentować z tą aplikacją, prawdopodobnie nie obejdzie się bez fizycznego urządzenia, obsługującego omawiany protokół. Wynika to z faktu, że w czasie, w którym powstawała niniejsza książka, emulatory Androida nie były wyposażone w obsługę standardu SIP (dokładniej rzecz biorąc — sieci Wi-Fi). W internecie pojawiają się opisy prób umożliwienia uruchomienia protokołu SIP na emulatorze i w momencie, kiedy Czytelnik otrzymał tę książkę, mogły się już pojawić jakieś wszechstronne i łatwe do zaimplementowania rozwiązania. Aby móc korzystać z testowej aplikacji, wymagane jest także utworzenie konta u dostawcy SIP. Będzie wymagany własny identyfikator, nazwa domeny (lub adres proxy), także hasło. Będzie je można wprowadzić na ekranie preferencji aplikacji SipDemo. Ostatnią wymaganą rzeczą jest połączenie z siecią Wi-Fi, umożliwiające urządzeniu łączenie się z internetem. Osoby, które nie chcą testować tej aplikacji, nie powinny mieć problemu ze zrozumieniem poniżej omówionych koncepcji. Okno aplikacji SipDemo zaprezentowano na rysunku 18.3.

Aby wczytać aplikację SipDemo jako nowy projekt środowiska Eclipse,łączmy kreator *New Android Project*, wybierzmy w nim jednak opcję *Create project from existing sample*, w sekcji *Build target* zaznaczmy wersję 2.3 Androida (lub wyższą), a następnie kliknijmy listę *Samples* i wybierzmy *SipDemo*. Po kliknięciu przycisku *Finish* środowisko zakończy tworzenie projektu. Możemy uruchomić tę aplikację bez zmian, jednak — jak już wcześniej wspomnieliśmy — bez urządzenia obsługującego standard SIP oraz sieć Wi-Fi niczego nie będziemy mogli w niej dokonać. Jeśli jednak zdecydujemy się na przetestowanie projektu, wciskamy przycisk menu i uzupełniamy dane swojego konta SIP. Do przetestowania połączenia będzie również potrzebne drugie konto SIP. Wciśnięcie obrazu mikrofonu pozwoli nam przesyłać dane głosowe do odbiorcy. Aplikacja demonstracyjna umożliwia również odbieranie połączeń. Zastanówmy się teraz nad mechanizmami wprowadzonymi w pakiecie `android.net.sip`.



Rysunek 18.3. Aplikacja SipDemo z widocznym głównym menu

Pakiet `android.net.sip` obejmuje cztery klasy: `SipManager`, `SipProfile`, `SipSession` oraz `SipAudioCall`. `SipManager` stanowi rdzeń pakietu, za pomocą którego uzyskujemy dostęp do pozostałych funkcji standardu SIP. Aby utworzyć obiekt SIP, wywołujemy statyczną metodę `newInstance()` klasy `SipManager`. Dzięki temu obiektowi możemy wprowadzić klasę `SipSession` w większości aktywności korzystających z protokołu SIP, ewentualnie możemy wykorzystać klasę `SipAudioCall` wyłącznie w celu połączenia dźwiękowego. Oznacza to, że firma Google wprowadziła możliwość wyboru funkcji oferowanych przez standard SIP, mianowicie konfigurowanie połączenia dźwiękowego.

Klasa `SipProfile` służy do definiowania komunikujących się ze sobą kont SIP. Klasa ta nie jest przeznaczona dla urządzenia użytkownika, lecz raczej dla usługi SIP przechowywanej u dostawcy protokołu. Serwery dostawcy będą wspomagały cały proces nawiązywania połączenia.

W klasie `SipSession` są przeprowadzane najważniejsze operacje. Do procesu konfiguracji sesji zalicza się także utworzenie obiektu `SipProfile`, dzięki czemu aplikacja będzie mogła zostać rozpoznana przez serwer dostawcy protokołu SIP. Przekazujemy mu także klasę `SipSession`. Ustanawiamy w tym celu obiekt nasłuchujący, który będzie informowany o interesujących nas zdarzeniach. Po skonfigurowaniu obiektu `SipSession` nasza aplikacja jest gotowa do nawiązania połączenia z innym obiektem `SipProfile` lub do odebrania połączenia. Obiekt nasłuchujący posiada kilka metod zwrotnych, dzięki czemu możemy odpowiednio sobie radzić ze zmieniającymi się stanami sesji.

W wersji Honeycomb najprościej jest wykorzystać klasę `SipAudioCall`. Cała logika w niej zawarta dotyczy powiązania mikrofonu i słuchawek bądź głośnika do strumienia danych, dzięki czemu można przeprowadzać rozmowy. W klasie `SipAudioCall` istnieje wiele metod zarządzających funkcjami wyciszenia, wstrzymania itd. Klasa ta zapewnia obsługę wszelkich funkcji dźwiękowych, jednak wszystkie inne elementy musi zaprojektować sam programista. Klasa `SipSession` zawiera metodę `makeCall()`, służącą do zamieszczania wychodzących połączeń. Głównym parametrem jest opis sesji (w postaci ciągu znaków). Właśnie tutaj

wymagany jest większy nakład pracy. Utworzenie opisu sesji wymaga formatowania zgodnego ze wspomnianym wcześniej protokołem SDP. Zrozumienie otrzymywanego opisu sesji jest równoznaczne z analizowaniem jego składni zgodnie z tym protokołem. Dokumentację tego standardu znajdziemy pod adresem <http://tools.ietf.org/html/rfc4566>. Niestety, zestaw Android SDK nie obsługuje tego formatu. Dzięki pewnym bardzo uprzejmym ludziom istnieje kilka bezpłatnych aplikacji dla Androida posiadających wbudowaną tę funkcję. Mamy na myśli siproid (<http://code.google.com/p/sipdroid/>) oraz csipsimple (<http://code.google.com/p/csipsimple/>).

Nie poruszaliśmy nawet zagadnienia kodeków służących do zarządzania strumieniami wideo pomiędzy połączonymi klientami SIP, chociaż siproid posiada taką możliwość. Innym bardzo atrakcyjnym aspektem protokołu SIP jest możliwość utworzenia połączenia konferencyjnego pomiędzy większą liczbą osób. Tematyka ta przekracza zakres tej książki, mamy jednak nadzieję, że wszyscy docenią możliwości oferowane przez standard SIP.

Zwrócmy uwagę, że aplikacje wykorzystujące protokół SIP będą wymagały do poprawnego działania będą wymagały przynajmniej uprawnień `android.permission.USE_SIP` oraz `android.permission.INTERNET`. Dodatkowe uprawnienia będą jeszcze potrzebne w przypadku używania klasy `SipAudioCall`. Dobrym pomysłem jest również dodanie poniższego znacznika do pliku `AndroidManifest.xml` jako obiektu podzielnego węzła `<manifest>`, dzięki czemu aplikacja będzie instalowana wyłącznie na urządzeniach obsługujących standard SIP:

```
<uses-feature android:name="android.hardware.sip.voip" />
```

## Odbośniki

Poniżej prezentujemy kilka odnośników do materiałów, z którymi warto się dokładniej zapoznać:

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów przygotowanych z myślą o tej książce. Przykłady z tego rozdziału znajdziemy w katalogu *ProAndroid3\_R18\_Telefonia*. Dołączliśmy także plik *Czytaj.TXT*, w którym dokładnie omówiliśmy proces importowania projektów do środowiska Eclipse.
- [http://pl.wikipedia.org/wiki/Session\\_Initiation\\_Protocol](http://pl.wikipedia.org/wiki/Session_Initiation_Protocol) — artykuł z Wikipedii dotyczący protokołu SIP.
- <http://tools.ietf.org/html/rfc3261> — oficjalny standard IETF dotyczący protokołu SIP.
- <http://tools.ietf.org/html/rfc4566> — oficjalny standard IETF dotyczący protokołu SDP.
- <http://code.google.com/p/sipdroid/>, <http://code.google.com/p/csipsimple/> — dwie przeznaczone dla Androida aplikacje o otwartym kodzie źródłowym, implementujące klientów SIP.

## Podsumowanie

W tym rozdziale zajmowaliśmy się interfejsami telefonii. Skupiliśmy się zwłaszcza na wysyłaniu wiadomości tekstowych, monitorowaniu otrzymywanych wiadomości SMS oraz zaprezentowaliśmy metody uzyskania dostępu do różnych folderów związanych z wiadomościami SMS, znajdujących się na urządzeniu. Dokonaliśmy również analizy klasy `TelephonyManager`. Rozdział zakończyliśmy omówieniem zestawu funkcji związanych z protokołem SIP, wprowadzonego w wersji 2.3 systemu Android.

# Używanie szkieletu multimedialnego

Obecnie zajmiemy się bardzo interesującym elementem zestawu Android SDK — szkieletem multimediiów. Pokażemy, w jaki sposób można odtwarzać oraz rejestrować dźwięk i wideo za pomocą różnych źródeł. Zajmiemy się także zagadnieniem wykonywania zdjęć za pomocą wbudowanego aparatu. Wszelkie opisy dotyczące multimediiów byłyby niekompletne, gdybyśmy pominęli tematykę kart SD (ang. *Secure Digital*) oraz możliwości ich wykorzystania, ponieważ będą one często używane do zapisu i odczytu danych.

## Stosowanie interfejsów API multimediiów

W Androidzie obsługa odtwarzania plików audio i wideo została zawarta w pakiecie `android.media`. W niniejszym podrozdziale omówimy interfejsy API multimediiów, dostępne w tym pakiecie.

Rdzeniem pakietu `android.media` jest klasa `android.media.MediaPlayer`. Klasa `MediaPlayer` odpowiada za odtwarzanie plików audio i wideo. Treść obsługiwana przez tę klasę może pochodzić z następujących źródeł:

- **Internet** — możemy odtwarzać multimedia umieszczone pod danym adresem URL.
- **Plik .apk** — istnieje możliwość odtwarzania plików skompilowanych w pliku `.apk`. Można umieścić odtwarzane pliki jako zasoby lub pliki dodatkowe (w folderze `assets`).
- **Karta SD** — pliki umieszczone na karcie SD urządzenia również mogą być odtwarzane.

Klasa `MediaPlayer` potrafi obsługiwać kilka różnych formatów multimediiów, w tym takie jak 3GPP (`.3gp`), MP3 (`.mp3`), MIDI (`.mid` i inne), Ogg Vorbis (`.ogg`), PCM/WAVE (`.wav`) oraz MPEG-4 (`.mp4`). W przypadku wersji 3.0 Androida obsługiwane są również media strumieniowe, przesypane protokołem HTTP w czasie rzeczywistym, oraz listy odtwarzania M3U. Pełną listę obsługiwanych formatów można znaleźć pod adresem:

<http://developer.android.com/guide/appendix/media-formats.html>

## Wykorzystywanie kart SD

Zanim przejdziemy do procesu tworzenia i stosowania różnych rodzajów multimedialnych, zobaczymy, w jaki sposób pracujemy z kartami SD. Karty SD są używane w urządzeniach obsługujących system Android do przechowywania dużej ilości danych użytkownika, na przykład plików obrazów, audio i wideo. Zasadniczo są to niewielkie układy scalone przechowujące dane nawet przy braku zasilania. W rzeczywistym telefonie karta SD jest umieszczana w gnieździe pamięci i staje się dostępna dla urządzenia. Większość urządzeń posiada tylko jedno gniazdo pamięci i karta SD zazwyczaj nie jest w nich wymieniana. W niektórych urządzeniach można posiadać wiele kart i przełączać się pomiędzy nimi w obrębie jednego urządzenia, a także można je wymieniać pomiędzy różnymi urządzeniami. Na szczęście emulator Androida potrafi symulować karty SD, a za ich pojemność służy przestrzeń dysku twardego.

Podczas tworzenia pierwszego urządzenia AVD w rozdziale 2. określiliśmy rozmiar karty SD, dzięki czemu stała się ona dostępna dla aplikacji podczas jej uruchomienia na emulatorze. Jeżeli przyjrzymy się zawartości utworzonego katalogu urządzenia AVD, ujrzymy plik *sdcard.img*, w którym zdefiniowano rozmiar karty. Nie korzystaliśmy wówczas z tej symulacji karty, będziemy jednak to robić w tym rozdziale.

Projektanci aplikacji mogą, po utworzeniu karty SD, używać narzędzi środowiska Eclipse do umieszczania plików multimedialnych (gwoli ściśności, dowolnego rodzaju plików) na karcie SD. Do umieszczania plików na takiej karcie lub usuwania ich z niej możemy również wykorzystać aplikację *adb*. Aplikacja ta jest umieszczona w podkatalogu *tools* środowiska Android SDK; w rozdziale 2. opisaliśmy łatwy sposób uzyskania do niej dostępu z okna narzędzi.

Wiemy już, w jaki sposób wygenerować symulację karty SD podczas procesu tworzenia urządzenia AVD. Oczywiście możemy również utworzyć wiele takich samych urządzeń AVD różniących się jedynie rozmiarem karty SD. Istnieje jeszcze inny sposób. Wśród narzędzi środowiska SDK znajduje się aplikacja *mksdcard*, służąca do tworzenia obrazu karty SD. W rzeczywistości aplikacja ta generuje sformatowany plik, wykorzystywany jako karta SD. Aby skorzystać z tej aplikacji, musimy najpierw wyszukać lub utworzyć folder, w którym będzie przechowywany obraz karty SD, na przykład *c:\Android\sdcard\*. Następnie otwieramy okno narzędzi (w rozdziale 2. zostały umieszczone informacje na temat okna narzędzi) i wpisujemy następujące polecenie, podając ścieżkę do obrazu karty SD:

```
mksdcard 256M c:\Android\sdcard\sdcard.img
```

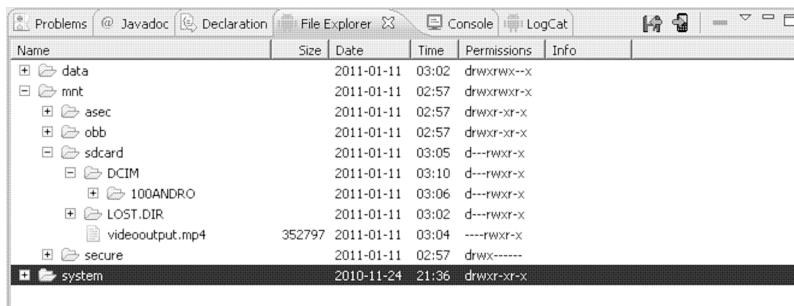
To przykładowe polecenie tworzy obraz karty SD nazwany *sdcard.img* w lokacji *c:\Android\sdcard\*. Rozmiar karty wynosi 256 MB. Do określania innych rozmiarów można stosować przedrostek K dla kilobajtów, jednak dla gigabajtów jeszcze nie zaimplementowano przedrostka jednostki G, zatem w celu określenia pojemności wyrażonej w gigabajtach należy podawać wielokrotności wartości 1024 M. Możemy również po prostu zdefiniować liczbę całkowitą reprezentującą całkowitą liczbę bajtów. Należy również pamiętać, że emulator Androida nie będzie obsługiwał kart SD o rozmiarze mniejszym niż 8 MB.

Narzędzie ADT środowiska Eclipse umożliwia zdefiniowanie dodatkowych argumentów wiersza poleceń podczas uruchamiania emulatora. Aby znaleźć pole umożliwiające dostęp do opcji emulatora, przejdźmy do okna *Preferences* w aplikacji Eclipse, a następnie wybierzmy *Android/Launch*. Teoretycznie moglibyśmy tutaj dodać polecenie *-sdcard "PATH\_TO\_YOUR\_SD\_CARD\_→IMAGE\_FILE"*, dzięki czemu zostałaby przesłonięta ścieżka do pliku karty SD urządzenia AVD. Sposób ten jednak nie działa już od kilku wersji Androida i zawsze otrzymujemy obraz karty SD utworzonej wraz z urządzeniem AVD. Najpewniejszym sposobem oddzielenia karty SD od

urządzenia AVD jest uruchomienie emulatora z poziomu wiersza poleceń i określenie tam, który obraz karty SD ma być wykorzystywany. Poniższe polecenie, wpisane w oknie narzędzi, uruchamia dane urządzenie AVD wraz z wybranym obrazem karty SD, a nie z obrazem karty SD utworzonej wraz z tym urządzeniem:

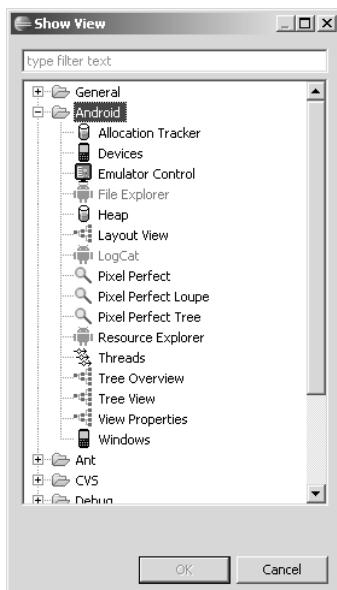
```
emulator -avd AVDName -sdcard "PATH_TO_YOUR_SD_CARD_IMAGE_FILE"
```

Tuż po utworzeniu karty SD jest ona pusta. Można na niej umieszczać pliki poprzez narzędzie File Explorer w środowisku Eclipse. W tym celu włączamy emulator i czekamy, aż się uruchomi do końca. Następnie w środowisku Eclipse przechodzimy do perspektywy Java, Debug lub DDMS i wyszukujemy zakładkę *File Explorer*, pokazaną na rysunku 19.1.



Rysunek 19.1. Widok zakładki File Explorer

Jeżeli zakładka ta nie jest widoczna, możemy ją wyświetlić, klikając *Windows>Show View/Other/Android* i zaznaczając opcję *File Explorer*. Ewentualnie możemy przejść do perspektywy DDMS, wybierając *Window/Open Perspective/Other/DDMS*. Widok *File Explorer* jest domyślny w perspektywie DDMS. Lista wszystkich dostępnych widoków została zaprezentowana na rysunku 19.2.



Rysunek 19.2. Włączanie widoków Androida

Aby umieścić plik na karcie SD, zaznaczamy folder *sdcard* w oknie *File Explorer* i wybieramy przycisk (w prawym górnym rogu ekranu), na którym widoczna jest strzałka skierowana w prawo, wskazująca ikonę telefonu. Zostanie uruchomione okno dialogowe, w którym możemy wybrać plik. Zaznaczamy plik, który chcemy umieścić na karcie pamięci. Obok opisanego przycisku znajduje się przycisk, na którym widoczna jest skierowana w lewo strzałka, wskazująca dyskietkę. Po zaznaczeniu pliku w oknie *File Explorer* za pomocą tego przycisku przenosimy plik z karty SD na dysk twardy stacji roboczej.

Jeśli w oknie *File Explorer* będzie wyświetlony pusty widok, to albo nie uruchomiliśmy emulatora, albo środowisko Eclipse rozłączyło się z emulatorem, albo urządzenie AVD uruchomione na emulatorze nie jest zaznaczone w zakładce *Devices*, tak jak zostało pokazane na rysunku 19.1. Aby przejść do zakładki *Devices*, należy wykonać takie same czynności jak w przypadku widoku *File Explorer*. Widok ten powinien być również domyślnie widoczny w perspektywie DDMS.

Innym sposobem przenoszenia plików z karty SD lub na nią jest zastosowanie narzędzia adb. Aby to sprawdzić, otwieramy okno narzędzi i wpisujemy następujące polecenie:

```
adb push c:\ścieżka_do_pliku\nazwa_pliku /mnt/sdcard/newfile
```

W ten sposób plik zostanie przeniesiony ze stacji roboczej na kartę SD. Zwróćmy uwagę, że urządzenie stosuje ukośniki (/) do oddzielania katalogów. Używamy dowolnego znaku oddzielania katalogu wykorzystywanego przez stację roboczą dla kopowanego pliku oraz wpisujemy właściwą ścieżkę dostępu do pliku umieszczonego w stacji roboczej. I odwrotnie, poniższe polecenie przekopiuje plik z karty SD do stacji roboczej:

```
adb pull /mnt/sdcard/nazwa_pliku c:\ścieżka_docelowa\nazwa_pliku
```

Jedną z ciekawszych funkcji tego polecenia jest tworzenie katalogów w razie potrzeby, a także pobieranie i wysyłanie plików po zdefiniowaniu docelowej ścieżki. Niestety, aplikacja adb nie posiada możliwości równoczesnego kopирования wielu plików. Każdy plik należy przenosić oddziennie.

#### Uwaga!

Aż do wersji 2.2 Androida karta SD była najczęściej umiejscowiona w folderze */sdcard*. W tej oraz nowszych wersjach karta SD znajduje się w folderze */mnt/sdcard*, jednak istnieje symboliczne powiązanie, nazwane */sdcard*, wskazujące ścieżkę */mnt/sdcard*. Zapewnia ono wsteczną kompatybilność.

Na karcie SD można zauważać katalog *DCIM*. Przechowywane są w nim obrazy wykonane za pomocą cyfrowego aparatu fotograficznego. Umieszczenie katalogu *DCIM*, służącego do przechowywania zdjęć cyfrowych, na głównym poziomie karty SD jest określonym standardem przemysłowym. Standardem jest również tworzenie wewnętrz katalogu *DCIM* podkatalogu reprezentującego aparat fotograficzny o nazwie w formacie *123ABCDE*, czyli pięć liter poprzedzonych trzema cyframi. Emulator tworzy podkatalog *100ANDRO*, jednak producenci aparatów fotograficznych i urządzeń obsługujących system Android mogą nadać mu dowolną nazwę. Emulator zawiera również, podobnie jak niektóre telefony, podkatalog *Camera*, nazwa ta nie jest jednak zgodna ze standardem. Niemniej możemy znaleźć zdjęcia w podkatalogu *Camera*, w podkatalogu *100ANDRO* lub w jakimś innym podkatalogu umieszczonym w katalogu *DCIM*.

Niestety nie istnieje żadna metoda informująca nas, który folder wewnętrz katalogu *DCIM* może zostać przeznaczony na zdjęcia. Mamy jednak do dyspozycji parę metod, dzięki którym można znaleźć główny katalog karty SD. Jedną z nich jest *Environment.getExternalStorageDirectory()*, przekazującą obiekt *File* nadzawanego katalogu na karcie SD. W urządzeniach

(nie wszystkich) wykorzystujących wersje Androida starsze od 2.2 prawdopodobnie otrzymaliśmy nazwę `/sdcard`. W przypadku wersji 2.2 i nowszych najczęściej spotykana jest ścieżka `/mnt/sdcard`. O wiele lepiej wykorzystywać metodę `Environment`, niż zakładając, że znamy nazwę głównego katalogu karty SD. Omówimy teraz drugą metodę.

Od wersji 2.2 Androida (o nazwie kodowej Froyo) do klasy `Environment` zostały wprowadzone nowe stałe, a także nowa metoda, służące do lokalizowania katalogów. Wcześniej hierarchia plików na karcie SD była dość nieuporządkowana, nie istniała bowiem tutaj, nie licząc katalogu `DCIM`, standardyzowana specyfikacja nazewnictwa katalogów. Wraz z wersją Froyo nastąpiła pewna standardyzacja nazw katalogów, co zostało zaprezentowane w tabeli 19.1. W trzeciej kolumnie została podana stosowana nazwa katalogu w emulatorze, gdzie główny katalog będzie prawdopodobnie wyglądał tak: `/mnt/sdcard` (w zależności od urządzenia). Z powodu różnic w nazewnictwie katalogów powinniśmy zawsze wykorzystywać klasę `Environment` do znalezienia tego właściwego.

**Tabela 19.1.** Ustandardyzowane nazwy folderów na karcie SD

Stała katalogu	Opis	Nazwa katalogu w emulatorze
<code>DIRECTORY_ALARMS</code>	W tym katalogu Android wyszukuje dźwięki używane w alarmach.	<code>Alarms</code>
<code>DIRECTORY_DCIM</code>	Standardowy katalog wykorzystywany do przechowywania zdjęć i plików wideo wykonanych za pomocą aparatu.	<code>DCIM</code>
<code>DIRECTORY_DOWNLOADS</code>	Standardowy katalog przechowujący pobrane pliki.	<code>Download</code> (uwaga: liczba pojedyncza)
<code>DIRECTORY_MOVIES</code>	W tym katalogu znajdują się filmy, które mogą być oglądane przez użytkownika.	<code>Movies</code>
<code>DIRECTORY_MUSIC</code>	Tutaj zamieszczone są pliki dźwiękowe, wykorzystywane jako utwory muzyczne.	<code>Music</code>
<code>DIRECTORY_NOTIFICATIONS</code>	Przechowywane są tu pliki dźwiękowe, stosowane przy powiadomieniach.	<code>Notifications</code>
<code>DIRECTORY_PICTURES</code>	Katalog, w którym magazynowane są obrazy niewykonane za pomocą aparatu.	<code>Pictures</code>
<code>DIRECTORY_PODCASTS</code>	Tutaj są przechowywane pliki dźwiękowe przyjmujące postać podcastów.	<code>Podcasts</code>
<code>DIRECTORY_RINGTONES</code>	Katalog dla plików dźwiękowych, wykorzystywanych jako dzwonki.	<code>Ringtones</code>

Nową metodą, służącą do lokalizowania katalogów, jest `Environment.getExternalStoragePublicDirectory(String type)`, gdzie w miejsce parametru `type` wstawia się jedną ze stałych widocznych w tabeli 19.1. Metoda ta przekazuje obiekt `File` reprezentujący poszukiwany katalog. Metoda ta nie istnieje w wersjach Androida starszych od Froyo, a nawet w nowszych urządzeniach możemy natrafić na pewne różnice. Na przykład urządzenia firmy Samsung posiadają dwa gniazda pamięci, zatem powyższe metody okazują się niewystarczające do określenia wszystkich katalogów dostępnych na kartach SD.

Należy jeszcze wspomnieć o zabezpieczeniach. Począwszy od wersji 1.6 środowiska Android SDK, aby umożliwić aplikacji zapisywanie plików na karcie SD, musimy wprowadzić następujące uprawnienie do pliku manifestu tej aplikacji:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Aplikacje tworzone w starszej wersji środowiska SDK nie wymagają podania tego uprawnienia. Oznacza to, że jeżeli wartość parametru `minSdkVersion` aplikacji jest mniejsza od 4 (odpowiada ona wersji 1.6 zestawu Android SDK), to nie musimy dodawać powyższego znacznika do pliku `AndroidManifest.xml`, nawet jeśli aplikacja jest uruchomiona na urządzeniu obsługującym nowszą wersję środowiska SDK. Zatem jeżeli podczas procesu tworzenia aplikacji wybierzemy wersję Android 1.6 lub nowszą (wartość `minSdkVersion` równa co najmniej 4) i chcemy, aby istniała możliwość zapisywania danych na karcie SD, musimy dodać powyższy znacznik do naszego pliku manifestu. Jeżeli używamy wersji 1.5 Androida lub niższej, nie potrzebujemy tego znacznika. Skoro zapoznaliśmy się już z podstawami kart SD, przejdźmy do multimedialnych dźwiękowych.

## Odtwarzanie multimedialnych

Rozpoczniemy od napisania prostej aplikacji odtwarzającej plik MP3 udostępniony w internecie (rysunek 19.3). Następnie omówimy zastosowanie metody `setDataSource()` klasy `MediaPlayer`, dzięki której możliwe jest odtwarzanie zawartości multimedialnej pliku `.apk` lub karty SD. Klasa `MediaPlayer` nie jest jednak jedynym sposobem umożliwiającym odtwarzanie dźwięku, zatem przyjrzymy się klasie `SoundPool`, a także klasom `MediaPlayer`, `AsyncPlayer` oraz występującej na najniższym poziomie złożoności klasie `AudioTrack`. Następnie opiszemy kilka braków dostrzeżonych w klasie `MediaPlayer`. Temat zamknieniśmy omówieniem sposobu odtwarzania plików wideo.



Rysunek 19.3. Interfejs aplikacji obsługującej multimedia

## Odtwarzanie źródeł dźwiękowych

Na rysunku 19.3 przedstawiono interfejs użytkownika dla naszego pierwszego przykładowego projektu. W aplikacji tej zademonstrujemy kilka podstawowych funkcji klasy `MediaPlayer`, na przykład odtwarzanie, wstrzymywanie oraz ponowne uruchamianie pliku multimedialnego. Przyjrzyjmy się układowi graficznemu interfejsu użytkownika.

Interfejs UI składa się z menedżera `LinearLayout` zawierającego cztery przyciski: jeden służy do uruchomienia odtwarzacza, drugi do jego wstrzymania, trzeci do jego ponownego uruchomienia, natomiast czwarty do jego zatrzymania. Plik układu graficznego aplikacji oraz jej kod Java zostały umieszczone na listingu 19.1. Zakładamy, że ten przykładowy projekt będzie tworzony dla wersji Androida o numerze co najmniej 2.2, ponieważ korzystamy w nim z metody `getExternalStoragePublicDirectory()` klasy `Environment`. Jeżeli zechcemy skorzystać ze starszej wersji Androida, po prostu zastąpmy tę metodę funkcją `getExternalStorageDirectory()` i wprowadźmy odpowiednie dane dotyczące lokalizacji plików, aby aplikacja mogła je odnaleźć.

**Uwaga!**

W umieszczonym na końcu rozdziału podrozdziale „Odnośniki” znajdziemy adres URL, z którego możemy pobrać i zimportować omówione tu projekty do środowiska Eclipse, zamiast mozołnie kopiować i wklejać kod.

**Listing 19.1.** Układ graficzny oraz kod aplikacji odtwarzającej multimedia

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<Button android:id="@+id/startPlayerBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Odtwarzaj plik audio" android:onClick="doClick"
    />
<Button android:id="@+id/restartPlayerBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Uruchom ponownie odtwarzacz" android:onClick="doClick"
    />
<Button android:id="@+id/pausePlayerBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Wstrzymaj odtwarzacz" android:onClick="doClick"
    />
<Button android:id="@+id/stopPlayerBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Zatrzymaj odtwarzacz" android:onClick="doClick" />
</LinearLayout>
```

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.content.res.AssetFileDescriptor;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.View;

public class MainActivity extends Activity
{
    static final String AUDIO_PATH =
        "http://www.androidbook.com/akc/filestorage/android/documentfiles/3389/play.mp3";
// Environment.getExternalStoragePublicDirectory(
// Environment.DIRECTORY_MUSIC) +
// "/music_file.mp3";
// Environment.getExternalStoragePublicDirectory(
// Environment.DIRECTORY_MOVIES) +
// "/movie.mp4";

    private MediaPlayer mediaPlayer;
    private int playbackPosition=0;

    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void doClick(View view) {
        switch(view.getId()) {
            case R.id.startPlayerBtn:
                try {
                    // Tylko jedna z tych metod nie powinna być wykomentowana
                    playAudio(AUDIO_PATH);
                    // playLocalAudio();
                    // playLocalAudio_UsingDescriptor();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
            case R.id.pausePlayerBtn:
                if(mediaPlayer != null && mediaPlayer.isPlaying()) {
                    playbackPosition = mediaPlayer.getCurrentPosition();
                    mediaPlayer.pause();
                }
                break;
            case R.id.restartPlayerBtn:
                if(mediaPlayer != null && !mediaPlayer.isPlaying()) {
                    mediaPlayer.seekTo(playbackPosition);
                    mediaPlayer.start();
                }
                break;
        }
    }
}
```

```
case R.id.stopPlayerBtn:
    if(mediaPlayer != null) {
        mediaPlayer.stop();
        playbackPosition = 0;
    }
    break;
}

private void playAudio(String url) throws Exception
{
    killMediaPlayer();

    mediaPlayer = new MediaPlayer();
    mediaPlayer.setDataSource(url);
    mediaPlayer.prepare();
    mediaPlayer.start();
}

private void playLocalAudio() throws Exception
{
    mediaPlayer = MediaPlayer.create(this, R.raw.music_file);
    // W tym przypadku wywoływanie metody prepare() nie jest wymagane
    mediaPlayer.start();
}

private void playLocalAudio_UsingDescriptor() throws Exception {

    AssetFileDescriptor fileDesc = getResources().openRawResourceFd(
        R.raw.music_file);
    if (fileDesc != null) {

        mediaPlayer = new MediaPlayer();
        mediaPlayer.setDataSource(fileDesc.getFileDescriptor(),
            fileDesc.getStartOffset(), fileDesc.getLength());

        fileDesc.close();

        mediaPlayer.prepare();
        mediaPlayer.start();
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    killMediaPlayer();
}

private void killMediaPlayer() {
    if(mediaPlayer!=null) {
        try {
            mediaPlayer.release();
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
    }
}
```

W tym przypadku odtwarzamy plik MP3 udostępniony w internecie, zatem musimy umieścić w pliku manifeście uprawnienie android.permission.INTERNET. W kodzie z listingu 19.1 widać, że klasa MainActivity obejmuje trzy elementy: ciąg znaków final określający adres URL pliku MP3, wystąpienie klasy MediaPlayer oraz obiekt playbackPosition przyjmujący wartości w postaci liczb całkowitych. Nasza metoda onCreate() konfiguruje jedynie interfejs graficzny z pliku XML. W procedurze obsługi kliknięcia przycisku *Odtwarzaj plik audio* jest wywoływana metoda playAudio(). W metodzie tej tworzymy nowe wystąpienie klasy MediaPlayer, a źródłem danych odtwarzacza staje się adres URL pliku MP3. Następnie wywołujemy metodę odtwarzacza prepare(), służącą do przygotowania odtwarzania, po niej zaś zostaje przywołana metoda start(), rozpoczynająca odtwarzanie.

Spójrzmy teraz na kod przycisków *Wstrzymaj odtwarzacz i Uruchom ponownie odtwarzacz*. Widzimy, że po kliknięciu przycisku *Wstrzymaj odtwarzacz* otrzymujemy bieżącą pozycję odtwarzacza za pomocą wywołania metody getCurrentPosition(). Następnie wywołujemy metodę pause(), aby wstrzymać odtwarzanie. Przed ponownym uruchomieniem odtwarzacza wywołujemy metodę seekTo(), która pobiera pozycję przechowywaną przez metodę getCurrentPosition(), a w dalszej kolejności przywołujemy metodę start().

Klasa MediaPlayer posiada także metodę stop(). Jeżeli za pomocą tej metody zatrzymamy odtwarzacz, przed ponownym wywołaniem metody start() musimy przywołać znowu metodę prepare(). W przypadku wstrzymania odtwarzacza poprzez metodę pause() nie musimy wywoływać metody prepare() przed ponownym uruchomieniem odtwarzania. Po zakończeniu korzystania z aplikacji musimy również wywołać metodę release(). W naszym przykładzie jest ona częścią metody killMediaPlayer().

Na listingu 19.1 pokazaliśmy sposób odtwarzania pliku audio udostępnionego w internecie. Klasa MediaPlayer obsługuje również odtwarzanie lokalnych plików multimedialnych, będących częścią pakietu .apk. Na listingu 19.2 przedstawiono technikę tworzenia odniesienia do pliku zawartego w folderze /res/raw pakietu .apk oraz sposób jego odtwarzania. Możemy utworzyć katalog raw w węźle res, jeśli nie został jeszcze utworzony podczas generowania projektu w środowisku Eclipse. Następnie skopiujmy dowolny plik MP3 nazwany *music\_file.mp3* do podkatalogu /res/raw.

---

**Listing 19.2.** Zastosowanie klasy MediaPlayer do odtwarzania lokalnego pliku w aplikacji

---

```
private void playLocalAudio() throws Exception
{
    mediaPlayer = MediaPlayer.create(this, R.raw.music_file);
    // W tym przypadku wywoływanie metody prepare() nie jest konieczne
    mediaPlayer.start();
}
```

Jeżeli plik audio lub wideo ma się znaleźć w aplikacji, powinniśmy go umieścić w katalogu /res/raw. Możemy następnie uzyskać wystąpienie klasy MediaPlayer dla tego zasobu poprzez przekazanie jej identyfikatora zasobu tego pliku; w tym celu wywołujemy statyczną metodę

`create()`, tak jak pokazano na listingu 19.2. Odnotujmy fakt, że klasa `MediaPlayer` również zapewnia metody `create()`, dzięki którym można uzyskać do niej dostęp, zamiast samemu tworzyć jej nowy egzemplarz. Na przykład na listingu 19.2 wywołujemy metodę `create()`, lecz równie dobrze moglibyśmy wywołać konstruktor `MediaPlayer(Context context, int resourceId)`. Zalecane jest stosowanie statycznych metod `create()`, ponieważ ukrywają one proces tworzenia klasy `MediaPlayer`, a w tym przypadku ważną rolę odgrywa wywołanie metody `prepare()`. Jednak, jak się wkrótce okaże, czasami nie można wybierać pomiędzy tymi opcjami — w przypadku gdy nie będzie można lokalizować źródeł danych multimedialnych za pomocą identyfikatora zasobów lub adresu URL, trzeba utworzyć obiekt domyślnego konstruktora.

## Metoda `setDataSource`

Na listingu 19.2 wywołaliśmy metodę `create()`, pozwalającą na wczytanie pliku audio z nieskompresowanego zasobu. Dzięki temu nie musimy wywoływać metody `setDataSource()`. Ewentualnie, jeżeli sami utworzymy klasę `MediaPlayer` za pomocą domyślnego konstruktora lub jeśli nie można uzyskać dostępu do pliku multimedialnego za pomocą identyfikatora zasobów bądź adresu URL, będzie potrzebna metoda `setDataSource()`.

Metoda `setDataSource()` istnieje w olbrzymiej liczbie wersji, dzięki którym możemy dostosować źródło danych do własnych potrzeb. Na przykład na listingu 19.3 został ukazany sposób wczytania pliku audio z nieskompresowanego zasobu za pomocą obiektu `FileDescriptor`.

**Listing 19.3.** Konfigurowanie źródła danych dla klasy `MediaPlayer` za pomocą obiektu `FileDescriptor`

---

```
private void playLocalAudio_UsingDescriptor() throws Exception {
    AssetFileDescriptor fileDesc = getResources().openRawResourceFd(
        R.raw.music_file);
    if (fileDesc != null) {
        mediaPlayer = new MediaPlayer();
        mediaPlayer.setDataSource(fileDesc.getFileDescriptor(),
            fileDesc.getStartOffset(), fileDesc.getLength());
        fileDesc.close();
        mediaPlayer.prepare();
        mediaPlayer.start();
    }
}
```

---

Zakładamy, że kod z listingu 19.3 znajduje się wewnętrz konekstu aktywności. Jak widać, wywołujemy metodę `getResources()`, aby uzyskać dostęp do zasobów aplikacji, a następnie korzystamy z metody `openRawResourceFd()` w celu otrzymania deskryptora pliku audio, znajdującego się w folderze `res/raw`. W dalszej kolejności wywołujemy metodę `setDataSource()`, która poprzez obiekt `AssetFileDescriptor` otrzymuje informację o początkowej i końcowej pozycji odtwarzania. Ta wersja metody `setDataSource()` może być również wykorzystana do odtwarzania określonego fragmentu pliku. Jeżeli chcemy zawsze odtwarzać plik w całości, możemy użyć prostszej wersji metody `setDataSource(FileDescriptor desc)`, która nie wymaga wartości początkowej i czasu trwania odtwarzania tego pliku.

Zastosowanie jednej z wersji metody `setDataSource()` zawierającej obiekt `FileDescriptor` może być również przydatne w przypadku konieczności odtwarzania pliku multimedialnego zlokalizowanego w katalogu aplikacji `/data`. Z powodu zabezpieczeń odtwarzacz multimediiów nie posiada dostępu do katalogu `/data` innej aplikacji, jednak ta aplikacja może otworzyć potrzebny plik, a następnie wprowadzić obiekt `FileDescriptor` (otwarty) do metody `setDataSource()`. Pamiętajmy, że katalog `/data` aplikacji znajduje się w zbiorze plików i folderów pod adresem `/data/data/APP_PACKAGE_NAME/`. Możemy uzyskać dostęp do tego katalogu, wywołując odpowiednią metodę klasy `Context`, zamiast umieszczać ścieżkę wewnętrz kodu. Na przykład możemy wywołać metodę `getFilesDir()` klasy `Context`, aby uzyskać dostęp do ścieżki plików znajdujących się w katalogu bieżącej aplikacji. Aktualnie ścieżka ta wygląda następująco: `/data/data/APP_PACKAGE_NAME/files`. W analogiczny sposób możemy wywołać metodę `getCacheDir()`, aby uzyskać dostęp do katalogu pamięci podręcznej aplikacji. Aplikacja będzie odczytywała oraz zapisywała uprawnienia zasobów znajdujących się w tych folderach, zatem możemy dynamicznie tworzyć pliki i przekazywać je odtwarzaczowi. Na koniec, w przypadku korzystania z obiektu `FileDescriptor` w podobny sposób jak na listingu 19.3 nie zapominajmy o zamknięciu procedury wywołującej po wywoaniu metody `setDataSource()`.

Zauważmy, że katalog `/data` aplikacji w znacznym stopniu różni się od jej folderu `/res/raw`. Folder `/res/raw` fizycznie stanowi część pliku `.apk` i jest statyczny — to znaczy, że nie możemy pliku `.apk` modyfikować w sposób dynamiczny. Z kolei zawartość katalogu `/data` jest dynamiczna.

Pozostało nam do omówienia jeszcze jedno źródło plików audio — karta SD. Na początku rozdziału pokazaliśmy, w jaki sposób pliki są umieszczane na karcie SD. Ich odtwarzanie za pomocą klasy `MediaPlayer` jest całkiem proste. W powyższym przykładzie użyliśmy metody `setDataSource()` do uzyskania dostępu do pliku umieszczonego w internecie poprzez przekazanie jej adresu URL tego pliku. W przypadku pliku audio na karcie SD stosujemy tę samą metodę `setDataSource()`, tym razem jednak przekazujemy jej ścieżkę do pliku MP3 umieszczonego na tej karcie. Jeśli na przykład umieścimy na karcie SD plik `music_file.mp3` w standardowym folderze `Music`, możemy zmodyfikować zmienną `AUDIO_PATH` i muzyka będzie odtwarzana po wstawieniu następującego fragmentu kodu:

```
static final String AUDIO_PATH =  
Environment.getExternalStoragePublicDirectory(  
Environment.DIRECTORY_MUSIC) + "/music_file.mp3";
```

## Zastosowanie klasy SoundPool do równoczesnego odtwarzania ścieżek

Klasa `MediaPlayer` stanowi istotne narzędzie w naszym warsztacie muzycznym, pozwala ona jednak na przetwarzanie w danej chwili tylko jednego pliku audio lub wideo. Co zatem można zrobić w przypadku potrzeby jednoczesnego odtwarzania większej liczby ścieżek dźwiękowych? Jednym z rozwiązań jest utworzenie wielu wystąpień klasy `MediaPlayer` i ich równoczesne przetwarzanie. W przypadku gdy chcemy odtwarzać krótkie fragmenty audio i zależy nam na płynności, możemy zastosować klasę `SoundPool`. Sama klasa `SoundPool` wykorzystuje klasę `MediaPlayer`, nie posiadamy jednak dostępu do jej interfejsu.

Kolejna różnica pomiędzy klasami `SoundPool` a `MediaPlayer` polega na tym, że ta pierwsza jest przeznaczona wyłącznie do pracy z plikami lokalnymi. Oznacza to, że możemy wczytywać dźwięki za pomocą plików zasobów, korzystając z deskryptorów lub ścieżek plików. Istnieje jeszcze kilka innych przydatnych funkcji klasy `SoundPool`, na przykład możemy zapętlić odtwarzaną ścieżkę czy wstrzymywać i ponawiać odtwarzanie pojedynczych lub wszystkich utworów jednocześnie.

Klasa SoundPool nie jest jednak pozbawiona wad. Istnieje wspólny bufor przeznaczony dla wszystkich ścieżek zarządzanych przez tę klasę i nie jest on zbyt pojemny. W zasadzie jego rozmiar wynosi 1 MB. Może się to wydawać dużą wielkością w przypadku plików MP3, których rozmiary często nie przekraczają kilku kilobajtów. Klasa SoundPool dekompresuje jednak plik audio w pamięci, aby odtwarzanie dźwięku przebiegało szybko i sprawnie. Rozmiar strumienia audio w pamięci zależy od prędkości transmisji, liczby kanałów (mono lub stereo), częstotliwości próbkowania oraz długości ścieżki. Jeżeli mamy problem z wczytaniem dźwięków do klasy SoundPool, powinniśmy wprowadzić plik źródłowy o nieco niższych parametrach jakościowych, aby zmniejszyć zużycie pamięci.

Zaprezentujemy teraz przykładową aplikację, pozwalającą na wczytanie i odtwarzanie dźwięków wydawanych przez zwierzęta. Jeden z dźwięków, odtwarzany bez przerwy w tle, jest wydawany przez świerszcze. Pozostałe dźwięki są odtwarzane w różnych odstępach czasowych. Czasami będziemy słyszeć wyłącznie świerszcze, a za innym razem odezwie się kilka zwierząt naraz. Umieścimy również w interfejsie przycisk pozwalający na wstrzymywanie i ponowne uruchamianie odtwarzania. Listing 19.4 zawiera plik układu graficznego oraz kod Java definiujący aktywność. zalecamy Czytelnikowi pobranie tego projektu z naszej oficjalnej strony, ponieważ oprócz kodu są tam również pliki dźwiękowe. Adres URL do tej strony można znaleźć na końcu rozdziału, w podrozdziale „Odnośniki”.

#### **Listing 19.4.** Odtwarzanie dźwięku za pomocą klasy SoundPool

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    >
<ToggleButton android:id="@+id/button"
    android:textOn="Wstrzymaj" android:textOff="Wznów"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:onClick="doClick" android:checked="true" />
</LinearLayout>

// Jest to plik MainActivity.java
import java.io.IOException;
import android.app.Activity;
import android.content.Context;
import android.content.res.AssetFileDescriptor;
import android.media.AudioManager;
import android.media.SoundPool;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.view.View;
import android.widget.ToggleButton;

public class MainActivity extends Activity implements
    SoundPool.OnLoadCompleteListener {
    private static final int SRC_QUALITY = 0;
    private static final int PRIORITY = 1;
    private SoundPool soundPool = null;
    private AudioManager aMgr;
```

```
private int sid_background;
private int sid_roar;
private int sid_bark;
private int sid_chimp;
private int sid_rooster;

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}

@Override
protected void onResume() {
    soundPool = new SoundPool(5, AudioManager.STREAM_MUSIC,
        SRC_QUALITY);
    soundPool.setOnLoadCompleteListener(this);

    aMgr =
        (AudioManager)this.getSystemService(Context.AUDIO_SERVICE);

    sid_background = soundPool.load(this, R.raw.crickets, PRIORITY);

    sid_chimp = soundPool.load(this, R.raw.chimp, PRIORITY);
    sid_rooster = soundPool.load(this, R.raw.rooster, PRIORITY);
    sid_roar = soundPool.load(this, R.raw.roar, PRIORITY);

    try {
        AssetFileDescriptor afd =
            this.getAssets().openFd("dogbark.mp3");
        sid_bark = soundPool.load(afd.getFileDescriptor(),
            0, afd.getLength(), PRIORITY);
        afd.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    //sid_bark = soundPool.load("/mnt/sdcard/dogbark.mp3", PRIORITY);

    super.onResume();
}

public void doClick(View view) {
    switch(view.getId()) {
    case R.id.button:
        if(((ToggleButton)view).isChecked()) {
            soundPool.autoResume();
        }
        else {
            soundPool.autoPause();
        }
        break;
    }
}

@Override
protected void onPause() {
```

```

        soundPool.release();
        soundPool = null;
        super.onPause();
    }

@Override
public void onLoadComplete(SoundPool sPool, int sid, int status) {
    Log.v("soundPool", "sid " + sid + " wczytany ze stanem " +
        status);

    final float currentVolume =
        ((float)aMgr.getStreamVolume(AudioManager.STREAM_MUSIC)) /
        ((float)aMgr.getStreamMaxVolume(AudioManager.STREAM_MUSIC));

    if(status != 0)
        return;
    if(sid == sid_background) {
        if(sPool.play(sid, currentVolume, currentVolume,
            PRIORITY, -1, 1.0f) == 0)
            Log.v("soundPool", "Proba odtworzenia dzwieku zakonczona niepowodzeniem");
    } else if(sid == sid_chimp) {
        queueSound(sid, 5000, currentVolume);
    } else if(sid == sid_rooster) {
        queueSound(sid, 6000, currentVolume);
    } else if(sid == sid_roar) {
        queueSound(sid, 12000, currentVolume);
    } else if(sid == sid_bark) {
        queueSound(sid, 7000, currentVolume);
    }
}

private void queueSound(final int sid, final long delay,
    final float volume)
{
    new Handler().postDelayed(new Runnable() {
        @Override
        public void run() {
            if(soundPool == null) return;
            if(soundPool.play(sid, volume, volume,
                PRIORITY, 0, 1.0f) == 0)
                Log.v("soundPool", "Nie udalo sie odtworzyc dzwieku (" + sid +
                    ")");
            queueSound(sid, delay, volume);
        }, delay);
    }
}

```

Struktura tego kodu nie jest złożona. Widzimy interfejs użytkownika zawierający jedną kontrolkę ToggleButton. Za jej pomocą będziemy wstrzymywać i wznowiać odtwarzanie aktywnych strumieni dźwiękowych. Po uruchomieniu aplikacji tworzymy obiekt klasy SoundPool i wczytujemy do niego pliki dźwiękowe. Jeżeli zostaną one prawidłowo wczytane, rozpoczynamy ich odtwarzanie. Plik z cykaniem świerszczy został zapętlony i odtwarza się w sposób ciągły, podczas gdy pozostałe dźwięki są odtwarzane po opóźnieniu, również w zapętleniu. Dzięki przyjęciu różnych czasów opóźnienia dźwięki nakładają się na siebie w rozmaitych konfiguracjach.

Do utworzenia klasy SoundPool wymagane są trzy parametry:

- Pierwszym z nich jest maksymalna liczba próbek, które obiekt SoundPool będzie równocześnie odtwarzał. Nie jest to równoznaczne z maksymalną liczbą próbek przechowywanych przez tę klasę.
- Drugi parametr definiuje strumień audio, w którym będą odtwarzane ścieżki. Typową wartością jest AudioManager.STREAM\_MUSIC, klasa SoundPool może jednak również obsługiwać alerty i dzwonki. Pełną listę strumieni dźwiękowych znajdziemy w dokumentacji klasy AudioManager.
- W momencie tworzenia obiektu SoundPool wartość parametru SRC\_QUALITY powinna wynosić 0.

Widzimy w kodzie kilka różnych metod load() klasy SoundPool. Najprostsza z nich wczytuje plik umieszczony w katalogu /res/raw w postaci zwykłego zasobu. Stosujemy tę metodę w przypadku pierwszych czterech plików dźwiękowych. Następnie pokazujemy, że możemy też wczytywać pliki dźwiękowe zamieszczone w katalogu /assets aplikacji. W tej metodzie load() wstawiamy również dodatkowe parametry definiujące pozycję i długość wczytywanego strumienia audio. W ten sposób moglibyśmy wykorzystywać pojedynczy plik zawierający wiele różnych próbek dźwiękowych, gdyż można wybrać jego fragment, którego w danej chwili należy użyć. W komentarzach pokazaliśmy, w jaki sposób możemy uzyskać dostęp do pliku audio przechowywanego na karcie SD. We wszystkich obecnie istniejących wersjach Androida (aż do wersji 3.0) parametr PRIORITY powinien posiadać wartość 1.

Postanowiliśmy w naszym przykładzie skorzystać z pewnych funkcji, dostępnych od wersji 2.2 Androida, konkretnie z interfejsu onLoadCompleteListener naszej aktywności oraz metod autoPause() i autoResume() wykorzystywanych w kodzie obsługującym przycisk.

W trakcie wczytywania plików dźwiękowych do obiektu SoundPool musimy poczekać, aż proces ten zostanie prawidłowo zakończony. W metodzie zwrotnej onLoadComplete() sprawdzamy postęp wczytywania i w zależności od rodzaju dźwięku włączamy jego odtwarzanie. Jeżeli mamy do czynienia z cykaniem świerszczy, włączamy zapętlanie (wartość -1 w piątym parametrze). W przypadku pozostałych plików tworzymy kolejkę dźwięków, odtwarzanych po krótkim odstępie czasowym. Długość przerwy jest podawana w milisekundach. Zwrócmy uwagę na ustawienia głośności. Bieżący poziom głośności poznajemy za pomocą klasy AudioManager. Za pomocą tej klasy określamy również maksymalny poziom głośności, dzięki czemu możemy obliczać wartość głośności dla metody play(), mieszczącą się w przedziale od 0 do 1 (wartość typu float). Metoda play() w rzeczywistości definiuje głośność oddzielnie dla kanalu lewego i prawego, jednak w naszym przypadku obydwa otrzymują taką samą wartość. Przypominamy, że wartość parametru PRIORITY powinna wynosić 1. Ostatni parametr metody play() służy do określania szybkości odtwarzania. Wartość ta powinna mieścić się w przedziale od 0.5 do 2, gdzie 1 definiuje standardową szybkość.

Metoda queueSound() wykorzystuje obiekt Handler przede wszystkim po to, aby skonfigurować zdarzenie, które nastąpi w niedalekiej przyszłości. Obiekt typu Runnable zostanie uruchomiony po upływie czasu przeznaczonego na przerwę. Następnie sprawdzamy, czy nadal istnieje obiekt SoundPool, ponownie odtwarzamy dźwięk i przygotowujemy go do kolejnego odtworzenia po upłynięciu dokładnie takiego samego czasu przerwy co uprzednio. Ponieważ wywołujemy metodę queueSound() zawierającą oddzielne identyfikatory plików oraz czasy przerwy, kolejność odtwarzania dźwięków wydawanych przez zwierzęta może się wydawać nieco losowa.

Po uruchomieniu tej aplikacji usłyszymy świerszcze, szympansa, koguta, psa i ryk (podejrzewamy, że niedźwiedzia). Usłyszymy, że świerszcze cykają nieprzerwanie, natomiast głosy pozostałych zwierząt pojawiają się i znikają. Jedna z najlepszych cech klasy SoundPool jest taka, że pozwala na odtwarzanie wielu dźwięków naraz bez wielkiego udziału ze strony programisty. Również urządzenie nie zostaje zbyt mocno obciążone, ponieważ dźwięki zostały zdekodowane już w trakcie wczytywania do pamięci i jedyne, co nam pozostaje, to przekazać je sprzętowi.

Jeśli klikniemy przycisk, odtwarzane głosy świerszczy oraz wszystkich aktualnie słyszanych zwierząt zostaną wstrzymane. Jednak metoda autoPause() nie zatrzymuje odtwarzania innych dźwięków. Po określonym czasie znowu usłyszymy głos jakiegoś zwierzaka (nie licząc świerszczy). Ponieważ stworzyliśmy kolejkę dźwięków, które mają być odtwarzane w przyszłości, będziemy je ciągle słyszeć. W rzeczywistości klasa SoundPool nie posiada żadnej metody pozwalającej na zatrzymywanie odtwarzanych i przewidzianych do odtworzenia dźwięków. Trzeba samodzielnie zaimplementować odpowiednią funkcję. Odgłosy świerszczy mogą zostać na nowo odtworzone wyłącznie po ponownym kliknięciu przycisku. Jednak nawet wtedy niekoniecznie zostaną odtworzone, ponieważ klasa SoundPool usuwa najstarszy dźwięk, aby zrobić miejsce dla najnowszego, w przypadku gdy zostanie osiągnięta maksymalna liczba odtwarzanych jednocześnie dźwięków.

## Odtwarzanie dźwięków za pomocą klasy JetPlayer

Klasa SoundPool spisuje się całkiem nieźle jako odtwarzacz, jednak ograniczenia pamięci mogą utrudniać wykonywanie niektórych zadań. Alternatywą pozwalającą na równoczesne odtwarzanie wielu plików jest klasa JetPlayer. Jest to bardzo wszechstronne, przystosowane głównie do gier narzędzie umożliwiające odtwarzanie wielu dźwięków oraz koordynujące odtwarzanie tych dźwięków z działaniami użytkownika. Wykorzystywane w tym przypadku dźwięki są definiowane za pomocą formatu MIDI (ang. *Musical Instrument Digital Interface* — cyfrowy interfejs instrumentów muzycznych).

Dźwięki przetwarzane przez klasę JetPlayer są tworzone za pomocą specjalnego narzędzia — **JETCreator**. Znajdziemy je w katalogu zestawu Android SDK, chociaż aby z niego korzystać, musimy zainstalować środowisko Python. Wynikowy plik o rozszerzeniu .jet może zostać wczytany do aplikacji, po czym następuje odtworzenie dźwięku. Cały proces jest dość zaawansowany i wykracza poza zakres tej książki, proponujemy więc zatrzeć do podrozdziału „Odnośniki”, aby znaleźć tam odniesienia do dalszych informacji.

## Odtwarzanie dźwięków odgrywanych w tle za pomocą klasy AsyncPlayer

Jeżeli zależy nam wyłącznie na odtwarzaniu jakiegoś pliku audio i nie chcemy obciążać bieżącego wątku, warto się zainteresować klasą AsyncPlayer. Źródło dźwiękowe jest przekazywane do tej klasy w postaci identyfikatora URI, dzięki czemu dostępne są zarówno pliki lokalne, jak i umieszczone w sieci. Klasa ta automatycznie tworzy wątek drugoplanowy, który uzyskuje dostęp do pliku audio i go odtwarza. Ponieważ jest to proces asynchroniczny, nie wiadomo dokładnie, w którym momencie odtwarzanie zostanie uruchomione. Analogicznie, nie będzie wiadomo, czy odtwarzanie dźwięku zostało zakończone lub czy w ogóle jeszcze trwa. Mozemy jednak wywołać metodę stop(), aby zatrzymać odtwarzanie dźwięku. Wywołanie metody play() przed zakończeniem odtwarzania poprzedniej ścieżki spowoduje jej zatrzymanie. Nowy plik dźwiękowy zostanie uruchomiony później, już po wyczyszczeniu i uporządkowaniu pamięci. Klasa AsyncPlayer jest bardzo prostą klasą, automatycznie tworzącą wątek drugoplanowy. Na listingu 19.5 zaprezentowano przykładową implementację tej klasy.

**Listing 19.5.** Odtwarzanie dźwięków za pomocą klasy AsyncPlayer

```
private static final String TAG = "AsyncPlayerDemo";
private AsyncPlayer mAsync = null;

[ ... ]

    mAsync = new AsyncPlayer(TAG);
    mAsync.play(this, Uri.parse("file://" + "/perry_ringtone.mp3"),
    false, AudioManager.STREAM_MUSIC);

[ ... ]

@Override
protected void onPause() {
    mAsync.stop();
    super.onPause();
}
```

---

**Niskopoziomowe odtwarzanie dźwięków za pomocą klasy AudioTrack**

Dotychczas zajmowaliśmy się dźwiękami pochodzącyimi z plików lokalnych oraz umieszczonych w sieci. Jeżeli chcemy zająć się odtwarzaniem dźwięku na nieco niższym poziomie, gdyż na przykład mamy do czynienia ze strumieniem danych audio, powinniśmy przyjrzeć się klasie `AudioTrack`. Oprócz standardowych metod `play()` i `pause()`, klasa `AudioTrack` posiada także kilka klas pozwalających na zapisywanie danych bezpośrednio dotyczących sprzętu odtwarzającego dźwięk. Dzięki tej klasie posiadamy największą kontrolę nad odtwarzaniem dźwięku, jest ona jednak o wiele bardziej skomplikowana niż klasy opisywane powyżej. Zaprezentujemy przykładową aplikację podczas omawiania klasy `AudioRecord`. Klasa `AudioRecord` bardzo przypomina klasę `AudioTrack`, więc w celu jej zrozumienia zapoznajmy się z informacjami dotyczącymi tej pierwszej.

**Osobliwości klasy MediaPlayer**

Ogólnie rzecz biorąc, klasa `MediaPlayer` wymaga dużego uporządkowania, dlatego aby odtwarzanie zostało właściwie przygotowane i uruchomione, musimy wywoływać operacje w ściśle określonej kolejności. Poniższa lista podsumowuje niektóre osobliwości związane ze stosowaniem interfejsów API multimedialnych:

- Po przypisaniu źródła danych do klasy `MediaPlayer` nie będzie łatwo zmienić je na inne — musimy utworzyć nową klasę `MediaPlayer` lub wywołać metodę `reset()`, służącą do przywrócenia pierwotnego stanu odtwarzacza.
- Po wywołaniu metody `prepare()` możemy wywołać metody `getCurrentPosition()`, `getDuration()` i `isPlaying()` w celu uzyskania bieżącego stanu odtwarzacza. Dostępne stają się również metody `setLooping()` oraz `setVolume()`.
- Po wywołaniu metody `start()` istnieje możliwość wywołania metod `pause()`, `stop()` i `seekTo()`.
- Każda klasa `MediaPlayer` tworzy nowy wątek, więc po zakończeniu pracy z odtwarzaczem musimy wywołać metodę `release()`. W przypadku klasy `VideoView` jest to przeprowadzane automatycznie, jednak jeżeli zdecydujemy się korzystać z klasy `MediaPlayer`, należy samemu wprowadzić tę metodę.

Na tym zakończymy dyskusję dotyczącą odtwarzania plików dźwiękowych. Zwrócimy teraz uwagę na odtwarzanie plików wideo. Jak zobaczymy, odniesienia do zawartości wideo są podobne do omawianych uprzednio odniesień do plików dźwiękowych.

## Odtwarzanie plików wideo

W tym punkcie zajmiemy się omówieniem zagadnień dotyczących odtwarzania plików wideo za pomocą środowiska SDK. W szczególności zaprezentujemy odtwarzanie plików wideo umieszczonych na serwerze internetowym oraz znajdujących się na karcie SD. Można sobie wyobrazić, że odtwarzanie plików wideo jest nieco bardziej skomplikowane niż uruchamianie plików dźwiękowych. Na szczęście środowisko Android SDK zawiera dodatkowe narzędzia wykonujące większość trudniejszej roboty.

### Uwaga!

Odtwarzanie plików wideo na emulatorze nie jest zbyt szczerliwym rozwiązaniem. Jeżeli plik wideo jest odtwarzany, to świetnie. W przeciwnym wypadku trzeba spróbować uruchomić aplikację na urządzeniu fizycznym. Emulacja polega wyłącznie na obliczeniach programowych, więc podczas odtwarzania plików wideo mogą się pojawić olbrzymie problemy, a samo uruchomienie aplikacji może dać nieprzewidziane rezultaty.

Odtwarzanie plików wideo jest bardziej złożonym procesem, ponieważ trzeba sobie poradzić nie tylko ze składową dźwiękową, ale również z wizualną. Aby nieco ułatwić sprawę, w Androidzie uwzględniono wyspecjalizowaną kontrolkę widoku, nazwaną `android.widget.VideoView`, która zajmuje się tworzeniem i uruchamianiem klasy `MediaPlayer`. Aby odtworzyć plik wideo, budujemy widżet `VideoView` w interfejsie użytkownika. Następnie wprowadzamy ścieżkę lub identyfikator URI pliku wideo i wywołujemy metodę `start()`. Na listingu 19.6 przedstawiamy kod odpowiedzialny za odtwarzanie plików wideo w Androidzie.

### **Listing 19.6.** Odtwarzanie pliku wideo za pomocą interfejsów API multimedialów

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <VideoView
        android:id="@+id/videoView"
        android:layout_width="200px"
        android:layout_height="200px" />
</LinearLayout>

// Jest to plik MainActivity.java
import android.app.Activity;
import android.net.Uri;
import android.os.Bundle;
import android.widget.MediaController;
import android.widget.VideoView;

public class MainActivity extends Activity {
    /** Wywoływanie podczas pierwszego utworzenia klasy. */
    @Override
```

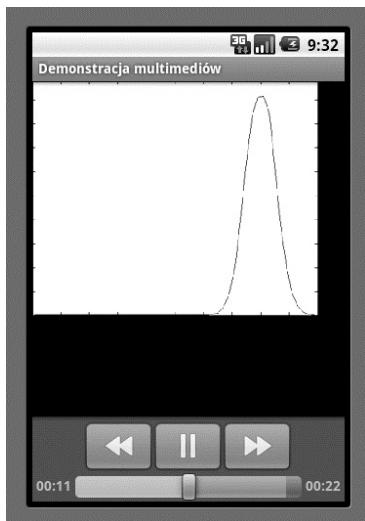
```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    this.setContentView(R.layout.main);

    VideoView videoView = (VideoView)this.findViewById(R.id.videoView);
    MediaController mc = new MediaController(this);
    videoView.setMediaController(mc);
    videoView.setVideoURI(Uri.parse(
        "http://www.androidbook.com/akc/filestorage/android/" +
        "documentfiles/3389/movie.mp4"));
/* videoView.setVideoPath(
    Environment.getExternalStoragePublicDirectory(
    Environment.DIRECTORY_MOVIES) +
    "/movie.mp4");
*/
    videoView.requestFocus();
    videoView.start();
}
}

```

Na listingu 19.6 pokazaliśmy, w jaki sposób można odtwarzać plik wideo dostępny w internecie pod adresem [www.androidbook.com/akc/filestorage/android/documentfiles/3389/movie.mp4](http://www.androidbook.com/akc/filestorage/android/documentfiles/3389/movie.mp4). Oznacza to, że aplikacja przetwarzająca ten kod będzie żądała uprawnień android.permission.INTERNET. Wszystkie funkcje odtwarzania pliku wideo znajdują się wewnątrz klasy VideoView. W rzeczywistości wystarczy podać odtwarzaczowi adres pliku wideo. Interfejs UI tej aplikacji jest pokazany na rysunku 19.4.



**Rysunek 19.4.** Interfejs UI odtwarzacza plików wideo z aktywnymi przyciskami kontroli

Po uruchomieniu aplikacji przez mniejszą więcej niż trzy sekundy w dolnej części ekranu będą widoczne przyciski, które następnie znikną. Kliknięcie gdziekolwiek w widoku odtwarzacza spowoduje ich ponowne wyświetlenie. Podczas odtwarzania pliku audio wystarczyły nam przyciski odpowiedzialne za odtwarzanie, wstrzymywanie i ponowne uruchamianie odtwarzania

pliku. Składowa graficzna przedstawiająca plik audio nie była nam do niczego potrzebna. Oczywiście w przypadku plików wideo potrzebujemy zarówno przycisków, jak i pojemnika, w którym możemy oglądać obraz. W naszym przykładzie korzystamy ze składowej VideoView do oglądania pliku. Jednak zamiast tworzyć własne kontrolki przycisków (w razie potrzeby mamy taką możliwość), tworzymy obiekt MediaController, zawierający predefiniowane przyciski. Jak zostało pokazane na rysunku 19.4 i listingu 19.6, konfigurujemy kontroler multimediiów poprzez wywołanie metody setMediaController() umożliwiającej odtwarzanie, wstrzymanie oraz przewijanie pliku wideo. Gdybyśmy chcieli wprowadzić własne przyciski, możemy wywołać metody start(), pause(), stopPlayback() i seekTo().

Nie zapominajmy, że w tym przykładzie ciągle korzystamy z klasy MediaPlayer — tylko jej nie widzimy. W rzeczywistości możemy „odtwarzać” pliki wideo bezpośrednio z poziomu tej klasy. Jeżeli wróćmy do przykładu umieszczonego na listingu 19.1, umieścimy plik wideo na karcie SD i wpiszemy ścieżkę dostępu do tego filmu jako wartość zmiennej AUDIO\_PATH, stwierdzimy, że dźwięk jest odtwarzany całkiem dobrze, chociaż nie widzimy obrazu.

Podczas gdy klasa MediaPlayer zawiera metodę setDataSource(), klasa VideoView jej nie posiada, korzysta za to z metod setVideoPath() lub setVideoURI(). Zakładając, że plik wideo znajduje się na karcie SD, zmieniamy kod z listingu 19.6, tak aby oznaczyć komentarzem wywołanie metody setVideoURI() oraz usunąć znaki komentarza z wywołania metody setVideoPath(), i wpisujemy poprawną ścieżkę do filmu. Po ponownym uruchomieniu aplikacji *usłyszmy i zobaczymy* odtwarzany plik wideo w widoku VideoView. Technicznie ten sam efekt moglibyśmy osiągnąć, wywołując metodę setVideoURI() w sposób przedstawiony poniżej:

```
videoView.setVideoURI(Uri.parse("file://" +
    Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_MOVIES) + "/movie.mp4"));
```

Być może Czytelnik zauważył, że w przeciwieństwie do klasy MediaPlayer klasa VideoView nie posiada metody pozwalającej na odczytywanie danych z deskryptora pliku. Można było także dostrzec, że klasa MediaPlayer posiada kilka metod pozwalających na dodanie obiektu SurfaceHolder (stanowi on odpowiednik okna roboczego dla obrazów lub plików wideo). Jeżeli musimy wyświetlać obraz wideo z prywatnego folderu aplikacji (na przykład odtwarzać plik znajdujący się w katalogu *data/data/...*), odpowiedniesze okażą się klasy MediaPlayer i SurfaceHolder niż VideoView. Jedną z nadających się do tego celu metod klasy MediaPlayer jest create(Context context, Uri uri, SurfaceHolder holder), natomiast drugą — setDisplay(holder).

Przejdźmy teraz do procesu rejestracji multimediiów.

## Rejestrowanie multimediiów

Jak pokazaliśmy, w systemie Android istnieje wiele sposobów odtwarzania multimediiów. W przypadku rejestrowania danych multimedialnych posiadamy nieco mniej możliwości. Główną klasą roboczą służącą do zapisywania takich danych jest MediaRecorder, która jest stosowana zarówno do plików audio, jak i wideo. W tym podrozdziale zaprezentujemy sposób korzystania z tej klasy do rejestrowania obydwu rodzajów danych. Drugą klasą, pozwalającą na nagrywanie dźwięku, jest klasa AudioRecord. Aby pokazać, w jaki sposób ją wykorzystywać, utworzymy osobny przykładowy projekt. Czasami nie musimy tworzyć w całości kodu, skoro istnieje aplikacja, która wykonuje zamierzone czynności. Pokażemy więc, w jaki sposób można uruchomić intencję służącą do rejestrowania dźwięku, a także jak fotografować za pomocą aplikacji Camera.

## Analiza procesu rejestracji dźwięku za pomocą klasy MediaRecorder

Szkielet multimedialny w Androidzie obsługuje proces nagrywania dźwięku. Jedną z klas służących do rejestrowania dźwięku jest `android.media.MediaRecorder`. W tym punkcie pożajemy sposób budowania aplikacji pozwalającej na nagranie i późniejsze odtworzenie treści dźwiękowej. Interfejs użytkownika tej aplikacji został pokazany na rysunku 19.5.



Rysunek 19.5. Interfejs UI przykładowej aplikacji nagrywającej dźwięk

Jak widać na rysunku 19.5, aplikacja posiada cztery przyciski: dwa służą do kontroli nagrywania, a dwa pozostałe — do odtwarzania i zatrzymywania odtwarzania nagranej treści. Na listingu 19.7 zawarto treść pliku układu graficznego oraz kod klasy aktywności tego interfejsu.

**Listing 19.7.** Nagrywanie i odtwarzanie dźwięku w Androidzie

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/record.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:id="@+id/beginBtn" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Rozpocznij nagrywanie"
        android:onClick="doClick"/>
    <Button android:id="@+id/stopBtn" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Zatrzymaj nagrywanie"
        android:onClick="doClick"/>
    <Button android:id="@+id/playRecordingBtn" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Odtwarzaj nagranie"
        android:onClick="doClick"/>
    <Button android:id="@+id/stopPlayingRecordingBtn" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Zatrzymaj odtwarzanie"
        android:onClick="doClick"/>
```

```
</LinearLayout>

// RecorderActivity.java
import java.io.File;
import android.app.Activity;
import android.media.MediaPlayer;
import android.media.MediaRecorder;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;

public class RecorderActivity extends Activity {
    private MediaPlayer mediaPlayer;
    private MediaRecorder recorder;
    private String OUTPUT_FILE;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.record);

        OUTPUT_FILE = Environment.getExternalStorageDirectory() +
                      "/recordaudio3.3gpp";
    }

    public void doClick(View view) {
        switch(view.getId()) {
            case R.id.beginBtn:
                try {
                    beginRecording();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
            case R.id.stopBtn:
                try {
                    stopRecording();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
            case R.id.playRecordingBtn:
                try {
                    playRecording();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
            case R.id.stopPlayingRecordingBtn:
                try {
                    stopPlayingRecording();
                } catch (Exception e) {
                    e.printStackTrace();
                }
                break;
        }
    }
}
```

```
        }
    }

private void beginRecording() throws Exception {
    killMediaRecorder();

    File outFile = new File(OUTPUT_FILE);

    if(outFile.exists()) {
        outFile.delete();
    }
    recorder = new MediaRecorder();
    recorder.set AudioSource(MediaRecorder.AudioSource.MIC);
    recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    recorder.setOutputFile(OUTPUT_FILE);
    recorder.prepare();
    recorder.start();
}

private void stopRecording() throws Exception {
    if (recorder != null) {
        recorder.stop();
    }
}

private void killMediaRecorder() {
    if (recorder != null) {
        recorder.release();
    }
}

private void killMediaPlayer() {
    if (mediaPlayer != null) {
        try {
            mediaPlayer.release();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

private void playRecording() throws Exception {
    killMediaPlayer();

    mediaPlayer = new MediaPlayer();
    mediaPlayer.setDataSource(OUTPUT_FILE);

    mediaPlayer.prepare();
    mediaPlayer.start();
}

private void stopPlayingRecording() throws Exception {
    if(mediaPlayer != null) {
        mediaPlayer.stop();
    }
}
```

```

    }

@Override
protected void onDestroy() {
    super.onDestroy();

    killMediaRecorder();
    killMediaPlayer();
}
}

```

---

Zanim zajmiemy się omawianiem listingu 19.7, musimy umieścić w pliku manifeście następujący wpis o uprawnieniu, aby móc rejestrować dźwięk:

```
<uses-permission android:name="android.permission.RECORD_AUDIO" />
```

W podroziale dotyczącym kart SD wspomnialiśmy również, że w przypadku wartości parametru `minSdkVersion` równej lub większej od 4 musimy dodać znacznik `uses-permission` dla klasy `"android.permission.WRITE_EXTERNAL_STORAGE"`. Oczywiście, jeżeli chcemy wypróbować funkcję rejestrowania dźwięku na emulatorze, musimy podłączyć mikrofon do stacji roboczej.

Jeśli przyjrzymy się metodzie `onCreate()` z listingu 19.7, zauważymy, że jedyną niezbędną czynnością jest wprowadzenie ścieżki do wyjściowego pliku dźwiękowego. Nasza metoda `onClick()` wykorzystuje standardowy wzorzec przełączania pomiędzy przyciskami, trzeba tylko przywołać funkcję wykonującą właściwe zadanie. Metoda `beginRecording()` obsługuje proces rejestracji dźwięku. Aby nagrywać dźwięk, musimy utworzyć wystąpienie klasy `MediaRecorder` i skonfigurować źródło dźwięku, format pliku wynikowego, koder audio oraz ścieżkę pliku wyjściowego.

W wersjach środowiska Android SDK starszych od 1.6 jedynym obsługiwanym źródłem dźwięku był mikrofon. Od czasu wydania wersji Android SDK 1.6 dostępne są trzy dodatkowe źródła dźwięku, wszystkie związane z rozmowami telefonicznymi. Możemy rejestrować całą rozmowę (`MediaRecorder.AudioSource.VOICE_CALL`), transmisję prowadzoną wyłącznie w górę sieci (`MediaRecorder.AudioSource.VOICE_UPLINK`) lub transmisję prowadzoną wyłącznie w dół sieci (`MediaRecorder.AudioSource.VOICE_DOWNLINK`). Transmisją prowadzoną w górę sieci może być głos użytkownika telefonu. Transmisją prowadzoną w dół określa się zazwyczaj dźwięki dochodzące z drugiego końca połączenia.

Wraz z wersją 2.1 Androida dodano obsługę dwóch nowych źródeł dźwięku: `CAMCORDER` oraz `VOICE_RECOGNITION`. Źródło `CAMCORDER` może być mikrofonem związanym z aparatem, w przeciwnym wypadku po wybraniu tej opcji będzie stosowany domyślny mikrofon urządzenia. Z kolei po wyborze trybu `VOICE_RECOGNITION` system wykorzystuje mikrofon przystosowany do rozpoznawania mowy, a w przypadku jego braku — znowu główny mikrofon urządzenia. Poprzez stwierdzenie „przystosowany do rozpoznawania mowy” mamy na myśli urządzenie, które zupełnie nie przetwarza strumienia audio ani nie wprowadza żadnych modyfikacji dźwięku na drodze pomiędzy mikrofonem a aplikacją. Przykładem urządzeń modyfikujących rejestrowany dźwięk są niektóre urządzenia firmy HTC, których mikrofony wyposażono w funkcję automatycznej regulacji wzmacnienia (ang. *Automatic Gain Control* — AGC). Wykorzystywanie tego urządzenia w procesie rozpoznawania mowy może stanowić problem. Źródło `VOICE_RECOGNITION` pomija tego typu dodatkowe przetwarzanie, dzięki czemu proces przetwarzania mowy okazuje się skuteczniejszy.

Najpopularniejszym formatem wyjściowym dźwięku jest format 3GPP (ang. *3rd Generation Partnership Project* — partnerski projekt trzeciej generacji). W wersjach Androida starszych

od 2.3.3 (Gingerbread) wartością kodowania musi być AMR\_NB, czyli wąskopasmowy koder audio AMR (ang. *Adaptive Multi-Rate*), ponieważ jest to jedyny obsługiwany format kodera dźwięku. W wersji 2.3.3 Androida wprowadzono również kodery AMR\_WB (szerokopasmowy) oraz AAC (ang. *Advance Audio Coding* — zaawansowane kodowanie dźwięku). W naszym przykładzie zarejestrowany dźwięk jest zapisany na karcie SD w pliku *recordoutput.3gpp*. Na listingu 19.7 założyliśmy, że utworzyliśmy obraz karty SD i powiązaliśmy go z emulatorem. Informacje potrzebne do przeprowadzenia tej czynności można znaleźć w podrozdziale „Wykorzystywanie kart SD”.

Dla klasy MediaRecorder dostępne są dodatkowe metody, które mogą się okazać przydatne. Metody *setMaxDuration(int length\_in\_ms)* oraz *setMaxFileSize(long length\_in\_bytes)* są stosowane do ograniczania długości i rozmiaru nagrań dźwiękowych. Po osiągnięciu limitu maksymalnej długości nagrania w milisekundach lub maksymalnego rozmiaru w bajtach rejestracja dźwięku zostanie zakończona. Obydwie metody zostały zaimplementowane w wersji 1.5 środowiska SDK, zatem są w zasadzie obsługiwane przez wszystkie rodzaje telefonów umożliwiające rejestrację dźwięku.

## Rejestracja dźwięków za pomocą klasy **AudioRecord**

Na razie pokazaliśmy, w jaki sposób można rejestrować dźwięki bezpośrednio do pliku. A jak należy postąpić w przypadku, gdy chcemy wprowadzić przetwarzanie danych dźwiękowych przed ich zapisaniem? A może nawet nie trzeba wysyłać rejestrowanych danych do pliku? Klasa **AudioRecord** spełnia wszystkie tego rodzaju wymagania. W trakcie tworzenia obiektu **AudioRecord** system przestawia się na zapisywanie danych audio do wewnętrznego bufora tej klasy, a następnie aplikacja może wprowadzać dowolne modyfikacje do rejestrowanych dźwięków. Na listingu 19.8 widzimy aktywność odczytującą i przetwarzającą dane audio za pomocą klasy **AudioRecord**. Nie jest nam tu potrzebny interfejs użytkownika, ponieważ wszystkie komunikaty będą wyświetlane w oknie *LogCat*. Plik *AndroidManifest.xml* również został pominięty, musimy jednak w nim dodać uprawnienie *android.permission.RECORD\_AUDIO*.

---

**Listing 19.8.** Rejestrowanie nieprzetwarzanych danych audio za pomocą klasy **AudioRecord**

---

```
import android.app.Activity;
import android.media.AudioFormat;
import android.media.AudioRecord;
import android.media.MediaRecorder;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends Activity {
    protected static final String TAG = "AudioRecord";
    private int mAudioBufferSize;
    private int mAudioBufferSampleSize;
    private AudioRecord mAudioRecord;
    private boolean inRecordMode = false;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        initAudioRecord();
    }

    @Override
```

```
public void onResume() {
    super.onResume();
    Log.v(TAG, "Wznawianie... ");
    inRecordMode = true;
    Thread t = new Thread(new Runnable() {

        @Override
        public void run() {
            getSamples();
        }
    });
    t.start();
}

protected void onPause() {
    Log.v(TAG, "Wstrzymywanie... ");
    inRecordMode = false;
    super.onPause();
}

@Override
protected void onDestroy() {
    Log.v(TAG, "Konczenie... ");
    if(mAudioRecord != null) {
        mAudioRecord.release();
        Log.v(TAG, "Zwolniono klase AudioRecord");
    }
    super.onDestroy();
}

private void initAudioRecord() {
    try {
        int sampleRate = 8000;
        int channelConfig = AudioFormat.CHANNEL_IN_MONO;
        int audioFormat = AudioFormat.ENCODING_PCM_16BIT;
        mAudioBufferSize =
            2 * AudioRecord.getMinBufferSize(sampleRate,
                channelConfig, audioFormat);
        mAudioBufferSampleSize = mAudioBufferSize / 2;
        mAudioRecord = new AudioRecord(
            MediaRecorder.AudioSource.MIC,
            sampleRate,
            channelConfig,
            audioFormat,
            mAudioBufferSize);
        Log.v(TAG, "Ustanawianie obiektu AudioRecord udane. Rozmiar buforu = " +
            mAudioBufferSize);
        Log.v(TAG, " Rozmiar buforu testowego = " +
            mAudioBufferSampleSize);
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    }

    int audioRecordState = mAudioRecord.getState();
    if(audioRecordState != AudioRecord.STATE_INITIALIZED) {
        Log.e(TAG, "Obiekt AudioRecord zostal niewlasciwie zainicjalizowany");
    }
}
```

```
        finish();
    }
    else {
        Log.v(TAG, "Obiekt AudioRecord został zainicjalizowany");
    }
}

private void getSamples() {
    if(mAudioRecord == null) return;

    short[] audioBuffer = new short[mAudioBufferSize];

    mAudioRecord.startRecording();

    int audioRecordingState = mAudioRecord.getRecordingState();
    if(audioRecordingState != AudioRecord.RECORDSTATE_RECORDING) {
        Log.e(TAG, "Obiekt AudioRecord nie rejestruje dźwięku");
        finish();
    }
    else {
        Log.v(TAG, "Obiekt AudioRecord rozpoczęł rejestrowanie dźwięku...");
    }

    while(inRecordMode) {
        int samplesRead = mAudioRecord.read(
            audioBuffer, 0, mAudioBufferSize);
        Log.v(TAG, "Uzyskano próbki: " + samplesRead);
        Log.v(TAG, "Wartości kilku pierwszych probek: " +
            audioBuffer[0] + ", " +
            audioBuffer[1] + ", " +
            audioBuffer[2] + ", " +
            audioBuffer[3] + ", " +
            audioBuffer[4] + ", " +
            audioBuffer[5] + ", " +
            audioBuffer[6] + ", " +
            audioBuffer[7] + ", " +
            audioBuffer[8] + ", " +
            audioBuffer[9] + ", ");
    }
}

mAudioRecord.stop();
Log.v(TAG, "Obiekt AudioRecord zakończył proces rejestrowania");
}
```

---

Nasza przykładowa aplikacja jest raczej nieskomplikowana. Rozpoczynamy od zainicjalizowania obiektu AudioRecord. Musimy w tym celu wybrać źródło dźwięku, częstotliwość próbkowania, konfigurację kanałów (mono, stereo, lewy, prawy itd.), format kodowania oraz rozmiar buforu wewnętrznego. Jeśli chodzi o źródło dźwięku, mamy do dyspozycji zestaw opcji zdefiniowanych w klasie MediaRecorder.AudioSource. Musimy tylko wspomnieć, że nie wszystkie urządzenia posiadają zaimplementowane źródło VOICE\_CALL, ponieważ w rzeczywistości wykorzystuje ono dwa urządzenia wejściowe. Wśród częstotliwości próbkowania powinniśmy wybrać jedną ze standardowych wartości: 8000, 16000, 44100, 22050 lub 11025 Hz.

Konfiguracja kanałów powinna być wybrana spośród wartości CHANNEL\* opisanych w klasie `AudioFormat`. Wśród formatów kodowania mamy do dyspozycji `ENCODING_PCM_8BIT` i `ENCODING_PCM_16BIT`. Zwróciły uwagę, że od wyboru tej wartości zależy jakość rejestrowanego dźwięku. Jeżeli nie potrzebujemy 16-bitowej dokładności, wybierzmy kodowanie 8-bitowe — zaoszczędzimy nieco pamięci i zyskamy na wydajności. W dokumentacji znaleźćśmy wzmiankę, że jedynie próbkowanie 44100 Hz będzie działać na wszystkich urządzeniach, ale — jak na ironię — emulator obsługuje wyłącznie wartości 8000 Hz, `CHANNEL_IN_MONO` oraz `ENCODING_PCM_8BIT`.

Klasa `AudioRecord` posiada statyczną metodę pomocniczą, noszącą nazwę `getMinBufferSize()`, która pobierze wszystkie zdefiniowane przez nas parametry iwróci najmniejszy dopuszczalny bufor, wystarczający do poprawnego inicjalizowania obiektu `AudioRecord` w danych warunkach. Nie mamy bezpośredniego dostępu do tego buforu, jednak klasa `AudioRecord` zajmuje wystarczająco wiele miejsca, aby przechowywać dane audio i jednocześnie przetwarzać wcześniejsze dźwięki. Można skorzystać z minimalnego rozmiaru buforu albo nieco zwiększyć jego pojemność. Na pewno nie powinniśmy ustanawiać mniejszego buforu od wartości zalecanej przez metodę pomocniczą. W naszym przykładzie ustanowiliśmy bufor dwukrotnie większy od zalecanego minimum. Jeżeli parametry nie będą akceptowane przez klasę `AudioRecord`, zostanie wyświetlony wyjątek `IllegalArgumentException`. Jeżeli na przykład wprowadzimy wartość częstotliwości próbkowania nieobsługiwany przez urządzenie, zobaczymy ten wyjątek. Niestety, nie ma łatwego sposobu, aby uzyskać listę obsługiwanych częstotliwości próbkowania, zatem naszym jedynym rozwiązaniem jest metoda prób i błędów; jeżeli dana częstotliwość powoduje wyświetlenie wyjątku, musimy sprawdzić inną.

Na samym końcu metody inicjalizującej sprawdzamy jeszcze, czy obiekt `AudioRecord` został poprawnie utworzony, i już można rozpocząć rejestrowanie dźwięku.

Postanowiliśmy włączyć próbkowanie w metodzie `onResume()`, a wyłączyć je w metodzie `onPause()` naszej aktywności. Nie chcemy łączyć głównego wątku z procesem próbkowania, zatem tworzymy osobny wątek, służący wyłącznie do obsługi tej czynności. Wprowadzamy również wartość logiczną (`inRecordMode`), dzięki czemu można zaprogramować przerwanie próbkowania przez wątek. Wewnątrz metody `getSamples()` tworzymy własny bufor danych dźwiękowych. Jak już wspomnialiśmy, nie mamy bezpośredniego dostępu do wewnętrznego buforu klasy `AudioRecord`, zatem odczytujemy nasz własny bufor. Zwróciły uwagę, że rozmiar tego buforu został zadeklarowany za pomocą `audioBufferSize`, a nie `audioBufferSize`. Odczytujemy wyłącznie rozmiar próbki, gdyż tylko ta informacja jest potrzebna naszemu buforowi. Uruchamiamy rejestrację w obiekcie `AudioRecord`, sprawdzamy, czy stan zmienił się na `RECORDING`, i zaczynamy zapętać odczyty. Są to odczyty blokujące, na szczęście znajdujemy się w osobnym wątku, więc nie ma żadnego problemu. Gdy obiekt `AudioRecord` zaczyna zbliżać się do zdefiniowanego rozmiaru danych, przekazuje odczytane wyniki, dzięki czemu można w dalszym ciągu przetwarzać tę próbę audio.

W międzyczasie klasa `AudioRecord` będzie zbierać dodatkowe informacje o dźwięku na wypadek następnego wywołania odczytu. Mamy ograniczony czas na przetwarzanie danych, zanim bufor klasy `AudioRecord` zostanie ponownie zapełniony, zatem musimy zachować tu ostrożność i nie pobierać zbyt dużej ilości danych. W zależności od ich przeznaczenia możemy po prostu zatrzymać proces rejestracji i później go ponowić. W naszym przykładzie wykorzystujemy po prostu okno `LogCat` do poinformowania o zarejestrowaniu próbek i wyświetlamy dziesięć pierwszych wartości. W trakcie rejestracji nagrywajmy różne dźwięki, aby przekonać się, że wartości wyświetlane w oknie `LogCat` ulegają zmianie.

Zapętlenie odczytu trwa do czasu, aż parametr `inRecordMode` osiągnie wartość `false`, co nastąpi w momencie ukrycia lub zamknięcia aplikacji.

W trakcie uważnego przeglądania dokumentacji klasy `AudioRecord` możemy natrafić na zwrotne interfejsy. Pozwalają one na konfigurowanie obiektów nasłuchujących dotyczących albo osiągnięcia znacznika wewnętrz strumienia audio, albo okazjonalnego uruchomienia metod zwrotnych. Na listingu 19.9 zmodyfikowaliśmy powyższy przykład poprzez dodanie odpowiednich instrukcji. Pełny kod źródłowy tego projektu znajdziemy na naszej stronie WWW.

**Listing 19.9.** Rejestrowanie nieskompresowanych danych audio za pomocą klasy `AudioRecord` i metod zwrotnych

---

```
// Ten kod znajduje się wewnątrz klasy aktywności
public OnRecordPositionUpdateListener mListener = new
OnRecordPositionUpdateListener() {

    public void onPeriodicNotification(AudioRecord recorder) {
        Log.v(TAG, "w metodzie onPeriodicNotification");
    }

    public void onMarkerReached(AudioRecord recorder) {
        Log.v(TAG, "w metodzie onMarkerReached");
        inRecordMode = false;
    }
};

// Poniższe instrukcje znajdują się wewnątrz metody initAudioRecord() po
// utworzeniu obiektu mAudioRecord, lecz jeszcze przed sprawdzeniem
// jego stanu.
mAudioRecord.setNotificationMarkerPosition(10000);
mAudioRecord.setPositionNotificationPeriod(1000);
mAudioRecord.setRecordPositionUpdateListener(mListener);
```

---

Zwróćmy uwagę, że obiekt nasłuchujący posiada dwie oddzielne metody zwrotnie. Pierwsza z nich jest wywoływana co 1000 ramek, co zostało zdefiniowane w metodzie inicjalizującej. Ten licznik ramek jest niezależny od rozmiaru buforu. Nawet jeśli jednorazowo będziemy odczytywać 1600 ramek, metoda ta będzie wywoływana co 1000 ramek. Zatem w tym przypadku wspomniana metoda będzie przywołana dwukrotnie w czasie jednej pętli. Druga metoda jest wywoływana, gdy osiągniemy całkowitą liczbę ramek. W naszym przykładzie zdefiniowaliśmy tę wartość jako 10 000 ramek; po jej osiągnięciu rejestrowanie dźwięku zostaje zakończone poprzez wprowadzenie wartości logicznej `false`. Gdybyśmy jedynie wypisali komunikat o osiągnięciu tej wartości i nie wyłączyli procesu rejestracji, wartość ta nie pojawiłaby się już ponownie, niezależnie od liczby ramek odczytanych w przyszłości. Jest to znacznik relatywny od momentu wywołania metody `startRecording()` w obiekcie `AudioRecord`.

## Analiza procesu rejestracji wideo

Od wersji 1.5 środowiska Android SDK wprowadzono możliwość rejestrowania obrazu wideo za pomocą struktury multimedialnej. Zasada działania jest podobna do procesu rejestracji dźwięku — i rzeczywiście, nagrany obraz wideo przeważnie posiada również ścieżkę dźwiękową. W przypadku rejestracji wideo istnieje jednak jedna różnica. Począwszy od środowiska Android

SDK 1.6, wymagane jest, aby na obiekcie Surface był tworzony podgląd rejestrowanego obrazu. W prostych aplikacjach nie stanowi to problemu, ponieważ użytkownik będzie chciał widać podgląd tego, co nagrywa. W bardziej złożonych programach może się pojawić problem. Nawet jeżeli aplikacja nie musi pokazywać użytkownikowi podglądu wideo w trakcie nagrywania, obiekt Surface nadal musi być obecny, gdyż takie jest wymaganie klasy camera. Spodziewamy się, że wymóg ten zostanie w przyszłych wersjach środowiska SDK złagodzony, tak aby aplikacje mogły pracować bezpośrednio na buforach wideo bez konieczności kopowania ich do interfejsu UI, na razie jednak musimy korzystać z obiektu Surface i pokażemy, jak należy to robić. Omawiany przykładowy kod jest dosyć długi, zatem podzieliśmy go na części, dzięki czemu łatwiej nam będzie omawiać jego poszczególne zagadnienia. Najprawdopodobniej Czytelnik zechce pobrać ten projekt z naszej strony i zimportować go do środowiska Eclipse. Niezbędne instrukcje znajdą się na końcu rozdziału, w podrozdziale „Odkrycia”. Rozpoczniemy od ukazania na listingu 19.10 wykorzystanego układu graficznego.

**Listing 19.10.** Układ graficzny aplikacji rejestrującej dane wideo

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout-land/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:orientation="horizontal" >
<LinearLayout
    android:orientation="vertical" android:layout_width="wrap_content"
    android:layout_height="wrap_content">

<Button android:id="@+id/initBtn"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Inicjalizacja rejestratora" android:onClick="doClick"
    android:enabled="false" />

<Button android:id="@+id/beginBtn"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Rozpocznij rejestrowanie" android:onClick="doClick"
    android:enabled="false" />

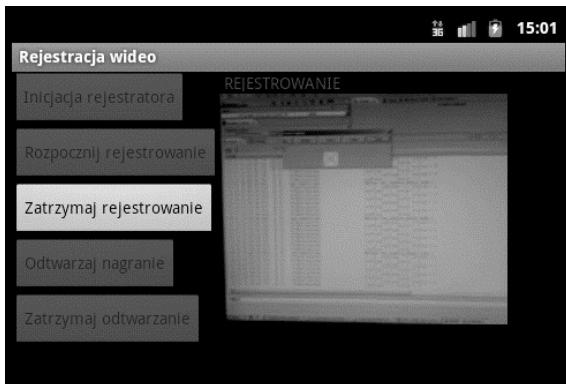
<Button android:id="@+id/stopBtn"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Zatrzymaj rejestrowanie" android:onClick="doClick" />

<Button android:id="@+id/playRecordingBtn"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Odtwarzaj nagranie" android:onClick="doClick" />

<Button android:id="@+id/stopPlayingRecordingBtn"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Zatrzymaj odtwarzanie" android:onClick="doClick" />
</LinearLayout>
<LinearLayout android:orientation="vertical"
    android:layout_width="fill_parent" android:layout_height="fill_parent" >
<TextView android:id="@+id/recording" android:text=" "
    android:textColor="#FF0000"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
<VideoView android:id="@+id/videoView"
```

```
    android:layout_width="250dip" android:layout_height="200dip" />
</LinearLayout>
</LinearLayout>
```

Na rysunku 19.6 widzimy powyższy układ graficzny w pełnej okazałości. Zrzut ekranu został wykonany na fizycznym urządzeniu podczas rejestrowania obrazu wideo, przedstawiającego okno robocze środowiska Eclipse na stacji roboczej.



**Rysunek 19.6.** Interfejs rejestratora wideo

Układ ten składa się z dwóch umieszczonych obok siebie pojemników `LinearLayout`, znajdujących się w nadzędnym kontenerze `LinearLayout`. Z lewej strony widzimy pięć przycisków, które nasza aplikacja będzie nam udostępniała i wyłączała w trakcie analizowania przykładu. Z prawej strony został umieszczony główny widok `VideoView`, nad nim zaś znajduje się napis `REJESTROWANIE`, który zostaje wyświetlony w trakcie nagrywania obrazu. Jak już się Czytelnik prawdopodobnie domyślił, wymuszamy ułożenie aplikacji w trybie krajobrazowym poprzez umieszczenie atrybutu `android:screenOrientation="landscape"` w znaczniku `<activity>` pliku `AndroidManifest.xml`. Rozpoczniemy analizę aplikacji od klasy `MainActivity`, zaprezentowanej na listingu 19.11.

**Listing. 19.11.** Główna aktywność rejestratora wideo

```
public class MainActivity extends Activity implements
    SurfaceHolder.Callback, OnInfoListener, OnErrorListener {

    private static final String TAG = "RecordVideo";
    private MediaRecorder mRecorder = null;
    private String mOutputFileName;
    private VideoView mVideoView = null;
    private SurfaceHolder mHolder = null;
    private Button mInitBtn = null;
    private Button mStartBtn = null;
    private Button mStopBtn = null;
    private Button mPlayBtn = null;
    private Button mStopPlayBtn = null;
    private Camera mCamera = null;
    private TextView mRecordingMsg = null;

    /** Tworzone podczas pierwszego wywołania aktywności. */
```

```

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.v(TAG, "w metodzie onCreate");
    setContentView(R.layout.main);

    mInitBtn = (Button) findViewById(R.id.initBtn);
    mStartBtn = (Button) findViewById(R.id.beginBtn);
    mStopBtn = (Button) findViewById(R.id.stopBtn);
    mPlayBtn = (Button) findViewById(R.id.playRecordingBtn);
    mStopPlayBtn = (Button) findViewById(R.id.stopPlayingRecordingBtn);
    mRecordingMsg = (TextView) findViewById(R.id.recording);

    mVideoView = (VideoView)this.findViewById(R.id.videoView);
}
// Pozostała część klasy została umieszczona na następnych listingach.
}

```

---

Korzystamy w tej aplikacji ze standardowej aktywności, jednak implementujemy w niej również trzy interfejsy. Pierwszy z nich, `SurfaceHolder.Callback`, służy do otrzymywania informacji o tym, kiedy obiekt `Surface` będzie przygotowany do wyświetlania obrazu wideo. Obiekt `Surface` w naszym przypadku pochodzi z klasy `VideoView`. Chcemy być również informowani o wszelkich komunikatach wychodzących z klasy `MediaRecorder`, dlatego implementujemy dwa pozostałe interfejsy: `OnInfoListener` oraz `OnErrorListener`. Wkrótce zajmiemy się omówieniem metod wspomnianych interfejsów.

Nasza aktywność zawiera kilka pól członkowskich, które będą potrzebne później. Część z nich inicjalizujemy w metodzie `onCreate()`. Na razie zamieściliśmy jedynie komentarz wskazujący, gdzie zostanie umieszczona reszta klasy `MainActivity`. Wspomniane metody klasy zostaną указанie na kolejnych listingach, począwszy od listingu 19.12, na którym prezentujemy standardowe metody `onResume()` i `onPause()`.

#### **Listing 19.12.** Kod obsługujący wstrzymywanie i ponawianie pracy rejestratora wideo

---

```

@Override
protected void onResume() {
    Log.v(TAG, "w metodzie onResume");
    super.onResume();
    mInitBtn.setEnabled(false);
    mStartBtn.setEnabled(false);
    mStopBtn.setEnabled(false);
    mPlayBtn.setEnabled(false);
    mStopPlayBtn.setEnabled(false);
    if(!initCamera())
        finish();
}

@Override
protected void onPause() {
    Log.v(TAG, "w metodzie onPause");
    super.onPause();
    releaseRecorder();
    releaseCamera();
}

```

---

**Uwaga!**

Na listingu 19.12 przedstawiliśmy metody będące częścią klasy MainActivity; umieściliśmy je na oddzielnych listingach jedynie w celu lepszego wyjaśnienia ich działania. Taka sama zasada dotyczy pozostałych listingów łączących się z kodem aplikacji Rejestrator Wideo.

Jak widać, w prezentowanym kodzie umieściliśmy całkowicie standardowe metody. W metodzie onResume() ustawiamy po prostu stan początkowy przycisków, a następnie uruchamiamy kamerę (wkrótce zapoznamy się z tą metodą). W metodzie onPause() musimy zwolnić zarówno obiekt MediaRecorder, jak i Camera. W ten sposób po każdym schowaniu aplikacji rejestrowanie będzie zatrzymane, a kamera zostanie zwolniona do dyspozycji innych programów. Jeżeli użytkownik powróci do naszej aplikacji, jej działanie zostanie wznowione i znów będzie można rejestrować obraz wideo. Na listingu 19.13 pokazujemy kod inicjalizujący kamerę, metody zwrotne interfejsu SurfaceHolder.Callback, a także metody obsługujące zwalnianie obiektów Camera i MediaRecorder.

**Listing 19.13.** Metody initCamera() oraz zwalniające kamerę

```
private boolean initCamera() {
    try {
        mCamera = Camera.open();
        Camera.Parameters camParams = mCamera.getParameters();
        mCamera.lock();
        //mCamera.setDisplayOrientation(90);
        // Można również ustawić tutaj inne parametry i zastosować:
        //mCamera.setParameters(camParams);

        mHolder = mVideoView.getHolder();
        mHolder.addCallback(this);
        mHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }
    catch(RuntimeException re) {
        Log.v(TAG, "Nie mozna zainicjalizowac obiektu Camera");
        re.printStackTrace();
        return false;
    }
    return true;
}

@Override
public void surfaceCreated(SurfaceHolder holder) {
    Log.v(TAG, "w metodzie surfaceCreated");

    try {
        mCamera.setPreviewDisplay(mHolder);
        mCamera.startPreview();
    } catch (IOException e) {
        Log.v(TAG, "Nie mozna uruchomic podgladu");
        e.printStackTrace();
    }
    mInitBtn.setEnabled(true);
}

@Override
public void surfaceDestroyed(SurfaceHolder holder) {
```

```

        Log.v(TAG, "w metodzie surfaceDestroyed");
    }

@Override
public void surfaceChanged(SurfaceHolder holder, int format, int width,
    int height) {
    Log.v(TAG, "surfaceChanged: Szerokosc x Wysokosc = " + width + "x" + height);
}

private void releaseRecorder() {
    if(mRecorder != null) {
        mRecorder.release();
        mRecorder = null;
    }
}

private void releaseCamera() {
    if(mCamera != null) {
        try {
            mCamera.reconnect();
        } catch (IOException e) {
            e.printStackTrace();
        }
        mCamera.release();
        mCamera = null;
    }
}

```

Metoda `initCamera()` służy do skonfigurowania połączenia z kamerą urządzenia. Od tego miejsca zaczyna się praca aplikacji. W tej aplikacji korzystamy z domyślnych parametrów kamery, możemy jednak w łatwy sposób dostać się do bieżących parametrów kamery, zaktualizować je i zapisać. W zaznaczonym jako komentarz fragmencie kodu widać, gdzie można zmieniać wygląd i zachowanie kamery. Po ustawieniu kamery bierzemy się za interfejs `SurfaceHolder`, w którym będzie się pojawił obraz wideo.

W metodzie zwrotnej `surfaceCreated()` prezentujemy obiektowi kamery miejsce do wyświetlanego bieżącego widoku, inaczej mówiąc, podglądu. Po uruchomieniu podglądu możemy uaktywnić przycisk inicjalizujący obiekt `MediaRecorder`. Podgląd kamery jest bardzo przydatną funkcją, dzięki której użytkownik może zobaczyć, na co w danej chwili jest skierowany obiekt, jeszcze zanim rozpoczęnie się proces rejestracji. Bez względu na to, czy rejestrujemy obraz wideo, czy robimy zwykłe zdjęcie, najlepiej jest zaopatrzyć aplikację w funkcję podglądu dokładnie w zaprezentowany powyżej sposób.

W celu zachowania ciągłości opisu pokazaliśmy również metody `releaseRecorder()` oraz `releaseCamera()`. Są one wywoływane w metodzie `onPause()`, co widzieliśmy na listingu 19.12.

W tym momencie mamy już skonfigurowaną kamerę, zainicjalizowane przyciski oraz widoczny podgląd kamery. Użytkownik może zacząć teraz klikać przyciski, chociaż obecnie jedynym aktywnym jest *Inicjalizacja rejestratora*. Po jego wcisnięciu zostanie wykonany kod widoczny na listingu 19.14. Widzimy w nim pięć działań, każde powiązane z jednym przyciskiem. Po wykonaniu każdego zadania odpowiednie przyciski będą włączane lub wyłączone, w zależności od następnego działania. Na przykład po zainicjalizowaniu rejestratora przycisk *Inicjalizacja rejestratora* zostanie wyłączony, natomiast zostanie uaktywniony przycisk *Rozpocznij rejestrowanie*.

**Listing 19.14.** Kod przetwarzania działań przycisków w rejestratorze wideo

```
public void doClick(View view) {
    switch(view.getId()) {
        case R.id.initBtn:
            initRecorder();
            break;
        case R.id.beginBtn:
            beginRecording();
            break;
        case R.id.stopBtn:
            stopRecording();
            break;
        case R.id.playRecordingBtn:
            playRecording();
            break;
        case R.id.stopPlayingRecordingBtn:
            stopPlayingRecording();
            break;
    }
}

private void initRecorder() {
    if(mRecorder != null) return;

    mOutputFileName = Environment.getExternalStorageDirectory() +
                      "/videooutput.mp4";

    File outFile = new File(mOutputFileName);
    if(outFile.exists()) {
        outFile.delete();
    }

    try {
        mCamera.stopPreview();
        mCamera.unlock();
        mRecorder = new MediaRecorder();
        mRecorder.setCamera(mCamera);

        mRecorder.set AudioSource(MediaRecorder.AudioSource.CAMCORDER);
        mRecorder.set VideoSource(MediaRecorder.VideoSource.CAMERA);
        mRecorder.set OutputFormat(MediaRecorder.OutputFormat.MPEG_4);
        mRecorder.set VideoSize(176, 144);
        mRecorder.set VideoFrameRate(15);
        mRecorder.set VideoEncoder(MediaRecorder.VideoEncoder.MPEG_4_SP);
        mRecorder.set AudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
        mRecorder.set MaxDuration(7000); // ograniczenie do 7 sekund
        mRecorder.set PreviewDisplay(mHolder.getSurface());
        mRecorder.set outputFile(mOutputFileName);

        mRecorder.prepare();
        Log.v(TAG, "Obiekt MediaRecorder zainicjalizowany");
        mInitBtn.setEnabled(false);
        mStartBtn.setEnabled(true);
    }
    catch(Exception e) {
```

```

        Log.v(TAG, "Inicjalizacja obiektu MediaRecorder zakończona niepowodzeniem");
        e.printStackTrace();
    }
}

private void beginRecording() {
    mRecorder.setOnInfoListener(this);
    mRecorder.setOnErrorListener(this);
    mRecorder.start();
    mRecordingMsg.setText("REJESTROWANIE");
    mStartBtn.setEnabled(false);
    mStopBtn.setEnabled(true);
}

private void stopRecording() {
    if (mRecorder != null) {
        mRecorder.setOnErrorListener(null);
        mRecorder.setOnInfoListener(null);
        try {
            mRecorder.stop();
        }
        catch(IllegalStateException e) {
            // Może się to przytrafić, jeśli rejestrator został już zatrzymany.
            Log.e(TAG, "Wyjątek IllegalStateException w metodzie stopRecording");
        }
        releaseRecorder();
        mRecordingMsg.setText("");
        releaseCamera();
        mStartBtn.setEnabled(false);
        mStopBtn.setEnabled(false);
        mPlayBtn.setEnabled(true);
    }
}

private void playRecording() {
    MediaController mc = new MediaController(this);
    mVideoView.setMediaController(mc);
    mVideoView.setVideoPath(mOutputFileName);
    mVideoView.start();
    mStopPlayBtn.setEnabled(true);
}

private void stopPlayingRecording() {
    mVideoView.stopPlayback();
}

```

W metodzie `initRecorder()` następuje spora część procesu konfiguracji. Rejestrator musi otrzymać ścieżkę docelową nagrywanego obrazu, zatem wprowadzamy jej wartość. Usuwamy plik, jeśli już taki istnieje. Warto zwrócić uwagę, w jaki sposób zatrzymaliśmy podgląd obrazu kamery, jej odblokowanie i podłączenie do obiektu `MediaRecorder`. Kamera jest w pewien sposób wrażliwa na blokowanie lub odblokowywanie, a czasami trzeba ją zablokować, aby uniemożliwić innym aplikacjom dostęp do niej. Z kolei w innych sytuacjach trzeba ją odblokować, aby wykonywać za jej pomocą wcześniej niedostępne operacje. Teraz właśnie musimy ją odblokować i podłączyć do obiektu `MediaRecorder`. Po podłączeniu kamery konfigurujemy resztę atrybutów

klasy MediaRecorder, w tym źródło dźwięku i obrazu. Czytelnik pewnie zorientował się, że przed chwilą podłączliśmy kamerę do tego obiektu. Owszem, podłączliśmy. Ciągle jednak musimy w jawnym sposobie ustawić źródło obrazu video. Poprzez podłączenie kamery do rejestratora unikamy konieczności usunięcia obiektu Camera tylko po to, aby rejestrator utworzył nowy obiekt Camera. Przed wywołaniem metody `prepare()` ustanawiamy jeszcze kodery audio i video, a także ścieżkę do wynikowego pliku na karcie SD. Metoda `prepare()` zostaje wywoływana pod sam koniec i rzeczywiście przygotowuje program do właściwego procesu rejestracji. Kończymy tę metodę uaktywnieniem przycisku *Rozpocznij rejestrowanie*.

Dla porównania — metoda `beginRecording()` jest dość nieskomplikowana. Dodaje ona obiekty nasłuchujące, wywołuje metodę `start()`, następnie ustanawia komunikat o trwającej rejestracji oraz zmienia stan przycisków. Po przetworzeniu kodu tej metody aplikacja powinna już rejestrować obraz video, a także wyświetlić napis *REJESTROWANIE*, widoczny na rysunku 19.6.

Metoda `stopRecording()` jest nieco bardziej złożona, częściowo dlatego, że może zostać wywołana z kilku miejsc. Za chwilę omówimy drugie takie miejsce, na razie jednak założymy, że przycisk *Zatrzymaj rejestrowanie* uruchomił tę metodę. Jeżeli cały czas rejestrator pracuje poprawnie, zatrzymujemy metody zwrotne, a następnie wywołujemy metodę `stop()`. Ponieważ istnieje możliwość, że metoda `stop()` zostanie wywołana wobec już wcześniej zatrzymanego rejestratora, wyświetli się informujący nas o tym fakcie wyjątek. Następnie program zwalnia rejestrator oraz kamerę i wyłącza napis *REJESTROWANIE*. Na koniec przyciski rejestrowania oraz odtwarzania zamieniają się stanami.

Metoda `playRecording()` jest również nieskomplikowana. Obiekt `MediaRecorder` umieszczamy w widoku `VideoView`, wskazujemy nowy plik i wywołujemy metodę `start()`. Metoda `stopPlayingRecording()` jest jeszcze prostsza: zatrzymujemy po prostu odtwarzanie pliku video. Gdy aplikacja znajduje się w trybie odtwarzania, nic się nie stanie, jeżeli klikniemy przycisk *Odtwarzaj* w czasie oglądania video lub przycisk *Zatrzymaj*, gdy odtwarzanie zostało już zatrzymane.

Zauważaliśmy, że rejestrowanie może zostać zatrzymane z kilku miejsc. Jedno z ustawień rejestratora definiuje maksymalnie siedmiosekundowy czas nagrywania. Oznacza to, że proces rejestrowania zostanie po siedmiu sekundach automatycznie zatrzymany i wtedy nastąpi wywołanie naszej informacyjnej metody zwrotnej. Spójrzmy teraz na fragment kodu służący do obsługi tego mechanizmu, umieszczony na listingu 19.15.

---

**Listing 19.15.** Informacyjne metody zwrotne rejestratora video

---

```
@Override
public void onInfo(MediaRecorder mr, int what, int extra) {
    Log.i(TAG, "nastąpiło zdarzenie rejestrowania");
    if(what == MediaRecorder.MEDIA_RECORDER_INFO_MAX_DURATION_REACHED) {
        Log.i(TAG, "...osiagnięty maksymalny czas rejestrowania");
        stopRecording();
        Toast.makeText(this, "Został osiągnięty limit rejestrowania. Zatrzymywanie.", Toast.LENGTH_SHORT).show();
    }
}

@Override
public void onError(MediaRecorder mr, int what, int extra) {
    Log.e(TAG, "nastąpił błąd rejestrowania");
```

```

        stopRecording();
        Toast.makeText(this, "Nastąpił błąd rejestrowania. Zatrzymywanie rejestrowania.",
                      Toast.LENGTH_SHORT).show();
    }
}

```

---

Te dwie metody zwrotne są do siebie bardzo podobne. Jedyna różnica pomiędzy nimi polega na okolicznościach, w jakich są wywoływane. W metodzie `onInfo()` komunikaty nie są uznawane za wyniki błędów. Metoda ta może być wywoływana w przypadku osiągnięcia limitu długości rejestrowanego materiału albo maksymalnego rozmiaru pliku, jeżeli takie ograniczenia znajdą się w rejestratorze. W przypadku metody `onError()` dokumentacja nie wyraża się zbyt jasno na temat okoliczności jej wywoływania, ale może tak być w przypadku, gdy wyczerpuje się miejsce w magazynie, na którym zapisywane są rejestrowane dane. Jeżeli metoda `onInfo()` zostanie wywołana z powodu osiągnięcia limitu czasowego lub jeśli pojawi się jakiś błąd rejestrowania, zostanie ono zatrzymane.

Podobnie jak w przypadku procesu rejestrowania audio, tak i teraz musimy przydzielić uprawnienia dla rejestracji dźwięku (`android.permission.RECORD_AUDIO`) i dostępu do karty SD (`android.permission.WRITE_EXTERNAL_STORAGE`), a dodatkowo musimy jeszcze dodać uprawnienie dostępu do kamery (`android.permission.CAMERA`). Na listingu zamieszczamy kod pliku *AndroidManifest.xml*. Zauważmy, że wymuszamy poziomą orientację naszej aplikacji, dlatego właśnie plik układu graficznego znajduje się w katalogu */res/layout-land/main.xml*.

**Listing 19.16.** Plik AndroidManifest.xml rejestratora wideo

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.record.video"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".MainActivity"
            android:label="@string/app_name"
            android:screenOrientation="landscape">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="4" />

    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.CAMERA" />
</manifest>

```

---

## Klasy CameraProfile i CamcorderProfile

Na listingu 19.14 dostrzegliśmy w metodzie `initRecorder()` szereg bardzo specyficznych ustawień rejestratora wideo. Pytanie brzmi: skąd możemy wiedzieć, jakie możliwości posiada urządzenie, na którym została uruchomiona nasza aplikacja? Przed wersją 2.2 Androida na to

pytanie nie było dobrej odpowiedzi. Fabrycznie instalowana aplikacja Aparat korzysta z nieudokumentowanej klasy `SystemProperties`. Zatem w starszych wersjach Androida trzeba było samodzielnie wybierać wartości, które działałyby na docelowych urządzeniach. Nie było to wcale satysfakcyjne rozwiązanie, zwłaszcza że nowsze urządzenia były wyposażane w coraz lepsze aparaty fotograficzne czy kamery. Aby naprawić sytuację, w wersji 2.2 Androida wprowadzono dwie nowe klasy: `CameraProfile` i `CamcorderProfile`. Klasy te są po prostu pojemnikami na potrzebne atrybuty aparatu fotograficznego bądź kamery. Klasa `CameraProfile` posiada tylko jedną wartość (`JPEG Encoding Quality Parameter`), podczas gdy w klasie `CamcorderProfile` możemy definiować częstotliwość odświeżania, rozmiar klatki (wysokość i szerokość), a także inne parametry dźwięku i obrazu. To nie wszystko, gdyż klasa `MediaRecorder` akceptuje parametry przechowywane w klasie `CamcorderProfile` i pozwala w ten sposób na definiowanie różnorodnych ustawień rejestrowania obrazu. Musimy tylko pamiętać, aby wywoływać metodę `setProfile()` po ustanowieniu źródeł audio i wideo, lecz przed zdefiniowaniem pliku wynikowego.

Wraz z wprowadzeniem wersji 2.3 Androida metody do obsługi kamery lub aparatu fotograficznego uzyskały alternatywne wersje, umożliwiające korzystanie z identyfikatorów kamer lub aparatów. Wcześniej większość urządzeń posiadała tylko jedną kamerę (aparat fotograficzny), zazwyczaj umieszczoną w tylnej ściance. W przypadku nowych urządzeń, w których oprócz standardowej kamery czy aparatu fotograficznego dostępna jest również kamera (aparat) umieszczona z przodu urządzenia, kod musi w jakiś sposób rozróżniać obsługiwane kamery i aparaty fotograficzne. Na przykład w klasie `Camera` metoda `open()` przekaże obiekt `Camera` dla aparatu fotograficznego znajdującego się z tyłu, jeśli takowy jest obecny. Mamy do dyspozycji metodę `open(int cameraId)`, definiującą określony aparat fotograficzny, dzięki czemu możemy korzystać z aparatu umieszczonego z przodu, jeśli jest dostępny. Aby określić liczbę dostępnych aparatów fotograficznych oraz je rozróżniać od siebie, możemy wykorzystać metodę `Camera.getNumberofCameras()`, przekazującą liczbę dostępnych aparatów, oraz `Camera.getCameraInfo()`, wyświetlającą informacje o danym aparacie, także o jego umiejscowieniu.

## Analiza klasy `MediaStore`

Dotychczas zajmowaliśmy się multimediami poprzez bezpośrednie tworzenie klas odpowiedzialnych za odtwarzanie i rejestrowanie plików w naszej aplikacji. Jedną z lepszych cech Androida jest możliwość łączenia się z innymi aplikacjami, które mogą wykonać całą pracę. W klasie `MediaStore` został zaimplementowany interfejs obsługujący multimedia przechowywaną w urządzeniu — zarówno w jego wnętrzu, jak i na zewnątrz.

Dostępne są tu również interfejsy API pozwalające na przeprowadzanie operacji na tych plikach. Możemy na przykład przeszukiwać urządzenie pod kątem określonych formatów plików multimedialnych, korzystać z intencji umożliwiających rejestrowanie dźwięku i wideo w magazynie, tworzyć listy odtwarzania i tak dalej. Klasa ta była dostępna w starszych wersjach środowiska Android SDK, ale od wersji 1.5 została znacznie usprawniona.

Ponieważ klasa `MediaStore`, podobnie jak klasa `MediaRecorder`, obsługuje intencje odpowiedzialne za rejestrowanie dźwięku i wideo, rodzi się oczywiste pytanie: kiedy należy stosować klasę `MediaStore`, a kiedy klasę `MediaRecorder`? Na przykładach aplikacji służących do rejestracji dźwięku oraz rejestracji wideo pokazaliśmy, że w przypadku klasы `MediaRecorder` możemy skonfigurować różne opcje dotyczące źródła rejestrowanych danych. Do dyspozycji mamy takie opcje, jak wejściowe źródło danych audio lub (i) wideo, częstotliwość wyświetlania klatek, rozmiar ramki obrazu, formaty plików wynikowych i tak dalej. Klasa `MediaStore` nie posiada takich możliwości konfiguracyjnych, jeśli jednak nie są nam potrzebne, być może łatwiej będzie stosować intencje klasy `MediaStore`. Co ważniejsze, dane utworzone za pomocą klasы `Media`

→ Recorder nie są automatycznie dostępne dla innych aplikacji korzystających z magazynu multimediiów. Jeżeli korzystamy z klasy MediaRecorder, możemy dodać nagranie do magazynu multimediiów za pomocą interfejsów API klasy MediaStore, więc może prostszym rozwiązaniem byłoby zastosowanie przede wszystkim klasy MediaStore. Kolejna ważna różnica polega na braku konieczności przyznawania aplikacji uprawnienia do rejestracji dźwięku, uzyskania dostępu do obiektu Camera oraz zapisu danych na kartę SD w przypadku wywołania klasy MediaStore poprzez intencję. Jest tak, ponieważ aplikacja wywołuje oddzielną aktywność, która musi posiadać uprawnienia do rejestracji dźwięku, dostępu do klasy Camera i zapisu na karcie SD. Aktywności klasy MediaStore mają wbudowane te uprawnienia, zatem aplikacja nie musi ich posiadać. Zobaczmy teraz, w jaki sposób możemy wykorzystać interfejsy API klasy MediaStore.

## Rejestrowanie dźwięku za pomocą intencji

Pokazaliśmy, że kod procesu rejestracji dźwięku jest prosty, lecz w przypadku wywołania intencji z klasy MediaStore staje się jeszcze prostszy. Listing 19.17 stanowi demonstrację zastosowania intencji do rejestracji dźwięku.

**Listing 19.17.** Wykorzystanie intencji do nagrania dźwięku

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:id="@+id/recordBtn"
        android:text="Rejestruj dźwięk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class UsingMediaStoreActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        Button btn = (Button)findViewById(R.id.recordBtn);
        btn.setOnClickListener(new OnClickListener(){
            @Override
            public void onClick(View view) {
```

```

        startRecording();
    });
}

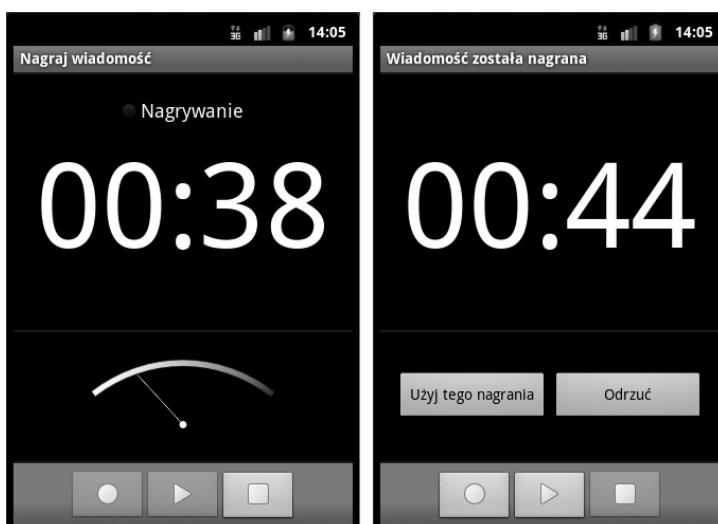
public void startRecording() {
    Intent intt = new Intent("android.provider.MediaStore.RECORD_SOUND");
    startActivityForResult(intt, 0);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {

    switch (requestCode) {
        case 0:
            if (resultCode == RESULT_OK) {
                Uri recordedAudioPath = data.getData();
                Log.v("Demo", "Identityfikator Uri wynosi " + recordedAudioPath.toString());
            }
    }
}
}

```

Kod z listingu 19.17 tworzy intencję żądającą od systemu przeprowadzenia procesu rejestracji dźwięku. Intencja zostaje uruchomiona wobec aktywności poprzez wywołanie metody `startActivityForResult()` oraz przekazanie tej intencji wraz z obiektem `requestCode`. Kiedy żądana aktywność wykona swoją pracę, zostaje wywołana metoda `onActivityResult()` wraz z obiektem `requestCode`. Jak wynika z kodu metody `onActivityResult()`, należy wyszukać obiekt `requestCode` odpowiadający obiektowi przekazanemu klasie `startActivityForResult()`, a następnie uzyskać identyfikator URI zapisanego pliku poprzez wywołanie metody `data.getData()`. Możemy następnie przekazać otrzymany identyfikator URI do intencji, aby odtworzyć nagranie. Utworzony na listingu 19.17 interfejs UI został zilustrowany na rysunku 19.7.



Rysunek 19.7. Wbudowany rejestrator dźwięku przed naganiem (po lewej) i po nagraniu (po prawej)

Na rysunku 19.7 zostały zaprezentowane dwa zrzuty ekranu. Obraz z lewej strony przedstawia rejestrator dźwięku w trakcie procesu nagrywania, a na zrzucie ekranu z prawej strony widoczny jest interfejs UI aktywności po zatrzymaniu procesu rejestracji.

W podobny sposób klasa MediaStore dostarcza intencję umożliwiającą wykonanie zdjęcia. Przykładem jest kod z listingu 19.18.

---

**Listing 19.18.** Uruchamianie intencji odpowiedzialnej za wykonywanie zdjęć

---

```
<?xml version="1.0" encoding="utf-8"?>
<!--Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:id="@+id	btn"
        android:text="Zrób zdjęcie"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="captureImage" />
</LinearLayout>

import android.app.Activity;
import android.content.ContentValues;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore;
import android.provider.MediaStore.Images.Media;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MainActivity extends Activity {

    Uri myPicture = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_LANDSCAPE);
    }

    public void captureImage(View view)
    {
        ContentValues values = new ContentValues();
        values.put(Media.TITLE, "Moje próbne zdjęcie");
        values.put(Media.DESCRIPTION, "Zdjęcie wykonane aparatem
        ↩fotograficznym za pomocą intencji");
    }
}
```

```
myPicture = getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, values);

Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
i.putExtra(MediaStore.EXTRA_OUTPUT, myPicture);

startActivityForResult(i, 0);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if(requestCode==0 && resultCode==Activity.RESULT_OK)
    {
        // Teraz wiemy, że identyfikator URI myPicture odnosi się do wykonanego przed
        // chwilą zdjęcia.
    }
}
```

Klasa aktywności pokazana na listingu 19.18 definiuje metodę `captureImage()`. Dzięki niej system tworzy intencję, której działanie nosi nazwę `MediaStore.ACTION_IMAGE_CAPTURE`. Po uruchomieniu tej intencji na pierwszym planie ekranu wyświetla się okno aplikacji aparatu fotograficznego i użytkownik może wykonać zdjęcie. Ponieważ wcześniej utworzyliśmy identyfikator URI, możemy umieścić dodatkowe informacje na temat zdjęcia, zanim zostanie zrobione. Do tego celu służy nam klasa `ContentValues`. Poza atrybutami `TITLE` i `DESCRIPTION` można dodać do obiektu `values` również inne parametry. Pełną listę atrybutów można znaleźć, przeglądając klasę `MediaStore.Images.ImageColumns`. Po wykonaniu zdjęcia zostaje wywołana metoda zwrotna `onActivityResult()`. W naszym przykładzie użyliśmy dostawcy treści multimediiów do utworzenia nowego pliku. Moglibyśmy również utworzyć nowy identyfikator URI z nowego pliku na karcie SD, tak jak poniżej:

```
myPicture = Uri.fromFile(new
File(Environment.getExternalStoragePublicDirectory(DIRECTORY_DCIM) +
"/100ANDRO/imageCaptureIntent.jpg"));
```

Jednak utworzenie identyfikatora URI w ten sposób utrudnia zaimplementowanie atrybutów zdjęcia, na przykład `TITLE` i `DESCRIPTION`. Istnieje jeszcze inny sposób wywołania intencji aparatu w celu wykonania zdjęcia. Jeżeli w ogóle nie przekażemy żadnego identyfikatora URI wraz z intencją, otrzymamy obiekt mapy bitowej, zwrócony w argumencie intencji dla metody `onActivityResult()`. W tym przypadku problemem jest domyślne zmniejszenie rozmiaru otrzymanej mapy bitowej, prawdopodobnie dlatego, że twórcy systemu Android nie zakładają przesyłania dużych ilości danych z aktywności aparatu do aktywności naszej aplikacji. Mapa bitowa będzie miała rozmiar 50 KB. Żeby uzyskać obiekt typu `Bitmap`, wprowadzamy następujący wiersz do metody `onActivityResult()`:

```
Bitmap myBitmap = (Bitmap) data.getExtras().get("data");
```

W podobny sposób zachowuje się intencja klasy `MediaStore` zapewniająca obsługę rejestracji wideo. W tym celu stosowany jest obiekt `MediaStore.ACTION_VIDEO_CAPTURE`.

## Dodawanie plików do magazynu multimediiów

Kolejną funkcją dostępną w szkieletie multimediiów Androida jest możliwość dodawania informacji o plikach multimedialnych do magazynu za pomocą klasy `MediaScannerConnection`.

Innymi słowy, jeśli w magazynie multimediiów nie znalazły się informacje o nowych plikach, dane tego typu można dodawać za pomocą klasy MediaScannerConnection. Informacje te mogą być później wykorzystywane przez inne aplikacje. Zobaczmy, jak to działa (listing 19.19).

**Listing 19.19.** Dodawanie pliku do magazynu MediaStore

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <EditText android:id="@+id/fileName"
        android:hint="Wprowadź nazwę nowego pliku"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
    <Button android:id="@+id/scanBtn"
        android:text="Dodaj plik"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="startScan" />
</LinearLayout>

import java.io.File;
import android.app.Activity;
import android.content.Intent;
import android.media.MediaScannerConnection;
import android.media.MediaScannerConnection.MediaScannerConnectionClient;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.EditText;
import android.widget.Toast;

public class MediaScannerActivity extends Activity implements
MediaScannerConnectionClient
{
    private EditText editText = null;
    private String filename = null;
    private MediaScannerConnection conn;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        editText = (EditText)findViewById(R.id.fileName);
    }

    public void startScan(View view)
```

```
{  
    if(conn!=null)  
    {  
        conn.disconnect();  
    }  
  
    filename = editText.getText().toString();  
  
    File fileCheck = new File(filename);  
    if(fileCheck.isFile()) {  
        conn = new MediaScannerConnection(this, this);  
        conn.connect();  
    }  
    else {  
        Toast.makeText(this,  
                    "Taki plik nie istnieje",  
                    Toast.LENGTH_SHORT).show();  
    }  
}  
  
@Override  
public void onMediaScannerConnected() {  
    conn.scanFile(filename, null);  
}  
  
@Override  
public void onScanCompleted(String path, Uri uri) {  
    try {  
        if (uri != null) {  
            Intent intent = new Intent(Intent.ACTION_VIEW);  
            intent.setData(uri);  
            startActivity(intent);  
        }  
        else {  
            Log.e("MediaScannerDemo", "Plik tego typu nie jest obsługiwany");  
        }  
    } finally {  
        conn.disconnect();  
        conn = null;  
    }  
}  
}
```

---

Na listingu 19.19 została ukazana klasa aktywności umożliwiająca dodawanie pliku do magazynu `MediaStore`. Jeśli proces dodawania przebiegnie pomyślnie, informacja o danym pliku zostanie wyświetlona użytkownikowi poprzez intencję. Poza wzrokiem użytkownika klasa `MediaStore` sprawdza typ pliku i inne dotyczące go informacje. W przypadku klasy `MediaStore` możemy jako drugi argument metody `scanFile()` podać typ MIME. Jeżeli klasa `MediaStore` nie potrafi rozpoznać typu pliku po rozszerzeniu jego nazwy, nie zostanie on dodany do magazynu. Jeżeli natomiast plik jest akceptowany przez klasę `MediaStore`, zostaje umieszczony wpis wewnętrz bazy danych dostawcy multimedialiów. Sam plik nie zostaje przeniesiony. Teraz jednak dostawca multimedialiów ma dostęp do wszystkich ważnych informacji na temat tego pliku. Jeżeli dodaliśmy plik obrazu, możemy uruchomić aplikację `Gallery` i go obejrzeć. W przypadku pliku muzycznego zostanie on odtworzony w aplikacji `Music`.

Jeżeli chcemy przejrzeć zawartość bazy danych dostawcy multimediiów, otwieramy okno narzędzi, następnie uruchamiamy aplikację adb shell i otwieramy plik /data/data/com.android.providers.media/databases znajdujący się na urządzeniu. Powinny się tu również znajdować zewnętrzne pliki bazodanowe, każdy reprezentujący kartę SD. Ponieważ w telefonie obsługującym system Android można umieścić kilka kart SD, w katalogu tym może się znajdować wiele plików odpowiadających tym kartom. W celu przeglądania tabel bazodanowych umieszczonych w tych plikach możemy posłużyć się aplikacją sqlite3. Istnieją oddzielne tabele dla plików audio, wideo i obrazów. W rozdziale 4. można znaleźć dodatkowe informacje na temat korzystania z aplikacji sqlite3.

## Podłączenie klasy MediaScanner do całej karty SD

W poprzednim przykładowym projekcie wykorzystaliśmy klasę MediaScanner do wyszukania pojedynczego, określonego pliku. Jest to wystarczające, w przypadku gdy chcemy dodać jeden plik. Co jednak należy zrobić, aby zmienić nazwę pliku albo go usunąć i zaktualizować klasę MediaStore? Istnieje bardzo proste rozwiązanie tego problemu. Jeżeli wprowadzimy następujący fragment kodu do naszej aplikacji, zostanie on rozpoznany przez klasę MediaScanner, która przeszuka całą kartę SD:

```
sendBroadcast(new Intent(Intent.ACTION_MEDIA_MOUNTED,
    Uri.parse("file://" +
    Environment.getExternalStorageDirectory())));

```

W ramach ćwiczenia Czytelnik może napisać prostą aplikację wykonującą wyłącznie powyższe polecenie w metodzie onCreate().

Na tym zakończymy omawianie interfejsów API multimediiów. Mamy nadzieję, że dla Czytelnika odtwarzanie i rejestrowanie multimediiów nie będzie skomplikowanym procesem.

## Odbońniki

Poniżej prezentujemy łącz do zasobów dotyczących zagadnień, które Czytelnik może zechcieć dokładniej zgłębić:

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu pełen zestaw projektów bezpośrednio związkanych z niniejszą książką. Projekty dotyczące tego rozdziału zostały umieszczone w katalogu *ProAndroid3\_R19\_Multimedia*. Dołączliśmy tu także plik *Czytaj.TXT*, w którym został dokładnie opisany sposób importowania tych projektów do środowiska Eclipse.
- [http://developer.android.com/guide/topics/media/jet/jetcreator\\_manual.html](http://developer.android.com/guide/topics/media/jet/jetcreator_manual.html) — instrukcja obsługi narzędzia JETCreator. Za jego pomocą możemy utworzyć plik dźwiękowy JET, który następnie będzie odtwarzany poprzez klasę JetPlayer. Narzędzie to jest dostępne wyłącznie w systemach Windows i Mac OS. Aby sprawdzić działanie klasy JetPlayer, należy wczytać, skompilować i włączyć przykładowy projekt *JetBoy*, dostępny w zestawie SDK. Warto wspomnieć, że przycisk *Fire* znajduje się na środku podkładki kierunkowej.

## Podsumowanie

W tym rozdziale zajmowaliśmy się strukturą multimedii w Androidzie. Pokazaliśmy, w jaki sposób można odtwarzać pliki audio i wideo. Omówiliśmy także sposoby rejestracji dźwięków i obrazów wideo — zarówno bezpośrednio, jak i za pomocą intencji.

W kolejnym rozdziale zwróciemy uwagę na grafikę trójwymiarową poprzez omówienie zastosowania technologii OpenGL w aplikacjach tworzonych dla systemu Android.

# Programowanie grafiki trójwymiarowej za pomocą biblioteki OpenGL

W niniejszym rozdziale przyjrzymy się dokładnie sposobom pracy z interfejsem API biblioteki OpenGL ES w systemie Android. Biblioteka OpenGL ES jest odmianą specyfikacji OpenGL, zoptymalizowaną pod kątem systemów wbudowanych oraz innych urządzeń o małej mocy, na przykład telefonów komórkowych.

System Android obsługuje biblioteki OpenGL ES w wersji 1.0 oraz 2.0. Wersja 2.0 została wprowadzona dopiero w interfejsie API poziomu 8., co odpowiada wersji 2.2 Androida. W czasie pisania tej książki występowali pewne problemy z powiązaniami kodu Java ze środowiskiem OpenGL ES 2.0. Radzimy przejrzeć uwagi i zalecenia dotyczące tej wersji środowiska, które zamieściliśmy w dalszej części rozdziału. Głównym problemem jest brak obsługi tego środowiska w emulatorze. Wersja 3.0 Androida została jeszcze bardziej wzbogacona o możliwości obsługi środowiska OpenGL ES 2.0 poprzez wprowadzenie języka Renderscript. Jest to język stworzony z myślą o poprawie wydajności, w którym natywny kod przypomina nieco języki z rodziny C. Taki kod może być nawet wykonywany przez jednostkę GPU (koprocesor graficzny, z ang. *Graphical Processing Unit*). Język Renderscript wykazuje również kompatybilność międzyplatformową. Jeżeli wydajność nie ma krytycznego znaczenia, zalecane jest używanie powiązań Java podczas większości pracy wykonywanej przez środowisko OpenGL. Z powodu różnorodnych ograniczeń nie omówiliśmy w tej książce języka Renderscript; na końcu rozdziału zamieściliśmy jednak odniesienie do podręcznika programowania w tym języku (wydanego przez firmę Google).

Środowisko Android SDK zostało wyposażone w wiele przykładowych plików, pokazujących możliwości biblioteki OpenGL ES, jednak w zestawie SDK niemal nie istnieje dokumentacja, która mogłaby posłużyć jako podręcznik do pracy z biblioteką OpenGL ES. Wynika to z założenia, że biblioteka OpenGL ES jest otwartym standardem, którego programiści mogą się uczyć z zewnętrznych źródeł. Wskutek tego w dostępnych źródłach internetowych lub przykładowych kodach traktujących o korzystaniu z biblioteki OpenGL ES w Androidzie przyjmuje się założenie, że programiści są już zaznajomieni z architekturą OpenGL.

W tym rozdziale pomożemy ominąć tę przeszkodę. Po spełnieniu kilku warunków wstępnych już pod koniec lektury tego rozdziału programowanie za pomocą biblioteki OpenGL ES stanie się przyjemnością. Dokonamy tego niemal bez udziału aparatu matematycznego (przeciwnie niż w wielu innych książkach poświęconych bibliotece OpenGL).

W pierwszym podrozdziale dokonamy przeglądu bibliotek OpenGL, OpenGL ES oraz niektórych konkurencyjnych standardów.

W drugim podrozdziale zajmiemy się częścią teoretyczną, dotyczącą technologii OpenGL. Jest to podstawowa sekcja dla osób dopiero rozpoczynających przygodę z tą biblioteką. Omówimy w niej współrzędne OpenGL, pojęcie kamery oraz podstawy interfejsów API rysowania.

Trzeci podrozdział jest poświęcony pracy z interfejsem API OpenGL w Androidzie. Opisujemy tutaj interfejsy `GLSurfaceView` oraz `Render器`, a także sposób, w jaki współpracują ze sobą podczas procesu rysowania. W jednym z prostych przykładów narysujemy trójkąt oraz pokażemy, jak na proces rysowania wpływa zmiana interfejsów API konfiguracji sceny.

#### Uwaga!

Pojęcie kamery w bibliotece OpenGL jest podobne, lecz nie identyczne z pojęciem klasy `Camera` z pakietu graficznego Androida, o którym była mowa w rozdziale 6. Podczas gdy klasa `Camera` symuluje wyświetlanie perspektywy trójwymiarowej poprzez rzutowanie dwuwymiarowego widoku poruszającego się w trójwymiarowej przestrzeni, kamera biblioteki OpenGL jest paradrygmatem reprezentującym wirtualny punkt widzenia. Innymi słowy, ukazuje ona rzeczywistą scenę, widzianą oczami obserwatora patrzącego przez obiektyw kamery. Więcej informacji znajduje się w punkcie „Kamera i współrzędne”, w podrozdziale „Podstawy struktury OpenGL”. Obydwa obiekty kamery nie są związane z fizyczną kamerą urządzenia podręcznego, za pomocą której wykonujemy zdjęcia i rejestrujemy filmy.

W czwartym podrozdziale zajmiemy się nieco bardziej zaawansowanymi kwestiami biblioteki OpenGL i wprowadzimy pojęcie kształtów. Opiszemy również tekstury oraz zaprezentujemy sposób rysowania wielu figur geometrycznych za pomocą jednej metody `draw`. Przyjrzymy się następnie obsłudze środowiska OpenGL ES 2.0, dokładniej zaś jednostkom cieniującym, którym poświęcimy krótki przykładowy projekt. Z góry przestrzegamy, że funkcje środowiska OpenGL 2.0 mogą być testowane wyłącznie na urządzeniu fizycznym.

Następnie rozdział zamknijmy listą źródeł, z których korzystaliśmy w trakcie opracowywania materiału do tego rozdziału.

Zatem przyjrzyjmy się historii i podstawom biblioteki OpenGL.

## Historia i podstawy biblioteki OpenGL

Biblioteka OpenGL (jej pierwotna nazwa to Open Graphics Library, czyli otwarta biblioteka graficzna) jest dwu- oraz trójwymiarowym interfejsem graficznym, zaprojektowanym przez firmę Silicon Graphics, Inc. (SGI) dla produkowanych przez nią stacji roboczych, pracujących pod kontrolą systemu UNIX. Chociaż stworzone przez firmę SGI wersje biblioteki OpenGL istnieją już od długiego czasu, pierwsza ustandaryzowana specyfikacja tej technologii pojawiła się dopiero w 1992 roku. Obecnie standard OpenGL został przystosowany do wszystkich systemów operacyjnych i jest szeroko wykorzystywany w procesie pisania gier, projektowania CAD (ang. *Computer Aided Design* — projektowanie wspomagane komputerowo), a nawet tworzenia wirtualnych rzeczywistości.

Standard OpenGL jest obecnie zarządzany przez konsorcjum nazwane grupą Khronos ([www.khronos.org](http://www.khronos.org)), założone w 2000 roku przez takie firmy, jak NVIDIA, Sun Microsystems, ATI Technologies oraz SGI. Informacje dotyczące specyfikacji technologii OpenGL można znaleźć na stronie konsorcjum:

[www.khronos.org/opengl/](http://www.khronos.org/opengl/)

Pod poniższym adresem jest dostępna oficjalna strona dokumentacji biblioteki OpenGL:

[www.opengl.org/documentation/](http://www.opengl.org/documentation/)

Po otwarciu powyższej strony uzyskujemy dostęp do podręczników oraz zasobów interneto-wych dotyczących biblioteki OpenGL. Spośród nich klasyczną pozycją jest książka *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*, znana także jako „czerwona księga” technologii OpenGL. Jest ona dostępna pod adresem:

[www.glprogramming.com/red/](http://www.glprogramming.com/red/)

Podręcznik ten jest całkiem dobrze i przystępnie napisany. Mieliśmy jednak pewne problemy ze zrozumieniem natury *jednostek i współrzędnych*, niezbędnych podczas rysowania. Spróbujemy wyjaśnić te ważne pojęcia na podstawie sporządzanych przez nas i widzianych na ekranie obiektów za pomocą standardu OpenGL. Pojęcia te dotyczą konfigurowania kamery OpenGL i definiowania *bryły widzenia* (ang. *viewing box*), zwanej także *pojemnością widzenia* (ang. *viewing volume*) lub *ostrosłupem widzenia* (ang. *frustum*).

## OpenGL ES

Grupa Khronos jest również odpowiedzialna za dwa dodatkowe standardy powiązane z technologią OpenGL: interfejs OpenGL ES oraz interfejs graficzny platformy natywnej EGL (zwany w skrócie interfejsem EGL). Jak już wspomnieliśmy, interfejs OpenGL ES jest mniejszą wersją standardu OpenGL, przeznaczoną dla systemów wbudowanych.

### Uwaga!

Proces JCP (ang. *Java Community Process*) również umożliwia zaprojektowanie abstrakcji obiektowej standardu OpenGL dla urządzeń mobilnych. Abstrakcja ta nosi nazwę interfejsu M3G (ang. *Mobile 3D Graphics* — mobilna grafika trójwymiarowa). Omówimy krótko ten interfejs w punkcie „M3G — inny standard grafiki trójwymiarowej środowiska Java”.

Zasadniczo standard EGL jest interfejsem łączącym system operacyjny z interfejsami renderującymi, dostępnymi w środowisku OpenGL ES. Ponieważ standardy OpenGL i OpenGL ES są ogólnymi interfejsami służącymi do rysowania, każdy system operacyjny musi im zapewnić standardowe środowisko bazowe umożliwiające współdziałanie. Od wersji 1.5 środowiska Android SDK informacje dotyczące tych parametrów platformy są całkiem skutecznie ukrywane. Zajmiemy się tym dokładniej w podrozdziale „Tworzenie interfejsu pomiędzy standardem OpenGL ES a Androidem”.

Docelowymi urządzeniami standardu OpenGL ES są telefony komórkowe, sprzęt RTV, a nawet pojazdy. Ponieważ standard ten musiał zostać mocno okrojony w porównaniu do podstawowej wersji biblioteki OpenGL, usunięto wiele przydatnych funkcji. Na przykład nie ma funkcji bezpośredniego rysowania prostokątów; należy w tym celu narysować dwa trójkąty.

Podczas nauki obsługi biblioteki OpenGL w Androidzie należy koncentrować się przede wszystkim na interfejsie OpenGL ES i jego powiązaniach z systemem poprzez języki Java oraz EGL. Dokumentację dla interfejsu OpenGL ES można znaleźć tutaj:

[www.khronos.org/opengles/documentation/opengles1\\_0/html/index.html](http://www.khronos.org/opengles/documentation/opengles1_0/html/index.html)

Podczas pisania tego rozdziału bez przerwy powracaliśmy do tego źródła, ponieważ są w nim wymienione i opisane wszystkie interfejsy API OpenGL ES oraz ich argumenty. Interfejsy te przypominają interfejsy API środowiska Java, a w tym rozdziale omówimy najważniejsze z nich.

## Środowisko OpenGL ES a Java ME

Podobnie jak biblioteka OpenGL, tak i środowisko OpenGL ES jest płaskim interfejsem opartym na języku C. Ponieważ zestaw Android SDK jest interfejsem programowania bazującym na języku Java, wymagane jest powiązanie języka Java z interfejsem OpenGL ES. W przypadku środowiska Java ME takie powiązanie zostało już zdefiniowane w specyfikacji JSR 239: Java Binding for the OpenGL ES API. Sama specyfikacja JSR 239 opiera się na specyfikacji JSR 231, stanowiącej powiązanie języka Java z biblioteką OpenGL 1.5. Specyfikacja JSR 239 mogłaby zostać podziobrem specyfikacji JSR 231, nie wchodzi to jednak w grę, ponieważ musi ona zaadaptować pewne rozszerzenia interfejsu OpenGL ES, których nie ma w bibliotece OpenGL 1.5.

Dokumentacja specyfikacji JSR 239 jest dostępna tutaj:

<http://java.sun.com/javame/reference/apis/jsr239/>

Informacje zawarte pod powyższym adresem dadzą Czytelnikowi pojęcie na temat interfejsów API dostępnych w bibliotece OpenGL ES. Można tam również znaleźć cenne informacje dotyczące następujących pakietów:

- *javax.microedition.khronos.egl*,
- *javax.microedition.khronos.opengles*,
- *java.nio*.

Pakiet *nio* jest niezbędny, ponieważ implementacje środowiska OpenGL ES przyjmują dane wejściowe jedynie w postaci strumieni bajtów, co pozwala zachować wysoką wydajność. W pakiecie tym zdefiniowano wiele narzędzi służących do przygotowania buforów natywnych pod kątem korzystania z nich przez bibliotekę OpenGL. Niektóre z tych interfejsów API zostaną zastosowane w podpunkcie „glVertexPointer i określanie wierzchołków rysowania” podrozdziału „Podstawy struktury OpenGL”.

Pod poniższym adresem można znaleźć dokumentację (niestety, bardzo ubogą) dotyczącą obsługi biblioteki OpenGL w środowisku Android:

<http://developer.android.com/guide/topics/graphics/opengl.html>

Pod tym adresem można znaleźć informację wskazującą na fakt, że implementacja tej biblioteki w Androidzie pokrywa się w większości ze specyfikacją JSR 239, jednak w kilku miejscach mogą się pojawić odstępstwa.

## M3G — inny standard grafiki trójwymiarowej środowiska Java

Specyfikacja JSR 239 jest jedynie powiązaniem języka Java z natywnym standardem OpenGL ES. Wspomnieliśmy побieżnie w punkcie „OpenGL ES”, że środowisko Java posiada inny interfejs API obsługujący trójwymiarową grafikę w urządzeniach mobilnych — M3G. Ten standard obiektowy jest zdefiniowany w specyfikacjach JST 184 i nowszej wersji JSR 297. W przypadku specyfikacji JSR 184 technologia M3G stanowi uniwersalny, obiektowy, interaktywny interfejs grafiki trójwymiarowej dla urządzeń mobilnych.

Obiektowa natura interfejsu M3G izoluje go od środowiska OpenGL ES. Ze szczegółami można się zapoznać na poniżej stronie specyfikacji JSR 184:

[www.jcp.org/en/jsr/detail?id=184](http://www.jcp.org/en/jsr/detail?id=184)

Interfejsy API dla środowiska M3G są dostępne w pakiecie Java noszącym nazwę *javax.microedition.m3g.\**.

W środowisku M3G zaimplementowano interfejs API wyższego poziomu w stosunku do środowiska OpenGL ES, więc jego nauka powinna przebiegać łatwiej. Jednak na razie trudno ocenić, jak ta technologia będzie się sprawowała na handheldach. Na razie Android nie obsługuje technologii M3G.

Dotychczas wymieniliśmy opcje dostępne w przestrzeni OpenGL pod kątem urządzeń przenośnych. Omówiliśmy środowisko OpenGL ES i wspomnialiśmy o standardzie M3G. Przejdzmy teraz do zapoznania się z podstawami biblioteki OpenGL.

## Podstawy struktury OpenGL

W tym ustępie pomożemy zrozumieć pojęcia kryjące się w interfejsach API OpenGL i OpenGL ES. Omówimy wszystkie zasadnicze interfejsy API. W razie potrzeby na końcu rozdziału znajduje się punkt „Dodatkowe źródła dotyczące środowiska OpenGL ES 2.0”, w którym została zamieszczona lista zasobów zawierających dodatkowe informacje. Wśród tych źródeł znajdują się odnośniki do czerwonej księgi, dokumentacji specyfikacji JSR 239 oraz interfejsów API grupy Khronos.

### Uwaga!

Podczas korzystania z zasobów dotyczących OpenGL można zauważyc, że w wersji OpenGL ES brakuje części interfejsów API. W takich momentach przydaje się podręcznik referencyjny środowiska OpenGL ES (ang. *OpenGL ES Reference Manual*) grupy Khronos.

Wymienione poniżej interfejsy API, jako niezbędne do zrozumienia działania bibliotek OpenGL i OpenGL ES, zostaną szczegółowo omówione:

- `glVertexPointer`,
- `glDrawElements`,
- `glColor`,
- `glClear`,
- `gluLookAt`,
- `glFrustum`,
- `glViewport`.

Podczas omawiania tych interfejsów przedstawimy:

- wykorzystywanie podstawowych interfejsów API rysowania za pomocą środowiska OpenGL ES,
- czyszczenie palety,
- określanie kolorów,
- używanie współrzędnych i kamery biblioteki OpenGL.

## Podstawy rysowania za pomocą biblioteki OpenGL

W środowisku OpenGL rysujemy w trójwymiarowej przestrzeni. Rozpoczynamy od określenia szeregu punktów zwanych wierzchołkami. Każdy punkt posiada trzy wartości: pierwsza odpowiada współrzędnej  $x$ , druga współrzędnej  $y$ , a trzecia — współrzędnej  $z$ .

Punkty te są ze sobą łączone w celu uzyskania kształtu. Za ich pomocą można uzyskać w środowisku OpenGL ES różne kształty, które są zwane **prostymi kształtami**; zaliczamy do nich punkty, linie i trójkąty. Zwróćmy uwagę, że do prostych kształtów zaliczane są również prostokąty i wielokąty. Po pewnym czasie pracy z bibliotekami OpenGL i OpenGL ES zauważymy, że ta druga udostępnia mniej funkcji niż pierwsza. Inny przykład: biblioteka OpenGL pozwala na oddzielne określanie każdego punktu, natomiast w bibliotece OpenGL ES możliwe jest jedynie określanie zbioru punktów za jednym razem. Jednak często możemy symulować brakujące opcje poprzez korzystanie z innych, prostszych funkcji. Na przykład możemy utworzyć prostokąt poprzez złożenie dwóch trójkątów.

Biblioteka OpenGL ES zawiera dwie podstawowe metody upraszczające proces rysowania:

- `glVertexPointer`,
- `glDrawElements`.

### Uwaga!

W przypadku biblioteki OpenGL ES używamy zamiennie pojęć „interfejs API” i „metoda”.

Metoda `glVertexPointer` służy do określania zbioru punktów lub wierzchołków, a dzięki interfejsowi `glDrawElements` są one rysowane za pomocą jednego z wspomnianych wcześniej prostych kształtów. Opisemy te metody dokładniej, ustalmy jednak najpierw nomenklaturę nazewnictwa stosowaną w przypadku biblioteki OpenGL.

Wszystkie nazwy interfejsów OpenGL rozpoczynają się od przedrostka `gl`. Po nim występuje nazwa metody. Po jej nazwie może pojawić się cyfra, na przykład `3`, wskazująca albo na liczbę wymiarów ( $x, y, z$ ), albo na liczbę argumentów. Kolejnym segmentem nazwy jest litera symbolizująca typ danych, na przykład litera `f` reprezentuje dane typu `float` (w internetowych zasobach dotyczących biblioteki OpenGL można znaleźć opis różnych typów danych i odpowiadające im symbole literowe).

Istnieje jeszcze jedna konwencja. Jeżeli metoda przyjmuje argumenty o typach danych `byte` (`b`) lub `float` (`f`), to będzie posiadała dwie nazwy: jedna będzie kończyła się literą `b`, a druga — literą `f`.

Przyjrzyjmy się teraz poszczególnym metodom służącym do rysowania, począwszy od interfejsu `glVertexPointer`.

### **glVertexPointer i określanie wierzchołków rysowania**

Metoda `glVertexPointer` służy do definiowania tablicy rysowanych punktów. Każdy punkt jest umiejscowiony w trzech wymiarach przestrzeni, zatem będzie posiadał trzy wartości:  $x$ ,  $y$  i  $z$ . Na listingu 20.1 pokazano tablicę ze zdefiniowanymi trzema punktami.

**Listing 20.1.** Przykładowe współrzędne wierzchołków trójkąta utworzonego dzięki bibliotece OpenGL

```
float[] coords = {
    -0.5f, -0.5f, 0,    //p1: (x1,y1,z1)
```

---

```

    0.5f, -0.5f, 0,   //p2: (x1,y1,z1)
    0.0f, 0.5f, 0    //p3: (x1,y1,z1)
};


```

---

Struktura przedstawiona na listingu 20.1 jest ciągłym zbiorem liczb zmienoprzecinkowych, przechowywanych w macierzy opartej na języku Java. Nie przejmujmy się na razie pisaniem lub komplikowaniem tego kodu — najpierw wyjaśnimy zasady działania tych metod. Po zaprojektowaniu środowiska testowego służącego do rysowania prostych kształtów pokażemy kilka działających przykładowych projektów. Umieściliśmy także na końcu rozdziału odnośnik do gotowego projektu.

Możemy się zastanawiać, jakie jednostki zostały użyte na listingu 20.1 dla współrzędnych punktów p1, p2 i p3. Odpowiedź jest krótka: podczas modelowania przestrzeni trójwymiarowej te jednostki współrzędnych mogą być dowolne. Później jednak trzeba zdefiniować obiekt noszący nazwę **bryły okalającej** (ang. *bounding box*) lub **objętości okalającej** (ang. *bounding volume*), dzięki któremu te jednostki zostają ilościowo wyrażone.

Na przykład możemy określić bryłę okalającą jako sześcian o boku wynoszącym 5 lub 2 cala. Te współrzędne noszą również nazwę **współrzędnych świata** (ang. *world coordinates*), ponieważ określamy dzięki nim świat niezależny od fizycznych ograniczeń urządzenia. Współrzędne świata wyjaśnimy dokładniej w punkcie „Kamera i współrzędne”. Na razie założymy, że korzystamy z sześcianu o długości boku 2 cala, którego środkiem są współrzędne ( $x=0$ ,  $y=0$ ,  $z=0$ ). Inaczej mówiąc, środek jest w centrum sześcianu, którego ściany znajdują się w odległości jednej jednostki.

#### Uwaga!

Terminy *objętość okalająca*, *bryła okalająca*, *objętość widzenia*, *bryła widzenia* oraz *ostrosłup widzenia* dotyczą tego samego pojęcia: trójwymiarowej objętości w kształcie piramidy, dzięki której określamy, co jest widoczne na ekranie. Więcej informacji na ten temat można znaleźć w podpunkcie „glFrustum i objętość widzenia”, w punkcie „Kamera i współrzędne”.

Możemy również założyć, że początek układu współrzędnych znajduje się pośrodku wyświetlanego obrazu. Oś  $z$  posiada wartości ujemne w kierunku dalszego planu („oddala się” od widza), a dodatnie w stronę pierwszego planu („zbliża się” w kierunku widza). Wartości osi  $x$  są dodatnie w kierunku prawym, a ujemne w kierunku lewym. Jednak te współrzędne zależą również od kierunku, z jakiego oglądamy scenę.

Aby narysować punkty widoczne na listingu 20.1, musimy przekazać je bibliotece OpenGL ES poprzez metodę `glVertexPointer`. Jednak w celu zachowania wydajności metoda `glVertexPointer` przyjmuje natywny bufor niezależny od języka programowania, a nie macierz wartości zmienoprzecinkowych. W tym celu musimy przekształcić macierz języka Java do akceptowanego bufora natywnego, przypominającego strukturę języka C. Dokonujemy tego za pomocą klas `java.nio`. Na listingu 20.2 został zaprezentowany przykład wykorzystania buforów `nio`.

#### **Listing 20.2.** Tworzenie zmienoprzecinkowych buforów nio

---

```

jva.nio.ByteBuffer vbb = java.nio.ByteBuffer.allocateDirect(3 * 3 * 4);
vbb.order(ByteOrder.nativeOrder());
java.nio.FloatBuffer mFVertexBuffer = vbb.asFloatBuffer();

```

---

Na listingu 20.2 obiekt `ByteBuffer` stanowi bufor pamięci zdefiniowany w bajtach. Każdy punkt posiada trzy wartości zmiennoprzecinkowe, co wynika z umiejscowienia go w trójwymiarowym układzie współrzędnych, natomiast każda wartość zmiennoprzecinkowa jest czterobajtowa. Zatem na każdy punkt przypada  $4 \times 3$  bajty. Ponadto należy pamiętać, że trójkąt posiada trzy punkty. Potrzebujemy więc  $3 \times 3 \times 4$  bajty do przechowania informacji o wszystkich trzech zmiennoprzecinkowych wierzchołkach trójkąta.

Po umieszczeniu punktów w buforze natywnym możemy wywołać metodę `glVertexPointer` zgodnie z kodem pokazanym na listingu 20.3.

---

**Listing 20.3.** Definicja interfejsu API `glVertexPointer`

```
glVertexPointer(  
    // Ile współrzędnych używamy dla każdego punktu  
    3,  
    // Każda wartość jest zmiennoprzecinkowa w buforze  
    GL10.GL_FLOAT,  
    // Nie ma przestrzeni pomiędzy dwoma punktami  
    0,  
    // Wskaźnik początku bufora  
    mFVertexBuffer);
```

---

Zatrzymajmy się na chwilę przy argumentach metody `glVertexPointer`. Pierwszy argument wskazuje bibliotece OpenGL ES, ile wymiarów przypada na punkt lub wierzchołek. W naszym przypadku podajemy wartość 3 dla współrzędnych  $x$ ,  $y$  i  $z$ . Istnieje również możliwość wpisania wartości 2 dla wymiarów  $x$  i  $y$ . Wtedy parametr `z` przyjmuje wartość 0. Pamiętajmy, że pierwszym argumentem nie jest liczba punktów w buforze, lecz liczba uwzględnianych wymiarów. Zatem jeżeli chcemy przekazać 20 punktów do narysowania większej liczby trójkątów, nie wpisujemy wartości 20 w pierwszym argumentie; umieszczamy w nim wartość 2 lub 3, w zależności od liczby używanych wymiarów.

Drugi argument określa, że współrzędne muszą być interpretowane jako liczby zmiennoprzecinkowe. Trzeci argument, noszący nazwę `stride`, wskazuje, ile bajtów oddziela każdy punkt od siebie. W naszym przypadku podajemy wartość 0, ponieważ jeden punkt zostaje umieszczony tuż przy drugim. Czasami możemy dodawać atrybuty kolorów jako część bufora po każdym punkcie. W tym celu używamy atrybutu `stride` do omijania tych części bufora, w których umieszczone są punkty — specyfikację wierzchołków. Ostatnim argumentem jest wskaźnik bufora zawierającego punkty.

Skoro wiemy już, w jaki sposób konfigurować tablicę rysowanych punktów, dowiedzmy się, co należy zrobić, aby narysować te punkty za pomocą metody `glDrawElements`.

## glDrawElements

Po określaniu zbioru punktów za pomocą metody `glVertexPointer` stosujemy metodę `glDrawElements` do narysowania tych punktów w postaci jednego z prostych kształtów dopuszczalnych przez bibliotekę OpenGL ES. Odnotujmy fakt, że biblioteka OpenGL jest maszyną stanową. Zapamiętuje w sposób narastający wartości ustanowione przez jedną metodę podczas wywoływanego kolejnej metody. Nie musimy więc jawnie przekazywać punktów ustanowionych przez metodę `glVertexPointer` metodzie `glDrawElements`. Ta ostatnia będzie z nich korzystała w sposób niejawny. Listing 20.4 ukazuje przykład zastosowania tej metody wraz z dopuszczalnymi argumentami.

**Listing 20.4.** Przykład metody glDrawElements

---

```
glDrawElements(
    // Rodzaj kształtu
    GL10.GL_TRIANGLE_STRIP,
    // Liczba indeksów
    3,
    // Rozmiar każdego indeksu
    GL10.GL_UNSIGNED_SHORT,
    // Bufor zawierający trzy indeksy
    mIndexBuffer);
```

---

Pierwszy argument definiuje typ rysowanego kształtu geometrycznego: `GL_TRIANGLE_STRIP` oznacza pas trójkątów. Pozostałymi wartościami mogą być same punkty (`GL_POINTS`), pasy linii (`GL_LINE_STRIP`), same linie (`GL_LINES`), pętle linii (`GL_LINE_LOOP`), same trójkąty (`GL_TRIANGLES`) lub wachlarze trójkątów (`GL_TRIANGLE_FAN`).

Koncepcja pasa (ang. *strip*) w argumentach `GL_LINE_STRIP` i `GL_TRIANGLE_STRIP` polega na dodawaniu nowych punktów podczas korzystania z punktów już istniejących. W ten sposób możemy uniknąć definiowania wszystkich punktów dla każdego nowego obiektu. Jeśli na przykład określmy cztery punkty w macierzy, możemy wykorzystać pasy do utworzenia pierwszego trójkąta z wierzchołków (1, 2, 3), a drugiego z wierzchołków (2, 3, 4). Każdy nowy punkt spowoduje dodanie kolejnego trójkąta (szczegóły można znaleźć w czerwonej księdze biblioteki OpenGL). Możemy również zróżnicować te parametry, aby zobaczyć, w jaki sposób będą rysowane trójkąty po dodaniu nowych punktów.

Koncepcja wachlarza (ang. *fan*) w argumencie `GL_TRIANGLE_FAN` polega na wykorzystaniu pierwszego punktu jako punktu zaczepienia dla wszystkich trójkątów. Zatem tworzymy w istocie obiekt w kształcie wachlarza lub koła, którego pierwszy wierzchołek znajduje się w środku. Założymy, że posiadamy sześć punktów w macierzy: (1, 2, 3, 4, 5, 6). Użycie argumentu wachlarza spowoduje narysowanie trójkątów o wierzchołkach (1, 2, 3), (1, 3, 4), (1, 4, 5) i (1, 5, 6). Każdy nowy punkt tworzy dodatkowy trójkąt, co można przypominać rozwijanie wachlarza lub talii kart.

Pozostałe argumenty metody `glDrawElements` służą do określania możliwości ponownego wykorzystania charakterystyki punktu. Na przykład kwadrat składa się z czterech punktów. Każdy kwadrat można narysować jako kombinację dwóch trójkątów. Czy w celu narysowania dwóch trójkątów tworzących kwadrat musimy wyznaczyć sześć punktów? Nie. Wystarczy określenie jedynie czterech punktów i sześciokrotne odniesienie się do nich w celu narysowania dwóch trójkątów. Proces ten nosi nazwę *indeksowania do bufora punktu*.

Przykład:

Punkty: (p1, p2, p3, p4)  
Rysuj indeksy (p1, p2, p3, p2, p3, p4)

Zauważmy, że pierwszy trójkąt składa się z punktów p1, p2, p3, a drugi z punktów p2, p3, p4. Dzięki tej wiedzy możemy poprzez drugi argument metody `glDrawElements` określić liczbę indeksów w buforze indeksów.

Trzeci argument metody `glDrawElements` (listing 20.4) wskazuje typ wartości w macierzy indeksów — czy jest to typ *unsigned short* (`GL_UNSIGNED_SHORT`), czy *unsigned byte* (`GL_UNSIGNED_BYTE`).

Ostatni argument metody `glDrawElements` wskazuje bufor indeksu. Aby go wypełnić, musimy przeprowadzić podobną operację jak w przypadku bufora wierzchołka. Rozpoczynamy od tablicy Java i konwertujemy ją za pomocą pakietu `java.nio` do bufora natywnego.

Na listingu 20.5 został przedstawiony przykładowy kod służący do konwersji krótkiej tablicy zawierającej elementy {0, 1, 2} do bufora natywnego, nadającego się do przekazania do metody `glDrawElements`.

#### **Listing 20.5.** Konwertowanie tablicy Java na postać bufora nio

---

```
//Oznaczamy sposób rozmieszczenia punktów
short[] myIndecesArray = {0,1,2};

//Uzyskujemy bufor typu short
java.nio.ShortBuffer mIndexBuffer;

//Wyznaczamy po 2 bajty dla każdej wartości indeksu
ByteBuffer ibb = ByteBuffer.allocateDirect(3 * 2);
ibb.order(ByteOrder.nativeOrder());
mIndexBuffer = ibb.asShortBuffer();

//Umieszczamy go w buforze
for (int i=0;i<3;i++)
{
    mIndexBuffer.put(myIndecesArray[i]);
}
```

---

Skoro wiemy już, jak działa metoda `mIndexBuffer` (listing 20.5), możemy cofnąć się do listingu 20.4, aby lepiej zrozumieć ideę tworzenia bufora indeksu i jego przekształcania.

**Uwaga!**

Bufor indeksu, zamiast tworzyć nowe punkty, indeksuje jedynie macierz punktów wskazaną przez metodę `glVertexPointer`. Jest to możliwe, ponieważ biblioteka OpenGL zapamiętuje za pomocą stanów zasoby ustanowione przez poprzednie wywołania.

Zajmijmy się teraz dwiema powszechnie wykorzystywanyimi metodami biblioteki OpenGL: `glClear` i `glColor`.

### **glClear**

Metodę `glClear` stosujemy do czyszczenia powierzchni rysowania. Za jej pomocą możemy zerować kolor, głębię oraz rodzaj wykorzystywanych szablonów. Zerowany element określamy poprzez odpowiednią stałą: `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT` lub `GL_STENCIL_BUFFER_BIT`.

Bufor koloru jest odpowiedzialny za widoczne na ekranie piksele, zatem jego wyczyszczenie spowoduje zniknięcie wszelkich kolorów z powierzchni. Bufer głębi jest związany z pikselami widzianymi w trójwymiarowej scenie, w zależności od odległości obiektu od kamery.

Bufor szablonowy jest nieco zbyt skomplikowany, aby go tutaj omówić, wystarczy jednak wiezieć, że jest używany do tworzenia efektów graficznych na podstawie pewnych dynamicznych kryteriów, a metoda `glClear` umożliwia jego wyczyszczenie.

**Uwaga!**

Szablon jest obiektem, dzięki któremu możemy wielokrotnie powielać proces rysowania. Jeśli na przykład używamy aplikacji Microsoft Office Visio, wszystkie obiekty zapisywane jako pliki \*.vss są szablonami. W świecie rzeczywistym tworzymy szablon poprzez wycięcie wzoru w papierze lub innej płaskiej powierzchni. Następnie przerysowujemy za pomocą szablonu jego obwiednię na arkuszu, a po zdaniu szablonu powstaje wrażenie powielenia rysunku. Widziany obraz zależy od aktywnych szablonów. Wyczyszczenie wszystkich szablonów spowoduje, że wszystkie rysowane elementy będą widoczne.

Dla naszych celów możemy zastosować poniższy kod do wyczyszczenia bufora koloru:

```
//Czyści powierzchnię ze wszelkich kolorów
glClear(gl.GL_COLOR_BUFFER_BIT);
```

Zastanówmy się teraz nad sposobem dołączenia koloru do rysowanego obiektu.

## glColor

Metoda glColor jest używana do ustawienia domyślnego koloru dla następnego rysowanego obiektu. W poniższym segmencie kodu metoda glColor4f generuje kolor czerwony:

```
//Ustanawia bieżący kolor
glColor4f(1.0f, 0, 0, 0.5f);
```

Przypomnijmy sobie nomenklaturę metod: oznaczenie **4f** odnosi się do czterech argumentów, posiadających wartości zmiennoprzecinkowe, pobieranych przez metodę. Czterema argumentami są składowe barwy czerwonej, zielonej, niebieskiej oraz współczynnika alfa (gradient koloru). Wartością początkową dla tych argumentów jest (1, 1, 1, 1). W naszym przykładzie wybraliśmy kolor czerwony z połową gradientu (określonego przez ostatni argument alfa).

Chociaż omówiliśmy podstawowe interfejsy API rysowania, musimy jeszcze przedstawić kilka kwestii związanych ze współrzędnymi punktów określanych w trójwymiarowej przestrzeni. W kolejnym punkcie wyjaśnimy, w jaki sposób biblioteka OpenGL modeluje rzeczywiste sceny w perspektywie widzenia operatora kamery.

## Kamera i współrzędne

W procesie rysowania w przestrzeni trójwymiarowej musimy w pewnym momencie rzutować trójwymiarowy widok na dwuwymiarowy ekran — tak samo jak w świecie rzeczywistym podczas rejestrowania trójwymiarowej sceny za pomocą kamery. Taka symbolika jest formalnie uznana w standardzie OpenGL, zatem wiele koncepcji jest w nim wyjaśnianych za pomocą pojęcia kamery.

Jak zobaczymy w tym punkcie, widoczna część obiektu rysowanego zależy od położenia kamery, kierunku ustawienia jej obiektywu, orientacji kamery (może być na przykład postawiona do góry nogami lub przekręcona), poziomu przybliżenia oraz rozmiaru „kliszy”.

Wymienione aspekty rzutowania obrazu trójwymiarowego na dwuwymiarowy ekran są kontrolowane przez trzy metody:

- gluLookAt — kontroluje kierunek, w jakim jest ustawiony obiektyw kamery.
- glFrustum — kontroluje objętość widzenia, poziom powiększenia lub odległość (od bądź do obiektu).
- glViewport — kontroluje rozmiar ekranu lub „kliszy”.

Bez zrozumienia znaczenia tych trzech interfejsów API nie można niczego zaprogramować w standardzie OpenGL. Rozwiniemy dalej symbolikę dotyczącą kamery, aby wyjaśnić, w jaki sposób te trzy metody wpływają na obraz widziany na ekranie. Rozpoczniemy od metody `gluLookAt`.

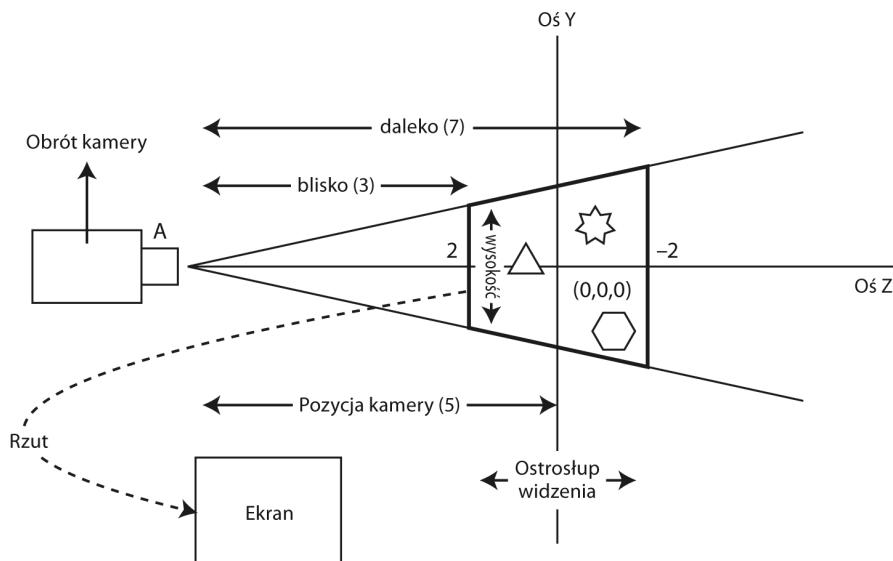
## gluLookAt i symbolika kamery

Wyobraźmy sobie, że fotografujemy krajobraz, którego elementem są kwiaty, drzewa, strumienie i góry. Przybywamy na łąkę; sceneria roztaczająca się przed naszymi oczami jest równoważna temu, co chcemy narysować w standardzie OpenGL. Możemy rysować obiekty tak duże jak góry lub tak małe jak kwiaty — dopóki zachowamy proporcje pomiędzy nimi. Jak już wcześniej wspomnieliśmy, współrzędne stosowane w przypadku tych obiektów są nazywane **współrzędnymi świata**. Za ich pomocą ustaliamy na osi  $x$  linię o długości 4 jednostek poprzez wprowadzenie punktów od  $(1, 0, 0)$  do  $(4, 0, 0)$ .

Podczas przygotowań do wykonania zdjęcia znajdujemy miejsce, w którym umieścimy statyw. Następnie na statwie montujemy aparat. Położenie aparatu — nie statwu, a samego urządzenia — jest punktem początkowym. Musimy więc wziąć kartkę i zaznaczyć tę lokalizację, noszącą nazwę **punktu ocznego** (ang. *eye point*).

Jeżeli nie zdefiniujemy punktu ocznego, kamera będzie się znajdować w punkcie o współrzędnych  $(0, 0, 0)$ , który jest umieszczony dokładnie na środku ekranu. Chcemy przeważnie odsunąć się od początku układu współrzędnych, aby zobaczyć płaszczyznę  $(x, y)$ , dla której wartość osi  $z$  wynosi 0. Założymy, że umieścimy aparat w punkcie  $(0, 0, 5)$ . Zostanie on „przesunięty” w naszą stronę o 5 jednostek.

Na rysunku 20.1 została ukazana pozycja aparatu.



Rysunek 20.1. Analogia kamery w koncepcji widzenia w standardzie OpenGL

Przyglądając się rysunkowi 20.1, możemy się zastanawiać, dlaczego mamy do czynienia z osiami  $x$  i  $z$ , a nie  $x$  i  $y$ . Stosujemy tu standardową konwencję biblioteki OpenGL, zgodnie z którą kamera jest skierowana na osią  $z$ , gdy płaszczyznę scenerii tworzą osie  $x$  i  $y$ . Taka konwencja sprawdza się, ponieważ zazwyczaj osią  $z$  zostaje powiązana z osią głębi.

Po umieszczeniu aparatu na statwie musimy sprawdzić, jaki fragment sceny chcemy uchwycić w obiektywie. Skierujmy obiektyw urządzenia w stronę, w którą patrzymy. Ten odległy punkt, na który spoglądamy, nazywany jest **punktem widoku** (ang. *viewing point*) lub **punktem spoglądania** (ang. *look-at point*). Określając ten punkt, definiujemy w rzeczywistości kierunek patrzenia. Jeśli nasz punkt widoku będzie posiadał współrzędne  $(0, 0, 0)$ , to aparat będzie „spoglądał” z odległości 5 — przy założeniu, że współrzędne aparatu wynoszą  $(0, 0, 5)$  — jednostek wzduż osi z na początek układu współrzędnych. Zostało to ukazane na rysunku 20.1.

Wyobraźmy sobie dalej, że w punkcie początkowym układu współrzędnych został umieszczony prostopadłościenny budynek. Chcemy zrobić mu zdjęcie nie w pozycji wertykalnej, a w horyzontalnej. Co robimy? Oczywiście nie zmieniamy położenia ani kierunku spoglądania aparatu, musimy go jednak obrócić o 90 stopni (analogicznie do przechylania głowy na boki). Jest to **orientacja aparatu „spoglądającego”** na określony punkt widoku. Orientacja taka nosi nazwę **wektora góry** (ang. *up vector*).

Wektor góry w prosty sposób określa orientację aparatu (góra, dół, lewo, prawo lub pod kątem). Orientacja aparatu jest również określana za pomocą punktu. Wyobraźmy sobie linię odchodzączą od środka układu współrzędnych — nie od środka aparatu, a od środka układu współrzędnych świata — do tego punktu. Kąt utworzony pomiędzy osiami a tą linią wyznacza właśnie orientację aparatu.

Na przykład wektor góry dla aparatu może przybrać wartość  $(0, 1, 0)$ , a nawet  $(0, 15, 0)$ , co da ten sam efekt. Punkt  $(0, 1, 0)$  wskazuje punkt odchodzący od początku układu współrzędnych po osi  $y$  w góre. Oznacza to, że ustawiemy aparat w pozycji pionowej. W przypadku wektora  $(0, -1, 0)$  obrócimy urządzenie do góry nogami. W obydwu przypadkach aparat umieszczony jest ciągle w tym samym punkcie  $(0, 0, 5)$  i „spogląda” na środek układu współrzędnych  $(0, 0, 0)$ . Wymienione współrzędne możemy podsumować w następujący sposób:

- **$(0, 0, 5)$**  — punkt oczny (położenie kamery).
- **$(0, 0, 0)$**  — punkt spoglądania (kierunek, w którym kamera jest zwrócona).
- **$(0, 1, 0)$**  — wektor góry (orientacja pozioma, pionowa lub nachylona).

Wszystkie trzy punkty — punkt oczny, punkt spoglądania oraz wektor góry — mogą zostać zdefiniowane w metodzie `gluLookAt` w następujący sposób:

```
gluLookAt(gl, 0,0,5,      0,0,0,      0,1,0);
```

Argumenty występują w kolejności: pierwszy zbiór współrzędnych odpowiada punktowi oczemu, drugi zestaw współrzędnych należy do punktu spoglądania, natomiast pozostałe trzy współrzędne określają wektor góry w odniesieniu do początku układu współrzędnych.

Przyjrzyjmy się teraz objętości widzenia.

## glFrustum i objętość widzenia

Można zauważyc, że żaden z punktów opisujących położenie aparatu za pomocą metody `gluLookAt` nie określa rozmiaru obrazu. Definiują one jedynie lokalizację, kierunek i orientację. W jaki sposób aparat ma ustawać ostrość? Jak daleko znajduje się obiekt, który fotografujemy? Do określenia interesującego nas obszaru sceny wykorzystujemy metodę `glFrustum`.

Gdybyśmy siedzieli w teatrze na przedstawieniu, scena stanowiłaby naszą *objętość widzenia*. Nie musimy wiedzieć, co się dzieje za kurtynami. Ważne są jednak dla nas rozmiary sceny, ponieważ chcemy widzieć dokładnie wszystko, co się na niej dzieje.

Wyobraźmy sobie obszar sceny otoczony przez bryłę, zwaną także **ostrosłupem widzenia** lub **objętością widzenia** (ostrosłup widzenia jest zaznaczony pogrubioną linią w środkowej części rysunku 20.1). Wszystkie elementy umieszczone wewnątrz bryły zostają zarejestrowane, a obiekty znajdujące się na zewnątrz zostają wycięte i zignorowane. Zatem w jaki sposób określamy bryłę widzenia? Najpierw wyznaczamy **bliski punkt** (ang. *near point*) lub odległość pomiędzy aparatem a początkiem bryły. Następnie zaznaczamy **daleki punkt** (ang. *far point*), definiujący dystans pomiędzy aparatem a końcem bryły. Odległość pomiędzy punktem bliskim a punktem dalekim wzduż osi z stanowi głębię bryły. Jeżeli zdefiniujemy punkt bliski o wartości 50 i punkt daleki o wartości 200, uchwycimy wszystkie obiekty znajdujące się pomiędzy tymi punktami, a głębia bryły będzie posiadała wartość 150. Będziemy musieli określić również lewą stronę bryły, jej prawą stronę, a także jej górę i dół za pomocą wyimaginowanego **promienia**, łączącego aparat z punktem spoglądania.

W bibliotece OpenGL istnieją dwa sposoby odwzorowania tej wyimaginowanej bryły. Jeden z nich nosi nazwę **rzutowania perspektywicznego** i wykorzystuje omówione przed chwilą pojęcie ostrosłupa widzenia. Widok ten, symulujący pracę normalnej kamery, wykorzystuje strukturę piramidową, której podstawę stanowi daleka płaszczyzna, a kamera jest jej wierzchołkiem. Płaszczyzna bliska odciina szczyt piramidy, co powoduje powstanie ostrosłupa ściętego pomiędzy płaszczyzną bliską a daleką.

Drugi sposób wyobrażenia tej bryły wymaga postrzegania jej w postaci sześciangu. Ten drugi scenariusz nosi nazwę **rzutowania ortograficznego** i jest wykorzystywany do rysowania obiektów geometrycznych, które muszą zachować rozmiary bez względu na odległość od kamery.

Listing 20.6 prezentuje nam sposób, w jaki definiujemy ostrosłup widzenia dla naszego przykładu.

---

#### **Listing 20.6.** Definiowanie ostrosłupa widzenia za pomocą metody glFrustum

---

```
//Oblicza najpierw proporcje obrazu
float ratio = (float) w / h;
//Wskazuje na fakt, że wymagamy rzutowania perspektywicznego
glMatrixMode(GL10.GL_PROJECTION);
//Definiuje ostrosłup widzenia: objętość widzenia
gl.glFrustumf(
    -ratio,                      // Lewa strona bryły widzenia
    ratio,                        // Prawa strona bryły widzenia
    1,                            // Szczyt bryły widzenia
    -1,                            // Spód bryły widzenia
    3,                            // Odległość przedniej ściany bryły od aparatu
    7);                           // Odległość tylnej ściany bryły od aparatu
```

---

Ponieważ w kodzie z listingu 20.6 przypisaliśmy szczytowi bryły wartość 1, a jej spodniej ścianie wartość -1, wysokość przedniej ściany wynosi 2 jednostki. Rozmiary lewej i prawej strony ostrosłupa określamy za pomocą proporcjonalnych liczb, biorąc pod uwagę proporcję obrazu. Z tego właśnie powodu kod wykorzystuje wysokość i szerokość okna do określenia proporcji. Zakłada on również, że obszar działania będzie znajdował się pomiędzy 3. a 7. jednostką wzduż osi z. Wszystkie obiekty narysowane poza tymi współrzędnymi, względnymi wobec aparatu, będą niewidoczne.

Ponieważ umieściliśmy aparat w punkcie (0, 0, 5) i skierowaliśmy go w stronę punktu (0, 0, 0), punkt zlokalizowany trzy jednostki od aparatu w stronę początku układu współrzędnych znajduje się w punkcie (0, 0, 2), a siedem jednostek od kamery znajduje się punkt (0, 0, -2). W ten sposób płaszczyzna początku układu współrzędnych znajduje się dokładnie pośrodku trójwymiarowej bryły.

Teraz już wiemy, jak wielka jest nasza objętość widzenia. Istnieje jeszcze jeden interfejs API, odwzorowujący te rozmiary na ekranie — `glViewport`.

## glViewport i rozmiar ekranu

Metoda `glViewport` pozwala na zdefiniowanie prostokątnego obszaru ekranu, na który będzie rzutowana objętość widzenia. Przyjmuje ona cztery argumenty określające prostokątnie pole: współrzędne *x* i *y* lewego dolnego rogu figury oraz szerokość i wysokość. Na listingu 20.7 zaprezentowano przykład określenia widoku jako celu rzutowania.

**Listing 20.7.** Definiowanie wziernika za pomocą metody `glViewport`

---

```
glViewport(0,           // Współrzędna x lewej dolnej krawędzi prostokąta
          0,           // Współrzędna y lewej dolnej krawędzi prostokąta
          width,        // Szerokość prostokąta na ekranie
          height);    // Wysokość prostokąta na ekranie
```

---

Jeżeli wysokość naszego okna lub widoku wynosi 100 pikseli, a wysokość ostrosłupa widzenia wynosi 10 pikseli, to każda jednostka logiczna współrzędnych zostanie przetłumaczona na 10 pikseli współrzędnych świata.

Dotychczas omówiliśmy niektóre podstawowe, istotne pojęcia dotyczące grafiki OpenGL. Zrozumienie tych podstaw przyda się podczas nauki programowania za pomocą biblioteki OpenGL. Po spełnieniu tego warunku możemy rozpocząć omawianie elementów wymaganych do wywołania opisanych powyżej interfejsów API.

## Tworzenie interfejsu pomiędzy standardem OpenGL ES a Androidem

Jak już zdążyliśmy wspomnieć, standard OpenGL ES jest obsługiwany na wielu rodzajach platform. U jego podstaw znajduje się, przypominający strukturę języka C, interfejs API, który zapewnia obsługę wszystkich aspektów rysowania. Jednak platformy i systemy operacyjne różnią się między sobą pod względem implementacji wyświetlania, buforów ekranu i tym podobnych elementów. Te specyficzne dla każdego systemu operacyjnego aspekty są przetwarzane i dokumentowane przez te systemy. Android nie jest pod tym względem wyjątkiem.

Począwszy od wersji 1.5 środowiska SDK, w Androidzie uproszczono procesy interakcji z funkcjami OpenGL oraz inicjalizacji rysowania w standardzie OpenGL. Jest to możliwe dzięki pakietowi `android.opengl`. Główną klasą zawierającą wiele funkcji jest `GLSurfaceView`, która posiada wewnętrzny interfejs `GLSurfaceView.Renderer`. Znajomość tych dwóch elementów wystarczy do poczynienia znaczących postępów na polu programowania w standardzie OpenGL w Androidzie.

## Stosowanie klasy GLSurfaceView i klas pokrewnych

Począwszy od wersji 1.5 środowiska SDK, powszechny wzorzec stosowania biblioteki OpenGL został znacznie uproszczony. Podczas rysowania za pomocą klas biblioteki OpenGL stosujemy zazwyczaj następujący algorytm:

1. Zaimplementuj interfejs Renderer.
2. Skonfiguruj w implementacji interfejsu Renderer ustawienia klasy Camera.
3. W implementacji wprowadź do metody onDrawFrame kod odpowiedzialny za rysowanie.
4. Skonstruuj widok GLSurfaceView.
5. Skonfiguruj silnik renderujący, utworzony w punktach 1. – 3., wewnątrz klasy GLSurfaceView.
6. Określ, czy jest wymagana animacja w klasie GLSurfaceView.
7. Skonfiguruj kontrolkę GLSurfaceView w aktywności jako widok treści. Możemy również używać tego widoku wszędzie tam, gdzie korzystamy ze zwykłego widoku.

Zacznijmy od implementacji silnika renderującego.

## Implementacja klasy Renderer

Sygnatura tego interfejsu została ukazana na listingu 20.8.

**Listing 20.8.** Interfejs Renderer

---

```
public static interface GLSurfaceView.Renderer
{
    void onDrawFrame(GL10 gl);
    void onSurfaceChanged(GL10 gl, int width, int height);
    void onSurfaceCreated(GL10 gl, EGLConfig config);
}
```

---

Główny proces rysowania przebiega w metodzie onDrawFrame(). Zawsze podczas tworzenia nowej powierzchni dla tego widoku zostaje wywołana metoda onSurfaceCreated(). Możemy wywołać wiele interfejsów API biblioteki OpenGL, takich jak roztrząsanie (ang. *dithering*), kontrola głębi oraz inne, które można wywołać bezpośrednio spoza metody onDrawFrame().

Analogicznie w przypadku zmiany powierzchni, na przykład szerokości i wysokości okna, zostaje wywołana metoda onSurfaceChanged(). Dzięki niej możemy konfigurować kamerę oraz objętość widzenia.

Nawet w metodzie onDrawFrame() istnieje wiele elementów, które mogą być wspólne dla określonego kontekstu rysowania. Możemy wykorzystać tę powszedniość i umieścić te metody na kolejnym poziomie abstrakcji, zwanym AbstractRenderer, zawierającym tylko jedną niezaimplementowaną metodę draw().

Na listingu 20.9 został zaprezentowany kod klasy AbstractRenderer.

**Listing 20.9.** Klasa AbstractRenderer

---

```
//nazwa pliku: AbstractRenderer.java
import android.opengl.*;
//...Za pomocą środowiska Eclipse wprowadźmy pozostałe instrukcje importu
```

```

public abstract class AbstractRenderer
implements android.opengl.GLSurfaceView.Renderer
{
    public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig) {
        gl.glDisable(GL10.GL_DITHER);
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,
                  GL10.GL_FASTEST);
        gl.glClearColor(.5f, .5f, .5f, 1);
        gl.glShadeModel(GL10.GL_SMOOTH);
        gl.glEnable(GL10.GL_DEPTH_TEST);
    }

    public void onSurfaceChanged(GL10 gl, int w, int h) {
        gl.glViewport(0, 0, w, h);
        float ratio = (float) w / h;
        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7);
    }

    public void onDrawFrame(GL10 gl)
    {
        gl.glDisable(GL10.GL_DITHER);
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
        GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 1.0f, 0.0f);
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        draw(gl);
    }
    protected abstract void draw(GL10 gl);
}

```

Klasa ta jest bardzo pożyteczna, ponieważ pozwala na skupienie się jedynie na metodach rysowania. Wykorzystamy ją do utworzenia naszej prostej klasy `SimpleTriangleRenderer`; listing 20.10 ukazuje jej kod źródłowy.

#### **Listing 20.10.** Klasa `SimpleTriangleRenderer`

---

```

//nazwa pliku: SimpleTriangleRenderer.java
public class SimpleTriangleRenderer extends AbstractRenderer
{
    //Liczba używanych punktów lub wierzchołków
    private final static int VERTS = 3;

    //Nieskompresowany, natywny bufor przechowujący współrzędne punktów
    private FloatBuffer mFVertexBuffer;

    //Nieskompresowany, natywny bufor przechowujący indeksy
    //pozwalające na wielokrotne wykorzystywanie punktów.
    private ShortBuffer mIndexBuffer;

    public SimpleTriangleRenderer(Context context)
    {

```

```
ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
vbb.order(ByteOrder.nativeOrder());
mFVertexBuffer = vbb.asFloatBuffer();

ByteBuffer ibb = ByteBuffer.allocateDirect(VERTS * 2);
ibb.order(ByteOrder.nativeOrder());
mIndexBuffer = ibb.asShortBuffer();

float[] coords = {
    -0.5f, -0.5f, 0, // (x1, y1, z1)
    0.5f, -0.5f, 0,
    0.0f, 0.5f, 0
};
for (int i = 0; i < VERTS; i++) {
    for(int j = 0; j < 3; j++) {
        mFVertexBuffer.put(coords[i*3+j]);
    }
}
short[] myIndecesArray = {0,1,2};
for (int i=0;i<3;i++)
{
    mIndexBuffer.put(myIndecesArray[i]);
}
mFVertexBuffer.position(0);
mIndexBuffer.position(0);
}

//przesłonięta metoda
protected void draw(GL10 gl)
{
    gl.glColor4f(1.0f, 0, 0, 0.5f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
    gl.glDrawElements(GL10.GL_TRIANGLES, VERTS,
                      GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
}
```

---

Chociaż wydaje się, że powyższy listing ma znaczne rozmiary, większość kodu służy do definiowania wierzchołków i konwertowania ich z buforów kodu języka Java do buforów nio. Sama metoda draw składa się jedynie z trzech wierszy: ustanowienia koloru, ustanowienia wierzchołków i rysowania.

**Uwaga!**

W naszym kodzie nigdy nie uwalniamy buforów nio, chociaż przydzielimy im pamięć. Zatem w jaki sposób są one uwalniane? Jak wykorzystanie tej pamięci wpływa na bibliotekę OpenGL?

Na podstawie badań stwierdziliśmy, że pakiet `java.nio` przydziela przestrzeń pamięci spoza stosu Java. Pamięć ta może być bezpośrednio wykorzystana przez takie systemy, jak OpenGL, File I/O i tak dalej. W rzeczywistości buforey nio są obiektami Java, które ostatecznie wskazują na bufor natywny. Te obiekty nio są odśmiecane (ang. *garbage collection — gc*). Oznacza to, że po wykonaniu pracy oczyszczają pamięć natywną. Programy Java nie muszą przeprowadzać żadnych specjalnych operacji, aby zwolnić pamięć.

Jednak proces odśmiecania nie zostanie przeprowadzony, dopóki w stosie Java istnieje wykorzystywana pamięć. Oznacza to, że mimo wykorzystania pamięci natywnej proces *gc* może nie zostać uruchomiony. W internecie można znaleźć informacje na temat wyjątku braku pamięci uruchamiającego odśmiecianie. Po wystąpieniu tego wyjątku można sprawdzić, czy pamięć stała się już dostępna.

W warunkach standardowych — to jest istotne w przypadku biblioteki OpenGL — możemy przydzielać bufore natywne i nie musimy się martwić jawnym zwalnianiem przydzielonej pamięci, ponieważ czyszczenie pamięci wykonuje proces *gc*.

Skoro uzyskaliśmy już przykładowy silnik renderujący, zobaczymy, w jaki sposób możemy go dostarczyć klasie *GLSurfaceView* i wyświetlić w aktywności.

## Zastosowanie klasy *GLSurfaceView* z poziomu aktywności

Na listingu 20.11 przedstawiono typową aktywność, wykorzystującą klasę *GLSurfaceView* wraz z odpowiednim silnikiem renderującym.

**Listing 20.11.** Proste środowisko testowe biblioteki OpenGL, nazwane *OpenGLTestHarnessActivity*

```
public class OpenGLTestHarnessActivity extends Activity {
    private GLSurfaceView mTestHarness;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mTestHarness = new GLSurfaceView(this);
        mTestHarness.setEGLConfigChooser(false);
        mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
        mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
        //mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
        setContentView(mTestHarness);
    }
    @Override
    protected void onResume() {
        super.onResume();
        mTestHarness.onResume();
    }
    @Override
    protected void onPause() {
        super.onPause();
        mTestHarness.onPause();
    }
}
```

Wyjaśnijmy kilka kluczowych elementów tego kodu źródłowego. Fragment, który tworzy widok *GLSurfaceView*, wygląda następująco:

```
mTestHarness = new GLSurfaceView(this);
```

Następny wiersz oznacza, że nie wymagamy wyboru specjalnej konfiguracji biblioteki EDL i że wystarczą ustawienia domyślne:

```
mTestHarness.setEGLConfigChooser(false);
```

Kolejnym etapem jest skonfigurowanie silnika renderowania:

```
mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
```

Jedna z dwóch metod umieszczonych w dalszej części kodu umożliwia proces animacji:

```
mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
//mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
```

W przypadku wyboru pierwszego z powyższych dwóch wierszy proces rysowania zostanie wywołany tylko jeden raz, a ściślej mówiąc, jeden raz po każdym wywołaniu metody. Wybór drugiej opcji spowoduje, że kod rysowania będzie wykonywany nieprzerwanie, co można wykorzystać do animowania obiektów rysowanych.

To tyle na temat korzystania z interfejsów biblioteki OpenGL w Androidzie.

Teraz Czytelnik posiada wszystkie elementy niezbędne do przetestowania procesu rysowania. Aktywność zaprezentowaliśmy na listingu 20.11; abstrakcyjny silnik renderujący pokazaliśmy na listingu 20.9, a samą klasę `SimpleTriangleRenderer` — na listingu 20.10. Teraz musimy jedynie wywołać klasę aktywności poprzez dowolny element menu w sposób przedstawiony poniżej:

```
private void invokeSimpleTriangle()
{
    Intent intent = new Intent(this,OpenGLTestHarnessActivity.class);
    startActivity(intent);
}
```

Oczywiście niezbędne jest zarejestrowanie aktywności w pliku manifeście Androida, na przykład tak:

```
<activity android:name=".OpenGLTestHarnessActivity"
        android:label="Środowisko testowe OpenGL"/>
```

Chociaż utworzenie samodzielnej aktywności, takiej jak `OpenGLTestHarnessActivity` z listingu 20.11, jest całkowicie rozsądnym zachowaniem, chcielibyśmy zaproponować alternatywę, która będzie o wiele lepiej pasowała do tematyki poruszanej w niniejszym rozdziale.

Nasza propozycja jest związana z umieszczeniem wielu przykładowych fragmentów kodu w obrębie rozdziału. Gdybyśmy osobno przedstawiali aktywności dla poszczególnych aplikacji, zapelniliśmy rozdział kodami bardzo podobnymi do przedstawionego na listingu 20.11, które nie wprowadzałyby niczego nowego. Ponadto każda taka aktywność musi być zarejestrowana w pliku manifeście.

Mając to na uwadze, utworzymy ogólną aktywność pozwalającą na testowanie wszystkich omawianych tu przykładowych projektów. Jej kod został umieszczony na listingu 20.12. Może się on wydawać dość rozbudowany w porównaniu do pierwotnej wersji, jeżeli jednak przyjrzymy się odpowiedzi menu zawartej w zasobie `R.id.mid_OpenGL_SimpleTriangle`, zauważymy, że zasadniczo zachowanie aplikacji nie ulega zmianie. Wraz ze zwiększeniem liczby elementów menu rośnie liczba instrukcji `if`, po jednej dla każdego przykładu.

Pozostałe opcje menu będą omawiane w dalszej części rozdziału. Po kodzie z listingu 20.12 zaprezentujemy zawartość pliku `main_menu.xml`, po czym dokładniej omówimy naszą wielozadaniową aktywność.

**Listing 20.12.** Aktywność MultiViewTestHarness

```
//nazwa pliku: MultiViewTestHarnessActivity.java
public class MultiViewTestHarnessActivity extends Activity {
    private GLSurfaceView mTestHarness;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mTestHarness = new GLSurfaceView(this);
        mTestHarness.setEGLConfigChooser(false);

        Intent intent = getIntent();
        int mid = intent.getIntExtra("com.ai.menuid", R.id.mid_OpenGL_Current);
        if (mid == R.id.mid_OpenGL_SimpleTriangle)
        {
            mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
            mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
            setContentView(mTestHarness);
            return;
        }
        if (mid == R.id.mid_OpenGL_Current)
        {
            //Wywołuje inny silnik renderujący OpenGL
            //i
            //wraca;
        }
        //w przeciwnym wypadku wykonuje poniższą czynność
        mTestHarness.setRenderer(new SimpleTriangleRenderer(this));
        mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
        setContentView(mTestHarness);
        return;
    }
    @Override
    protected void onResume() {
        super.onResume();
        mTestHarness.onResume();
    }
    @Override
    protected void onPause() {
        super.onPause();
        mTestHarness.onPause();
    }
}
```

Kod menu z listingu 20.13 obsługuje aktywność widoczną na listingu 20.12. Dokładniej mówiąc, jest to plik *res/menu/main\_menu.xml*. Wyprzedziliśmy nieco fakty i utworzyliśmy wszystkie elementy menu dotyczące przykładów zamieszczonych w tym rozdziale.

**Listing 20.13.** Plik głównego menu

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa wykorzystuje domyślną kategorię. -->
    <group android:id="@+id/menuGroup_Main">
```

```
<item android:id="@+id/mid_OpenGL_SimpleTriangle"
      android:title="Prosty trójkąt" />

<item android:id="@+id/mid_OpenGL_SimpleTriangle2"
      android:title="Dwa trójkąty" />

<item android:id="@+id/mid_OpenGL_AnimatedTriangle"
      android:title="Animowany trójkąt" />

<item android:id="@+id/mid_rectangle"
      android:title="Prostokąt" />

<item android:id="@+id/mid_square_polygon"
      android:title="Kwadrat" />

<item android:id="@+id/mid_polygon"
      android:title="Wielokąt" />

<item android:id="@+id/mid_textured_square"
      android:title="Teksturowany kwadrat" />

<item android:id="@+id/mid_textured_polygon"
      android:title="Teksturowany wielokąt" />

<item android:id="@+id/mid_multiple_figures"
      android:title="Wiele figur" />

<item android:id="@+id/mid_OpenGL_Current"
      android:title="Bieżąca" />

<item android:id="@+id/mid_es20_triangle"
      android:title="Trójkąt w wersji ES20" />
</group>
</menu>
```

---

Przyglądając się zawartości pliku menu, możemy się domyślać, jakie silniki renderujące zostaną zademonstrowane. Jeżeli cofniemy się do aktywności z listingu 20.12, stwierdzimy, że przełączka ona silniki renderowania na podstawie zdefiniowanych w tym pliku identyfikatorów menu.

W jaki sposób ta aktywność uzyskuje identyfikatory poszczególnych elementów menu? Odbywa się to dzięki następującemu fragmentowi kodu (został on skopiowany z listingu 20.12):

```
Intent intent = getIntent();
int mid = intent.getIntExtra("com.ai.menuid",
    R.id.mid_OpenGL_Current);
```

Dzięki powyższemu fragmentowi kodu intencja przywołująca aktywność przekazuje dodatkowe dane, zwane com.ai.menuid. Jeżeli dane te są nieobecne, kod użyje identyfikatora menu mid\_opengl\_current, który stanie się domyślnym identyfikatorem.

Co powoduje wstawianie dodatkowych danych do intencji? Gdzie się znajduje przywołująca, sterująca aktywnością? Została ona zaprezentowana na listingu 20.14.

#### **Listing 20.14. Aktywność TestOpenGLMainDriver**

---

```
public class TestOpenGLMainDriverActivity extends Activity {
    /** Wywoływana podczas pierwszego uruchomienia aktywności. */
    @Override
```

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater(); //z aktywności
    inflater.inflate(R.menu.main_menu, menu);
    return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item)
{
    this.invokeMultiView(item.getItemId());
    return true;
}
private void invokeMultiView(int mid)
{
    Intent intent =
    new Intent(this,MultiViewTestHarnessActivity.class);
    intent.putExtra("com.ai.menuid", mid);
    startActivity(intent);
}
}

```

Do umożliwienia procesu komplikacji potrzebny jest nam jeszcze plik układu graficznego. Został on zaprezentowany na listingu 20.15.

**Listing 20.15.** Układ graficzny aktywności TestOpenGLMainDriver (layout/main.xml)

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Prosta aktywność główna. Kliknij przycisk menu, aby kontynuować"
    />
</LinearLayout>

```

Oczywiście, w systemie Android konieczny jest jeszcze plik manifest. Zapoznamy się z jego kodem na listingu 20.16.

**Listing 20.16.** Plik AndroidManifest.xml

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.OpenGL"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"

```

```
    android:label="Środowisko testowe OpenGL"
    android:debuggable="true">
<activity android:name=".TestOpenGLMainDriverActivity"
    android:label="Środowisko testowe OpenGL">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity android:name="MultiViewTestHarnessActivity"
    android:label="Środowisko testowe OpenGL – wiele widoków"/>
</application>
<uses-sdk android:minSdkVersion="3" />
</manifest>
```

---

Podsumowując, do skompilowania i uruchomienia naszego programu wymagane będą następujące pliki:

- *TestOpenGLMainDriverActivity.java* (główna aktywność sterująca; listing 20.14),
- *AbstractRenderer.java* (listing 20.9),
- *SimpleTriangleRenderer.java* (listing 20.10),
- *MultiViewTestHarnessActivity.java* (listing 20.12),
- *res/menu/main\_menu.xml* (plik menu; listing 20.13),
- *layout/main.xml* (plik układu graficznego; listing 20.15).

Po skompilowaniu i uruchomieniu kodu zobaczymy wyświetlzoną aktywność sterującą. Możemy kliknąć przycisk menu, aby została wyświetlona lista dostępnych opcji, co zostało zaprezentowane na rysunku 20.2.

Jeżeli klikniemy teraz element *Prosty trójkąt*, ujrzymy trójkąt przedstawiony na rysunku 20.3.

## Zmiana ustawień kamery

Aby lepiej zrozumieć znaczenie współrzędnych biblioteki OpenGL, poeksperymentujmy z metodami definiującymi kamerę i sprawdźmy, w jaki sposób wpływają one na wygląd trójkąta z rysunku 20.3. Zapamiętajmy współrzędne jego wierzchołków: (-0.5, -0.5, 0; 0.5, -0.5, 0; 0, 0.5, 0). Za ich pomocą poniższe trzy metody, użyte w obiekcie *AbstractRenderer* (listing 20.9), wygenerowały trójkąt widoczny na rysunku 20.3:

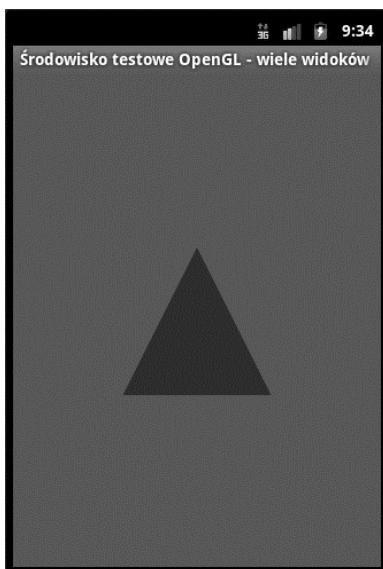
```
//Spogląda na ekran (początek układu współrzędnych) z odległości 5 jednostek od
//przedniej części ekranu
GLU.gluLookAt(gl, 0,0,5, 0,0,0, 0,1,0);

//Ustanawia 2 jednostki wysokości i 4 jednostki głębi
gl.glFrustumf(-ratio, ratio, -1, 1, 3, 7);

//Definiuje standardowe okno
gl.glViewport(0, 0, w, h);
```



Rysunek 20.2. Interfejs aktywności sterującej dla środowiska testowego OpenGL

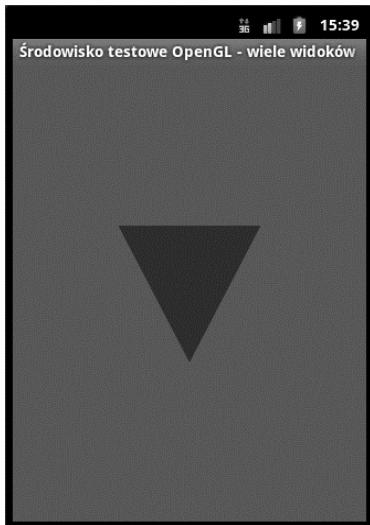


Rysunek 20.3. Prosty trójkąt utworzony za pomocą biblioteki OpenGL

Załóżmy teraz, że przypisujemy przeciwny zwrot wektorowi góry kamery:

```
GLU.gluLookAt(gl, 0,0,5,      0,0,0,      0,-1,0);
```

Jeśli przeprowadzimy taką operację, ujrzymy odwrócony trójkąt z rysunku 20.4. Aby wprowadzić taką zmianę, powinniśmy znaleźć właściwą metodę w pliku *AbstractRenderer.java* (listing 20.9).

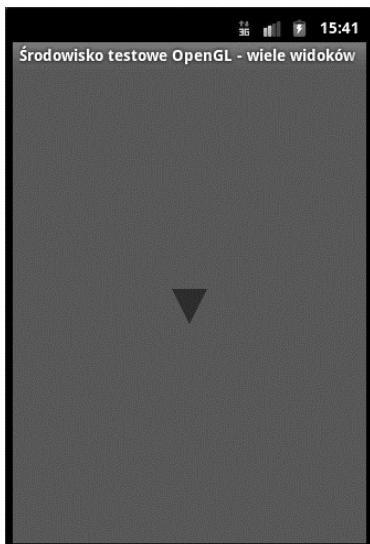


Rysunek 20.4. Trójkąt rejestrowany przez odwróconą kamerę

Spójrzmy teraz, co się stanie, jeśli zmienimy ostrosłup widzenia (zwany także objętością lub bryłą widzenia). Dzięki poniższej linii kodu zwiększa się wysokość i szerokość bryły widzenia o współczynnik 4 (wymiary zostały zilustrowane na rysunku 20.1). Przypominamy, że pierwsze cztery argumenty klasy `glFrustum` definiują przednią ścianę bryły widzenia. Mnożąc każdą wartość przez 4, powiększyliśmy bryłę widzenia czterokrotnie, tak jak poniżej:

```
gl.glFrustumf(-ratio * 4, ratio * 4, -1 * 4, 1 * 4, 3, 7);
```

W wyniku tego kodu trójkąt zostaje zmniejszony, ponieważ jego rozmiary nie uległy zmianie, ale bryła widzenia została powiększona (rysunek 20.5). Wywołanie tej metody pojawia się w klasie `AbstractRenderer.java` (listing 20.9).



Rysunek 20.5. Trójkąt umieszczony w czterokrotnie powiększonej bryle widzenia

## Wykorzystanie indeksów do dodania kolejnego trójkąta

Omówienie tych prostych przykładów z trójkątem zakończymy opisem dziedziczenia z klasy `AbstractRenderer` i utworzenia jeszcze jednego trójkąta poprzez wstawienie dodatkowego punktu i skorzystanie z indeksów. W teorii zdefiniujemy cztery punkty  $(-1, -1; 1, -1; 0, 1; 1, 1)$ . Następnie za pomocą biblioteki OpenGL zaprogramujemy narysowanie wierzchołków  $(0, 1, 2; 0, 2, 3)$ . Listing 20.17 przedstawia odpowiedzialny za to kod (zwróćmy uwagę, że zmieniliśmy rozmiary trójkąta).

**Listing 20.17.** Klasa SimpleTriangleRenderer2

```
//nazwa pliku: SimpleTriangleRenderer2.java
public class SimpleTriangleRenderer2 extends AbstractRenderer
{
    private final static int VERTS = 4;
    private FloatBuffer mFVertexBuffer;
    private ShortBuffer mIndexBuffer;

    public SimpleTriangleRenderer2(Context context)
    {
        ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
        vbb.order(ByteOrder.nativeOrder());
        mFVertexBuffer = vbb.asFloatBuffer();

        ByteBuffer ibb = ByteBuffer.allocateDirect(6 * 2);
        ibb.order(ByteOrder.nativeOrder());
        mIndexBuffer = ibb.asShortBuffer();

        float[] coords = {
            -1.0f, -1.0f, 0, // (x1,y1,z1)
            1.0f, -1.0f, 0,
            0.0f, 1.0f, 0,
            1.0f, 1.0f, 0
        };
        for (int i = 0; i < VERTS; i++) {
            for(int j = 0; j < 3; j++) {
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
        short[] myIndecesArray = {0,1,2, 0,2,3};
        for (int i=0;i<6;i++)
        {
            mIndexBuffer.put(myIndecesArray[i]);
        }
        mFVertexBuffer.position(0);
        mIndexBuffer.position(0);
    }

    protected void draw(GL10 gl)
    {
        gl.glColor4f(1.0f, 0, 0, 0.5f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, 6, GL10.GL_UNSIGNED_SHORT,
                          mIndexBuffer);
    }
}
```

Po utworzeniu klasy SimpleTriangleRenderer2 możemy dodać instrukcję warunkową `if` z listingu 20.18 do aktywności OpenGLTestHarnessActivity (listing 20.12).

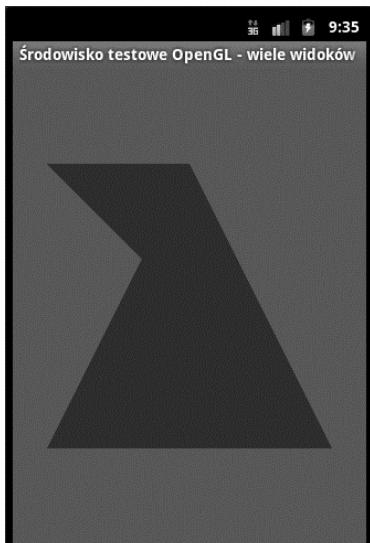
---

**Listing 20.18.** Stosowanie klasy SimpleTriangleRenderer2

```
if (mid == R.id.mid_OpenGL_SimpleTriangle2)
{
    mTestHarness.setRendereder(new SimpleTriangleRendereder2(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(mTestHarness);
    return;
}
```

---

Po dodaniu tego fragmentu możemy ponownie uruchomić aplikację i wybrać tym razem opcję *Dwa trójkąty* (rysunek 20.6). Zwróćmy uwagę, że dzięki konstrukcji klasy MultiViewTestHarness nie ma już konieczności tworzenia nowej aktywności, a tym samym również rejestrowania jej w pliku manifeście. W dalszym ciągu będziemy wykorzystywać ten mechanizm poprzez ustawiczne dodawanie klauzuli `if` dla każdego następnego przykładu.



Rysunek 20.6. Dwa trójkąty utworzone za pomocą czterech punktów

## Animowanie prostego trójkąta w bibliotece OpenGL

Zmiany trybu renderowania widoku GLSurfaceView pozwalają na proste dostosowywanie animacji tworzonych za pomocą biblioteki OpenGL. Na listingu 20.19 przedstawiliśmy przykładowy kod.

---

**Listing 20.19.** Zdefiniowanie trybu ciągłego renderowania

```
//Pobiera widok GLSurfaceView
GLSurfaceView openGLView;
```

---

```
//Ustanawia tryb ciągłego rysowania
openGLView.setRenderingMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
```

---

Zwracamy uwagę na tryb renderowania, ponieważ w poprzednim przykładzie określiliśmy tryb RENDERMODE\_WHEN\_DIRTY (na listingu 20.18). Jak już wspomnieliśmy, tryb RENDERMODE\_→CONTINUOUSLY jest domyślnym ustawieniem, zatem animacja jest dostępna domyślnie.

Jeżeli wybrano tryb rysowania ciągłego, zjawiska wpływające na animację są zależne od metody onDraw silnika renderującego. W celach demonstracyjnych obróćmy wokół własnej osi utworzony wcześniej trójkąt (listing 20.10 i rysunek 20.3).

## AnimatedSimpleTriangleRender

Klasa AnimatedSimpleTriangleRender bardziej przypomina klasę SimpleTriangleRender (listing 20.10), nie licząc tego, co się dzieje w metodzie onDraw. W metodzie tej definiujemy nowy kąt dla obrotu wykonywanego co cztery sekundy. Ponieważ obraz będzie systematycznie przerysowywany, odniesiemy wrażenie powoli obracającego się trójkąta. Listing 20.20 zawiera pełną implementację klasy AnimatedSimpleTriangleRender.

### Listing 20.20. Kod źródłowy klasy AnimatedSimpleTriangleRender

---

```
//nazwa pliku: AnimatedSimpleTriangleRender.java
public class AnimatedSimpleTriangleRender extends AbstractRenderer
{
    private int scale = 1;
    //Liczba używanych punktów lub wierzchołków
    private final static int VERTS = 3;

    //Nieskompresowany bufor natywny, przechowujący współrzędne punktu
    private FloatBuffer mFVertexBuffer;

    //Nieskompresowany bufor natywny, przechowujący indeksy
    //pozwalające na wielokrotne wykorzystywanie punktów.
    private ShortBuffer mIndexBuffer;

    public AnimatedSimpleTriangleRender(Context context)
    {
        ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
        vbb.order(ByteOrder.nativeOrder());
        mFVertexBuffer = vbb.asFloatBuffer();

        ByteBuffer ibb = ByteBuffer.allocateDirect(VERTS * 2);
        ibb.order(ByteOrder.nativeOrder());
        mIndexBuffer = ibb.asShortBuffer();

        float[] coords = {
            -0.5f, -0.5f, 0, // (x1, y1, z1)
            0.5f, -0.5f, 0,
            0.0f, 0.5f, 0
        };
        for (int i = 0; i < VERTS; i++) {
            for(int j = 0; j < 3; j++) {
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
    }
}
```

```
        }
    }
    short[] myIndecesArray = {0,1,2};
    for (int i=0;i<3;i++)
    {
        mIndexBuffer.put(myIndecesArray[i]);
    }
    mFVertexBuffer.position(0);
    mIndexBuffer.position(0);
}

//przesłonięta metoda
protected void draw(GL10 gl)
{
    long time = SystemClock.uptimeMillis() % 4000L;
    float angle = 0.090f * ((int) time);

    gl.glRotatef(angle, 0, 0, 1.0f);

    gl.glColor4f(1.0f, 0, 0, 0.5f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
    gl.glDrawElements(GL10.GL_TRIANGLES, VERTS,
                      GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
}
}
```

---

Po utworzeniu klasy `AnimatedSimpleTriangleRenderer` możemy wstawić widoczną na listingu 20.21 instrukcję warunkową `if` do klasy `MultiViewTestHarness` z listingu 20.12.

---

#### **Listing 20.21.** Korzystanie z klasy `AnimatedSimpleTriangleRenderer`

---

```
if (mid == R.id.mid_OpenGL_AnimatedTriangle)
{
    mTestHarness.setRenderer(new AnimatedSimpleTriangleRenderer(this));
    setContentView(mTestHarness);
    return;
}
```

---

Po dodaniu tego kodu możemy ponownie uruchomić aplikację i wybrać z menu opcję *Animowany trójkąt*, aby ujrzeć obracający się trójkąt widoczny na rysunku 20.3. <<F1-p>>

## **Stawianie czoła bibliotece OpenGL — kształty i tekstury**

W dotychczas ukazanych przykładach definiowaliśmy wierzchołki trójkąta w sposób jawnny. Takie podejście staje się niewygodne w przypadku rysowania kwadratów, pięciokątów, sześciokątów i tak dalej. W ich przypadku potrzebne będą wysokopoziomowe abstrakcje obiektów, takie jak kształty, a nawet grafy sceny, w przypadku których kształty decydują o ich współrzędnych. W taki sposób pokażemy algorytm rysowania dowolnego wielokąta w dowolnym miejscu.

W tym podrozdziale zajmiemy się również teksturami biblioteki OpenGL. Umożliwiają one umieszczanie map bitowych oraz innych obrazów na powierzchni narysowanego obiektu. W celach demonstracyjnych pokażemy, w jaki sposób umieścić tekstury na znanych nam już wielokątach. Przy tej okazji przedstawimy kolejną bardzo istotną czynność wykonywaną w środowisku OpenGL — rysowanie wielu figur geometrycznych lub kształtów za pomocą potoku rysowania.

Znajomość wymienionych powyżej tematów powinna przybliżyć nas do skutecznego tworzenia trójwymiarowych obiektów i scen.

## Rysowanie prostokąta

Zanim przejdziemy do pojęcia kształtów, musimy rozszerzyć znajomość rysowania za pomocą jawnego definiowania wierzchołków na przykładzie prostokąta utworzonego z dwóch trójkątów. W ten sposób również Czytelnik przygotuje się do rozwijania trójkąta do dowolnego wieloboku.

Mamy już wystarczającą wiedzę na temat podstawowego trójkąta, pokażemy więc teraz opatrzony krótkim komentarzem kod służący do utworzenia prostokąta (listing 20.22).

**Listing 20.22.** Silnik renderujący prostokąt

```
public class SimpleRectangleRenderer extends AbstractRenderer
{
    //Liczba używanych punktów lub wierzchołków
    private final static int VERTS = 4;

    //Nieskompresowany bufor natywny, przechowujący współrzędne punktu
    private FloatBuffer mFVertexBuffer;

    //Nieskompresowany bufor natywny, przechowujący indeksy
    //pozwalające na wielokrotne wykorzystywanie punktów.
    private ShortBuffer mIndexBuffer;

    public SimpleRectRenderer(Context context)
    {
        ByteBuffer vbb = ByteBuffer.allocateDirect(VERTS * 3 * 4);
        vbb.order(ByteOrder.nativeOrder());
        mFVertexBuffer = vbb.asFloatBuffer();

        ByteBuffer ibb = ByteBuffer.allocateDirect(6 * 2);
        ibb.order(ByteOrder.nativeOrder());
        mIndexBuffer = ibb.asShortBuffer();

        float[] coords = {
            -0.5f, -0.5f, 0, // (x1, y1, z1)
            0.5f, -0.5f, 0,
            0.5f, 0.5f, 0,
            -0.5f, 0.5f, 0,
        };

        for (int i = 0; i < VERTS; i++) {
            for(int j = 0; j < 3; j++) {
                mFVertexBuffer.put(coords[i*3+j]);
            }
        }
    }
}
```

```
        }
        short[] myIndecesArray = {0,1,2,0,2,3};
        for (int i=0;i<6;i++)
        {
            mIndexBuffer.put(myIndecesArray[i]);
        }
        mFVertexBuffer.position(0);
        mIndexBuffer.position(0);
    }

    //przesłonięta metoda
    protected void draw(GL10 gl)
    {
        gl.glColor4f(1.0f, 0, 0, 0.5f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, 6,
                           GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

---

Zauważmy, że prostokąt jest rysowany w sposób bardzo przypominający rysowanie trójkąta. Zamiast trzech wierzchołków określiliśmy cztery. Wykorzystaliśmy następnie indeksy:

```
short[] myIndecesArray = {0,1,2,0,2,3};
```

Dwukrotnie wykorzystaliśmy ponumerowane wierzchołki (od 0 do 3) w taki sposób, żeby każda trójka wierzchołków utworzyła trójkąt. Zatem wierzchołki (0, 1, 2) tworzą jeden trójkąt, a punkty (0, 2, 3) tworzą drugi. Narysowanie tych dwóch trójkątów za pomocą prostego obiektu `GL_TRIANGLES` w efekcie pozwoliło na uzyskanie prostokąta.

Po utworzeniu tego silnika renderującego możemy dodać w klasie `MultiViewTestHarness` (listing 20.12) instrukcję `if`, zaprezentowaną na listingu 20.23.

---

**Listing 20.23.** Zastosowanie klasy `SimpleRectangleRenderer`

---

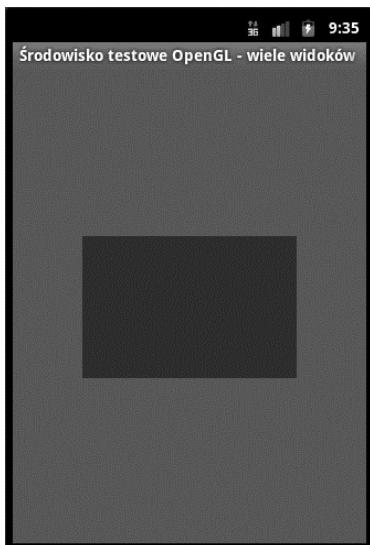
```
if (mid == R.id.mid_rectangle)
{
    mTestHarness.setRenderer(new SimpleRectangleRenderer(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(mTestHarness);
    return;
}
```

---

Po dodaniu powyższego kodu możemy uruchomić ponownie nasz program i tym razem wybrać opcję *Prostokąt*, aby ujrzeć figurę geometryczną ukazaną na rysunku 20.7.

## Praca z kształtami

Jawne określanie wierzchołków figur geometrycznych może być nużącej czynnością. Jeżeli na przykład chcemy narysować dwudziestobok, musimy zdefiniować 20 wierzchołków, gdzie każda definicja wymaga określenia do trzech parametrów. Łącznie narysowanie dwudziestoboku wymaga podania 60 wartości. W przypadku bardziej skomplikowanych rysunków staje się to niewykonalne.



Rysunek 20.7. Prostokąt złożony z dwóch trójkątów, narysowany w środowisku OpenGL

## Wielobok foremny jako kształt

Lepszym sposobem rysowania takich figur, jak trójkąt lub kwadrat, jest zdefiniowanie abstrakcyjnego wieloboku poprzez zdefiniowanie jego różnych parametrów, na przykład współrzędnych środka i długości promienia, co spowoduje utworzenie macierzy jego wierzchołków oraz ewentualnie macierzy indeksów (abyśmy mogli rysować pojedyncze trójkąty). Służąca do tego klasa nosi nazwę `RegularPolygon`. Po utworzeniu takiego obiektu możemy go użyć w kodzie z listingu 20.24 do wygenerowania różnych wieloboków foremnych.

**Listing 20.24.** Zastosowanie klasy `RegularPolygon`

---

```
//Czworobok o promieniu 0.5
//zlokalizowany w punkcie (0, 0, 0) współrzędnych (x, y, z)
RegularPolygon square = new RegularPolygon(0,0,0,0.5f,4);

//Wielobok zwraca współrzędne wierzchołków
mFVertexBuffer = square.getVertexBuffer();

//Wielobok zwraca trójkąty
mIndexBuffer = square.getIndexBuffer();

//Wiersz wymagany do metody glDrawElements
numOfIndices = square.getNumberOfIndices();

//Przypisuje bufore do początku układu
this.mFVertexBuffer.position(0);
this.mIndexBuffer.position(0);

//Ustanawia wskaźnik wierzchołków
gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
```

```
//Rysuje obiekt za pomocą danej liczby indeksów
gl.glDrawElements(GL10.GL_TRIANGLES, numOfIndices,
    GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
```

---

Zobaczmy, w jaki sposób uzyskaliśmy niezbędne wierzchołki i indeksy z kształtu square. Choć nie omówiliśmy idei uzyskiwania wierzchołków i indeksów wobec prostego kształtu, możliwe jest, że klasa RegularPolygon może pochodzić od takiego podstawowego kształtu, definiującego interfejs dla takiego prostego kontraktu. Na listingu 20.25 pokazaliśmy stosowny przykład:

#### **Listing 20.25.** Interfejs Shape

---

```
public interface Shape
{
    FloatBuffer           getVertexBuffer();
    ShortBuffer          getIndexBuffer();
    int                  getNumberOfIndices();
}
```

---

Określenie sposobu definiowania podstawowego interfejsu dla kształtu pozostawiamy jako ćwiczenie dla Czytelnika. Na razie wbudowaliśmy te metody bezpośrednio do klasy Regular Polygon.

### **Implementacja kształtu RegularPolygon**

Jak już stwierdziliśmy, klasa RegularPolygon w bibliotece OpenGL określa wartości parametrów potrzebnych do rysowania figur metodą definiowania wierzchołków. Najpierw trzeba utworzyć mechanizm definiujący ten kształt oraz jego umiejscowienie w geometrii.

W przypadku wielokąta foremnego można tego dokonać na wiele sposobów. W omawianym przykładzie definiujemy wielobok foremny za pomocą określenia liczby boków i odległości wierzchołków od środka figury geometrycznej. Nazwalimy tę odległość promieniem, ponieważ wierzchołki wielokąta foremnego mieszczą się na obwodzie okręgu, którego środek pokrywa się ze środkiem wieloboku. Zatem promień takiego okręgu oraz liczba boków wystarczą nam do opisania takiego wielokąta. Poprzez podanie współrzędnych jego środka możemy także umieścić ten wielokąt w dowolnym miejscu naszej geometrii.

Zadaniem klasy RegularPolygon jest wygenerowanie na podstawie wartości współrzędnych punktu środkowego i liczby boków wielokąta współrzędnych wszystkich jego wierzchołków. Mamy do dyspozycji wiele sposobów wykonania tej czynności. Jakkolwiek aparat matematyczny (na poziomie licealnym) zastosujemy, najważniejsze, aby uzyskiwać współrzędne wierzchołków.

W naszym przykładzie założyliśmy najpierw, że promień wynosi 1 jednostkę. Zdefiniowaliśmy kąty każdej linii łączącej środek z wierzchołkami wieloboku. Umieściliśmy wartości tych kątów w macierzy. Dla każdego kąta obliczyliśmy rzutowanie na oś  $x$  i nazwalimy ten współczynnik „macierz wielokrotności osi  $x$ ” (jest to „macierz wielokrotności”, ponieważ rozpoczęliśmy od jednostki promienia). Gdy będziemy już znać rzeczywisty promień, pomnożymy omawiane wartości przez jego wartość i otrzymamy rzeczywistą współrzędną w osi  $x$ . Następnie te współrzędne zostają zachowane w tak zwanej „macierzy osi  $x$ ”. Taka sama procedura jest wykonywana dla współrzędnych w osi  $y$ .

Skoro już zarysowaliśmy sposób działania naszej implementacji klasy RegularPolygon, zadeemonstrujemy kod źródłowy odpowiedzialny za implementację tych działań. Listing 20.26 prezentuje cały kod tego obiektu (zwróćmy uwagę, że mieści się on na kilku stronach). Aby zachować przejrzystość, zaznaczylismy nazwy funkcji oraz zamieściliśmy komentarze na początku każdej z nich.

Definiujemy najważniejsze funkcje, których lista została umieszczona po listingu 20.26. Ważne jest, aby zrozumieć proces określania i przekazywania wierzchołków. Jeżeli nasz przykład okaże się zbyt trudny, Czytelnik nie powinien mieć problemu z napisaniem własnej wersji kodu definiującego wierzchołki. Zauważmy także, że w kodzie tym zostały również umieszczone funkcje przeprowadzające proces nakładania tekstury. Zostaną one omówione w punkcie „Praca z teksturami”.

#### **Listing 20.26.** Implementacja kształtu RegularPolygon

```
public class RegularPolygon
{
    //Przestrzeń przechowująca współrzędne (x, y, z) środka: cx, cy, cz
    //oraz promień „r”
    private float cx, cy, cz, r;
    private int sides;

    //Macierz współrzędnych: punkty wierzchołków (x, y)
    private float[] xarray = null;
    private float[] yarray = null;

    //Macierz tekstury: punkty (x, y), zwane także punktami (s, t)
    //Figura zostanie odwzorowana na mapie bitowej tekstury
    private float[] sarray = null;
    private float[] tarray = null;

    //*****
    // Constructor
    //*****
    public RegularPolygon(float incx, float incy, float incz, // środek (x, y, z)
                          float inr, // promień
                          int insides) // liczba boków
    {
        cx = incx;
        cy = incy;
        cz = incz;
        r = inr;
        sides = insides;

        //Przydziela pamięć dla macierzy
        xarray = new float[sides];
        yarray = new float[sides];

        //Przydziela pamięć dla macierzy punktów tekstur
        sarray = new float[sides];
        tarray = new float[sides];
    }
}
```

```
//Oblicza wierzchołki
calcArrays();

//Oblicza punkty tekstury
calcTextureArrays();
}

//*********************************************************************
//Pobiera i konwertuje współrzędne wierzchołków
//na podstawie środka i promienia.
//Działania logiczne na kątach są przeprowadzane wewnątrz funkcji
//getMultiplierArray()
//********************************************************************

private void calcArrays()
{
    //Definiuje wierzchołki na podstawie okręgu
    //o promieniu równym „1” i środku w początku układu współrzędnych
    float[] xmarray = this.getXMultiplierArray();
    float[] ymarray = this.getYMultiplierArray();
    //Oblicza xarray: otrzymuje wierzchołek
    //poprzez dodanie składowej „x” do początku układu współrzędnych
    //Mnoży współrzędną przez promień (skalę)
    for(int i=0;i<sides;i++)
    {
        float curm = xmarray[i];
        float xcoord = cx + r * curm;
        xarray[i] = xcoord;
    }
    this.printArray(xarray, "xarray");

    //Oblicza yarray: wykonuje te same czynności dla współrzędnej y
    for(int i=0;i<sides;i++)
    {
        float curm = ymarray[i];
        float ycoord = cy + r * curm;
        yarray[i] = ycoord;
    }
    this.printArray(yarray, "yarray");
}

//Oblicza macierze tekstury
//Więcej informacji można znaleźć w punkcie poświęconym teksturom
//Bardzo podobne rozwiążanie.
//Tutaj wielokąt musi zostać odwzorowany na kwadratowej przestrzeni
//********************************************************************

private void calcTextureArrays()
{
    float[] xmarray = this.getXMultiplierArray();
    float[] ymarray = this.getYMultiplierArray();

    //Oblicza xarray
    for(int i=0;i<sides;i++)
```

```

{
    float curm = xmarray[i];
    float xcoord = 0.5f + 0.5f * curm;
    sarray[i] = xcoord;
}
this.printArray(sarray, "sarray");

//Oblicza yarray
for(int i=0;i<sides;i++)
{
    float curm = ymarray[i];
    float ycoord = 0.5f + 0.5f * curm;
    tarray[i] = ycoord;
}
this.printArray(tarray, "tarray");
}

//*****
//Konwertuje macierz java wierzchołków
//do zmiennoprzecinkowego bufora nio
//*****
public FloatBuffer getVertexBuffer()
{
    int vertices = sides + 1;
    int coordinates = 3;
    int floatszie = 4;
    int spacePerVertex = coordinates * floatszie;

    ByteBuffer vbb = ByteBuffer.allocateDirect(spacePerVertex * vertices);
    vbb.order(ByteOrder.nativeOrder());
    FloatBuffer mFVertexBuffer = vbb.asFloatBuffer();

    //Umieszcza pierwszą współrzędną (x, y, z: 0, 0, 0)
    mFVertexBuffer.put(cx); //x
    mFVertexBuffer.put(cy); //y
    mFVertexBuffer.put(0.0f); //z

    int totalPuts = 3;
    for (int i=0;i<sides;i++)
    {
        mFVertexBuffer.put(xarray[i]); //x
        mFVertexBuffer.put(yarray[i]); //y
        mFVertexBuffer.put(0.0f); //z
        totalPuts += 3;
    }
    Log.d("Łącznie wstawiono:", Integer.toString(totalPuts));
    return mFVertexBuffer;
}

//*****
//Konwertuje bufor tekstury do bufora nio
//*****
public FloatBuffer getTextureBuffer()
{

```

```
int vertices = sides + 1;
int coordinates = 2;
int floatszie = 4;
int spacePerVertex = coordinates * floatszie;

ByteBuffer vbb = ByteBuffer.allocateDirect(spacePerVertex * vertices);
vbb.order(ByteOrder.nativeOrder());
FloatBuffer mFTextureBuffer = vbb.asFloatBuffer();

//Umieszcza pierwszą współrzędną (x, y (s, t):0, 0)
mFTextureBuffer.put(0.5f); //x lub s
mFTextureBuffer.put(0.5f); //y lub t

int totalPuts = 2;
for (int i=0;i<sides;i++)
{
    mFTextureBuffer.put(sarray[i]); //x
    mFTextureBuffer.put(tarray[i]); //y
    totalPuts += 2;
}
Log.d("Łączna liczba wstawionych tekstur:",Integer.toString(totalPuts));
return mFTextureBuffer;
}

//*****
//Oblicza indeksy tworzące wiele trójkątów.
//Rozpoczyna od środkowego wierzchołka (punkt 0)
//Następnie numeruje je zgodnie z kierunkiem ruchu wskazówek zegara, na przykład
//0, 1, 2; 0, 2, 3; 0, 3, 4... itd.
//*****
public ShortBuffer getIndexBuffer()
{
    short[] iarray = new short[sides * 3];
    ByteBuffer ibb = ByteBuffer.allocateDirect(sides * 3 * 2);
    ibb.order(ByteOrder.nativeOrder());
    ShortBuffer mIndexBuffer = ibb.asShortBuffer();
    for (int i=0;i<sides;i++)
    {
        short index1 = 0;
        short index2 = (short)(i+1);
        short index3 = (short)(i+2);
        if (index3 == sides+1)
        {
            index3 = 1;
        }
        mIndexBuffer.put(index1);
        mIndexBuffer.put(index2);
        mIndexBuffer.put(index3);

        iarray[i*3 + 0]=index1;
        iarray[i*3 + 1]=index2;
        iarray[i*3 + 2]=index3;
    }
    this.printShortArray(iarray, "index array");
    return mIndexBuffer;
}
```

```
}

//*****
//Stąd jest pobierana macierz kątów
//dla każdego wierzchołka i zostaje obliczony ich współczynnik rzutowania
//na oś x
//*****

private float[] getXMultiplierArray()
{
    float[] angleArray = getAngleArrays();
    float[] xmultiplierArray = new float[sides];
    for(int i=0;i<angleArray.length;i++)
    {
        float curAngle = angleArray[i];
        float sinvalue = (float)Math.cos(Math.toRadians(curAngle));
        float absSinValue = Math.abs(sinvalue);
        if (isXPositiveQuadrant(curAngle))
        {
            sinvalue = absSinValue;
        }
        else
        {
            sinvalue = -absSinValue;
        }
        xmultiplierArray[i] = this.getApproxValue(sinvalue);
    }
    this.printArray(xmultiplierArray, "xmultiplierArray");
    return xmultiplierArray;
}

//*****
//Stąd jest pobierana macierz kątów
//dla każdego wierzchołka i zostaje obliczony ich współczynnik rzutowania
//na oś y
//*****

private float[] getYMultiplierArray() {
    float[] angleArray = getAngleArrays();
    float[] ymultiplierArray = new float[sides];
    for(int i=0;i<angleArray.length;i++) {
        float curAngle = angleArray[i];
        float sinvalue = (float)Math.sin(Math.toRadians(curAngle));
        float absSinValue = Math.abs(sinvalue);
        if (isYPositiveQuadrant(curAngle)) {
            sinvalue = absSinValue;
        }
        else {
            sinvalue = -absSinValue;
        }
        ymultiplierArray[i] = this.getApproxValue(sinvalue);
    }
    this.printArray(ymultiplierArray, "ymultiplierArray");
    return ymultiplierArray;
}

//*****
//Ta funkcja może być niepotrzebna
```

```
//Należy ją samodzielnie sprawdzić i usunąć, jeśli nie okaże się przydatna
//*****
private boolean isXPositiveQuadrant(float angle) {
    if ((0 <= angle) && (angle <= 90)) { return true; }
    if ((angle < 0) && (angle >= -90)) { return true; }
    return false;
}
//*****
//Ta funkcja może być niepotrzebna
//Należy ją samodzielnie sprawdzić i usunąć, jeśli nie okaże się przydatna
//*****
private boolean isYPositiveQuadrant(float angle) {
    if ((0 <= angle) && (angle <= 90)) { return true; }
    if ((angle < 180) && (angle >= 90)) { return true; }
    return false;
}
//*****
//Tutaj są obliczane kąty
//dla każdej linii wychodzącej ze środka do wierzchołka
//*****
private float[] getAngleArrays() {
    float[] angleArray = new float[sides];
    float commonAngle = 360.0f/sides;
    float halfAngle = commonAngle/2.0f;
    float firstAngle = 360.0f - (90+halfAngle);
    angleArray[0] = firstAngle;

    float curAngle = firstAngle;
    for(int i=1;i<sides;i++)
    {
        float newAngle = curAngle - commonAngle;
        angleArray[i] = newAngle;
        curAngle = newAngle;
    }
    printArray(angleArray, "angleArray");
    return angleArray;
}
//*****
//Opcjonalne zaokrąglanie
//*****
private float getApproxValue(float f) {
    return (Math.abs(f) < 0.001) ? 0 : f;
}
//*****
//Zwraca liczbę potrzebnych indeksów
//na podstawie liczby boków
//Jest to liczba trójkątów potrzebnych do
//pomnożenia wielokąta przez wartość 3
//Tak się składa, że liczba trójkątów jest
//równa liczbie boków
//*****
public int getNumberOfIndices() {
```

```

        return sides * 3;
    }
    public static void test() {
        RegularPolygon triangle = new RegularPolygon(0,0,0,1,3);
    }
    private void printArray(float array[], String tag) {
        StringBuilder sb = new StringBuilder(tag);
        for(int i=0;i<array.length;i++) {
            sb.append(";").append(array[i]);
        }
        Log.d("hh",sb.toString());
    }
    private void printShortArray(short array[], String tag) {
        StringBuilder sb = new StringBuilder(tag);
        for(int i=0;i<array.length;i++) {
            sb.append(";").append(array[i]);
        }
        Log.d(tag,sb.toString());
    }
}

```

---

Poniżej przedstawiamy najważniejsze elementy kodu:

- **Constructor** — konstruktor klasy `RegularPolygon` jako dane wejściowe pobiera współrzędne środka, promień i liczbę boków.
- **getAngleArrays** — jest to kluczowa metoda, obliczająca kąty pomiędzy bokami wielokąta foremnego przy założeniu, że jeden z jego boków jest położony równolegle do osi *x*.
- **getXMultiplierArray, getYMultiplierArray** — metody te pobierają kąty od metody `getAngleArrays` i rzutują je na osie *x* oraz *y* w celu uzyskania odpowiednich współrzędnych — przy założeniu, że długość boku wynosi 1 jednostkę długości.
- **calcArrays** — metoda ta wykorzystuje metody `getXMultiplierArray` i `getYMultiplierArray` do pobrania wierzchołków i dopasowania ich do określonego promienia i środka wieloboku. Po zakończeniu pracy tej metody obiekt `RegularPolygon` będzie posiadał odpowiednie współrzędne, lecz w formacie zmiennoprzecinkowych macierzy Java.
- **getVertexBuffer** — ta metoda pobiera macierze współrzędnych Java i przetwarza je na bufory `nio`, wymagane przez metody rysowania w bibliotece OpenGL.
- **getIndexBuffer** — metoda ta pobiera pogrupowane wierzchołki i umieszcza je w takiej kolejności, że każdy trójkąt będzie częścią tworzonego wieloboku.

Pozostałe metody, obsługujące tekstury, są wykorzystywane według bardzo podobnego algorytmu i staną się jeszcze bardziej zrozumiałe po przeczytaniu punktu poświęconego teksturom.

## Renderowanie kwadratu za pomocą klasy `RegularPolygon`

Skoro już poznaliśmy podstawowe bloki budulcowe, zobaczymy, w jaki sposób możemy narysować kwadrat za pomocą czterobocznego obiektu klasy `RegularPolygon`. Listing 20.27 przedstawia kod klasy `SquareRenderer`.

**Listing 20.27.** Silnik SquareRenderер

```
public class SquareRenderер extends AbstractRenderер
{
    //Nieskompresowany bufor natywny, przechowujący współrzędne punktów
    private FloatBuffer mFVertexBuffer;

    //Nieskompresowany bufor natywny, przechowujący indeksy
    //umożliwiające wielokrotne wykorzystywanie punktów
    private ShortBuffer mIndexBuffer;

    private int numOfIndices = 0;
    private int sides = 4;

    public SquareRenderер(Context context)
    {
        prepareBuffers(sides);
    }

    private void prepareBuffers(int sides)
    {
        RegularPolygon t = new RegularPolygon(0,0,0,0.5f,sides);
        //RegularPolygon t = new RegularPolygon(1,1,0,1,sides);
        this.mFVertexBuffer = t.getVertexBuffer();
        this.mIndexBuffer = t.getIndexBuffer();
        this.numOfIndices = t.getNumberOfIndices();
        this.mFVertexBuffer.position(0);
        this.mIndexBuffer.position(0);
    }

    //przesłonięta metoda
    protected void draw(GL10 gl)
    {
        prepareBuffers(sides);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
            GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

---

Ten kod powinien być całkowicie zrozumiały. Pochodzi z klasy `AbstractRenderер` (listing 20.9), przesłonił metodę `draw` i wykorzystał klasę `RegularPolygon` do narysowania kwadratu.

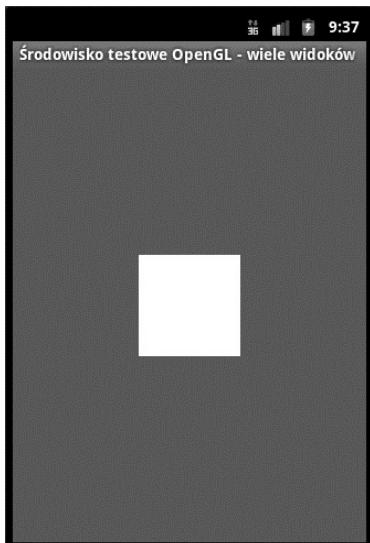
Po utworzeniu tego silnika renderującego możemy dodać instrukcję warunkową `if` (listing 20.28) do klasy `MultiViewTestHarness`, zaprezentowanej na listingu 20.12.

**Listing 20.28.** Zastosowanie klasy SimpleRectagleRenderер

```
if (mid == R.id.mid_square_polygon)
{
    mTestHarness.setRenderер(new SquareRenderер(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(mTestHarness);
    return;
}
```

---

Po dodaniu powyższego fragmentu kodu możemy ponownie uruchomić program i wybrać element menu *Kwadrat*, aby ujrzeć figurę przedstawioną na rysunku 20.8.



Rysunek 20.8. Kwadrat narysowany za pomocą klasy RegularPolygon

## Animowanie obiektów RegularPolygon

Po zaprezentowaniu ogólnej zasady rysowania kształtów za pomocą klasy RegularPolygon przejdźmy do bardziej skomplikowanych kwestii. Sprawdźmy, czy możemy utworzyć animację, która rozpoczyna się od narysowania trójkąta, a kończy się po przekształceniu go na okrąg. W tym celu najpierw narysujemy wielokąt, któremu co cztery sekundy będą dodawane kolejne boki. Odpowiedni kod znajdziemy na listingu 20.29).

**Listing 20.29.** Klasa PolygonRenderer

```
public class PolygonRenderer extends AbstractRenderer
{
    //Liczba wykorzystanych punktów lub wierzchołków
    private final static int VERTS = 4;

    //Nieskompresowany bufor natywny, przechowujący współrzędne punktów
    private FloatBuffer mFVertexBuffer;

    //Nieskompresowany bufor natywny, przechowujący indeksy
    //umożliwiające wielokrotne wykorzystywanie punktów

    private ShortBuffer mIndexBuffer;

    private int numOfIndices = 0;

    private long prevtime = SystemClock.uptimeMillis();

    private int sides = 3;
```

```
public PolygonRenderer(Context context)
{
    prepareBuffers(sides);
}

private void prepareBuffers(int sides)
{
    RegularPolygon t = new RegularPolygon(0,0,0,1,sides);
    this.mFVertexBuffer = t.getVertexBuffer();
    this.mIndexBuffer = t.getIndexBuffer();
    this.numOfIndices = t.getNumberOfIndices();
    this.mFVertexBuffer.position(0);
    this.mIndexBuffer.position(0);
}

//przesłonięta metoda
protected void draw(GL10 gl)
{
    long curtime = SystemClock.uptimeMillis();
    if ((curtime - prevtime) > 2000)
    {
        prevtime = curtime;
        sides += 1;
        if (sides > 20)
        {
            sides = 3;
        }
        this.prepareBuffers(sides);
    }
    gl.glColor4f(1.0f, 0, 0, 0.5f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
    gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
                      GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
}
}
```

---

Powyższy kod jedynie zmienia wartość zmiennej `sides` co cztery sekundy. Animacja powstaje dzięki temu, że klasa `Render` jest rejestrowana z widokiem powierzchni.

Skoro już posiadamy tę klasę renderującą, musimy dodać kod widoczny na listingu 20.30 do klasy `MultiViewTestHarness`.

---

#### **Listing 20.30.** Element menu umożliwiający testowanie wielokąta

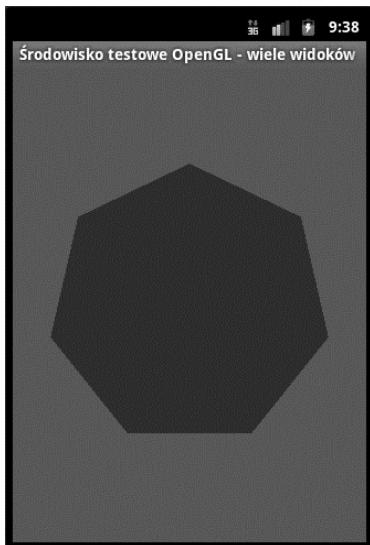
---

```
if (mid == R.id.mid_polygon)
{
    mTestHarness.setRenderer(new PolygonRenderer(this));
    setContentView(mTestHarness);
    return;
}
```

---

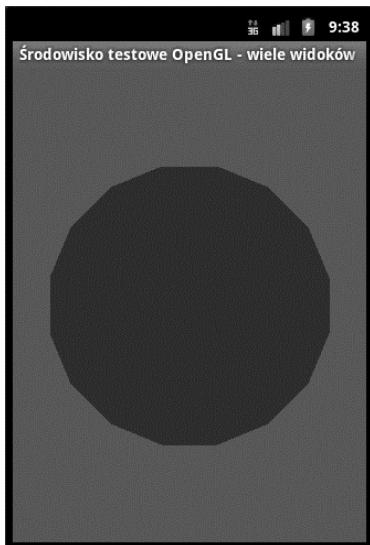
Jeżeli ponownie uruchomimy nasz program i wybierzemy opcję *Wielokąt*, ujrzymy zbiór przekształcających się figur geometrycznych, których liczba wierzchołków będzie ustawicznie rosła.

Warto się jednak przyjrzeć, w jaki sposób wielokąty ulegają przekształceniom. Na rysunku 20.9 został pokazany początek cyklu przekształcania wielokąta.



**Rysunek 20.9.** Początek cyklu rysowania wielokąta

Na rysunku 20.10 widać koniec cyklu transformacji wielokąta.



**Rysunek 20.10.** Koło narysowane za pomocą klasy RegularPolygon

Możemy rozwinąć koncepcję kształtów na bardziej złożone figury geometryczne, a nawet graf sceny, składający się z dużej liczby innych obiektów, które mogą zostać zdefiniowane w języku XML, a następnie renderowane w środowisku OpenGL poprzez te właśnie kształty.

Zajmijmy się teraz teksturami, aby dowiedzieć się, w jaki sposób powiązać obrazy z takimi powierzchniami, jak kwadraty i wielokąty.

## Praca z teksturami

Jednym z podstawowych pojęć stosowanych w terminologii OpenGL są tekstury. W środowisku OpenGL wiążą się one z wieloma niuansami. Omówimy tu jedynie podstawowe koncepcje, umożliwiające rozpoczęcie pracy z teksturami w środowisku OpenGL. W celu pogłębienia wiedzy na temat tekstur można skorzystać z listy zasobów umieszczonej w końcowej części rozdziału.

### Tekstury

Teksturą OpenGL nazywamy mapę bitową umieszczaną na danej powierzchni w środowisku OpenGL (w tym rozdziale zajmujemy się jedynie powierzchniami). Możemy na przykład użyć obrazu znaczka pocztowego i umieścić go na powierzchni kwadratu, dzięki czemu uzyskamy obraz znaczka pocztowego. Możemy także wykorzystać mapę bitową przedstawiającą wizerunek cegły, umieścić ją na powierzchni prostokąta i poprzez powielanie takich obrazów cegły utworzyć obraz muru.

Proces przyłączania mapy bitowej tekstury do powierzchni w środowisku OpenGL przypomina proces naklejania fragmentu tapety (kwadratowej) na boku obiektu posiadającego regularny lub nieregularny kształt. Kształt powierzchni nie ma znaczenia, dopóki wymiary papieru umożliwiają całkowite pokrycie powierzchni.

Aby jednak umieścić papier w odpowiedniej orientacji, pozwalającej na właściwe uformowanie obrazu, musimy pobrać każdy wierzchołek kształtu i dokładnie zaznaczyć go na tapecie, co spowoduje idealne dopasowanie tapety do kształtu obiektu. Jeżeli mamy do czynienia z niestandardowym kształtem posiadającym wiele wierzchołków, każdy z nich musi zostać zaznaczony na tapecie.

Można to sobie wyobrazić również w inny sposób: kładziemy obiekt na ziemi, przednią ścianą skierowaną do góry, rozkładamy na tej ścianie tapetę i obracamy ją, dopóki nie zostanie zorientowana we właściwym kierunku. Teraz dziurkami zaznaczamy na tapecie każdy wierzchołek kształtu. Ściągamy tapetę, sprawdzamy położenie wierzchołków i zapisujemy ich współrzędne, przy założeniu, że tapeta jest wykalibrowana. Są to tak zwane **współrzędne tekstury**.

### Znormalizowane współrzędne tekstury

Jednym z nieroziwiązanych i niezdefiniowanych szczegółów są rozmiary obiektu i „tapety”. W środowisku OpenGL rozwiązuje się ten problem za pomocą normalizacji. Tekstura jest tu zawsze kwadratem o wymiarach  $1 \times 1$ , którego początek posiada współrzędne  $(0, 0)$ , prawy górny róg —  $(1, 1)$ . Następnie należy zmniejszyć powierzchnię obiektu, tak aby mieściła się w tych wymiarach  $1 \times 1$ . Zatem zadaniem programisty jest określenie wierzchołków powierzchni obiektu o rozmiarach  $1 \times 1$ .

W projekcie demonstrującym klasę `RegularPolygon` z listingu 20.26 w podobny sposób rysowaliśmy wielokąt za pomocą okręgu o promieniu 1. Następnie określaliśmy położenie każdego wierzchołka. Gdybyśmy założyli, że okrąg zajmuje powierzchnię  $1 \times 1$  kwadratu, to ten kwadrat mógłby być naszą tapetą. Zatem określenie współrzędnych tekstury jest analogczną czynnością do procesu wyznaczania współrzędnych wierzchołków wielokąta. Dlatego na listingu 20.26 znalazła się następująca funkcja, obliczająca współrzędne tekstury:

```
calcTextureArray()
getTextureBuffer()
```

Jeżeli przyjrzymy się uważniej, zauważmy, że wszystkie pozostałe funkcje są współdzielone pomiędzy metodami calcTextureArray i calcArray. Taka wspólność pomiędzy współrzędnymi wierzchołków i współrzędnymi tekstur stanowi ważny wniosek podczas nauki obsługi środowiska OpenGL.

## Analiza procesu standardowej obsługi tekstuury

Po zrozumieniu związku pomiędzy współrzędnymi tekstury i współrzędnymi wierzchołków oraz po określeniu współrzędnych mapy tekstuury reszta czynności jest już wystarczająco prosta (w środowisku OpenGL nic nie może być wyraźnie uznane za „całkiem proste”!). Kolejnymi etapami są wczytanie mapy bitowej tekstuury do pamięci i przydzielenie identyfikatora, umożliwiającego wielokrotne jej użytkowanie. Następnie wykorzystujemy mechanizm ustanawiania bieżącej tekstuury za pomocą jej identyfikatora, co pozwala na jednocześnie wczytanie wielu tekstur. W potoku rysowania określamy współrzędne tekstuury oraz współrzędne rysowania. Na koniec pozostaje sam proces rysowania.

Ponieważ proces wczytywania tekstuur jest dosyć powszechnie stosowany, wydzieliśmy go poprzez utworzenie abstrakcyjnej klasy AbstractSingleTextureRender, wywodzącej się z klasy AbstractRender.

Na listingu 20.31 został umieszczony kod źródłowy pozwalający na pełną konfigurację pojedynczej tekstuury.

**Listing 20.31.** Wydzielanie obsługi procesu nakładania pojedynczej tekstuury

```
public abstract class AbstractSingleTexturedRender
extends AbstractRender {
    int mTextureID;
    int mImageResourceId;
    Context mContext;
    public AbstractSingleTexturedRender(Context ctx,
                                         int imageResourceId) {
        mImageResourceId = imageResourceId;
        mContext = ctx;
    }
    public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig) {
        super.onSurfaceCreated(gl, eglConfig);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        prepareTexture(gl);
    }
    private void prepareTexture(GL10 gl) {
        int[] textures = new int[1];
        gl.glGenTextures(1, textures, 0);
        mTextureID = textures[0];
        gl glBindTexture(GL10.GL_TEXTURE_2D, mTextureID);
        gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER,
                           GL10.GL_NEAREST);
```

```
gl.glTexParameterf(GL10.GL_TEXTURE_2D,
    GL10.GL_TEXTURE_MAG_FILTER,
    GL10.GL_LINEAR);

gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,
    GL10.GL_CLAMP_TO_EDGE);

gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,
    GL10.GL_CLAMP_TO_EDGE);

gl.glTexEnvf(GL10.GL_TEXTURE_ENV, GL10.GL_TEXTURE_ENV_MODE,
    GL10.GL_REPLACE);

InputStream is = mContext.getResources()
    .openRawResource(this.mImageResourceId);
Bitmap bitmap;
try {
    bitmap = BitmapFactory.decodeStream(is);
} finally {
    try {
        is.close();
    } catch(IOException e) {
        // Ignorujemy.
    }
}

GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
bitmap.recycle();
}

public void onDrawFrame(GL10 gl)
{
    gl.glDisable(GL10.GL_DITHER);
    gl.glTexEnvx(GL10.GL_TEXTURE_ENV, GL10.GL_TEXTURE_ENV_MODE,
        GL10.GL_MODULATE);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();
    GLU.gluLookAt(gl, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    gl glEnableClientState(GL10.GL_VERTEX_ARRAY);

    gl glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

    gl.glActiveTexture(GL10.GL_TEXTURE0);
    gl.glBindTexture(GL10.GL_TEXTURE_2D, mTextureID);
    gl.glTexParameterx(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_S,
        GL10.GL_REPEAT);
    gl.glTexParameterx(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_WRAP_T,
        GL10.GL_REPEAT);

    draw(gl);
}
}
```

---

W powyższym kodzie pojedyncza tekstura (mapa bitowa) została wczytana i przygotowana w metodzie `onSurfaceCreated`. Tak samo jak w przypadku klasy `AbstractRenderer`, metoda `onDrawFrame` konfiguruje wymiary naszej przestrzeni rysowania, dzięki czemu współrzędne nabierają sensu. W zależności od sytuacji możemy zmieniać ten kod, aby uzyskać optymalną objętość widzenia.

Spójrzmy także, w jaki sposób konstruktor przyjmuje mapę bitową tekstury i przygotowuje ją do późniejszego użytku. W zależności od liczby dostępnych tekstur możemy odpowiednio projektować abstrakcyjne klasy.

Jak zostało pokazane na listingu 20.31, do przetwarzania tekstur wymagane są specyficzne interfejsy API:

- `glGenTextures` — ta metoda tworzy niepowtarzalny identyfikator tekstur, za pomocą którego można uzyskiwać do nich odniesienia. Po wczytaniu mapy bitowej tekstury za pomocą narzędzia `GLUtils.texImage2D` powiążemy tę teksturę z określonym identyfikatorem. Dopóki tekstura nie zostanie powiązana z identyfikatorem wygenerowanym przez metodę `glGenTextures`, jest on jedynie ciągiem znaków. W literaturze dotyczącej standardu OpenGL identyfikatory te są określane jako **nazwy tekstur**.
- `glBindTexture` — ta metoda jest stosowana do wiązania bieżącej tekstury z identyfikatorem uzyskanym z metody `glGenTextures`.
- `glTexParameter` — podczas wstawiania tekstury można wykorzystać wiele opcjonalnych parametrów. Omawiany interfejs pozwala na ich zdefiniowanie. Za przykłady mogą posłużyć parametry `GL_REPEAT`, `GL_CLAMP` i tak dalej. Parametr `GL_REPEAT` służy do wielokrotnego powielania mapy bitowej, w przypadku gdy obiekt jest dużo większy. Pełną listę dostępnych parametrów można znaleźć pod adresem:  
[www.khronos.org/opengles/documentation/opengles1\\_0/html/glTexParameter.html](http://www.khronos.org/opengles/documentation/opengles1_0/html/glTexParameter.html).
- `glTexEnv` — niektóre opcje związane z teksturami są dostępne w metodzie `glTexEnv`. Wśród wartości można znaleźć `GL_DECAL`, `GL_MODULATE`, `GL_BLEND`, `GL_REPLACE` i tak dalej. Na przykład w przypadku parametru `GL_DECAL` tekstura pokrywa obiekt. Jak sama nazwa wskazuje, metoda `GL_MODULATE` moduluje kolory, zamiast je zamieniać. Pełną listę opcji dostępnych w tym interfejsie API można znaleźć pod następującym adresem:  
[www.khronos.org/opengles/documentation/opengles1\\_0/html/glTexEnv.html](http://www.khronos.org/opengles/documentation/opengles1_0/html/glTexEnv.html).
- `GLUtils.texImage2D` — jest to interfejs API Androida, pozwalający na wczytanie mapy bitowej pełniącej rolę tekstury. Interfejs ten wywołuje wewnętrznie metodę `glTexImage2D`.
- `glActiveTexture` — ustanawia identyfikator danej tekstury jako aktywną strukturę.
- `glTexCoordPointer` — ta metoda środowiska OpenGL jest używana do określania współrzędnych tekstury. Każda współrzędna musi pasować do współrzędnej zdefiniowanej w metodzie `glVertexPointer`.

Większość informacji na temat wymienionych interfejsów API można znaleźć w dokumentacji środowiska OpenGL ES dostępnej pod adresem:

[www.khronos.org/opengles/documentation/opengles1\\_0/html/index.html](http://www.khronos.org/opengles/documentation/opengles1_0/html/index.html)

## Rysowanie za pomocą tekstur

Po wczytaniu mapy bitowej i skonfigurowaniu jej jako tekstury powinniśmy mieć możliwość zastosowania klasy `RegularPolygon` i wykorzystać wspólnie współrzędne tekstury ze wspólnymi wierzchołkami do narysowania wieloboku foremnego pokrytego teksturową. Listing 20.32 prezentuje rzeczywistą klasę rysującą teksturowany kwadrat.

**Listing 20.32.** Klasa `TexturedSquareRenderer`

---

```
public class TexturedSquareRenderer extends AbstractSingleTexturedRender
{
    //Liczba wykorzystywanych punktów lub wierzchołków
    private final static int VERTS = 4;

    //Nieskompresowany bufor natywny, przechowujący współrzędne punktów
    private FloatBuffer mFVertexBuffer;
    //Nieskompresowany bufor natywny, przechowujący współrzędne punktów
    private FloatBuffer mFTextureBuffer;
    // Nieskompresowany bufor natywny, przechowujący indeksy
    //pozwalające na wielokrotne wykorzystywanie punktów
    private ShortBuffer mIndexBuffer;

    private int numOfIndices = 0;
    private int sides = 4;

    public TexturedSquareRenderer(Context context)
    {
        super(context, com.androidbook.OpenGL.R.drawable.robot);
        prepareBuffers(sides);
    }

    private void prepareBuffers(int sides)
    {
        RegularPolygon t = new RegularPolygon(0, 0, 0, 0.5f, sides);
        this.mFVertexBuffer = t.getVertexBuffer();
        this.mFTextureBuffer = t.getTextureBuffer();
        this.mIndexBuffer = t.getIndexBuffer();
        this.numOfIndices = t.getNumberOfIndices();
        this.mFVertexBuffer.position(0);
        this.mIndexBuffer.position(0);
        this.mFTextureBuffer.position(0);
    }

    //przesłonięta metoda
    protected void draw(GL10 gl)
    {
        prepareBuffers(sides);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mFTextureBuffer);
        gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
            GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
    }
}
```

---

Jak widać, większość pracy wykonuje abstrakcyjna klasa silnika renderowania tekstuury i obiekt `RegularPolygon`, obliczający wierzchołki odwzorowania tekstuury (listing 20.26).

Po utworzeniu tej klasy renderującej, aby przetestować tekstury kwadrat, musimy dodać kod z listingu 20.33 do aktywności `MultiViewTestHarness` z listingu 20.12.

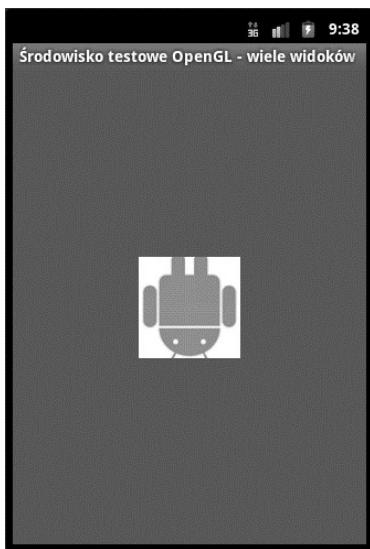
#### **Listing 20.33.** Odpowiedź na element menu Teksturowany kwadrat

---

```
if (mid == R.id.mid_textured_square)
{
    mTestHarness.setRenderer(new TexturedSquareRenderer(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(mTestHarness);
    return;
}
```

---

Ponowne uruchomienie programu i wybranie elementu menu *Teksturowany kwadrat* powoduje wyświetlenie figury geometrycznej widocznej na rysunku 20.11.



**Rysunek 20.11.** Teksturowany kwadrat

## Rysowanie wielu figur geometrycznych

Wszystkie przykładowe projekty omówione w tym rozdziale były do siebie podobne: rysowaliśmy prostą figurę geometryczną za pomocą standardowego wzorca. Wzorzec ten wygląda następująco: konfigurowanie wierzchołków, wczytanie tekstuury, skonfigurowanie jej współrzędnych, rysowanie pojedynczego obiektu. A jak postąpić w przypadku, gdy chcemy narysować dwie figury geometryczne? Co zrobić, jeśli chcemy narysować trójkąt tradycyjnym sposobem definiowania wierzchołków, a następnie utworzyć wielokąt za pomocą kształtu, na przykład `RegularPolygon`? W jaki sposób powiązemy ze sobą wierzchołki zdefiniowane dwoma różnymi sposobami? Czy musimy jednorazowo określać wierzchołki dla obydwu obiektów, a następnie wywoływać metodę rysowania?

Okazuje się, że pomiędzy dwoma wywołaniami metody `draw()` w interfejsie silnika renderowania środowisko OpenGL umożliwia wstawienie wielu metod `glDraw`. Pomiędzy wywołaniami tych metod możemy określić nowe tekstury i wierzchołki. Wyniki otrzymywane za pomocą tych wszystkich metod zostaną odzwierciedlone na ekranie po zakończeniu działania metody `draw()`.

Możemy skorzystać również z innej sztuczki środowiska OpenGL pozwalającej na rysowanie wielu obiektów. Zastanówmy się nad dotychczas tworzonymi wielokątami. Mogą być wyświetlane w dowolnym punkcie początkowym po określeniu współrzędnych tego punktu jako parametru rysowanych figur. W środowisku OpenGL jest to dokonywane natywnie, co pozwala nam zawsze definiować obiekt `RegularPolygon` w punkcie `(0, 0, 0)` i wykorzystać mechanizm „translacji” w środowisku OpenGL, dzięki któremu punkt początkowy można przesunąć na żądaną pozycję. Taką samą czynność możemy przeprowadzić dla kolejnego wielokąta i przesunąć go na inną pozycję, w wyniku czego zostaną narysowane dwa wielokąty w dwóch różnych miejscach ekranu.

Na listingu 20.34 pokazano kod ilustrujący te koncepcje poprzez wielokrotne narysowanie teksturowanego wielokąta.

**Listing 20.34.** Silnik renderujący teksturowany wielobok

---

```
public class TexturedPolygonRenderer extends AbstractSingleTexturedRenderer
{
    //Liczba wykorzystywanych punktów lub wierzchołków
    private final static int VERTS = 4;

    // Nieskompresowany bufor natywny, przechowujący współrzędne punktów
    private FloatBuffer mFVertexBuffer;

    // Nieskompresowany bufor natywny, przechowujący współrzędne punktów
    private FloatBuffer mFTextureBuffer;

    // Nieskompresowany bufor natywny, przechowujący indeksy
    //pozwalające na wielokrotne wykorzystywanie punktów
    private ShortBuffer mIndexBuffer;

    private int numOfIndices = 0;

    private long prevtime = SystemClock.uptimeMillis();
    private int sides = 3;

    public TexturedPolygonRenderer(Context context)
    {
        super(context, com.ai.android.OpenGL.R.drawable.robot);
        prepareBuffers(sides);
    }

    private void prepareBuffers(int sides)
    {
        RegularPolygon t = new RegularPolygon(0, 0, 0, 0.5f, sides);
        this.mFVertexBuffer = t.getVertexBuffer();
        this.mFTextureBuffer = t.getTextureBuffer();
        this.mIndexBuffer = t.getIndexBuffer();
        this.numOfIndices = t.getNumberOfIndices();
        this.mFVertexBuffer.position(0);
```

```

this.mIndexBuffer.position(0);
this.mFTextureBuffer.position(0);
}

//przesłonięta metoda
protected void draw(GL10 gl)
{
    long curtime = SystemClock.uptimeMillis();
    if ((curtime - prevtime) > 2000)
    {
        prevtime = curtime;
        sides += 1;
        if (sides > 20)
        {
            sides = 3;
        }
        this.prepareBuffers(sides);
    }
    gl.glEnable(GL10.GL_TEXTURE_2D);

//Rysuje jednokrotnie po lewej stronie
gl.glVertexPointer(3, GL10.GL_FLOAT, 0, mFVertexBuffer);
gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mFTextureBuffer);

gl.glPushMatrix();
gl.glScalef(0.5f, 0.5f, 1.0f);
gl.glTranslatef(0.5f, 0, 0);
gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
GL10.GL_UNSIGNED_SHORT, mIndexBuffer);

//Rysuje ponownie po stronie prawej
gl.glPopMatrix();
gl.glPushMatrix();
gl.glScalef(0.5f, 0.5f, 1.0f);
gl.glTranslatef(-0.5f, 0, 0);
gl.glDrawElements(GL10.GL_TRIANGLES, this.numOfIndices,
GL10.GL_UNSIGNED_SHORT, mIndexBuffer);
gl.glPopMatrix();
}
}

```

W tym przykładowym kodzie ukazaliśmy następujące koncepcje:

- rysowanie za pomocą kształtów,
- rysowanie wielu kształtów za pomocą macierzy transformacji,
- wprowadzanie tekstur,
- animacje.

Główny kod na listingu 20.34, umożliwiający wielokrotne rysowanie, znajduje się w metodzie `draw()`. Pogrubioną czcionką zaznaczyliśmy odpowiednie wiersze. Zauważmy, że wewnątrz metody `draw()` wywołaliśmy dwukrotnie metodę `glDrawElements`. Za każdym razem konfigurujemy proste obiekty rysowane, niezależne od siebie.

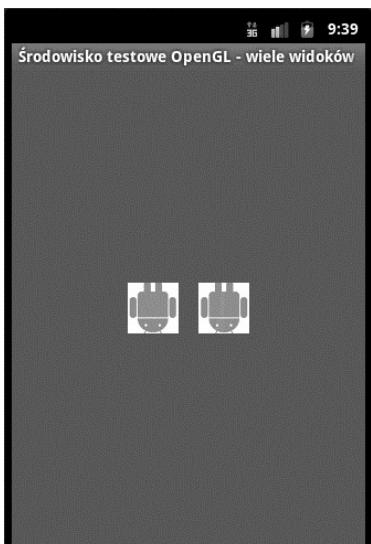
Wyjaśnijmy jeszcze zastosowanie macierzy transformacji. Za wywołaniem metody `glDrawElements()` korzysta ona ze specyficznej macierzy transformacji. Gdybyśmy chcieli wykorzystać tę macierz do zmiany położenia figury geometrycznej (lub innego jej aspektu), musielibyśmy przywrócić jej pierwotne ustawienia, aby następny obiekt mógł zostać poprawnie narysowany. Dokonujemy tego poprzez operacje `PUSH` i `POP` dostępne na matrycach środowiska OpenGL.

Po utworzeniu tej klasy renderującej musimy dodać kod z listingu 20.35 do aktywności `MultiViewTestHarness` z listingu 20.12, aby móc przetestować rysowanie wielu figur geometrycznych.

### **Listing 20.33.** Odpowiedź na element menu *Wiele figur*

```
if (mid == R.id.mid_multiple_figures)
{
    mTestHarness.setRenderer(new TexturedPolygonRenderer(this));
    mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
    setContentView(mTestHarness);
    return;
}
```

Ponowne uruchomienie programu i wybranie elementu menu *Wiele figur* spowoduje narysowanie na początku animacji dwóch zestawów zmieniających się wielokątów (widocznych na rysunku 20.12). Zwróćmy uwagę, że został zdefiniowany ciągły tryb renderowania.

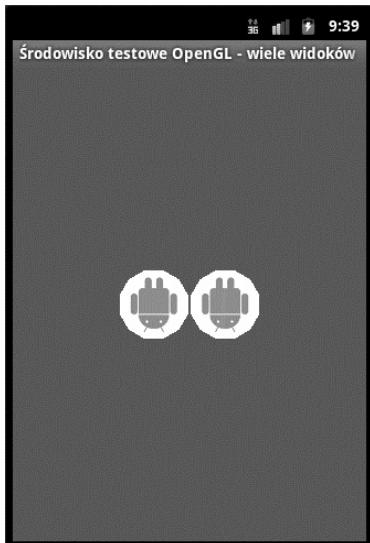


**Rysunek 20.12.** Para teksturowanych wielokątów

Na rysunku 20.13 został ukazany ten sam przykład w połowie procesu animacji.

Na tym zakończymy omawianie kolejnego istotnego aspektu środowiska OpenGL. W tym podrozdziale pokazaliśmy, w jaki sposób można zebrać wiele różnych figur geometrycznych lub scen i narysować je razem, dzięki czemu można otrzymać dosyć złożoną scenerię w środowisku OpenGL.

Zajmiemy się teraz obsługą środowiska OpenGL ES 2.0 w Androidzie.



Rysunek 20.13. Para teksturowanych kół

## OpenGL ES 2.0

Dobre wieści są takie, że oprócz samej obsługi środowiska OpenGL ES 2.0 system Android od wersji 2.2 (poziom 8. interfejsów API) posiada także odpowiednie powiązania języka Java. Musimy jednak pamiętać o następujących ograniczeniach:

- Środowisko OpenGL ES 2.0 nie jest jeszcze obsługiwane przez emulator.
- Środowisko OpenGL ES 2.0 różni się znacznie od poprzedniej wersji, więc większość książek poświęconych środowisku OpenGL posiada nowe wydania, w których omówiono ten aspekt. Programowanie w OpenGL odbywa się w obrębie jednostki GPU, co powoduje, że emulowanie takiego kodu staje się skomplikowaną czynnością. Z tego powodu nie jest nawet pewne, czy emulator będzie kiedykolwiek obsługiwał środowisko OpenGL ES 2.0.
- Jedynym sposobem testowania czy też poznawania środowiska OpenGL ES 2.0 w zestawie Android SDK jest wykorzystanie fizycznego urządzenia. Wkrótce większość urządzeń będzie pracowała pod kontrolą wersji 2.2 Androida, jednak istnieje pewne prawdopodobieństwo, że część z nich nie będzie obsługiwała nowej wersji środowiska OpenGL.

Środowisko OpenGL ES 2.0 jest znacząco odmienne od wersji 1.x. Dodatkowe komplikacje wynikają z braku wstępnej kompatybilności. Początkującym programistom najwięcej problemu sprawi jego inicializowanie oraz nauka rysowania najprostszych obiektów.

Dokładna analiza środowiska OpenGL ES 2.0 wymaga skrupulatnego przeczytania wielu stron różnorodnych informacji. Zamiast tego zapoznamy Czytelnika z tematem w stopniu umożliwiającym korzystanie z tego środowiska. Po utworzeniu podstawowego środowiska testowego Czytelnik będzie mógł skorzystać z odnośników umieszczonych na końcu rozdziału, w których znajdzie informacje pozwalające na wdrożenie środowiska OpenGL ES 2.0 do struktury aplikacji.

Potęga bibliotek OpenGL ES 2.0 polega na możliwości pisania dla jednostki graficznej programów, które są kompilowane na bieżąco, w czasie działania, i które pozwalają na interpretowanie rysowanych wierzchołków i fragmentów. Noszą one nazwę jednostek cieniujących (ang. *shader*). Niestety, wspomniane fragmenty kodu są wymagane nawet w przypadku najprostszych programów środowiska OpenGL ES 2.0. Wynika z tego, że zapoznanie się z pojęciem jednostek cieniujących jest niezbędne do korzystania z bibliotek OpenGL ES 2.0.

Pomogą nam w tym różnorodne źródła zamieszczone na końcu rozdziału.

## Powiązania środowiska Java z bibliotekami OpenGL ES 2.0

Powiązania środowiska Java z tym interfejsem API są dostępne w pakiecie `android.opengl`.  
→`GLES20`. Wszystkie funkcje tej klasy są statyczne i posiadają swoje odpowiedniki w określonych interfejsach API języka C, zdefiniowanych w specyfikacji grupy Khronos (adres URL znajdziemy na końcu rozdziału).

Omówione w tym rozdziale klasa `GLSurfaceView` oraz odpowiadająca jej klasa abstrakcyjna `Renderer`, używane w bibliotekach OpenGL ES 1.0, znajdują również zastosowanie w wersji 2.0. Wkrótce omówimy to zagadnienie.Więcej informacji na ten temat znajdziemy również w dokumentacji interfejsu API, w punkcie dotyczącym funkcji `GLSurfaceView.setEGLContextClientVersion`.

Sprawdźmy najpierw za pomocą kodu zawartego na listingu 20.36, czy dane urządzenie lub emulator obsługuje wersję 2.0 środowiska OpenGL ES.

---

**Listing 20.36.** Wykrywanie dostępności środowiska OpenGL ES 2.0

---

```
private boolean detectOpenGLES20() {
    ActivityManager am =
        (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    ConfigurationInfo info = am.getDeviceConfigurationInfo();
    return (info.reqGlEsVersion >= 0x20000);
}
```

---

Po wprowadzeniu tej funkcji (`detectOpenGLES20`) możemy zacząć korzystać w aktywności z klasą `GLSurfaceView`, tak jak zostało to ukazane na listingu 20.37.

---

**Listing 20.37.** Korzystanie z klasy `GLSurfaceView` w środowisku OpenGL ES 2.0

---

```
if (detectOpenGLES20())
{
    GLSurfaceView glview = new GLSurfaceView(this);
    // glview.setEGLConfigChooser(false);
    glview.setEGLContextClientVersion(2);

    glview.setRenderer(new YourGLES20Renderer(this));
    glview.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
    setContentView(glview);
}
```

---

Zwróćmy uwagę, że klasa `SurfaceView` została skonfigurowana do obsługi nowej wersji bibliotek OpenGL poprzez ustanowienie wersji 2 klienta. Tworzona tu klasa `TwojaKlasa`

→ Renderująca GLES będzie podobna do omawianych wcześniej klas typu Renderer. Jednak w ciele klasy renderującej będziemy korzystać z interfejsów GLES20, a nie GL10.

W tworzonym przez nas przykładzie klasa renderująca będzie nosiła nazwę ES20SimpleTriangleRenderer. Wkrótce zajmiemy się jej omówieniem, jednak tymczasem przyjrzymy się aktywności z listingu 20.38, do której wstawiliśmy fragmenty kodów z listingów 20.36 oraz 20.37.

#### **Listing 20.38.** Aktywność OpenGL20MultiViewTestHarness

---

```
public class OpenGL20MultiViewTestHarnessActivity extends Activity
{
    final String tag="es20";
    private GLSurfaceView mTestHarness;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (detectOpenGLES20())
        {
            mTestHarness = new GLSurfaceView(this);
            //NIE wywołujmy poniższej funkcji
            //mTestHarness.setEGLConfigChooser(false);
            mTestHarness.setEGLContextClientVersion(2);
        }
        else
        {
            throw new RuntimeException("wersja 2.0 nieobsługiwana");
        }

        Intent intent = getIntent();
        int mid = intent.getIntExtra("com.ai.menuid", R.id.MenuId_OpenGL15_Current);
        if (mid == R.id.mid_es20_triangle)
        {
            mTestHarness.setRenderer(new ES20SimpleTriangleRenderer(this));
            mTestHarness.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
            setContentView(mTestHarness);
            return;
        }
        return;
    }
    private boolean detectOpenGLES20() {
        ActivityManager am =
            (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        ConfigurationInfo info = am.getDeviceConfigurationInfo();
        return (info.reqGLESVersion >= 0x20000);
    }
    @Override
    protected void onResume() {
        super.onResume();
        mTestHarness.onResume();
    }
    @Override
    protected void onPause() {
        super.onPause();
```

```
    mTestHarness.onPause();
}
}
```

---

Zaprezentowana na listingu 20.38 aktywność środowiska testowego dla biblioteki OpenGL ES 2.0 bardzo przypomina aktywność przeznaczoną dla wersji 1.0 OpenGL ES z listingu 20.12. Czytelnik zastanawia się zapewne, czy nie można by po prostu użyć tamtej wcześniejszej aktywności i po prostu dodać kolejną opcję menu. Do wybrania prezentowanego rozwiązania skłoniły nas dwa powody.

Po pierwsze, nie mamy pewności, czy możemy wykorzystywać tę samą klasę SurfaceView pomiędzy wywołaniami różnych wersji bibliotek OpenGL ES. Przezorny zawsze ubezpieczony.

Drugi powód wynika z różnic w inicjalizowaniu obydwu bibliotek — nie chcemy komplikować kodu poprzez umieszczenie dwóch różnych rozwiązań wewnętrz jednej klasy. Przykładowo podczas inicjalizacji środowiska OpenGL ES 2.0 sprawdzamy dostępność obsługiwanej wersji bibliotek itd.; tego typu kod mógłby zaburzyć działanie prostszego systemu inicjalizacji środowiska OpenGL ES 1.0, widocznego na listingu 20.12.

Tak czy inaczej, motyw kierujące nami podczas tworzenia środowiska testowego dla nowej wersji biblioteki OpenGL są takie same jak poprzednio.

Aby móc korzystać z funkcji środowiska OpenGL ES 2.0 w aktywnościach, takich jak zaprezentowana na listingu 20.38, musimy wprowadzić znacznik <uses-feature> wewnętrz węźła aplikacji (listing 20.39).

---

**Listing 20.39.** Korzystanie z funkcji środowiska OpenGL ES 2.0

```
<application...>
    ....inne węzły
    <uses-feature android:glEsVersion="0x00020000" />
</application>
```

---

Nasza aplikacja musi posiadać włączony tryb debugowania za pomocą odpowiedniego atrybutu w węźle aplikacji, ponieważ będziemy mogli testować aplikacje wykorzystujące środowisko OpenGL ES 2.0 wyłącznie na urządzeniach fizycznych (listing 20.40).

---

**Listing 20.40.** Definiowanie aplikacji włączonej w trybie debugowania

```
<application android:icon="@drawable/icon"
    android:label="Środowisko testowe OpenGL"
    android:debuggable="true">
```

---

W celu przywołania nowej aktywności środowiska testowego musimy zmienić aktywność sterującą z listingu 20.14 w taki sposób, aby wyglądała jak na listingu 20.41.

---

**Listing 20.41.** Nowa główna aktywność sterująca

```
public class TestOpenGLMainActivity extends Activity {
    /** Wywoływana podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
```

```

        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu){
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater(); //z aktywności
        inflater.inflate(R.menu.main_menu, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        if (item.getItemId() >= R.id.mid_es20_triangle)
        {
            this.invoke20MultiView(item.getItemId());
            return true;
        }
        this.invokeMultiView(item.getItemId());
        return true;
    }
    private void invokeMultiView(int mid)
    {
        Intent intent = new Intent(this,MultiViewTestHarnessActivity.class);
        intent.putExtra("com.ai.menuid", mid);
        startActivity(intent);
    }
    private void invoke20MultiView(int mid)
    {
        Intent intent = new Intent(this,OpenGL20MultiViewTestHarnessActivity.class);
        intent.putExtra("com.ai.menuid", mid);
        startActivity(intent);
    }
}

```

Na listingu 20.41 wstawiliśmy dwa dodatkowe elementy: metodę wywołującą aktywność `OpenGL20MultiViewTestHarnessActivity`, która zostanie przywołana, gdy identyfikator menu znajdzie się powyżej lub na równi wartości elementu `mid_es20_triangle`. Nasz pomysł polega na tym, że ten element będzie uruchamiał wersje demonstracyjne funkcji środowiska OpenGL ES 2.0. Tym razem jednak korzystamy tylko z jednego przykładu.

## Etapy renderowania

Renderowanie figury geometrycznej w środowisku OpenGL ES 2.0 składa się z następujących etapów:

1. Zaprogramuj jednostki cieniące, które, działając we wnętrzu jednostki GPU, wykorzystują takie elementy, jak współrzędne rysowania oraz macierze modelu lub widoku bądź rzutowania pobrane z pamięci klienta, i rysują dane obiekty. W środowisku OpenGL ES 1.0 nie znajdziemy odpowiednika tego etapu. W uproszczonym ujęciu mamy tu do czynienia z kolejnym etapem pośrednim pomiędzy rysowaniem wierzchołków i powierzchni.
2. Skompiluj w układzie GPU jednostki cieniące utworzone w punkcie 1.

3. Połącz skompilowane jednostki z punktu 2. w obiekt programu, wykorzystywany w procesie rysowania.
4. Odczytaj procedury obsługi adresu z programu utworzonego w punkcie 3., dzięki czemu będziemy mogli przydzielać dane do tych wskaźników.
5. Zdefiniuj bufora wierzchołków.
6. Zdefiniuj macierze widoku modelu (dokonasz tego poprzez konfigurację ostrosłupa widzenia, położenia kamery itd.; w bardzo podobny sposób wykonuje się te czynności dla środowiska OpenGL ES 1.1).
7. Za pomocą procedur obsługi przekaż elementy z punktów 5. i 6. do programu.
8. Ostatnim etapem jest sam proces rysowania.

Pokażemy każdy z wymienionych etapów na przykładzie fragmentów kodów, a następnie zaprezentujemy silnik renderowania odpowiadający klasie `SimpleTriangleRenderer`, którą opisaliśmy w części poświęconej środowisku OpenGL ES 1.0. Rozpoczniemy od największej nowości w bibliotece OpenGL ES 2.0, mianowicie od jednostek cieniujących.

## Jednostki cieniujące

Nawet najprostsze obiekty w środowisku OpenGL 2.0 wymagają wykorzystania fragmentów kodów nazywanych **jednostkami cieniującymi**. Stanowią one rdzeń biblioteki OpenGL ES 2.0. Przekażemy teraz minimalną ilość informacji wymaganych do utworzenia prostego trójkąta; zalecamy zapoznanie się z materiałami źródłowymi wymienionymi na końcu rozdziału.

Każdy obiekt zawierający wierzchołki podlega przetwarzaniu przez **jednostki cieniujące wierzchołki** (ang. *vertex shader*). Z kolei każdy obiekt związany z fragmentami, czyli przestrzenią pomiędzy wierzchołkami, będzie objęty działaniem **jednostek cieniujących fragmenty** (ang. *fragment shader*). Zatem jednostka cieniąca wierzchołki przetwarza wyłącznie współrzędne wierzchołków, mimo że jednostka cieniąca fragmenty przetwarza każdy piksel.

Listing 20.42 stanowi prosty przykład jednostki cieniującej wierzchołki.

**Listing 20.42.** Prosta jednostka cieniująca wierzchołki

---

```
uniform mat4 uMVPMatrix;
attribute vec4 aPosition;
void main() {
    gl_Position = uMVPMatrix * aPosition;
}
```

---

Jest to kod zapisany w języku cieniowania. Z pierwszego wiersza dowiadujemy się, że zmienna `uMVPMatrix` jest zmienną wejściową programu, zadeklarowaną dla typu `mat4` (macierz  $4 \times 4$ ). Jest to również macierz typu `uniform`, ponieważ została ona zdefiniowana dla wszystkich wierzchołków, a nie tylko jednego.

Z drugiej strony mamy zmienną `aPosition`, która służy do definiowania współrzędnych wierzchołka. Została ona określona jako atrybut wierzchołka i jest niepowtarzalna dla każdego z nich. Wśród innych atrybutów wierzchołka znajdziemy takie, jak kolor, tekstura itp. Również zmiana `aPosition` jest wektorem o wartości równej 4. Następnie sam program (listing 20.42) pobiera współrzędne wierzchołka i przekształca je za pomocą macierzy MVP (ang. *Model View*

*Projection* — rzutowanie widoku modelu), która będzie ustanawiana przez program wywołujący, po czym przemnaża współrzędne wierzchołka, aby uzyskać ostateczne położenie, określone przez zarezerwowaną zmienną `gl_Position` jednostki cieniącej.

Jednostka cieniąjąca zapewnia rysowanie lub pozycjonowanie wierzchołków. Na przykład program wywołujący ustanowi bufor dla wierzchołków trójkąta w sposób zaprezentowany na listingu 20.43.

#### **Listing 20.43.** Ustanawianie danych wierzchołków

---

```
GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT, false,
    TRIANGLE_VERTICES_DATA_STRIDE_BYTES, mFVertexBuffer);
```

---

Bufor wierzchołka jest ostatnim argumentem tej metody `GLES20`. Bardzo przypomina ona metodę `glVertexPointer` z biblioteki OpenGL ES 1.0, oprócz pierwszego argumentu, który tutaj stanowi uchwyt położenia (`positionHandle`). Argument ten wskazuje atrybut `aPosition` z jednostki cieniącej, zamieszczonej na listingu 20.42. Taki uchwyt uzyskujemy za pomocą kodu podobnego do przedstawionego poniżej:

```
positionHandle = GLES20.glGetAttribLocation(shaderProgram, "aPosition");
```

Zasadniczo jednostka cieniąjąca ma przekazać uchwyt do zmiennej wejściowej. Sam obiekt `shaderProgram` musi zostać skonstruowany poprzez przekazanie fragmentów kodu jednostce GPU, w której następują ich komplikacja i łączenie. Aby napisać program, w którym możemy zacząć rysować, potrzebna nam będzie również jednostka cieniąca fragmenty. Listing 20.44 stanowi przykład takiej jednostki cieniącej.

#### **Listing 20.44.** Przykład jednostki cieniącej fragmenty

---

```
void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

---

Ponownie pobieramy zarezerwowaną zmienną `gl_FragColor` i przypisujemy jej czerwony kolor. Zamiast przypisywać jej kolor w kodzie, możemy przekazywać wartości kolorów od programu użytkownika poprzez jednostkę cieniąjącą wierzchołki aż po jednostkę cieniąjącą fragmenty. Opisanie tego mechanizmu wykracza poza zakres książki, jednak odpowiednie informacje można znaleźć w różnorodnych źródłach wymienionych na końcu tego rozdziału.

Obydwia rodzaje jednostek cieniących są niezbędne do rozpoczęcia procesu rysowania.

## Kompilowanie jednostek cieniących w programie

Po napisaniu segmentów jednostek cieniących, których przykłady znajdziemy na listingach 20.42 i 20.44, możemy wykorzystać kod zawarty na listingu 20.45 do skompilowania i wczytania danej jednostki cieniącej.

#### **Listing 20.45.** Kompilowanie i wczytywanie jednostki cieniącej

---

```
private int loadShader(int shaderType, String source) {
    int shader = GLES20.glCreateShader(shaderType);
    if (shader != 0) {
```

```
GLES20.glShaderSource(shader, source);
GLES20.glCompileShader(shader);
int[] compiled = new int[1];
GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compiled, 0);
if (compiled[0] == 0) {
    Log.e(TAG, "Nie mozna skompilowac jednostki cieniuujacej " + shaderType
          + ":");
    Log.e(TAG, GLES20.glGetShaderInfoLog(shader));
    GLES20.glDeleteShader(shader);
    shader = 0;
}
return shader;
}
```

---

W tym fragmencie kodu wartością argumentu shaderType może być GLES20.GL\_VERTEX\_SHADER albo GLES20.GL\_FRAGMENT\_SHADER. W zmiennej source musimy umieścić ciąg znaków zawierający źródło, takie jak zaprezentowane na listingach 20.42 lub 20.44.

Listing 20.46 ukazuje nam sposób, w jaki funkcja loadShader (listing 20.45) jest wykorzystywana podczas konstruowania obiektu programu.

#### **Listing 20.46.** Tworzenie programu i uzyskiwanie uchwytów zmiennych

---

```
private int createProgram(String vertexSource, String fragmentSource) {
    int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexSource);
    if (vertexShader == 0) {
        return 0;
    }
    Log.d(TAG, "utworzono jednostke cieniujaca wierzcholki");
    int pixelShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentSource);
    if (pixelShader == 0) {
        return 0;
    }
    Log.d(TAG, "utworzono jednostke cieniujaca fragmenty");
    int program = GLES20.glCreateProgram();
    if (program != 0) {
        Log.d(TAG, "utworzono program");
        GLES20.glAttachShader(program, vertexShader);
        checkGlError("glAttachShader");
        GLES20.glAttachShader(program, pixelShader);
        checkGlError("glAttachShader");
        GLES20.glLinkProgram(program);
        int[] linkStatus = new int[1];
        GLES20.glGetProgramiv(program, GLES20.GL_LINK_STATUS, linkStatus, 0);
        if (linkStatus[0] != GLES20.GL_TRUE) {
            Log.e(TAG, "Nie mozna dolaczyc programu: ");
            Log.e(TAG, GLES20.glGetProgramInfoLog(program));
            GLES20.glDeleteProgram(program);
            program = 0;
        }
    }
    return program;
}
```

---

## Uzyskiwanie dostępu do zmiennych jednostek cieniowania

Po skonfigurowaniu programu można wykorzystać jego uchwyt do uzyskania dostępu do zmiennych wejściowych wymaganych przez jednostki cieniące. Na listingu 20.47 widzimy, w jaki sposób możemy osiągnąć ten cel.

**Listing 20.47.** Uzyskiwanie uchwytów zmiennych aPosition i uMVPMatrix

---

```
int maPositionHandle =
    GLES20.glGetAttribLocation(mProgram, "aPosition");
int muMVPMatrixHandle =
    GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
```

---

## Prosty trójkąt napisany w środowisku OpenGL ES 2.0

Omówiliśmy już wszystkie podstawy wymagane do utworzenia struktury podobnej do zaprezentowanej w przypadku wersji 1.0 środowiska OpenGL. Zbierzemy teraz w całość wszystkie informacje dotyczące abstrakcyjnego silnika renderującego, który obsłuży cały proces inicjalizacji (na przykład utworzy jednostki cieniące, programy itp.). Na listingu 20.48 zaprezentowaliśmy jego kod.

**Listing 20.48.** Klasa ES20AbstractRenderer

---

```
public abstract class ES20AbstractRenderer
implements android.opengl.GLSurfaceView.Renderer
{
    public static String TAG = "ES20AbstractRenderer";

    private float[] mMMatrix = new float[16];
    private float[] mProjMatrix = new float[16];
    private float[] mVMMatrix = new float[16];
    private float[] mMVPMatrix = new float[16];

    private int mProgram;
    private int muMVPMatrixHandle;
    private int maPositionHandle;

    public void onSurfaceCreated(GL10 gl, EGLConfig eglConfig)
    {
        prepareSurface(gl, eglConfig);
    }
    public void prepareSurface(GL10 gl, EGLConfig eglConfig)
    {
        Log.d(TAG, "przygotowywanie powierzchni");
        mProgram = createProgram(mVertexShader, mFragmentShader);
        if (mProgram == 0) {
            return;
        }
        Log.d(TAG, "Uzyskiwanie uchwytu polozenia:aPosition");
        maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");
        checkGLError("glGetAttribLocation aPosition");
        if (maPositionHandle == -1) {
            throw new RuntimeException("Nie mozna uzyskac atr. polozenia dla aPosition");
        }
    }
}
```

---

```
    }
    Log.d(TAG,"Uzyskiwanie uchwytu macierzy:uMVPMatrix");
    muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");
    checkGlError("glGetUniformLocation uMVPMatrix");
    if (muMVPMatrixHandle == -1) {
        throw new RuntimeException("Nie mozna uzyskac atr. polozenia dla uMVPMatrix");
    }
}
public void onSurfaceChanged(GL10 gl, int w, int h)
{
    Log.d(TAG,"powierzchnia zmieniona. Ustanawianie macierzy ostrosl. widzenia:
        macierz rzutowania");
    GLES20.glViewport(0, 0, w, h);
    float ratio = (float) w / h;
    Matrix.frustumM(mProjMatrix, 0, -ratio, ratio, -1, 1, 3, 7);
}
public void onDrawFrame(GL10 gl)
{
    Log.d(TAG,"Ustanawianie macierzy widzenia");
    Matrix.setLookAtM(mVMatrix, 0, 0, 0, -5, 0f, 0f, 0f, 0f, 1.0f, 0.0f);

    Log.d(TAG,"podstawowa funkcja drawframe");
    GLES20.glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    GLES20.glClear( GLES20.GL_DEPTH_BUFFER_BIT | GLES20.GL_COLOR_BUFFER_BIT);

    GLES20.glUseProgram(mProgram);
    checkGlError("glUseProgram");

    draw(gl,this.maPositionHandle);
}
private int createProgram(String vertexSource, String fragmentSource) {
    int vertexShader = loadShader(GLES20.GL_VERTEX_SHADER, vertexSource);
    if (vertexShader == 0) {
        return 0;
    }
    Log.d(TAG,"utworzono jednostke cieniujaca wierzcholki");
    int pixelShader = loadShader(GLES20.GL_FRAGMENT_SHADER, fragmentSource);
    if (pixelShader == 0) {
        return 0;
    }
    Log.d(TAG,"utworzono jednostke cieniujaca fragmenty");
    int program = GLES20.glCreateProgram();
    if (program != 0) {
        Log.d(TAG,"utworzono program");
        GLES20.glAttachShader(program, vertexShader);
        checkGlError("glAttachShader");
        GLES20.glAttachShader(program, pixelShader);
        checkGlError("glAttachShader");
        GLES20.glLinkProgram(program);
        int[] linkStatus = new int[1];
        GLES20.glGetProgramiv(program, GLES20.GL_LINK_STATUS, linkStatus, 0);
        if (linkStatus[0] != GLES20.GL_TRUE) {
            Log.e(TAG, "Nie mozna dolaczyc programu: ");
            Log.e(TAG, GLES20.glGetProgramInfoLog(program));
            GLES20.glDeleteProgram(program);
            program = 0;
        }
    }
}
```

```

        }
    }
    return program;
}
private int loadShader(int shaderType, String source) {
    int shader = GLES20.glCreateShader(shaderType);
    if (shader != 0) {
        GLES20.glShaderSource(shader, source);
        GLES20.glCompileShader(shader);
        int[] compiled = new int[1];
        GLES20.glGetShaderiv(shader, GLES20.GL_COMPILE_STATUS, compiled, 0);
        if (compiled[0] == 0) {
            Log.e(TAG, "Nie mozna skompilowac jednostki cieniujacej "
                    + shaderType + ":");
            Log.e(TAG, GLES20.glGetShaderInfoLog(shader));
            GLES20.glDeleteShader(shader);
            shader = 0;
        }
    }
    return shader;
}
private final String mVertexShader =
    "uniform mat4 uMVPMatrix;\n" +
    "attribute vec4 aPosition;\n" +
    "void main() {\n" +
    "    gl_Position = uMVPMatrix * aPosition;\n" +
    "}\n";
private final String mFragmentShader =
    "void main() {\n" +
    "    gl_FragColor = vec4(0.5, 0.25, 0.5, 1.0);\n" +
    "}\n";
protected void checkGlError(String op) {
    int error;
    while ((error = GLES20.glGetError()) != GLES20.GL_NO_ERROR) {
        Log.e(TAG, op + ": glError " + error);
        throw new RuntimeException(op + ": glError " + error);
    }
}
protected void setupMatrices()
{
    Matrix.setIdentityM(mMMatrix, 0);
    Matrix.multiplyMM(mMVPMatrix, 0, mMMatrix, 0, mMMatrix, 0);
    Matrix.multiplyMM(mMVPMatrix, 0, mProjMatrix, 0, mMVPMatrix, 0);
    GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, mMVPMatrix, 0);
}
protected abstract void draw(GL10 gl, int positionHandle);
}

```

Większość powyższego kodu stanowi połączenie uprzednio omawianych koncepcji, za wyjątkiem jednego szczególnego. Funkcja `setupMatrices` ukazuje nam sposób, w jaki klasa `Matrix` jest wykorzystywana do łączenia wielu macierzy w jedną wspólną, zwaną `mMVPMatrix`, poprzez przemnożenie przez inne macierze, począwszy od macierzy jednostkowej.

Zatem zmienna `mMMatrix` jest macierzą jednostkową. Wartość zmiennej `mVMatrix` uzyskujemy poprzez zastosowanie interfejsu punktu ocznego lub punktu spoglądania kamery. Macierz rzutowania `mProjMatrix` jest dostępna za pomocą specyfikacji ostrosłupa widzenia. Obydwie koncepcje — punktu ocznego i ostrosłupa widzenia — są tak samo zdefiniowane jak w przypadku środowiska OpenGL 1.0. Macierz MVP stanowi jedynie iloczyn tych macierzy. W końcu, wywołanie funkcji `glUniformMatrix4fv` powoduje ustanowienie tej macierzy jako zmiennej w jednostce cieniowania wierzchołków, dzięki czemu poprzez przemnożenie współrzędnych wierzchołka przez tę macierz uzyskujemy ostateczne położenie (listing 20.42).

Na listingu 20.49 widzimy kod klasy `GS20SimpleTriangleRenderer` rozszerzającej abstrakcyjny silnik renderowania, w której została umieszczona minimalna liczba algorytmów umożliwiających zdefiniowanie punktów oraz ich narysowanie.

**Listing 20.49.** Klasa `GS20SimpleTriangleRenderer`

---

```
public class GS20SimpleTriangleRenderer extends ES20AbstractRenderer
{
    //Nieprzetworzony, natwryny bufor, przechowujacy wspolrzedne punktu
    private FloatBuffer mFVertexBuffer;
    private static final int FLOAT_SIZE_BYTES = 4;
    private final float[] mTriangleVerticesData = {
        //X, Y, Z
        -1.0f, -0.5f, 0,
        1.0f, -0.5f, 0,
        0.0f, 1.11803399f, 0 };

    public GS20SimpleTriangleRenderer(Context context)
    {
        ByteBuffer vbb = ByteBuffer.allocateDirect(mTriangleVerticesData.length
            * FLOAT_SIZE_BYTES);
        vbb.order(ByteOrder.nativeOrder());
        mFVertexBuffer = vbb.asFloatBuffer();
        mFVertexBuffer.put(mTriangleVerticesData);
        mFVertexBuffer.position(0);
    }

    protected void draw(GL10 gl, int positionHandle)
    {
        GLES20.glVertexAttribPointer(positionHandle, 3, GLES20.GL_FLOAT, false,
            0, mFVertexBuffer);
        checkGlError("glVertexAttribPointer maPosition");
        GLES20.glEnableVertexAttribArray(positionHandle);
        checkGlError("glEnableVertexAttribArray maPositionHandle");
        this.setupMatrices();
        GLES20.glDrawArrays(GLES20.GL_TRIANGLES, 0, 3);
        checkGlError("glDrawArrays");
    }
}
```

---

Jeżeli teraz wywołamy aktywność z listingu 20.38, ujrzymy trójkąt rysowany w danym kierunku. Aby aplikacja zadziałała, potrzebne nam będą dodatkowe pliki:

- `ES20AbstractRenderer.java` (listing 20.48),
- `ES20SimpleTriangleRenderer.java` (listing 20.49),
- `OpenGL20MultiveTestHarnessActivity.java` (listing 20.38).

Po skompilowaniu tych plików możemy ponownie uruchomić program i wybrać z menu opcję *Trójkąt ES20*. Zostanie wyświetlony jeden trójkąt, taki jak ten na rysunku 20.3.

Jednak, jak już stwierdziliśmy, powyższy przykładowy projekt nie zadziała na emulatorze. W celu jego przetestowania musimy podłączyć fizyczne urządzenie do środowiska Eclipse. Sprawdzaliśmy go na pierwszym modelu Motorola Droid firmy Verizon. Instrukcje dotyczące podłączenia urządzenia znajdziemy w rozdziale 2. W podrozdziale z odnośnikami zamieściliśmy także adres URL do witryny, w której zamierzamy dodawać zaktualizowane informacje powiązane również z innymi urządzeniami.

## Dodatkowe źródła dotyczące środowiska OpenGL ES 2.0

W podrozdziale „Odnośniki” znajdziemy informacje o zasobach związanych z biblioteką OpenGL ES 2.0. Gdy już zrozumiemy koncepcje jednostek cieniujących, mechanizmy działania środowiska OpenGL ES 1.0 oraz podstawowe wytyczne na temat jego wersji 2.0, możemy przetestować przykłady znajdujące się w zestawie Android SDK, pod warunkiem że posiadamy fizyczne urządzenie.

Zamieściliśmy cały kod źródłowy aplikacji generującej trójkąt w środowisku Eclipse wewnętrz projektu, który można pobrać z naszej oficjalnej strony. Projekt ten zawiera wszystkie etapy niezbędne do uruchomienia środowiska OpenGL ES 2.0.

## Instrukcje związane z komplikowaniem kodu

Najlepszym rozwiązaniem w kwestii testowania kodów umieszczonych w tym rozdziale jest pobranie pliku ZIP utworzonego specjalnie na potrzeby tego rozdziału. Adres URL tego pliku został umieszczony w podrozdziale „Odnośniki”. Znajdziemy w tym pliku kod każdej omówionej w tym rozdziale klasy. Jeżeli chcemy utworzyć aplikację bezpośrednio z listingów, znajdziemy w tym rozdziale wszystkie potrzebne pliki. Być może Czytelnik będzie musiał uwzględnić kilka własnych zasobów, na przykład ikonę aplikacji itd. W razie wątpliwości co do sposobu wstawiania ich do kodu należy posiłkować się gotowym projektem.

## Odnośniki

Następujące zasoby uznaliśmy za przydatne w zrozumieniu oraz podczas pracy w środowisku OpenGL:

- <http://developer.android.com/reference/android/opengl/GLSurfaceView.html> — adres odnoszący się do pakietu Androida `android.opengl`.
- [www.khronos.org/opengles/documentation/opengles1\\_0/html/index.html](http://www.khronos.org/opengles/documentation/opengles1_0/html/index.html) — podręcznik referencyjny środowiska OpenGL ES utworzony przez grupę Khronos.
- <http://www.glprogramming.com/red/> — podręcznik programowania w środowisku OpenGL („czerwona księga”). Chociaż materiały tu zawarte są bardzo przydatne, zakres informacji kończy się na wersji OpenGL ES 1.1. W celu zapoznania się z najnowszymi informacjami, w tym dotyczącymi jednostek cieniujących, musimy zaopatrzyć się w siódmą edycję księgi.
- [http://msdn.microsoft.com/en-us/library/ms970772\(printer\).aspx](http://msdn.microsoft.com/en-us/library/ms970772(printer).aspx) — bardzo dobry artykuł firmy Microsoft, dotyczący mapowania tekstur.

- <http://ezekiel.vancouver.wsu.edu/~cs442/> — bardzo wnikliwy kurs dotyczący środowiska OpenGL autorstwa Wayne'a O. Cochranego ze Stanowego Uniwersytetu Waszyngtońskiego.
- <http://java.sun.com/javame/reference/apis/jsr239/> — dokumentacja specyfikacji JSR 239 (Java Binding for the OpenGL ES API).
- [www.khronos.org/opengles/sdk/docs/man/](http://www.khronos.org/opengles/sdk/docs/man/) — fragmenty podręcznika dotyczące środowiska OpenGL ES 2.0 grupy Khronos, dobre jako źródło dalszych odnośników, a nie samych informacji.
- [www.opengl.org/documentation/glsl/](http://www.opengl.org/documentation/glsl/) — przydatny dla osoby, która chce zrozumieć kierunek obrany w środowisku OpenGL ES 2.0, i do ułatwienia niezbędnego opanowania języka cieniowania.
- *OpenGL Shading Language*, wydanie trzecie, Randi J. Rost i inni. Osobiście nie czytaliśmy tej książki, ale zapowiada się obiecująco.
- <http://developer.android.com/reference/android/opengl/GLES20.html> — odniesienie do interfejsu GLES20 z zestawu Android SDK.
- [http://developer.android.com/reference/android/opengl/GLSurfaceView.html#setEGLContextClientVersion\(int\)](http://developer.android.com/reference/android/opengl/GLSurfaceView.html#setEGLContextClientVersion(int)) — odniesienie do klasy GLSurfaceView.
- [http://www.androidbook.com/akc/display?url=NotesIMPTitlesURL&ownerUserId=saty&folderName=OpenGL&order\\_by\\_format=news](http://www.androidbook.com/akc/display?url=NotesIMPTitlesURL&ownerUserId=saty&folderName=OpenGL&order_by_format=news) — badania środowiska OpenGL przeprowadzone przez jednego z autorów książki.
- <http://www.androidbook.com/item/3190> — badania dotyczące tekstur w środowisku OpenGL przeprowadzone przez jednego z autorów książki.
- <http://www.androidbook.com/item/3574> — instrukcje dotyczące uruchomienia aplikacji na fizycznym urządzeniu z poziomu środowiska Eclipse.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — z tego adresu możemy pobrać projekty utworzone z myślą o niniejszej książce. Projekty z tego rozdziału zostały umieszczone w katalogu *ProAndroid3\_R20\_OpenGL*.

## Podsumowanie

Poświęciliśmy mnóstwo miejsca standardowi OpenGL, co może być przydatne zwłaszcza dla osób mających z nim do czynienia po raz pierwszy. Chcielibyśmy, aby było to znakomite wprowadzenie do środowiska OpenGL nie tylko dla systemu Android, lecz również dla innych systemów obsługujących ten standard.

W rozdziale tym przedstawiliśmy podstawowe informacje na temat biblioteki OpenGL. Pokazaliśmy interfejs API (dostępny wyłącznie w systemie Android), umożliwiający pracę ze standardowymi interfejsami środowiska OpenGL. Omówiliśmy pojęcia kształtów i tekstur, a także zademonstrowaliśmy, w jaki sposób można wykorzystać potok rysowania do tworzenia wielu obiektów. Poznaliśmy podstawy środowiska OpenGL ES 2.0, jego język cieniowania, podstawowe różnice pomiędzy tą a poprzednią wersją bibliotek, a także różnorodne odnośniki pozwalające na dalsze zgłębianie zagadnienia.

# Badanie aktywnych folderów

Wprowadzone w wersji 1.5 środowiska Android aktywne foldery umożliwiają projektantom umieszczanie takich dostawców treści, jak kontakty, notatki oraz multimedia, w domyślnym ekranie startowym urządzenia (które będziemy nazywać **stroną startową** urządzenia). Kiedy dostawca treści, na przykład *contacts*, zostanie umieszczony na stronie głównej jako aktywny folder, będzie on automatycznie odświeżany w razie dodawania, usuwania lub modyfikowania kontaktów w bazie danych. W tym rozdziale wyjaśnimy sens istnienia aktywnych folderów, sposoby ich implementacji oraz ich „uaktywnienia”.

## Badanie aktywnych folderów

W Androidzie aktywny folder (ang. *live folder*) ma się do dostawcy treści tak, jak czytnik RSS do publikowania strony internetowej. W rozdziale 4. stwierdziliśmy, że dostawcy treści przypominają strony internetowe dostarczające informacje poprzez identyfikatory URI. W miarę rozpowszechniania witryn sieciowych, z których w każdej publikowano informacje na swój własny sposób, pojawiła się potrzeba zbierania treści z wielu różnych stron internetowych, aby dać użytkownikowi możliwość ich śledzenia za pomocą jednego czytnika. RSS stał się wspólnym wzorcem wielu różnorodnych zbiorów informacji. Dzięki takiemu wzorcowi istnieje możliwość zaprojektowania czytnika, który będzie odczytywał dane, dopóki będą one posiadały jednolitą strukturę.

Taka sama jest koncepcja aktywnych folderów. Podobnie jak czytnik RSS zapewnia wspólny interfejs dla treści opublikowanych w sieci, tak aktywny folder definiuje wspólny interfejs dla dostawcy treści w Androidzie. Dopóki dostawca treści lub klasa osłonowa tego dostawcy spełniają wymagania protokołu, możemy w Androidzie utworzyć ikonę aktywnego folderu, reprezentującą tego dostawcę na stronie startowej urządzenia. Po kliknięciu tej ikony system automatycznie skontaktuje się z dostawcą treści. Spodziewamy się zatem, że dostawca treści przekaże kursor. Zgodnie z kontraktem aktywnego folderu kursor ten musi posiadać predefiniowany zbiór kolumn. Następnie jest on wyświetlany poprzez widoki *ListView* lub *GridView*.

Opierając się na tych koncepcjach, działanie aktywnych folderów można przedstawić w następujący sposób:

1. Najpierw na stronie startowej tworzymy ikonę reprezentującą zbiór wierszy pochodzących od dostawcy treści. Tworzymy takie powiązanie poprzez przypisanie identyfikatora URI do ikony.
2. Kiedy użytkownik kliknie tę ikonę, system pobiera identyfikator URI i wykorzystuje go do wywołania dostawcy treści. Dostawca ten poprzez kurSOR zwraca zbiór wierszy.
3. Dopóki kurSOR zawiera kolumny, które mogą być rozpoznane przez aktywny folder (na przykład nazwę, opis oraz program wywoływany po kliknięciu wiersza), system będzie je wyświetlał jako widoki *ListView* lub *GridView*.
4. Ponieważ widoki *ListView* oraz *GridView* mają możliwość aktualizowania swoich danych podczas wprowadzania zmian w bazie danych, widoki te są nazywane **aktywnymi** — stąd wzięła się nazwa „aktywne foldery”.

Pracując z aktywnymi folderami, należy pamiętać o dwóch podstawowych zasadach. Pierwsza z nich polega na definiowaniu wspólnych nazw kolumn dla wszystkich kurSORów. Dzięki niej wszystkie kurSory przeznaczone dla aktywnych folderów w Androidzie są traktowane tak samo. Wedle drugiej zasady widoki w Androidzie potrafią wyszukiwać aktualizacje danych kurSORa i odpowiednio się do nich dostosowywać. Nie jest to reguła specyficzna jedynie dla aktywnych folderów; w rzeczywistości okazuje się ona standardowa dla wszystkich widoków interfejsu UI w Androidzie, zwłaszcza dla widoków korzystających z kurSORa.

Skoro przedstawiliśmy już koncepcję aktywnych folderów, będziemy systematycznie zagłębiać się w ich struktury. Poszczególne informacje przedstawimy w dwóch podrozdziałach. W pierwszym z nich wnikliwie przeanalizujemy, w jaki sposób użytkownik korzysta z aktywnego folderu. Po tej części aktywne foldery staną się jeszcze bardziej zrozumiałe.

W drugim podrozdziale zaprezentujemy sposoby poprawnego tworzenia tych struktur, aby były naprawdę aktywne. Aby „uaktywnić” folder, należy przeprowadzić kilka dodatkowych czynności, zatem zajmiemy się tym wcale nie tak oczywistym aspektem aktywnych folderów.

## **W jaki sposób użytkownik korzysta z aktywnych folderów**

Aktywne foldery są dostępne dla użytkowników poprzez stronę startową urządzenia. Poniższa sekwencja przedstawia sposób wykorzystywania aktywnych folderów:

1. Otwórz stronę startową urządzenia.
2. Otwórz menu kontekstowe strony startowej. Zostaje ono wyświetlone po długim kliknięciu pustej przestrzeni na ekranie startowym.
3. Znajdź w menu kontekstowym opcję *Foldery* i kliknij ją, aby ujrzeć listę dostępnych aktywnych folderów.
4. Na wyświetlonej liście zaznacz nazwę aktywnego folderu, który chcesz umieścić na stronie startowej. Zostanie utworzona ikona reprezentująca wybrany folder aktywny.
5. Kliknij ikonę konfiguracji aktywnego folderu, utworzoną w punkcie 4., aby wyświetlić wiersze zawierające informacje (dane reprezentowane przez ten aktywny folder) w widokach *ListView* lub *GridView*.
6. Kliknij jeden z wierszy, aby przywołać aplikację wyświetlającą dane zawarte w tym wierszu.
7. Za pomocą opcji menu wyświetlanych przez aplikację możesz przeglądać elementy lub manipulować danym elementem. Za ich pomocą możesz również tworzyć nowe elementy dozwolone przez aplikację.

8. Zwróć uwagę, że aktywne foldery automatycznie odzwierciedlają wszelkie zmiany dokonane na elemencie lub zbiorze elementów.

Omówimy wszystkie powyższe etapy, ilustrując każdy z nich zrzutem ekranu. Rozpoczniemy od punktu 1., czyli typowej strony startowej Androida (rysunek 21.1). Strona startowa może się nieznacznie różnić w zależności od stosowanej wersji Androida oraz urządzenia.



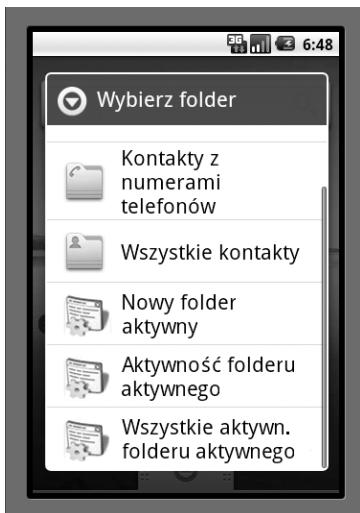
Rysunek 21.1. Strona startowa Androida

Jeżeli na ekranie startowym wykonamy długie kliknięcie, Android wyświetli jego menu kontekstowe (rysunek 21.2).



Rysunek 21.2. Menu kontekstowe strony startowej w Androidzie

Po wybraniu opcji *Foldery* pojawi się kolejne menu, przedstawiające dostępne foldery aktywne (rysunek 21.3). W następnym podrozdziale pokażemy, jak utworzyć aktywny folder, na razie jednak założmy, że jest on już zbudowany i nosi nazwę *Nowy aktywny folder* (rysunek 21.3).



Rysunek 21.3. Przeglądanie listy dostępnych aktywnych folderów

#### Uwaga!

Jeżeli Czytelnik chce przyjrzeć się naszej przykładowej aplikacji jeszcze przed jej utworzeniem, może pobrać tworzący ją projekt i zainstalować go na emulatorze. Adres strony, z której można pobrać projekt, znajduje się w podrozdziale „Odnośniki”. Potrzebna będzie również aplikacja obsługująca kontakty, która stanowi część zestawu SDK i pozwala umieścić na emulatorze kilka przykładowych kontaktów. Po pobraniu i zimportowaniu projektu do środowiska Eclipse możemy go uruchomić na emulatorze w postaci aktywnego folderu. Efekt końcowy będzie przypominał ekran widoczny na rysunku 21.3.

Po kliknięciu opcji *Nowy aktywny folder* na stronie startowej Androida zostanie utworzona ikona reprezentująca aktywny folder. W naszym przykładzie nazwą tej ikony będzie *Kontakty AF* — skrót od „Kontakty Aktywny Folder” (rysunek 21.4). W folderze tym będą wyświetlane kontakty z bazy kontaktów. W trakcie implementacji aktywnego folderu pokażemy sposob, w jaki jest definiowana jego nazwa.

W następnym podrozdziale będzie się można przekonać, że to aktywność zapewnia utworzenie folderu *Kontakty AF*. Na razie interesują nas wrażenia użytkownika, zatem kliknijmy ikonę *Kontakty AF*, aby ujrzeć listę kontaktów wyświetlzoną w widoku *ListView* (rysunek 21.5). Jeszcze raz przypominamy, że w zależności od posiadanej wersji systemu lista ta może wyglądać inaczej.

Wygląd listy może być różny od zaprezentowanego, gdyż zależy on od liczby posiadanych kontaktów. Po kliknięciu jednego z kontaktów ujrzymy jego szczegóły (rysunek 21.6). Zwrócmy uwagę, że szczegóły kontaktu są wyświetlane poprzez aplikację obsługującą kontakty, zatem wygląd tego aspektu również zależy od wersji systemu Android.



Rysunek 21.4. Ikona aktywnego folderu na ekranie startowym



Rysunek 21.5. Wyświetlanie aktywnego folderu kontaktów

Możemy kliknąć znajdujący się u dołu ekranu przycisk *Menu*, aby zobaczyć możliwości edycyjne danego kontaktu (rysunek 21.7). Także ten etap jest obsługiwany przez aplikację, więc jego stylistyka jest zależna od rodzaju urządzenia i wersji Androida.

Po wybraniu opcji edycji kontaktu pojawi się ekran (również o wyglądzie zależnym od wersji systemu) ukazany na rysunku 21.8.

Aby ujrzeć „aktywne” zachowanie się tego folderu, można zaktualizować imię lub nazwisko kontaktu. Po powrocie do widoku folderu *Kontakty AF* ujrzymy, że wprowadzone zmiany zostały uwzględnione. W tym celu należy wielokrotnie kliknąć przycisk cofania, dopóki nie wróćmy do folderu *Kontakty AF*.



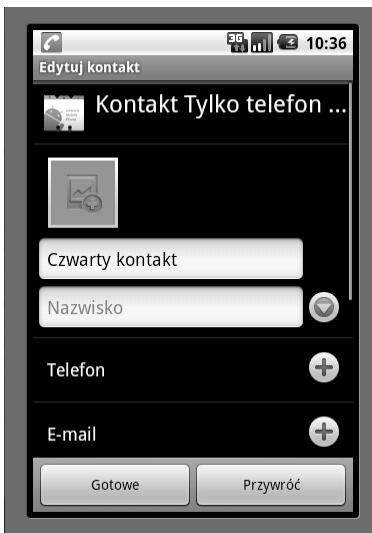
Rysunek 21.6. Otwieranie aktywnego folderu kontaktów



Rysunek 21.7. Opcje menu pojedynczego kontaktu

## Tworzenie aktywnego folderu

Wyjaśniliśmy, czym są aktywne foldery oraz do czego służą. W celu wygenerowania aktywnego folderu wymagane są dwa elementy: aktywność i wyspecjalizowany dostawca treści. Android wykorzystuje **etykietę** tej aktywności do zapełnienia widocznej na rysunku 21.3 listy dostępnych aktywnych folderów. Android wywołuje również tę aktywność w celu uzyskania identyfikatora URI umożliwiającego otrzymanie listy wyświetlanych wierszy.



**Rysunek 21.8.** Edycja szczegółowych informacji o kontakcie

Dostarczany przez aktywność identyfikator URI powinien wskazywać wyspecjalizowanego dostawcę treści, który będzie przekazywał krotki. Dostawca przekazuje te krotki poprzez właściwie zdefiniowany kursor. Stwierdzamy, że kursor jest **właściwie zdefiniowany**, ponieważ oczekujemy, że będzie zawierał predefiniowany zestaw nazw kolumn.

Zazwyczaj umieszcza się te dwa elementy w aplikacji, a następnie wdraża się ją na urządzenie. Trzeba również zapewnić sobie kilka pomocniczych plików, bez których aktywne foldery nie będą działać. Objaśnimy i zademonstrujemy wymienione koncepcje na przykładowym projekcie, składającym się z następujących plików:

- **AndroidManifest.xml** — w tym pliku określono aktywności, wywoływanie w celu utworzenia definicji aktywnego folderu.
- **AllContactsLiveFolderCreatorActivity.java** — ta aktywność dostarcza definicję aktywnego folderu, pozwalającą na wyświetlanie wszystkich kontaktów z bazy danych.
- **MyContactsProvider.java** — ten dostawca treści reaguje na identyfikator URI aktywnego folderu, przekazującego kursor z kontaktami. Dostawca ten wewnętrznie wykorzystuje dostępnego w Androidzie dostawcę treści kontaktów.
- **MyCursor.java** — jest to wyspecjalizowany kursor, wykonujący operację `requery` podczas zmiany danych.
- **BetterCursorWrapper.java** — plik ten służy klasie `MyCursor` do przeprowadzania operacji `requery`.

Omówimy każdy plik po kolejno, aby łatwiej było zrozumieć zasadę działania aktywnych folderów.

## AndroidManifest.xml

Mieliśmy już wielokrotnie do czynienia z plikiem `AndroidManifest.xml`. Jest on konieczny do działania wszystkich aplikacji. Fragment pliku dotyczący aktywnych folderów, który został odzielony komentarzem, wskazuje istnienie aktywności `AllContactsLiveFolderCreatorActivity`

umożliwiającej utworzenie aktywnego folderu (listing 21.1). Fakt ten został wyrażony poprzez deklarację intencji, której działaniem jest `android.intent.action.CREATE_LIVE_FOLDER`.

#### **Listing 21.1.** Plik AndroidManifest.xml definicji aktywnego folderu

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.livefolders"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <!-- AKTYWNE FOLDERY -->
        <activity
            android:name=".AllContactsLiveFolderCreatorActivity"
            android:label="Nowy aktywny folder"
            android:icon="@drawable/icon">
            <intent-filter>
                <action android:name="android.intent.action.CREATE_LIVE_FOLDER" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
        <provider android:authorities="com.androidbook.livefolders.contacts"
            android:multiprocess="true"
            android:name=".MyContactsProvider" />
    </application>
    <uses-sdk android:minSdkVersion="3" />
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
</manifest>
```

---

Etykieta tej aktywności, *Nowy aktywny folder*, pojawi się w menu kontekstowym strony startowej (rysunek 21.3). Jak zostało wyjaśnione w punkcie „W jaki sposób użytkownik korzysta z aktywnych folderów”, dostęp do menu kontekstowego strony startowej uzyskujemy poprzez długie kliknięcie na obszarze tego ekranu startowego.

Kolejnym godnym uwagi elementem kodu z listingu 21.1 jest deklaracja provider, zakotwiczona do identyfikatora URI `content://com.androidbook.livefolders.contacts` i obsługiwana przez klasę dostawcy `MyContactsProvider`. Dostawca ten dostarcza kursor wypełniający kontrolkę `ListView`, która zostaje otwarta po kliknięciu odpowiedniej ikony aktywnego folderu (rysunek 21.5). Aktywność `AllContactsLiveFolderCreatorActivity` takiego folderu musi „wiedzieć”, czym jest ten identyfikator URI, i przekazać go po wywołaniu przez system. Android przywołuje tę aktywność po wybraniu nazwy aktywnego folderu, aby utworzyć jego ikonę na ekranie startowym.

Zgodnie z protokołem aktywnych folderów intencja `CREATE_LIVE_FOLDER` będzie pozwalała wyświetlać aktywność `AllContactsLiveFolderCreatorActivity` jako opcję zatytułowaną *Nowy aktywny folder* w menu kontekstowym strony startowej (rysunek 21.3). Kliknięcie tej opcji spowoduje utworzenie ikony na stronie startowej, co zostało pokazane na rysunku 21.4.

Zadaniem aktywności AllContactsLiveFolderCreatorActivity jest zdefiniowanie ikony, składającej się z obrazu i etykiety. W naszym przypadku kod aktywności AllContactsLiveFolderCreatorActivity nadaje etykiecie nazwę *Kontakty AF* (listing 21.2). Przyjrzyjmy się zatem kodowi źródłowemu tego kreatora aktywnych folderów.

## AllContactsLiveFolderCreatorActivity.java

Klasa AllContactsLiveFolderCreatorActivity ma do spełnienia jedną rolę: generatorki czy też kreatora aktywnego folderu (listing 21.2). Możemy ją sobie wyobrazić jako szablon takiego folderu. Po każdym kliknięciu tej aktywności (poprzez opcję *Foldery* w menu kontekstowym ekranu startowego) zostanie wygenerowany aktywny folder.

**Listing 21.2.** Kod źródłowy klasy AllContactsLiveFolderCreatorActivity

```
public class AllContactsLiveFolderCreatorActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        final Intent intent = getIntent();
        final String action = intent.getAction();

        if (LiveFolders.ACTION_CREATE_LIVE_FOLDER.equals(action)) {
            setResult(RESULT_OK,
                      createLiveFolder(MyContactsProvider.CONTACTS_URI,
                                      "Kontakty AF",
                                      R.drawable.icon)
            );
        }
        else {
            setResult(RESULT_CANCELED);
        }
        finish();
    }

    private Intent createLiveFolder(Uri uri, String name, int icon)
    {
        final Intent intent = new Intent();
        intent.setData(uri);
        intent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_NAME, name);
        intent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_ICON,
                       Intent.ShortcutIconResource.fromContext(this, icon));
        intent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_DISPLAY_MODE,
                       LiveFolders.DISPLAY_MODE_LIST);
        return intent;
    }
}
```

Aby wykonać to zadanie, aktywność podaje obiektowi wywołującemu — stronie startowej lub, w naszym przypadku, strukturze aktywnego folderu — nazwę aktywnego folderu, obraz ikony tego folderu, identyfikator URI danych oraz tryb wyświetlania (lista lub siatka). Z kolei struktura zapewnia utworzenie ikony aktywnego folderu na ekranie startowym.

**Uwaga!**

W dokumentacji zestawu Android SDK dotyczącej klasy android.provider.LiveFolders znajduje się spis wszystkich kontraktów wymaganych przez aktywny folder.

Metoda `createLiveFolder` przede wszystkim ustanawia wartości wobec intencji wywołującej. Po przekazaniu intencji procedurze wywołującej procedura ta otrzyma następujące informacje:

- nazwę aktywnego folderu;
- obraz wykorzystywany jako ikona aktywnego folderu;
- tryb wyświetlania: lista lub siatka;
- identyfikator URI danych lub treści, służący do przywoływania informacji.

Informacje te wystarczą do utworzenia ikony aktywnego folderu, przedstawionej na rysunku 21.4. Kiedy użytkownik ją kliknie, zostanie wywołany identyfikator URI, dzięki któremu zostaną odczytane dane. Zadaniem dostawcy treści określonego dzięki temu identyfikatorowi jest dostarczenie znormalizowanego kurSORA. Zademonstrujemy teraz kod tego dostawcy treści — klasę MyContactsProvider.

## MyContactsProvider.java

Przed klasą MyContactsProvider stoją następujące zadania:

1. Rozpoznanie przychodzącego identyfikatora URI  
`content://com.androidbook.livefolders.contacts/contacts`.
2. Wewnętrzne wywołanie dostawcy treści kontaktów AndroIDA, identyfikowanych adresem `content://contacts/people/` (zwracajmy szczególną uwagę na aplikację obsługującą kontakty, ponieważ adres URL do niej może ulegać zmianie wraz z każdą nową wersją systemu).
3. Odczytanie wszystkich krotek kurSORA oraz odwzorowanie ich na kurSORZE typu `MatrixCursor` wraz z umieszczeniem poprawnych nazw kolumn wymaganych przez strukturę aktywnego folderu.
4. Umieszczenie obiektu `MatrixCursor` w innym kurSORZE, aby przeprowadzenie operacji `requery` na tym obiekcie powodowało w razie potrzeby wywołanie dostawcy treści kontaktów.

Kod dostawcy MyContactsProvider został umieszczony na listingu 21.3. Istotne elementy zostały zaznaczone pogrubioną czcionką i są oparte na omówionych powyżej założeniach. Objaśnienie kodu znajdziemy pod listingiem.

---

**Listing 21.3.** Kod źródłowy klasy MyContactsProvider

```
public class MyContactsProvider extends ContentProvider {  
  
    public static final String AUTHORITY = "com.androidbook.livefolders.contacts";  
  
    //Identyfikator Uri biorący udział w procesie tworzenia aktywnego folderu jako dane  
    //wejściowe  
    public static final Uri CONTACTS_URI = Uri.parse("content://" +  
        AUTHORITY + "/contacts");  
    //Aby ten identyfikator URI został rozpoznany  
    private static final int TYPE_MY_URI = 0;  
    private static final UriMatcher URI_MATCHER;  
    static{
```

```
URI_MATCHER = new UriMatcher(UriMatcher.NO_MATCH);
URI_MATCHER.addURI(AUTHORITY, "contacts", TYPE_MY_URI);
}

@Override
public boolean onCreate() {
    return true;
}

@Override
public int bulkInsert(Uri arg0, ContentValues[] values) {
    return 0; //Niczego nie wstawiamy
}

//Zbiór kolumn wymaganych przez aktywny folder
//Jest to kontrakt aktywnego folderu
private static final String[] CURSOR_COLUMNS = new String[]
{
    BaseColumns._ID,
    LiveFolders.NAME,
    LiveFolders.DESCRIPTION,
    LiveFolders.INTENT,
    LiveFolders.ICON_PACKAGE,
    LiveFolders.ICON_RESOURCE
};

//W przypadku braku krotek
//wprowadzamy zastępstwo w postaci komunikatu o błędzie
//Zauważmy, że posiada taki sam zbiór kolumn jak aktywny folder
private static final String[] CURSOR_ERROR_COLUMNS = new String[]
{
    BaseColumns._ID,
    LiveFolders.NAME,
    LiveFolders.DESCRIPTION
};

//Krotka komunikatu o błędzie
private static final Object[] ERROR_MESSAGE_ROW =
    new Object[]
    {
        -1,                                     //identyfikator
        "Nie znaleziono kontaktów",             //nazwa
        "Sprawdź bazę kontaktów"               //opis
    };

//Stosowany kursor błędu
private static MatrixCursor sErrorCursor = new
    MatrixCursor(CURSOR_ERROR_COLUMNS);
static
{
    sErrorCursor.addRow(ERROR_MESSAGE_ROW);
}
```

```
//Kolumny odczytywane z bazy kontaktów
private static final String[] CONTACTS_COLUMN_NAMES = new String[]
{
    ContactsContract.Contacts._ID,
    ContactsContract.Contacts.DISPLAY_NAME,
    ContactsContract.Contacts.TIMES_CONTACTED,
    ContactsContract.Contacts.STARRED
};

public Cursor query(Uri uri, String[] projection, String selection,
                     String[] selectionArgs, String sortOrder)
{
    //Sprawdza identyfikator Uri i zwraca błąd, jeżeli nie znajdzie dopasowania
    int type = UriMatcher.match(uri);
    if(type == UriMatcher.NO_MATCH)
    {
        return sErrorCursor;
    }

    Log.i("ss", "kwerenda wywołana");

    try
    {
        MatrixCursor mc = loadNewData(this);
        mc.setNotificationUri(getContext().getContentResolver(),
                             Uri.parse("content://contacts/people/"));
        MyCursor wmc = new MyCursor(mc,this);
        return wmc;
    }
    catch (Throwable e)
    {
        return sErrorCursor;
    }
}

public static MatrixCursor loadNewData(ContentProvider cp)
{
    MatrixCursor mc = new MatrixCursor(CURSOR_COLUMNS);
    Cursor allContacts = null;
    try
    {
        allContacts = cp.getContext().getContentResolver().query(
            ContactsContract.Contacts.CONTENT_URI,
            CONTACTS_COLUMN_NAMES,
            null, //filtr krotek
            null,
            ContactsContract.Contacts.DISPLAY_NAME); //sortowanie

        while(allContacts.moveToNext())
        {
            String timesContacted = "Nawiązane połączenia: "+allContacts.getInt(2);

            Object[] rowObject = new Object[]
            {
                allContacts.getLong(0), //identyfikator
                allContacts.getString(1), //nazwa
            };
        }
    }
}
```

```

        timesContacted,                                //opis
        Uri.parse("content://contacts/people/"           //id. Uri intencji
                  +allContacts.getLong(0)),           //pakiet
        cp.getContext().getPackageName(),               //ikona
        R.drawable.icon
    };
    mc.addRow(rowObject);
}
return mc;
}
finally
{
    allContacts.close();
}
}

@Override
public String getType(Uri uri)
{
    //Wskazuje typ MIME danego identyfikatora Uri
    //zdefiniowanego dla osłonowego dostawcy
    //Typ ten wygląda zazwyczaj następująco:
    // "vnd.android.cursor.dir/vnd.google.note"
    return ContactsContract.Contacts.CONTENT_TYPE;
}

public Uri insert(Uri uri, ContentValues initialValues) {
    throw new UnsupportedOperationException(
        "nic nie zostaje wstawione, ponieważ jest to wyłącznie osłona");
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    throw new UnsupportedOperationException(
        "nic nie zostaje usunięte, ponieważ jest to wyłącznie osłona");
}

public int update(Uri uri, ContentValues values,
                  String selection, String[] selectionArgs)
{
    throw new UnsupportedOperationException(
        "nic nie zostaje zaktualizowane, ponieważ jest to wyłącznie osłona");
}
}

```

Zwróćmy uwagę, że wymagany przez strukturę aktywnego folderu zbiór kolumn zostaje zainicjalizowany w kodzie z listingu 21.3, a na listingu 21.4 zostaje wywołany w postaci natymiarstwego odniesienia.

#### **Listing 21.4.** Kolumny potrzebne do wypełnienia kontraktu aktywnego folderu

```

private static final String[] CURSOR_COLUMNS = new String[]
{
    BaseColumns._ID,

```

```
LiveFolders.NAME,  
LiveFolders.DESCRIPTION,  
LiveFolders.INTENT,  
LiveFolders.ICON_PACKAGE,  
LiveFolders.ICON_RESOURCE  
};
```

---

Poza elementem **INTENT** przeznaczenie pozostałych obiektów jest oczywiste. Jeżeli przyjrzymy się rysunkowi 21.5, zauważymy, że obiekt **NAME** dotyczy nazwy elementu na liście. Atrybut **DESCRIPTION** został umieszczony na tej samej liście pod obiektem **NAME**.

Pole **INTENT** jest w rzeczywistości polem typu **string**, wskazującym identyfikator URI danego elementu w dostawcy treści. W przypadku kliknięcia tego elementu Android zastosuje działanie **VIEW** poprzez ten identyfikator URI. Dlatego właśnie pole to nosi nazwę pola **INTENT**, ponieważ wewnętrznie Android uzyska obiekt **INTENT** z identyfikatora URI.

Dwa ostatnie elementy są związane z obiektem **ICON**, wyświetlany jako część listy. Przyjrzyjmy się ponownie rysunkowi 21.5, aby zobaczyć ikony, oraz listingowi 21.3, aby sprawdzić, w jaki sposób kolumny te dostarczają wartości z bazy kontaktów.

Zwróciły również uwagę, że klasa **MyContactsContentProvider** (osłonowy dostawca treści) wykonuje kod z listingu 21.5 wymuszając na podstawowym kurSORZE obsługę wszelkich zmian danych.

#### **Listing 21.5.** Rejestrowanie identyfikatora URI za pomocą kursora

---

```
MatrixCursor mc = loadNewData(this);  
mc.setNotificationUri(getContext().getContentResolver(),  
    Uri.parse("content://contacts/people/"));
```

---

Funkcja **loadNewData()** uzyskuje od dostawcy treści zbiór kontaktów i tworzy obiekt **MatrixCursor**, którego kolumny są widoczne na listingu 21.4. Następnie obiekt ten otrzymuje informację, że ma się zarejestrować wraz z klasą **ContentResolver**, aby mogła ona przekazać do kursora powiadomienie o jakiejkolwiek zmianie danych wskazywanych przez identyfikator URI (**content://contacts/people**).

Interesujący jest fakt, że śledzonym identyfikatorem URI nie jest identyfikator naszego dostawcy treści **MyContactsProvider**, lecz identyfikator dostawcy treści kontaktów dostarczony przez Androida. Wynika to z faktu, że dostawca **MyContactsProvider** stanowi jedynie osłonę „prawdziwego” dostawcy treści. Zatem kurSOR ten musi śledzić właściwego dostawcę treści, a nie jego osłonę.

Ważne jest również, aby osłonić obiekt **MatrixCursor** we własnym kurSORZE, co zostało pokazane na listingu 21.6.

#### **Listing 21.6.** Osłanianie kursora

---

```
MatrixCursor mc = loadNewData(this);  
mc.setNotificationUri(getContext().getContentResolver(),  
    Uri.parse("content://contacts/people/"));  
MyCursor wmc = new MyCursor(mc, this);
```

---

Aby zrozumieć sens osłaniania kursora, musimy dowiedzieć się, w jaki sposób widoki przeprowadzają aktualizację zmienionej treści. Taki dostawca treści, jak `Contacts`, zazwyczaj rejestruje identyfikator URI jako część implementacji metody `query` i w ten sposób powiadamia kursor o potrzebie śledzenia zmian. Do tego służy metoda `cursor.setNotificationUri`. Kursor może następnie zarejestrować ten identyfikator URI oraz jego wszystkie podzielne identyfikatory wraz z dostawcą treści. Podczas przeprowadzenia na dostawcy treści operacji wstawienia lub usunięcia danych kod obsługujący te operacje musi wprowadzić zdarzenie oznaczające zmianę danych w krotkach definiowanych przez określony identyfikator URI.

W ten sposób kursor będzie aktualizowany za pomocą operacji `requery`, a widok zostanie stosownie odświeżony. Niestety, klasa `MatrixCursor` nie jest dostosowana do operacji `requery`. Obsługuje ją kursor `SQLiteDatabase`, jednak nie możemy z niego tutaj skorzystać, ponieważ odwzorowujemy kolumny zgodnie z nowym zestawem kolumn.

Aby pominać to ograniczenie, umieściliśmy obiekt `MatrixCursor` w osłonie kursora i przesłoniliśmy metodę `requery` w celu pozostawienia tego obiektu i utworzenia nowego, zawierającego zaktualizowane dane. Chcemy również, żeby przy każdej zmianie danych był generowany nowy obiekt `MatrixCursor`. Jednak do struktury aktywnego folderu w Androidzie zwracamy jedynie zewnętrzny kursor osłaniający. Szkielet aktywnego folderu będzie rozpoznawał tylko jeden kursor, w jego wnętrzu jednak będą pojawiały się nowe kursory w miarę wprowadzania zmian w danych.

Do tego służą dwie następne klasy.

## MyCursor.java

Zauważmy, w jaki sposób jest inicjalizowany obiekt `MyCursor` zawierający na początku klasę `MatrixCursor` (listing 21.7). Podczas przeprowadzania operacji `requery` kursor `MyCursor` będzie zwrotnie wywoływał dostawcę w celu przekazania obiektu `MatrixCursor`. Nowy obiekt `MatrixCursor` zastąpi stary za pomocą metody `set`.

**Listing 21.7.** Kod źródłowy klasy MyCursor

```
public class MyCursor extends BetterCursorWrapper
{
    private ContentProvider mcp = null;

    public MyCursor(MatrixCursor mc, ContentProvider inCp)
    {
        super(mc);
        mcp = inCp;
    }

    public boolean requery()
    {
        MatrixCursor mc = MyContactsProvider.loadNewData(mcp);
        this.setInternalCursor(mc);
        return super.requery();
    }
}
```

**Uwaga!**

Moglibyśmy tego dokonać, przesłaniając metodę `requery` klasy `MatrixCursor`, klasa ta nie może jednak w żaden sposób wyczyścić danych i rozpoczęć działania od początku. Jest to więc rozsądne obejście (zwróćmy uwagę, że klasa `MyCursor` rozszerza klasę `BetterCursorWrapper`, co zostanie omówione w dalszej części rozdziału).

Przyjrzymy się teraz klasie `BetterCursorWrapper`, aby poznać technikę osłaniania kurSORA.

### **BetterCursorWrapper.java**

Klasa `BetterCursorWrapper` (listing 21.8) przypomina klasę `CursorWrapper` struktury bazodanowej w Androidzie. Potrzebne są jednak dwa elementy, których brakuje klasie `CursorWrapper`. Po pierwsze, nie zawiera ona metody `set`, służącej do zastąpienia wewnętrznego kurSORA, pochodzącego z metody `requery`. Po drugie, obiekt `CursorWrapper` nie jest częścią klasy `CrossProcessCursor`. Aktywne foldery wymagają klasy `CrossProcessCursor`, a nie zwykłego kurSORA, ponieważ przekraczają one granice procesów.

#### **Listing 21.8. Kod źródłowy klasy BetterCursorWrapper**

---

```
public class BetterCursorWrapper implements CrossProcessCursor
{
    //Przechowuje wewnętrzny kurSOR służący do delegowania metod
    protected CrossProcessCursor internalCursor;

    //Konstruktor pobiera obiekt CrossProcessCursor w postaci danych wejściowych
    public BetterCursorWrapper(CrossProcessCursor inCursor)
    {
        this.setInternalCursor(inCursor);
    }

    //Możemy zresetować w jednej z metod klasy pochodnej
    public void setInternalCursor(CrossProcessCursor inCursor)
    {
        internalCursor = inCursor;
    }

    //Tu znajdują się wszystkie delegowane metody
    public void fillWindow(int arg0, CursorWindow arg1) {
        internalCursor.fillWindow(arg0, arg1);
    }
    // ...inne delegowane metody
}
```

---

Na listingu 21.8 nie pokazaliśmy całej klasy `BetterCursorWrapper`, można ją jednak łatwo wygenerować w środowisku Eclipse. Po wczytaniu powyższego fragmentu umiesczamy kurSOR w zmiennej `internalCursor`. Klikamy prawym przyciskiem myszy i wybieramy opcję `Source/Generate Delegated Methods`. W ten sposób zostanie zapelniona reszta klasy. Po wygenerowaniu delegowanych klas przez środowisko Eclipse musimy je oddelegować do wewnętrznej klasy kurSORA, tak jak to zrobiliśmy w przypadku metody `fillWindow` z listingu 21.8 (jeżeli nie chcemy przeprowadzać tego procesu, odpowiedni plik znajdziemy w pliku ZIP zawierającym gotowy projekt).

Posiadamy teraz wszystkie klasy niezbędne do zbudowania, wdrożenia i uruchomienia przykładowego projektu demonstrującego działanie aktywnych folderów w środowisku Eclipse. Ponieważ żadna aktywność nie została zarejestrowana w kategorii *MAIN*, nie ujrzymy interfejsu użytkownika po wdrożeniu projektu, ale w konsoli środowiska Eclipse pojawi się informacja o jego instalacji zakończonej sukcesem.

Podsumujmy ten podrozdział omówieniem zjawisk zachodzących podczas uzyskiwania dostępu do aktywnego folderu.

## Testowanie aktywnych folderów

Po przygotowaniu wszystkich plików projektu aktywnych folderów możemy je skompilować i wdrożyć na emulatorze. Jesteśmy teraz gotowi do wykorzystania utworzonego przez nas aktywnego folderu.

Przejdzmy do ekranu startowego urządzenia, powinien on przypominać zrzut z rysunku 21.1. Przeprowadźmy czynności wypunktowane na początku podrozdziału, w punkcie „W jaki sposób użytkownik korzysta z aktywnych folderów”. Zlokalizujmy zwłaszcza nasz aktywny folder i utwórzmy jego ikonę, tak jak pokazano na rysunku 21.4. Kliknijmy ikonę *Kontakty AF*, a ujrzymy listę zapełnioną kontaktami, podobnie jak na rysunku 21.5.

## Instrukcje dotyczące kompilowania kodu

Najlepszym rozwiązaniem pozwalającym na manipulowanie kodem omówionym w tym rozdziale jest pobranie pliku ZIP, utworzonego specjalnie na potrzeby rozdziału. Adres URL do tego pliku znajdziemy w podrozdziale „Odnośniki”. W pliku tym znajdziemy wszystkie klasy, jakie zostały tutaj omówione.

W przeciwieństwie do wielu innych projektów opisywanych w książce, ten nie posiada aktywności, która zostanie uruchomiona po włączeniu emulatora; jednak komunikaty w konsoli środowiska Eclipse poinformują nas o zakończonej sukcesem instalacji pakietów.

## Odnośniki

Poniższe adresy mogą się okazać bardzo przydatne podczas nauki korzystania z aktywnych folderów oraz pracy z nimi:

- <http://developer.android.com/reference/android/provider/LiveFolders.html> — ten adres umożliwia zapoznanie się z dokumentacją klasy `LiveFolders`.
- <http://developer.android.com/resources/articles/contacts.html> — w tym artykule znajdziemy informacje dotyczące korzystania z interfejsu kontaktów. Okaże się on szczególnie przydatny podczas pracy z aktywnymi folderami wykorzystującymi kontakty.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — pod tym adresem znajdują się projekty utworzone na potrzeby niniejszej książki. Właściwy plik został umieszczony w katalogu o nazwie *ProAndroid3\_R21\_AktywneFoldery*.

## Podsumowanie

Dzięki aktywnym folderom otrzymujemy innowacyjny, obsługiwany jednym kliknięciem mechanizm, wyświetlający zmienione dane na ekranie startowym. Potencjalnie możemy umieszczać tu dane dowolnego rodzaju — pod warunkiem że istnieje możliwość ich zdefiniowania w postaci listy tworzonej przez wiersze. Wszystkie dane muszą posiadać właściwości nazwy i opisu, pozwalające na ich zidentyfikowanie. Niemal każdy typ danych spełnia ten wymóg, gdyż mogą one zostać w jakiś sposób nazwane i opisane. Przydatna okazuje się także obecność aktywności wyświetlającej szczegółowe informacje na temat klikniętego obiektu w aktywnym folderze. Dane mogą być lokalne, na przykład kontakty, a nawet sieciowe, czego przykładem może być spis blogów.

Rozdział ten zawiera opis nowości w kurSORach aktywnych folderów oraz mechanizmów wymaganych do wyeksponowania istniejących dostawców treści jako źródeł aktywnych folderów. Wyjaśniliśmy przyczyny osłaniania kurSORów, a także zaprezentowaliśmy sposób rejestracji klasy `ContentResolver` w celu otrzymywania aktualizacji danych.

Następny rozdział został poświęcony kolejnej innowacji ekranu startowego, znanej pod nazwą widżetów ekranu startowego (ang. *home screen widgets*).

# Widżety ekranu startowego

W niniejszym rozdziale szczegółowo omówimy zagadnienie widżetów ekranu startowego. Podobnie jak przedstawione w rozdziale 21. aktywne foldery, tak i widżety ekranu startowego stanowią kolejny sposób prezentowania często aktualizowanych danych na stronie startowej urządzenia pracującego pod kontrolą systemu Android. Ogólnie rzecz ujmując, widżety ekranu startowego są odłączonymi widokami (chociaż wypełnionymi danymi), wyświetlanymi na ekranie startowym. Dane znajdujące się w tych widokach są aktualizowane w regularnych odstępach czasu przez procesy zachodzące w tle.

Na przykład widżet poczty elektronicznej może informować użytkownika o liczbie nieprzeczytanych wiadomości, z podkreśleniem, że widżet może przedstawać jedynie liczbę tych wiadomości, a nie ich treść. Kliknięcie licznika wiadomości może uruchomić aktywność wyświetlającą treść wiadomości. Mogą to być nawet zewnętrzne źródła poczty e-mail, na przykład Yahoo, Gmail lub Hotmail, dopóki urządzenie posiada możliwość połączenia się z serwerem poczty za pomocą protokołu HTTP albo innego mechanizmu sieciowego.

**Uwaga!**

W wersji 3.0 Androida widżety ekranu startowego uzyskały nową funkcjonalność. Te dodatkowe mechanizmy zostały omówione w rozdziale 31.

Rozdział podzielimy na trzy części. W pierwszym podrozdziale omówimy pojęcie widżetów ekranu startowego oraz ich architekturę. Wyjaśnimy, w jaki sposób Android wykorzystuje widok RemoteViews do wyświetlania widżetów oraz jak dołącza odbiorców komunikatów, służących do aktualizowania tych widoków. Pokażemy technikę tworzenia aktywności umożliwiających konfigurowanie widżetów na ekranie startowym oraz zaobserwujemy związek pomiędzy usługami a widżetami. Po przeczytaniu tego podrozdziału Czytelnik będzie dobrze rozumiał architekturę oraz cykl życia widżetów ekranu startowego.

W drugim podrozdziale zademonstrujemy sposób projektowania i rozwijania widżetów ekranu startowego — jak zwykle posłużymy się przykładami kodu z komentarzem. Czytelnicy dowiedzą się, jak można definiować widżety w Androidzie oraz jak pisać kod tworzący odbiorców komunikatów, które będą aktualizować te widżety. Pokażemy metody zarządzania stanem widżetu za pomocą współdzielonych preferencji oraz przedstawimy kod aktywności służącej do konfigurowania widżetów.

W podrozdziale trzecim zajmiemy się kwestiami przydatności i ograniczeń, damy też ogólne wskazówki ułatwiające pracę z widżetami. Dodatkowo omówimy zakres i stosowalność widżetów. Przedstawimy również porady dotyczące pisania widżetów wymagających bardzo częstych aktualizacji.

Rozdział zakończymy listą zasobów dotyczących programowania widżetów w Androidzie.

## Architektura widżetów ekranu startowego

Omówienie architektury widżetów ekranu startowego rozpoczęmy od ustalenia ich dokładnej definicji.

### Czym są widżety ekranu startowego?

Widżety ekranu startowego są często aktualizowanymi widokami, wyświetlonymi na ekranie startowym. Skoro widżet strony startowej jest widokiem, jego wygląd i działanie są definiowane w pliku XML układu graficznego. Poza tym układem graficznym będziemy musieli jeszcze zdefiniować ilość zajmowanej przez widżet przestrzeni na ekranie.

W definicji widżetu zawarto również kilka klas Java, zapewniającychinicjalizację widoków i jego częste aktualizacje. Klasy te zarządzają cyklem życia widżetu na ekranie startowym. Reagują one na procesy przeciągania widżetu na ekran startowy oraz jego usuwania do kosza.

#### Uwaga!

Widok oraz odpowiadająca mu klasa Java są zaprojektowane w taki sposób, że obydwa obiekty są od siebie oddzielone. Na przykład każda usługa lub aktywność w Androidzie może odczytać widok za pomocą identyfikatora jego układu graficznego, wypełnić go danymi (podobnie jak w przypadku wypełniania szablonu), a następnie wysłać go na ekran startowy. Po przesłaniu widżetu na ekran startowy zostaje on oddzielony od obsługującego go kodu Java.

Najprostsza definicja widżetu zawiera następujące elementy:

- Układ graficzny widoku wyświetlany na ekranie startowym, a także określony rozmiar jego dopasowania na stronie startowej. Pamiętajmy, że jest to jedynie widok bez żadnych wstawionych danych. Zadaniem klasy Java będzie jego aktualizacja.
- Zegar określający częstotliwość aktualizacji.
- Klasa Java nazywana **dostawcą widżetu** (ang. *widget provider*), reagująca na aktualizacje zegara w celu zmiany widoku w określony sposób, umożliwiający zapelnienie go danymi.

Po zdefiniowaniu widżetu oraz wprowadzeniu klas Java będzie on zdatny do użytku. Najpierw jednak omówimy przykład widżetu ekranu startowego, wykorzystywanego w praktyce.

### W jaki sposób użytkownik korzysta z widżetów ekranu startowego?

Jedną z funkcjonalności widżetu ekranu startowego w Androidzie jest możliwość umieszczenia utworzonego fabrycznie widżetu na ekranie startowym. Po umieszczeniu na ekranie startowym można go w razie konieczności skonfigurować za pomocą aktywności (zdefiniowanej jako część pakietu widżetu). Bardzo istotne jest zrozumienie tej interakcji przed właściwym zagłębieniem się w szczegóły kodu widżetu.

Innymi słowy, przed nauką programowania widżetu należy się przekonać, w jaki sposób się go użytkuje.

W naszym przykładzie zajmiemy się widżetem noszącym nazwę *Urodziny*, utworzonym specjalnie na potrzeby tego rozdziału. W dalszej części rozdziału zaprezentujemy jego kod źródłowy. Najpierw posłuży nam on do wyjaśnienia zasady działania widżetów. Kod źródłowy tego widżetu zamieściliśmy w dalszej części rozdziału, zatem teraz zalecamy Czytelnikowi skoncentrowanie się na treści rozdziału i oglądaniu rzutów ekranu, a dopiero później na samodzielnym testowaniu widżetu. Jeżeli Czytelnik przeanalizuje rysunki i dołączone wyjaśnienia, nie powinien mieć problemów ze zrozumieniem natury i zachowania widżetu *Urodziny*, dzięki czemu opanowanie kodu źródłowego stanie się łatwiejszym zadaniem.

Rozpoczniemy od zlokalizowania poszukiwanego widżetu i utworzenia jego instancji na ekranie startowym.

## Utworzenie instancji widżetu na ekranie startowym

Aby uzyskać dostęp do listy dostępnych widżetów, należy — poprzez długie kliknięcie na ekranie startowym — wywołać menu kontekstowe strony startowej, przedstawione na rysunku 22.1.



**Rysunek 22.1.** Menu kontekstowe ekranu startowego

Po wybraniu opcji *Widżety* pojawi się kolejny ekran, zawierający listę dostępnych widżetów, co zostało przedstawione na rysunku 22.2.

Większość widżetów stanowi integralną część Androida. Lista dostępnych widżetów może wyglądać inaczej w zależności od wersji używanego oprogramowania. W celach demonstracyjnych wybraliśmy widok *Birthday Widget*. Po jego kliknięciu zostanie utworzona odpowiednia instancja widżetu na ekranie startowym, wyglądająca jak przykładowy widżet *Urodziny* z rysunku 22.3.

W nagłówku widżetu *Urodziny* są wyświetlane takie dane, jak imię osoby, liczba dni do jej urodzin, a także data urodzin oraz łącze do sklepu z upominkami.



Rysunek 22.2. Lista widżetów ekranu startowego



Rysunek 22.3. Przykładowy widżet

Możemy się zastanawiać, w jaki sposób zostały skonfigurowane dane jubilata. Co należałoby zrobić w przypadku, gdyby potrzebne były dwa wystąpienia widżetu, prezentujące informacje o dwóch różnych osobach? Do tego służy aktywność konfiguratora widżetów, która jest tematem następnego podpunktu.

**Uwaga!**

Widok utworzony na ekranie startowym dla tej definicji widżetu nosi nazwę **instancji widżetu**. Wynika z tego wniosek, że można utworzyć więcej definicji instancji tego widżetu.

## Konfigurator widżetów

Definicja widżetu może opcjonalnie zawierać specyfikację aktywności, zwanej aktywnością konfiguradora widżetów. Po wybraniu widżetu z listy dostępnych widżetów w celu utworzeniu jego instancji Android wywołuje powiązaną z nim aktywność konfiguracji widżetu. Pokażemy, w jaki sposób samodzielnie napisać taką aktywność. Umożliwi ona konfigurację wystąpienia widżetu.

W przypadku naszego widżetu urodzinowego aktywność konfiguracji wyświetli monit o wprowadzenie imienia jubilata oraz daty jego urodzin, tak jak zostało pokazane na rysunku 22.4. Zadaniem konfiguratora jest zapisanie tych informacji w stałym miejscu, aby po wywołaniu aktualizacji przez dostawcę widżetu dostawca ten mógł zlokalizować informacje i zaktualizować je o nowe wartości, wstawiane następnie do widżetu przez konfigurator.



Rysunek 22.4. Aktywność konfiguratora widżetu

### Uwaga!

Jeżeli użytkownik wybierze utworzenie dwóch instancji widżetu urodzinowego na stronie startowej, aktywność konfiguratora zostanie wywołana dwukrotnie (jednorazowo dla każdego wystąpienia widżetu).

Android wewnętrznie śledzi instancje widżetów poprzez przypisywanie im identyfikatora. Identyfikator ten jest przekazywany metodom zwrotnym kodu Java oraz klasie konfiguracyjnej w celu skierowania początkowej konfiguracji i aktualizacji do właściwej instancji. Na rysunku 22.3, w ciągu znaków `satya:3`, cyfra 3 stanowi identyfikator widżetu — a ściślej, identyfikator instancji widżetu. Sam zaś widżet jest identyfikowany za pomocą nazwy (na którą składa się nazwa klasy oraz pakietu, w którym ta klasa się znajduje); w tym rozdziale pojęcia „identyfikator widżetu” oraz „identyfikator instancji widżetu” są używane zamiennie i odnoszą się do identyfikatora wystąpienia widżetu. Identyfikator instancji widżetu został pokazany na rysunku 22.3 w celach demonstracyjnych.

Po ogólnym omówieniu widżetu nadszedł czas na szczegółowe zapoznanie się z jego cyklem życia.

## Cykl życia widżetu

Wspomnieliśmy kilkakrotnie o definicji widżetu. Poruszyliśmy także temat roli klas Java. W tym punkcie poświęcimy o wiele więcej uwagi obydwu zagadnieniom oraz prześledzimy cykl życia widżetu.

Cykl ten składa się z następujących faz:

1. Definiowanie widżetu.
2. Tworzenie instancji widżetu.
3. Zastosowanie metody `onUpdate()` (po wygaśnięciu interwału czasowego).
4. Odpowiedź na kliknięcia (w widoku widżetu na ekranie startowym).
5. Usunięcie widżetu (z ekranu startowego).
6. Odinstalowanie.

Omówimy teraz szczegółowo każdy z wymienionych etapów.

### Faza definiowania widżetu

Cykl życia widżetu rozpoczyna się od zdefiniowania widoku widżetu. W definicji tej określamy nazwę widżetu wyświetlaną na liście dostępnych widżetów (rysunek 22.2), wywoływanej z poziomu ekranu startowego. Do utworzenia definicji wymagane są dwa elementy: klasa Java implementująca dostawcę `AppWidgetProvider` oraz układ graficzny widżetu.

Rozpoczniemy definiowanie widżetu od następującego wpisu w pliku manifeście, definiującego dostawcę `AppWidgetProvider` (listing 22.1).

---

**Listing 22.1.** Definicja widżetu w pliku manifeście Androida

---

```
<manifest..>
<application>
...
<receiver android:name=".BDayWidgetProvider">
    <meta-data android:name="android.appwidget.provider"
              android:resource="@xml/bday_appwidget_provider" />
    <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
    </intent-filter>
</receiver>
...
<activity>
.....
</activity>
<application>
</manifest>
```

---

Definicja ta wskazuje istnienie odbiorcy komunikatów klasy Java noszącego nazwę `BdayWidgetProvider` (jak się przekonamy, wywodzi się on z podstawowej klasy Androida `AppWidgetProvider` mieszącej się w pakiecie `widget`), który odbiera komunikaty zawierające aktualizacje widżetu.

**Uwaga!** Android dostarcza informacje o aktualizacji w formie komunikatów rozgłoszeniowych, generowanych na podstawie częstości interwałów czasowych.

Definicja widżetu z listingu 22.1 jest również związana z plikiem XML w katalogu */res/xml*, który z kolei określa widok widżetu oraz częstotliwość odświeżania, co zostało zaprezentowane na listingu 22.2.

#### **Listing 22.2.** Definicja widoku widżetu w pliku XML informacji o dostawcy widżetu

---

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="150dp"
    android:minHeight="120dp"
    android:updatePeriodMillis="43200000"
    android:initialLayout="@layout/bday_widget"
    android:configure="com.ai.android.BDayWidget.ConfigureBDayWidgetActivity"
    >
</appwidget-provider>
```

---

Plik ten jest nazywany plikiem informacji o dostawcy widżetu. Zostaje on wewnętrznie przetłumaczony na klasę Java *AppWidgetProviderInfo*. Wartości szerokości i wysokości układu graficznego zostają tu ustalone odpowiednio na 150dp i 120dp. Zostaje tu również określony wyrażony w milisekundach czas przedziału czasowego równy 12 godzinom. Definicja ta wskazuje także plik układu graficznego (listing 22.7), opisujący widok widżetu (rysunek 22.5).

Zauważmy jednak, że układ graficzny tych widoków widżetów może zawierać jedynie niektóre rodzaje elementów widoku. Dopuszczalne kontrolki w układzie graficznym widżetu należą wyłącznie do klasy widoków znanej jako *RemoteViews* — ta klasa widoków zdalnych akceptuje jedynie określone rodzaje widoków potomnych. Te dopuszczone podeszczalne elementy widoku są wypisane na listingu 22.3.

#### **Listing 22.3.** Dopuszczalne kontrolki widoku w klasie *RemoteViews*

---

```
FrameLayout
LinearLayout
RelativeLayout

AnalogClock
Button
Chronometer
ImageButton
ImageView
ProgressBar
TextView
```

---

Powyższa lista może w przyszłości rozrastać się z każdą kolejną wersją środowiska SDK. Zasadniczym powodem ograniczenia dopuszczalnych typów elementów w widoku zdalnym jest fakt, że są one odcięte od kontrolujących je procesów. Te widoki widżetów są obsługiwane przez takie aplikacje, jak na przykład Home. Kontrolerami tych widoków są przetwarzane w tle procesy, wywoływane przez zegary. Z tego powodu obiekty te noszą nazwę widoków zdalnych. Istnieje odpowiednia klasa Java, nazwana *RemoteViews*, udzielająca dostępu do tych widoków. Innymi słowy, programiści nie muszą uzyskiwać bezpośredniego dostępu do tych widoków, aby wywołać wobec nich metody. Dostęp do nich następuje wyłącznie poprzez klasę *RemoteViews* (pełni rolę bramki).

Metody klasy `RemoteViews` zostaną omówione w następnym podrozdziale, podczas prezentowania przykładowego widżetu. Na razie wystarczy pamiętać, że w pliku układu graficznego widżetu możemy umieścić ograniczony zestaw widoków (listing 22.3).

Definicja widżetu (listing 22.2) zawiera również specyfikację aktywności konfiguracyjnej, która musi zostać wywołana podczas tworzenia instancji widżetu przez użytkownika. Ta aktywność na listingu 22.2 nosi nazwę `ConfigureBDayWidgetActivity`. Jest to standardowa aktywność zawierająca kilka pól formularza. Pola te mają na celu uzyskanie od użytkownika informacji wymaganych przez instancję widżetu.

## Faza tworzenia instancji widżetu

Po utworzeniu wszystkich elementów XML wymaganych przez definicję widżetu oraz udogodnieniu wszystkich klas Java widżetów możemy się przyjrzeć, co się stanie po wybraniu przez użytkownika nazwy widżetu z listy dostępnych widżetów (rysunek 22.2). Android wywoła aktywność konfiguratora (rysunek 22.3), która przeprowadza następujące czynności:

1. Odebranie identyfikatora instancji widżetu od intencji wywołującej, która uruchomiła konfigurator.
2. Zebranie informacji potrzebnych instancji widżetu za pomocą formularza wyświetlanego użytkownikowi.
3. Zachowanie uzyskanych przez widżet informacji. Dostęp do nich będzie potrzebny podczas wywołania metody `update`.
4. Przygotowanie do wyświetlania widoku widżetu po raz pierwszy poprzez odczytanie układu graficznego tego widoku i utworzenie obiektu `RemoteViews`.
5. Wywołanie metod klasy `RemoteViews` ustanawiających wartości pojedynczych obiektów widoku, takich jak tekst, obraz i tak dalej.
6. Wykorzystanie obiektu `RemoteViews` do zarejestrowania wszelkich zdarzeń `onClick` wobec dowolnego podelementu widżetu.
7. Wywołanie klasy `AppWidgetManager` w celu narysowania obiektu klasy `RemoteViews` na stronie startowej z wykorzystaniem identyfikatora instancji tego widżetu.
8. Powrót do identyfikatora widżetu i zakończenie działania.

Zwrócić uwagę, że w tym przypadku pierwsze rysowanie jest wykonane przez konfigurator, a nie przez metodę `onUpdate()` klasy `AppWidgetProvider`.

### Uwaga!

Aktywność konfiguratora jest elementem dodatkowym. Jeżeli nie zostanie ona zdefiniowana, wywołanie przejdzie bezpośrednio do metody `onUpdate()` klasy `AppWidgetProvider`. Metoda `onUpdate()` służy do aktualizowania widoku.

Android będzie powtarzał ten proces dla każdej instancji widżetu utworzonej przez użytkownika. Warto również zauważyć, że nie istnieją udokumentowane ograniczenia zmuszające użytkownika do korzystania z tylko jednej instancji widżetu.

Poza przywoływaniem aktywności konfiguratora Android wywołuje zwrotnie również metodę `onEnabled` klasy `AppWidgetProvider`. Poświęćmy chwilę metodom zwrotnym klasy `AppWidgetProvider` i przyjrzymy się powloce naszego dostawcy `BDayWidgetProvider` (listing 22.4). Pełny kod tego pliku został umieszczony na listingu 22.9.

**Listing 22.4.** Powłoka dostawcy widżetu

---

```
public class BDayWidgetProvider extends AppWidgetProvider
{
    public void onUpdate(Context context,
                        AppWidgetManager appWidgetManager,
                        int[] appWidgetIds){}
    public void onDeleted(Context context, int[] appWidgetIds){}
    public void onEnabled(Context context){}
    public void onDisabled(Context context) {}
}
```

---

Metoda zwrotna `onEnabled()` wskazuje, że istnieje co najmniej jedna instancja widżetu działająca na ekranie startowym. Oznacza to, że użytkownik musiał umieścić na stronie startowej przynajmniej jeden widżet. A zatem w wywołaniu musimy uruchomić otrzymywanie komunikatów dla tego składnika (z listingu 22.9 dowiemy się, jak tego dokonać). W Androidzie klasy są nazywane czasami składnikami, szczególnie gdy tworzą wielokrotnie wykorzystywane jednostki, takie jak aktywność, usługa lub odbiorca transmisji. W naszym przypadku klasa `AppWidgetProvider` jest składnikiem odbiorcy transmisji; możemy ją włączać lub wyłączać w celu otrzymywania transmitowanych komunikatów.

Metoda zwrotna `onDeleted()` jest wywoływana podczas przenoszenia przez użytkownika instancji widżetu do kosza. To właśnie tu musimy usunąć wszystkie przechowywane wartości dla instancji widżetu.

Metoda zwrotna `onDisabled()` jest wywoływana po usunięciu ostatniej instancji widżetu z ekranu startowego. Następuje to w momencie przeniesienia ostatniej instancji do kosza. Powinniśmy używać tej metody do wyrejestrowania procesu otrzymywania transmitowanych komunikatów przez ten składnik (zobaczmy to na listingu 22.9).

Metoda zwrotna `onUpdate()` jest wywoływana za każdym razem, gdy wygaśnie zegar zaprezentowany na listingu 22.2. Metoda ta jest również wywoływana na samym początku, podczas generowania instancji widżetu, w przypadku gdy nie zdefiniowaliśmy aktywności konfiguradora. Jeżeli aktywność konfiguradora jest dostępna, ta metoda nie jest wywoływana podczas procesu utworzenia instancji widżetu. Następnie będzie ona wywoływana z częstotliwością równą wygaśnięciom zegara.

## Faza metody `onUpdate`

Następnym ważnym zdarzeniem po wyświetleniu instancji widżetu na ekranie startowym jest wygaśnięcie zegara. Jak już wspomnieliśmy, zostaje wtedy wywołana metoda `onUpdate()`. Do jej wywołania służy odbiorca komunikatów. Oznacza to, że zostanie wczytany właściwy proces Java, w którym jest zdefiniowana metoda `onUpdate()`, i będzie on trwał aż do chwili zakończenia wywołania. Po zwrocie wywołania proces ten będzie gotowy do zamknięcia.

Zalecane jest także wprowadzenie mechanizmu wykorzystującego na przykład długoterminowych odbiorców komunikatów (rozdział 14.), jeżeli przetwarzanie odpowiedzi dłuższych od dziesięciu sekund ma zostać zakończone sukcesem. W przeciwnym razie zostanie wyświetlony komunikat o błędzie ANR (Android nie odpowiada).

W każdym razie po otrzymaniu danych niezbędnych do zaktualizowania widżetu za pomocą metody `onUpdate()` możemy przywołać klasę `AppWidgetManager`, aby narysować zdalny widok. Gdybyśmy do przeprowadzenia aktualizacji wykorzystali długoterminową usługę, musielibyśmy przekazać intencji uruchamiającej tę usługę identyfikator widżetu w postaci dodatkowych danych.

Chcemy w ten sposób pokazać, że klasa `AppWidgetProvider` jest bezstanowa i być może nie będzie nawet zdolna do utrzymania zmiennych statycznych pomiędzy wywołaniami. Powodem jest możliwość zamknięcia procesu Java zawierającego tę klasę odbiorcy komunikatów oraz jego rekonstrukcji pomiędzy dwoma wywołaniami, co powoduje ponowną inicjalizację zmiennych statycznych.

W wyniku tego musimy w razie potrzeby wprowadzić schemat zapamiętywania stanów. Jeżeli aktualizacje nie są zbyt częste — powiedzmy, co kilka sekund — sensownym rozwiązaniem jest zapisywanie stanu instancji widżetu w trwałym magazynie, na przykład pliku, we współdzielonych preferencjach lub w bazie danych SQLite. W następnym przykładzie wykorzystamy do tego celu współdzielone preferencje.

#### Ostrzeżenie

Aby zaoszczędzić energię, bardzo zalecane jest ustalenie interwałów aktualizacji dłuższych niż godzina, dzięki czemu urządzenie nie będzie zbyt często wychodziło ze stanu wstrzymania. W dokumentacji Androida widnieje również ostrzeżenie, że w przyszłych wersjach oprogramowania może zostać wprowadzone ustawienie minimalnego interwału równego 30 minutom lub dłuższego.

W przypadku krótszych przedziałów czasowych, rzędu pojedynczych sekund, musimy samodzielnie wywołać metodę `onUpdate()` za pomocą funkcji dostępnych w klasie `AlarmManager`. Jeżeli korzystamy z klasy `AlarmManager`, zamiast wywoływaniem metody `onUpdate()` można się posłużyć metodami zwrotnymi alarmu. Stosowanie menedżera alarmu zostało omówione w rozdziale 15.

Poniżej wymieniliśmy standardowe czynności wymagane podczas pracy z metodą `onUpdate()`:

1. Upewnij się, że konfigurator zakończył pracę; jeśli nie skończył, po prostu sprawdź to ponownie po chwili. Nie powinno to stanowić problemu w wersji oprogramowania co najmniej 2.0, gdzie są oczekiwane dłuższe interwały czasowe. Jeżeli będzie inaczej, istnieje możliwość, że metoda `onUpdate()` zostanie wywołana przed zakończeniem procesu konfiguracji widżetu przez użytkownika w konfiguratorze.
2. Pobierz dane przechowywane dla tej instancji widżetu.
3. Pobierz układ graficzny widoku widżetu i utwórz wraz z nim obiekt `RemoteViews`.
4. Wywołaj metody klasy `RemoteViews` w celu ustawienia wartości dla pojedynczych obiektów widoku, na przykład dla tekstu, obrazu i tak dalej.
5. Zarejestruj wszelkie zdarzenia `onClick` w dowolnym widoku poprzez wykorzystanie oczekujących intencji.
6. Zaprogramuj klasę `AppWidgetManager`, aby narysowała obiekt `RemoteViews`, korzystając z identyfikatora instancji.

Jak widać, istnieje duża zbieżność pomiędzy działaniem konfiguratora a działaniem metody `onUpdate()`. Istnieje możliwość wykorzystywania tej zbieżności.

## Metody zwrotnych zdarzeń generowanych przez kliknięcie widoku widżetu

Ustaliliśmy już, że metoda `onUpdate()` na bieżąco aktualizuje widoki widżetu. Widok widżetu oraz jego podelementy mogą posiadać metody zwrotne rejestrowania kliknięć przycisku myszy. Zazwyczaj do zarejestrowania działania takiego jak kliknięcie służy przetwarzana intencja. Działanie to może następnie uruchomić usługę lub wykonać konkretną czynność, na przykład otwarcie okna przeglądarki.

Taka wywołana usługa lub aktywność może następnie nawiązać w razie potrzeby komunikację z widokiem za pomocą identyfikatora instancji widżetu i klasy `AppWidgetManager`. Ważne jest zatem, aby przetwarzana intencja zawierała identyfikator instancji widżetu.

## Usunięcie instancji widżetu

Kolejnym elementem cyklu życia instancji widżetu jest jej usunięcie. Aby tego dokonać, użytkownik musi kliknąć widżet ekranu startowego. U spodu ekranu zostanie wyświetlony kosz. Można do niego przenieśćinstancję widżetu w celu jej usunięcia z ekranu.

Powoduje to również wywołanie metody `onDeleted()` wobec dostawcy widżetu. Jeżeli zachowalibyśmy dla tej instancji jakieś informacje o stanie, musimy usunąć te dane w metodzie `onDeleted`.

Android wywołuje także metodę `onDisabled()` w przypadku usunięcia ostatniej instancji danego typu. Ta metoda zwrotna służy do czyszczenia wszystkich atrybutów przechowywanych dla każdej instancji widżetu oraz do wyrejestrowania metod zwrotnych z transmisji metody `onUpdate()` (listing 22.9).

## Odinstalowanie pakietów widżetów

Przedstawiliśmy pełny cykl życia widżetu. Zanim przejdziemy do następnej części, krótko wspomnijmy o potrzebie uporządkowania widżetów w przypadku planowania odinstalowania oraz zainstalowania nowej wersji pliku `.apk`, zawierającego te widżety.

Zalecane jest usunięcie wszystkich instancji widżetów przed próbą odinstalowania pakietu. Należy postępować zgodnie z instrukcjami opisanymi w podpunkcie „Usunięcie instancji widżetu” aż do usunięcia ostatniej instancji.

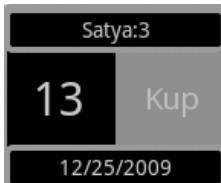
Teraz możemy odinstalować starą wersję i zainstalować nową. Jest to szczególnie istotne w przypadku osób wykorzystujących do tworzenia widżetów wtyczkę Eclipse ADT, ponieważ w trakcie projektowania próbuje ona przeprowadzić ten proces podczas każdego uruchomienia aplikacji. Zatem pomiędzy momentami działania aplikacji należy usunąć instancje widżetu.

## Przykładowy widżet

Do tej pory przedstawiliśmy podstawy teoretyczne, pokazaliśmy też sposoby tworzenia widżetów. Wykorzystajmy tę wiedzę do utworzenia przykładowego widżetu, którego zachowanie było wykorzystane jako przykład podczas omawiania architektury widżetów. Zaprojektujemy, wdrożymy i przetestujemy ten stary-nowy widżet *Urodziny*.

Każda instancja widżetu będzie wyświetlała imię jubilata, datę jego urodzin oraz liczbę dni, które pozostały do tego święta. Zostanie także utworzony obszar `onClick`, którego kliknięcie pozwoli na wizytę w sklepie w celu zakupu upominku. Z tego właśnie względu zostanie uruchomiona przeglądarka wyświetlająca stronę <http://www.google.com>.

Końcowy układ graficzny widżetu został zilustrowany na rysunku 22.5.



Rysunek 22.5. Wygląd i działanie widżetu urodzinowego

Implementacja tego widżetu składa się z wymienionych poniżej plików. W zależności od stosowanego źródłowego pakietu Java pliki Java będą przechowywane w podkatalogu `src` wraz ze strukturą katalogów, którą wykorzystamy w przypadku tych pakietów. W celu zachowania związek te podkatalogi są reprezentowane przez wyrażenie „...”.

- ***AndroidManifest.xml*** // — jest tu zdefiniowany dostawca `AppWidgetProvider` (listing 22.5).
- ***res/xml/bday\_appwidget\_provider.xml*** // — wymiary i układ graficzny widżetu (listing 22.6).
- ***res/layout/bday\_widget.xml*** // — układ graficzny widżetu (listing 22.7).
- ***res/drawable/box1.xml*** // — zapewnia ramki dla sekcji układu graficznego widżetu (listing 22.8).
- ***src/.../BDayWidgetProvider*** // — implementacja klasy `AppWidgetProvider` (listing 22.9).

W implementacji zawarte są również następujące pliki zarządzające stanem widżetu:

- ***src/.../IWidgetModelSaveContract*** // — model kontraktu zachowywania widżetu (listing 22.10).
- ***src/.../APrefWidgetModel*** // — abstrakcyjny model widżetu opartego na preferencjach (listing 22.11).
- ***src/.../BDayWidgetModel*** // — model widżetu przechowującego dane dla jego widoku (listing 22.12).
- ***src/.../Utils.java*** // — kilka klas użytkowych (listing 22.13).

Ponadto w implementacji zostały umieszczone następujące pliki, odpowiedzialne za aktywność konfiguracji:

- ***src/.../ConfigureBDayWidgetActivity.java*** // — aktywność konfiguracji (listing 22.14).
- ***layout/edit\_bday\_widget.xml*** // — układ graficzny do wpisywania imienia jubilata i daty urodzin (listing 22.15).

Omówimy każdy z wymienionych plików oraz wyjaśnimy wszelkie pominięte do tej pory pojęcia. Pod koniec lektury tego podrozdziału Czytelnik zapewne będzie mógł utworzyć te pliki i przetestować widżet urodzinowy we własnym środowisku.

## Definiowanie dostawcy widżetu

Definiowanie widżetu rozpoczyna się w pliku manifeście aplikacji Androida. To właśnie tutaj określamy dostawcę widżetu, aktywność konfiguracji widżetu oraz wskaźnik pliku XML definiującego układ graficzny widżetu.

Wszystkie te elementy zostały zaznaczone pogrubioną czcionką w pliku manifeście, widocznym na listingu 22.5. Zwróćmy uwagę na definicję dostawcy **BDayAppWidgetProvider** pełniącego funkcję odbiorcy komunikatów, a także na definicję aktywności konfiguracji **ConfigureBDayWidgetActivity**.

**Listing 22.5.** Plik manifest przykładowej aplikacji **BDayWidget**

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ai.android.BDayWidget"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="Urodziny">
        <!--
        ****
        * Odbiorca komunikatów dostawcy widżetu urodzinowego
        ****
        -->
        <receiver android:name=".BDayWidgetProvider">
            <meta-data android:name="android.appwidget.provider"
                android:resource="@xml/bday_appwidget_provider" />
            <intent-filter>
                <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
        </receiver>
        <!--
        ****
        * Aktywność konfiguratora widżetu urodzinowego
        ****
        -->
        <activity android:name=".ConfigureBDayWidgetActivity"
            android:label="Konfiguruj widżet Urodziny">
            <intent-filter>
                <action android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="3" />
</manifest>
```

**Uwaga!**

Po przejrzeniu pliku manifestu okazuje się, że odbiorca jest węzłem równorzędnym z węzłem aktywności. Jest on również bezpośrednim potomkiem węzła aplikacji.

Etykieta aplikacji *Urodziny*, widoczna w wierszu:

```
<application android:icon="@drawable/icon" android:label="Urodziny">
```

stanowi nazwę widżetu wyświetlana na liście dostępnych widżetów (rysunek 22.2). Jeżeli tworzymy po raz pierwszy definicję widżetu, upewnijmy się, że poniższy wiersz zostanie dokładnie skopiowany:

```
<meta-data android:name="android.appwidget.provider"
```

Określenie `android.appwidget.provider` jest charakterystyczne dla Androida i powinno pozostać umieszczone w kodzie; to samo dotyczy poniższego fragmentu:

```
<intent-filter>
    <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
</intent-filter>
```

Definicja aktywności konfiguracji nie różni się od standardowej aktywności poza koniecznością zadeklarowania możliwości jej odpowiedzi na działania `APPWIDGET_CONFIGURE`.

## Definiowanie rozmiaru widżetu

Chociaż w pliku manifeście zdefiniowano dostawcę widżetu, szczegóły dotyczące jego układu graficznego umieszczono w osobnym pliku XML. Do dodatkowych informacji zaliczamy rozmiar widżetu, nazwę jego pliku układu graficznego, przedział czasu aktualizacji oraz nazwę składnika (lub klasy) aktywności konfiguracji.

Taki dodatkowy plik XML jest wskazywany przez węzeł `android:resource`, widoczny w uprzednio omówionej definicji dostawcy widżetu (listing 22.5). Na listingu 22.6 została przedstawiona zawartość pliku informacyjnego dostawcy widżetu (`/res/xml/bday_appwidget_provider.xml`).

---

### Listing 22.6. Definicja widoku widżetu BDayWidget

---

```
<!-- res/xml/bday_appwidget_provider.xml -->
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="150dp"
    android:minHeight="120dp"
    android:updatePeriodMillis="4320000"
    android:initialLayout="@layout/bday_widget"
    android:configure="com.ai.android.BDayWidget.ConfigureBDayWidgetActivity"
    >
</appwidget-provider>
```

---

W pliku tym określa się w pikselach szerokość i wysokość widżetu, jednak Android zaokrągli te wymiary do rozmiaru najbliższej wielokrotności komórki. Obszar ekranu startowego jest organizowany w macierz komórek: każda komórka stanowi kwadrat o boku 74 dp (piksele niezależne od gęstości). W dokumentacji Androida można znaleźć zalecenie, aby wymiary dwóch różnych widżetów stanowiły wielokrotność wymiarów tych komórek minus 2 piksele (margines dla zaokrąglenia rogów i tak dalej).

Można tu także znaleźć wartość częstotliwości wywoływania metody `onUpdate()`. Zaleca się, aby ta wartość nie przekraczała kilku razy na dobę. Wprowadzenie wartości 0 oznacza całkowity brak wywoływania aktualizacji. Można ją wykorzystać, w przypadku gdy chcemy sami kontrolować aktualizacje za pomocą klasy `AlarmManager`.

Wartość atrybutu `initialLayout` wskazuje rzeczywisty układ graficzny widżetu (listing 22.7). Natomiast atrybut `configure` określa klasę aktywności konfiguracji. Należy umieścić pełną nazwę tej klasy w tej definicji.

Przyjrzyjmy się teraz właściwemu układowi graficznemu widżetowi.

## Pliki związane z układem graficznym widżetu

Z poprzedniego punktu i z listingu 22.6 wiemy, że układ graficzny widżetu jest skonfigurowany w określonym pliku. Plik ten jest standardowym plikiem układu graficznego widoku w Androidzie.

Jednak w celu ustandaryzowania procesu tworzenia widżetów Android opublikował zestaw wytycznych projektowania. Wytyczne te można znaleźć pod adresem:

[http://developer.android.com/guide/practices/ui\\_guidelines/widget\\_design.html](http://developer.android.com/guide/practices/ui_guidelines/widget_design.html)

Oprócz wytycznych pod tym adresem umieszczono zbiór tel widoków, poprawiających wygląd i działanie widżetów. W naszym przykładzie poszliśmy inną ścieżką i wykorzystaliśmy tradycyjny sposób implementowania układów graficznych, w których tlami są kształty.

### Plik układu graficznego widżetu

Na listingu 22.7 zaprezentowaliśmy treść pliku tworzącego układ graficzny widżetu, ukazany na rysunku 22.5.

**Listing 22.7.** Definicja układu graficznego widoku widżetu BDayWidget

```
<!-- res/layout/bday_widget.xml -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="150dp"
    android:layout_height="120dp"
    android:background="@drawable/box1"
    >
<TextView
    android:id="@+id/bdw_w_name"
    android:layout_width="fill_parent"
    android:layout_height="30dp"
    android:text="Anonim"
    android:background="@drawable/box1"
    android:gravity="center"
    />
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="60dp"
    >
<TextView
    android:id="@+id/bdw_w_days"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:text="0"
    android:gravity="center"
    android:textSize="30sp"
```

```
    android:layout_weight="50"
  />
<TextView
  android:id="@+id/bdw_w_button_buy"
  android:layout_width="wrap_content"
  android:layout_height="fill_parent"
  android:textSize="20sp"
  android:text="Kup"
  android:layout_weight="50"
  android:background="#FF6633"
  android:gravity="center"
/>
</LinearLayout>
<TextView
  android:id="@+id/bdw_w_date"
  android:layout_width="fill_parent"
  android:layout_height="30dp"
  android:text="1/1/2000"
  android:background="@drawable/box1"
  android:gravity="center"
/>
</LinearLayout>
```

---

Aby osiągnąć zamierzony efekt, układ graficzny wykorzystuje zagnieźdzone węzły `LinearLayout`. Niektóre kontrolki używają również pliku definicji kształtu `box1.xml` do zdefiniowania granic.

## Plik kształtu tła widżetu

Kod tej definicji kształtu został umieszczony na listingu 22.8 (plik ten powinien się znajdować w podkatalogu `/res/drawable`).

**Listing 22.8.** Definicja kształtu krawędzi

---

```
<!-- res/drawable/box1.xml -->
<shape xmlns:android="http://schemas.android.com/apk/res/android">
  <stroke android:width="4dp" android:color="#888888" />
  <padding android:left="2dp" android:top="2dp"
    android:right="2dp" android:bottom="2dp" />
  <corners android:radius="4dp" />
</shape>
```

---

Skorzystaliśmy z takiej metody tworzenia układu graficznego, ponieważ jest przydatna nie tylko w przypadku widżetów, lecz także do tworzenia innych układów graficznych.

Dobrze jest utworzyć aktywność i oddziennie przetestować układy graficzne przed wprowadzeniem ich do widżetu (przynajmniej my tak zrobiliśmy). Uzyskanie odpowiedniego wyglądu i działania widżetu kosztowało nas wiele prób. Bezpośrednie eksperymentowanie na widżetach może się okazać dosyć nużącej czynnością; po każdym uruchomieniu aplikacji należy kolejno usunąć, odinstalować, zainstalować i ponownie umieścić widżety na stronie głównej.

Do tej pory omówione pliki stanowią pełne definicje XML, niezbędne do działania typowego widżetu. Zobaczmy teraz, w jaki sposób będziemy reagować na wydarzenia cyklu życia widżetu. Aby się tego dowiedzieć, przebadamy klasy dostawcy widżetu.

## Implementacja dostawcy widżetu

Przy okazji omawiania architektury widżetu zajelismy się zadaniami klasy dostawcy widżetu. Dostawca ten musi posiadać zaimplementowane następujące metody zwrotne odbiorcy komunikatów:

- `onUpdate()`,
- `onDeleted()`,
- `onEnabled()`,
- `onDisabled()`.

Kod Java zaprezentowany na listingu 22.9 przedstawia implementację każdej z tych metod.

**Listing 22.9.** Przykładowy dostawca widżetu — `BDayWidgetProvider`

```
//src/<your-package>/BDayWidgetProvider.java
public class BDayWidgetProvider extends AppWidgetProvider
{
    private static final String tag = "BDayWidgetProvider";
    public void onUpdate(Context context,
                         AppWidgetManager appWidgetManager,
                         int[] appWidgetIds) {
        final int N = appWidgetIds.length;
        for (int i=0; i<N; i++)
        {
            int appWidgetId = appWidgetIds[i];
            updateAppWidget(context, appWidgetManager, appWidgetId);
        }
    }

    public void onDeleted(Context context, int[] appWidgetIds)
    {
        final int N = appWidgetIds.length;
        for (int i=0; i<N; i++)
        {
            BDayWidgetModel bwm =
                BDayWidgetModel.retrieveModel(context, appWidgetIds[i]);
            bwm.removePrefs(context);
        }
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        final String action = intent.getAction();
        if (AppWidgetManager.ACTION_APPWIDGET_DELETED.equals(action)) {
            Bundle extras = intent.getExtras();
            final int appWidgetId = extras.getInt(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID);

            if (appWidgetId != AppWidgetManager.INVALID_APPWIDGET_ID) {
                this.onDeleted(context, new int[] { appWidgetId });
            }
        }
        else {
            super.onReceive(context, intent);
        }
    }
}
```

```
    }

}

public void onEnabled(Context context) {
    BDayWidgetModel.clearAllPreferences(context);
    PackageManager pm = context.getPackageManager();
    pm.setComponentEnabledSetting(
        new ComponentName("com.ai.android.BDayWidget",
            ".BDayWidgetProvider"),
        PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
        PackageManager.DONT_KILL_APP);
}

public void onDisabled(Context context) {
    BDayWidgetModel.clearAllPreferences(context);
    PackageManager pm = context.getPackageManager();
    pm.setComponentEnabledSetting(
        new ComponentName("com.ai.android.BDayWidget",
            ".BDayWidgetProvider"),
        PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
        PackageManager.DONT_KILL_APP);
}

private void updateAppWidget(Context context,
                            AppWidgetManager appWidgetManager,
                            int appWidgetId) {
    BDayWidgetModel bwm = BDayWidgetModel.retrieveModel(context, appWidgetId);
    if (bwm == null) {
        return;
    }
    ConfigureBDayWidgetActivity
        .updateAppWidget(context, appWidgetManager, bwm);
}
}
```

---

W podrozdziale „Architektura widżetów ekranu startowego” omówiliśmy działanie każdej z tych metod. W naszym widżecie urodzinowym metody te korzystają z kolei z metod umieszczonych w klasie `BDayWidgetModel`. Wśród tych metod są takie, jak `removePrefs()`, `retrievePrefs()` oraz `clearAllPreferences()`.

Klasa `BDayWidgetModel` służy do obudowania stanu instancji widżetu (klaś ta zostanie omówiona za chwilę). Aby zrozumieć tę klasę dostawcy widżetów, wystarczy wiedzieć, że korzystamy z klasy modelu do odczytywania danych wymaganych przez instancję widżetu. Dane te są przechowywane w preferencjach. Dlatego właśnie metody tej klasy noszą nazwy `removePrefs()`, `retrievePrefs()` i `clearAllPreferences()`. Nazwy stają się bardziej zrozumiałe, gdybyśmy zamiast słowa `Prefs` (preferencje) wprowadzili `Data` (dane), w wyniku czego metody te nosiłyby nazwy odpowiednio: `removeData()`, `retrieveData()`, `clearAllData()`. Takie przekształcenie pełni jedynie rolę poglądową i nie znajdziemy metod zawierających przyrostek `Data()`.

Jak już stwierdziliśmy, metoda aktualizacji jest wywoływana dla wszystkich instancji widżetu. Metoda ta musi zaktualizować wszystkie wystąpienia widżetu. Instancje te są przekazywane w postaci tabeli identyfikatorów. Metoda `onUpdate()` zlokalizuje dla każdego atrybutu `id` odpowiedni model instancji widżetu i wywoła tę samą metodę, która jest używana przez aktywność konfiguratora (listing 22.14) w celu wyświetlenia uzyskanego modelu widżetu.

W metodzie `onDeleted()` utworzyliśmy obiekt `BDayWidgetModel1`, a następnie zaprogramowaliśmy jego automatyczne skasowanie z magazynu przechowywanych preferencji.

Ponieważ metoda `onEnabled()` jest wywoływana tylko raz, podczas tworzenia pierwszej instancji, wyczyściliśmy wszystkie przechowywane dane modeli widżetu. Instancia ta zatem rozpoczyna działanie jako czysta — bez danych. Taka sama czynność jest przeprowadzana w przypadku metody `onDisabled()`, dzięki czemu pamięć po instancjach widżetów zostaje całkowicie oczyszczona.

W metodzie `onEnabled()` uruchamiamy dostawcę treści, który może teraz odbierać transmityowane komunikaty. Metoda `onDisabled()` wyłącza ten składnik, więc nie będzie on już wyszukiwał rozwijanych komunikatów.

#### Uwaga!

Szczególnym przypadkiem jest metoda `onReceive()`. Przed wprowadzeniem oprogramowania w wersji 1.6 występował błąd uniemożliwiający wywołanie metody `onDeleted()`. Można było obejść ten problem poprzez jawnie dostarczenie metody `onReceive()`. Od wersji 1.6 Androida metoda ta stała się zbyteczna; wystarczająca okazuje się ta sama metoda z bazowej klasy.

Dzięki implementacji modeli widżetów kod pozostaje czysty. Zajmiemy się teraz kwestią modeli widżetów i sposobem ich implementacji.

## Implementacja modeli widżetów

Czym jest model widżetu? Nie jest to pojęcie swoiste dla Androida. Osoby mające styczność z tradycyjnym programowaniem interfejsów UI z pewnością znają pojęcie wzorca programowania MVC (ang. *Model-View-Controller* — model-widok-kontroler). Zgodnie z tym wzorcem model przechowuje dane wymagane przez widok; widok zapewnia prawidłowe wyświetlanie; natomiast kontroler pośredniczy w komunikacji pomiędzy modelem a widokiem.

Chociaż Android nie zapewnia obsługi takiego rozwiązania, wykorzystaliśmy wzorzec MVC, aby uprościć programowanie widżetów. W tym podejściu każdy widok instancji widżetu posiada ekwiwalentną klasę Java, zwaną modelem widżetu. Model ten zawiera wszystkie metody służące do zapewnienia danych, które będą wymagane do działania instancji widżetu.

Modele te otrzymały także pewne podstawowe klasy, umożliwiające samoistne ich zapisywanie i odczytywanie z trwałych magazynów, na przykład „współdzieronych preferencji”. Prześledźmy hierarchię klas modeli oraz pokażemy, w jaki sposób można wykorzystać współdzielone preferencje do przechowywania i odczytywania danych. Szczegółowe informacje na temat preferencji znajdziemy w rozdziale 9.

## Interfejs modelu widżetu

Rozpoczniemy od omówienia interfejsu zachowującego się jak kontrakt wobec modelu widżetu, dzięki czemu model ten może deklarować zapisywanie pól w trwałej bazie danych. Kontrakt ten definiuje również sposób konfigurowania pola podczas jego odczytu z bazy danych. W dodatku interfejs zawiera metodę zwrotną `init()`, która jest wywoływana podczas odczytywania modelu z bazy danych, a przed przesłaniem tego modelu do klienta wysyłającego żądanie.

Listing 22.10 przedstawia kod źródłowy interfejsu kontraktu naszego widżetu.

**Listing 22.10.** Zachowywanie stanu widżetu — kontrakt

```
//nazwa pliku: src/.../IWidgetModelSaveContract.java
public interface IWidgetModelSaveContract
{
    public void setValueForPref(String key, String value);
    public String getPrefname();

    //zwraca kluczowe pary wartości, które chcemy zachować
    public Map<String, String> getPrefsToSave();

    //zostaje wywołany po odzyskaniu
    public void init();
}
```

Ten interfejs jest zaprojektowany w taki sposób, że wywodząca się z niego abstrakcyjna klasa przeprowadza implementację za pomocą określonego, trwałego magazynu. Wcześniej już wspomnieliśmy, że będziemy korzystać z tej funkcji współdzielonych preferencji Androida jako z trwałego magazynu. Jak wskazuje sama nazwa interfejsu, jest to kontrakt służący wyłącznie do zapisywania. Takie klienty jak `BDayWidgetProvider` nadal będą zależne od najbardziej wydzielonej klasy tego interfejsu, posiadającej swoiste metody.

Realizator tego interfejsu musi dostarczyć nazwę pliku preferencji w odpowiedzi na metodę `getPrefName()`. Plik ten zostaje następnie wykorzystany do zapisania pary klucz – wartość, uzywanej poprzez metodę `getPrefsToSave()`. W odwrotnej operacji (metoda `setValueForPref()`) pochodna klasa ma za zadanie ustanowienie wewnętrznej wartości za pomocą danej pary klucz – wartość, uzyskanej z magazynu preferencji.

Na koniec następuje wywołanie metody `init()` w pochodnej klasie w celu zaznaczenia, że wartości zostały odczytane z trwałego magazynu, oraz umożliwienia przeprowadzenia wszelkich innych inicjalizacji.

**Uwaga!**

Pamiętajmy, że w użytkowej aplikacji wprowadza się nieco inną strukturę dziedziczenia: zamiast dziedziczenia będziemy prawdopodobnie wykorzystywać mechanizm delegacji umożliwiający wielokrotne wykorzystywanie obiektów. Jednak hierarchia dziedziczenia będzie się dobrze sprawowała w naszym testowym widżecie, przedstawionym jako przykład modeli widżetów.

Rozważmy teraz abstrakcyjną implementację przechowującą pola danych widżetu w postaci współdzielonych preferencji.

**Abstrakcyjna implementacja modelu widżetu**

Kod odpowiedzialny za interakcję z trwałym magazynem został zaimplementowany w klasie `APrefWidgetModel` (listing 22.11). Skrót `Pref` w nazwie klasy wywodzi się od wyrazu *Preference* (preferencja), ponieważ klasa ta wykorzystuje funkcję Androida `SharedPreferences` do przechowywania danych modelu widżetu.

Ponadto klasa ta reprezentuje koncepcję prostego widżetu. Pole `iid` stanowi „identyfikator instancji” widżetu. Klasa ta zawsze wymaga obecności konstruktora przyjmującego argument w postaci identyfikatora instancji widżetu, dzięki czemu następuje dostosowanie do wymogów tego identyfikatora.

Przyjrzyjmy się przedstawionemu na listingu 22.11 kodowi źródłowemu tej klasy. Pogrubioną czcionką zaznaczyliśmy jej najważniejsze metody.

**Listing 22.11.** Implementacja procesu zapisywania widżetu poprzez współdzielone preferencje

```
//nazwa pliku: /src/.../APrefWidgetModel.java
public abstract class APrefWidgetModel
implements IWidgetModelSaveContract
{
    private static String tag = "AWidgetModel";

    public int iid;
    public APrefWidgetModel(int instanceId) {
        iid = instanceId;
    }
    //abstrakcyjne metody
    public abstract String getPrefname();
    public abstract void init();
    public Map<String, String> getPrefsToSave(){ return null; }

    public void savePreferences(Context context){
        Map<String, String> keyValuePairs = getPrefsToSave();
        if (keyValuePairs == null){
            return;
        }
        //przechodzi do zapisywania wartości
        SharedPreferences.Editor prefs =
            context.getSharedPreferences(getPrefname(), 0).edit();

        for(String key: keyValuePairs.keySet()){
            String value = keyValuePairs.get(key);
            savePref(prefs, key, value);
        }
        //ostatecznie zapisuje wartości
        prefs.commit();
    }

    private void savePref(SharedPreferences.Editor prefs,
                         String key, String value) {
        String newkey = getStoredKeyForFieldName(key);
        prefs.putString(newkey, value);
    }

    private void removePref(SharedPreferences.Editor prefs, String key) {
        String newkey = getStoredKeyForFieldName(key);
        prefs.remove(newkey);
    }

    protected String getStoredKeyForFieldName(String fieldName){
        return fieldName + "_" + iid;
    }

    public static void clearAllPreferences(Context context, String prefname) {
        SharedPreferences prefs=context.getSharedPreferences(prefname, 0);
        SharedPreferences.Editor prefsEdit = prefs.edit();
        prefsEdit.clear();
        prefsEdit.commit();
    }
}
```

```
public boolean retrievePrefs(Context ctx) {
    SharedPreferences prefs = ctx.getSharedPreferences(getPrefname(), 0);
    Map<String,?> keyValuePairs = prefs.getAll();
    boolean prefFound = false;
    for (String key: keyValuePairs.keySet()){
        if (isItMyPref(key) == true){
            String value = (String)keyValuePairs.get(key);
            setValueForPref(key,value);
            prefFound = true;
        }
    }
    return prefFound;
}

public void removePrefs(Context context) {
    Map<String,String> keyValuePairs = getPrefsToSave();
    if (keyValuePairs == null){
        return;
    }
    //przechodzi do zapisywania wartości
    SharedPreferences.Editor prefs =
        context.getSharedPreferences(getPrefname(), 0).edit();

    for(String key: keyValuePairs.keySet()){
        removePref(prefs,key);
    }
    //ostatecznie zapisuje wartości
    prefs.commit();
}
private boolean isItMyPref(String keyname) {
    if (keyname.indexOf("_" + iid) > 0){
        return true;
    }
    return false;
}
public void setValueForPref(String key, String value) {
    return;
}
}
```

---

Zobaczmy, w jaki sposób są implementowane kluczowe metody tej klasy. Rozpoczniemy od zapisania atrybutów modelu widżetu w pliku współdzielonych preferencji:

```
public void savePreferences(Context context)
{
    Map<String,String> keyValuePairs = getPrefsToSave();
    if (keyValuePairs == null){ return; }

    //przechodzi do zapisywania wartości
    SharedPreferences.Editor prefs =
        context.getSharedPreferences(getPrefname(), 0).edit();

    for(String key: keyValuePairs.keySet()){
        String value = keyValuePairs.get(key);
```

```

        savePref(prefs, key, value);
    }
    //ostatecznie zapisuje wartości
    prefs.commit();
}

```

Działanie tej metody rozpoczyna się od uzyskania od pochodnych klas odwzorowania par klucz – wartość, gdzie kluczami są atrybuty modelu, a wartościami – ciągi znaków reprezentujące wartości tych atrybutów. Metoda ta następnie przekazuje obiektowi context plik *SharedPreferences* poprzez metodę *context.getSharedPreferences()*. Interfejs API wymaga unikatowej nazwy dla tego pakietu. Model pochodny jest odpowiedzialny za jej dostarczenie.

Po uzyskaniu współdzielonych preferencji — zgodnie z dokumentacją Androida — należy uzyskać ich modyfikowalne wersje. Następnie należy je kolejno zaktualizować. Po przeprowadzeniu procesu aktualizacji uruchamiamy metodę *commit()*, co spowoduje zapisanie preferencji.

Więcej informacji można uzyskać, przeglądając dokumentację dotyczącą interfejsów API klas *SharedPreferences* i *SharedPreferences.Editor* oraz rozdział 9.; w zamieszczonym na końcu rozdziału podrozdziale „Odnośniki” znajdują się adresy URL, dzięki którym można uzyskać powyższe informacje. Warto również zwrócić uwagę na fakt, że pliki współdzielonych preferencji są napisane w języku XML i są przechowywane w katalogu danych pakietu.

Ponieważ do przechowywania danych użyliśmy jednego pliku dla wszystkich instancji widżetu, potrzebny jest mechanizm rozróżniania nazw pól pomiędzy wieloma instancjami widżetu. Jeśli na przykład posiadamy dwie instancje widżetu nazwane 1 i 2, wymagane będzie zastosowanie dwóch kluczy przechowujących atrybut Name, tak że będą istniały wartości *name\_1* oraz *name\_2*. Takie przekształcenie przeprowadzamy w następującej metodzie:

```

protected String getStoredKeyForFieldName(String fieldName) {
    return fieldName + "_" + iid;
}

```

Klasa pochodna również wykorzystuje tę metodę do określenia aktualizowanych pól podczas jej wywołania przez metodę *setValue()*.

## Implementacja modelu widżetu Urodziny

Ostatecznie klasa będąca ostatnim potomkiem w tej hierarchii modeli widżetów zapewnia rzeczywiste utrzymywanie wszystkich pól wymaganych przez widok. Klasy bazowe są jej potrzebne do przechowywania i odczytywania danych. Zaprojektowaliśmy tę klasę w taki sposób, że klienty korzystające z tych modeli mają bezpośrednio z nią do czynienia, ponieważ jest to klasa najsilniej z nimi związana.

Na przykład podczas pierwszego utworzenia instancji widżetu przez aktywność konfiguratora aktywność ta konkretyzuje jedną z takich klas, zapełnia ją wartościami i powoduje jej samoistne zapisanie się.

Z powodu wymogów widoku klasa ta przechowuje trzy pola:

- *name* — imię osoby.
- *bday* — data urodzin tej osoby.
- *url* — adres witryny, w której można dokonać zakupu prezentu urodzinowego.

W dalszej kolejności klasa ta zawiera obliczony atrybut `howManyDays`, przedstawiający liczbę dni, jakie pozostały do dnia urodzin danej osoby.

Zobaczmy także, że klasa ta wypełnia kontrakt zapisywania. Potrzebne są do tego następujące metody:

```
public void setValueForPref(String key, String value);
public String getPrefname();
public Map<String, String> getPrefsToSave();
```

Na listingu 22.12 umieszczono kod przeprowadzający te wszystkie czynności.

---

**Listing 22.12.** BDayWidgetModel — implementacja modelu stanów

---

```
//nazwa pliku: /src/.../BDayWidgetModel.java
public class BDayWidgetModel extends APrefWidgetModel
{
    private static String tag="BDayWidgetModel";
    //Generuje niepowtarzalną nazwę służącą do przechowywania daty
    private static String BDAY_WIDGET_PROVIDER_NAME=
        "com.ai.android.BDayWidget.BDayWidgetProvider";

    // Zmienne umożliwiające rysowanie widoku widżetu
    private String name = "anon";
    private static String F_NAME = "name";

    private String bday = "1/1/2001";
    private static String F_BDAY = "bday";

    private String url="http://www.google.com";

    // Instrukcje /get/set konstruktora
    public BDayWidgetModel(int instanceId){
        super(instanceId);
    }
    public BDayWidgetModel(int instanceId, String inName, String inBday){
        super(instanceId);
        name=inName;
        bday=inBday;
    }
    public void init(){}
    public void setName(String inname){name=inname;}
    public void setBday(String inbday){bday=inbday;}

    public String getName(){return name;}
    public String getBday(){return bday;}

    public long howManyDays(){
        try {
            return Utils.howfarInDays(Utils.getDate(this.bday));
        }
        catch(ParseException x){
            return 20000;
        }
    }
}
```

---

//Implementacja kontraktu zapisywania

```

public void setValueForPref(String key, String value){
    if (key.equals(getStoredKeyForFieldName(BDayWidgetModel.F_NAME))){
        this.name = value;
        return;
    }
    if (key.equals(getStoredKeyForFieldName(BDayWidgetModel.F_BDAY))){
        this.bday = value;
        return;
    }
}

public String getPrefname() {
    return BDayWidgetModel.BDAY_WIDGET_PROVIDER_NAME;
}

//Zwraca pary wartości, które chcemy zachować
public Map getPrefsToSave() {
    Map map
    = new HashMap();
    map.put(BDayWidgetModel.F_NAME, this.name);
    map.put(BDayWidgetModel.F_BDAY, this.bday);
    return map;
}
public String toString() {
    StringBuffer sbuf = new StringBuffer();
    sbuf.append("iid:" + iid);
    sbuf.append("name:" + name);
    sbuf.append("bday:" + bday);
    return sbuf.toString();
}
public static void clearAllPreferences(Context ctx){
    APrefWidgetModel.clearAllPreferences(ctx,
        BDayWidgetModel.BDAY_WIDGET_PROVIDER_NAME);
}

public static BDayWidgetModel retrieveModel(Context ctx, int widgetId){
    BDayWidgetModel m = new BDayWidgetModel(widgetId);
    boolean found = m.retrievePrefs(ctx);
    return found ? m:null;
}
}

```

---

Jak widać, w klasie tej znalazły się pewne funkcje związane z datą. Zanim przejdziemy do omówienia implementacji aktywności konfiguracji, zademonstrujemy kody źródłowe tych funkcji.

## Kilka funkcji przetwarzających daty

Na listingu 22.13 przedstawiono kod klasy służącej do pracy z datami. Pobiera ona ciąg znaków daty i sprawdza jego poprawność. Oblicza także różnicę pomiędzy bieżącym dniem a wprowadzoną datą. Kod jest całkowicie zrozumiały. Umieściliśmy go ze względu na zachowanie ciągłości opisu.

**Listing 22.13.** Funkcje daty

```
public class Utils
{
    private static String tag = "Utils";
    public static Date getDate(String dateString)
        throws ParseException {
        DateFormat a = getDateFormat();
        Date date = a.parse(dateString);
        return date;
    }
    public static String test(String sdate){
        try {
            Date d = getDate(sdate);
            DateFormat a = getDateFormat();
            String s = a.format(d);
            return s;
        }
        catch(Exception x){
            return "problem z datą:" + sdate;
        }
    }
    public static DateFormat getDateFormat(){
        SimpleDateFormat df = new SimpleDateFormat("MM/dd/yyyy");
        //DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
        df.setLenient(false);
        return df;
    }

    //poprawne formaty: 1/1/2009, 11/11/2009,
    //niepoprawne formaty: 13/1/2009, 1/32/2009
    public static boolean validateDate(String dateString){
        try {
            SimpleDateFormat df = new SimpleDateFormat("MM/dd/yyyy");
            df.setLenient(false);
            Date date = df.parse(dateString);
            return true;
        }
        catch(ParseException x) {
            return false;
        }
    }
    public static long howfarInDays(Date date){
        Calendar cal = Calendar.getInstance();
        Date today = cal.getTime();
        long today_ms = today.getTime();
        long target_ms = date.getTime();
        return (target_ms - today_ms)/(1000 * 60 * 60 * 24);
    }
}
```

---

Przyjrzyjmy się teraz omówionej wcześniej implementacji aktywności konfiguracji.

## Implementacja aktywności konfiguracji widżetu

W podrozdziale „Architektura widżetów ekranu startowego” wyjaśniliśmy rolę i zadania aktywności konfiguracji. Implementacja tej klasy aktywności w przypadku naszego przykładowego widżetu nosi nazwę `ConfigureBDayWidgetActivity`. Jej kod źródłowy został zaprezentowany na listingu 22.14.

Klasa ta pobiera imię jubilata oraz datę jego następnych urodzin. Następnie tworzy obiekt `BDayWidgetModel`, który jest przechowywany we współdzielonych preferencjach. W klasie tej znajduje się również funkcja, która przenosi obiekt `BDayWidgetModel` do właściwego widoku widżetu.

**Listing 22.14.** Implementacja aktywności konfiguratora

```
public class ConfigureBDayWidgetActivity extends Activity
{
    private static String tag = "ConfigureBDayWidgetActivity";
    private int mAppWidgetId = AppWidgetManager.INVALID_APPWIDGET_ID;
    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.edit_bday_widget);
        setupButton();

        Intent intent = getIntent();
        Bundle extras = intent.getExtras();
        if (extras != null) {
            mAppWidgetId = extras.getInt(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID);
        }
    }

    private void setupButton(){
        Button b = (Button)this.findViewById(R.id.bdw_button_update_bday_widget);
        b.setOnClickListener(
            new Button.OnClickListener(){
                public void onClick(View v)
                {
                    parentButtonClicked(v);
                }
            });
    }

    private void parentButtonClicked(View v){
        String name = this.getName();
        String date = this.getDate();
        if (Utils.validateDate(date) == false){
            this.setDate("nieprawidłowa data:" + date);
            return;
        }
        if (this.mAppWidgetId == AppWidgetManager.INVALID_APPWIDGET_ID){
            return;
        }
        updateAppWidgetLocal(name,date);
    }
}
```

```
Intent resultValue = new Intent();
resultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, mAppWidgetId);
setResult(RESULT_OK, resultValue);
finish();
}

private String getName(){
    EditText nameEdit = (EditText)this.findViewById(R.id.bdw_bday_name_id);
    String name = nameEdit.getText().toString();
    return name;
}

private String getDate(){
    EditText dateEdit = (EditText)this.findViewById(R.id.bdw_bday_date_id);
    String dateString = dateEdit.getText().toString();
    return dateString;
}

private void setDate(String errorDate){
    EditText dateEdit = (EditText)this.findViewById(R.id.bdw_bday_date_id);
    dateEdit.setText("błąd");
    dateEdit.requestFocus();
}

private void updateAppWidgetLocal(String name, String dob){
    BDayWidgetModel m = new BDayWidgetModel(mAppWidgetId, name, dob);
    updateAppWidget(this, AppWidgetManager.getInstance(this), m);
    m.savePreferences(this);
}

public static void updateAppWidget(Context context,
        AppWidgetManager appWidgetManager,
        BDayWidgetModel widgetModel)
{
    RemoteViews views = new RemoteViews(context.getPackageName(),
            R.layout.bday_widget);

    views.setTextViewText(R.id.bdw_w_name
            , widgetModel.getName() + ":" + widgetModel.iid);

    views.setTextViewText(R.id.bdw_w_date
            , widgetModel.getBday());

    //aktualizuje nazwę
    views.setTextViewText(R.id.bdw_w_days, Long.toString(widgetModel.howManyDays()));

    Intent defineIntent = new Intent(Intent.ACTION_VIEW,
            Uri.parse("http://www.google.com"));
    PendingIntent pendingIntent =
            PendingIntent.getActivity(context,
                    0 /*brak requestCode*/,
                    defineIntent,
                    0 /*brak flag*/);
    views.setOnClickPendingIntent(R.id.bdw_w_button_buy, pendingIntent);

    //Informuje menedżer widżetu
    appWidgetManager.updateAppWidget(widgetModel.iid, views);
}
}
```

Jeżeli przyjrzymy się kodowi metody `updateAppWidgetLocal()`, stwierdzimy, że tworzy ona i przechowuje model. Do jego wyświetlania służy natomiast funkcja `updateAppWidget()`. Warto zwrócić uwagę, w jaki sposób wykorzystuje ona intencję oczekującą do rejestrowania metody zwrotnej. Intencja ta pobiera główną intencję, na przykład:

```
Intent defineIntent = new Intent(Intent.ACTION_VIEW,
    Uri.parse("http://www.google.com"));
```

i tworzy kolejną intencję w celu uruchomienia aktywności. I na odwrót — trwająca intencja może zostać również wykorzystana do uruchomienia usługi. Odnotujmy fakt, że funkcja ta działa z klasami `RemoteViews` i `AppWidgetManager`. Zwróćmy uwagę na następujące zadania tej funkcji:

- Uzyskanie widoku `RemoteViews` z układu graficznego.
- Ustanowienie wartości tekstowych w widoku `RemoteViews`.
- Zarejestrowanie trwającej intencji poprzez widok `RemoteViews`.
- Wywołanie klasy `AppWidgetManager` w celu wysłania widoku `RemoteViews` do widżetu.
- Przekazanie wyniku końcowego.

#### Uwaga!

Funkcja statyczna `updateAppWidget` może zostać wywołana z dowolnego miejsca, pod warunkiem że znamy identyfikator widżetu. Wynika z tego wniosek, że można aktualizować widżet z dowolnego miejsca na urządzeniu oraz z dowolnego procesu — zarówno widocznego, jak i niewidocznego.

Istotne jest także, aby zakończyć aktywność w następujący sposób:

```
Intent resultValue = new Intent();
resultValue.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, mAppWidgetId);
setResult(RESULT_OK, resultValue);
finish();
```

Zaobserwujmy, w jaki sposób przekazuje się identyfikator widżetu obiektowi wywołującemu. W ten sposób klasa `AppWidgetManager` uzyskuje informację, że aktywność konfiguradora została zakończona wobec instancji widżetu.

Podsumujemy omawianie tematu konfiguracji widżetu prezentacją układu graficznego formularza tej aktywności, widoczną na listingu 22.15. Widok ten jest bardzo prosty: składa się z kilku pól tekstowych i kontrolek edycji, a także z przycisku aktualizowania. Graficznie zostało to ukazane na rysunku 22.4.

**Listing 22.15.** Definicja układu graficznego dla aktywności konfiguradora

```
<!-- res/layout/edit_bday_widget.xml -->
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/root_layout_id"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/bdw_text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
```

```
    android:text="Imię:"  
    />  
<EditText  
    android:id="@+id/bdw_bday_name_id"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="Anonim"  
    />  
<TextView  
    android:id="@+id/bdw_text2"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="Data urodzenia (9/1/2001):"  
    />  
<EditText  
    android:id="@+id/bdw_bday_date_id"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="np. 10/1/2009"  
    />  
<Button  
    android:id="@+id/bdw_button_update_bday_widget"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="aktualizuj"  
/>  
</LinearLayout>
```

---

Na tym zakończymy temat implementowania przykładowego widżetu. Podczas omawiania tego ćwiczenia zaprezentowaliśmy następujące czynności:

- definiowanie widżetu,
- odpowiedź na metody zwrotne widżetu,
- tworzenie aktywności konfiguracji widżetu,
- zastosowanie klasy `RemoteViews`,
- wprowadzenie struktury zarządzania stanami,
- projektowanie przyjemnego dla oka układu graficznego widżetu.

Teraz przekażemy Czytelnikowi kilka wskazówek dotyczących widżetów.

## Ograniczenia i rozszerzenia widżetów

Na pierwszy rzut oka widżety w Androidzie zdają się bardzo prostymi obiektami. Posiadają one jednak kilka niestandardowych cech, z którymi należy się zaznajomić przed przystąpieniem do ich tworzenia.

Jeżeli dany widżet nie wymaga zarządzania stanami oraz ma być wywoływany co najwyżej kilka razy dziennie, nie powinien istnieć najmniejszy problem z jego napisaniem.

Kolejnego stopnia wtajemniczenia wymaga pisanie widżetu o zarządzanym stanie, który jednak nie będzie wywoływany zbyt często. Do tej kategorii zalicza się omówiony w tym rozdziale widżet *Urodziny*. Struktura zarządzania stanami okazuje się dla takich widżetów bardzo ko-

rzystna. W tym rozdziale pokazaliśmy podstawowy szkielet zarządzania stanami. Zakładamy, że będą dostępne bardziej skomplikowane struktury lub Czytelnik sam napisze bardziej złożony i elastyczny szkielet.

Na kolejnym poziomie trudności znajdą się widżety o częstotliwości wywoływania rzędu sekund lub milisekund. W takim przypadku musimy samodzielnie zaprogramować wywołania aktualizacji za pomocą klasy `AlarmManager`. Będziemy również prawdopodobnie potrzebować usługi częstego zarządzania stanami, niezależnej od trwałej struktury. Jeżeli na przykład chcemy napisać widżet `Stopper`, musimy wprowadzić zegar odmierzający co najmniej sekundowe prze działy oraz śledzić liczniki, w czym pomoże nam zarządzanie stanami.

Kolejnym czynnikiem, który należy wziąć pod uwagę, jest brak mechanizmu pozwalającego klasie `RemoteViews` — od której jest zależna struktura widoku widżetu — na bezpośrednią edycję widżetu (a przynajmniej taki mechanizm nie został udokumentowany). Klasa `RemoteViews` nakłada również ograniczenia na rodzaje wykorzystywanych widoków i układów graficznych. Nie posiadamy bezpośrednią kontroli nad widokami, jedynie pośrednią — poprzez metody obecne w klasie `RemoteViews`.

Opierając się na aktualnie tworzonych projektach i przeznaczeniu widżetów, można dojść do wniosku, że zaleca się pisanie widżetów należących do pierwszej i drugiej kategorii. Istnieje spora doza prawdopodobieństwa, że struktura widżetów zostanie rozszerzona w kolejnych wersjach systemu Android.

## Odbońniki

Podczas przygotowywania materiałów do tego rozdziału odkryliśmy następujące przydatne zasoby (zostały one uporządkowane pod względem przydatności):

- <http://developer.android.com/guide/topics/appwidgets/index.html> — pod tym adresem jest dostępna oficjalna dokumentacja zestawu Android SDK dotycząca widżetów.
- <http://developer.android.com/reference/android/content/SharedPreferences.html> — na tej stronie można znaleźć informacje na temat interfejsu `SharedPreferences`, którego znajomość jest wymagana do zarządzania stanami.
- <http://developer.android.com/reference/android/content/SharedPreferences.Editor.html> — pod tym adresem można poczytać na temat interfejsu `SharedPreferences.Editor`, który jest związany ze wspólnie dostępnymi preferencjami.
- [http://developer.android.com/guide/practices/ui\\_guidelines/widget\\_design.html](http://developer.android.com/guide/practices/ui_guidelines/widget_design.html) — informacje dostępne pod tym adresem przydadzą się w procesie projektowania miłych dla oka układów graficznych widżetów.
- <http://developer.android.com/reference/android/widget/RemoteViews.html> — na tej stronie dostępne są informacje na temat interfejsu `RemoteViews`, który musimy opanować, aby rysować i kontrolować widoki widżetu.
- <http://developer.android.com/reference/android/appwidget/AppWidgetManager.html> — widżety są zarządzane przez klasę menedżera widżetów; interfejs API tej klasy został omówiony na powyższej stronie.
- <http://www.androidbook.com/item/3300> — pod tym adresem jeden ze współautorów książek zamieścił informacje oraz użyteczne fragmenty kodu, które mogą się okazać przydatne, jeżeli musimy szybko zapożyczyć kod stanowiący podstawę widżetów.

- <http://www.androidbook.com/item/3299> — to łącze skieruje nas do notatek z badań, wykorzystanych podczas pisania tego rozdziału.
- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu testowy projekt, przygotowany specjalnie z myślą o niniejszym rozdziale. Właściwy plik jest umieszczony w katalogu o nazwie *ProAndroid3\_R22\_Widżety*.

## Podsumowanie

Poznawanie możliwości widżetów ekranu startowego w Androidzie okazało się dla nas przyjemnym zajęciem. Widżety te stanowią nieskomplikowaną koncepcję, która może w znaczący sposób wpływać na wrażenia użytkownika.

Omówiliśmy teorię dotyczącą widżetów oraz zademonstrowaliśmy działający przykład, ułatwiający zrozumienie ich koncepcji. Wyjaśniliśmy konieczność stosowania modeli widżetów i zarządzania ich stanem. Mamy też nadzieję, że Czytelnikom przyda się zaprezentowany przez nas kod zarządzania stanem podczas tworzenia własnych widżetów. Na końcu poruszliśmy tematykę problemów projektowych oraz ograniczeń widżetów. W rozdziale 31. znajdziemy znacznie dokładniejszą charakterystykę widżetów zaprezentowanych w wersji 3.0 Androida.

## **Wyszukiwanie w Androidzie**

W poprzednich dwóch rozdziałach, 21. i 22., zajmowaliśmy się mechanizmami związanymi ze stroną startową systemu Android. W rozdziale 21. wyjaśniliśmy, w jaki sposób aktywne foldery są umieszczane na stronie startowej oraz jak zapewniają szybki dostęp do zmieniających się danych w dostawcach treści. Rozdział 22. poświęciliśmy analizie widżetów ekranu startowego, wyświetlających fragmenty informacji na stronie startowej.

Kontynuujemy teraz opis takiej koncepcji *informacji w zasięgu ręki*. W niniejszym rozdziale przedstawimy strukturę wyszukiwania w Androidzie. Szkielet wyszukiwania jest w Androidzie niezmiernie rozbudowany. Chociaż wydaje się, że opcja wyszukiwania jest dostępna wyłącznie na ekranie startowym urządzenia, jej zasięg można rozwinąć na aktywności różnorakich aplikacji.

Rozpoczniemy rozdział od opisu funkcji wyszukiwania w Androidzie. Zaprezentujemy koncepcje przeszukiwania globalnego, propozycji wyszukiwania, przepisywania propozycji oraz przeszukiwania sieci WWW. Pokażemy, w jaki sposób umieszczać lokalne aplikacje w procesie przeszukiwania globalnego lub je z niego wykluczać.

Następnie przeanalizujemy integrację aktywności naszej aplikacji z przyciskiem wyszukiwania. Będziemy pracować z aktywnościami, które nie są jawnie przystosowane do wyszukiwania, a także przyjrzymy się aktywności wyłączającej możliwość wyszukiwania. Przeanalizujemy również mechanizm noszący nazwę *type-to-search* („napisz, aby szukać”), dzięki któremu aktywności mają możliwość wywołania procesu wyszukiwania. Ponadto zademonstrujemy aktywność, która w jawnym sposób wywołuje proces wyszukiwania za pomocą menu.

Podstawowym elementem, odpowiedzialnym za elastyczność wyszukiwania w Androidzie, jest tak zwany *dostawca propozycji*. Przeanalizujemy uważnie tę koncepcję i pokażemy, jak utworzyć prostego dostawcę propozycji poprzez dziedziczenie po bazowym dostawcy dostępnym w Androidzie.

Często jednak trzeba samodzielnie napisać dostawcę propozycji od podstaw. Będzie to kolejne zagadnienie, które będzie dotyczyć zasadniczej części architektury wyszukiwania w Androidzie.

Na koniec zajmiemy się dwoma zaawansowanymi zagadnieniami. Pokażemy, w jaki sposób możemy wykorzystać przyciski działania dostępne w urządzeniu do wywoływania niestandardowych działań korzystających z propozycji wyszukiwania. Zastanowimy się także na sposobem przekazania danych określonej aplikacji do wywołanego procesu wyszukiwania. Rozdział zamkniemy zestawem odnośników.

## Wyszukiwanie w Androidzie

Wyszukiwanie w Androidzie rozszerza możliwości znanego nam internetowego paska wyszukiwania Google o zdolność przeszukiwania zarówno lokalnej zawartości urządzenia, jak i treści zewnętrznych, udostępnionych w internecie. Mechanizm ten może być również wykorzystany do bezpośredniego wywoływania aplikacji z poziomu paska wyszukiwania na stronie startowej. Android umożliwia korzystanie z tych funkcji poprzez wprowadzenie struktury wyszukiwania pozwalającej na współuczestniczenie w niej aplikacji lokalnych.

Protokół przeszukiwania w Androidzie jest prosty. Dostępne jest pojedyncze pole wyszukiwania, w którym użytkownicy wpisują poszukiwany ciąg znaków. Jest tak zarówno w przypadku stosowania pola wyszukiwania globalnego na stronie startowej, jak i przeszukiwania własnej aplikacji — wykorzystywane jest to samo pole wyszukiwania.

Tekst wprowadzany przez użytkownika jest pobierany przez system już w trakcie wpisywania i przekazywany różnym aplikacjom, które zostały zarejestrowane do odpowiadania na proces wyszukiwania. Zareagują one w ten sposób, że przekażą zestaw odpowiedzi. Android zbiera te odpowiedzi z różnych aplikacji i wyświetla je w postaci listy możliwych **propozycji**.

Po kliknięciu jednej z tych odpowiedzi system wywołuje aplikację przedkładającą wybraną propozycję. W tym sensie mamy do czynienia z wyszukiwaniem sfederowanym (pojęcie to oznacza mechanizm umożliwiający zintegrowany dostęp do rozproszonych zasobów) wewnętrz zbioru współuczestniczących aplikacji.

Chociaż ogólna idea jest całkiem prosta, szczegóły protokołu wyszukiwania są dosyć złożone. W dalszej części rozdziału będziemy je objaśniać na działających przykładach. W tym podroziale przyjrzymy się procesowi wyszukiwania z perspektywy użytkownika.

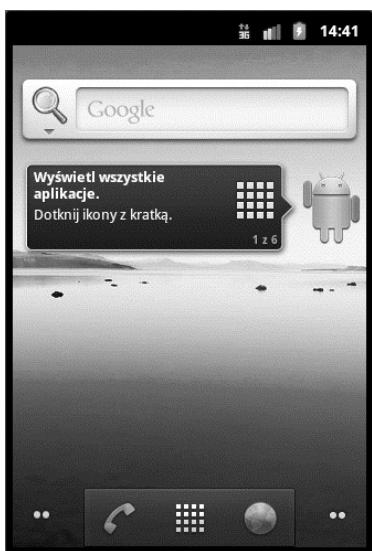
## Badanie procesu przeszukiwania globalnego w Androidzie

Chociaż nie wymagamy tego bezwzględnie, bardzo zalecamy przejrzenie działu poświęconego „wyszukiwaniu” w instrukcji obsługi systemu Android w trakcie zapoznawania się z treścią niniejszego rozdziału. W podroziale „Odnośniki” zamieściliśmy łącze do najnowszej wersji takiej instrukcji użytkownika.

### Uwaga!

W trakcie pisania tej książki pojawiały się kolejno wersje: 2.0, 2.2, 2.3 oraz 3.0 Androida. Chociaż sam interfejs API nie został zmieniony, nieznaczny modyfikacjom ulegał sam interfejs użytkownika. Zrzuty ekranu w tym rozdziale zostały wykonane na emulatorze pracującym z wersją 2.2 Androida. Chociaż przetestowaliśmy kody w wersjach 2.3 i 3.0 systemu, nie wykonaliśmy w nich przypadku żadnego rysunku. W odpowiednich miejscach tego rozdziału wspominamy jednak o różnicach. Niezależnie od posiadanej wersji Androida Czytelnik nie powinien mieć problemu z wyobrażeniem sobie sposobu działania równoważnych wersji interfejsu użytkownika. Zastanówmy się na przykład nad ustawieniami wyszukiwania. W każdej kolejnej wersji systemu miejsce wywoływania ekranu ustawień wyszukiwania ulegało zmianie, jednak sam ekran ustawień wyglądał tak samo. Sugerujemy więc, aby w trakcie czytania rozdziału brać pod uwagę wspomniane rozbieżności.

Nie da się ominąć wyszukiwania w Androidzie; pole wyszukiwania jest zazwyczaj umieszczone na stronie startowej, co zostało ukazane na rysunku 23.1. Pole wyszukiwania nazywane jest także polem QSB (ang. *Quick Search Box* — pole szybkiego wyszukiwania). W niektórych wersjach Androida lub w przypadku niektórych producentów urządzeń czy operatorów pole to może nie być domyślnie widoczne na ekranie startowym. Z pewnością jednak ujrzymy je po wcisnięciu przycisku wyszukiwania. W przypadku urządzeń nieposiadających przycisków fizycznych (na przykład tabletów) dostrzeżemy jakiś inny oczywisty mechanizm wywoływanego pola QSB. Wszystko zostało dokładnie opisane w instrukcji użytkowania danej wersji Androida.



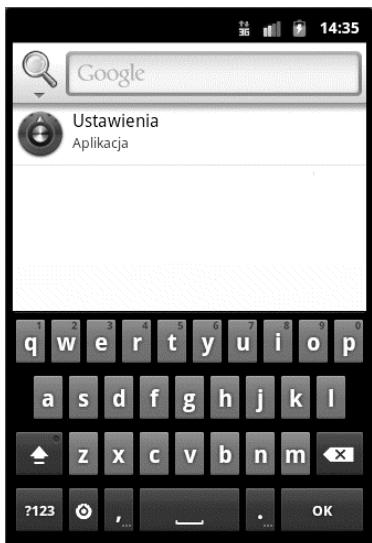
**Rysunek 23.1.** Strona startowa Androida z widocznym polem QSB i przyciskiem wyszukiwania

Ponieważ pole QSB zostało wstawione w formie widżetu (rozdział 22. został poświęcony tematyce widżetów), możemy przenieść je na ekran startowy, jeżeli jeszcze go tam nie ma. Równie dobrze możemy to pole usunąć z ekranu startowego — wystarczy je przenieść do kosza. Oczywiście zawsze możemy je później przywrócić do poziomu ekranu widżetów.

W celu rozpoczęcia wyszukiwania możemy bezpośrednio pisać w polu QSB. Możemy wtedy zaobserwować interesujący efekt spowodowany tym, że pole QSB jest widżetem: bezpośrednio po uaktywnieniu pola QSB na ekranie startowym system uruchamia aktywność wyszukiwania globalnego (rysunek 23.2), w wyniku czego opuszczamy kontekst tego ekranu. Rysunek 23.2 został wykonany w wersji 2.2 Androida. Ekran ten wygląda identycznie w wersji 2.3 systemu.

Jak już wspomnieliśmy, możemy także wywołać proces wyszukiwania, klikając odpowiedni przycisk działania. Przyciski działania stanowią zestaw przycisków widocznych na rysunku 23.1 po prawej stronie. Na interesującym nas przycisku został umieszczony symbol lupy.

Podobnie jak w przypadku przycisku ekranu startowego, możemy kliknąć przycisk wyszukiwania w dowolnym momencie, bez względu na uruchomioną aplikację. Jednak jeżeli aplikacja została umieszczona w głównym wątku, umożliwia ona zawężenie wyszukiwania, czym zajmiemy się w dalszej części rozdziału. Takie zawężone wyszukiwanie nazywane jest **wyszukiwaniem lokalnym**. Bardziej ogólne, powszechnie i niewyspecjalizowane wyszukiwanie nosi miano **wyszukiwania globalnego**.



**Rysunek 23.2.** Aktywność wyszukiwania globalnego uruchomiona za pomocą widżetu ekranu startowego

#### Uwaga!

Jeśli wciśniemy przycisk wyszukiwania w obrębie aktywnej aplikacji, to od tej aplikacji zależy, czy pozwoli ona na korzystanie z lokalnego, czy globalnego wyszukiwania. W wersjach systemu starszych od 2.0 domyślnym zachowaniem było umożliwienie wyszukiwania globalnego. W wersjach 2.2 i 2.3 z kolei standardowo wyszukiwanie globalne zostaje w takim przypadku uniemożliwione. Oznacza to, że jeżeli użytkownik korzysta z danej aktywności, musi najpierw wcisnąć przycisk ekranu startowego, a dopiero w dalszej kolejności — przycisk wyszukiwania.

W wersjach systemu starszych od 2.2 pole wyszukiwania globalnego nie rozróżniało poszczególnych dostawców propozycji wyszukiwania (lub wyszukiwarek). Począwszy od wersji 2.2, aplikacja Android Search pozwala wybrać konkretny kontekst wyszukiwania (synonim dostawcy propozycji).

W tym celu należy kliknąć ikonę widoczną po lewej stronie pola QSB. Zostanie otwarta lista poszczególnych aplikacji przeszukujących. Lista taka (dla wersji 2.2 Androida) została zaprezentowana na rysunku 23.3. W przypadku wersji 2.3 systemu widok ten ulega niewielkiej zmianie — w prawej górnej części rozwiniętej sekcji kryteriów wyszukiwania została wprowadzona mała ikona ustawień przeszukiwania.

Jest to domyślny zbiór aplikacji wyszukujących (lub kontekstów czy też typów wyszukiwania, ewentualnie dostawców propozycji), dostępnych w emulatorze dla wersji 2.2 i 2.3 systemu. W nowszych wersjach lista ta może wyglądać inaczej. Kontekst wyszukiwania *Wszystko* zajął miejsce wyszukiwania globalnego, znanego z wcześniejszych wersji Androida.

Możemy również utworzyć własny kontekst wyszukiwania poprzez napisanie dostawców propozycji wyszukiwania oraz aktywności przeszukiwania lokalnego. Zajmiemy się tym zagadniением podczas omawiania różnorodnych przykładów zawartych w tym rozdziale.



**Rysunek 23.3.** Pole wyszukiwania globalnego z widocznymi kontekstami różnych aplikacji wyszukujących

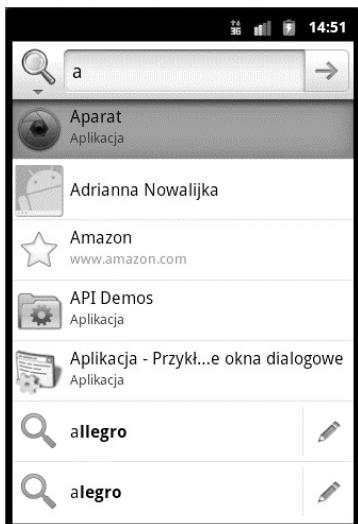
Przyjrzyjmy się kontekstowi wyszukiwania symbolizowanemu przez ikonę lupy. Przechodzimy do tego pola (rysunek 23.1), klikając bezpośrednio jego obszar albo ikonę wyszukiwania. Nie wpisujemy jeszcze niczego w polu QSB. W tym momencie Android będzie wyświetlał ekran przypominający zrzut z rysunku 23.2.

W zależności od wcześniej dokonywanych czynności obraz widoczny na rysunku 23.2 może wyglądać nieco inaczej, ponieważ Android na podstawie uprzednich działań stara się odgadnąć, czego szukamy. Taki tryb wyszukiwania, w którym w polu QSB nie został wprowadzony tekst, nosi nazwę **trybu propozycji zerowych**. Zależnie od wpisanego tekstu Android wyświetli różną liczbę propozycji. Zostaną one wyświetcone pod polem QSB w formie listy. Elementy tej listy często są nazywane **propozycjami wyszukiwania**. W miarę wpisywania kolejnych liter Android będzie dynamicznie aktualizował wyniki wyszukiwania. Jeżeli w polu wyszukiwania nie zostanie wpisany tekst, zostaną wyświetcone tak zwane **propozycje zerowe**. Na rysunku 23.2 widać, że aplikacja Spare Parts została uznana przez Android za używaną w przeszłości, zatem nadającą się na propozycję, mimo że żaden tekst nie został wpisany w polu wyszukiwania. Chociaż nie wprowadziliśmy tekstu w polu QSB, zostaje wyświetlona klawiatura programowa. Jest ona widoczna na rysunku 23.2.

Po wprowadzeniu znaku *a* w polu QSB Android wyszukuje propozycje rozpoczynające się od litery *a* lub zawierające tę literę. Stwierdzimy, że Android przeszukał już lokalnie zainstalowane aplikacje rozpoczynające się na literę *a* oraz dużą liczbę innych propozycji wyszukiwania.

Skorzystamy teraz z przycisku przesunięcia kurSORA w dół w celu zaznaczenia pierwszej propozycji. Widok ten został ukazany na rysunku 23.4.

Zauważmy, że pierwsza propozycja została zaznaczona, a pierwszoplanowym obiektem już nie jest pole QSB, tylko ta zaznaczona propozycja. Kliknijmy ikonę widoczną po prawej stronie pola QSB, aby kontynuować proces wyszukiwania. Obszar ekranu został również powiększony poprzez schowanie klawiatury programowej, gdyż nie będziemy z niej korzystać podczas przeglądania listy propozycji. Dzięki temu na ekranie widać równocześnie więcej propozycji.



**Rysunek 23.4.** Propozycje wyszukiwania

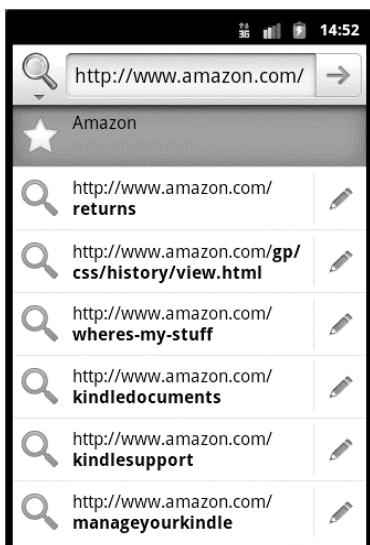
Przyjrzyjmy się jednak ponownie propozycjom. Android pobiera tekst wpisany w polu wyszukiwania i wyszukuje obiekty znanego jako **dostawcy propozycji**. Android wywołuje asynchronicznie każdego dostawcę propozycji w celu uzyskania pasujących propozycji, przybierających postać zbioru krotek. Android oczekuje, że te krotki (zwane **propozycjami wyszukiwania**) będą pasować do zestawu predefiniowanych kolumn (**kolumny propozycji**). W miarę przeszukiwania tych znanych kolumn Android będzie kompletował listę propozycji. Po zmianie tekstu wpisanego w polu QSB Android przeprowadzi cały proces od początku. Taki sposób pracy, polegający na wywoływaniu wszystkich dostawców propozycji w celu uzyskania propozycji wyszukiwania, jest właściwy w przypadku kontekstu odpowiedzialnego za wyszukiwanie globalne. Jeżeli jednak wybierzemy konkretny kontekst wyszukiwania, na przykład taki, jaki jest widoczny na rysunku 23.3, w celu odczytania propozycji wyszukiwania zostanie przywołany jedynie dostawca propozycji zdefiniowany dla tej aplikacji.

**Uwaga!**

Zbiór propozycji wyszukiwania zwany jest także **kursorem propozycji**. Wynika to z faktu, że dostawca treści reprezentujący dostawcę propozycji przekazuje obiekt cursor.

Jeżeli w tym momencie ponownie klikniemy w polu QSB, system znowu wyświetli klawiaturę programową. Kolejną rzeczą z rysunku 23.4 wartą odnotowania jest zależność pomiędzy zaznaczoną propozycją a tekstem wyszukiwania w polu QSB. Tekst wyszukiwania ciągle składa się wyłącznie z litery *a*, mimo że został zaznaczony konkretny element, w naszym wypadku aplikacja Aparat. Jednak nie zawsze tak jest, co widać na rysunku 23.5, który przedstawia zaznaczenie propozycji wskazującej adres sklepu Amazon.

Zauważmy, że wstawiona przez nas litera *a* została zastąpiona przez pełny adres URL serwisu Amazon. Możemy teraz kliknąć strzałkę (która będziemy nazywać strzałką nawigacji), aby otworzyć stronę Amazon, lub zwyczajnie kliknąć zaznaczoną propozycję. W obydwu przypadkach skutek będzie identyczny.



Rysunek 23.5. Przepisanie propozycji

**Uwaga!**

Taki proces modyfikowania tekstu wyszukiwania na podstawie zaznaczonej propozycji nosi nazwę **przepisywania propozycji**.

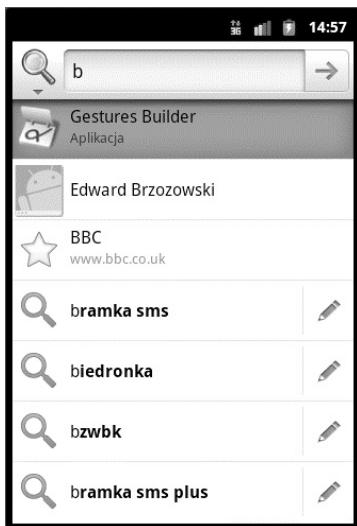
Nieco później zajmiemy się dokładniej procesem przepisywania propozycji, w skrócie jednak Android wykorzystuje jedną z kolumn kurSORA propozycji do wyszukiwania tego tekstu. Jeżeli taka kolumna istnieje, zostanie przepisany tekst wyszukiwania, w przeciwnym razie pozostanie on niezmieniony.

Jeżeli propozycja nie zostanie przepisana, istnieją dwie możliwości. Jeżeli klikniemy ikonę wyszukiwania w polu QSB, niezależnie od tego, co zostało zaznaczone, Android przystąpi do przeszukiwania w bazie Google. Jeśli zaś bezpośrednio klikniemy element propozycji, w aplikacji, która wygenerowała daną propozycję, zostanie wywołana **aktywność wyszukiwania**. Aktywność ta jest odpowiedzialna za wyświetlenie wyników wyszukiwania.

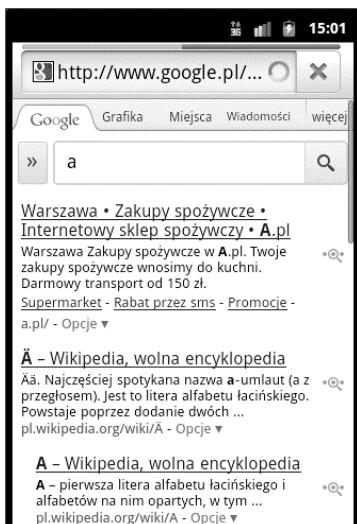
Na rysunku 23.6 widać przykład bezpośredniego wywoływania propozycji. W naszym wypadku takim przykładem jest aplikacja APIDemos. Po kliknięciu propozycji aplikacja ta zostanie bezpośrednio wywołana. Przebieg tego procesu jest dosyć skomplikowany i zajmiemy się jego omówieniem w dalszej części rozdziału (w punkcie „Implementacja niestandardowego dostawcy propozycji”).

Na rysunku 23.7 zaprezentowano, co się stanie po kliknięciu strzałki nawigacji w przypadku wprowadzenia litery *a* w polu QSB.

Po przedstawieniu techniki wyszukiwania za pomocą pola QSB przejdziemy do omówienia, w jaki sposób możemy włączać i wyłączać określone aplikacje ze wspólnego dnia w przeszukiwaniu globalnym.



Rysunek 23.6. Wywoływanie aplikacji za pomocą wyszukiwania



Rysunek 23.7. Przeszukiwanie internetu

## Włączanie dostawców propozycji do procesu wyszukiwania globalnego

Jak już stwierdziliśmy, aplikacje posługują się dostawcami propozycji w celu przekazania odpowiedzi na proces wyszukiwania. Choć aplikacja może posiadać infrastrukturę niezbędną do odpowiedzi na wyszukiwanie, nie oznacza to wcale, że jej propozycje będą automatycznie wyświetlane w polu QSB. Użytkownik musi zezwolić dostawcy propozycji na udział w tym procesie. Przedstawimy teraz kolejne etapy włączania lub wyłączania posiadanych dostawców propozycji. Proces uzyskiwania dostępu do omawianych poniżej ustawień różni się nieznacznie pomiędzy wersjami 2.2 i 2.3 Androida. Najpierw zajmiemy się wersją 2.2.

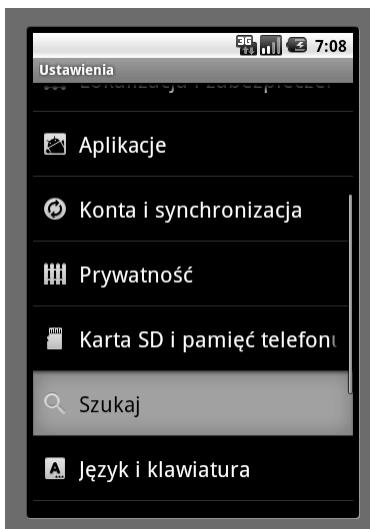
## Praca z ustawieniami wyszukiwania w wersji 2.2 Androida

Rozpoczniemy od ekranu ustawień Androida (rysunek 23.8).



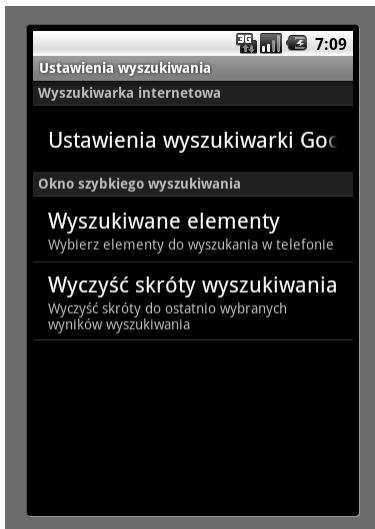
**Rysunek 23.8.** Lokalizowanie aplikacji odpowiedzialnej za ustawienia

Dostęp do tego widoku uzyskujemy poprzez kliknięcie ikony reprezentującej listę aplikacji, umieszczonej w dolnej części ekranu urządzenia (ekran startowy został pokazany na rysunku 23.1). Należy wyszukać tam aplikację Ustawienia, której ikonę pokazaliśmy na rysunku 23.8, i uruchomić ją. Zostanie wyświetlona strona z ustawieniami Androida, przypominająca ekran z rysunku 23.9.



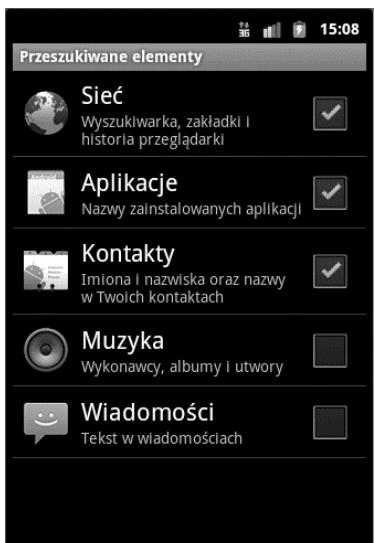
**Rysunek 23.9.** Wyszukiwanie ustawień aplikacji Ustawienia wyszukiwania

Spośród różnorodnych ustawień Androida wybieramy opcję *Szukaj*. Zostanie uruchomiona aplikacja Ustawienia wyszukiwania, zilustrowana na rysunku 23.10.



Rysunek 23.10. Aplikacja Ustawienia wyszukiwania

Znajdujemy w tej aktywności zakładkę *Okno szybkiego wyszukiwania* i wybieramy opcję *Wyszukiwane elementy* (*Wybierz elementy do wyszukania w telefonie*). Ukaże się zaprezentowana na rysunku 23.11 lista dostępnych dostawców propozycji (nazywanych czasem aplikacjami wyszukującymi). Przypominamy, że zależnie od wersji systemu lista ta może wyglądać inaczej.



Rysunek 23.11. Włączone i wyłączone aplikacje wyszukujące

Dostawcy propozycji (lub stanowiące ich części aplikacje), włączone do wyszukiwania globalnego, są pokazane na rysunku 23.11 jako zaznaczone. Nowy dostawca propozycji nie jest domyślnie zaznaczony. Aby został dołączony do procesu wyszukiwania, trzeba kliknąć jego nazwę. Po takim włączeniu dany dostawca będzie umieszczał propozycje w oknie globalnego wyszukiwania. Taki włączony dostawca propozycji zostanie również wyświetlony pośród aplikacji wyszukujących, widocznych na rysunku 23.3.

## Praca z ustawieniami wyszukiwania w wersji 2.3 Androida

Różnica w dostępie do ustawień dostawców propozycji pomiędzy wersjami 2.2 i 2.3 systemu (oraz, mniejmy nadzieję, przyszłymi wersjami) polega na sposobie otwierania ekranu ustawień wyszukiwania, widocznego na rysunkach 23.10 i 23.11.

W wersji 2.3 Androida możemy bezpośrednio otworzyć widok pokazany na rysunku 23.11 z poziomu rozwiniętego ekranu kryteriów wyszukiwania (rysunek 23.2). Znajdziemy tu niewielką ikonę ustawień. Po jej kliknięciu system natychmiast uruchomi ekran widoczny na rysunku 23.11, gdzie ujrzymy wszystkie niestandardowe aktywności wyszukiwania.

Aby przejść do ogólnego ekranu ustawień wyszukiwania (rysunek 23.10), musimy otworzyć któryś z ekranów widocznych na rysunkach 23.2, 23.3 lub 23.4. W istocie następuje kliknięcie pola QSB. Jeżeli pole QSB jest aktywne, po kliknięciu przycisku *Menu* ujrzymy obiekt menu nazwany *Ustawienia wyszukiwania*. Jego kliknięcie umożliwia dostęp do ogólnych ustawień wyszukiwania, znanych z rysunku 23.10. Gdy już otworzymy ten ekran, instrukcje korzystania z ustawień są takie same jak w przypadku wersji 2.2 Androida.

Do tej pory zapoznaliśmy się z ogólnym mechanizmem działania wyszukiwania w Androidzie. Teraz przyjrzymy się dokładniej omawianym pojęciom oraz zademonstrujemy na przykładach zasadę ich działania. Rozpoczniemy od oddziaływania prostych aktywności na proces wyszukiwania.

## Interakcja aktywności z przyciskiem wyszukiwania

Co się stanie po wciśnięciu przycisku wyszukiwania, w przypadku gdy na pierwszym planie znajduje się aktywność? Odpowiedź zależy od rodzaju tej aktywności. Prześledzimy zachowanie następujących typów aktywności:

- zwykła aktywność niezwiązana z wyszukiwaniem,
- aktywność jawnie wyłączająca wyszukiwanie,
- aktywność jawnie wywołującą wyszukiwanie globalne,
- aktywność określająca wyszukiwanie lokalne.

Przeanalizujemy te opcje na przykładowych projektach składających się z następujących plików (po omówieniu każdego z nich zaprezentujemy zrzuty ekranu, ukazujące koncepcje tej aplikacji):

- *RegularActivity.java* (listing 23.1),
- *NoSearchActivity.java* (listing 23.6),
- *SearchInvokerActivity.java* (listing 23.8),
- *LocalSearchEnabledActivity.java* (listing 23.13),
- *SearchActivity.java* (listing 23.11).

Poza ostatnią aktywnością (*SearchActivity.java*) każda z pozostałych reprezentuje po jednym rodzaju z wymienionych wcześniej aktywności. Plik *SearchActivity.java* jest wymagany przez aktywność *LocalSearchEnabledActivity*. Każda aktywność, w tym *SearchActivity*, zawiera prosty układ graficzny z widokiem tekstowym. Obsługiwane są następujące pliki układu graficznego:

- *res/layout/main.xml* (dla aktywności *RegularActivity*; listing 23.3),
- *res/layout/no\_search\_activity.xml* (listing 23.7),
- *res/layout/search\_invoker\_activity.xml* (listing 23.9),
- *res/layout/local\_search\_enabled\_activity.xml* (listing 23.14),
- *res/layout/search\_activity.xml* (listing 23.11).

Dwa następne pliki definiują te aktywności w Androidzie oraz przeszukują metadane w przypadku aktywności obsługującej wyszukiwanie lokalne:

- *AndroidManifest.xml* (listing 23.2),
- *xml/searchable.xml* (listing 23.12).

W kolejnym pliku zostały umieszczone dane tekstowe każdego układu graficznego w postaci ciągu znaków:

- *res/values/strings.xml* (listing 23.4).

Wymienione poniżej dwa pliki dostarczają menu niezbędne do wywołania tych aktywności w razie potrzeby oraz do wyszukiwania globalnego:

- *res/menu/main\_menu.xml* (listing 23.5),
- *res/menu/search\_invoker\_menu.xml* (listing 23.10).

Zbadamy teraz interakcję pomiędzy tymi aktywnościami a przyciskiem wyszukiwania poprzez metodyczny przegląd kodu źródłowego tych plików pod kątem rodzajów aktywności.

**Uwaga!**

Jeśli Czytelnik zechce skompilować i przetestować omawiane pliki, zalecamy pobranie projektów, które można zimportować do środowiska Eclipse. Adres URL do tych projektów znajduje się w podrozdziale „Odbońniki”.

Rozpoczniemy od przeanalizowania zachowania przycisku wyszukiwania w obecności standardowej aktywności Androida.

## Zachowanie przycisku wyszukiwania wobec standardowej aktywności

Na listingu 23.1 przedstawiliśmy kod źródłowy Java przykładowej aktywności *RegularActivity*. Umożliwi on nam sprawdzenie, co się dzieje, w przypadku gdy na pierwszym planie systemu znajduje się niezwiązana z wyszukiwaniem aktywność.

### Listing 23.1. Kod źródłowy standardowej aktywności

---

```
//nazwa pliku: RegularActivity.java
public class RegularActivity extends Activity
{
    private final String tag = "RegularActivity";
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}

@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    //wywołuje nadziedną klasę w celu dołączenia menu systemowych
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    //metoda getMenuInflater() pochodzi z bazowej aktywności

    inflater.inflate(R.menu.main_menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item)
{
    appendMenuItemText(item);
    if (item.getItemId() == R.id.menu_clear) {
        this.emptyText();
        return true;
    }

    if (item.getItemId() == R.id.mid_no_search) {
        this.invokeNoSearchActivity();
        return true;
    }
    if (item.getItemId() == R.id.mid_local_search) {
        this.invokeLocalSearchActivity();
        return true;
    }
    if (item.getItemId() == R.id.mid_invoke_search) {
        this.invokeSearchInvokerActivity();
        return true;
    }
    return true;
}

private TextView getTextView()
{
    return (TextView)this.findViewById(R.id.text1);
}

private void appendMenuItemText(MenuItem menuItem)
{
    String title = menuItem.getTitle().toString();
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + title);
}

private void emptyText()
{
    TextView tv = getTextView();
```

```
        tv.setText("");
    }
private void invokeNoSearchActivity()
{
    //odkomentujmy poniższe wiersze w momencie
    //dodania tej aktywności do projektu

    //Intent intent =
    // new Intent(this,NoSearchActivity.class);
    //startActivity(intent);
}
private void invokeSearchInvokerActivity()
{
    //odkomentujmy poniższe wiersze w momencie
    //dodania tej aktywności do projektu

    //Intent intent =
    // new Intent(this,SearchInvokerActivity.class);
    //startActivity(intent);
}
private void invokeLocalSearchActivity()
{
    //odkomentujmy poniższe wiersze w momencie
    //dodania tej aktywności do projektu

    //Intent intent =
    // new Intent(this,LocalSearchEnabledActivity.class);
    //startActivity(intent);
}
}
```

---

Zadaniem tego kodu jest odgrywanie roli prostej aktywności, niezwiązanej z wyszukiwaniem. Jednak ta przykładowa aktywność steruje również wywoływaniem pozostałych testowanych przez nas aktywności. Dlatego widać w kodzie dodatkowe elementy menu reprezentujące te aktywności. Każda funkcja rozpoczynająca się od instrukcji typu `invoke...` zawiera kod pozwalający na uruchomienie pozostałych rodzajów aktywności.

Wszystkie pliki niezbędne do komplikacji zostały umieszczone jeden po drugim, jednak już teraz Czytelnik może oznaczyć jako komentarze funkcje typu `invoke...` lub dołączyć do projektu kod tych klas. Aby ułatwić zadanie, opatrzyliśmy już znakami komentarzy odpowiednie wiersze.

Spójrzmy na plik manifest, aby dowiedzieć się, w jaki sposób omawiana aktywność jest definiowana (listing 23.2). Widoczne są również definicje pozostałych aktywności, zostaną one jednak omówione w dalszej części rozdziału. Także tym razem oznaczamy jako komentarze te dodatkowe aktywności — aż do czasu, gdy będą potrzebne.

---

**Listing 23.2.** Interakcja aktywności z przyciskiem wyszukiwania — plik manifest

```
//nazwa pliku: manifest.xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.search.nosearch">
<application android:icon="@drawable/icon"
```

```

    android:label="Interakcja aktywności testowej z polem QSB">
<activity android:name=".RegularActivity"
    android:label="Interakcja aktywność/QSB: Standardowa aktywność">
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

<!-- W trakcie tworzenia poszczególnych aktywności odkomentujmy je.
Podpowiemy, gdy nadziejdie właściwa chwila.

<activity android:name=".NoSearchActivity"
    android:label=" Interakcja aktywność/QSB:Wyłączone wyszukiwanie">
</activity>

<activity android:name=".SearchInvokerActivity"
    android:label=" Interakcja aktywność/QSB:Wywołanie wyszukiwania">
</activity>
<activity android:name=".LocalSearchEnabledActivity"
    android:label=" Interakcja aktywność/QSB:Wyszukiwanie lokalne">
<meta-data android:name="android.app.default_searchable"
    android:value=".SearchActivity" />
</activity>

<activity android:name=".SearchActivity"
    android:label=" Interakcja aktywność/QSB:Wyniki wyszukiwania">
<intent-filter>
    <action android:name="android.intent.action.SEARCH" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
<meta-data android:name="android.app.searchable"
    android:resource="@xml/searchable" />
</activity>
-->
</application>
<uses-sdk android:minSdkVersion="4" />
</manifest>

```

Zauważmy, że klasa RegularActivity jest zdefiniowana jako główna aktywność tego projektu i nie posiada innych parametrów związanych z wyszukiwaniem.

Plik układu graficznego tej aktywności został ukazany na listingu 23.3.

### **Listing 23.3.** Plik układu graficznego standardowej aktywności

```

//nazwa pliku: layout/main.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"

```

---

```

    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/regular_activity_prompt"
    />
</LinearLayout>
```

---

Zaprezentujemy teraz wykorzystywane w tym projekcie zasoby znakowe. Na listingu 23.4 zostały umieszczone ciągi znaków wykorzystywane w innych aktywnościach. Jednak te dodatkowe zasoby nie powinny mieć wpływu na proces komplikowania bieżącej aktywności, nawet jeśli nie zostały wprowadzone pozostałe klasy.

W ten sposób na listingu 23.4 została zaprezentowana zawartość pliku *strings.xml* przechowującego tekst wyświetlany przez naszą aktywność. Pogrubioną czcionką i jako komentarze zaznaczyliśmy sekcje odpowiedzialne za poszczególne aktywności.

**Listing 23.4.** Interakcja aktywność – przycisk wyszukiwania — plik strings.xml

---

```

//nazwa pliku: /res/values/strings.xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
<!--
*****
* regular_activity_prompt
*****
-->
<string name="regular_activity_prompt">
Jest to przykładowa aplikacja, w której testowana jest interakcja pomiędzy polem QSB
z przyciskiem wyszukiwania a aktywnością. Zostały w niej zawarte cztery aktywności,
włącznie z niniejszą. Obecnie jest uruchomiona standardowa aktywność. Dostęp do
pozostałych trzech aktywności uzyskujemy poprzez menu.
\n\n
Jest to standardowa aktywność, nieposiadająca funkcji związanych z wyszukiwaniem.
Jeżeli klikniemy teraz przycisk wyszukiwania, zostanie wywołane przeszukiwanie
globalne.
\n
\nPozostałe aktywności to:
\n\n1) Aktywność braku wyszukiwania: aktywność, w której wyszukiwanie zostało
wyłączone.
\n2) Wywołanie wyszukiwania: programowe wywołanie wyszukiwania globalnego.
\n3) Aktywność wyszukiwania lokalnego: przywołuje wyszukiwanie lokalne.
\n
\nTutaj będą się pojawiały informacje o błędach.
</string>

<!--
*****
* no_search_activity_prompt
*****
-->
<string name="no_search_activity_prompt">
W tej aktywności metoda onSearchRequested
zwraca wartość false. Przycisk wyszukiwania
będzie teraz ignorowany.
```

```
\n\nMożna kliknąć przycisk powrotu, aby uzyskać dostęp\ndo poprzedniej aktywności i wybrać w menu\ninną aktywność.\n</string>\n<!--\n*****\n* search_activity_prompt\n*****\n-->\n<string name="search_activity_prompt">\nJest to tak zwana aktywność wyszukiwania lub aktywność wyników wyszukiwania.\nTa aktywność jest wywoływana podczas wcisnięcia przycisku wyszukiwania w trakcie\nużywania tej aktywności przez inną aktywność w roli aktywności wyników wyszukiwania.\n\\n\nZazwyczaj możemy uzyskać z intencji ciąg znaków kwerendy,\naby dowiedzieć się, czym jest ta kwerenda.\n</string>\n<!--\n*****\n* search_invoker_activity_prompt\n*****\n-->\n<string name="search_invoker_activity_prompt">\nW tej aktywności element menu wyszukiwania jest stosowany\ndo wywołania domyślnego wyszukiwania. W tym przypadku\nnie zostało dla tej aktywności określone wyszukiwanie lokalne, więc\nzostaje wywołane wyszukiwanie globalne. Kliknięcie\nprzycisku menu spowoduje wyświetlenie menu „wyszukiwania”. Po jego\nkliknięciu zostanie uruchomione wyszukiwanie globalne.\n</string>\n<!--\n*****\n* local_search_enabled_activity_prompt\n*****\n-->\n<string name="local_search_enabled_activity_prompt">\nJest to bardzo prosta aktywność, dla której w pliku manifeście\nzostała wskazana powiązana aktywność wyszukiwania.\nW ten sposób po wcisnięciu przycisku wyszukiwania zostaje wyświetlone\nwyszukiwanie lokalne.\n\\n\nJego lokalność jest widoczna w etykiecie pola QSB oraz w jego podpowiedzi.\nObydwa te elementy pochodzą z metadanych wyszukiwania.\n\\n\nPo kliknięciu ikony kwerendy zostaniemy przeniesieni\ndo aktywności wyszukiwania lokalnego.\n</string>\n<!--\n*****\n* Inne wartości\n*****\n-->
```

```
<string name="search_label">Demonstracja wyszukiwania lokalnego</string>
<string name="search_hint">Podpowiedź do demonstracji wyszukiwania
    ↵ lokalnego</string>
</resources>
```

---

Podobnie jak w przypadku pliku manifestu, plik *strings.xml* jest wykorzystywany przez wszystkie aktywności tego projektu. Możemy zauważyc, że obecna w tym pliku stała typu `string regular_activity` wskazuje tekst widoczny na ekranie aktywności standardowej.

Aby wspomóc proces komplikacji standardowej aktywności, na listingu 23.5 zaprezentujemy plik z zasobami menu. Chociaż zawarte są w nim elementy związane z pozostałymi aktywnościami, nie będą one wpływać na proces komplikowania i umożliwią nam korzystanie ze standardowej aktywności, dostępnej na listingu 23.1.

#### **Listing 23.5.** Plik menu standardowej aktywności

---

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<!-- nazwa pliku: /res/menu/main_menu.xml -->
<!--Ta grupa korzysta z domyślnej kategorii. -->
<group android:id="@+id/menuGroup_Main">
    <item android:id="@+id/mid_no_search"
        android:title="Aktywność braku wyszukiwania" />
    <item android:id="@+id/mid_local_search"
        android:title="Aktywność wyszukiwania lokalnego" />
    <item android:id="@+id/mid_invoke_search"
        android:title="Aktywność wywołania wyszukiwania" />
    <item android:id="@+id/menu_clear"
        android:title="wyczyszc" />
</group>
</menu>
```

---

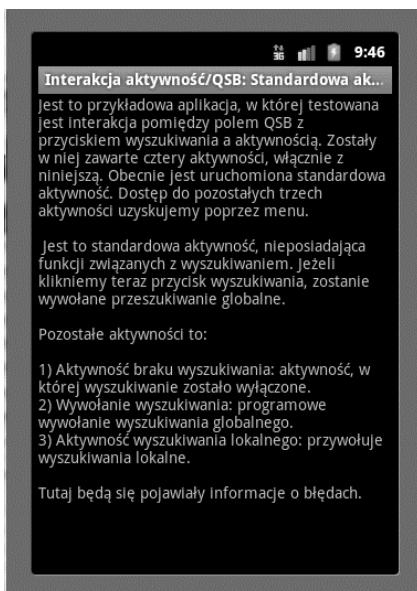
Po utworzeniu tych plików możemy skompilować i przetestować tę aktywność (lub poczekać do momentu zakończenia omawiania wszystkich rodzajów aktywności tego projektu). Jeżeli chcemy dokonać komplikacji teraz, na listingach 23.1 oraz 23.2 (w pliku manifeście) musimy oznaczyć komentarzem wszystkie pozostałe aktywności. Możemy ewentualnie skorzystać z listingów zdefiniowanych na początku podrozdziału, aby skompilować cały projekt, a dopiero później go testować.

Po skompilowaniu aplikacji i uruchomieniu głównej, standardowej aktywności układ graficzny powinien być podobny do przedstawionego na rysunku 23.12.

Listing 23.5 prezentuje plik XML menu stosowanego w tej standardowej aktywności. Wygląd tego menu został pokazany na rysunku 23.13.

Po uruchomieniu tej aktywności (powinna wyglądać jak na rysunku 23.12) kliknijmy przycisk wyszukiwania (jest on widoczny na rysunku 23.1). Zgodnie z dokumentacją powinno zostać wywołane okno dialogowe globalnego wyszukiwania.

W wersjach systemu starszych od 2.0 wciśnięcie przycisku wyszukiwania w odpowiedzi uruchamiało proces wyszukiwania globalnego. W wersjach 2.2 i 2.3 tak się nie dzieje.



**Rysunek 23.12.** Interakcja standardowa aktywność – wyszukiwanie



**Rysunek 23.13.** Uzyskiwanie dostępu do pozostałych aktywności testowych.

Jeżeli chcemy wymusić na standardowej aktywności korzystanie z wyszukiwania globalnego, musi ona przesłaniać metodę `onSearchRequested()` i wykonywać następujące operacje:

```
@Override
public boolean onSearchRequested()
{
    Log.d(tag, "Wywołano zadanie metody onSearch");
}
```

```
    this.startSearch("test",true,null,true);
    return true;
}
```

Po umieszczeniu powyższego kodu w pliku *RegularActivity.java* możemy wcisnąć przycisk wyszukiwania, co spowoduje wywołanie pola globalnego przeszukiwania. Metoda `startSearch()` wraz z argumentami zostanie omówiona w dalszej części rozdziału. Pole globalnego wyszukiwania będzie wyglądało tak jak przedstawione na rysunku 23.2.

## Zachowanie aktywności wyłączającej wyszukiwanie

Aktywność posiada możliwość całkowitego wyłączenia wyszukiwania (zarówno globalnego, jak i lokalnego) poprzez przekazanie wartości `false` z metody zwrotnej `onSearchRequested()` danej klasy aktywności. Na listingu 23.6 został przedstawiony kod źródłowy takiej aktywności, nazwanej *NoSearchActivity*.

**Listing 23.6.** Aktywność wyłączająca wyszukiwanie

---

```
//nazwa pliku: NoSearchActivity.java
public class NoSearchActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.no_search_activity);
        return;
    }
    @Override
    public boolean onSearchRequested()
    {
        return false;
    }
}
```

---

Listing 23.7 prezentuje plik układu graficznego tej aktywności.

**Listing 23.7.** Plik XML NoSearchActivity

---

```
//nazwa pliku: layout/no_search_activity.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/no_search_activity_prompt"
    />
</LinearLayout>
```

---

Po utworzeniu tych dwóch plików (listingi 23.6 oraz 23.7) musimy usunąć znaki komentarzy dla kilku sekcji w dwóch następujących plikach:

- *RegularActivity.java* (listing 23.1),
- *AndroidManifest.xml* (listing 23.2).

W pliku *RegularActivity.java* (listing 23.1) usuwamy znaki komentarza z kodu znajdującego się w segmencie funkcji `invokeNoSearchActivity()`.

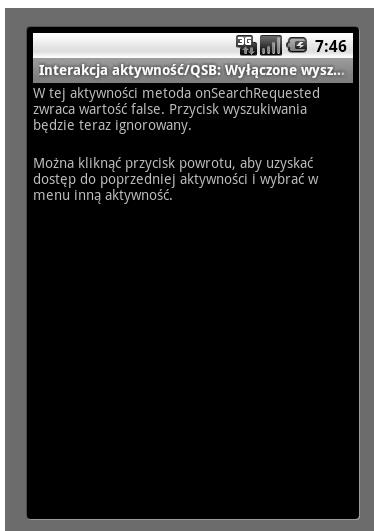
W pliku *AndroidManifest.xml* (listing 23.2) usuwamy znaki komentarza z definicji aktywności `NoSearchActivity`. Zwróćmy uwagę, że mamy do czynienia z plikiem XML. Sposób zaznaczania jako komentarz i usuwania tego oznaczenia w przypadku pliku XML różni się od analogicznej czynności przeprowadzanej w pliku Java.

Możemy teraz wywołać aktywność `NoSearchActivity`, klikając element menu *Aktywność braku wyszukiwania*, widoczny na rysunku 23.13.

Po jego kliknięciu zostanie wyświetlony ekran pokazany na rysunku 23.14. Teraz wciśnięcie przycisku wyszukiwania nie spowoduje żadnej reakcji; jego wciśnięcie nie zostanie odnotowane.

#### Uwaga!

W przypadku obecności aktywności wyłączającej wyszukiwanie kliknięcie przycisku wyszukiwania spowoduje zablokowanie wywoływanego zarówno lokalnego, jak i globalnego wyszukiwania.



Rysunek 23.14. Aktywność wyłączonego wyszukiwania

## Jawne wywoływanie wyszukiwania za pomocą menu

Oprócz odpowiedzi na kliknięcie przycisku wyszukiwania, aktywność może również w jawnym sposobie wywoływać wyszukiwanie za pomocą elementu menu wyszukiwania. Listing 23.8 ukazuje kod źródłowy przykładowej aktywności (`SearchInvokerActivity`) posiadającej taką możliwość.

**Listing 23.8.** Aktywność SearchInvokerActivity

```
//nazwa pliku: SearchInvokerActivity.java
public class SearchInvokerActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.search_invoker_activity);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu)
    {
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(R.menu.search_invoker_menu, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item)
    {
        appendMenuItemText(item);
        if (item.getItemId() == R.id.mid_si_clear)
        {
            this.emptyText();
            return true;
        }
        if (item.getItemId() == R.id.mid_si_search)
        {
            this.invokeSearch();
            return true;
        }
        return true;
    }

    private TextView getTextView()
    {
        return (TextView)this.findViewById(R.id.text1);
    }

    private void appendMenuItemText(MenuItem menuItem)
    {
        String title = menuItem.getTitle().toString();
        TextView tv = getTextView();
        tv.setText(tv.getText() + "\n" + title);
    }

    private void emptyText()
    {
        TextView tv = getTextView();
        tv.setText("");
    }

    private void invokeSearch()
    {
        this.onSearchRequested();
    }
}
```

---

```

    }
    @Override
    public boolean onSearchRequested()
    {
        this.startSearch("test",true,null,true);
        return true;
    }
}

```

---

Najważniejsze fragmenty kodu zostały zaznaczone pogrubioną czcionką. Warto przeanalizować sposób, w jaki identyfikator menu (`R.id.mid_si_search`) wywołuje funkcję `invokeSearch`, która z kolei wywołuje metodę `onSearchRequested()`. Metoda ta przywołuje proces wyszukiwania.

Metoda bazowa `startSearch()` posiada następujące argumenty:

- `initialQuery` — wyszukiwany tekst.
- `selectInitialQuery` — wartość logiczna określająca, czy wyszukiwany tekst ma zostać zaznaczony, czy nie. W tym przypadku wprowadzamy wartość `true`, dzięki czemu tekst zostanie zaznaczony, co z kolei pozwala na jego wykasowanie i wstawienie nowego tekstu w razie potrzeby.
- `appSearchData` — obiekt typu `Bundle` przekazywany aktywności przeszukującej. W tym przypadku nie określamy konkretnej aktywności wyszukującej, wprowadzamy więc tu wartość `null`.
- `globalSearch` — w przypadku wartości `true` zostanie wywołane pole wyszukiwania globalnego. W przeciwnym wypadku zostanie w razie możliwości wywołany proces wyszukiwania lokalnego; jeżeli nie będzie on dostępny, zostanie wywołane pole wyszukiwania globalnego.

W przeciwnieństwie do tego, co pokazaliśmy na listingu 23.8, dokumentacja zestawu SDK zaleca wywoływanie bazowej metody `onSearchRequested()`. Jednak domyślnie metoda ta w ostatnim argumencie metody `startSearch()` posiada wartość `false`. Zgodnie z dokumentacją oznacza to, że w przypadku braku wyszukiwania lokalnego zostanie wywołane pole wyszukiwania globalnego. W tej wersji systemu jednak (dotyczy to zarówno wersji 2.2, jak i 2.3 Androida) wyszukiwanie globalne nie zostaje przywołane. Być może mamy tu do czynienia z błędem lub zamierzonym działaniem, którego opis nie został zaktualizowany w dokumentacji.

W naszym przykładzie wymusiliśmy wywołanie wyszukiwania globalnego poprzez wstawienie wartości `true` do ostatniego argumentu metody `startSearch()`.

Na listingu 23.9 przedstawiono układ graficzny tej aktywności.

**Listing 23.9.** Plik XML układu graficznego aktywności `SearchInvokerActivity`

---

```

//nazwa pliku: layout/search_invoker_activity.xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"

```

```
    android:layout_height="wrap_content"
    android:text="@string/search_invoker_activity_prompt"
  />
</LinearLayout>
```

---

Z kolei listing 23.10 zawiera kod XML menu tej aktywności.

**Listing 23.10.** Plik XML menu aktywności SearchInvokerActivity

---

```
//nazwa pliku: menu/search_invoker_menu.xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <!-- Ta grupa korzysta z domyślnej kategorii. -->
  <group android:id="@+id/menuGroup_Main">
    <item android:id="@+id/mid_si_search"
          android:title="Szukaj" />

    <item android:id="@+id/mid_si_clear"
          android:title="wyczyść" />
  </group>
</menu>
```

---

Po utworzeniu tych trzech plików (listingi 23.8, 23.9 oraz 23.10) musimy usunąć znaki komentarza z kilku sekcji w dwóch następujących plikach:

- *RegularActivity.java* (listing 23.1),
- *AndroidManifest.xml* (listing 23.2).

W pliku *RegularActivity.java* (listing 23.1) usuwamy znaki komentarza z kodu znajdującego się w segmencie funkcji *invokeSearchInvokerActivity()*.

W pliku *AndroidManifest.xml* (listing 23.2) usuwamy znaki komentarza z definicji aktywności *SearchInvokerActivity*.

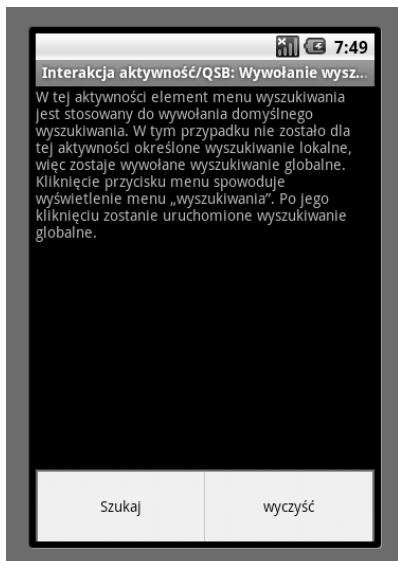
Rysunek 23.15 ukazuje wygląd tej aktywności po wywołaniu jej z menu głównego aktywności *RegularActivity* (na rysunku 23.13 widzimy element menu *Aktywność wywoływanie wyszukiwania*).

W przypadku tej aktywności kliknięcie przycisku *Szukaj* spowoduje wywołanie znanego nam pola wyszukiwania globalnego, widocznego na rysunku 23.2. Wciśnięcie przycisku wyszukiwania również spowoduje wyświetlenie globalnego pola QSB, ponieważ przesłoniliśmy metodę *onSearchRequested()* bazowej aktywności.

## Wyszukiwanie lokalne i pokrewne aktywności

Określmy teraz warunki, w jakich wciśnięcie przycisku wyszukiwania *nie* wywoła wyszukiwania globalnego, lecz lokalne. Najpierw jednak musimy nieco lepiej wyjaśnić pojęcie wyszukiwania lokalnego.

Mechanizm wyszukiwania lokalnego składa się z trzech elementów. Pierwszy z nich stanowi pole wyszukiwania bardzo przypominające pole QSB (jeżeli nie identyczne). Bez względu na to, czy pole QSB jest lokalne, czy globalne, jest ono kontrolką tekstową pozwalającą na wprowadzenie danych i uruchomienie przeszukiwania za pomocą ikony wyszukiwania. Lokalne pole



**Rysunek 23.15.** Aktywność wywoływanie wyszukiwania

QSB jest wywoływanie, w przypadku gdy aktywność deklaruje w pliku manifestie potrzebę wyszukiwania lokalnego. Lokalne pole QSB można odróżnić od pola QSB globalnego po nagłówku (rysunek 23.18) i podpowiedzi (tekście znajdującym się wewnątrz pola wyszukiwania) tego widoku. Jak się będzie mogła przekonać, te dwie wartości pochodzą z pliku XML metadanych wyszukiwania.

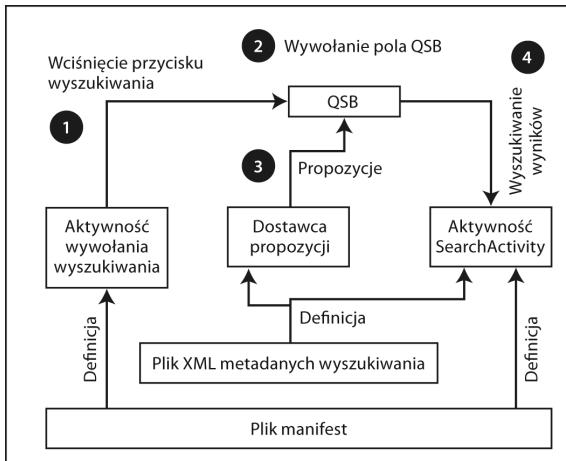
Drugim elementem wyszukiwania lokalnego jest aktywność odbierająca z lokalnego pola QSB wpisany ciąg znaków oraz wyświetlającą zbiór wyników lub danych wyjściowych, związań z tym ciągiem znaków. Aktywność ta często jest nazywana aktywnością wyszukiwania lub aktywnością wyników wyszukiwania.

Trzecim, opcjonalnym składnikiem wyszukiwania lokalnego jest aktywność, która może wywoływać dopiero co wspomnianą aktywność wyników wyszukiwania (drugiego składnika). Aktywność ta jest często nazywana wywołaniem wyszukiwania lub aktywnością wywołującą wyszukiwanie. Jest ona opcjonalna, ponieważ istnieje możliwość bezpośredniego wywołania aktywności wyszukiwania lokalnego (drugiego elementu) z poziomu wyszukiwania globalnego za pomocą propozycji.

Na rysunku 23.16 zostało przedstawione oddziaływanie tych składników między sobą wewnątrz kontekstu.

Najważniejsze interakcje zaprezentowane na rysunku 23.16 zostały oznaczone numerami. Poniżej zostały dokładniej omówione zjawiska ukazane na tym rysunku.

- Aktywność `SearchActivity` musi posiadać zdefiniowaną w pliku manifestie możliwość odbierania żądań wyszukiwania. Wykorzystuje ona również obowiązkowy plik XML do odpowiedniego wyświetlenia lokalnego pola QSB (na przykład jego tytułu, podpowiedzi itd.) oraz powiadomienia o obecności powiązanego z nim dostawcy propozycji (listing 23.12). Na rysunku 23.16 etap ten został zaznaczony liniami nazwanymi *Definicja*, przebiegającymi pomiędzy aktywnością `SearchActivity` a dwoma plikami XML (plikiem manifestem oraz plikiem metadanych wyszukiwania).



Rysunek 23.16. Interakcja aktywności wyszukiwania lokalnego

- Po zdefiniowaniu aktywności SearchActivity w pliku manifeście (listing 23.2) aktywność wywołania wyszukiwania wskazuje na ich powiązanie ze sobą poprzez definicję metadanych `android.app.default_searchable`.
- Po utworzeniu obydwóch definicji wciśnięcie przycisku wyszukiwania spowoduje wywołanie lokalnego pola QSB, w przypadku gdy aktywność wywołania wyszukiwania znajduje się na pierwszym planie. Na rysunku 23.16 jest to oznaczone cyframi 1 i 2. Możemy stwierdzić, że pole QSB jest lokalne po jego tytule oraz podpowiedzi. Te dwie wartości są konfigurowane w obowiązkowym pliku definicji XML metadanych wyszukiwania. Po wywołaniu pola QSB za pomocą przycisku wyszukiwania możemy umieszczać w nim zapytania dotyczące wyszukiwania. Podobnie jak w przypadku globalnego pola QSB, również pole lokalne posiada możliwość wyświetlania propozycji. Zostało to zaznaczone na rysunku 23.16 cyfrą 3.
- Po wprowadzeniu zapytania i kliknięciu ikony wyszukiwania dane z pola QSB zostaną przeniesione do aktywności SearchActivity, która przetwarza kwerendy, na przykład na ich podstawie wyświetla zestaw wyników. Jest to etap oznaczony na rysunku 23.16 cyfrą 4.

Każda z tych interakcji przeanalizujemy pod kątem odpowiedniego kodu źródłowego. Rozpoczniemy od listingu 23.11, zawierającego kod źródłowy aktywności SearchActivity (której zadaniem jest odbieranie zapytań i wyświetlanie wyników wyszukiwania).

**Listing 23.11. Aktywność SearchActivity oraz jej układ graficzny**

```
//nazwa pliku: SearchActivity.java
public class SearchActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.search_activity);
        return;
    }
}
```

//A także odpowiadający jej plik res/layout/search\_activity.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/search_activity_prompt"
    />
</LinearLayout>
```

Wprowadziliśmy możliwie najprostszą aktywność wyszukiwania. Później wyjaśnimy, w jaki sposób pytania są odbierane przez tę aktywność. Na razie zademonstrujemy sposób jej wywoływania przez pole QSB. Jest ona zdefiniowana w pliku manifeście jako aktywność wyszukiwania odpowiedzialna za wyniki w następujący sposób (listing 23.2):

```
<activity android:name=".SearchActivity"
          android:label="Interakcja aktywność/QSB:Wyniki wyszukiwania">
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
    <meta-data android:name="android.app.searchable"
              android:resource="@xml/searchable" />
</activity>
```

#### Uwaga!

Należy określić dwa elementy dla aktywności wyszukiwania. Musi ona zadeklarować możliwość odpowiedzi na działania SEARCH, a także określić plik XML, w którym zawarty jest opis metadanych wymaganych do interakcji z tą aktywnością.

Na listingu 23.12 prezentujemy zawartość pliku XML metadanych wyszukiwania dla aktywności SearchActivity.

#### Listing 23.12. Plik searchable.xml — metadane wyszukiwania

```
<!-- /res/xml/searchable.xml -->
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/search_label"
    android:hint="@string/search_hint"
    android:searchMode="showSearchLabelAsBadge"
/>
```

#### Wskazówka

Omówienie różnych opcji dostępnych w tym pliku XML znajduje się pod adresem <http://developer.android.com/reference/android/app/SearchManager.html>.

W dalszej części rozdziału zajmiemy się omówieniem większości tych atrybutów. Na razie wystarczy zapamiętać, że atrybut `android:label` służy do przypisania etykiety polu wyszukiwania. Dzięki atrybutowi `android:hint` możemy umieszczać tekst wewnątrz pola wyszukiwania, co jest widoczne na rysunku 23.18.

Zobaczmy teraz, w jaki sposób dowolna aktywność może wyznaczyć tę aktywność `SearchActivity` jako swojego adresata wyszukiwania. Nazwiemy tę aktywność `LocalSearchEnabledActivity`. Na listingu 23.13 został umieszczony jej kod źródłowy.

---

**Listing 23.13.** Aktywność LocalSearchEnabledActivity

```
//nazwa pliku: LocalSearchEnabledActivity.java
public class LocalSearchEnabledActivity extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.local_search_enabled_activity);
        return;
    }
}
```

---

Listing 23.14 przedstawia plik XML układu graficznego tej aktywności.

---

**Listing 23.14.** Plik układu graficznego aktywności LocalSearchEnabledActivity

```
<?xml version="1.0" encoding="utf-8"?>
<!-- nazwa pliku: layout/local_search_enabled_activity.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/local_search_enabled_activity_prompt"
    />
</LinearLayout>
```

---

Zwróćmy również uwagę, że klasa `LocalSearchEnabledActivity` (listing 23.14) definiuje klasę `SearchActivity` (listing 23.11) jako swoją docelową aktywność wyszukiwania. Relację tę odnajdziemy w definicji aktywności `LocalSearchEnabledActivity` (listing 23.2). Poniżej przypominamy treść tej definicji:

```
<activity android:name=".LocalSearchEnabledActivity"
    android:label="Interakcja aktywność/QSB::Wyszukiwanie lokalne">
    <meta-data android:name="android.app.default_searchable"
        android:value=".SearchActivity" />
</activity>
```

A teraz możemy skompletować wszystkie nowe pliki w celu przetestowania dwóch nowych aktywności: `LocalSearchEnabledActivity` oraz `SearchActivity`. Nazwy plików oraz numery odpowiadających im listingów znajdziemy poniżej:

- `SearchActivity.java` (listing 23.11),
- `layout/search_activity.xml` (część listingu 23.11),
- `res/xml/searchable.xml` (listing 23.12),
- `LocalSearchEnabledActivity.java` (listing 23.13),
- `local_search_enabled_activity` (listing 23.14).

Po utworzeniu tych plików musimy usunąć znaki komentarza z kilku sekcji w dwóch następujących plikach:

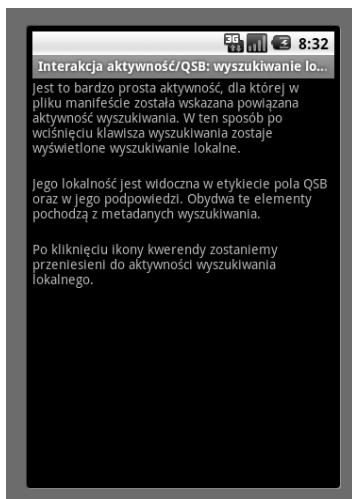
- `RegularActivity.java` (listing 23.1),
- `AndroidManifest.xml` (listing 23.2).

W pliku `RegularActivity.java` (listing 23.1) usuwamy znaki komentarza z kodu znajdującego się w segmencie funkcji `invokeLocalSearchActivity()`.

W pliku `AndroidManifest.xml` (listing 23.2) usuwamy znaki komentarza z definicji aktywności `LocalSearchEnabledActivity` oraz `SearchActivity`.

Po udanym usunięciu znaków komentarzy z kodu w obydwu plikach można ponownie skompilować projekt.

Po uzyskaniu tych aktywności wraz z ich układami graficznymi możemy wywołać aktywność `LocalSearchEnabledActivity` z poziomu głównej aktywności `RegularActivity`, klikając element menu *Aktywność wyszukiwania lokalnego* (na rysunku 23.13 są widoczne elementy menu). Po wywołaniu aktywność będzie wyglądała tak jak na rysunku 23.17.



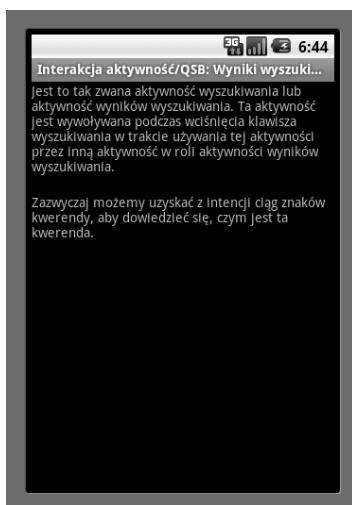
**Rysunek 23.17.** Aktywność uruchamiająca wyszukiwanie lokalne

Jeżeli aktywność ta znajduje się na pierwszym planie, wciśnięcie na urządzeniu przycisku wyszukiwania wywoła pole lokalnego wyszukiwania (lokalne pole QSB), uwidocznione na rysunku 23.18.



**Rysunek 23.18.** Pole wyszukiwania lokalnego

Spójrzmy na etykietę oraz podpowiedź tego pola wyszukiwania. Różnią się one od analogicznych elementów pola globalnego wyszukiwania (rysunek 23.2). Pochodzą one z metadanych wyszukiwania zdefiniowanych dla aktywności `SearchActivity` (plik `searchable.xml` z listingu 23.12). Jeżeli wprowadzimy teraz jakiś tekst w polu QSB i klikniemy ikonę wyszukiwania, zostanie wywołana aktywność `SearchActivity` (listing 23.11). Ekran tej aktywności został pokazany na rysunku 23.19.



**Rysunek 23.19.** Wyniki wyszukiwania w odpowiedzi na dane wpisane w polu wyszukiwania lokalnego

Chociaż ta aktywność nie wykorzystuje żadnej kwerendy wyszukiwania tekstu do uzyskiwania zestawu wyników, służy ona do pokazania sposobu definiowania aktywności oraz jej wywoływanego. W dalszej części rozdziału zademonstrujemy mechanizm wykorzystywania przez aktywność `SearchResults` kwerend wyszukiwania oraz różnych działań wymaganych do przetwarzania tych zapytań.

## Uruchomienie funkcji type-to-search

Dotychczas zaprezentowaliśmy kilka sposobów wywoływania wyszukiwania lokalnego i globalnego. Pokazaliśmy, w jaki sposób można przeprowadzać wyszukiwanie za pomocą pola QSB na stronie startowej urządzenia. Wyjaśniliśmy, jak można wywołać wyszukiwanie globalne z poziomu dowolnej aktywności, pod warunkiem że możliwość wyszukiwania nie została w tej aktywności wyłączona. Omówiliśmy również określanie wyszukiwania lokalnego za pomocą aktywności. Zakończymy ten temat, pokazując jeszcze jeden sposób wywoływania wyszukiwania, zwany *type-to-search* (wpisz, aby szukać).

Jeśli przeanalizujemy takie aktywności, jak `RegularActivity` ukazana na rysunku 23.12, to stwierdzimy, że można wywołać wyszukiwanie, wpisując losowo wybraną literę (na przykład t). Jest to tryb noszący nazwę *type-to-search*, gdyż naciśnięcie dowolnego przycisku niewykorzystywanego przez aktywność wywoła proces wyszukiwania.

Założenia mechanizmu *type-to-search* są całkiem proste. W przypadku dowolnej aktywności Androida możemy wprowadzić ustawienie, które spowoduje, że naciśnięcie jakiegokolwiek przycisku — oprócz przycisków jawnie obsługiwanych przez aktywność — wywoła wyszukiwanie. Jeżeli na przykład aktywność obsługuje wyłącznie przyciski x i y, do wywołania wyszukiwania mogą posłużyć wszystkie pozostałe, chociażby z lub a. Jest to przydatny tryb w przypadku aktywności już wyświetlającej wyniki wyszukiwania. Aktywność taka może interpretować naciśnięcie przycisku jako sygnał do rozpoczęcia nowego wyszukiwania.

Poniżej umieściliśmy dwa przykładowe wiersze kodu służące do uruchomienia takiego zachowania, wstawiane do metody `onCreate()` (pierwszy wiersz jest odpowiedzialny za wywoływanie wyszukiwania globalnego, a drugi — za wywoływanie wyszukiwania lokalnego):

```
this.setDefaultKeyMode(Activity.DEFAULT_KEYS_SEARCH_GLOBAL);
```

lub

```
this.setDefaultKeyMode(Activity.DEFAULT_KEYS_SEARCH_LOCAL);
```

Wydaje się, że wywoływanie globalnego wyszukiwania za pomocą mechanizmu *type-to-search* nie podążą ścieżką z wykorzystaniem metody `onSearchRequested()`. Naciśnięcie przycisku wywołuje wyszukiwanie globalne w bezpośredni sposób. W wyniku tego może się wydawać, że utworzona w naszym przykładzie klasa `RegularActivity` będzie wywoływać wyszukiwanie globalne w momencie aktywowania funkcji *type-to-search* (przypomnijmy sobie, że w trakcie testowania standardowej aktywności, która nie włączała ani nie wyłączała w jawnym sposób wyszukiwania, próba przywołania pola wyszukiwania globalnego za pomocą przycisku kończyła się niepowodzeniem). Możemy przetestować zachowanie funkcji *type-to-search*, umieszczając następujący wiersz kodu na końcu metody `onCreate()` w klasie `RegularActivity` (listing 23.1):

```
this.setDefaultKeyMode(Activity.DEFAULT_KEYS_SEARCH_GLOBAL);
```

Teraz wprowadzenie jakiejś litery, np. t, na ekranie widocznym na rysunku 23.12 spowoduje wywołanie pola przeszukiwania globalnego.

Na tym zakończymy omawianie interakcji wyszukiwania z aktywnościami w Androidzie oraz mechanizmów korzystania z funkcji wyszukiwania. Dowiemy się teraz, w jaki sposób możemy nie tylko korzystać z procesu wyszukiwania, lecz również wpływać na to, jak przebiega. Zaimportujemy w tym celu prostego dostawcę propozycji wobec globalnego oraz lokalnego pola wyszukiwania.

## Implementacja prostego dostawcy propozycji

Mamy do czynienia z obszernym rozdziałem, jeśli więc Czytelnik dotychczas pracował z nim bez przerwy, warto chwilę odpocząć — czeka nas teraz kolejna duża porcja materiału, wymagająca całkowitego skupienia.

Zdążyliśmy wspomnieć, w jaki sposób wykorzystuje się dostawców propozycji do umożliwienia aplikacjom współudziału w procesie wyszukiwania globalnego. Przyszedł czas na zaprojektowanie i napisanie prostego dostawcy propozycji. Wystarczy nam w tym celu kilka wierszy kodu pochodzących z przygotowanego wcześniej dostawcy `SearchRecentSuggestionsProvider`, umieszczonego w zestawie Android SDK.

Rozpoczniemy od wyjaśnienia zasady działania aplikacji zawierającej prostego dostawcę propozycji. Zamieścimy listę plików stosowanych w procesie implementacji. Wspomniana lista powinna dać Czytelnikowi ogólne pojęcie na temat aplikacji oraz implementowanych w niej mechanizmów.

Podczas pisania dostawcy propozycji wykorzystywane są trzy główne składniki. Pierwszym z nich jest sam dostawca propozycji, który przekazuje propozycje procesowi wyszukiwania. Drugim składnikiem jest aktywność wyszukiwania, pobierająca zapytanie lub propozycję i przekształcająca je do wyników wyszukiwania. Trzeci element nosi nazwę metadanych wyszukiwania i jest zdefiniowany wewnątrz kontekstu aktywności wyszukiwania. Omówimy zadania każdego z wymienionych elementów i zademonstrujemy sposób ich implementacji w kodzie źródłowym.

Najpierw jednak utwórzmy plan naszej aplikacji prostego dostawcy propozycji.

## Planowanie prostego dostawcy propozycji

Działanie wynikowego dostawcy propozycji jest z góry określone, ponieważ planujemy jego dziedziczenie od dostawcy `SearchRecentSuggestionsProvider`.

Dostawca `SearchRecentSuggestionsProvider` umożliwia nam zapisywanie kwerend w czasie prezentowania aktywności wyszukiwania. Po ich zapisaniu przez aktywność wyszukiwania zostaną one przekazane polu QSB poprzez dostawcę propozycji w czasie wpisywania liter lub tekstu w polu QSB.

W pochodnym dostawcy treści po prostu inicjalizujemy bazowego dostawcę poprzez wskazanie fragmentów wyszukiwanego tekstu, które będą powtarzane. W tym przypadku mamy niewiele więcej pracy. Wstawimy także minimalistyczną wersję aktywności wyszukiwania, stanowiącej zwykły widok tekstu, dzięki czemu dowiemy się o wywołaniu tej aktywności. Wewnątrz tej aktywności zaprezentujemy metody służące do odczytywania i zapisywania kwerend, dzięki czemu będą one dostępne dla dostawcy wyszukiwania.

Naszym zadaniem po utworzeniu aplikacji będzie przejrzenie wcześniejszych kwerend, umieszczonych w globalnym oraz lokalnym polu QSB w postaci propozycji.

Zapoznamy się teraz z listą plików potrzebnych do zaimplementowania naszego projektu. Możemy również pobrać poszczególne pliki spod adresu zamieszczonego na końcu rozdziału.

## Pliki implementacji prostego dostawcy propozycji

Do podstawowych plików biorących udział w implementacji aplikacji dostawcy propozycji należą *SearchActivity.java*, *SimpleSuggestionProvider.java* oraz *searchable.xml* (metadane wyszukiwania). Jednak aby nasza przykładowa aplikacja działała poprawnie, potrzebne będą także inne pliki. Wymienimy je wszystkie wraz z krótkim opisem każdego z nich. W trakcie omawiania budowy aplikacji będziemy prezentować kod źródłowy poszczególnych plików.

Na pierwszy ogień pójdu pliki Java:

- ***SimpleSuggestionProvider.java*** — implementuje omawianego w tym podrozdziale dostawcę propozycji w procesie dziedziczenia po bazowym dostawcy propozycji, dostępnym w zestawie SDK (listing 23.15).
- ***SearchActivity.java*** — plik wymagany do działania dostawcy propozycji, otrzymujący poszukiwany tekst oraz przekazujący wyniki wyszukiwania. Klasa ta również zapewnia zapisywanie kwerend dla dostawcy propozycji (listing 23.17).
- ***SimpleMainActivity.java*** — aktywność wywołująca pole lokalnego wyszukiwania oraz zawierająca lokalne propozycje (listing 23.19).

Poniżej znajdują się odpowiednie pliki układu graficznego:

- ***main.xml*** — plik układu graficznego aktywności *SimpleMainActivity* (część listingu 23.19).
- ***/res/layout/layout\_search\_activity.xml*** — układ graficzny aktywności *SearchActivity* (część listingu 23.17).
- ***/res/values/strings.xml*** — pliki układu graficznego korzystają z obecnych tu definicji ciągów znaków (część listingu 23.19).

Tutaj znajduje się plik metadanych wyszukiwania.

- ***/xml/searchable.xml*** — poprzez ten plik aktywność wyszukiwania jest połączona z dostawcą propozycji (listing 23.18).

Oczywiście nie może zabraknąć pliku manifestu:

- ***AndroidManifest.xml*** — w tym pliku są zdefiniowane wszystkie składniki aplikacji (listing 23.16).

Jeżeli Czytelnik planuje komplikowanie tego projektu bezpośrednio poprzez kopiowanie i wklejanie kodu źródłowego z tej książki, radzimy dokonywać tego zgodnie z numeracją listingów. Alternatywnym rozwiązaniem jest pobranie projektów związanych z tym rozdziałem, umieszczonych pod adresem URL, który znajduje się na końcu rozdziału.

Rozpoczniemy analizę tych plików od implementacji klasy *SimpleSuggestionProvider*.

## Implementacja klasy SimpleSuggestionProvider

W tym projekcie prostego dostawcy propozycji klasa *SimpleSuggestionProvider* pełni rolę dostawcy propozycji poprzez dziedziczenie po klasie *SearchRecentSuggestionsProvider*. Przyjrzyjmy się najpierw zadaniom tego prostego dostawcy propozycji.

## Zadania prostego dostawcy propozycji

Ponieważ nasz prosty dostawca propozycji wywodzi się z klasy `SearchRecentSuggestionsProvider`, większość czynności jest przeprowadzana przez bazowego dostawcę. Aby przekazywać podpowiedzi do bazowego dostawcy, klasa prostego dostawcy propozycji musi zainicjalizować bazową klasę wraz z unikatowym uprawnieniem. Wynika to z faktu, że proces wyszukiwania w Androidzie wywołuje dostawcę propozycji na podstawie niepowtarzalnego identyfikatora URI dostawcy treści. Z kolei dostawcy treści w Androidzie są przywoływani za pomocą nazw ich domen, stanowiących ciągi znaków zwane uprawnieniami (w rozdziale 4. znajdziemy szczegółowe informacje dotyczące ciągów znaków uprawnień).

Po zaimplementowaniu dostawcy propozycji za pomocą takiego prostego wywołania bazowej klasy trzeba go skonfigurować w pliku manifeście jako standardowego dostawcę treści zawierającego uprawnienie. Następnie trzeba go powiązać (nie bezpośrednio, za pomocą pliku `searchable.xml`) z aktywnością wyszukiwania. Definicja aktywności wyszukiwania odnosi się do pliku `searchable.xml`, który z kolei wskazuje dostawcę propozycji.

Przeanalizujmy kod źródłowy tego dostawcy i sprawdźmy, jak się ma ten kod do części wymienionych zadań tej klasy.

## Pełny kod źródłowy klasy SimpleSuggestionProvider

Ponieważ dziedziczymy po klasie `SearchRecentSuggestionsProvider`, kod źródłowy prostego dostawcy treści nie będzie skomplikowany — taki jak przedstawiony na listingu 23.15.

---

**Listing 23.15.** Plik `SimpleSuggestionProvider.java`

---

```
//SimpleSuggestionProvider.java
public class SimpleSuggestionProvider
extends SearchRecentSuggestionsProvider {

    final static String AUTHORITY =
        "com.androidbook.search.simplesp.SimpleSuggestionProvider";
    final static int MODE =
        DATABASE_MODE_QUERIES | DATABASE_MODE_2LINES;

    public SimpleSuggestionProvider() {
        super();
        setupSuggestions(AUTHORITY, MODE);
    }
}
```

---

Warto zwrócić uwagę na kilka istotnych elementów na listingu 23.15.

1. Inicjalizacja nadzędnej klasy.
2. Konfiguracja bazowego dostawcy za pomocą uprawnienia i trybu (wskazującego, które fragmenty poszukiwanego tekstu mają zostać zapamiętane).

Ciąg znaków uprawnienia musi być niepowtarzalny oraz odpowiadać definicji jego dostawcy treści umieszczonej w pliku manifeście (kod pliku manifestu znajdziemy na listingu 23.16).

Przyjrzyjmy się trybowi bazowanemu, czyli drugiemu argumentowi metody `setupSuggestions()`.

## Tryby bazodanowe klasy SearchRecentSuggestionsProvider

Podstawą działania dostępnej w Androidzie klasy `SearchRecentSuggestionsProvider` jest przechowywanie i odtwarzanie kwerend z bazy danych w celu późniejszego ich wyświetlania jako propozycji wyszukiwania. Taka propozycja zawiera dwa ciągi znaków (rysunek 23.2). Jedynie pierwszy ciąg znaków jest obowiązkowy. Podczas używania klasy `SearchRecentSuggestionsProvider` do odtwarzania tych danych musimy określić, czy chcemy przechowywać jeden ciąg, czy dwa ciągi znaków.

Do tego celu służą dwa tryby (bitły trybu pracy) obsługiwane przez tego bazowego dostawcę propozycji. W obydwu przypadkach jest stosowany przedrostek:

`DATABASE_MODE_...`

Oto te tryby:

- `DATABASE_MODE_QUERIES` (wartość binarna 1),
- `DATABASE_MODE_2LINES` (wartość binarna 2).

Za pomocą pierwszego trybu określamy potrzebę przechowywania i wyświetlania tylko jednego ciągu znaków. Dzięki drugiemu trybowi dostawca propozycji może przechowywać dwa ciągi znaków. Pierwszym ciągiem znaków jest kwerenda, natomiast drugim — opis danej propozycji.

Aktywność `SearchActivity` zachowuje te ciągi znaków, jeśli zostanie wywołana w celu udzielenia odpowiedzi na kwerendę. W celu zachowania tych elementów klasa `SearchActivity` skorzysta z następującej metody (zostanie ona szczegółowo omówiona podczas analizy aktywności wyszukiwania):

```
public class SearchRecentSuggestions
{
    ...
    public void saveRecentQuery (String queryString, String line2);
    ...
}
```

### Uwaga!

Klasa `SearchRecentSuggestions` stanowi część pakietu SDK. Omówimy ją dokładniej podczas analizy aktywności wyszukiwania, umieszczonej na listingu 23.17.

Wartość `queryString` stanowi ciąg znaków wpisany przez użytkownika. Zostanie ona wyświetlona jako propozycja, a jeżeli użytkownik ją kliknie, ciąg ten zostanie przesłany do aktywności wyszukiwania (jako nowe zapytanie wyszukiwania).

Poniżej przedstawiamy opis dokumentacji Androida na temat argumentu `line2`:

*Jeżeli bieżący dostawca propozycji został skonfigurowany za pomocą trybu `DATABASE_MODE_2LINES`, można w tym argumencie przesłać drugą linię tekstu. Tekst ten zostanie wyświetlony mniejszą czcionką pod główną propozycją. W trakcie wpisywania tekstu na liście będą się ukazywać pasujące wyrażenia, obecne w obydwu wierszach propozycji. Jeżeli nie zostanie skonfigurowany tryb dwóch linii lub jeśli dana propozycja nie zawiera dodatkowego tekstu, można wstawić tu wartość null.*

W naszym przykładzie chcemy zachować zarówno kwerendę, jak i pomocniczy tekst, wyświetlany wraz z kwerendą w propozycji, a przynajmniej spróbujemy zaprezentować go w postaci dostawcy SSSP (ang. *Search Simple Suggestion Provider* — prosty dostawca propozycji wyszukiwania).

kiwania) pod propozycją. Kiedy więc propozycje generowane przez dostawcę zostaną wyświetcone w polu globalnego wyszukiwania, będziemy wiedzieć, która aplikacja jest odpowiedzialna za przeszukiwanie tekstu.

Aby zapewnić, że propozycja wraz z pomocniczym tekstem będą zapisywane, musimy, zgodnie z listingiem 23.15, ustawić dwa bity trybu pracy. Jeżeli zdefiniujemy wyłącznie jeden bit odpowiedzialny za zachowywanie dwóch wierszy, będzie się pojawiał wyjątek nieprawidłowego argumentu. Wśród bitów trybu pracy musi się znajdować przynajmniej bit DATABASE\_MODE\_QUERIES. W istocie należy zaimplementować bitową operację OR. Zatem tryby te są komplementarne i nie wykluczają się wzajemnie.

**Wskazówka** Informacje na temat domyślnego dostawcy treści można znaleźć pod adresem:  
<http://developer.android.com/reference/android/provider/SearchRecentSuggestions.html>.

Po utworzeniu kodu źródłowego naszego prostego dostawcy treści sprawdźmy, jak możemy go zarejestrować w pliku manifeście.

## Deklarowanie dostawcy propozycji w pliku manifeście

Ponieważ klasa SimpleSuggestionProvider jest zasadniczo dostawcą treści, musi zostać zarejestrowana w pliku manifeście. Zawartość tego pliku została pokazana na listingu 23.16. Najważniejsze fragmenty kodu zostały zaznaczone pogrubioną czcionką.

**Listing 23.16.** Plik manifest zawierający definicję dostawcy SimpleSuggestionProvider

---

```
//nazwa pliku: AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.search.simplesp"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"
        android:label="Prosty dostawca propozycji wyszukiwania (SSSP)">
        <activity android:name=".SimpleMainActivity"
            android:label="SSSP:Prosta aktywność główna">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    <!--
    ****
    * Kod związany z wyszukiwaniem: aktywność wyszukiwania
    ****
-->
        <activity android:name=".SearchActivity"
            android:label="SSSP: Aktywność wyszukiwania"
            android:launchMode="singleTop">
            <intent-filter>
                <action android:name="android.intent.action.SEARCH" />
```

---

```

        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
<meta-data android:name="android.app.searchable"
           android:resource="@xml/searchable" />
</activity>

<meta-data android:name="android.app.default_searchable"
           android:value=".SearchActivity" />

<provider android:name=".SimpleSuggestionProvider"
          android:authorities
          ="com.androidbook.search.simplesp.SimpleSuggestionProvider" />
</application>
<uses-sdk android:minSdkVersion="4" />
</manifest>
```

---

Zwróćmy uwagę na uprawnienie prostego dostawcy propozycji w pliku kodu źródłowego (listing 23.15) oraz w pliku manifeście (listing 23.16). W obydwu przypadkach jego wartość jest następująca:

`com.androidbook.search.simplesp.SimpleSuggestionProvider`

Pozostałymi sekcjami pliku manifestu zajmiemy się po omówieniu innych aspektów prostego dostawcy propozycji. Jak widać po pliku manifeście, kluczową rolę odgrywa aktywność wyszukiwania. Przyjrzyjmy się więc jej teraz uważniej. Drugą aktywnością, `SimpleMainActivity`, zajmiemy się pod koniec podrozdziału, ponieważ jest ona wyłącznie aktywnością sterującą, służącą tylko do uruchomienia aplikacji.

## Aktywność wyszukiwania dostępna w prostym dostawcy propozycji

Aktywność wyszukiwania jest wywoływana przez system (pole QSB) za pomocą ciągu znaków stanowiącego kwerendę. Aktywność ta musi z kolei odczytywać tę kwerendę z intencji i wykonywać odpowiednie czynności, a także, potencjalnie, wyświetlać wyniki.

Ponieważ mamy do czynienia z aktywnością, istnieje możliwość jej wywołania za pomocą innych intencji lub działań. Z tego powodu dobrym zwyczajem jest sprawdzanie intencji wywołującej tę aktywność. W naszym przypadku działaniem wywołującym aktywność jest `ACTION_SEARCH`.

W pewnych okolicznościach aktywność wyszukiwania może zostać samoistnie wywołana. W przypadku dużego prawdopodobieństwa wystąpienia takiej sytuacji należy zdefiniować tryb `singleTop` uruchamiania tej aktywności. Aktywność powinna również posiadać zdolność uruchamiania metody `onNewIntent()`. Ta kwestia zostanie poruszona w podpunkcie „Metody `onCreate()` i `onNewIntent()`”.

Każda czynność wykonywana na ciągu znaków kwerendy zostanie zapisana w dzienniku. Po zapisaniu kwerendy w dzienniku musimy ją zapisać w dostawcy `SearchRecentSuggestionsProvider`, aby w przypadku kolejnych wyszukiwań była wyświetlana jako jedna z propozycji.

Spójrzmy teraz na kod źródłowy klasy aktywności wyszukiwania.

## Pełny kod źródłowy aktywności wyszukiwania

Na listingu 23.17 został przedstawiony kod źródłowy klasy SearchActivity.

**Listing 23.17.** Aktywność wyszukiwania dla dostawcy SimpleSuggestionProvider

---

```
//nazwa pliku: SearchActivity.java
public class SearchActivity extends Activity
{
    private final static String tag = "SearchActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        Log.d(tag, "Jestem tworzona");
        //w przeciwnym wypadku wykonuje tę czynność
        setContentView(R.layout.layout_test_search_activity);
        //this.setDefaultCloseOperation(Activity.DEFAULT_KEYS_SEARCH_GLOBAL);
        this.setDefaultCloseOperation(Activity.DEFAULT_KEYS_SEARCH_LOCAL);

        // umieszcza tutaj i przetwarza kwerendę wyszukiwania
        final Intent queryIntent = getIntent();
        final String queryAction = queryIntent.getAction();
        if (Intent.ACTION_SEARCH.equals(queryAction))
        {
            Log.d(tag, "nowa intencja dla wyszukiwania");
            this.doSearchQuery(queryIntent);
        }
        else {
            Log.d(tag, "nowa intencja NIE dla wyszukiwania");
        }
        return;
    }

    @Override
    public void onNewIntent(final Intent newIntent)
    {
        super.onNewIntent(newIntent);
        Log.d(tag, "nowa intencja mnie wywołująca");

        // umieszcza tutaj i przetwarza kwerendę wyszukiwania
        final Intent queryIntent = getIntent();
        final String queryAction = queryIntent.getAction();
        if (Intent.ACTION_SEARCH.equals(queryAction))
        {
            this.doSearchQuery(queryIntent);
            Log.d(tag, "nowa intencja dla wyszukiwania");
        }
        else {
            Log.d(tag, "nowa intencja NIE dla wyszukiwania");
        }
    }
    private void doSearchQuery(final Intent queryIntent)
    {
        final String queryString =
```

```

        queryIntent.getStringExtra(SearchManager.QUERY);

    // zapisuje ciąg znaków kwerendy w bieżącym dostawcy propozycji kwerend.
    SearchRecentSuggestions suggestions = new SearchRecentSuggestions(this,
        SimpleSuggestionProvider.AUTHORITY,
        SimpleSuggestionProvider.MODE);
    suggestions.saveRecentQuery(queryString, "SSSP");
}

//Poniżej został umieszczony plik układu graficznego odpowiadający omawianej
//aktywności. Wytnijmy ten kod i wstawmy go do osobnego pliku
//układu graficznego. W komentarzu została wstawiona lokalizacja pliku.

<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/layout_search_activity.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Widok aktywności testującej wyszukiwanie"
    />
</LinearLayout>

```

Po zapoznaniu się z zawartością listingu 23.17 prześledźmy, w jaki sposób aktywność wyszukiwania sprawdza działanie i odczytuje ciąg znaków kwerendy.

## Sprawdzanie działania i odczytywanie kwerendy

Kod aktywności wyszukiwania sprawdza wywołujące ją działanie poprzez przeanalizowanie intencji wywołującej i porównanie jej ze stałą `intent.ACTION_SEARCH`. Jeżeli mamy do czynienia z takim samym działaniem, zostaje wywołana funkcja `doSearchQuery()`.

Z pomocą tej funkcji aktywność wyszukiwania odczytuje ciąg znaków kwerendy, używając dodatkowej intencji. Służy do tego celu kod:

```
final String queryString =
    queryIntent.getStringExtra(SearchManager.QUERY);
```

Zauważmy, że ta dodatkowa intencja została zdefiniowana jako `SearchManager.QUERY`. W tym rozdziale znajduje się wiele takich elementów dodatkowych, zdefiniowanych w odniesieniu do interfejsu API `SearchManager` (adres do dodatkowych materiałów dotyczących tego zakresu został umieszczony w podrozdziale „Odnośniki”).

## Metody `onCreate()` i `onNewIntent()`

Uruchomienie aktywności wyszukiwania następuje po wpisaniu przez użytkownika tekstu w polu wyszukiwania i kliknięciu propozycji lub strzałki nawigacji. W wyniku tego system tworzy

aktywność wyszukiwania i wywołuje jej metodę `onCreate()`. Intencja przekazana tej metodzie ustanowi działanie `ACTION_SEARCH`.

Czasem zdarza się, że aktywność nie zostaje utworzona, lecz są przekazywane nowe kryteria wyszukiwania poprzez metodę `onNewIntent()`. Jak to się dzieje? Metoda zwrotna `onNewIntent()` jest ściśle związana z trybem uruchamiania aktywności. Przeglądając się listingowi 23.16, możemy stwierdzić, że aktywność wyszukiwania otrzymuje wartość `singleTop` w pliku manifeście.

Aktywność skonfigurowana w trybie `singleTop` informuje system, aby nie tworzyć nowej aktywności, jeżeli znajdzie się ona na wierzchu stosu. W takim przypadku zamiast metody `onCreate()` zostaje wywołana metoda `onNewIntent()`. Właśnie dlatego w kodzie źródłowym aktywności (listing 23.17) intencja jest sprawdzana w dwóch miejscach.

## Testowanie metody `onNewIntent()`

Po zaimplementowaniu metody `onNewIntent()` stwierdzimy, że jest ona wywoływana w nie-standardowy sposób. Rodzi się następujące pytanie: kiedy aktywność wyszukiwania znajduje się na wierzchu stosu? To się zazwyczaj nie zdarza.

Wyjaśnijmy dlaczego. Założymy, że aktywność A wywołuje proces wyszukiwania, wskutek czego pojawia się aktywność wyszukiwania B. Aktywność B powoduje wyświetlenie wyników, a użytkownik wraca do poprzedniego ekranu za pomocą przycisku powrotu. W tym czasie aktywność B, stanowiąca naszą aktywność wyszukiwania, ustępuje miejsca na szczytce stosu aktywności A. Użytkownik może ewentualnie wcisnąć przycisk powrotu do ekranu startowego i skorzystać z wyszukiwania globalnego, co spowoduje umieszczenie aktywności ekranu startowego na wierzchu stosu.

Aktywność wyszukiwania można umieścić na wierzchu stosu w następujący sposób: powiedzmy, że w efekcie wyszukiwania wynikiem aktywności A jest aktywność B. Jeżeli aktywność B definiuje tryb *type-to-search*, to po przejściu do tej aktywności zostanie ona ponownie wywołana, z nowymi kryteriami. Listing 23.17 ukazuje sposób konfiguracji trybu *type-to-search*. Oto odpowiedni fragment kodu:

```
this.setDefaultKeyMode(Activity.DEFAULT_KEYS_SEARCH_LOCAL);
```

## Zapisywanie kwerendy za pomocą dostawcy `SearchRecentSuggestionsProvider`

Stwierdziliśmy, że istnieje konieczność zachowywania napotkanych kwerend przez aktywność wyszukiwania w celu wyświetlania ich w formie propozycji za pomocą dostawcy. Poniżej przedstawiamy segment kodu odpowiedzialny za zapisywanie kwerend:

```
final String queryString =
    queryIntent.getStringExtra(SearchManager.QUERY);

// Zapisuje ciąg znaków kwerendy w bieżącym dostawcy propozycji kwerend.
SearchRecentSuggestions suggestions = new SearchRecentSuggestions(this,
    SimpleSuggestionProvider.AUTHORITY,
    SimpleSuggestionProvider.MODE);
suggestions.saveRecentQuery(queryString, "SSSP");
```

Powyższy kod udowadnia, że Android przekazuje informację kwerendy w postaci parametru `EXTRA` (`SearchManager.Query`) poprzez intencję.

Po wprowadzeniu kwerendy można wykorzystać klasę zestawu SDK SearchRecentSuggestions do zachowania kwerendy oraz podpowiedź ("SSSP") poprzez utworzenie i zachowanie nowej propozycji. Ponieważ skorzystaliśmy z trybu zachowywania dwóch wierszy oraz trybu kwerendy, wartość drugiego argumentu metody saveRecentQuery wynosi SSSP (prosty dostawca propozycji wyszukiwania). Tekst zawarty w tym argumencie będzie się pojawiał pod propozycjami generowanymi przez dostawcę.

Zajmiemy się teraz definicją metadanych wyszukiwania, łączącą aktywność wyszukiwania z dostawcą propozycji wyszukiwania.

## Metadane wyszukiwania

Definiowanie wyszukiwania w Androidzie rozpoczyna się od aktywności wyszukiwania. Definiujemy ją najpierw w pliku manifeście. Częścią jej definicji jest określenie miejsca, w którym znajdzie się plik XML metadanych wyszukiwania. Przyjrzyjmy się listingowi 23.16, w którym zdefiniowaliśmy aktywność wyszukiwania wraz ze ścieżką do pliku metadanych (*searchable.xml*).

Listing 23.18 przedstawia plik metadanych wyszukiwania wykorzystywany przez naszą aplikację.

**Listing 23.18.** Metadane wyszukiwania dla dostawcy SimpleSuggestionProvider

---

```
<!-- nazwa pliku: /res/xml/searchable.xml -->
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/search_label"
    android:hint="@string/search_hint"
    android:searchMode="showSearchLabelAsBadge"
    android:queryAfterZeroResults="true"

    android:includeInGlobalSearch="true"
    android:searchSuggestAuthority=
        "com.androidbook.search.simplesp.SimpleSuggestionProvider"
    android:searchSuggestSelection=" ? "
/>
```

---

Przeanalizujmy kluczowe atrybuty widoczne na listingu 23.18.

Atrybut `includeInGlobalSearch` wskazuje, aby używać tego dostawcy jako jednego ze źródeł globalnego pola QSB.

Atrybut `searchSuggestAuthority` określa zdefiniowane w pliku manifeście uprawnienie tego dostawcy propozycji (listing 23.16).

Atrybut `queryAfterZeroResults` służy do określania, czy pole QSB ma wysyłać więcej liter do dostawcy propozycji, w przypadku gdy bieżący zbiór znaków nie pozwolił na otrzymanie żadnych wyników. Ponieważ znajdujemy się na etapie testowania i nie chcemy pomijać żadnej funkcji, przydzielimy temu atrybutowi wartość `true`, dzięki czemu dostawca będzie reagował przy każdej nadarzającej się okazji.

Kolejny atrybut, `searchSuggestSelection`, przyjmuje zawsze wartość `?`, w przypadku gdy wiodzi się z poprzedniego dostawcy propozycji wyszukiwania. Taki ciąg znaków jest przekazywany dostawcy propozycji jako wartość `selection` (klauzula `where`) metody `query` dostawcy treści. Zazwyczaj w ten sposób jest reprezentowana klauzula `where` przechodząca do instrukcji wyboru dowolnego dostawcy treści.

Charakterystyczny dla dostawców propozycji jest fakt, że w przypadku posiadania wartości dla argumentu `searchSuggestSelection` (jako protokołu) Android przekazuje wartość kwerendy wyszukiwania (wprowadzaną w polu QSB) jako pierwszy wpis w tablicy argumentów wyboru będącej częścią metody danego dostawcy treści.

Kod przeprowadzający te czynności (mechanizmy definiujące wewnętrzny sposób wykorzystywania ciągów znaków przez dostawcę) jest ukryty w poprzednim dostawcy propozycji wyszukiwania, więc nie mamy możliwości przedstawienia sposobu używania tych argumentów w metodzie `query` dostawcy treści.

Szczegółowo zajmiemy się tym nieco dalej, gdy dokładnie przedstawimy znaczenie wartości `?`. W rzeczywistości jest dość mało prawdopodobne, że ta wartość jest w ogóle używana do zawężania wyników, ponieważ żadne pole nie jest kwalifikowane w następujący sposób: jakiś `→identyfikator = ?`. Natomiast istnieje możliwość, że jej wyraźna obecność zachęca system do przekazywania zawartości pola QSB w postaci pierwszego argumentu do dostawcy. Do tego umieszczony w zestawie SDK dostawca propozycji wyszukiwania polega wyłącznie na tym protokole podczas odczytywania wartości pola QSB w postaci dopasowanej tablicy, generowanej przez listę argumentów wyboru z metody `query()`.

Zajmijmy się teraz aktywnością wywołania wyszukiwania, która posłuży nam jako główny punkt wyjścia dla tej aplikacji. Aplikacja ta umożliwi nam przetestowanie wyszukiwania lokalnego.

## Aktywność wywołania wyszukiwania

Aktywność ta pozwala nam na wywołanie wyszukiwania lokalnego w czasie, gdy jest obsługiwana na pierwszym planie. Listing 23.19 przedstawia kod źródłowy takiej aktywności wywołania wyszukiwania, jej układ graficzny oraz plik `strings.xml`, będące częścią projektu.

**Listing 23.19.** Dostawca SimpleSuggestionProvider — główna aktywność

---

```
public class SimpleMainActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}

//nazwa pliku: /res/layout/main.xml
//Skopiujmy poniższy kod XML do pliku main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
```

```

    android:layout_height="wrap_content"
    android:text="@string/main_activity_text"
  />
</LinearLayout>

//nazwa pliku: /res/values/strings.xml
//Skopiujmy poniższy kod XML do pliku strings.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="main_activity_text">
    Jest to prosta aktywność. Kliknij przycisk wyszukiwania,
    aby przywołać proces przeszukiwania lokalnego.
    \n\n
    Dostawca propozycji będzie również uczestniczył w
    procesie wyszukiwania globalnego. Kiedy uruchomimy
    tę aplikację z poziomu wyszukiwania globalnego, nie
    ujrzymy niniejszego widoku, lecz bezpośrednio
    widok klasy SearchActivity.
  </string>

  <string name="search_activity_text">
    Jeżeli widzimy tę aktywność, zostaliśmy tutaj skierowani
    przez proces wyszukiwania globalnego lub lokalnego.
    \n\n
    Aktywność ta uruchamia również funkcję type-to-search. Prezentuje ona
    także koncepcje trybu singletop/new intencji.
  </string>

  <string name="app_name">Prosty dostawca propozycji</string>
  <string name="search_label">Demonstracja wyszukiwania lokalnego</string>
  <string name="search_hint">Podpowiedź wyszukiwania lokalnego</string>
</resources>
```

Jeśli spojrzymy na definicję tej aktywności w pliku manifeście (listing 23.16), stwierdzimy, że klasa SearchActivity nie została jawnie zdefiniowana dla tej aktywności jako domyślne wyszukiwanie lokalne. Wynika to z faktu, że tę specyfikację zastosowaliśmy na poziomie aplikacji, a nie aktywności, wprowadzając w pliku manifeście następujący fragment kodu:

```
<meta-data android:name="android.app.default_searchable"
          android:value=".SearchActivity" />
```

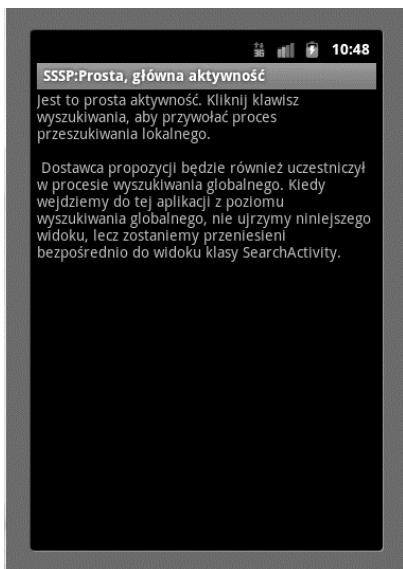
Zwróćmy uwagę, że powyższy fragment został umieszczony w pliku manifeście poza aktywnościami (listing 23.16). Dzięki tej specyfikacji Android „wie”, że dla wszystkich aktywności zawartych w tej aplikacji, w tym dla samej aktywności SearchActivity, domyślną aktywnością wyszukiwania jest SearchActivity. Można wykorzystać tę wiedzę do wywołania metody `onNewIntent()`, klikając przycisk wyszukiwania podczas sprawdzania wyników w klasie `SearchActivity`. W przypadku zdefiniowania domyślnego wyszukiwania jedynie na poziomie prostej aktywności wywołania wyszukiwania, a nie na poziomie całej aplikacji, wspomniana technika nie ma prawa bytu.

## Użytkowanie prostego dostawcy propozycji

Podczas przygotowywania tego programu musimy się upewnić, że uprawnienie naszego dostawcy propozycji jest takie samo w następujących plikach:

- *AndroidManifest.xml*,
- *searchable.xml*,
- *SimpleSuggestionProvider.java*.

Po uruchomieniu aplikacji ujrzymy jej ekran startowy, wyglądający tak jak na rysunku 23.20 (jest to nasza główna aktywność wywołania wyszukiwania).



**Rysunek 23.20.** Prosty dostawca propozycji — główna aktywność (ustawiona dla wyszukiwania lokalnego)

Jeżeli klikniemy przycisk wyszukiwania w trakcie działania aktywności na pierwszym planie, zostanie wywołany proces wyszukiwania lokalnego, widoczny na rysunku 23.21.

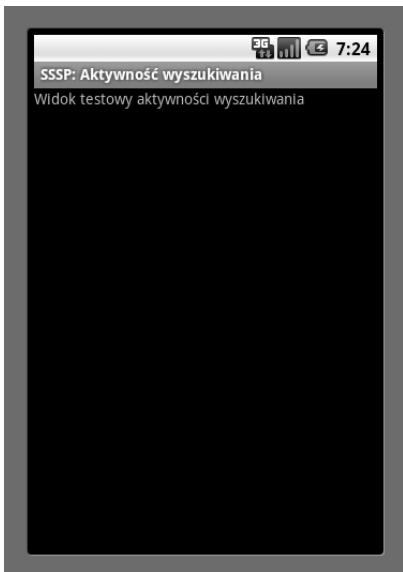
Na rysunku 23.21 nie widać żadnych propozycji, ponieważ jak na razie niczego nie szukaliśmy. Natomiast na podstawie etykiety i podpowiedzi, zdefiniowanych w pliku XML metadanych wyszukiwania, dowiadujemy się, że mamy do czynienia z wyszukiwaniem lokalnym.

Przejdzmy do następnego etapu i poszukajmy ciągu znaków `test1`. Zostaniemy wyświetlony ekran *Aktywność wyszukiwania*, przedstawiony na rysunku 23.22.

Jak wynika z widocznego na listingu 23.17 kodu źródłowego klasy `SearchActivity`, jej działanie na ekranie nie jest widoczne, jednak niedostrzegalnie zachowuje ona kwerendy w bazie danych. Jeśli teraz wrócimy do głównego ekranu (wciskając przycisk cofania) i ponownie wywołamy wyszukiwanie, ujrzymy ekran, widoczny na rysunku 23.23, na którym propozycje wyszukiwania zostały zapisane wcześniejszym wpisem kwerendy. Na tym rysunku widzimy również podpowiedź SSSP znajdującą się pod propozycją. Może się to wydawać w tym miejscu nieistotne,



Rysunek 23.21. Prosty dostawca propozycji — pole wyszukiwania lokalnego



Rysunek 23.22. Prosty dostawca propozycji — aktywność wyświetlająca wyniki wyszukiwania lokalnego

ponieważ mamy do czynienia z wyszukiwaniem lokalnym i wyraźnie widać, że propozycja wywodzi się z naszej aplikacji. Jednak podpowiedź ta pozwoli nam odróżnić element `test1` od innych elementów o podobnej nazwie w procesie wyszukiwania globalnego.

Nadarza się teraz dobra sposobność wywołania metody `onNewIntent()`. Na ekranie aktywności wyszukiwania (rysunek 23.22) możemy wpisać na przykład literę `t`, w wyniku czego zostanie wywołane wyszukiwanie za pomocą trybu *type-to-search*, a wywołanie metody `onNewIntent()` zostanie zapisane w pliku dziennika.



**Rysunek 23.23.** Prosty dostawca propozycji — odczytana propozycja lokalna

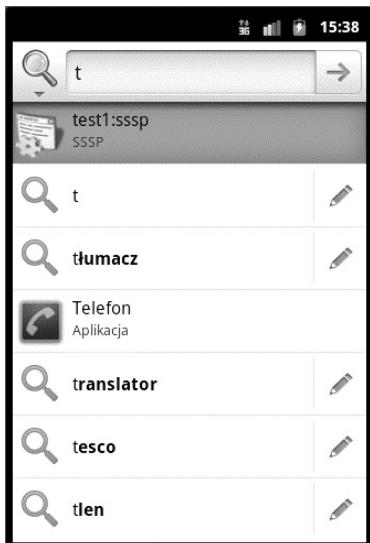
Zobaczmy, co należy zrobić, aby nasze propozycje zostały wyświetlane w polu wyszukiwania globalnego. Ponieważ zamieściliśmy obiekt `includeInGlobalSearch` w pliku `searchable.xml`, propozycje te powinny być widoczne również w przypadku wyszukiwania globalnego. Musimy jednak przedtem zezwolić tej aplikacji na obsługę globalnych propozycji wyszukiwania, co zostało zaprezentowane na rysunku 23.24.



**Rysunek 23.24.** Włączanie prostego dostawcy propozycji wyszukiwania

Na początku rozdziału podaliśmy instrukcję wyświetlania tego ekranu. Utworzony przez nas prosty, niestandardowy dostawca propozycji znajduje się teraz na liście aplikacji pozwalających na wyszukiwanie, pod nazwą `SSSP: Aktywność wyszukiwania`. Ciąg znaków definiujący tę nazwę wywodzi się od nazwy aktywności `SearchActivity` (listing 23.16).

Po zaznaczeniu utworzonego dostawcy umożliwimy współpracę wyszukiwania globalnego, pokazanego na rysunku 23.25, z tym dostawcą propozycji.



**Rysunek 23.25.** Propozycje globalne pochodzące od prostego dostawcy

Jeżeli w polu wyszukiwania globalnego wprowadzimy na przykład literę t, zobaczymy propozycje pochodzące od omawianego w tym podrozdziale dostawcy. Gdy w trybie wyszukiwania globalnego poszukujemy określonego elementu, zauważymy aktywność wyszukiwania lokalnego, zilustrowaną na rysunku 23.22.

Na tym zakończymy analizę prostych dostawców propozycji. Dowiedzieliśmy się, w jaki sposób można wykorzystać wbudowaną klasę `SearchRecentSuggestionsProvider` do zapamiętywania zapytań swoistych dla danej aplikacji. Korzystając z tej techniki, możemy nawet wprowadzać propozycje lokalne do kontekstu globalnego.

Jednak dzięki temu prostemu ćwiczeniu nie pokazaliśmy sposobu pisania dostawów propozycji od podstaw. Co ważniejsze, nie przedstawiliśmy jeszcze jakichkolwiek informacji o sposobie przekazywania zestawu propozycji przez dostawcę. Nie powiedzieliśmy, jakie kolumny są dostępne w takim zestawie. Aby zrozumieć te oraz inne kwestie, trzeba zaimplementować od podstaw własnego dostawcę propozycji.

## Implementacja niestandardowego dostawcy propozycji

Technologia wyszukiwania w Androidzie jest zbyt elastyczna, aby nie można było jej dostosować do własnych potrzeb. Ponieważ w poprzednim podrozdziale pokazaliśmy, jak korzystać z wbudowanego dostawcy propozycji, wiele funkcji w klasie `SearchRecentSuggestionsProvider` pozostało ukrytych i nieomówionych. Przeanalizujemy te pominięte szczegóły, implementując niestandardowego dostawcę treści, nazwanego `SuggestUrlProvider`.

Rozpoczniemy od wyjaśnienia mechanizmu działania tego dostawcy. Następnie podamy listę plików potrzebnych do jego implementacji. Lista ta powinna dać Czytelnikowi ogólne pojęcie na temat procesu budowania własnego dostawcy propozycji.

Na koniec pokażemy zastosowanie utworzonej aplikacji. Zaczynajmy.

## Implementacja niestandardowego dostawcy propozycji

Nasz niestandardowy dostawca propozycji będzie nosił nazwę `SuggestUrlProvider`. Głównym zadaniem tego dostawcy jest monitorowanie tekstu wpisywanego w polu QSB. Jeżeli kwerendę wyszukiwania stanowi tekst wyglądający jak na przykład `great.m` (sufiks `.m` jest skrótem od wyrazu „znaczenie”, z ang. *meaning*), dostawca zinterpretuje pierwszą część zapytania jako słowo i zaproponuje wywołanie internetowego adresu URL, umożliwiającego wyszukanie znaczenia tego słowa.

Dla każdego wyrazu są proponowane dwa adresy URL. Pierwszy adres pozwala użytkownikowi na wyszukanie słowa za pomocą witryny <http://www.thefreedictionary.com>, drugi uruchamia stronę <http://www.google.com>. Wybranie jednej z propozycji powoduje uruchomienie przeglądarki z otwartą witryną. Jeżeli użytkownik kliknie ikonę wyszukiwania w polu QSB, aktywność wyszukiwania utworzy wpis tej kwerendy w dzienniku, znajdującym się w prostym układzie graficznym tej aktywności. Stanie się to bardziej zrozumiałe po zaprezentowaniu odpowiednich zrzutów ekranu.

Przyjrzyjmy się liście plików tworzących nasz projekt. Za pomocą adresu URL zamieszczonego na końcu rozdziału możemy również pobrać plik ZIP zawierający gotowy projekt.

## Pliki wymagane do implementacji projektu `SuggestUrlProvider`

Dwoma głównymi plikami są `SearchActivity.java` i `SuggestUrlProvider.java`, jednak do poprawnego działania projektu wymagana będzie implementacja kilku dodatkowych plików. Poniżej przedstawiamy ich listę wraz z krótkim opisem. W dalszej części rozdziału zamieściliśmy kod źródłowy każdego z nich.

- **`SuggestUrlProvider.java`** — w tym pliku zostaje zaimplementowany protokół niestandardowego dostawcy propozycji. W naszym przypadku dostawca ten tłumaczy ciągi znaków kwerend na słowa i zwraca parę propozycji za pomocą kurSORA propozycji (listing 23.20).
- **`SearchActivity.java`** — aktywność ta jest odpowiedzialna za odbieranie kwerend lub propozycji dostarczanych przez dostawcę propozycji. Definicja aktywności `SearchActivity` ma również za zadanie powiązanie dostawcy propozycji z tą aktywnością (listing 23.23).
- **`layout/layout_search_activity.xml`** — ten plik układu graficznego jest używany opcjonalnie przez klasę `SearchActivity`. W naszym przykładzie służy on do umieszczenia wysłanej kwerendy we wpisie dziennika (listing 23.24).
- **`values/strings.xml`** — zawiera definicje ciągów znaków układu graficznego, tytułu i podpowiedź wyszukiwania lokalnego itp. (listing 23.25).

- ***xml/searchable.xml*** — plik XML metadanych wyszukiwania łączący klasę `SearchActivity`, dostawcę propozycji i pole QSB (listing 23.21).
- ***AndroidManifest.xml*** — plik manifest aplikacji, w którym są definiowane aktywność wyszukiwania i dostawca propozycji. Tutaj również określamy, że klasa `SearchActivity` będzie wywoływana wobec naszej aplikacji jako wyszukiwanie lokalne (listing 23.27).

Zaczniemy najpierw od analizy dostawcy `SuggestUrlProvider`.

## Implementacja klasy `SuggestUrlProvider`

W przypadku naszego projektu niestandardowego dostawcy propozycji klasa `SuggestUrlProvider` zapewnia implementację protokołu dostawcy propozycji. Badanie implementacji tej klasy rozpoczęmy od analizy jej zadań.

### Zadania dostawcy propozycji

W swej istocie dostawca propozycji jest dostawcą treści. Podobnie jak dostawca treści jest on wywoływany przez proces wyszukiwania w Androidzie za pomocą identyfikatora URI określającego dostawcę oraz dodatkowego argumentu reprezentującego kwerendę.

W Androidzie do wywoływania dostawcy propozycji są wykorzystywane dwa rodzaje identyfikatorów URI. Pierwszy z nich nosi nazwę **identyfikatora URI wyszukiwania**. Służy on do zbierniania zestawu propozycji. Odpowiedzią musi być co najmniej jeden wiersz, zawierający zbiór znanych kolumn.

Drugi identyfikator URI nosi nazwę **identyfikatora URI propozycji**. Pozwala on na aktualizowanie propozycji przechowywanej w pamięci podręcznej. Odpowiedzią musi być pojedynczy wiersz składający się z grupy znanych kolumn.

Dostawca propozycji musi również określić w metadanych wyszukiwania (plik `searchable.xml`) sposób otrzymywania fragmentu kwerendy wyszukiwania, także w trakcie wpisywania danych. Można tego dokonać za pomocą argumentu `select` metody `query` lub za pomocą ostatniego segmentu samego identyfikatora URI (również przekazywanego w postaci jednego z argumentów do metody kwerendy danego dostawcy).

Dostępna jest duża liczba kolumn dla dostawcy treści, z których każda uruchamia określone działanie wyszukiwania. Dostawca musi najpierw określić zbiór przekazywanych kolumn sterujących. Poniżej wymieniamy niektóre spośród kolumn tego typu:

- Kolumny włączające i wyłączające zapis w pamięci podręcznej propozycji zwracanych procesowi wyszukiwania w Androidzie.
- Kolumny kontrolujące proces przepisywania tekstu propozycji do pola kwerendy.
- Kolumny służące do bezpośredniego wywołania działania, w przeciwnieństwie do pokazywania zestawu wyników po kliknięciu propozycji przez użytkownika.

## Kod źródłowy dostawcy `SuggestUrlProvider`

Listing 23.20 przedstawia kod źródłowy klasy `SuggestUrlProvider`. W dalszej części rozdziału, podczas omawiania każdego z zadań dostawcy, szczegółowo przeanalizujemy poszczególne fragmenty kodu.

**Listing 23.20.** Kod źródłowy niestandardowego dostawcy propozycji

```
public class SuggestUrlProvider extends ContentProvider
{
    private static final String tag = "SuggestUrlProvider";
    public static String AUTHORITY =
        "com.androidbook.search.custom.suggesturlprovider";

    private static final int SEARCH_SUGGEST = 0;
    private static final int SHORTCUT_REFRESH = 1;
    private static final UriMatcher sURIMatcher = buildUriMatcher();

    private static final String[] COLUMNS = {
        "_id", // musi zawrzeć tę kolumnę
        SearchManager.SUGGEST_COLUMN_TEXT_1,
        SearchManager.SUGGEST_COLUMN_TEXT_2,
        SearchManager.SUGGEST_COLUMN_INTENT_DATA,
        SearchManager.SUGGEST_COLUMN_INTENT_ACTION,
        SearchManager.SUGGEST_COLUMN_SHORTCUT_ID
    };

    private static UriMatcher buildUriMatcher()
    {
        UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
        matcher.addURI(AUTHORITY,
            SearchManager.SUGGEST_URI_PATH_QUERY,
            SEARCH_SUGGEST);
        matcher.addURI(AUTHORITY,
            SearchManager.SUGGEST_URI_PATH_QUERY +
            "/*",
            SEARCH_SUGGEST);
        matcher.addURI(AUTHORITY,
            SearchManager.SUGGEST_URI_PATH_SHORTCUT,
            SHORTCUT_REFRESH);
        matcher.addURI(AUTHORITY,
            SearchManager.SUGGEST_URI_PATH_SHORTCUT +
            "/*",
            SHORTCUT_REFRESH);
        return matcher;
    }

    @Override
    public boolean onCreate() {
        //nie robimy tu nic szczególnego
        Log.d(tag,"wywołana metoda onCreate");
        return true;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder)
    {
        Log.d(tag,"kwerenda wywołana za pomocą identyfikatora uri:" + uri);
        Log.d(tag,"wybór:" + selection);

        String query = selectionArgs[0];
        Log.d(tag,"kwerenda:" + query);

        switch (sURIMatcher.match(uri)) {
```

```
case SEARCH_SUGGEST:
    Log.d(tag, "wywołana propozycja wyszukiwania");
    return getSuggestions(query);
case SHORTCUT_REFRESH:
    Log.d(tag, "wywołane odświeżenie skrotu");
    return null;
default:
    throw new IllegalArgumentException("Nieznany adres URL " + uri);
}

private Cursor getSuggestions(String query)
{
    if (query == null) return null;
    String word = getWord(query);
    if (word == null)
        return null;

    Log.d(tag, "kwerenda przekracza długosc 3 liter");

    MatrixCursor cursor = new MatrixCursor(COLUMNS);
    cursor.addRow(createRow1(word));
    cursor.addRow(createRow2(word));
    return cursor;
}
private Object[] createRow1(String query)
{
    return columnValuesOfQuery(query,
        "android.intent.action.VIEW",
        "http://www.thefreedictionary.com/" + query,
        "Wyszukaj na stronie freedictionary.com wyraz",
        query);
}

private Object[] createRow2(String query)
{
    return columnValuesOfQuery(query,
        "android.intent.action.VIEW",
        "http://www.google.com/search?hl=en&source=hp&q=define%3A/" +
        query,
        "wyszukaj na stronie google.com wyraz",
        query);
}
private Object[] columnValuesOfQuery(String query,
    String intentAction,
    String url,
    String text1,
    String text2)
{
    return new String[] {
        query, // _id
        text1, // text1
        text2, // text2
        url,
        // intent_data (umieszczony po kliknięciu elementu)
        intentAction, // działanie
    };
}
```

```
        SearchManager.SUGGEST_NEVER_MAKE_SHORTCUT
    );
}

private Cursor refreshShortcut(String shortcutId, String[] projection) {
    return null;
}

public String getType(Uri uri) {
    switch (sURIMatcher.match(uri)) {
        case SEARCH_SUGGEST:
            return SearchManager.SUGGEST_MIME_TYPE;
        case SHORTCUT_REFRESH:
            return SearchManager.SHORTCUT_MIME_TYPE;
        default:
            throw new IllegalArgumentException("Nieznany adres URL " + uri);
    }
}

public Uri insert(Uri uri, ContentValues values) {
    throw new UnsupportedOperationException();
}

public int delete(Uri uri, String selection, String[] selectionArgs) {
    throw new UnsupportedOperationException();
}

public int update(Uri uri, ContentValues values, String selection,
    String[] selectionArgs) {
    throw new UnsupportedOperationException();
}

private String getWord(String query)
{
    int dotIndex = query.indexOf('.');
    if (dotIndex < 0)
        return null;
    return query.substring(0,dotIndex);
}
```

---

## Identyfikatory URI dostawcy propozycji

Po zaprezentowaniu pełnego kodu źródłowego niestandardowego dostawcy propozycji warto się przyjrzeć, w jaki sposób fragmenty tego kodu spełniają zadania związane z identyfikatorami URI.

Najpierw zobaczymy, jaki jest format identyfikatorów URI wykorzystywanych do wywoływania dostawcy propozycji. Jeżeli nasz dostawca propozycji posiada uprawnienie:

com.androidbook.search.custom.suggesturlprovider

to Android będzie wysyłał dwa możliwe identyfikatory URI. Pierwszy typ — identyfikator URI wyszukiwania — wygląda następująco:

content://com.androidbook.search.suggesturlprovider/search\_suggest\_query

albo

```
content://com.androidbook.search.suggesturlprovider/search_suggest_query/<kwerenda>
```

Identyfikator tego typu jest nadawany na początku procesu wpisywania tekstu w polu QSB. W jednej z odmian tego identyfikatora kwerenda jest przekazywana na jego końcu jako dodatkowy element (segment ścieżki). Możliwość dołączania kwerendy jako segmentu ścieżki jest definiowana w pliku metadanych wyszukiwania *searchable.xml*. Powrócimy do tego tematu podczas szczegółowego omówienia metadanych wyszukiwania.

Drugi rodzaj identyfikatora URI jest przeznaczony dla dostawcy propozycji i jest związany ze skrótami wyszukiwania. Skróty wyszukiwania w Androidzie są propozycjami (rysunek 23.3), które system przechowuje w pamięci podręcznej, zamiast wywoływać dostawcę propozycji w celu uzyskania nowej treści. Tematykę skrótów wyszukiwania poruszmy podczas analizowania kolumn propozycji. Na razie wystarczy wiedzieć, że drugi rodzaj identyfikatora propozycji przybiera następujące kształty:

```
content://com.androidbook.search.suggesturlprovider/search_suggest_shortcut
```

lub

```
content://com.androidbook.search.suggesturlprovider/search_suggest_shortcut/
-><identyfikator-skrótu>
```

Identyfikator tego typu jest nadawany przez system podczas próby określenia ważności skrótów przechowywanych w pamięci podręcznej. Taki identyfikator URI nosi nazwę identyfikatora URI skrótu. Jeżeli dostawca przekaże pojedynczy wiersz, bieżący skrót zostanie zastąpiony nowym. Jeżeli zostanie przesłana wartość null, bieżąca propozycja przestanie być uznawana za ważną.

Klasa *SearchManager* definiuje w Androidzie dwie stałe, pozwalające na odróżnianie tych segmentów identyfikatorów URI (*search\_suggest\_search* i *search\_suggest\_shortcut*). Są to odpowiednio:

```
SearchManager.SUGGEST_URI_PATH_QUERY
SearchManager.SUGGEST_URI_PATH_SHORTCUT
```

Zadaniem dostawcy propozycji jest rozpoznawanie tych identyfikatorów, przychodzących w metodzie *query()*. Na listingu 23.20 został zaprezentowany sposób przeprowadzania tej czynności za pomocą klasy *UriMatcher* (zastosowanie klasy *UriMatcher* zostało szczegółowo omówione w rozdziale 5.).

## Implementacja metody *getType()* i określenie typów MIME

Ponieważ dostawca propozycji jest ogólnie dostawcą treści, ma zaimplementować kontrakt definiujący implementację metody *getType()*.

W naszym przypadku implementacja metody *getType()* została ukazana na listingu 23.20. Poniżej przypominamy odpowiedni fragment kodu:

```
public String getType(Uri uri) {
    switch (sURIMatcher.match(uri)) {
        case SEARCH_SUGGEST:
            return SearchManager.SUGGEST_MIME_TYPE;
        case SHORTCUT_REFRESH:
            return SearchManager.SHORTCUT_MIME_TYPE;
        default:
            throw
```

```
        new IllegalArgumentException("Nieznany adres URL" + uri);
    }
}
```

W strukturze wyszukiwania poprzez klasę SearchManager wprowadzono parę stałych, wspomagających przetwarzanie takich typów MIME. Tymi typami MIME są:

```
SearchManager.SUGGEST_MIME_TYPE  
SearchManager.SHORTCUT_MIME_TYPE
```

Są one tłumaczone na wyrażenia:

```
vnd.android.cursor.dir/vnd.android.search.suggest  
vnd.android.cursor.item/vnd.android.search.suggest
```

## Przekazywanie kwerendy do dostawcy propozycji — argument Selection

Ostatnim etapem zastosowania któregoś z identyfikatorów URI do wywołania dostawcy jest wywołanie metody query() dostawcy propozycji w celu uzyskania kurSORA propozycji. Jeżeli przyjrzymy się implementacji metody query() z listingu 23.20, to zauważymy, że do sformułowania i przekazania kurSORA używamy argumentów selection oraz selectionArgs. Dla przypomnienia poniżej wstawiliśmy omawiany fragment kodu:

```
public Cursor query(Uri uri, String[] projection,  
                     String selection,  
                     String[] selectionArgs, String sortOrder)  
{  
    Log.d(tag, "kwerenda wywołana za pomocą identyfikatora uri:" + uri);  
    Log.d(tag, "wybor:" + selection);  
  
    String query = selectionArgs[0];  
    Log.d(tag, "kwerenda:" + query);  
  
    switch (sURIMatcher.match(uri)) {  
        case SEARCH_SUGGEST:  
            Log.d(tag, "wywołana propozycja wyszukiwania");  
            return getSuggestions(query);  
        case SHORTCUT_REFRESH:  
            Log.d(tag, "wywołane odświeżenie skrotu");  
            return null;  
        default:  
            throw  
                new IllegalArgumentException("Nieznany adres URL " + uri);  
    }  
}
```

Aby zrozumieć, jakie dane są przekazywane za pomocą tych dwóch argumentów (selection oraz selectionArgs), musimy spojrzeć na plik metadanych wyszukiwania — *searchable.xml*. Na listingu 23.21 został przedstawiony kod tego pliku.

---

**Listing 23.21.** Metadane wyszukiwania dla niestandardowego dostawcy propozycji

```
//xml/searchable.xml  
<searchable xmlns:android="http://schemas.android.com/apk/res/android"  
           android:label="@string/search_label"  
           android:hint="@string/search_hint"
```

```

    android:searchMode="showSearchLabelAsBadge"
    android:searchSettingsDescription="proponuje adresy url"
    android:includeInGlobalSearch="true"
    android:queryAfterZeroResults="true"

    android:searchSuggestAuthority="com.androidbook.search.custom.suggesturlprovider"

    android:searchSuggestIntentAction="android.intent.action.VIEW"
    android:searchSuggestSelection=" ? "
/>

```

**Uwaga!**

Zwróćmy uwagę na wartość ciągu znaków searchSuggestAuthority. Powinna ona odpowiadać właściwej definicji adresu URL dostawcy treści, umieszczonej w pliku manifeście.

Zwróćmy uwagę na atrybut searchSuggestSelection. Jest on bezpośrednio związany z argumentem selection metody query() znajdującej się w naszym dostawcy treści. Jak pamiętamy z rozdziału 4., argument ten służy przeważnie do przekazywania klauzuli where wraz z wymienialnymi symbolami ?.

Następnie tablica wymienialnych wartości jest przekazywana do argumentu tablicy selection →Args. Tak się rzeczywiście dzieje. W przypadku określenia atrybutu searchSuggestSelection istnieje założenie, że nie chcemy otrzymywać tekstu wyszukiwania poprzez identyfikator URI, lecz za pomocą argumentu selection metody query(). W takim wypadku proces wyszukiwania w Androidzie będzie wysyłał symbol ? (zwróćmy uwagę na spację poprzedzającą znak ?) jako wartość argumentu selection i przekazywał tekst kwerendy w postaci pierwszego elementu tablicy argumentów selection.

Jeśli nie zdefiniujemy argumentu searchSuggestSelection, tekst wyszukiwania zostanie przekazany w formie ostatniego segmentu identyfikatora URI. Możemy wybrać dowolny sposób. W naszym przykładzie wykorzystaliśmy argument selection, a nie identyfikator URI.

## **Badanie metadanych wyszukiwania pod kątem niestandardowego dostawcy propozycji**

Skoro już poruszyliśmy temat metadanych wyszukiwania, to sprawdźmy, jakie są dostępne pozostałe atrybuty. Omówimy atrybuty najczęściej stosowane wobec dostawcy propozycji oraz najbliższej z nim związane. Pełną listę atrybutów interfejsu API SearchManager możemy przejrzeć pod adresem: <http://developer.android.com/guide/topics/search/searchable-config.html>.

Atrybut searchSuggestIntentAction (listing 23.21) używany jest do przekazywania lub definiowania działania danej intencji podczas wywołania aktywności SearchActivity za pomocą tej intencji. Dzięki temu klasa SearchActivity nie jest ograniczona wyłącznie do domyślnego wyszukiwania. Poniżej prezentujemy przykład wykorzystania działania intencji wewnętrz metody onCreate(), będącej częścią odpowiadającą aktywności wyszukiwania:

```

// Ciało metody onCreate

// Wprowadzamy tutaj i przetwarzamy kwerendę wyszukiwania
final Intent queryIntent = getIntent();
//działanie kwerendy
final String queryAction = queryIntent.getAction();

```

```
if (Intent.ACTION_SEARCH.equals(queryAction))
{
    this.doSearchQuery(queryIntent);
}
else if (Intent.ACTION_VIEW.equals(queryAction))
{
    this.doView(queryIntent);
}
else {
    Log.d(tag,"NIE tworzy kwerendy z poziomu wyszukiwania");
}
```

Zobaczmy ten kod umieszczony w kontekście (listing 23.23), gdzie klasa `SearchActivity` oczekuje działania `VIEW` lub `SEARCH` poprzez sprawdzanie wartości działania danej intencji.

Kolejny atrybut, którego nie wykorzystaliśmy, lecz który jest dostępny dla dostawców propozycji, nosi nazwę `searchSuggestPath`. Po określeniu tej wartości o typie `string` zostanie ona przyłączona do identyfikatora URI (wywołującego dostawcę propozycji) tuż po segmencie `SUGGEST_URI_PATH_QUERY`. Umożliwia to pojedynczemu, niestandardowemu dostawcy propozycji przetworzenie dwóch różnych aktywności wyszukiwania. Każda aktywność `SearchActivity` będzie posiadała inny sufiks identyfikatora URI. Dostawca propozycji może korzystać z tego sufiksu ścieżki, aby przekazywać różne zestawy wyników do docelowej aktywności wyszukiwania.

Podobnie jak w przypadku działania `Intent`, również i teraz możemy określić dane intencji za pomocą atrybutu `searchSuggestIntentData`. Jest to identyfikator URI danych, który podczas wywołania może być przekazany — jako część intencji, wraz z działaniem — do aktywności wyszukiwania.

Atrybut `searchSuggestThreshold` definiuje liczbę znaków, jaką należy wpisać w polu QSB, aby wywołać dostawcę propozycji. Domyślną wartością progową jest 0.

Kolejny atrybut, `queryAfterZeroResults` (przyjmujący wartości `true` lub `false`), wskazuje, czy dla następnego zestawu znaków ma zostać wywołany dostawca, w przypadku gdy dla bieżącego zestawu znaków otrzymano zerowy zestaw wyników. W przypadku naszego adresu URL istotne jest, aby włączyć tę flagę, dzięki czemu cały tekst kwerendy staje się za każdym razem widoczny.

Po zapoznaniu się z identyfikatorami URI, argumentami `selection` i metadanymi wyszukiwania zajmiemy się najistotniejszym aspektem dostawcy propozycji — kursorem propozycji.

## Kolumny kursora propozycji

Kursor propozycji jest przede wszystkim kursem. Nie różni się niczym od kursorów bazodanowych, obszernie omówionych w rozdziale 4. Kursor propozycji pełni rolę kontraktu pomiędzy procesem wyszukiwania w Androidzie a dostawcą propozycji. Oznacza to, że nazwy i typy kolumn przekazywane przez kursor są niezmienne i znane obydwu stronom.

W celu uzyskania elastyczności procesu wyszukiwania Android oferuje olbrzymią liczbę kolumn, w większości opcjonalnych. Dostawca propozycji nie musi przekazywać wszystkich kolumn; może zignorować kolumny, które nie są z nim ściśle powiązane. W tym punkcie przyjrzymy się przeznaczeniu części kolumn (opis pozostałych kolumn znajdziemy we wspomnianym już kilkakrotnie omówieniu interfejsu API `SearchManager`).

Najpierw zajmiemy się kolumnami, które dostawca propozycji może przekazywać, omówimy ich przeznaczenie oraz wpływ na wyszukiwanie.

Tak jak wszystkie kursory, również kurSOR propozycji musi zawierać kolumnę `_id`. Jest to kolumna obowiązkowa. Nazwy pozostałych kolumn rozpoczynają się od przedrostka `SUGGEST_COLUMN_`. Stałe te są zdefiniowane jako część odniesienia do interfejsu API SearchManager. Poniżej zostaną omówione najczęściej używane kolumny. Pełną ich listę można znaleźć w umieszczonych na końcu rozdziału zasobach dotyczących tego interfejsu API.

- `text_1` — jest to pierwszy wiersz tekstu propozycji (rysunek 23.3).
- `text_2` — jest to drugi wiersz tekstu propozycji (rysunek 23.3).
- `icon_1` — jest to ikona umieszczona po lewej stronie propozycji; przechowuje ona zazwyczaj identyfikator zasobu.
- `icon_2` — jest to ikona umieszczona po prawej stronie propozycji; przechowuje ona zazwyczaj identyfikator zasobu.
- `intent_action` — jest to argument przekazywany aktywności `SearchActivity` podczas jej wywoływania w postaci działania intencji. W przypadku obecności tej kolumny w metadanych wyszukiwania będzie ona przesyłała odpowiednie działanie intencji (listing 23.21).
- `intent_data` — są to informacje przekazywane aktywności `SearchActivity` podczas jej wywoływania w postaci danych intencji. W przypadku obecności tej kolumny w metadanych wyszukiwania będzie ona przesyłała odpowiednie działanie intencji (listing 23.21). Jest to identyfikator URI danych.
- `intent_data_id` — zostaje ona dodana do identyfikatora URI danych. Jest szczególnie przydatna, gdy chcemy jednorazowo wspomnieć o głównej partii danych w metadanych, a następnie zmieniać tę partię dla każdej propozycji. Dzięki tej kolumnie można nieco skuteczniej przeprowadzić taką czynność.
- `query` — ciąg znaków kwerendy, wysyłany do aktywności wyszukiwania.
- `shortcut_id` — jak zostało wcześniej wspomniane, wyszukiwanie w Androidzie przechowuje w pamięci podręcznej propozycje dostarczane przez dostawcę. Takie przechowywane propozycje noszą nazwę skrótów. Jeżeli ta kolumna jest nieobecna, Android będzie przechowywał propozycję i nigdy nie zażąda jej aktualizacji. Jeżeli zostanie umieszczona wartość `SUGGEST_NEVER_MAKE_SHORTCUT`, Android przestanie przechowywać propozycje w pamięci podręcznej. Jeżeli wstawimy tu dowolną inną wartość, identyfikator ten zostanie przekazany jako ostatni segment identyfikatora URI skrótu (więcej informacji znajduje się w podpunkcie „[Identyfikatory URI dostawcy propozycji](#)”).
- `spinner_while_refreshing` — ta wartość logiczna pozwala określić, czy podczas procesu aktualizowania skrótów ma być używana kontrolka Spinner.

Istnieje również zmienny zestaw dodatkowych kolumn, umożliwiających reagowanie na wcisnięcie przycisków działania. Zajmiemy się tą kwestią podczas omawiania przycisków działania. Zobaczmy, w jaki sposób nasz niestandardowy dostawca propozycji przekazuje te kolumny.

## Zapełnianie i przekazywanie listy kolumn

Niestandardowy dostawca propozycji nie musi przekazywać wszystkich kolumn wymienionych we wcześniejszym podpunkcie. Nasz dostawca będzie przekazywał jedynie podzbior kolumn określony na podstawie zadań omówionych w podrozdziale „[Implementacja niestandardowego dostawcy propozycji](#)”.

Po analizie listingu 23.20 możemy stwierdzić, że wyjściowa lista kolumn jest następująca (została ona wstawiona do listingu 23.22):

#### **Listing 23.22.** Definiowanie kolumn kurSORA propozycji

---

```
private static final String[] COLUMNS = {  
    "_id", // musi zawierać tę kolumnę  
    SearchManager.SUGGEST_COLUMN_TEXT_1,  
    SearchManager.SUGGEST_COLUMN_TEXT_2,  
    SearchManager.SUGGEST_COLUMN_INTENT_DATA,  
    SearchManager.SUGGEST_COLUMN_INTENT_ACTION,  
    SearchManager.SUGGEST_COLUMN_SHORTCUT_ID  
};
```

---

Kolumny te zostały dobrane w taki sposób, aby spełniały poniższe wymagania.

Kiedy użytkownik wpisuje w polu QSB słowo z podpowiedzią, taką jak great.m, nasz dostawca propozycji nie odpowie, dopóki w tekście wyszukiwania znajduje się kropka. Po rozpoznaniu wyrazu dostawca propozycji wyizoluje go z całego wyrażenia (w naszym przypadku jest to wyraz *great*), a następnie przekaże dwie propozycje.

Pierwsza propozycja służy do wywołania witryny *thefreewebdictionary.com* wraz z danym słowem, natomiast druga propozycja spowoduje przeszukanie bazy danych Google za pomocą wyrażenia `define:great`.

W tym celu dostawca wczytuje kolumnę `intent_action` jako klasę `intent.action.view` oraz dane intencji zawierające pełny identyfikator URI. Android powinien uruchomić przeglądarkę po rozpoznaniu identyfikatora URI danych, rozpoczynającego się od segmentu `http://`.

Wypełnimy kolumnę `text1` wartościami `search some-website with:`, a kolumnę `text2` właściwym słowem (przypominamy, że w naszym przypadku jest to wyraz *great*). Aby uprościć zadanie, identyfikatorowi skrótu przypiszemy wartość `SUGGEST_NEVER_MAKE_SHORTCUT`. W ten sposób wyłączymy przechowywanie propozycji w pamięci podręcznej i uniemożliwimy usunięcie skrótu identyfikatora URI propozycji.

Na tym zakończymy analizę kodu źródłowego klasy niestandardowego dostawcy propozycji. Czytelnicy uzyskali wiedzę na temat identyfikatorów URI, kurSORów propozycji oraz metadanych wyszukiwania powiązanych z określonym dostawcą. Wiedzą już także, w jaki sposób można zapełniać kolumny propozycji.

Zastanówmy się teraz nad sposobem zaimplementowania aktywności wyszukiwania do naszego niestandardowego dostawcy propozycji.

## **Implementacja aktywności wyszukiwania dla niestandardowego dostawcy propozycji**

Podczas omawiania tematu implementacji prostego dostawcy propozycji zajęliśmy się pojęciem tematem zadań aktywności wyszukiwania. Przeanalizujmy teraz pominięte aspekty tego zagadnienia.

Proces przeszukiwania w Androidzie wywołuje aktywność wyszukiwania w celu przetworzenia działań wyszukiwania, uruchomionych na jeden z dwóch sposobów. Jednym ze sposobów

uruchomienia działania wyszukiwania jest kliknięcie ikony wyszukiwania, będącej częścią pola QSB; drugi natomiast polega na bezpośrednim kliknięciu propozycji.

Aktywność wyszukiwania musi ustalić powód, dla którego została wywołana. Informacja ta jest umieszczona w intencji działania. Dlatego intencja ta musi zostać przeanalizowana przez aktywność wyszukiwania. W wielu przypadkach działaniem takim jest ACTION\_SEARCH. Jednak dostawca propozycji posiada możliwość przesłonięcia tego działania poprzez jawne określenie innego działania, korzystając z metadanych wyszukiwania lub kolumny kurSORA propozycji. W naszym przykładzie stosujemy działanie VIEW.

W trakcie omawiania prostego dostawcy propozycji wspomnialiśmy, że istnieje również możliwość skonfigurowania trybu uruchamiania singleTop wobec aktywności wyszukiwania. W takim przypadku aktywność musi dodatkowo odpowiedzieć na metodę onNewIntent(), a także na metodę onCreate(). Omówimy przypadki odpowiadania na obydwie metody i pokażemy podobieństwa występujące pomiędzy nimi.

Zastosujemy zarówno metodę onNewIntent(), jak i onCreate() w celu przeanalizowania działań ACTION\_SEARCH i ACTION\_VIEW. W trakcie korzystania z działania wyszukiwania spowodujemy po prostu przekazanie tekstu kwerendy użytkownikowi. W przypadku działania widoku przeniesiemy kontrolkę do wyszukiwarki i zakończymy bieżącą aktywność, dzięki czemu użytkownik będzie miał wrażenie wywołania przeglądarki bezpośrednio po kliknięciu propozycji.

#### Uwaga!

Klasa SearchActivity nie musi być aktywnością uruchamianą z poziomu głównego menu aplikacji. Upewnijmy się, że nie skonfigurujemy nieświadomie filtrów intencji w taki sposób jak w przypadku innych aktywności, wywoływanych z poziomu głównego menu aplikacji.

Skoro już wiemy, czego oczekiwany, spójrzmy na kod źródłowy pliku *SearchActivity.java*.

## Klasa SearchActivity niestandardowego dostawcy propozycji

Po zapoznaniu się z zadaniami aktywności wyszukiwania, szczególnie zaś z tymi przedstawionymi w naszym przykładzie, można zapoznać się z jej kodem źródłowym (listing 23.23).

### Listing 23.23. Klasa SearchActivity

```
//plik: SearchActivity.java
public class SearchActivity extends Activity
{
    private final static String tag = "SearchActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        Log.d(tag, "Jestem tworzona");
        setContentView(R.layout.layout_test_search_activity);

        // uzyskuje i przetwarza tu kwerendę wyszukiwania
        final Intent queryIntent = getIntent();
        //działanie kwerendy
        final String queryAction = queryIntent.getAction();
```

```
Log.d(tag,"Utworz dzialanie intencji:"+queryAction);

final String queryString =
    queryIntent.getStringExtra(SearchManager.QUERY);
Log.d(tag,"Utworz kwerende intencji:"+queryString);

if (Intent.ACTION_SEARCH.equals(queryAction))
{
    this.doSearchQuery(queryIntent);
}
else if (Intent.ACTION_VIEW.equals(queryAction))
{
    this.doView(queryIntent);
}
else {
    Log.d(tag,"Utworz intencje NIE z poziomu wyszukiwania");
}
return;
}

@Override
public void onNewIntent(final Intent newIntent)
{
    super.onNewIntent(newIntent);
    Log.d(tag,"wywoluje mnie nowa intencja");

    // uzyskuje tu i przetwarza kwerendę wyszukiwania
    final Intent queryIntent = newIntent;

    //dzialanie kwerendy
    final String queryAction = queryIntent.getAction();
    Log.d(tag,"Nowe dzialanie intencji:"+queryAction);

    final String queryString =
        queryIntent.getStringExtra(SearchManager.QUERY);
    Log.d(tag,"Nowa kwerenda intencji:"+queryString);

    if (Intent.ACTION_SEARCH.equals(queryAction))
    {
        this.doSearchQuery(queryIntent);
    }
    else if (Intent.ACTION_VIEW.equals(queryAction))
    {
        this.doView(queryIntent);
    }
    else {
        Log.d(tag,"Nowa intencja utworzona NIE z poziomu wyszukiwania");
    }
    return;
}
private void doSearchQuery(final Intent queryIntent)
{
    final String queryString =
        queryIntent.getStringExtra(SearchManager.QUERY);
    appendText("Szukasz obiektu:" + queryString);
}
```

---

```

private void appendText(String msg)
{
    TextView tv = (TextView)this.findViewById(R.id.text1);
    tv.setText(tv.getText() + "\n" + msg);
}
private void doView(final Intent queryIntent)
{
    Uri uri = queryIntent.getData();
    String action = queryIntent.getAction();
    Intent i = new Intent(action);
    i.setData(uri);
    startActivity(i);
    this.finish();
}
}

```

---

Rozpoczniemy analizę kodu źródłowego od określenia sposobu wywołania naszej aktywności wyszukiwania.

## Szczegóły wywołania klasy SearchActivity

Tak jak wszystkie aktywności, również aktywność wyszukiwania musi zostać wywołana za pomocą intencji. Jednak założenie, że za wywołanie aktywności jest zawsze odpowiedzialna intencja `action`, jest niewłaściwe. Okazuje się, że aktywność wyszukiwania jest jawnie wywoływana poprzez specyfikację jej składowej nazwy.

Zastanówmy się, dlaczego jest to takie istotne. Jak wiemy, w naszym dostawcy propozycji jawnie definiujemy intencję `action` w krotce propozycji. Jeżeli działaniem intencji jest `VIEW`, a jej danymi — adres URL HTTP, to nieświadomu programista mógłby uznać, że w odpowiedzi zostanie uruchomiona przeglądarka, a nie aktywność wyszukiwania. Takie zjawisko byłoby z pewnością bardzo pożądane. Jednak ponieważ intencja posiada dostęp do nazwy aktywności wyszukiwania oraz działania i danych intencji, ostatecznie to nazwa aktywności uzupełnia pierwszeństwo.

Nie jesteśmy pewni, dlaczego zostało wprowadzone takie ograniczenie oraz w jaki sposób można je ominąć. Faktem jest jednak, że niezależnie od działania intencji definiowanej przez dostawcę propozycji to właśnie aktywność wyszukiwania zostanie wywołana. W naszym przykładzie uruchomimy po prostu przeglądarkę z poziomu aktywności wyszukiwania i zamknijmy tę aktywność.

Zademonstrujemy to rozwiązywanie na podstawie intencji uruchamianej przez system po kliknięciu propozycji w celu wywołania naszej aktywności wyszukiwania:

```

launching Intent {
act=android.intent.action.VIEW
dat=http://www.google.com
flg=0x10000000
cmp=com.androidbook.search.custom/.SearchActivity (has extras)
}

```

Zwróćmy uwagę na specyfikację składnika intencji. Wskazuje ona bezpośrednio aktywność wyszukiwania. Zatem bez względu na określone przez nas działanie intencji Android będzie zawsze wywoływał aktywność wyszukiwania. W wyniku tego wywołanie przeglądarki jest zadaniem tej aktywności.

Przyjrzyjmy się teraz, co się dzieje z tymi intencjami w aktywności wyszukiwania.

## Odpowiedź na działania ACTION\_SEARCH i ACTION\_VIEW

Wiemy, że do wywołania aktywności wyszukiwania proces wyszukiwania w Androidzie jawnie wykorzystuje jej nazwę. Jednak intencja wywołująca przechowuje również określone działanie. Kiedy pole QSB wywołuje tę aktywność po kliknięciu przez użytkownika ikony wyszukiwania, mamy do czynienia z działaniem ACTION\_SEARCH.

Działanie może być inne w przypadku jego wywołania przez propozycję wyszukiwania. Zależy to od skonfigurowania propozycji przez dostawcę. W naszym przypadku dostawca propozycji konfiguruje działanie ACTION\_VIEW.

Z powodu obecności różnych działań aktywność wyszukiwania musi je rozpoznawać. Przedstawiamy poniżej kod umożliwiający określenie, czy ma zostać wywołana metoda wyszukiwania kwerendy, czy metoda widoku (segment kodu został zaczerpnięty z listingu 23.23):

```
if (Intent.ACTION_SEARCH.equals(queryAction))
{
    this.doSearchQuery(queryIntent);
}
else if (Intent.ACTION_VIEW.equals(queryAction))
{
    this.doView(queryIntent);
}
```

Widzimy, że dla działania widoku wywołujemy metodę doView(), a w przypadku działania wyszukiwania — doSearchQuery().

Z pomocą funkcji doView() odczytujemy działanie oraz identyfikator URI danych i zapełniajemy nimi nową intencję, a następnie wywołujemy aktywność. W ten sposób zostanie przywołana przeglądarka. Zakończymy aktywność w taki sposób, aby przycisk cofania spowodował powrót do wywołującego ją procesu wyszukiwania.

W metodzie doSearchQuery() wyświetlamy jedynie tekst kwerendy wyszukiwania w widoku. Przyjrzyjmy się układowi graficznemu wykorzystanemu do obsługi funkcji doSearchQuery().

## Układ graficzny aktywności wyszukiwania

Listing 23.24 przedstawia prosty układ graficzny, wykorzystywany przez aktywność wyszukiwania po uruchomieniu metody doSearchQuery(). Jedyny istotny fragment został zaznaczony pogrubioną czcionką.

**Listing 23.24.** Układ graficzny klasy SearchActivity

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- plik: layout/layout_search_activity.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
```

---

```

    android:layout_height="wrap_content"
    android:text="@string/search_activity_main_text"
  />
</LinearLayout>

```

---

Nadszedł odpowiedni moment na zaprezentowanie zawartości pliku *strings.xml*, odpowiadnego za wyświetlanie niektórych ciągów znaków w naszej aplikacji.

## Plik strings.xml

Przedstawiony na listingu 23.25 plik *strings.xml* definiuje ciągi znaków tekstowych układu graficznego oraz takie elementy, jak nazwa aplikacji, niektóre ciągi znaków konfiguracji wyszukiwania itp.

**Listing 23.25.** Plik strings.xml

---

```

<?xml version="1.0" encoding="utf-8"?>
<!-- plik: values/strings.xml -->
<resources>
  <string name="search_activity_main_text">
    Jest to aktywność wyszukiwania.
    \n\n
    Ten widok zostanie wywołany, jeżeli zostanie
    użyte działanie ACTION_SEARCH, a nie ACTION_VIEW.
    \n\n
    Działanie ACTION_SEARCH zostaje uruchomione po kliknięciu ikony wyszukiwania.
    \n\n
    Działanie ACTION_VIEW zostaje uruchomione po kliknięciu propozycji.
  </string>
  <string name="app_name">Niestandardowa aplikacja propozycji</string>
  <string name="search_label">Demonstracja niestandardowej propozycji</string>
  <string name="search_hint">Demonstracja podpowiedzi niestandardowej
  propozycji</string>
</resources>

```

---

## Odpowiedź na metody onCreate() i onNewIntent()

Jeżeli przyjrzymy się kodowi z listingu 23.23, zauważmy, że fragmenty w metodach *onCreate()* i *onNewIntent()* są niemal identyczne. Jest to dosyć powszechny wzorzec.

W zależności od trybu uruchamiania aktywności wyszukiwania po jej wywołaniu przywoływana jest metoda *onCreate()* lub *onNewIntent()*.

**Uwaga!**

W umieszczonej pod koniec rozdziału sekcji „Odnośniki” znajduje się łącze do użytecznych materiałów na temat trybów uruchamiania aktywności wyszukiwania.

## Uwagi na temat zakończenia aktywności wyszukiwania

Wspomnieliśmy wcześniej o sposobie odpowiedzi na metodę *doView()*. Na listingu 23.26 pokazaliśmy kod tej funkcji (jest to wyciąg z listingu 23.23).

**Listing 23.26.** Zakończenie aktywności wyszukiwania

```
private void doView(final Intent queryIntent)
{
    Uri uri = queryIntent.getData();
    String action = queryIntent.getAction();
    Intent i = new Intent(action);
    i.setData(uri);
    startActivity(i);
    this.finish();
}
```

---

Celem tej funkcji jest wywołanie przeglądarki. Gdybyśmy na końcu nie wywołały funkcji `finish()`, użytkownik po kliknięciu przycisku cofania zobaczyłby widok aktywności wyszukiwania, a nie z powrotem ekran wyszukiwania, jak należałoby się spodziewać.

W idealnym przypadku, aby zapewnić użytkownikowi jak najlepszy komfort pracy, kontrolka nie powinna nigdy przechodzić przez aktywność wyszukiwania. Zakończenie tej aktywności rozwiązuje problem. Kod z listingu 23.26 daje nam również okazję zbadania, w jaki sposób przenosimy działanie i dane intencji z oryginalnej intencji (ustanowionej przez dostawcę propozycji), a następnie przekazujemy je do nowej intencji przeglądarki.

Wprowadziłmy w tym podrozdziale mnóstwo nowych wiadomości. Przedstawiliśmy szczegółowo implementacje dostawcy propozycji oraz aktywności wyszukiwania. W międzyczasie pokazaliśmy również plik metadanych wyszukiwania i plik `strings.xml`. Analizę plików wymaganych do implementacji naszego projektu zamkniami badaniem pliku manifestu aplikacji.

## Plik manifest niestandardowego dostawcy propozycji

W pliku manifeście zbieramy wszystkie składniki naszej aplikacji. Tak samo jak w przypadku innych przykładowych aplikacji, deklarujemy tutaj składniki naszego dostawcy propozycji, takie jak aktywność wyszukiwania oraz właściwy kod dostawcy propozycji. Plik ten służy również do zadeklarowania możliwości wyszukiwania lokalnego przez aplikację poprzez ustanowienie „aktywności wyszukiwania” domyślnym procesem przeszukiwania. Zwróćmy również uwagę na filtry intencji zdefiniowane dla aktywności wyszukiwania.

Poszczególne wymienione powyżej informacje zostały wyróżnione pogrubioną czcionką w kódrze pliku manifestu (listing 23.27).

**Listing 23.27.** Plik manifest niestandardowego dostawcy propozycji

```
//plik: AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.search.custom"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"
        android:label="Niestandardowy dostawca propozycji">
        <!--
        ****
        * Kod związany z wyszukiwaniem: aktywność wyszukiwania
        ****-->
```

```
-->
<activity android:name=".SearchActivity"
          android:label="Etykieta aktywności wyszukiwania"
          android:launchMode="singleTop">
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>

    <meta-data android:name="android.app.searchable"
              android:resource="@xml/searchable" />
</activity>

<!-- Deklaracja domyślnego wyszukiwania -->
<meta-data android:name="android.app.default_searchable"
          android:value=".SearchActivity" />

<!-- Deklaracja dostawcy propozycji -->
<provider android:name="SuggestUrlProvider"
          android:authorities="com.androidbook.search.
          custom.suggesturlprovider"
/>
</application>
<uses-sdk android:minSdkVersion="4" />
</manifest>
```

Jak widać, zaznaczyliśmy trzy elementy:

- definicję aktywności wyszukiwania oraz związany z nią plik XML metadanych wyszukiwania,
- definicję aktywności wyszukiwania jako domyślnego procesu wyszukiwania w aplikacji,
- definicję dostawcy propozycji oraz jego uprawnienia.

Po utworzeniu niezbędnego kodu czas uruchomić aplikację i sprawdzić, jak się ona prezentuje na emulatorze.

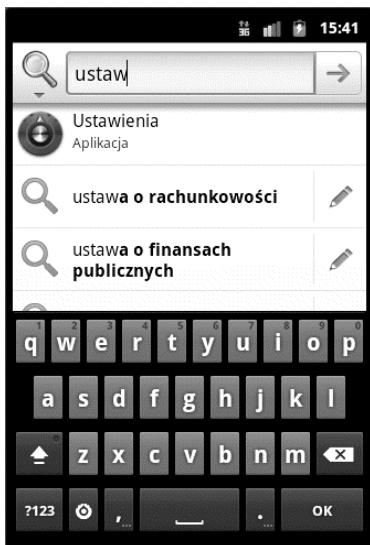
## Korzystanie z niestandardowego dostawcy propozycji

Po skompilowaniu i wdrożeniu aplikacji za pomocą narzędzi ADT nie ujrzymy pojawiących się aktywności, ponieważ żadna nie została uruchomiona. Zamiast tego zobaczymy w konsoli Eclipse komunikat, że aplikacja została poprawnie zainstalowana.

Oznacza to, że dostawca propozycji jest gotowy na przetwarzanie wpisów w globalnym polu QSB. Zanim to jednak nastąpi, musimy dołączyć naszego dostawcę do udziału w procesie wyszukiwania globalnego.

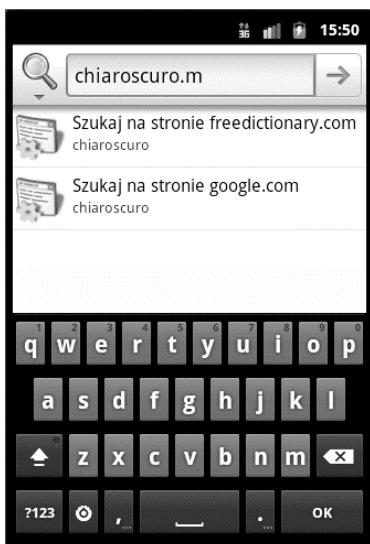
Pokazaliśmy na początku rozdziału, w jaki sposób możemy uruchomić aplikację ustawień wyszukiwania. Przedstawimy teraz szybsze rozwiązanie, korzystające z tej samej funkcji wyszukiwania, którą omawiamy w tym rozdziale.

Otwórzmy globalne pole QSB i wpiszmy w nim ustaw. System wyświetli nazwę aplikacji Ustawienia jako jedną z propozycji wyszukiwania (rysunek 23.26).



**Rysunek 23.26.** Wywoływanie aplikacji ustawień za pomocą procesu wyszukiwania

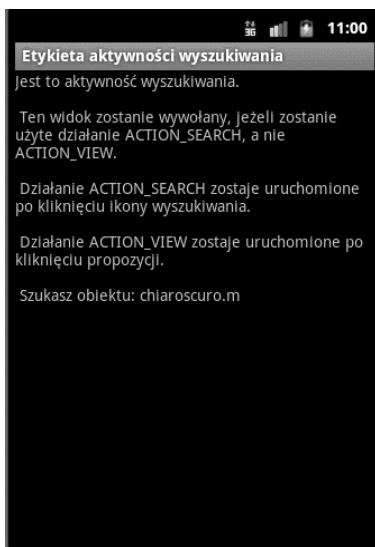
Wykorzystujemy tę samą wiedzę, którą uzyskaliśmy na temat pola QSB, aby wywołać aplikację Ustawienia. Skorzystajmy z rozwiązań, które omówiliśmy na początku rozdziału, i dołączmy naszą aplikację do propozycji. Gdy to zrobimy, wpiszymy w polu QSB tekst widoczny na rysunku 23.27.



**Rysunek 23.27.** Więcej wyników od niestandardowego dostawcy propozycji

Zwróćmy uwagę na sposób prezentowania propozycji dostarczanych przez naszego dostawcę. Jeśli teraz klikniemy ikonę wyszukiwania widoczną w lewym górnym rogu ekranu i zmienimy aplikację wyszukującą na *Niestandardowy dostawca propozycji*, następnie zaś przejdziemy do jednej z propozycji prezentowanych przez niestandardowego dostawcę, po czym klikniemy

ikonę wyszukiwania, Android bezpośrednio uruchomi aktywność wyszukiwania z pominięciem przeglądarki, co zostało pokazane na rysunku 23.28 (w ten sposób zostają zademonstrowane dwa rodzaje omawianych przez nas działań intencji: wyszukiwanie i widok).



**Rysunek 23.28.** Kwerenda wyszukiwania wywołująca wyniki wyszukiwania

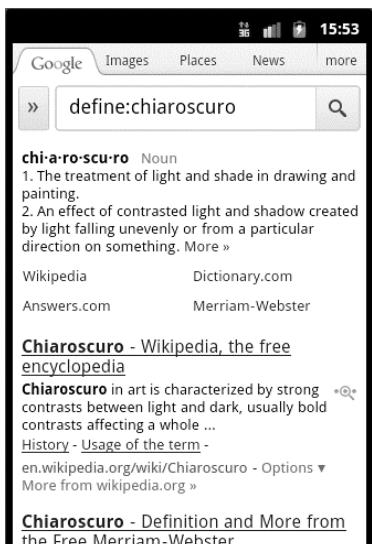
Przykład ten więc pozwala na porównanie działania ACTION\_SEARCH i ACTION\_VIEW.

Jeśli klikniemy teraz pierwszą propozycję z rysunku 23.27 (darmowy słownik), system wywoła przeglądarkę, co zostało zaprezentowane na rysunku 23.29.



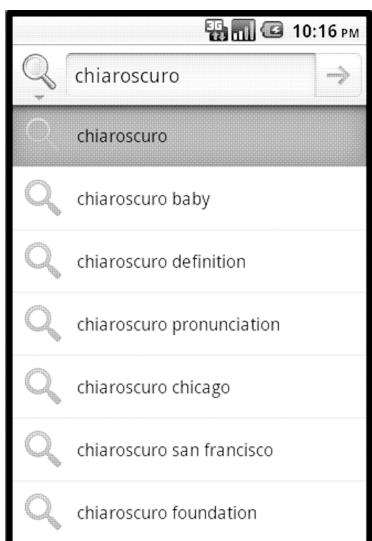
**Rysunek 23.29.** Darmowy słownik

Jeżeli klikniemy propozycję otwarcia witryny Google (rysunek 23.27), ujrzymy przeglądarkę widoczną na rysunku 23.30.



Rysunek 23.30. Wyszukiwanie definicji na stronie google.com

Na rysunku 23.31 przedstawiamy widok, jaki pojawi się, jeśli nie dodamy przyrostka .m w polu wyszukiwania globalnego.



Rysunek 23.31. Niestandardowy dostawca pozbawiony podpowiedzi

Odnosząc się do tego, że nasz dostawca propozycji nie przekazał żadnego wyniku.

W ten sposób kończymy dyskusję dotyczącą tematu budowania od podstaw funkcjonalnego, niestandardowego dostawcy propozycji. Choć omówiliśmy każdy aspekt procesu wyszukiwania, istnieje jeszcze kilka tematów, których do tej pory nie poruszyliśmy. Należą do nich pojęcia przycisków działania oraz danych wyszukiwania specyficznych dla aplikacji. Omówienie tych tematów jest naszym kolejnym celem.

## Zastosowanie przycisków działania i danych wyszukiwania specyficznych dla aplikacji

Przyciski działania oraz specyficzne dla aplikacji dane wyszukiwania zwiększą elastyczność procesu wyszukiwania w Androidzie.

Przyciski działania umożliwiają przystosowanie przycisków funkcyjnych urządzenia do obsługi funkcji wyszukiwania. Specyficzne dla aplikacji dane wyszukiwania stanowią dodatkowe informacje, które są przekazywane aktywności wyszukiwania.

### Uwaga!

Należy zauważyć, że kody zawarte na następnych listingach nie tworzą projektu, który można przetestować. Służą one wyłącznie do zilustrowania koncepcji omawianych w tekście.

Zaczniemy od przycisków działania.

## Wykorzystanie przycisków działania w procesie wyszukiwania

Do tej pory zademonstrowaliśmy wiele sposobów wywołania procesu wyszukiwania:

- poprzez ikonę wyszukiwania dostępną w polu QSB,
- poprzez przycisk wyszukiwania, stanowiący jeden z przycisków działania (ukazanych na rysunku 23.1),
- jawnie — za pomocą ikony lub przycisku, które są wyświetlane przez aktywność,
- poprzez naciśnięcie dowolnego przycisku na podstawie deklaracji trybu *type-to-search*.

W tym podrozdziale przyjrzymy się technice wywoływania procesu wyszukiwania za pomocą przycisków działania. Przyciskami działania nazywamy zestaw przycisków umieszczonych na urządzeniu, którym są przypisane określone działania. Przykłady kilku takich przycisków działania zostały przedstawione na listingu 23.28.

**Listing 23.28.** Lista kodów przycisków działania

---

```
 keycode_dp此_up
keycode_dp此_down
keycode_dp此_left
keycode_dp此_right
keycode_dp此_center
keycode_back
keycode_call
keycode_camera
keycode_clear
keycode_endcall
keycode_home
```

```
keycode_menu  
keycode_mute  
keycode_power  
keycode_search  
keycode_volume_up  
keycode_volume_down
```

---

Przyciski te są zdefiniowane w interfejsie API klasy KeyEvent, której opis można znaleźć na następującej stronie: [http://developer.android.com/reference/android/view\(KeyEvent.html](http://developer.android.com/reference/android/view(KeyEvent.html)).

**Uwaga!**

Nie wszystkie wymienione przyciski działania można wykorzystać w procesie wyszukiwania, jednak część z nich nie sprawia pod tym względem problemów, na przykład keycode\_call. Należy samodzielnie sprawdzić każdy przycisk działania pod kątem działania i przydatności.

Po wybraniu interesującego nas przycisku działania możemy powiadomić o tym system poprzez umieszczenie informacji na ten temat w metadanych w sposób przedstawiony na listingu 23.29.

**Listing 23.29.** Przykładowa definicja przycisku działania

---

```
<searchable xmlns:android="http://schemas.android.com/apk/res/android"  
    android:label="@string/search_label"  
    android:hint="@string/search_hint"  
  
    android:searchMode="showSearchLabelAsBadge"  
  
    android:includeInGlobalSearch="true"  
  
    android:searchSuggestAuthority="com.androidbook.  
    ↳search.simplesp.SimpleSuggestionProvider"  
    android:searchSuggestSelection=" ? "  
>  
    <actionkey  
        android: keycode="KEYCODE_CALL"  
        android:queryActionMsg="call"  
        android:suggestActionMsg="call"  
        android:suggestActionMsgColumn="call_column" />  
  
    <actionkey  
        android: keycode="KEYCODE_DPAD_CENTER"  
        android:queryActionMsg="doquery"  
        android:suggestActionMsg="dosuggest"  
        android:suggestActionMsgColumn="my_column" />  
    ....  
</searchable>
```

---

Możemy również przyporządkować kilka przycisków działania dla tego samego kontekstu wyszukiwania. Poniżej wyjaśniamy, do czego służy każdy element actionkey oraz w jaki sposób jest stosowany do przetwarzania naciśnięcia przycisku.

- **keycode** — jest to kod przycisku zdefiniowany w interfejsie API klasy KeyEvent, który powinien zostać użyty do wywołania aktywności wyszukiwania. Możliwość naciśnięcia przycisku identyfikowanego przez taki kod pojawia się dwukrotnie. Pierwszy

raz występuje ona podczas wpisywania przez użytkownika tekstu wyszukiwania w polu wyszukiwania, po którym nie zostają wyświetcone propozycje. Jeżeli funkcja przycisku działania nie została zaimplementowana, użytkownik przeważnie kliknie ikonę wyszukiwania w polu QSB. W przypadku zaimplementowania przycisku działania w metadanych wyszukiwania Android pozwala użytkownikowi na wcisnięcie tego przycisku zamiast ikony wyszukiwania. Druga sytuacja pojawia się, gdy użytkownik przejdzie do określonej propozycji i wcisnie przycisk działania. W obydwu przypadkach dla aktywności wyszukiwania system wywołuje działanie ACTION\_SEARCH. Informację na ten temat przenosi dodatkowy ciąg znaków SearchManager.ACTION\_KEY. Jeżeli znajduje się w nim jakaś wartość, to wiadomo, że nastąpi wywołanie w odpowiedzi na wcisnięcie przycisku działania.

- queryActionMsg — dowolny tekst wpisany w tym elemencie zostaje przekazany jako wartość dodatkowego ciągu znaków SearchManager.ACTION\_MSG do aktywności wyszukiwania wywołującej intencję. Jeżeli odczytamy tę informację z intencji i jest ona identyczna z danymi zdefiniowanymi w metadanych, wiadomo będzie, że następuje bezpośrednie wywołanie z poziomu pola QSB w wyniku wcisnięcia przycisku działania. Bez takiego testu nie będziemy mieli pewności, czy działanie ACTION\_SEARCH zostało bezpośrednio wywołane wobec propozycji w odpowiedzi na wcisnięcie przycisku działania.
- suggestActionMsg — dowolny tekst wpisany w tym elemencie zostaje przekazany jako wartość dodatkowego ciągu znaków SearchManager.ACTION\_MSG do aktywności wyszukiwania wywołującej intencję. Dodatkowe przyciski dla tego argumentu i pola queryActionMsg są takie same. Jeżeli przypiszemy tym polom identyczną wartość, na przykład call, nie dowiemy się, w jaki sposób użytkownik wywołał przycisk działania. W wielu przypadkach jest to nieistotne, zatem można obydwu argumentom przypisać taką samą wartość. Jednak w razie konieczności odróżnienia obydwu sposobów wywołania musimy wstawić tu wartość inną niż obecna w argumencie queryActionMsg.
- suggestActionMsgColumn — wartości argumentów queryActionMsg i suggestActionMsg są stosowane globalnie wobec danej aktywności wyszukiwania oraz dostawcy propozycji. Nie ma możliwości zmiany znaczenia działania opartego na propozycji. Można jednak umieścić w metadanych informację o istnieniu dodatkowej kolumny w kurSORZE propozycji. W ten sposób Android pobierze informacje z tej kolumny i prześle ją aktywności jako część intencji wywołującej ACTION\_SEARCH. Co ciekawe, wartość tej dodatkowej kolumny jest przesyłana w intencji za pomocą identycznego, dodatkowego klucza, mianowicie SearchManager.ACTION\_MSG.

Spośród wymienionych atrybutów obowiązkowy jest kod przycisku. Ponadto musi być obecny przynajmniej jeden z pozostałych trzech atrybutów, aby przycisk działania został uruchomiony.

Jeżeli chcemy korzystać z atrybutu suggestActionMsgColumn, musimy zapisać tę kolumnę w klasie dostawcy propozycji. Gdybyśmy chcieli używać obydwu przycisków, musielibyśmy w kodzie z listingu 23.29 umieścić dwie dodatkowe kolumny typu string zdefiniowane w kurSORZE propozycji (listing 23.22), mianowicie kolumny call\_column i my\_column. W takim przypadku nasza tablica kolumn kursora powinna przypominać tabelę przedstawioną na listingu 23.30.

#### **Listing 23.30.** Przykładowe kolumny przycisków działania w kurSORZE propozycji

---

```
private static final String[] COLUMNS = {
    "_id", // musi zawierać tę kolumnę
    SearchManager.SUGGEST_COLUMN_TEXT_1,
    SearchManager.SUGGEST_COLUMN_TEXT_2,
```

```
SearchManager.SUGGEST_COLUMN_INTENT_DATA,  
SearchManager.SUGGEST_COLUMN_INTENT_ACTION,  
SearchManager.SUGGEST_COLUMN_SHORTCUT_ID,  
"call_column",  
"my_column"  
};
```

---

## Praca ze specyficznym dla aplikacji kontekstem wyszukiwania

Proces wyszukiwania w Androidzie umożliwia aktywności przekazanie dodatkowych danych wyszukiwania do wywołanej aktywności wyszukiwania. Omówimy teraz szczegóły tego procesu.

Pokazaliśmy wcześniej, że aktywność aplikacji może przesłonić metodę `onSearchRequested()`, dzięki czemu otrzymujemy wartość `false` i proces wyszukiwania zostaje wyłączony. Co ciekawe, w ten sam sposób możemy przekazać aktywności wyszukiwania dodatkowe dane, specyficzne dla aplikacji. Na listingu 23.31 został zaprezentowany przykład.

**Listing 23.31.** Przekazywanie dodatkowego kontekstu

---

```
public boolean onSearchRequested()  
{  
    Bundle applicationData = new Bundle();  
    applicationData.putString("string_key", "jakaś wartość typu string");  
    applicationData.putLong("long_key", 290904);  
    applicationData.putFloat("float_key", 2.0f);  
  
    startSearch(null, // Początkowy ciąg znaków kwerendy wyszukiwania  
               false, // nie „zaznaczaj początkowej kwerendy”  
               applicationData, // dodatkowe dane  
               false // nie wymusza wyszukiwania globalnego  
               );  
  
    return true;  
}
```

---

**Uwaga!**

Różnorodne funkcje interfejsu API `Bundle` są omówione pod adresem:  
<http://developer.android.com/reference/android/os/Bundle.html>.

Po uruchomieniu w ten sposób procesu wyszukiwania aktywność może wykorzystać obiekt `SearchManager.APP_DATA` do odczytania kompletnego danych aplikacji. Listing 23.32 przedstawia sposób odczytania każdego z powyższych pól.

**Listing 23.32.** Odzyskiwanie dodatkowego kontekstu

---

```
Bundle applicationData =  
    queryIntent.getBundleExtra(SearchManager.APP_DATA);  
if (applicationData != null)  
{  
    String s = applicationData.getString("string_key");
```

---

```

long l = applicationData.getLong("long_key");
float f = applicationData.getFloat("float_key");
}

```

---

Wcześniej omówiliśmy побieżnie metodę `startSearch()`. Jej opis znajdziemy pod następującym adresem URL, stanowiącym część dokumentacji klasy `Activity`: <http://developer.android.com/reference/android/app/Activity.html>.

Także tutaj metoda ta przyjmuje cztery wymienione poniżej argumenty:

- `initialQuery` (argument typu `String`),
- `selectInitialQuery` (argument logiczny),
- `applicationDataBundle` (argument typu `Bundle`),
- `globalSearchOnly` (argument logiczny).

Jeżeli pierwszy argument zostanie zastosowany, wypełni tekst kwerendy w polu QSB.

Wartość `true` w drugim argumencie spowoduje zaznaczenie tekstu. Użytkownik będzie mógł zamienić cały zaznaczony tekst kwerendy na jego poprawioną wersję. W przypadku wartości `false` kurSOR zostanie umieszczony na końcu tekstu kwerendy.

Trzeci argument stanowi oczywiście przygotowywany przez nas zestaw danych.

Jeżeli w czwartym argumencie zostanie umieszczona wartość `true`, zawsze będzie wywoływany proces wyszukiwania globalnego. Po wprowadzeniu wartości `false` najpierw zostanie wywołane wyszukiwanie lokalne (jeżeli jest dostępne); w przeciwnym razie będzie stosowane wyszukiwanie globalne.

## Odbońniki

Na zakończenie tego rozdziału chcielibyśmy podzielić się listą zasobów, które uznaliśmy za przydatne podczas jego pisania.

- [www.google.com/googlephone/AndroidUsersGuide.pdf](http://www.google.com/googlephone/AndroidUsersGuide.pdf) — zawarto tu przydatne materiały dla wersji 2.2.1 Androida, pomagające zrozumieć techniki korzystania z funkcji wyszukiwania przez użytkownika.
- [www.google.com/help/hc/pdfs/mobile/AndroidUsersGuide-30-100.pdf](http://www.google.com/help/hc/pdfs/mobile/AndroidUsersGuide-30-100.pdf) — instrukcja obsługi wersji 3.0 Androida. Adres ten był dość często zmieniany w ciągu ostatnich kilku miesięcy. W przypadku kolejnej zmiany powinniśmy bez trudu odnaleźć odpowiedni plik, wpisując w wyszukiwarce Google hasło: *Android User's Guide*.
- <http://developer.android.com/reference/android/app/SearchManager.html> — adres ten zawiera główną dokumentację dotyczącą procesu wyszukiwania w Androidzie, utworzoną przez firmę Google. To samo łącze stanowi również źródło materiałów na temat głównego obiektu odpowiedzialnego za proces wyszukiwania, mianowicie klasy `SearchManager`.
- [http://developer.android.com/reference/android/app/Activity.html#onNewIntent\(android.content.Intent\)](http://developer.android.com/reference/android/app/Activity.html#onNewIntent(android.content.Intent)) — w miarę tworzenia własnych aktywności wyszukiwania warto czasem konfigurować je w trybie `singleTop`, dzięki czemu zostanie wygenerowana metoda `onNewIntent()`. Tutaj znajdziemy informacje na jej temat.

- <http://developer.android.com/resources/samples/SearchableDictionary/index.html>  
— pod tym adresem znajduje się przykładowa implementacja dostawcy propozycji. Łącznie to wskazuje przede wszystkim kod źródłowy implementacji.
- <http://developer.android.com/reference/android/provider/SearchRecentSuggestions.html>  
— materiały dotyczące interfejsu API przeszukiwania ostatnich propozycji.
- <http://developer.android.com/guide/topics/fundamentals.html> — zaprezentowane tutaj dane pomagają zrozumieć aktywności, zadania oraz tryby uruchamiania, zwłaszcza tryb singleTop, często używany jako aktywność wyszukiwania.
- <http://developer.android.com/reference/android/os/Bundle.html> — dzięki temu odnośnikowi poznamy różne funkcje dostępne w obiekcie Bundle. Są to informacje przydatne zwłaszcza podczas implementacji danych wyszukiwania specyficznych dla aplikacji.
- [http://www.androidbook.com/notes\\_on\\_search](http://www.androidbook.com/notes_on_search) — na tej stronie znajdziemy wyniki badań autorów dotyczących procesu wyszukiwania w Androidzie. Nawet po opublikowaniu tej książki będziemy aktualizować zawartość tej witryny.
- <ftp://ftp.helion.pl/przyklady/and2ta.zip> — znajdziemy tu projekty testowe przygotowane z myślą o niniejszej książce. Obiektem naszego zainteresowania jest plik umieszczony w katalogu *ProAndroid3\_R23\_Wyszukiwanie*.

## Wyszukiwanie w tabletach

W wersji 3.0 Androida zasadnicza struktura interfejsu API wyszukiwania pozostała niezmieniona. Jednak pole QSB oraz ustawienia wyszukiwania (dostrzegalne zwłaszcza z punktu widzenia użytkownika) zostały nieco zmodyfikowane w celu umożliwienia korzystania z większej powierzchni. Poza tym wszystkie omówione w tym rozdziale koncepcje znajdują zastosowanie również w przypadku tabletów.

## Podsumowanie

W tym rozdziale zaprezentowaliśmy przede wszystkim szczegółowe omówienie działania procesu wyszukiwania w Androidzie. Wyjaśniliśmy, w jaki sposób aktywności i dostawcy propozycji współpracują z procesem wyszukiwania. Zademonstrowaliśmy sposób wykorzystania klasy `SearchRecentSuggestionsProvider`.

Zaprojektowaliśmy od podstaw niestandardowego dostawcę propozycji, a w międzyczasie dokładnie opisaliśmy kurSOR podpowiedzi i jego kolumny. Przyjrzaliśmy się identyfikatorom URI odpowiedzialnym za uzyskiwanie danych od dostawców propozycji. Zaprezentowaliśmy wiele fragmentów kodu źródłowego, ułatwiających opracowanie i zaimplementowanie własnych strategii wyszukiwania.

Dzięki elastyczności samego kurSora propozycji wyszukiwanie w Androidzie przekształca się z prostego procesu w przewodnik po informacjach dostępnych w zasięgu ręki.

# Analiza interfejsu przetwarzania tekstu na mowę

Począwszy od wersji 1.6 środowiska Android, dostępny stał się silnik odpowiadający za syntezę mowy, nazwany Pico. Za jego pomocą aplikacje w Androidzie mogą przekształcać ciągi znaków tekstowych na dźwięk — mowę z akcentem typowym dla wybranego języka. Technologia przetwarzania tekstu na mowę umożliwia użytkownikowi korzystanie z urządzenia bez konieczności spoglądania na ekran. W przypadku platformy mobilnej jest to niezmiernie istotna funkcja. Ilu ludziom zdarzyło się wyjść przypadkiem na środek jezdni w trakcie czytania wiadomości tekstowej? Czy nie wystarczyłoby po prostu odsłuchać tej wiadomości? A gdyby można było posłuchać przewodnika turystycznego, zamiast czytać go w trakcie zwiedzania? Istnieje olbrzymia liczba aplikacji, w przypadku których zaimplementowanie mowy zwiększyłoby ich użyteczność. W tym rozdziale przyjrzymy się klasie `TextToSpeech` i wyjaśnimy, co zrobić, aby wprowadzony przez nas tekst został wypowiedziany przez urządzenie. Nauczymy się także zarządzać dostępnymi ustawieniami regionalnymi, językami i głosami.

## Podstawy technologii przetwarzania tekstu na mowę w Androidzie

Zanim wykorzystamy funkcję TTS (ang. *Text To Speech* — tekst na mowę) we własnej aplikacji, sprawdźmy, jak działa w rzeczywistości. W emulatorze lub w urządzeniu (wersja systemu co najmniej 1.6) otworzymy główny ekran *Ustawienia*, stamtąd przejdźmy do widoku *Ustawienia głosowe* i wybierzmy element *Ustawienia przetwarzania tekstu na mowę* (lub, w zależności od posiadanej wersji, *Synteza mowy*). Kliknijmy opcję *Posłuchaj przykładu*, dzięki czemu usłyszmy słowa „This is an example of speech synthesis in English with Pico” (co oznacza: „Jest to przykład syntezy mowy w języku angielskim za pomocą silnika Pico”). Zwróćmy uwagę na pozostałe elementy listy (rysunek 24.1).



Rysunek 24.1. Ekran ustawień silnika przetwarzania tekstu na mowę

Możemy zmienić używany język oraz prędkość mowy. Opcja *Język* zarówno tłumaczy wymawiane słowa, jak i zmienia akcent głosu, chociaż ciągle wymawiany będzie tekst: „Jest to przykład syntezy mowy” w tłumaczeniu na inne języki. Pamiętajmy, że funkcja TTS obsługuje jedynie generowanie głosu. Tłumaczenie zapewnia oddzielnego składnika, na przykład usługa tłumacza Google, omówiona w rozdziale 11. W trakcie przykładowej implementacji funkcji TTS w naszej aplikacji będziemy chcieli, aby syntezowany głos mówił w sposób zgodny z wybranym językiem, zatem żeby francuski tekst był wypowiadany przez głos mówiący w sposób właściwy dla języka francuskiego. Prędkość mowy jest konfigurowana w zakresie od *Bardzo wolno* do *Bardzo szybko*.

Należy zachować ostrożność przed ustawieniem opcji *Zawsze używaj moich ustawień*. Wybranie jej w opcjach systemowych przez dowolnego użytkownika może spowodować nieprzewidywalne zachowanie aplikacji, ponieważ definiowane przez nią parametry mogą przesłonić ustawienia tego programu.

Wraz z pojawiением się wersji 2.2 Androida uzyskaliśmy możliwość korzystania z mechanizmów TTS innych niż Pico (zatem we wcześniejszych wersjach systemu w widoku ustawień nie byłaby dostępna opcja *Domyślny mechanizm*). Uzyskujemy w ten sposób większe możliwości, ponieważ silnik Pico nie nadaje się do wszystkich zastosowań. Nawet w przypadku posiadania kilku mechanizmów TTS w urządzeniu znajduje się tylko jedna usługa przetwarzania mowy. Jest ona współdzielona przez wszystkie aktywności urządzenia, zatem musimy mieć świadomość, że nie tylko określona aplikacja może korzystać z tej funkcji. Oznacza to także, że nie możemy być pewni, kiedy (jeżeli w ogóle) nasz tekst zostanie wypowiedziany. Jednak dzięki interfejsowi funkcji TTS posiadamy dostęp do metod zwrotnych, zatem mamy pojęcie, co się dzieje z wysłanym przez nas tekstem. Usługa TTS będzie śledziła wybrany przez nas mechanizm przetwarzania mowy oraz będzie go wykorzystywała do zlecanych operacji. Może ona korzystać z dowolnego silnika wywoływanego przez aktywność, zatem pozostałe aplikacje mogą korzystać z innych silników przetwarzania mowy, a my nie musimy się przejmować tym aspektem.

Sprawdźmy, co się dzieje podczas konfigurowania tych ustawień funkcji TTS. Android uruchamia poza wzrokiem użytkownika usługę przetwarzania tekstu na mowę oraz silnik Pico, czyli wielojęzyczny mechanizm syntezy mowy. Aktywność ustawień, którą uruchomiliśmy, zainicjalizowała silnik przetwarzający bieżący język i prędkość mowy. Po kliknięciu opcji *Posłuchaj przykładu*

chaj przykładu aktywność preferencji przesyła tekst do usługi, po czym z kolei silnik przetwarzania mowy wymawia ten tekst poprzez głośnik. Silnik Pico rozbija tekst na fragmenty, które potrafi wymówić, i składa te porcje dźwięków w całkiem naturalny sposób. Algorytmy tworzące ten silnik są o wiele bardziej złożone — możemy jednak udawać, że mamy do czynienia z magią. Na szczęście dla nas magia ta zajmuje bardzo niewiele pamięci oraz pojemności dyskowej, zatem silnik Pico jest idealnym dodatkiem do telefonu.

W poniższym przykładzie utworzymy aplikację, która będzie odczytywała na głos wpisywane przez nas informacje. Nie jest ona skomplikowana; została zaprojektowana w celu pokazania, jak łatwo można zaimplementować funkcję przetwarzania tekstu na mowę. Na początku utworzymy nowy projekt w Androidzie, korzystając z artefaktów zamieszczonych na listingu 24.1.

**Uwaga!**

Na końcu rozdziału zamieszczamy odnośniki do pliku zawierającego omawiane tu projekty. W ten sposób można zimportować te projekty bezpośrednio do środowiska Eclipse.

**Listing 24.1.** Kody XML i Java prostej wersji demonstracyjnej funkcji TTS

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <EditText android:id="@+id/wordsToSpeak"
        android:hint="Wpisz słowa do wymówienia"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>

    <Button android:id="@+id/speak"
        android:text="Powiedz"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="doSpeak"
        android:enabled="false" />

</LinearLayout>

// Jest to plik MainActivity.java
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.speech.tts.TextToSpeech;
import android.speech.tts.TextToSpeech.OnInitListener;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class MainActivity extends Activity implements OnInitListener {
    private EditText words = null;
    private Button speakBtn = null;
```

```
private static final int REQ_TTS_STATUS_CHECK = 0;
private static final String TAG = "TTS Demo";
private TextToSpeech mTts;

/** Wywoływanie podczas pierwszego utworzenia aktywności. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    words = (EditText)findViewById(R.id.wordsToSpeak);
    speakBtn = (Button)findViewById(R.id.speak);

    // Upewnia się, czy funkcja TTS istnieje i jest sprawna
    Intent checkIntent = new Intent();
    checkIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
    startActivityForResult(checkIntent, REQ_TTS_STATUS_CHECK);
}

public void doSpeak(View view) {
    mTts.speak(words.getText().toString(),
        TextToSpeech.QUEUE_ADD, null);
}

protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQ_TTS_STATUS_CHECK) {
        switch (resultCode) {
            case TextToSpeech.Engine.CHECK_VOICE_DATA_PASS:
                // Funkcja TTS jest sprawna i działa
                mTts = new TextToSpeech(this, this);
                Log.v(TAG, "Silnik Pico jest poprawnie zainstalowany");
                break;
            case TextToSpeech.Engine.CHECK_VOICE_DATA_BAD_DATA:
            case TextToSpeech.Engine.CHECK_VOICE_DATA_MISSING_DATA:
            case TextToSpeech.Engine.CHECK_VOICE_DATA_MISSING_VOLUME:
                // Brakujące dane, instaluje je
                Log.v(TAG, "Potrzebny jest język: " + resultCode);
                Intent installIntent = new Intent();
                installIntent.setAction(
                    TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
                startActivity(installIntent);
                break;
            case TextToSpeech.Engine.CHECK_VOICE_DATA_FAIL:
            default:
                Log.e(TAG, "Niepowodzenie. Funkcja TTS jest najwidoczniej niedostępna");
        }
    } else {
        // Otrzymał coś innego
    }
}

public void OnInit(int status) {
    // Skoro funkcja TTS działa, aktywujemy przycisk
    if( status == TextToSpeech.SUCCESS) {
```

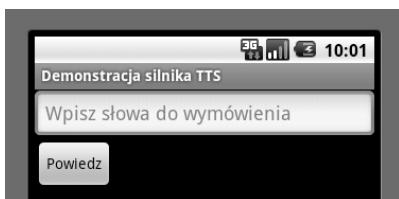
```

        speakBtn.setEnabled(true);
    }
}

@Override
public void onPause()
{
    super.onPause();
    // Przy zmianie aktywności pierwszoplanowej przestaje mówić
    if( mTts != null)
        mTts.stop();
}
@Override
public void onDestroy()
{
    super.onDestroy();
    mTts.shutdown();
}
}

```

W powyższym przykładzie interfejsem użytkownika jest widok `EditText`, umożliwiający wpisywanie słów do głośnego odczytania, oraz przycisk inicjalizujący syntezę mowy (rysunek 24.2). Nasz przycisk zawiera metodę `doSpeak()`, pobierającą ciąg znaków tekstowych z widoku `EditText` i kolejującą ten tekst za pomocą funkcji `speak()` zawierającej argument `QUEUE_ADD`. Pamiętajmy, że silnik TTS jest współdzielony, więc nasz tekst może być kolejkowany za dowolnym innym obiektem (nie zdarza się to jednak zbyt często). Inną opcją poza `QUEUE_ADD` jest `QUEUE_FLUSH`, powodującą usunięcie z kolejki innego tekstu i natychmiastowe odtworzenie naszego. Na końcu metody `onCreate()` uruchamiamy obiekt `Intent`, żądający od funkcji TTS, aby powiadomiła nas o poprawności wpisanego tekstu. Ponieważ wymagamy zwrotu odpowiedzi, stosujemy metodę `startActivityForResult()` i przekazujemy kod żądania. Odpowiedź otrzymujemy w metodzie `onActivityResult()`, w której szukamy argumentu `CHECK_VOICE_DATA_PASS`. Ponieważ silnik TTS może przekazać różne typy kodu `resultCode` oznaczającego poprawność wykonanych działań, nie możemy korzystać z argumentu `RESULT_OK`. Inne uzyskiwane wartości możemy poznać, przeglądając instrukcję przełączania.



**Rysunek 24.2.** Demonstracja interfejsu użytkownika silnika TTS

Jeżeli otrzymamy z powrotem argument `CHECK_VOICE_DATA_PASS`, zostanie utworzony obiekt `TextToSpeech`. Zauważmy, że aktywność `MainActivity` implementuje obiekt nasłuchujący `OnInitListener`. Dzięki temu otrzymujemy (wraz z metodą `onInit()`) metodę zwrotną w przypadku utworzenia i udostępnienia interfejsu TTS. Jeżeli wewnętrz metody `onInit()` zostanie przekazana wartość `SUCCESS`, urządzenie jest gotowe do odczytania tekstu, a w interfejsie użytkownika zostaje uaktywniony przycisk. Warto również zwrócić uwagę na wywołanie funkcji `stop()` w metodzie `onPause()`, a także na wywołanie funkcji `shutdown()` w metodzie

`onDestroy()`. Funkcję `stop()` wywołujemy, ponieważ jeżeli nasza aplikacja nie będzie przetwarzana na pierwszym planie, będzie to oznaczać, że użytkownik jest zajęty czymś innym, zatem głośne odczytywanie tekstu powinno zostać przerwane. Nie chcemy zakłócać żadnej związanej z dźwiękiem czynności w innej aktywności, która pojawiła się na ekranie. Za pomocą metody `shutdown()` powiadamiamy system, że skończyliśmy korzystać z silnika TTS, a zasoby, jeśli nie są już wykorzystywane, stają się niepotrzebne i mogą zostać zwolnione.

Warto poeksperymentować na tym przykładzie. Sprawdźmy różne zdania lub wyrażenia. Umieścimy duży blok tekstu, aby sprawdzić, jak syntezator się sprawuje przy dłuższych wypowiedziach. Zastanówmy się teraz, co by się stało, gdyby w trakcie czytania takiego dużego bloku tekstu praca naszej aplikacji została przerwana, na przykład z powodu innej aplikacji uzyskującej dostęp do silnika TTS, działającej w trybie `QUEUE_FLUSH`, albo gdyby nasza aplikacja zeszła po prostu na dalszy plan. Przetestujmy takie zachowanie, wciskając przycisk ekranu startowego w trakcie odczytywania dużej partii tekstu. Z powodu implementacji funkcji `stop()` w metodzie `onPause()` proces przetwarzania tekstu na mowę zostaje zatrzymany, nawet jeśli aplikacja cały czas działa w tle. Skąd mamy wiedzieć, w którym miejscu aplikacja skończyła odczyt, w przypadku gdy znów znajdzie się na pierwszym planie? Byłaby to przydatna funkcjonalność, gdyby istniał sposób zapamiętywania punktu przerwania odczytywania tekstu, aby po ponownym uruchomieniu aplikacji proces głośnego odczytywania tekstu rozpoczął się od punktu przerwania, a przynajmniej w jego pobliżu. Istnieje rozwiązanie, lecz jest nieco pracochłonne.

## Używanie wyrażeń do śledzenia toku wypowiedzi

Silnik TTS może zwrotnie wywołać aplikację po odczytaniu fragmentu tekstu, w świecie przetwarzania tekstu na mowę zwanego **wyrażeniem**. Wywołanie to konfigurujemy za pomocą metody `setOnUtteranceCompletedListener()` wobec wystąpienia silnika TTS, w powyższym przykładzie wobec zmiennej `mTts`. W trakcie wywoływanego metody `speak()` możemy dodać parę nazwa – klucz, abyśmy zostali powiadomieni przez silnik TTS o zakończeniu odzwierciedlania wyrażenia. Poprzez wysyłanie niepowtarzalnych identyfikatorów wyrażeń do silnika TTS wiemy, które wyrażenia zostały przeczytane, a które jeszcze czekają w kolejce. Jeżeli aplikacja wróci na pierwszy plan po uprzednim przerwaniu jej działania, możemy wznowić przetwarzanie tekstu na mowę od pierwszego nieodczytanego wyrażenia. W tym celu należy w powyższym przykładowym projekcie zmienić kod, tak jak zostało to pokazane na listingu 24.2, lub zatrudnić do projektu `TTSDemo2`, zamieszczonego na naszej stronie internetowej.

**Listing 24.2.** Zmiany aktywności `MainActivity` umożliwiające śledzenie wyrażeń

```
// Dodajmy następujące instrukcje importowania
import java.util.HashMap;
import java.util.StringTokenizer;
import android.speech.tts.TextToSpeech.OnUtteranceCompletedListener;

// Zmieńmy klasę MainActivity
public class MainActivity extends Activity implements OnInitListener,
OnUtteranceCompletedListener {

    // Dodajmy następujące prywatne pola
    private int uttCount = 0;
    private int lastUtterance = -1;
    private HashMap<String, String> params = new HashMap<String, String>();
```

```

// Zmodyfikujmy metodę onInit
public void onInit(int status) {
    // Teraz silnik TTS jest przygotowany, więc uaktywniamy przycisk
    if( status == TextToSpeech.SUCCESS) {
        speakBtn.setEnabled(true);
        mTts.setOnUtteranceCompletedListener(this);
    }
}

// Dodajemy nową metodę onUtteranceCompleted
public void onUtteranceCompleted(String uttId) {
    Log.v(TAG, "Utworzono komunikat dla obiektu uttId: " + uttId);
    lastUtterance = Integer.parseInt(uttId);
}

// Modyfikujemy metodę doSpeak
public void doSpeak(View view) {
    StringTokenizer st = new StringTokenizer(words.getText().toString(),".,");
    while (st.hasMoreTokens()) {
        params.put(TextToSpeech.Engine.KEY_PARAM_UTTERANCE_ID,
                   String.valueOf(uttCount++));
        mTts.speak(st.nextToken(), TextToSpeech.QUEUE_ADD, params);
    }
}

```

W pierwszej kolejności trzeba się upewnić, że nasza aktywność `MainActivity` implementuje również interfejs `OnUtteranceCompletedListener`. Dzięki temu uzyskamy metodę zwrotną z silnika TTS po zakończeniu odczytywania wyrażeń. Musimy także zmodyfikować metodę `doSpeak()` przycisku, aby przekazywała dodatkowe informacje, łączące identyfikator wyrażenia z każdym fragmentem wysyłanego tekstu. W naszej nowej wersji aplikacji separatorami wyrażeń będą przecinki i kropki. Następnie utworzymy pętlę przekazywania wyrażeń za pomocą argumentu `QUEUE_ADD`, a nie `QUEUE_FLUSH` (nie chcemy sami sobie przerywać!), oraz unikatowego identyfikatora wyrażenia, który w istocie jest prostym licznikiem inkrementacyjnym, oczywiście przekonwertowanym na dane typu `String`. Z tego powodu identyfikatorem wyrażenia może być dowolny tekst; nie jesteśmy ograniczeni wyłącznie do cyfr. W rzeczywistości możemy wykorzystać dowolny ciąg znaków, chociaż zbyt długie identyfikatory typu `string` mogą być niekorzystne pod kątem wydajności. Musimy tak zmodyfikować metodę `onInit()`, abyśmy mogli rejestrować wywołania oznaczające zakończenie przetwarzania wyrażenia. Powinniśmy także wprowadzić na końcu metodę zwrotną `onUtteranceCompleted()` wywołującą silnik TTS po zakończeniu przetwarzania wyrażenia. W naszym przykładzie każde zakończone wyrażenie spowoduje utworzenie wpisu w dzienniku w narzędziu `LogCat`.

Po uruchomieniu naszej nowej aplikacji wpiszmy jakiś tekst zawierający przecinki i kropki, a następnie kliknijmy przycisk *Powiedz*. Obserwujmy okno narzędzia *LogCat* w trakcie słuchania głosu odczytującego tekst. Zauważmy, że tekst zostanie natychmiast zakolejkowany, a po zakończeniu każdego wyrażenia następuje wykonanie metody zwrotnej, powodującej wyświetlenie odpowiedniego wpisu w dzienniku. Jeżeli w trakcie odczytu tekstu przerwiemy działanie tej aplikacji, na przykład wciskając przycisk ekranu startowego, zostaną przerwane zarówno proces przetwarzania tekstu na mowę, jak i metody zwrotne. Znamy teraz ostatnie odczytane wyrażenie i możemy rozpocząć od tego miejsca dalsze odsłuchiwanie.

## Zastosowanie plików dźwiękowych do przetwarzania tekstu na mowę

W silniku TTS wprowadzono rozwiązań pozwalające na poprawne wypowiadanie słów i wyrażeń, które w domyślnym przypadku byłyby źle wymawiane. Jeśli na przykład wpiszymy Don Quixote jako wymawiany tekst, imię to zostanie niepoprawnie przeczytane. Trzeba jednak uczciwie przyznać, że choć silnik TTS potrafi dobrze przewidzieć brzmienie poszczególnych wyrazów, to trudno się spodziewać, że zna wszystkie wyjątki. Jak zatem można sobie z tym poradzić? Jednym ze sposobów jest zarejestrowanie fragmentu dźwiękowego, który będzie odtwarzany zamiast domyślnego głosu. Aby uzyskać efekt brzmienia tego samego głosu, spowodujemy, że silnik TTS wypowie słowo w pożądany sposób, zarejestrujemy wynik, a następnie poinformujemy mechanizm przetwarzania tekstu na mowę o zastąpieniu domyślnego dźwięku nagrany przed chwilą dźwiękiem. Sztuka polega na dostarczeniu tekstu, który będzie odczytywany w pożądany przez nas sposób. Zobaczmy, jak się to robi.

Utwórzmy nowy projekt Androida w środowisku Eclipse. Wykorzystajmy plik XML z listingu 24.3 do utworzenia głównego układu graficznego. Cały proces uprościmy, umieszczając tekst bezpośrednio w pliku układu graficznego, a nie za pomocą odnośników do ciągów znaków. W przypadku zwykłej aplikacji umieścilibyśmy tu identyfikator zasobu typu `string`. Wygląd układu graficznego został przedstawiony na rysunku 24.3.

**Listing 24.3.** Plik XML układu graficznego demonstrujący zapisany plik audio z zarejestrowaną wymową tekstu

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <EditText android:id="@+id/wordsToSpeak"
        android:text="Dohn Keyhotay"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>

    <Button android:id="@+id/speakBtn"
        android:text="Powiedz"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="doButton"
        android:enabled="false" />

    <TextView android:id="@+id/filenameLabel"
        android:text="Nazwa pliku:"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

    <EditText android:id="@+id/filename"
        android:text="/sdcard/donquixote.wav"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
```

```
<Button android:id="@+id/recordBtn"
    android:text="Rejestruj"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:enabled="false"
    android:onClick="doButton"/>

<Button android:id="@+id/playBtn"
    android:text="Odtwórz"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="doButton"
    android:enabled="false" />

<TextView android:id="@+id/useWithLabel"
    android:text="Wykorzystaj z:"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

<EditText android:id="@+id/realText"
    android:text="Don Quixote"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>

<Button android:id="@+id/assocBtn"
    android:text="Skojarz"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="doButton"
    android:enabled="false" />

</LinearLayout>
```



Rysunek 24.3. Interfejs użytkownika aplikacji demonstracyjnej kojarzącej plik dźwiękowy z tekstem

Potrzebne jest pole, w którym będzie przechowywany specjalny tekst, nagrywany przez silnik TTS do pliku dźwiękowego. W układzie graficznym zamieściliśmy również nazwę tego pliku. Ostatnim etapem będzie powiązanie naszego pliku dźwiękowego z rzeczywistym ciągiem znaków, dla którego ten plik ma być odtwarzany.

Przyjrzyjmy się teraz kodowi Java aktywności `MainActivity` (listing 24.4). W metodzie `onCreate()` konfigurujemy procedury obsługi kliknięć przycisków *Powiedz*, *Odtwórz*, *Rejestruj* i *Skojarz*, a następnie inicjalizujemy silnik TTS za pomocą intencji. Pozostała część kodu składa się z wywołań przetwarzających wyniki zwarcane z intencji sprawdzającej poprawność konfiguracji silnika TTS, przetwarzających wyniki inicjalizacji z silnika TTS oraz ze zwykłych metod zwrotnych, wstrzymujących i zamkujących naszą aktywność.

**Listing 24.4.** Kod Java przedstawiający proces zapisywania pliku dźwiękowego dla tekstu

---

```
import java.io.File;
import java.util.ArrayList;

import android.app.Activity;
import android.content.Intent;
import android.media.MediaPlayer;
import android.os.Bundle;
import android.speech.tts.TextToSpeech;
import android.speech.tts.TextToSpeech.OnInitListener;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivity extends Activity implements OnInitListener {
    private EditText words = null;
    private Button speakBtn = null;
    private EditText filename = null;
    private Button recordBtn = null;
    private Button playBtn = null;
    private EditText useWith = null;
    private Button assocBtn = null;
    private String soundFilename = null;
    private File soundFile = null;
    private static final int REQ_TTS_STATUS_CHECK = 0;
    private static final String TAG = "Demonstracja silnika TTS";
    private TextToSpeech mTts = null;
    private MediaPlayer player = null;

    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        words = (EditText)findViewById(R.id.wordsToSpeak);
        filename = (EditText)findViewById(R.id.filename);
        useWith = (EditText)findViewById(R.id.realText);
```



```
        break;
    }
}

protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
    if (requestCode == REQ_TTS_STATUS_CHECK) {
        switch (resultCode) {
            case TextToSpeech.Engine.CHECK_VOICE_DATA_PASS:
                // Silnik TTS jest utworzony i uruchomiony
                mTts = new TextToSpeech(this, this);
                Log.v(TAG, "Silnik Pico został poprawnie zainstalowany!");
                ArrayList<String> available =
                    data.getStringArrayListExtra("availableVoices");
                break;
            case TextToSpeech.Engine.CHECK_VOICE_DATA_BAD_DATA:
            case TextToSpeech.Engine.CHECK_VOICE_DATA_MISSING_DATA:
            case TextToSpeech.Engine.CHECK_VOICE_DATA_MISSING_VOLUME:
                // Brakuje danych, należy je zainstalować
                Log.v(TAG, "Potrzebny pakiet językowy: " + resultCode);
                Intent installIntent = new Intent();
                installIntent.setAction(
                    TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
                startActivity(installIntent);
                break;
            case TextToSpeech.Engine.CHECK_VOICE_DATA_FAIL:
            default:
                Log.e(TAG, "Wystąpił błąd. Silnik TTS jest niedostępny");
        }
    } else {
        // Jakaś innna operacja
    }
}

public void OnInit(int status) {
    // Po przygotowaniu silnika TTS uaktywniamy przyciski
    if( status == TextToSpeech.SUCCESS) {
        speakBtn.setEnabled(true);
        recordBtn.setEnabled(true);
    }
}

@Override
public void onPause()
{
    super.onPause();
    // Jeżeli aplikacja schodzi z pierwszego planu, zatrzymujemy odtwarzanie
    if(player != null) {
        player.stop();
    }
    // Jeżeli aplikacja schodzi z pierwszego planu, przerywamy syntezę mowy
    if( mTts != null)
        mTts.stop();
}
```

```

@Override
public void onDestroy()
{
    super.onDestroy();
    if(player != null) {
        player.release();
    }
    if( mTts != null) {
        mTts.shutdown();
    }
}
}

```

---

Aby ten przykładowy kod zadziałał, musimy w pliku *AndroidManifest.xml* dodać uprawnienie `android.permission.WRITE_EXTERNAL_STORAGE`. Po uruchomieniu tego przykładu powinniśmy ujrzeć interfejs UI, zilustrowany na rysunku 24.3.

Zarejestrujemy poprawne brzmienie tekstu „Don Quixote”, nie będziemy więc korzystać z rzeczywistych słów. Musimy skonstruować tekst, który zostanie wypowiedziany z właściwym brzmieniem słów. Kliknijmy przycisk *Powiedz*, aby usłyszeć, jak brzmi utworzona przez nas imitacja tekstu. Nie najgorzej! Wybierzmy teraz przycisk *Rejestruj*, dzięki czemu brzmienie tego wyrazu zostanie zapisane w pliku WAV. Po pomyślnym zarejestrowaniu dźwięku staną się aktywne przyciski *Odtwórz* i *Skojarz*. Kliknijmy przycisk *Odtwórz*, a usłyszmy nagrany plik WAV bezpośrednio w odtwarzaczu MediaPlayer. Jeżeli odpowiada nam brzmienie tego wyrazu, użyjmy przycisku *Skojarz*. W silniku TTS zostanie wywołana metoda `addSpeech()`, która powiąże następnie nasz nowy plik dźwiękowy z ciągiem znaków umieszczonym w polu *Wykorzystaj z*. Jeżeli cały proces przebiegnie pomyślnie, wpiszmy w górnym widoku *EditText* wyraz *Don Quixote* i kliknijmy przycisk *Powiedz*. Teraz imię to zostaje odczytane we właściwy sposób.

Zwróćmy uwagę, że metoda `synthesizeToFile()` zapisuje dźwięk wyłącznie do formatu WAV, niezależnie od wstawionego rozszerzenia. Istnieje jednak możliwość skojarzenia innych formatów dźwiękowych za pomocą metody `addSpeech()` — na przykład plików MP3. Pliki MP3 nie mogą zostać utworzone za pomocą metody `synthesizeToFile()` silnika TTS.

Zastosowania tej metody w przypadku mowy są bardzo ograniczone. W przypadku scenariusza z nieskojarzonymi słowami — to znaczy gdy nie wiemy, jakie słowa będą wypowiadane — nie ma możliwości przygotowaniaawczasu wszystkich plików dźwiękowych, które posłużłyby do „naprawienia” wyrazów niepoprawnie syntezowanych przez silnik Pico. W scenariuszach zawierających skojarzone słowa — na przykład odczyt prognozy pogody — możemy przetestować wszystkie wyrazy w aplikacji, aby znaleźć i odpowiednio zmodyfikować te, które nie brzmią właściwie. Możemy na przykład przygotować wcześniej plik dźwiękowy gotowy do wypowiedzenia nazwy firmy lub nawet nazwiska!

Istnieje jednak ograniczenie stosowania tej metody: tekst przekazywany metodzie `speak()` musi idealnie odpowiadać tekstowi wykorzystanemu do wywołania metody `addSpeech()`. Niestety, nie możemy zapewnić pliku dźwiękowego dla pojedynczego wyrazu, a następnie oczekwać, że silnik TTS zastosuje ten plik dla wspomnianego słowa, podczas gdy jest ono przekazywane metodzie `speak()` jako część dłuższego zdania. Aby nasz plik dźwiękowy został odtworzony, musimy wprowadzić tekst dokładnie odpowiadający temu plikowi. Jeden wyraz mniej lub więcej, a silnik Pico gubi się i stara się syntezować dźwięk najlepiej, jak potrafi.

Pewnym rozwiązaniem jest podzielenie tekstu na wyrazy i przekazywanie każdego z nich do silnika TTS oddzielnie. Chociaż pliki dźwiękowe będą w ten sposób odtwarzane (oczywiście musielibyśmy oddziennie zarejestrować brzmienie wyrazów „Don” i „Quixote”), sama mowa będzie brzmiała bardzo nierównomiernie, jak gdyby każdy wyraz był jednocześnie całym zdaniem. Takie podejście nie przeszkała w pewnych aplikacjach. Wykorzystywanie plików dźwiękowych nadaje się najlepiej w przypadku predefiniowanych zestawów wyrazów lub wyrażeń, o których doskonale wiemy, w jakich momentach są wypowiadane.

Co zatem powinniśmy zrobić, gdy wiemy, że pojawią się w zdaniach słowa, które nie będą poprawnie wypowiadane przez silnik Pico? Jedną z metod jest wyszukanie kłopotliwych wyrazów i zastąpienie ich imitacjami tych słów, które po odczytaniu brzmiałby tak, jak powinny brzmieć właściwe wyrazy. Nie musimy pokazywać użytkownikowi tekstu umieszczonego w metodzie `speak()`. Może wystarczy więc zastąpić wyraz „Quixote” słowem „Keyhotay”, zanim wywołamy metodę `speak()`. W rezultacie otrzymamy właściwe brzmienie wyrazów, a użytkownik nie będzie miał pojęcia o naszej sztuczce. Z perspektywy zarządzania zasobami przechowywanie ciągu znaków imitacji jest o wiele bardziej skuteczne od przechowywania pliku dźwiękowego, nawet jeśli oznacza to ciągłe wywoływanie silnika Pico. Musiał być on wywoływany dla pozostały części tekstu, więc tak naprawdę nie tracimy wiele. Nie należy jednak zbytnio przeceniać zdolności przewidywania silnika Pico. Mamy na myśli, że algorytmy silnika Pico potrafią w intelligentny sposób składać wyrazy i jeżeli spróbujemy wykonywać ich obowiązki, możemy szybko popaść w kłopoty.

W powyższym przykładzie zarejestrowaliśmy plik dźwiękowy dla fragmentu tekstu, dzięki czemu podczas czytania tego tekstu silnik TTS uzyskał dostęp do źródła audio, zamiast dokonać syntezy mowy. Można się spodziewać, że odtworzenie małego pliku dźwiękowego obciąża mniejszą ilość zasobów urządzenia niż uruchomienie silnika TTS i nawiązanie z nim połączenia. Zatem jeżeli posiadamy zarządzaną pulę wyrazów, które mają być wymawiane przez urządzenie, może nam się opłacić utworzenie biblioteki plików dźwiękowych, nawet jeśli silnik Pico bezbłędnie je wymawia. W ten sposób aplikacja będzie działać szybciej. Prawdopodobnie w przypadku posiadania niewielkiej liczby plików dźwiękowych będziemy również obciążać mniejszą ilość pamięci. W przypadku takiego rozwiązania przydatne może się okazać wywołanie następującej metody:

```
TextToSpeech.addSpeech(String text, String packagename, int soundFileResourceId)
```

Jest bardzo prosty sposób dodania plików dźwiękowych do silnika TTS. Argument `text` definiuje ciąg znaków, dla którego będzie odtwarzany plik audio, `packagename` stanowi nazwę pakietu, w którym jest przechowywany ten zasób, a `soundFileResourceId` jest identyfikatorem zasobu tego pliku dźwiękowego. Pliki dźwiękowe powinny być przechowywane w katalogu `/res/raw` aplikacji. Podczas uruchamiania aplikacji dodajemy przygotowane pliki dźwiękowe do silnika TTS poprzez odniesienie się do ich identyfikatora zasobów (np. `R.raw.quixote`). Oczywiście będzie potrzebna jakaś baza danych albo predefiniowana lista, dzięki której będzie wiadomo, jakiemu tekstowi odpowiada dany plik dźwiękowy. Jeżeli internacjonalizujemy aplikację, możemy przechowywać alternatywne pliki dźwiękowe w odpowiednim podkatalogu `/res/raw;` na przykład `/res/raw-fr` dla plików języka francuskiego.

## Zaawansowane funkcje silnika TTS

Po zapoznaniu się z podstawami technologii przetwarzania tekstu na mowę zbadajmy nieco bardziej zaawansowane funkcje silnika Pico. Zaczniemy od konfigurowania strumieni dźwiękowych, co umożliwia kierowanie wypowiedzi do odpowiedniego kanału wyjściowego audio.

Następnie omówimy pojęcia ikony akustycznej oraz odtwarzania ciszy. Rozdział zamkniami tematami konfiguracji ustawień językowych oraz wywołań rozmaitych metod.

## Konfiguracja strumieni audio

Skorzystaliśmy wcześniej z parametrów `HashMap` do przekazania dodatkowych argumentów silnikowi TTS. Jeden z przekazywanych argumentów (`KEY_PARAM_STREAM`) określa, które źródło dźwiękowe ma zostać użyte przez silnik TTS dla odtwarzanego tekstu. W tabeli 24.1 została zamieszczona lista dostępnych strumieni audio.

**Tabela 24.1.** Dostępne strumienie audio

Strumień audio	Opis
<code>STREAM_ALARM</code>	Strumień dźwiękowy dla alarmów.
<code>STREAM_DTMF</code>	Strumień dźwiękowy dla sygnałów DTMF (np. sygnałów wciskania przycisków).
<code>STREAM_MUSIC</code>	Strumień dźwiękowy dla odtwarzania muzyki.
<code>STREAM_NOTIFICATION</code>	Strumień dźwiękowy dla powiadomień.
<code>STREAM_RING</code>	Strumień dźwiękowy dla dzwonka telefonicznego.
<code>STREAM_SYSTEM</code>	Strumień dźwiękowy dla dźwięków systemowych.
<code>STREAM_VOICE_CALL</code>	Strumień dźwiękowy dla rozmów telefonicznych.

Jeżeli syntezowane wyrażenie jest związane z alarmem, powinniśmy zdefiniować w silniku TTS odtwarzanie dźwięku poprzez strumień audio przeznaczony dla alarmów. Zatem przed wywołaniem metody `speak()` musimy wprowadzić następujący wiersz kodu:

```
params.put(TextToSpeech.Engine.KEY_PARAM_STREAM,
           String.valueOf(AudioManager.STREAM_ALARM));
```

Na listingu 24.2 pokazaliśmy sposób konfigurowania i przekazywania parametrów `HashMap` dla metody `speak()`. Do parametrów `HashMap` precyzujących strumień audio możemy również dołączyć identyfikator wyrażenia.

## Stosowanie ikon akustycznych

Istnieje jeszcze jeden rodzaj dźwięku, nazwany ikoną akustyczną (ang. *earcon*), który może być odtwarzany przez silnik TTS. Ikona akustyczna nie służy do reprezentowania tekstu, lecz do informowania za pomocą dźwięku o jakimś zdarzeniu lub do wskazywania elementu, który nie jest tekstem. Może ona stanowić dźwięk powiadamiający o odczytywaniu wypunktowania w prezentacji albo o przejściu na kolejną stronę. Jeśli z kolei tworzymy multimedialny przewodnik turystyczny, aplikacja może w ten sposób zaprosić użytkownika do przejścia do następnego punktu podróży.

Aby przypisać ikonę akustyczną do odtwarzania, musimy wywołać metodę `addEarcon()`, która — w podobny sposób jak metoda `addSpeech()` — pobiera dwa lub trzy argumenty. Pierwszym argumentem, podobnym do pola tekstowego w metodzie `addSpeech()`, jest nazwa ikony akustycznej. Zgodnie z konwencją powinniśmy zamieścić nazwę ikony akustycznej w nawiasie kwadratowym, np. `[boing]`. W przypadku metody pobierającej dwa argumenty tym drugim argumentem jest ciąg znaków reprezentujący nazwę pliku. W przypadku wstawienia trzech

argumentów drugim argumentem jest nazwa pakietu, a trzecim — identyfikator zasobu odnoszącego się do pliku audio, przechowywanego najprawdopodobniej w katalogu `/res/raw`. Do odtworzenia ikony akustycznej wykorzystujemy metodę `playEarcon()`, która ze swoimi trzema argumentami bardzo przypomina metodę `speak()`. Na listingu 24.5 zamieściliśmy przykład korzystania z ikony akustycznej.

---

**Listing 24.5.** Przykładowy kod wykorzystujący ikony akustyczne

```
String turnPageEarcon = "[turnPage]";
mTts.addEarcon(turnPageEarcon, "com.androidbook.tts.demo",
    R.raw.turnpage);
mTts.playEarcon(turnPageEarcon, TextToSpeech.QUEUE_ADD, params);
```

---

Powodem stosowania ikon akustycznych zamiast standardowego odtwarzania plików dźwiękowych w programie MediaPlayer jest mechanizm kolejkowania, zastosowany w silniku TTS. Nie trzeba obliczać stosownego momentu, w którym zostanie odtworzony sygnał akustyczny, ani polegać na właściwym dopasowaniu czasowym metod zwrotnych; wystarczy, że zakolejkujemy ikony akustyczne wśród tekstu wysyłanego do silnika TTS. Jesteśmy wtedy pewni, że nasze ikony akustyczne będą odtwarzane we właściwym momencie, więc możemy wykorzystać takie samo rozwiązanie do zaprezentowania dźwięków użytkownikowi. Istnieje także możliwość wprowadzenia wywołania metody `onUtteranceCompleted()` powiadamiającej nas o aktualnej pozycji odtwarzania.

## Odtwarzanie ciszy

Silnik TTS zawiera jeszcze jedną metodę, którą możemy wykorzystać — `playSilence()`. Podobnie jak w przypadku metod `speak()` i `playEarcon()`, także ta funkcja posiada trzy argumenty, z których drugi definiuje tryb kolejki, a w trzecim umieszczone są opcjonalne parametry `HashMap`. Pierwszy parametr metody `playSilence()` jest typu `long` i określa w milisekundach czas trwania ciszy. Najprawdopodobniej metoda ta będzie stosowana w trybie `QUEUE_ADD`, umożliwiającym oddzielenie w czasie dwóch różnych tekstowych ciągów znaków.

Innymi słowy, możemy pomiędzy dwoma tekstami odtwarzać ciszę bez konieczności zarządzania czasem oczekiwania w aplikacji. Po prostu wywołujemy metodę `speak()`, następnie `playSilence()` i znowu `speak()` w celu osiągnięcia zamierzonego efektu. Poniżej przedstawiamy przykładowy sposób wykorzystania metody `playSilence()` do wprowadzenia dwusekundowego opóźnienia:

```
mTts.playSilence(2000, TextToSpeech.QUEUE_ADD, params);
```

## Wybór innych mechanizmów przetwarzania tekstu na mowę

Aby zdefiniować dany silnik TTS, możemy wykorzystać metodę `setEnginePackageName()`, w której argumentem jest nazwa pakietu danego mechanizmu. W przypadku silnika Pico pakiet nosi nazwę `com.svox.pico`. Aby umożliwić użytkownikowi korzystanie z domyślnego silnika TTS, stosujemy metodę `getDefautlEngine()`. Te dwie metody nie mogą zostać wywołane przed metodą `onInit()`, gdyż w przeciwnym wypadku nie zadziałają. Nie są one również interpretowane przez wersje Androida starsze od 2.2.

## Stosowanie metod językowych

Nie poruszaliśmy dotychczas tematyki wymawiania słów w różnych językach, zatem zajmiemy się nią teraz. Technologia TTS odczytuje tekst za pomocą głosu przemawiającego w języku tego tekstu, to znaczy, że tekst napisany w języku włoskim będzie czytany przez Włocha. W celu zapewnienia poprawności wypowiedzi rozpoznawane są różne cechy tekstu. Z tego powodu stosowanie głosu przemawiającego w języku innym od używanego w tekście nie ma sensu. Odczytywanie tekstu napisanego w języku francuskim przez syntezator generujący wyrazy w języku włoskim spowoduje prawdopodobnie kłopoty; najlepiej dopasować ustawienia regionalne tekstu do ustawień regionalnych głosu.

Silnik TTS zawiera pewne metody obsługujące języki, służące zarówno do rozpoznawania używanego języka, jak i do ustawienia własnego. Domyslnie jest dostępnych tylko kilka pakietów językowych, jednak większa ich liczba będzie dostępna w sklepie Android Market. Na listingu 24.1 został zaprezentowany fragment kodu w metodzie zwrotnej `onActivityResult()`, za pomocą którego została utworzona klasa Intent wyszukująca brakujący pakiet językowy. Oczywiście istnieje możliwość, że pakiet danego języka nie został jeszcze utworzony, jednak ich liczba z każdym dniem rośnie.

Metodą pozwalającą na sprawdzenie dostępności języka jest `isLanguageAvailable(Locale locale)`. Ponieważ ustawienia lokalne mogą reprezentować zarówno kraj, jak i język, a czasem także różne ich odmiany, nie wystarczy przekazanie odpowiedzi `true` albo `false`. Istnieją następujące odpowiedzi: `TextToSpeech.LANG_COUNTRY_AVAILABLE`, oznaczająca, że obsługiwane są państwo i język; `TextToSpeech.LANG_AVAILABLE` — obsługiwany jest język, lecz nie państwo; oraz `TextToSpeech.LANG_NOT_SUPPORTED` — ani państwo, ani język nie są obsługiwane. Jeżeli otrzymamy wartość `TextToSpeech.LANG_MISSING_DATA`, to znaczy, że język jest obsługiwany, lecz silnik TTS nie znalazł plików danych. Aplikacja powinna skierować użytkownika do Android Market lub innego źródła, w którym będzie można znaleźć pliki pakietu językowego. Na przykład może być obsługiwany język francuski, lecz nie kanadyjski francuski. W takim przypadku po przekazaniu argumentu `Locale.CANADA_FRENCH` silnikowi TTS uzyskamy odpowiedź `TextToSpeech.LANG_AVAILABLE`, a nie `TextToSpeech.LANG_COUNTRY_AVAILABLE`. Kolejną wartość, jaka może zostać zwrócona (`TextToSpeech.LANG_COUNTRY_VAR_AVAILABLE`), stanowi specjalny przypadek, gdy ustawienia lokalne obejmują również wariant językowy; w takim wypadku obsługiwany jest kraj oraz język.

Stosowanie metody `isLanguageAvailable()` stanowi żmudny sposób określania wszystkich języków obsługiwanych przez silnik TTS. Na szczęście możemy się dowiedzieć, które języki są dostępne od ręki. Jeżeli przyjrzymy się uważnie listingowi 24.4, a dokładniej metodzie zwrotnej `onActivityResult()`, w miejscu, w którym odczytujemy odpowiedź intencji, zauważymy, że obiekt danych zawiera listę języków obsługiwanych przez silnik przetwarzania tekstu na mowę. Znajdziemy tam zmienną typu `ArrayList`, nazwaną `available`, pod przykładem `CHECK_VOICE_DATA_PASS`. Stanowi ona tablicę zawierającą wartości języków. Wartości te przyjmują na przykład postać `eng-USA` lub `fra-FRA`. Podczas gdy wartości ustawień regionalnych posiadają składnię `jj-kk`, gdzie `jj` stanowi dwuznakową reprezentację języka, a `kk` analogicznie definiuje kraj, w przypadku silnika TTS do utworzenia obiektu określającego ustawienia regionalne wykorzystywana jest trzyliterowa składnia `jjj-kkk`. Niestety, otrzymujemy z powrotem tablicę zawierającą ciągi znaków, a nie ustawienia regionalne, zatem musimy wprowadzić jakiś mechanizm analizowania składni lub mapowania w celu określenia, które języki są rzeczywiście dostępne dla danego silnika TTS.

Język ustawiamy za pomocą metody `setLanguage(Locale locale)`. Zostają zwrócone takie same kody jak w przypadku metody `isLanguageAvailable()`. Jeżeli chcemy skorzystać z tej metody, wywołajmy ją już po inicjalizacji silnika TTS, to znaczy wewnątrz metody `onInit()` lub później. W przeciwnym wypadku nie zadziała mechanizm wyboru języka. Bieżące, domyślne ustawienie lokalne urządzenia poznajemy za pomocą metody `Locale.getDefault()`, która przekazuje wartość tego ustawienia, na przykład `en_US`. Metoda `getLanguage()` klasy `TextToSpeech` pozwala poznać bieżące ustawienie lokalne silnika TTS. Podobnie jak w przypadku metody `setLanguage()`, nie należy wywoływać metody `getLanguage()` przed wystąpieniem funkcji `onInit()`. Wartości przekazywane przez tę metodę wyglądają na przykład jak `eng_USA`. Zwrócić uwagę, że człon definiujący język jest oddzielony od członu określającego państwo za pomocą znaku podkreślenia, a nie myślnika. Chociaż wydaje się, że Android jest wyrozumiały w kwestii wartości ustawień lokalnych, byłoby miło, gdyby w przyszłości interfejs ten zstał w jakiś sposób ujednolicony. Umieszczenie w naszym przykładzie następującego wiersza, ustanawiającego język stosowany w silniku TTS, jest dopuszczalne:

```
switch(mTts.setLanguage(Locale.getDefault())) {  
    case TextToSpeech.LANG_COUNTRY_AVAILABLE: ...
```

Na początku rozdziału wspomnialiśmy o głównym ustawieniu *Zawsze używaj moich ustawień*, przesłaniającym ustawienia językowe aplikacji. W wersji 2.2 Androida metoda `areDefaultsEnforced()` klasy `TextToSpeech` pozwala na określenie, czy użytkownik zaznaczył tę opcję. Metoda ta przyjmuje wartości `true` i `false`. Możemy zadecydować wewnątrz aplikacji, czy wybór danego języka może zostać przesłonięty, w wyniku czego aplikacja może podjąć odpowiednie działanie.

Na zakończenie omówienia technologii przetwarzania tekstu na mowę wspomnimy o kilku innych metodach, które możemy wykorzystać. Funkcja `setPitch(float pitch)` zmienia wysokość głosu bez wpływu na prędkość mówienia. Standardowa wartość wysokości głosu wynosi 1.0. Najmniejszą wartością rozróżnialną przez zmysł słuchu jest 0.5, najwyższą natomiast — 2.0; możemy ustawać wartości wyższe i niższe, nie słysząc jednak żadnej różnicy po przekroczeniu wartości progowych. Takie same wartości progowe zostały zdefiniowane dla metody `setSpeechRate(float rate)`. Oznacza to, że możemy przekazać tej metodzie zmiennoprzecinkowy argument o wartości mieszczącej się w zakresie od 0.5 do 2.0, gdzie 1.0 oznacza standardową prędkość mówienia. Wartości wyższe od 1.0 definiują szybszą mowę, a niższe od zwykłej wartości zwalniają szybkość wypowiedzi. Kolejną przydatną metodą jest `isSpeaking()`. Poprzez zwrot wartości `false` lub `true` wskazuje ona, czy silnik TTS w bieżącym momencie przeprowadza proces syntezy mowy (w tym również odtwarzanie ciśnów generowane przez metodę `playSilence()`). Jeżeli chcemy zostać powiadomieni o zakończeniu wypowiadania wszelkich kolejkowanych tekstów, możemy zaimplementować odbiorcę `BroadcastReceiver` wobec transmisji `ACTION_TTS_QUEUE_PROCESSING_COMPLETED`.

## Odbośniki

Poniżej prezentujemy przydatne adresy, pod którymi można znaleźć materiały rozwijające zagadnienia omawiane w tym rozdziale:

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu listę projektów, które możemy pobrać i zimportować do środowiska Eclipse. Projekty utworzone z myślą o niniejszym rozdziale zostały umieszczone w katalogu *ProAndroid3\_R24\_TekstNaMowę*.

W katalogu znajdziemy również plik *Czytaj.TXT*, zawierający dokładne instrukcje importowania projektów do środowiska Eclipse.

- <http://groups.google.com/group/tts-for-android> — adres ten kieruje nas do grupy dyskusyjnej, zajmującej się interfejsem przetwarzania tekstu na mowę.
- <https://groups.google.com/group/eyes-free> — łączy do strony grupy zajmującej się projektem Eyes-Free, mechanizmem o otwartym kodzie źródłowym, dostarczającym funkcje ułatwień dostępu do systemu Android. Znajdziemy tam również odnośniki do kodu źródłowego.

## Podsumowanie

W tym rozdziale pokazaliśmy, w jaki sposób aplikacja systemu Android może przemówić do użytkownika. Android został wyposażony w bardzo przyjemny silnik TTS, pozwalający na łatwe wykorzystanie tej funkcji. Dla programisty nie ma tu zbyt wiele do nauki. Silnik Pico obsługuje większość operacji. Pokazaliśmy, że jeżeli synteza natrafi na problematyczne słowo, istnieją sposoby uzyskania pożądanego efektu jego wymawiania. Zaawansowane funkcje silnika również znacznie ułatwiają życie. Podczas pracy z technologią przetwarzania tekstu na mowę musimy pamiętać o kilku podstawowych zasadach: o oszczędzaniu zasobów, rozsądnym współdzieleniu silnika TTS oraz właściwym wykorzystaniu syntezy mowy.



# Ekrany dotykowe

Wiele urządzeń obsługiwanych przez system Android posiada ekran dotykowy. W przypadku braku klawiatury fizycznej dane *muszą* być wprowadzane przez użytkownika za pomocą takiego dotyковego ekranu. Nieraz więc aplikacje muszą mieć możliwość przetwarzania danych wprowadzanych poprzez dotyk. Czytelnicy najprawdopodobniej mieli już do czynienia z wirtualną klawiaturą, która jest wyświetlana w momentach, gdy należy wprowadzić jakieś dane. W rozdziale 17., poświęconym programowaniu aplikacji wyświetlających mapy, dotyk posłużył nam do przesuwania mapy. Jeszcze nie omawialiśmy takich implementacji interfejsu ekranu dotykowego, teraz jednak pokażemy, w jaki sposób można wykorzystać jego możliwości.

Podzieliliśmy ten rozdział na cztery zasadnicze części. Część pierwsza została poświęcona obiektom klasy `MotionEvent`, które powiadamiają aplikację o tym, że użytkownik dotyka ekranu. Zajmiemy się w tej części także obiektami klasy `VelocityTracker` oraz funkcją przeciągania elementów na ekranie. W drugim podrozdziale skupimy się na wielodotykowości (ang. *multi-touch*), czyli możliwości wprowadzania danych za pomocą wielu palców jednocześnie. W części trzeciej omówimy funkcje dotyku w aplikacjach obsługujących mapy, ponieważ są w nich stosowane specjalne klasy i metody, ułatwiające powiązanie map z ekranem dotykowym. Ostatni podrozdział dotyczy gestów, czyli wyspecjalizowanego rozwiązania, w którym zaprogramowane sekwencje dotykowe są interpretowane jako polecenia.

## Klasa MotionEvent

W tej części rozdziału skupimy się na tym, w jaki sposób system przekazuje do aplikacji informacje o zdarzeniach dotykowych, czyli o tym, że użytkownik dotknął ekranu. Na razie skupimy się na dotykaniu ekranu za pomocą jednego palca (funkcja wielodotykowości zostanie omówiona w dalszej części rozdziału).

Ekran dotykowy jest wykonany ze specjalnych tworzyw, które mogą przetworzyć informację o naciśnięciu na współrzędne ekranu. Informacja o dotyku jest przetwarzana na dane, które są z kolei przekazywane oprogramowaniu.

## Obiekt MotionEvent

Dotknięcie ekranu przez użytkownika powoduje utworzenie obiektu `MotionEvent`. Obiekt ten zawiera informacje o czasie i miejscu dotknięcia oraz inne informacje na temat tego zdarzenia. Zostaje on przekazany odpowiedniej metodzie danej aplikacji. Może to być na przykład metoda `onTouchEvent()` klasy `View`. Nie zapominajmy, że klasa `View` jest nadziedzona w stosunku do dość licznej grupy innych klas, w tym takich jak `Layout`, `Button`, `List`, `Surface`, `Clock` i tak dalej. Oznacza to, że za pomocą zdarzeń dotykowych możemy oddziaływać na te wszystkie rodzaje obiektów klasy `View`. Wywołana metoda może przeanalizować obiekt `MotionEvent` w celu podjęcia decyzji o rodzaju przeprowadzanej czynności. Na przykład klasa `MapView` może wykorzystywać zdarzenia dotykowe do przesuwania mapy na boki, dzięki czemu użytkownik ręcznie wyszukuje interesujące go punkty. Obiekt wirtualnej klawiatury może odbierać zdarzenia dotykowe, które po wciśnięciu klawisza dotykowego są przetwarzane na znaki wprowadzane do innej części interfejsu użytkownika.

Obiekt `MotionEvent` bierze udział w sekwencji zdarzeń związanych ze zjawiskiem dotykania ekranu przez użytkownika. Sekwencja taka zostaje uruchomiona w chwili dotknięcia ekranu, jest kontynuowana w trakcie przesuwania palca po ekranie, a kończy się po jego oderwaniu od wyświetlacza. Zdarzenia przyłożenia palca do ekranu (działanie `ACTION_DOWN`), przesuwania palca (`ACTION_MOVE`) i oderwania palca (`ACTION_UP`) powodują tworzenie obiektów `MotionEvent`. Działania typu `ACTION_MOVE` mogą jednorazowo powodować tworzenie kilku obiektów `MotionEvent`, gdyż palec porusza się po ekranie, zanim pojawi się ostatnie działanie — `ACTION_UP`. Każdy obiekt `MotionEvent` zawiera informacje o rodzaju wykonywanego aktualnie działania, lokalizacji dotknięcia, sile nacisku, obszarze dotyku, czasie wykonania działania oraz o momencie zainicjalizowania działania `ACTION_DOWN`. Istnieje również czwarty typ działania — `ACTION_CANCEL`. Powiadamia ono aplikację o zakończeniu sekwencji dotykowej bez przetwarzania jej na czynność. Ostatnim rodzajem działania jest `ACTION_OUTSIDE`, które jest ustanawiane w specjalnym przypadku, gdy dotknięcie odbywa się poza oknem danej aplikacji, lecz musi być zarejestrowane.

Istnieje również inny sposób odbierania zdarzeń dotykowych, polegający na zarejestrowaniu w obiekcie klasy `View` procedury obsługi metod zwrótnych dla zdarzeń dotykowych. Klasa odbierająca zdarzenia musi zaimplementować interfejs `View.OnTouchListener` i powinna zostać wywołana metoda `setOnTouchListener()` klasy `View` w celu skonfigurowania procedury obsługi tego obiektu. Implementacja klasy interfejsu `View.OnTouchListener` musi zawierać metodę `onTouch()`. Podczas gdy parametrem metody `onTouchEvent()` jest jedynie obiekt `MotionEvent`, metoda `onTouch()` przyjmuje jako parametry obiekty klas `View` i `MotionEvent`. Wynika to z faktu, że klasa `OnTouchListener` może odbierać obiekty `MotionEvent` dla wielu widoków. Stanie się to bardziej zrozumiałe po przeanalizowaniu przykładowej aplikacji.

Jeżeli procedura obsługi obiektu `MotionEvent` (poprzez metodę `onTouchEvent()` lub `onTouch()`) obsługuje dane zdarzenie i informacja o nim nie musi być przekazywana dalej, metoda powinna powrócić z wartością `true`. W ten sposób system otrzymuje informację, że nie trzeba przesyłać zdarzenia do innych widoków. Jeżeli klasa `View` nie potrzebuje informacji o tym zdarzeniu ani o żadnym przyszłym zdarzeniu związanym z tą sekwencją dotyku, powraca z wartością `false`. Metoda `onTouchEvent()` bazowej klasy `View` nie wykonuje żadnej czynności i przekazuje wartość `false`. Jej klasy podrzędne mogą, ale nie muszą zachowywać się w ten sam sposób. Na przykład kontrolka `Button` obsługuje zdarzenie dotyku, ponieważ dotknięcie jest równoważne kliknięciu, zatem powraca z wartością `true` z metody `onTouchEvent()`. Po otrzymaniu działania `ACTION_DOWN` kontrolka `Button` zmieni swój kolor, aby w ten sposób poinformować,

że przetwarza zdarzenie kliknięcia. Obiekt ten czeka również na działanie ACTION\_UP, które oznacza zakończenie czynności przez użytkownika, dzięki czemu zostaje uruchomiony proces kliknięcia. Jeżeli kontrolka Button przekazała z metody onTouchEvent() wartość false, nie otrzyma już żadnego obiektu MotionEvent, informującego o zdjęciu przez użytkownika palca z ekranu.

Jeżeli chcemy, aby zdarzenia dotyku definiowały nową czynność na określonym obiekcie View, możemy rozszerzyć klasę, przesłonić metodę onTouchEvent() i wstawić własny algorytm. Możemy także zaimplementować interfejs View.OnTouchListener i skonfigurować procedurę obsługi metod zwrotnych wobec klasy View. Jeżeli skonfigurujemy procedurę obsługi wywołania metody onTouch(), zostaną do niej dostarczone wszelkie obiekty MotionEvent, zanim przejdą do metody onTouchEvent() klasy View. Metoda onTouchEvent() zostanie wywołana jedynie w przypadku przekazania wartości false przez metodę onTouch(). Przejdzmy do przykładowej aplikacji, która powinna ułatwić zrozumienie omówionych zjawisk.

**Uwaga!**

Na końcu rozdziału zamieściliśmy adres URL strony, z której można pobrać i zaimportować bezpośrednio do środowiska Eclipse projekty utworzone z myślą o niniejszym rozdziale.

Na listingu 25.1 przedstawiliśmy kod XML układu graficznego aplikacji. Utwórzmy nowy projekt w środowisku Eclipse i wstawmy w nim ten układ graficzny.

**Listing 25.1.** Plik XML układu graficznego dla aplikacji TouchDemo1

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <RelativeLayout
        android:id="@+id/layout1"
        android:tag="trueLayoutTop"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        >

        <com.androidbook.touch.demo1.TrueButton android:text="returns true"
            android:id="@+id/trueBtn1"
            android:tag="trueBtnTop"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />

        <com.androidbook.touch.demo1.FalseButton android:text="returns false"
            android:id="@+id/falseBtn1"
            android:tag="falseBtnTop"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@+id/trueBtn1" />

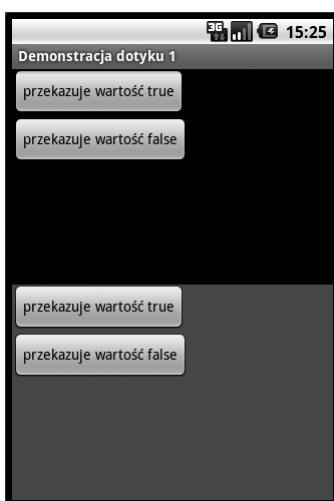
    </RelativeLayout>
<RelativeLayout
```

```
    android:id="@+id/layout2"
    android:tag="falseLayoutBottom"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:background="#FF00FF"
    >

    <com.androidbook.touch.demo1.TrueButton android:text="returns true"
        android:id="@+id/trueBtn2"
        android:tag="trueBtnBottom"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

    <com.androidbook.touch.demo1.FalseButton android:text="returns false"
        android:id="@+id/falseBtn2"
        android:tag="falseBtnBottom"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/trueBtn2" />
</RelativeLayout>
</LinearLayout>
```

Należy wyjaśnić kilka spraw na temat tego układu graficznego. Wprowadziliśmy znaczniki do obiektów interfejsu użytkownika. Będziemy odnosić się do tych znaczników w kodzie źródłowy w przypadku występowania zdarzeń dla tych obiektów. Wykorzystaliśmy także menedżery `RelativeLayout` do rozmieszczenia obiektów. Zwrócmy uwagę na zastosowanie niestandardowych kontrolek (`TrueButton` i `FalseButton`). Z kodu Java dowiadujemy się, że klasy te wywodzą się z klasy `Button`. Z tego powodu możemy stosować wobec nich wszystkie atrybuty standardowych przycisków. Na rysunku 25.1 ukazaliśmy wygląd tego układu graficznego, a na listingu 25.2 prezentujemy kod Java naszych przycisków.



Rysunek 25.1. Interfejs użytkownika aplikacji TouchDemo1

**Listing 25.2.** Kod Java klas Button dla aplikacji TouchDemo1

```
// Jest to plik BooleanButton.java
import android.content.Context;
import android.util.AttributeSet;
import android.util.Log;
import android.view.MotionEvent;
import android.widget.Button;

public abstract class BooleanButton extends Button {
    protected boolean myValue() {
        return false;
    }

    public BooleanButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        String myTag = this.getTag().toString();
        Log.v(myTag, "-----");
        Log.v(myTag, MainActivity.describeEvent(this, event));
        Log.v(myTag, "metoda super onTouchEvent() przekazuje " +
            super.onTouchEvent(event));
        Log.v(myTag, "a ja przekazuje " + myValue());
        return(myValue());
    }
}

// Jest to plik TrueButton.java
import android.content.Context;
import android.util.AttributeSet;

public class TrueButton extends BooleanButton {
    protected boolean myValue() {
        return true;
    }

    public TrueButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}

// Jest to plik FalseButton.java
import android.content.Context;
import android.util.AttributeSet;

public class FalseButton extends BooleanButton {

    public FalseButton(Context context, AttributeSet attrs) {
        super(context, attrs);
    }
}
```

Wprowadziliśmy klasę BooleanButton, abyśmy mogli wielokrotnie stosować metodę onTouch→Event(), do której dodaliśmy funkcję zapisywania informacji w dzienniku. Następnie utworzyliśmy klasy TrueButton i FalseButton, które będą inaczej reagowały na przekazywane im obiekty MotionEvent. Stanie się to bardziej oczywiste po przyjrzeniu się kodowi głównej aktywności, ukazanemu na listingu 25.3.

**Listing 25.3.** Kod Java naszej głównej aktywności

---

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.widget.Button;
import android.widget.RelativeLayout;

public class MainActivity extends Activity implements OnTouchListener {
    /** Wywoływanie podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        RelativeLayout layout1 = (RelativeLayout) findViewById(R.id.layout1);
        layout1.setOnTouchListener(this);
        Button trueBtn1 = (Button) findViewById(R.id.trueBtn1);
        trueBtn1.setOnTouchListener(this);
        Button falseBtn1 = (Button) findViewById(R.id.falseBtn1);
        falseBtn1.setOnTouchListener(this);

        RelativeLayout layout2 = (RelativeLayout) findViewById(R.id.layout2);
        layout2.setOnTouchListener(this);
        Button trueBtn2 = (Button) findViewById(R.id.trueBtn2);
        trueBtn2.setOnTouchListener(this);
        Button falseBtn2 = (Button) findViewById(R.id.falseBtn2);
        falseBtn2.setOnTouchListener(this);
    }

    @Override
    public boolean onTouch(View v, MotionEvent event) {
        String myTag = v.getTag().toString();
        Log.v(myTag, "-----");
        Log.v(myTag, "Umieszczono widok " + myTag + " w metodzie onTouch");
        Log.v(myTag, describeEvent(v, event));
        if( "true".equals(myTag.substring(0, 4))) {
            /* Log.v(myTag, "*** wywoływanie mojej metody onTouchEvent() ***");
            v.onTouchEvent(event);
            Log.v(myTag, "*** powrot z metody onTouchEvent() ***"); */
            Log.v(myTag, "i przekazuję wartość true");
            return true;
        }
        else {
            Log.v(myTag, "i przekazuję wartość false");
        }
    }
}
```

```

        return false;
    }
}

protected static String describeEvent(View view, MotionEvent event) {
    StringBuilder result = new StringBuilder(300);
    result.append("Działanie: ").append(event.getAction()).append("\n");
    result.append("Lokacja: ").append(event.getX()).append(" x ")
.append(event.getY()).append("\n");
    if( event.getX() < 0 || event.getX() > view.getWidth() ||
        event.getY() < 0 || event.getY() > view.getHeight() ) {
        result.append(">>> Zdarzenie dotyku opuściło widok <<<\n");
    }
    result.append("Flagi krawędzi: ").append(event.getEdgeFlags()).append("\n");
    result.append("Siła nacisku: ").append(event.getPressure()).append(" ");
    result.append("Rozmiar: ").append(event.getSize()).append("\n");
    result.append("Czas dotknięcia: ").append(event.getDownTime()).append("ms\n");
    result.append("Czas zdarzenia: ").append(event.getEventTime()).append("ms");
    result.append(" Szacowany: ").append(event.getEventTime()-event.getDownTime());
    result.append(" ms\n");
    return result.toString();
}
}

```

Kod naszej aktywności konfiguruje metody zwrotne dla przycisków i układów graficznych w taki sposób, aby zdarzenia dotyku (na przykład obiekty `MotionEvent`) były przetwarzane dla każdego elementu interfejsu użytkownika. Utworzyliśmy rozbudowany system zapisywania informacji o zdarzeniach w dzienniku, zatem będziemy doskonale wiedzieć, co się wydarzyło podczas dotknięcia ekranu. Po skompilowaniu i uruchomieniu aplikacji powinniśmy ujrzeć taki sam ekran jak na rysunku 25.1.

Aby w pełni wykorzystać aplikację, musimy otworzyć w środowisku Eclipse narzędzie *LogCat*, aby na bieżąco obserwować komunikaty wyświetlane podczas dotykania ekranu. Aplikacja działa również dobrze na emulatorze, jak w rzeczywistym urządzeniu. Zalecamy także zmaksymalizowanie okna narzędzia *LogCat*, aby móc łatwiej przewijać ekran i przeglądać wszystkie komunikaty wygenerowane przez aplikację. Okno to zostanie zmaksymalizowane po dwukrotnym kliknięciu zakładki *LogCat*. Przejdzmy teraz do interfejsu UI aplikacji i dotknijmy przycisku *przekazuje wartość true*, znajdującego się u góry ekranu (w przypadku korzystania z emulatora kliknijmy krótko ten przycisk). W oknie *LogCat* powinny pojawić się przynajmniej dwa komunikaty o zdarzeniach. Zostaną one oznaczone jako przychodzące z obiektu `trueBtnTop` i zapisane w dzienniku za pomocą metody `onTouch()` klasy `MainActivity`. Możemy spojrzeć na kod metody `onTouch()` w pliku `MainActivity.java`. Po przeanalizowaniu wyników w oknie *LogCat*, można stwierdzić, wywołania których metod generują wartości. Na przykład wartość wyświetlana po etykiecie *Działanie*: pochodzi z metody `getAction()`. Na listingu 25.4 został przedstawiony przykładowy wpis z dziennika podczas korzystania z emulatora, a listing 25.5 ukazuje to samo, ale podczas użytkowania rzeczywistego urządzenia.

**Listing 25.4.** Przykładowe komunikaty narzędzia LogCat, pochodzące z aplikacji `TouchDemo1` zainstalowanej na emulatorze

```

trueBtnTop      -----
trueBtnTop      Umieszczone widok trueBtnTop w metodzie onTouch
trueBtnTop      Działanie: 0

```

```

trueBtnTop      Lokacja: 52.0 x 20.0
trueBtnTop      Flagi krawędzi: 0
trueBtnTop      Siła nacisku: 0.0 Rozmiar: 0.0
trueBtnTop      Czas dotknięcia: 163669ms
trueBtnTop      Czas zdarzenia: 163669ms Szacowany: 0 ms
trueBtnTop      i przekazuję wartość true
-----
trueBtnTop      Umieszczono widok trueBtnTop w metodzie onTouch
trueBtnTop      Działanie: 1
trueBtnTop      Lokacja: 52.0 x 20.0
trueBtnTop      Flagi krawędzi: 0
trueBtnTop      Siła nacisku: 0.0 Rozmiar: 0.0
trueBtnTop      Czas dotknięcia: 163669ms
trueBtnTop      Czas zdarzenia: 163831ms Szacowany: 162 ms
trueBtnTop      i przekazuję wartość true

```

**Listing 25.5.** Przykładowe komunikaty narzędzia LogCat, pochodzące z aplikacji TouchDemo1 zainstalowanej na rzeczywistym urządzeniu

```

trueBtnTop      -----
trueBtnTop      Umieszczono widok trueBtnTop w metodzie onTouch
trueBtnTop      Działanie: 0
trueBtnTop      Lokacja: 42.8374 x 25.293747
trueBtnTop      Flagi krawędzi: 0
trueBtnTop      Siła nacisku: 0.05490196 Rozmiar: 0.2
trueBtnTop      Czas dotknięcia: 24959412ms
trueBtnTop      Czas zdarzenia: 24959412ms Szacowany: 0 ms
trueBtnTop      i przekazuję wartość true
-----
trueBtnTop      Umieszczono widok trueBtnTop w metodzie onTouch
trueBtnTop      Działanie: 2
trueBtnTop      Lokacja: 42.8374 x 25.293747
trueBtnTop      Flagi krawędzi: 0
trueBtnTop      Siła nacisku: 0.05490196 Size: 0.2
trueBtnTop      Czas dotknięcia: 24959412ms
trueBtnTop      Czas zdarzenia: 24959530ms Szacowany: 118 ms
trueBtnTop      i przekazuję wartość true
-----
trueBtnTop      Umieszczono widok trueBtnTop w metodzie onTouch
trueBtnTop      Działanie: 1
trueBtnTop      Lokacja: 42.8374 x 25.293747
trueBtnTop      Flagi krawędzi: 0
trueBtnTop      Siła nacisku: 0.05490196 Rozmiar: 0.2
trueBtnTop      Czas dotknięcia: 24959412ms
trueBtnTop      Czas zdarzenia: 24959567ms Szacowany: 155 ms
trueBtnTop      i przekazuję wartość true

```

Pierwsze zdarzenie posiada działanie oznaczone jako 0, którego odpowiednikiem jest ACTION\_DOWN. Działanie ostatniego zdarzenia posiada etykietę 1, oznaczającą ACTION\_UP. W przypadku używania rzeczywistego urządzenia może się pojawić więcej zdarzeń. Wszystkie zdarzenia pośrednie między działaniami ACTION\_DOWN a ACTION\_UP będą prawdopodobnie określane przez działanie o wartości 2, reprezentujące ACTION\_MOVE. Pozostałymi możliwościami są wartości 3 dla działania ACTION\_CANCEL i 4 dla działania ACTION\_OUTSIDE. Podczas

używania palców na prawdziwym wyświetlaczu czasami nie ma możliwości dotknięcia i ode- rwania palca bez nieznacznego przesunięcia nim po ekranie, zatem pewne niespodziewane zdarzenia ACTION\_MOVE nie powinny budzić zdziwienia.

Istnieją również inne różnice pomiędzy emulatorem a rzeczywistym urządzeniem. Zauważmy, że dokładność wskazywania położenia na emulatorze jest rzędu liczb naturalnych (52 na 20), podczas gdy rzeczywiste urządzenie generuje wartości w formie ułamków (42.8374 na 25.293747). Położenie zdarzenia MotionEvent posiada składowe X i Y, gdzie wartość X oznacza odległość od lewej krawędzi widoku View do dotkniętego punktu, natomiast wartość Y reprezentuje dystans od górnej części widoku View do punktu dotknięcia.

Zauważmy również, że siła nacisku, jak również jego rozmiar w emulatorze posiadają wartość 0. W przypadku rzeczywistego urządzenia siła nacisku wskazuje na to, jak mocno został przyciśnięty palec do ekranu dotyковego, a rozmiar oznacza obszar dotkniętego wyświetlacza. Jeżeli dotknemy leciutko ekran czubkiem małego palca, wartości siły nacisku oraz rozmiaru będą nieznaczne. Jeśli natomiast mocno przyciśniemy ekran kciukiem, wartości obydwu parametrów osiągną większe wartości. Zgodnie z dokumentacją wartości siły nacisku i rozmiaru mieszczą się w przedziale od 0 do 1. Jednak ze względu na różnice sprzętowe określenie bezwzględnych wartości tych dwóch parametrów może być sporym problemem. Dobrze byłoby porównać wartości tych dwóch parametrów pomiędzy poszczególnymi zdarzeniami MotionEvent, jednak uznanie wartości większej od 0.8 za mocne wcisnięcie może sprawić nam nie lada kłopot. Na danym urządzeniu można nigdy nie przekroczyć wartości 0.8. Może nawet nie uda się przekroczyć wartości 0.2.

Wartości czasu dotyku i czasu zdarzenia są wyliczane tak samo na emulatorze i w rzeczywistym urządzeniu, jedyna różnica polega na tym, że w rzeczywistym urządzeniu pojawiają się o wiele większe wartości. Również obliczenia szacowanego czasu przebiegają w identyczny sposób.

Flagi krawędzi służą do wykrycia momentu, w którym dotyk przekroczył fizyczną granicę wyświetlacza. W dokumentacji zestawu Android SDK występuje stwierdzenie, że flagi te służą do wskazywania momentu zetknięcia palca z krawędzią ekranu (górną, dolną, lewą lub prawą). Jednak metoda `getEdgeFlags()` może zawsze przekazywać wartość 0, w zależności od rodzaju używanego urządzenia lub emulatora. W przypadku niektórych układów elektronicznych trudno wykryć krawędź ekranu, zatem spodziewamy się, że Android przypnie lokację do brzegu i skonfiguruje odpowiednią flagę brzegową. Nie zawsze tak się dzieje, zatem nie powinniśmy zbytnio polegać na konfiguracji flag brzegowych. Klasa MotionEvent zawiera metodę `setEdgeFlags()`, dzięki której możemy samodzielnie wyznaczyć takie flagi.

Należy jeszcze wspomnieć, że nasza metoda `onTouch()` powraca z wartością `true`, ponieważ kontrolka TrueButton została zakodowana w taki sposób, aby przekazywana była właśnie ta wartość. Otrzymanie tej wartości oznacza, że klasa MotionEvent została przetworzona i nie ma potrzeby przekazywać jej dalej. W ten sposób system zostaje również poinformowany, żeby w dalszym ciągu przesyłał zdarzenia tej sekwencji dotknięcia do tej metody. To dlatego otrzymujemy komunikat o zdarzeniach ACTION\_UP oraz ACTION\_MOVE w przypadku rzeczywistego urządzenia.

Dotknijmy teraz przycisku *przekazuje wartość false*, znajdującego się w górnej części ekranu. Zademonstrujemy dla pozostałej części podrozdziału jedynie przykładowy wynik z okna *LogCat*, wygenerowany po użyciu rzeczywistego urządzenia. Wyjaśniliśmy wszystkie różnice, zatem osoby pracujące na emulatorze powinny umieć poprawnie zinterpretować wyniki pojawiające się na ekranie. Listing 25.6 prezentuje wynikowe dane narzędzia *LogCat* dla dotknięcia przycisku *przekazuje wartość false*.

**Listing 25.6.** Przykładowe wpisy narzędzia LogCat powstałe po dotknięciu górnego przycisku „przekazuję wartość false”

---

```
falseBtnTop      -----
falseBtnTop      Umieszczono widok falseBtnTop w metodzie onTouch
falseBtnTop      Działanie: 0
falseBtnTop      Lokacja: 61.309372 x 44.281494
falseBtnTop      Flagi krawędzi: 0
falseBtnTop      Siła nacisku: 0.0627451 Rozmiar: 0.26666668
falseBtnTop      Czas dotknięcia: 28612178ms
falseBtnTop      Czas zdarzenia: 28612178ms Szacowany: 0 ms
falseBtnTop      i przekazuję wartość false
-----
falseBtnTop      Działanie: 0
falseBtnTop      Lokacja: 61.309372 x 44.281494
falseBtnTop      Flagi krawędzi: 0
falseBtnTop      Siła nacisku: 0.0627451 Rozmiar: 0.26666668
falseBtnTop      Czas dotknięcia: 28612178ms
falseBtnTop      Czas zdarzenia: 28612178ms Szacowany: 0 ms
super onTouchEvent() przekazuje wartość true
falseBtnTop      i przekazuję wartość false
-----
trueLayoutTop    Umieszczono widok trueLayoutTop w metodzie onTouch
trueLayoutTop    Działanie: 0
trueLayoutTop    Lokacja: 61.309372 x 116.281494
trueLayoutTop    Flagi krawędzi: 0
trueLayoutTop    Siła nacisku: 0.0627451 Rozmiar: 0.26666668
trueLayoutTop    Czas dotyku: 28612178ms
trueLayoutTop    Czas zdarzenia: 28612178ms Szacowany: 0 ms
trueLayoutTop    i przekazuję wartość true
-----
trueLayoutTop    Umieszczono widok trueLayoutTop w metodzie onTouch
trueLayoutTop    Działanie: 2
trueLayoutTop    Lokacja: 61.309372 x 111.90039
trueLayoutTop    Flagi krawędzi: 0
trueLayoutTop    Siła nacisku: 0.0627451 Rozmiar: 0.26666668
trueLayoutTop    Czas dotyku: 28612178ms
trueLayoutTop    Czas zdarzenia: 28612217ms Szacowany: 39 ms
trueLayoutTop    i przekazuję wartość true
-----
trueLayoutTop    Umieszczono widok trueLayoutTop w metodzie onTouch
trueLayoutTop    Działanie: 1
trueLayoutTop    Lokacja: 55.08958 x 115.30792
trueLayoutTop    Flagi krawędzi: 0
trueLayoutTop    Siła nacisku: 0.0627451 Rozmiar: 0.26666668
trueLayoutTop    Czas dotknięcia: 28612178ms
trueLayoutTop    Czas zdarzenia: 28612361ms Szacowany: 183 ms
trueLayoutTop    i przekazuję wartość true
```

---

Widzimy tutaj zupełnie odmienne zachowanie, wyjaśnijmy zatem, co tu zaszło. Android odbiera zdarzenie ACTION\_DOWN w obiekcie MotionEvent i przekazuje je naszej metodzie onTouch() klasy MainActivity. Metoda ta rejestruje informację w oknie LogCat i powraca z wartością false. Android dowiaduje się w ten sposób, że zdarzenie nie zostało przetworzone, zatem wyszukuje kolejną metodę do wywołania, którą w naszym wypadku okazuje się przesłonięta metoda onTouchEvent() klasy FalseButton. Ponieważ klasa ta jest rozszerzeniem klasy BooleanButton,

kod metody `onTouchEvent()` znajdziemy w pliku `BooleanButton.java`. W metodzie `onTouch→Event()` ponownie zapisujemy informację w oknie `LogCat`, wywołujemy nadrzedną klasę tej metody, a następnie znów otrzymujemy wartość `false`. Zwróćmy uwagę, że zapisana w dzienniku informacja o położeniu jest taka sama jak poprzednio. Należało się tego spodziewać, ponieważ w tym momencie jest nadal przetwarzany ten sam obiekt klasy `View`, w widoku `FalseButton`. Widzimy, że nadrzedna klasa przekaże wartość `true` z metody `onTouchEvent()`, i znamy tego powód. Jeżeli spojrzmy na ten przycisk w interfejsie UI, powinniśmy dostrzec zmianę jego koloru w stosunku do przycisku *przekazuje wartość true*. Przycisk *przekazuje wartość false* wygląda, jakby został częściowo wcisnięty, tj. zachowuje się jak przycisk, który naciśnięto, lecz go nie puszczone. Nasza niestandardowa metoda powróciła z wartością `false` zamiast z wartością `true`. Ponieważ kolejny raz poinformowaliśmy w ten sposób system, że zdarzenie nie zostało obsłużone, Android nie wyše zdarzenia `ACTION_UP` do przycisku, co jest równoznaczne z pominięciem informacji o zdjęciu palca z wyświetlacza. Zatem przycisk ten znajduje się cały czas w stanie naciśnięcia. Gdybyśmy zgodnie z działaniem klasy nadrzednej przekazali wartość `true`, otrzymalibyśmy w końcu zdarzenie `ACTION_UP`, a przycisk odzyskałby domyślny kolor. Reasumując: za każdym razem, gdy otrzymujemy wartość `false` z elementu interfejsu UI dla otrzymanego obiektu `MotionEvent`, Android przestaje wysyłać obiekty `MotionEvent` do tego elementu interfejsu UI i rozpoczyna wyszukiwanie kolejnego składnika interfejsu użytkownika, który obsługuje element `MotionEvent`.

Być może Czytelnik zwrócił uwagę, że przycisk *przekazuje wartość true* nie zmienił koloru po jego dotknięciu. Dlaczego tak się stało? Ponieważ metoda `onTouch()` została wywołana przed metodami obsługi przycisku, a że metoda ta przekazała wartość `true`, system nie zadecydował o wywołaniu metody `onTouchEvent()` kontrolki *przekazuje wartość true*. Jeżeli tuż przed otrzymaniem wartości `true` z metody `onTouch()` dodamy wiersz `v.onTouchEvent(event);`, kolor przycisku ulegnie zmianie. Pojawi się także więcej informacji w dzienniku `LogCat`, ponieważ metoda `onTouchEvent()` zacznie wyświetlać w nim komunikaty.

Wróćmy do wyników w oknie `LogCat`. Android dwukrotnie bez skutku próbował znaleźć odbiorcę dla zdarzenia `ACTION_DOWN`, więc teraz przechodzi do następnego widoku `View` (w naszym przypadku będzie to układ graficzny stanowiący pojemnik przycisku), który potencjalnie mógłby odebrać to zdarzenie. Ten układ graficzny nosi nazwę `trueLayoutTop` i zobaczyliśmy, że odebrał to zdarzenie.

Zwróćmy uwagę, że nasza metoda `onTouch()` została wywołana ponownie, teraz jednak z widkiem układu graficznego, a nie z widokiem przycisku. Wszystkie informacje dotyczące obiektu `MotionEvent` zostały przekazane metodzie `onTouch()` klasy `trueLayoutTop` w niezmienionej postaci, nawet informacje o czasie, oprócz współrzędnej Y lokacji. Zmieniła ona wartość ze współrzędnej 44.281494 przycisku na 116.281494 układu graficznego. Jest to logiczne rozwiązanie, ponieważ przycisk nie znajduje się w lewym górnym rogu układu graficznego, jest natomiast umieszczony poniżej przycisku *przekazuje wartość true*. Zatem wartość współrzędnej Y dotyku dla układu graficznego jest większa od wartości takiego samego dotyku w odniesieniu do przycisku; dotykamy ekranu w większej odległości od górnej krawędzi układu graficznego niż od górnej krawędzi przycisku. Ponieważ metoda `onTouch()` klasy `trueLayoutTop` przekazuje wartość `true`, Android przesyła pozostałe zdarzenia do układu graficznego i widzimy wpisy w dzienniku odpowiadające zdarzeniom `ACTION_MOVE` i `ACTION_UP`. Dotknijmy teraz przycisku *przekazuje wartość false*, a zostanie wyświetlony ten sam zbiór komunikatów dziennika, to znaczy zostanie wywołana metoda `onTouch()` dla klasy `falseBtnTop`, metoda `onTouchEvent()` dla klasy `falseBtnTop`, a następnie, w przypadku pozostałych zdarzeń, metoda `onTouch()` dla klasy `trueLayoutTop`. Android zaprzestaje jedynie wysyłania do przycisku zdarzeń odpowiedzialnych

za sekwencje pojedynczego dotknięcia. W przypadku nowej sekwencji dотyку Android będzie ją wysyłał do przycisku, dopóki wywołana metoda nie przekaże znowu wartości `false`, co w naszym przykładzie jest jej zaprogramowanym zadaniem.

Dotknijmy teraz obszaru w górnym układzie graficznym, lecz nie dotykajmy przycisku, a następnie przesuńmy palec po ekranie i oderwijmy go od wyświetlacza (na emulatorze wystarczy wykonać podobny ruch myszą). Zwrócić uwagę na strumień komunikatów dziennika, gdzie pierwszym zarejestrowanym działaniem jest `ACTION_DOWN`, następnie pojawia się wiele zdarzeń `ACTION_MOVE`, a wpisy zamyka zdarzenie `ACTION_UP`.

Dotknijmy teraz przycisku *przekazuje wartość true*, teraz jednak poprzesuwajmy palcem po wyświetlaczu i oderwijmy go od ekranu. Listing 25.7 przedstawia nowe informacje, które pojawiły się w oknie *LogCat*.

#### **Listing 25.7. Rejestry dziennika LogCat dotyczące zdarzenia dотyku wykraczaj±cego poza widok**

[ ... wpisy dziennika dotyczące zdarzenia `ACTION_DOWN` oraz zdarzeń `ACTION_MOVE` ... ]

```
trueBtnTop      Umieszczono widok trueBtnTop w metodzie onTouch  
trueBtnTop      Działanie: 2  
trueBtnTop      Lokacja: 150.41768 x 22.628128  
trueBtnTop      >>> Zdarzenie dотyku opuści³o widok <<<  
trueBtnTop      Flagi krawędzi: 0  
trueBtnTop      Siła nacisku: 0.047058824 Rozmiar: 0.13333334  
trueBtnTop      Czas dотyku: 31690859ms  
trueBtnTop      Czas zdarzenia: 31691344ms Szacowany: 485 ms  
trueBtnTop      i przekazuję wartość true
```

[ ... więcej komunikatów dotyczących zdarzeń `ACTION_MOVE` ... ]

```
trueBtnTop      Umieszczono widok trueBtnTop w metodzie onTouch  
trueBtnTop      Działanie: 1  
trueBtnTop      Lokacja: 291.5864 x 223.43854  
trueBtnTop      >>> Zdarzenie dотyku opuści³o widok <<<  
trueBtnTop      Flagi krawędzi: 0  
trueBtnTop      Siła nacisku: 0.047058824 Rozmiar: 0.13333334  
trueBtnTop      Czas dотyku: 31690859ms  
trueBtnTop      Czas zdarzenia: 31692493ms Szacowany: 1634 ms  
trueBtnTop      i przekazuję wartość true
```

---

Nawet jeżeli zdejmiemy palec z przycisku, będziemy otrzymywać związane z nim powiadomienia o zdarzeniach dотyku. Pierwszy rejestr na listingu 25.7 prezentuje wpis zdarzenia zchodzącego poza obrębem przycisku. W naszym przypadku współrzędna X zdarzenia znajduje się z prawej strony od krawędzi przycisku. Ciągle jednak występują wywołania obiektu `MotionEvent`, aż do momentu wystąpienia zdarzenia `ACTION_UP`, ponieważ ciągle otrzymujemy wartość `true` z metody `onTouch()`. Nawet jeśli w końcu oderwiemy palec od ekranu dотykowego i nie będzie się on znajdował na przycisku, metoda `onTouch()` będzie ciągle wywoływana w celu przekazywania zdarzenia `ACTION_UP`, ponieważ nieprzerwanie jest przekazywana wartość `true`. Należy o tym pamiętać podczas korzystania z obiektów `MotionEvent`. Po przesunięciu palca poza widok możemy zadecydować o anulowaniu wykonywanej czynności i otrzymać wartość `false` z metody `onTouch()`, dzięki czemu nie będzie powiadamiana o następnych zdarzeniach. Ewentualnie możemy pozwolić na dalsze otrzymywanie zdarzeń (poprzez przekazanie wartości `true` z metody `onTouch()`) i wykonywanie operacji jedynie po powrocie palca do widoku.

Sekwencja zdarzeń dotyku została powiązana z górnym przyciskiem *przekazuje wartość true* po otrzymaniu wartości `true` z metody `onTouch()`. W ten sposób Android został poinformowany, że może przestać szukać elementu odbierającego obiekty `MotionEvent` i wysyłać nam wszystkie następne obiekty `MotionEvent` związane z tą sekwencją dotyku. Nawet jeśli natrafimy palcem na inny widok, sekwencja dotyku ciągle będzie dotyczyła pierwotnego widoku.

Zobaczmy, co się dzieje z dolną połową aplikacji. Dotknijmy umieszczonego tam przycisku *przekazuje wartość true*. Widzimy, że następuje takie same zjawisko jak w przypadku dotknięcia górnego przycisku *przekazuje wartość true*. Ponieważ metoda `onTouch()` powraca z wartością `true`, Android będzie przesyłał resztę zdarzeń w sekwencji dotyku, dopóki palec styka się z wyświetlaczem. Teraz dotknijmy dolnego przycisku *przekazuje wartość false*. I znowu — metody `onTouch()` oraz `onTouchEvent()` powracają z wartościami `false` (obydwie są związane z kontrolką `falseBtnBottom`). Jednak tym razem następnym widokiem, który otrzyma obiekt `MotionEvent`, jest układ graficzny `falseLayoutBottom`, również przekazujący wartość `false`. I to tyle.

Ponieważ metoda `onTouchEvent()` wywołała metodę `onTouchEvent()` klasy bazowej, przycisk zmienił kolor, wskazując tym samym, że został częściowo wciśnięty. Pozostanie on jednak w tym stanie, ponieważ nigdy nie otrzymamy zdarzenia `ACTION_UP` w tej sekwencji, gdyż nasze metody cały czas zwracają wartość `false`. W przeciwieństwie do poprzedniego przykładu, nawet układ graficzny nie reaguje na takie zdarzenie. Gdybyśmy dotknęli przycisku *przekazuje wartość false* i przytrzymali go, a następnie poprzesuwali palec po ekranie, nie zobaczylibyśmy żadnego wpisu w oknie *LogCat*, ponieważ obiekty `MotionEvent` nie są wysyłane. Zawsze przekazywana jest wartość `false`, system nie będzie więc informował o zdarzeniach z tej sekwencji dotyku. Jeżeli rozpoczniemy nową sekwencję dotyku, ujrzymy rejestyry pojawiające się w narzędziu *LogCat*. Jeżeli zainicjalizujemy taką sekwencję w dolnym układzie graficznym, lecz nie na obszarze przycisku, zostanie zarejestrowane pojedyncze zdarzenie w obiekcie `falseLayoutBottom`, przekazujące wartość `false`, i nic poza tym (dopóki nie rozpoczniemy nowej sekwencji dotyku).

Dotychczas wykorzystywaliśmy przyciski do ukazania efektów zdarzeń `MotionEvent` związanych z ekranem dotykowym. Warto zauważyć, że w normalnej sytuacji zaimplementowalibyśmy operacje na przyciskach za pomocą metody `onClick()`. Wybrałyśmy do tego przykładu przyciski, ponieważ nie są trudne do utworzenia oraz są klasami podistruktami w stosunku do klasy `View`, mogą więc odbierać zdarzenia dotyku tam samo jak inne widoki. Pamiętajmy, że omówione techniki odnoszą się do dowolnego widoku `View` w aplikacji, bez względu na to, czy mamy do czynienia ze standardową, czy niestandardową klasą widoku.

## Wielokrotne wykorzystywanie obiektów `MotionEvent`

Podczas przeglądania dokumentacji klasy `MotionEvent` możemy natrafić na metodę `recycle()`. Wielokrotne wykorzystywanie obiektów `MotionEvent` otrzymywanych w metodach `onTouch()` lub `onTouchEvent()` może być kuszącym pomysłem, nie powinniśmy jednak tego robić. Jeżeli metoda zwrotna nie obsługuje obiektu `MotionEvent` iwróci z wartością `false`, prawdopodobnie obiekt ten zostanie przekazany innej metodzie, widokowi lub aktywności. Z tego wynika, że nie należy go jeszcze przeznaczać do ponownego wykorzystania. Nawet jeśli to zdarzenie zostało obsłużone i otrzymałyśmy wartość `true`, nie powinniśmy go przetwarzać w ten sposób — chodzi o kwestie przynależności tego obiektu.

Jeżeli przyjrzymy się obiektowi `MotionEvent`, zauważymy kilka odmian metody `obtain()`. Służy ona albo do tworzenia kopii obiektu `MotionEvent`, albo do tworzenia jego zupełnie nowego wystąpienia. To właśnie taki nowy obiekt lub kopia obiektu mogą zostać ponownie wykorzystane. Jeżeli na przykład nie chcemy pozbywać się obiektu, który został przekazany za pomocą metody

zwrotnej, powinniśmy przy użyciu metody `obtain()` utworzyć jego kopię, ponieważ po wyjściu z tej metody obiekt ten zostanie ponownie wykorzystany, co może dać nieoczekiwane rezultaty. Gdy skończymy korzystać z kopii obiektu, wywołujemy wobec niej metodę `recycle()`.

## Stosowanie klasy VelocityTracker

Android zawiera klasę `VelocityTracker`, która obsługuje sekwencję dotyku. Dobrze byłoby znać prędkość palca poruszającego się po powierzchni ekranu. Jeśli na przykład użytkownik szybko przesuwa palec po ekranie, może to oznaczać wykonanie gestu przerzucania (ang. *fling*), który aplikacja przetwarza na odpowiednią operację. Klasa `VelocityTracker` dokonuje odpowiednich obliczeń.

Aby skorzystać z tej klasy, musimy najpierw utworzyć jej instancję poprzez wywołanie statycznej metody `VelocityTracker.obtain()`. Następnie możemy dodać do niej obiekty `MotionEvent` wraz z metodą `addMovement(MotionEvent ev)`. Metoda ta byłaby wywoływana przez procedurę obsługi obiektów `MotionEvent`, otrzymywanych od takich metod, jak `onTouch()` lub `onTouchEvent()`. Klasa `VelocityTracker` wykorzystuje obiekty `MotionEvent` do określenia zachowania sekwencji dotyku. Po otrzymaniu przez klasę `VelocityTracker` przynajmniej dwóch obiektów `MotionEvent` możemy zastosować inne metody do określenia sytuacji.

Dwie metody klasy `VelocityTracker` — `getXVelocity()` i `getYVelocity()` — przekazują wyliczoną prędkość palca, odpowiednio, w kierunkach X i Y. Wartość przekazana przez te dwie metody będzie symbolizowała liczbę pikseli w jednostce czasu. Mogą to być piksele na milisekundę, piksele na sekundę lub dowolna inna jednostka. Aby zdefiniować jednostkę czasu w klasie `VelocityTracker`, zanim zostaną wywołane dwie metody typu getter, musimy wywołać jej metodę `computeCurrentVelocity(int units)`. Wartość atrybutu `units` reprezentuje liczbę milisekund w czasie mierzenia prędkości. Jeżeli chcemy, aby była mierzona liczba pikseli na milisekundę, wstawiamy wartość jednostki równą 1; w przypadku chęci mierzenia liczby pikseli w sekundzie wpisujemy wartość 1000. Wartość zwrócona przez metody `getXVelocity()` i `getYVelocity()` będzie dodatnia, jeśli wektor prędkości zostanie skierowany w prawo (dla kierunku X) lub w dół (dla kierunku Y). Wartości będą ujemne dla prędkości skierowanej w lewo lub do góry.

Po otrzymaniu obiektu klasy `VelocityTracker` wraz z metodą `obtain()` wywołujemy metodę `recycle()`. Na listingu 25.8 umieszczono przykładową procedurę obsługi metody `onTouch->Event()` dla aktywności. Okazuje się, że aktywność posiada metodę zwrotną `onTouchEvent()`, która jest wywoływana za każdym razem, gdy żaden widok nie przetworzy zdarzenia dotknięcia. Ponieważ korzystamy ze standardowego, pustego układu graficznego, żaden widok nie obsługuje zdarzeń dotknięcia.

---

**Listing 25.8.** Przykładowa aktywność wykorzystująca klasę `VelocityTracker`

---

```
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.VelocityTracker;

public class MainActivity extends Activity {
    private static final String TAG = "VelocityTracker";

    /** Wywoływaną podczas pierwszego utworzenia aktywności. */
    @Override
```

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}

private VelocityTracker vTracker = null;

public boolean onTouchEvent(MotionEvent event) {
    int action = event.getAction();
    switch(action) {
        case MotionEvent.ACTION_DOWN:
            if(vTracker == null) {
                vTracker = VelocityTracker.obtain();
            }
            else {
                vTracker.clear();
            }
            vTracker.addMovement(event);
            break;
        case MotionEvent.ACTION_MOVE:
            vTracker.addMovement(event);
            vTracker.computeCurrentVelocity(1000);
            Log.v(TAG, "Predkosc w osi X wynosi " + vTracker.getXVelocity() +
                  " pikseli na sekunde");
            Log.v(TAG, "Predkosc w osi Y wynosi " + vTracker.getYVelocity() +
                  " pikseli na sekunde");
            break;
        case MotionEvent.ACTION_UP:
        case MotionEvent.ACTION_CANCEL:
            vTracker.recycle();
            break;
    }
    return true;
}
}
```

Czytelnikowi należy się jeszcze kilka najważniejszych uwag na temat klasy *VelocityTracker*. Oczywiście w przypadku dodania tylko jednego obiektu *MotionEvent* do klasy *VelocityTracker* (na przykład zdarzenia *ACTION\_DOWN*) wartość obliczonej prędkości będzie zawsze równa 0. Musimy dodać jednak punkt początkowy, aby — począwszy od niego — mogła zostać obliczona prędkość dla kolejnych zdarzeń *ACTION\_MOVE*. Okazuje się, że prędkości zgłoszone po zdarzeniu *ACTION\_UP* również mają zdefiniowaną wartość 0. Zatem nie spodziewajmy się, że prędkości w kierunkach *X* i *Y* odczytane po dodaniu zdarzenia *ACTION\_UP* będą reprezentowały ruch. Jeżeli na przykład tworzymy grę, w której użytkownik rzuca obiekt na ekranie, powinniśmy wykorzystać prędkości zebrane od momentu ostatniego zdarzenia *ACTION\_MOVE* do obliczenia trajektorii lotu obiektu w widoku gry.

Klasa *VelocityTracker* wymaga dużej ilości zasobów, zatem należy rozsądnie z niej korzystać. Powinniśmy się także upewnić, że klasa ta będzie mogła być wielokrotnie wykorzystywana, jeśli będzie potrzebna innym aplikacjom. W Androidzie może być uruchomionych kilka klas *VelocityTracker*, jednak pochłaniają one wiele pamięci. Kiedy więc utworzona klasa *VelocityTracker* przestanie być potrzebna, należałoby ją zwolnić. Na listingu 25.8, zamiast ponownie wykorzystywać starą sekwencję obsługi dotyku, stosujemy metodę *clear()* do uruchomienia nowej sekwencji (jeśli na przykład generujemy nowe zdarzenie *ACTION\_DOWN*, a obiekt *VelocityTracker* już istnieje).

## Analiza funkcji przeciągania

Skoro już Czytelnik poznał kod służący do odbierania obiektów MotionEvent, warto wykorzystać tę wiedzę w ciekawy sposób. Wyjaśnimy sposób implementacji funkcji przeciągania obiektów. Rozpoczniemy od procesu przesuwania. W następnej przykładowej aplikacji utworzymy białą kropkę i przeciągniemy ją do innego miejsca w układzie graficznym. Za pomocą kodu pokazanego na listingu 25.9 utworzmy nowy projekt i skonfigurujmy plik XML układu graficznego, a także dodajmy nową klasę, nazwaną Dot. Pamiętajmy, że nazwa pakietu w pliku XML układu graficznego dla elementu Dot musi dokładnie pasować do nazwy pakietu stosowanego dla tej aplikacji. Zwróćmy także uwagę, że możemy pozostawić główną klasę Activity bez zmian, gdyż nie musimy jej modyfikować. Interfejs UI tej aplikacji został zaprezentowany na rysunku 25.2.

**Listing 25.9.** Przykładowy plik XML układu graficznego i kod Java przykładowej aplikacji służącej do przeciągania obiektów

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

    <com.androidbook.touch.dragdemo1.Dot
        android:id="@+id/dot"
        android:tag="trueDot"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.View;

public class Dot extends View {
    private static final float RADIUS = 20;
    private float x = 30;
    private float y = 30;
    private float initialX;
    private float initialY;
    private float offsetX;
    private float offsetY;
    private Paint backgroundPaint;
    private Paint myPaint;

    public Dot(Context context, AttributeSet attrs) {
        super(context, attrs);
```

```
backgroundPaint = new Paint();
backgroundPaint.setColor(Color.BLUE);

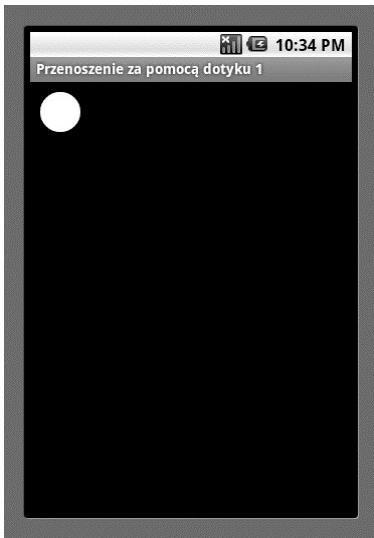
myPaint = new Paint();
myPaint.setColor(Color.WHITE);
myPaint.setAntiAlias(true);
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    int action = event.getAction();
    switch(action) {
        case MotionEvent.ACTION_DOWN:
            // Musi zapamiętać położenie środka punktu startowego
            // naszego obiektu Dot oraz miejsce, w którym następuje dotknięcie
            initialX = x;
            initialY = y;
            offsetX = event.getX();
            offsetY = event.getY();
            break;
        case MotionEvent.ACTION_MOVE:
        case MotionEvent.ACTION_UP:
        case MotionEvent.ACTION_CANCEL:
            x = initialX + event.getX() - offsetX;
            y = initialY + event.getY() - offsetY;
            break;
    }
    return(true);
}

@Override
public void draw(Canvas canvas) {
    int width = canvas.getWidth();
    int height = canvas.getHeight();
    canvas.drawRect(0, 0, width, height, backgroundPaint);

    canvas.drawCircle(x, y, RADIUS, myPaint);
    invalidate();
}
}
```

Po uruchomieniu tej aplikacji ujrzymy białą kropkę na niebieskim tle. Możemy ją dotknąć i przesuwać po obszarze ekranu. Po oderwaniu palca od ekranu kropka nie wraca na pozycję początkową i jest gotowa do dalszego przesuwania. Znacznie uprościliśmy cały proces, aby zaprezentować jedynie podstawy przesuwania obiektu po ekranie. Metoda `draw()` umieszcza kropkę w bieżącej pozycji  $X$  i  $Y$ . Dzięki obiektom `MotionEvent` przekazywanym w metodzie `onTouchEvent()` możemy w miarę przesuwania palca po ekranie modyfikować współrzędne  $X$  i  $Y$ . Rejestrujemy położenie początkowe kropki oraz początkowy punkt dotknięcia ekranu w momencie otrzymania zdarzenia `ACTION_DOWN`. Ponieważ użytkownik nie zawsze dotyka środka obiektu, współrzędne dotyku nie są tożsame z współrzednymi położenia obiektu. Musimy również wziąć pod uwagę przypadek, gdy punktem odniesienia dla naszego obiektu jest nie środek, a lewy górny róg ekranu.



**Rysunek 25.2.** Interfejs użytkownika aplikacji demonstrującej zjawisko przesuwania obiektu

Kiedy palec porusza się po ekranie, dopasowujemy położenie obiektu poprzez obliczanie różnic współrzędnych X i Y na podstawie otrzymywanych zdarzeń MotionEvent. Po zakończeniu ruchu (na przykład z powodu wywołania działania ACTION\_UP) definiujemy końcowe położenie obiektu, stanowiące ostatnie współrzędne dotyku. Troszeczkę tu oszukujemy, ponieważ widok Dot jest umieszczony na ekranie względem punktu (0, 0). Oznacza to, że możemy po prostu narysować kropkę zależną od punktu (0, 0), w przeciwieństwie do innego punktu referencyjnego. Gdyby nasz obiekt nie odnosił się do punktu (0, 0), musielibyśmy wstawić dodatkowe pozycje dla położenia tego obiektu. Nie musimy się także przejmować w naszym przykładzie paskami przewijania, które skomplikowałyby obliczenia położenia obiektu na ekranie. Podstawowa zasada jest jednak zawsze taka sama. Znając początkową pozycję przesuwanego obiektu oraz śledząc zmiany wartości dotyku, począwszy od zdarzenia ACTION\_DOWN do zdarzenia ACTION\_UP, możemy dokładnie ustalić położenie obiektu na ekranie.

Upuszczanie obiektu na inny obiekt wymaga jedynie znajomości położenia elementów na ekranie. Nie zaprezentujemy przykładowi upuszczania obiektu, wyjaśnimy jednak zasady działania tego mechanizmu. Jak widzieliśmy wcześniej, w czasie przesuwania obiektu po ekranie znamy jego relatywne położenie w stosunku do jednego lub kilku punktów na ekranie. Możemy również poznać rozmiary i położenie innych elementów. Mając te dane, możemy określić, czy przenoszony obiekt znajduje się „ponad” innym obiektem. Zazwyczaj stosowanym algorytmem jest iterowanie po dostępnych obiektach, nad którymi możemy umieścić nasz element, oraz sprawdzanie, czy bieżące położenie przesuwanego obiektu pokrywa się z położeniem obiektu znajdującego się „pod spodem”. W tym celu mogą zostać wykorzystane rozmiar i położenie każdego obiektu (czasami również kształt). Jeżeli otrzymamy zdarzenie ACTION\_UP, oznaczające, że użytkownik upuścił przenoszony element, oraz obiekt ten znajduje się nad innym elementem, muszą zostać uruchomione algorytmy przetwarzające działanie upuszczenia. Może to być na przykład proces przenoszenia obiektu do kosza, z którego umieszczony obiekt może zostać usunięty, albo proces kopiowania lub przenoszenia pliku do folderu.

#### Uwaga!

W wersji 3.0 Androida (Honeycomb) została wprowadzona bezpośrednia obsługa funkcji przeciągania. Została ona omówiona w rozdziale 31.

## Wielodotykowość

Po omówieniu obsługi ekranu za pomocą jednego palca możemy przejść do kwestii wielodotykowości. Technologia wielodotykowości zyskała olbrzymie zainteresowanie od czasu konferencji TED w 2006 roku, w czasie której Jeff Han zademonstrował wielodotykową powierzchnię dla komputerowego interfejsu UI. Używanie wielu palców na ekranie otwiera mnóstwo możliwości manipulowania jego zawartością. Na przykład rozsunięcie dwóch palców na pliku graficznym może skutkować powiększeniem obrazu. Przyłożenie kilku palców na takim obrazie i ich obrót zgodnie z ruchem wskazówek zegara może obrócić ten rysunek. Obsługa wielodotykowości została wprowadzona w wersji 2.0 zestawu Android SDK. Pojawiła się możliwość wykorzystywania maksymalnie trzech palców jednocześnie (z technicznego punktu widzenia) do wykonywania takich czynności, jak przybliżanie, obracanie oraz wszelkie inne operacje, podczas których korzysta się z technologii wielodotykowości. Piszemy: „z technicznego punktu widzenia”, ponieważ pierwsze urządzenia wykorzystujące funkcję wielodotykowości pozwalały wyłącznie na obsługę dwóch palców. Jednak po dłuższym zastanowieniu można stwierdzić, że nie ma w tym żadnej magii. Jeżeli elektronika urządzenia pozwala na wykrywanie wielodotykowości oraz informuje aplikację o przesuwaniu palców po ekranie i o ich zdjęciu z wyświetlacza, aplikacja ta może odczytać, co użytkownik chciał przekazać tym gestem. Mimo że nie ma tu żadnej magii, technologia ta nie należy do najprostszych wynalazków. W tym podrozdziale spróbujemy pomóc w zrozumieniu idei wielodotykowości.

**Uwaga!**

W wersji 2.2 Androida wprowadzono w klasie `MotionEvent` pewne zmiany, jeszcze bardziej komplikujące już i tak złożony mechanizm wielodotykowości, w tym również przedawnienie dwóch omawianych w tym podrozdziale stałych (`ACTION_POINTER_ID_MASK` oraz `ACTION_POINTER_ID_SHIFT`). Oznacza to, że w przypadku urządzeń starszych generacji będziemy stosować mechanizmy omówione poniżej, natomiast sposób wprowadzania funkcji wielodotykowości w urządzeniach posiadających wersję 2.2 Androida lub nowsze zostanie wyjaśniony w dalszej części podrozdziału.

## Funkcja wielodotykowości przed wersją 2.2 Androida

Podstawy wielodotykowości są dokładnie takie same jak w przypadku pojedynczego dotyku. Tak jak poprzednio, po wykryciu zdarzenia dotknięcia są tworzone obiekty `MotionEvent`, a następnie obiekty te są przekazywane do odpowiednich metod. Program może przeczytać informacje o dotknięciach i zadecydować, jak je zinterpretować. Na podstawowym poziomie metody klasy `MotionEvent` są identyczne: to znaczy wywołujemy metody `getAction()`, `getDownTime()`, `getX()` i tak dalej. Jednak w przypadku dotknięcia przez liczbę palców większą od jednego obiekt `MotionEvent` musi zawrzeć informacje o dotknięciach wszystkich palców wraz z odpowiednim wyjaśnieniem. Wartość działania z metody `getAction()` jest przeznaczona dla obsługi dotknięcia jednego palca, nie dla wszystkich. Wartość `getDownTime()` jest definiowana przez dotknięcie powierzchni przez pierwszy palec i w przypadku tej metody czas jest mierzony, dopóki przynajmniej jeden z palców dotyka ekranu. Wartości położenia z metod `getX()` oraz `getY()`, a także metody `getPressure()` i `getSize()` mogą pobierać argumenty dla danego palca; musimy zatem skorzystać z jakiejś wartości indeksu, aby móc rozróżniać dotyk poszczególnych palców. Mamy do dyspozycji wcześniej opisane wywołania metod, które nie pobierają żadnych argumentów identyfikujących palec (na przykład wywołania metod `getX()` i `getY()`), zatem dotknięcie którego palca będzie źródłem wartości dla tych metod? Można to

odkryć samemu, jest to jednak dość czasochłonne zajęcie. Dlatego jeżeli nie bierzemy cały czas pod uwagę wielodotykowości, możemy uzyskać dziwne rezultaty. Przeanalizujemy dokładnie ten temat, żeby szczegółowo wyjaśnić, co należy robić.

Podstawową metodą klasy `MotionEvent` wykorzystywaną w przypadku obsługi wielodotykowości jest `getPointerCount()`. Zostaje w niej zdefiniowana liczba palców reprezentowanych w obiekcie `MotionEvent`, nie musi ona jednak informować o faktycznej liczbie palców dotykających ekran wielodotykowy — jest to zależne od sprzętu oraz od wersji systemu Android. Może się zdarzyć, że w pewnych urządzeniach metoda `getPointerCount()` nie będzie zgłaszała wszystkich palców dotykających ekranu, a tylko niektóre. Idźmy jednak dalej. Jak tylko zostanie zgłoszona większa liczba palców w obiektach `MotionEvent`, musimy się zająć obsługą indeksu oraz identyfikatorów wskaźnika.

Klasa `MotionEvent` przechowuje informacje dla wskaźników, począwszy od indeksu 0 do wartości reprezentującej liczbę palców zgłoszonych do tego obiektu. Indeks wskaźnika zawsze rozpoczyna się od wartości 0; jeżeli zostały zgłoszone trzy palce, indeksy ich wskaźnika przybiorą wartości 0, 1 i 2. Wywołania takich metod jak `getX()` muszą zawierać wartość indeksu wskaźnika palca, o którym chcemy uzyskać informacje. Identyfikatory wskaźnika stanowią wartości typu całkowitego, wskazujące obserwowany palec. Identyfikator wskaźnika posiada wartość 0 dla pierwszego palca, który dotknął ekranu, jednak nie zawsze jest rozpoczynany od tej wartości w przypadku palców naprzemiennie przytykanych i odrywanych od ekranu. Uznajmy identyfikator wskaźnika za nazwę palca śledzonego przez system. Na przykład wyobraźmy sobie dwuetapową sekwencję dotyku, wykonywaną przez dwa palce w następującej kolejności: przykładamy do ekranu palec 1, następnie palec 2, odrywamy palec 1 i po nim podnosimy palec 2. Pierwszy przyłożony palec otrzyma identyfikator od wartości 0. Drugi przyłożony palec otrzymuje identyfikator 1. Po podniesieniu pierwszego palca drugi palec nadal będzie posiadał identyfikator 1. W tym momencie indeks wskaźnika dla palca 2 uzyska wartość 0, ponieważ indeks ten zawsze rozpoczyna się od wartości 0. W omawianym przykładzie drugi palec (identyfikator wskaźnika 1) rozpoczyna jako indeks wskaźnika 1, gdy dotnie ekranu jako pierwszy, a następnie przeskakuje do indeksu wskaźnika 0 po podniesieniu pierwszego palca. Jednak nawet jeśli wyłącznie drugi palec pozostanie na powierzchni ekranu, będzie nadal zdefiniowany identyfikatorem o wartości 1. Nasze aplikacje będą korzystały z identyfikatorów wskaźnika do powiązania zdarzeń z określonym palcem, nawet jeżeli w tym czasie ekranu dotykają także inne palce. Spójrzmy na przykład.

Listing 25.10 przedstawia nowy plik XML układu graficznego oraz kod Java dla aplikacji obsługującej funkcję wielodotykowości. Utworzymy nowy projekt, posługując się informacjami zawartymi na listingu 25.10, i uruchommy go. Rysunek 25.3 stanowi ilustrację interfejsu użytkownika tej aplikacji.

**Listing 25.10.** Plik XML układu graficznego i kod Java aplikacji demonstrującej technologię wielodotykowości

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout1"
    android:tag="trueLayout"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
>
```

```
<TextView android:text="Dotknij palcami ekranu i spójrz na zawartość okna LogCat"
    android:id="@+id/message"
    android:tag="trueText"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true" />

</RelativeLayout>

// Jest to plik MainActivity.java
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.View;
import android.view.View.OnTouchListener;
import android.widget.RelativeLayout;

public class MainActivity extends Activity implements OnTouchListener {
    /** Wywoływane podczas pierwszego utworzenia aktywności. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        RelativeLayout layout1 = (RelativeLayout) findViewById(R.id.layout1);
        layout1.setOnTouchListener(this);
    }

    public boolean onTouch(View v, MotionEvent event) {
        String myTag = v.getTag().toString();
        Log.v(myTag, "-----");
        Log.v(myTag, "Umieszczono widok " + myTag + " w metodzie onTouch");
        Log.v(myTag, describeEvent(event));
        logAction(event);
        if( "true".equals(myTag.substring(0, 4))) {
            return true;
        }
        else {
            return false;
        }
    }

    protected static String describeEvent(MotionEvent event) {
        StringBuilder result = new StringBuilder(500);
        result.append("Działanie: ").append(event.getAction()).append("\n");
        int numPointers = event.getPointerCount();
        result.append("Ilość wskaźników: ")
        result.append(numPointers).append("\n");
        int ptrIdx = 0;
        while (ptrIdx < numPointers) {
            int ptrId = event.getPointerId(ptrIdx);
            result.append("Indeks wskaźnika: ").append(ptrIdx);
            result.append(", identyfikator wskaźnika: ").append(ptrId).append("\n");
            result.append("Lokacja: ").append(event.getX(ptrIdx));
        }
    }
}
```

```
result.append(" x ").append(event.getY(ptrIdx)).append("\n");
result.append(" Siła nacisku: ")
result.append(event.getPressure(ptrIdx));
result.append(" Rozmiar: ").append(event.getSize(ptrIdx))
result.append("\n");

ptrIdx++;
}
result.append("Czas dotknięcia: ").append(event.getDownTime());
result.append("ms\n").append("Czas zdarzenia: ");
result.append(event.getEventTime()).append("ms");
result.append(" Szacowany: ");
result.append(event.getEventTime()-event.getDownTime());
result.append(" ms\n");
return result.toString();
}

private void logAction(MotionEvent event) {
    int action = event.getAction();
    int ptrIndex = (action & MotionEvent.ACTION_POINTER_ID_MASK) >>>
                    MotionEvent.ACTION_POINTER_ID_SHIFT;
    action = action & MotionEvent.ACTION_MASK;
    if(action == 5 || action == 6)
        action = action - 5;
    int ptrId = event.getPointerId(ptrIndex);

    Log.v("Action", "Indeks wskaznika: " + ptrIndex);
    Log.v("Action", "Identyfikator wskaznika: " + ptrId);
    Log.v("Action", "Prawdziwa wartość działania: " + action);
}
}
```



Rysunek 25.3. Aplikacja demonstrująca zastosowanie wielodotykowości

Aplikacja działa na emulatorze, nie ma jednak możliwości symulowania dotknięcia ekranu kilkoma palcami. W takim przypadku zostaną wyświetcone wyniki podobne do przedstawionych w poprzednim przykładzie. Na listingu 25.11 przedstawiamy przykładowe komunikaty narzędzia *LogCat* dla opisanej kilka stron wcześniej sekwencji dotyku, to znaczy: najpierw palec 1 dotyka ekranu, następnie ekranu dotyka palec 2, potem palec 1 odrywa się od ekranu, wreszcie to samo dzieje się z palcem 2.

**Listing 25.11.** Przykładowe wyniki narzędzia LogCat dla aplikacji wielodotykowej

```
trueLayout -----  
trueLayout Uzyskano widok trueLayout w metodzie onTouch  
trueLayout Działanie: 0  
trueLayout Ilość wskaźników: 1  
trueLayout Indeks wskaźnika: 0, Id wskaźnika: 0  
trueLayout Położenie: 114.88211 x 499.77502  
trueLayout Siła nacisku: 0.047058824 Rozmiar: 0.13333334  
trueLayout Czas dotknięcia: 33733650ms  
trueLayout Czas zdarzenia: 33733650ms Szacowany: 0 ms  
trueLayout Indeks wskaźnika: 0  
Action Identyfikator wskaźnika: 0  
Action Prawdziwa wartość działania: 0  
-----  
trueLayout -----  
trueLayout Uzyskano widok trueLayout w metodzie onTouch  
trueLayout Działanie: 2  
trueLayout Ilość wskaźników: 1  
trueLayout Indeks wskaźnika: 0, Id wskaźnika: 0  
trueLayout Położenie: 114.88211 x 499.77502  
trueLayout Siła nacisku: 0.05882353 Rozmiar: 0.13333334  
trueLayout Czas dotknięcia: 33733650ms  
trueLayout Czas zdarzenia: 33733740ms Szacowany: 90 ms  
Action Indeks wskaźnika: 0  
Action Identyfikator wskaźnika: 0  
Action Prawdziwa wartość działania: 2  
-----  
trueLayout -----  
trueLayout Uzyskano widok trueLayout w metodzie onTouch  
trueLayout Działanie: 261  
trueLayout Ilość wskaźników: 2  
trueLayout Indeks wskaźnika: 0, Id wskaźnika: 0  
trueLayout Położenie: 114.88211 x 499.77502  
trueLayout Siła nacisku: 0.05882353 Rozmiar: 0.13333334  
trueLayout Indeks wskaźnika: 1, Id wskaźnika: 1  
trueLayout Położenie: 320.30692 x 189.67395  
trueLayout Siła nacisku: 0.0509080393 Rozmiar: 0.13333334  
trueLayout Czas dotknięcia: 33733650ms  
trueLayout Czas zdarzenia: 33733962ms Szacowany: 312 ms  
Action Indeks wskaźnika: 1  
Action Identyfikator wskaźnika: 1  
Action Prawdziwa wartość działania: 0  
-----  
trueLayout -----  
trueLayout Uzyskano widok trueLayout w metodzie onTouch  
trueLayout Działanie: 2  
trueLayout Ilość wskaźników: 2  
trueLayout Indeks wskaźnika: 0, Id wskaźnika: 0  
trueLayout Położenie: 111.474594 x 499.77502  
trueLayout Siła nacisku: 0.05882353 Rozmiar: 0.13333334  
trueLayout Indeks wskaźnika: 1, Id wskaźnika: 1
```

```
trueLayout      Położenie: 320.30692 x 189.67395
trueLayout      Siła nacisku: 0.050980393 Rozmiar: 0.13333334
trueLayout      Czas dotknięcia: 33733650ms
trueLayout      Czas zdarzenia: 33734189ms Szacowany: 539 ms
Action          Indeks wskaźnika: 0
Action          Identyfikator wskaźnika: 0
Action          Prawdziwa wartość działania: 2
-----
trueLayout      Uzyskano widok trueLayout w metodzie onTouch
trueLayout      Działanie: 6
trueLayout      Ilość wskaźników: 2
trueLayout      Indeks wskaźnika: 0, Id wskaźnika: 0
trueLayout      Położenie: 111.474594 x 499.77502
trueLayout      Siła nacisku: 0.05882353 Rozmiar: 0.13333334
trueLayout      Indeks wskaźnika: 1, Id wskaźnika: 1
trueLayout      Położenie: 320.30692 x 189.67395
trueLayout      Siła nacisku: 0.050980393 Rozmiar: 0.13333334
trueLayout      Czas dotknięcia: 33733650ms
trueLayout      Czas zdarzenia: 33734228ms Szacowany: 578 ms
Action          Indeks wskaźnika: 0
Action          Identyfikator wskaźnika: 0
Action          Prawdziwa wartość działania: 1
-----
trueLayout      Uzyskano widok trueLayout w metodzie onTouch
trueLayout      Działanie: 2
trueLayout      Ilość wskaźników: 1
trueLayout      Indeks wskaźnika: 0, Id wskaźnika: 1
trueLayout      Położenie: 318.84656 x 191.45105
trueLayout      Siła nacisku: 0.050980393 Rozmiar: 0.13333334
trueLayout      Czas dotknięcia: 33733650ms
trueLayout      Czas zdarzenia: 33734240ms Szacowany: 590 ms
Action          Indeks wskaźnika: 0
Action          Identyfikator wskaźnika: 1
Action          Prawdziwa wartość działania: 2
-----
trueLayout      Uzyskano widok trueLayout w metodzie onTouch
trueLayout      Działanie: 1
trueLayout      Ilość wskaźników: 1
trueLayout      Indeks wskaźnika: 0, Id wskaźnika: 1
trueLayout      Położenie: 314.95224 x 190.5625
trueLayout      Siła nacisku: 0.050980393 Rozmiar: 0.13333334
trueLayout      Czas dotknięcia: 33733650ms
trueLayout      Czas zdarzenia: 33734549ms Szacowany: 899 ms
Action          Indeks wskaźnika: 0
Action          Identyfikator wskaźnika: 1
Action          Prawdziwa wartość działania: 1
```

---

Zastanówmy się, co się dzieje w tej aplikacji. Pierwszym zdarzeniem jest działanie ACTION\_DOWN (wartość działania równa 0) wywołane przyłożeniem pierwszego palca. Mówią nam o tym metoda `getAction()`. Aby dowiedzieć się, jakie wyniki są generowane przez poszczególne metody, wystarczy spojrzeć na kod metody `describeEvent()` umieszczony w pliku *MainActivity.java*. Otrzymujemy jeden wskaźnik, którego indeks i identyfikator mają przypisaną wartość 0. Następnie zostanie prawdopodobnie wygenerowanych kilka zdarzeń ACTION\_MOVE (wartość działania równa 2) dla tego palca, chociaż na listingu 25.11 widzimy tylko jedno z nich. Ciągle posiadamy jeden wskaźnik zawierający wymienione wcześniej wartości.

Chwilę później dotykamy ekranu drugim palcem. Działanie otrzymuje teraz wartość dziesiętną równą 261. Co to oznacza? Wartość działania składa się z dwóch części: wskaźnika, dla którego jest wykonywane działanie, oraz rodzaju przeprowadzanej czynności. Po przekształceniu wartości dziesiętnej 261 na wartość szesnastkową otrzymujemy 0x00000105. Działanie jest oznaczone przez najmniejszy bajt (w naszym wypadku 5), natomiast identyfikator wskaźnika jest definiowany przez następny dostępny bajt (u nas jest to 1). Zwróćmy uwagę, że jesteśmy informowani o identyfikatorze wskaźnika, a nie o jego indeksie. Po dotknięciu ekranu trzecim palcem działanie uzyska wartość 0x00000205 (517 w systemie dziesiętnym). Dotknięcie czwartym palcem zmieniłoby wartość na 0x00000305 (dziesięćte 773) i tak dalej. Nie widzieliśmy jeszcze działania o wartości równej 5, jest ono jednak znane jako ACTION\_POINTER\_DOWN. Różni się od działania ACTION\_DOWN jedynie tym, że jest wykorzystywane w mechanizmie wielodotykowości.

Przyjrzyjmy się następnej parze rekordów okna *LogCat* z listingu 25.11. Pierwszy rejestr odpowiada za zdarzenie ACTION\_MOVE (wartość działania równa 2). Pamiętajmy, że trudno utrzymać palce w bezruchu na prawdziwym ekranie. Demonstrujemy tylko jedno zdarzenie ACTION\_MOVE, może ich jednak wystąpić więcej. Po podniesieniu pierwszego palca otrzymujemy wartość działania równą 0x00000006 (w systemie dziesiętnym jest to wartość 6). Podobnie jak we wcześniejszym przypadku, indeks wskaźnika wynosi 0, a działanie nosi nazwę ACTION\_POINTER\_UP (analogicznym działaniem w mechanizmie wykorzystującym jeden palec jest ACTION\_UP). Jeżeli podniemy drugi palec, otrzymamy działanie 0x00000106 (262 w kodzie decimalnym). Zauważmy, że nadal otrzymujemy informacje o dwóch palcach, gdy jeden z nich generuje zdarzenie ACTION\_UP.

Ostatnia para rekordów na listingu 25.11 przedstawia jeszcze jedno zdarzenie ACTION\_MOVE dla drugiego palca, po którym następuje działanie ACTION\_UP. Tym razem widzimy wartość działania równą 1 (ACTION\_UP). Nie otrzymaliśmy wartości 262, ale za chwilę wyjaśnimy dlaczego. Odnotujmy również fakt, że po oderwaniu pierwszego palca od ekranu indeks wskaźnika uległ zmianie z wartości 1 na 0, ale identyfikator wskaźnika ciągle wynosi 0.

Zdarzenia ACTION\_MOVE nie pozwalają określić, który palec został przesunięty. Dla każdego ruchu zawsze będzie definiowana wartość 2, bez względu na liczbę wykrywanych palców czy to, z którym palcem mamy do czynienia. Wszystkie pozycje przyłożonych palców są przechowywane w obiekcie *MotionEvent*, więc musimy odczytywać współrzędne i analizować sytuację. Jeżeli do ekranu zostanie przyłożony tylko jeden palec, identyfikator wskaźnika pozwoli nam określić, który palec się porusza, gdyż będzie to jedyny dostępny wskaźnik. Na listingu 25.11 w miejscu, gdzie pozostał przyłożony tylko drugi palec, zdarzenie ACTION\_MOVE posiadało indeks o wartości 0 oraz identyfikator wskaźnika równy 1, nie było więc problemu z określeniem palca.

Wracając do początku listingu 25.11, indeks wskaźnika o wartości 0 dla pierwszego przyłożonego palca posiada wartość 0, dlaczego więc nie otrzymujemy wartości 0x00000005 (dziesięćte 5) dla działania, skoro palec ten dotknął ekranu przed innymi palcami? Jest to, niestety, pytanie, na które nie ma dobrej odpowiedzi. Możemy otrzymać wartość działania równą 5 w następującym przypadku: najpierw dotykamy ekranu pierwszym palcem, następnie drugim palcem, dzięki czemu otrzymujemy wartości działania, odpowiednio, 0 i 261 (pomijamy na razie zdarzenia ACTION\_MOVE). Teraz podnosimy pierwszy palec (wartość działania 6) i ponownie przykładamy go do ekranu. Identyfikator wskaźnika dla drugiego palca pozostał niezmieniony i posiada wartość 1. Przez czas oderwania pierwszego palca od ekranu aplikacja miała jedynie informację o identyfikatorze wskaźnika posiadającym wartość 1. Po ponownym dotknięciu wyświetlacza pierwszy palec ponownie otrzymał identyfikator 0. Skoro używamy wielu palców,

otrzymujemy wartość działania równą 5 (indeks wskaźnika o wartości 0 i działanie o wartości 5). Odpowiedią na zadane wcześniej pytanie jest zatem zapewnienie kompatybilności wstępnej, nie jest to jednak powód do radości. Wartości działań 0 i 1 pochodzą jeszcze sprzed ery wielodotykowości, więc aplikacje napisane w tamtych czasach będą działać, pod warunkiem że będzie wykorzystywany tylko jeden palec. W przypadku korzystania z dwóch palców, jeżeli pierwszy palec dotknie ekranu w jednym miejscu, a po nim drugi palec dotknie innego obszaru ekranu, podniesienie pierwszego palca nie zostanie rozpoznane przez aplikację nieobsługującą zdarzeń wielodotykowości. Wynika to z faktu, że podniesienie tego palca spowoduje wygenerowanie wartości działania 6, a nie 1. Dopiero podniesienie drugiego palca spowoduje utworzenie wartości działania równej 1. Taka aplikacja będzie odbierała opisaną sekwencję dotykania ekranu dwoma palcami, jakby w grę wchodził tylko jeden palec — tak jakby pierwszy palec miał się magicznie teleportować do miejsca, w którym znajduje się drugi.

Gdy ekranu dotyka tylko jeden palec, Android interpretuje jego zachowanie jako działania obsługiwane za pomocą jednego palca. Otrzymujemy zatem starą wartość ACTION\_UP równą 1 zamiast wielodotykowej wartości ACTION\_UP równej 6. Kod aplikacji musi takie przypadki obsłużyć z ostrożnością. Identyfikator wskaźnika o wartości 0 mógłby wygenerować wartość zdarzenia ACTION\_DOWN w przedziale od 0 do 5, w zależności od wykorzystywanych wskaźników. Ostatni podniesiony palec zawsze wygeneruje wartość zdarzenia ACTION\_UP równą 1, bez względu na identyfikator wskaźnika.

Klasa MotionEvent zawiera pewne stałe pomocnicze, które przydają się do zrozumienia sytuacji: na przykład stała MotionEvent.ACTION\_POINTER\_3\_DOWN posiada wartość 0x00000205 (dziesiętnie 517), która — jak już wiemy — definiuje trzeci palec przyłożony do ekranu. Wartości te mogą się jednak okazać nie tak bardzo przydatne, jeżeli zdecydujemy się rozpoznawać przyłożone palce poprzez identyfikator wskaźnika umieszczony w drugim bajcie oraz poprzez działanie widoczne w pierwszym bajcie. W rzeczywistości byłoby nawet lepiej, gdybyśmy używali innych stałych klasy MotionEvent do odczytywania wartości przekazywanych przez metodę `getAction()`. Myślimy tu o stałych MotionEvent.ACTION\_POINTER\_ID\_MASK, MotionEvent.ACTION\_MASK i MotionEvent.ACTION\_POINTER\_ID\_SHIFT. Poprzez połączenie wartości działania z każdą wymienioną maską za pomocą operacji AND oraz przeniesienie wyniku wygenerowanego dla identyfikatora wskaźnika będziemy mogli w rzetelny sposób określić sytuację, bez względu na to, ile palców jest obsługiwanych przez urządzenie. Twórcy systemu Androida musieli również to zrozumieć, ponieważ takie stałe, jak ACTION\_POINTER\_3\_DOWN, zostały przedawnione.

Sprawa jednak wymaga dłuższego zastanowienia. Omawiane stałe indeksów mają w nazwach człony ID, a nie INDEX, niedawno zaś wspominaliśmy, że w drugim bajcie jest przechowywany indeks wskaźnika. Niestety, w wersjach Androida starszych od 2.2 nie można jednoznacznie określić, co jest definiowane przez ten bajt. W wersji 2.2 systemu nazwy tych stałych zostały pozmieniane na ACTION\_POINTER\_INDEX\_MASK oraz ACTION\_POINTER\_INDEX\_SHIFT, ich wartości jednak pozostały takie same. Drugi bajt zawsze przechowuje indeks wskaźnika, jednak w starszych wersjach systemu nazwy tych stałych wprowadzały w błąd. Zostały one zaktualizowane w wersji 2.2 Androida, a wersje tych stałych zawierających w nazwie człon ID zostały uznane za przestarzałe. Nie bójmy się tworzyć własnych stałych, wykorzystywanych we wszystkich wersjach Androida.

W poprzednim przykładzie wprowadziliśmy metodę `logAction()`, pozwalającą na rozszerzanie wartości działania. Na listingu 25.12 umieściliśmy odpowiedni fragment kodu.

**Listing 25.12.** Przykładowy kod służący do określenia rezultatu działania metody MotionEvent.getAction()

---

```
int action = event.getAction();
int ptrIndex = (action & MotionEvent.ACTION_POINTER_ID_MASK) >>>
                MotionEvent.ACTION_POINTER_ID_SHIFT;
action = action & MotionEvent.ACTION_MASK;
if(action == 5 || action == 6)
    action = action - 5;
int ptrId = event.getPointerId(ptrIndex);
```

---

Po wykonaniu instrukcji umieszczonych na listingu 25.12 atrybut `ptrId` będzie przechowywał identyfikator wskaźnika związanego z określonym działaniem, `action` będzie posiadał wartości w przedziale od 0 do 4, a w atrybucie `ptrIndex` zostanie umieszczona wartość indeksu wskaźnika stosowana przez metodę `getX()` oraz inne metody klasy `MotionEvent`. Spoglądając na wartości przekazywane z metody `getAction()`, można je interpretować w taki sposób, że wartości większe od 4 reprezentują wartości związane z identyfikatorem wskaźnika. Wartości mniejsze lub równe 4 są związane jedynie z używanym palcem, bez względu na jego identyfikator wskaźnika. W niektórych przypadkach może zaistnieć potrzeba odjęcia wartości 5 od wartości działania, aby móc wykorzystać zdarzenia `ACTION_DOWN` oraz `ACTION_UP` nawet w mechanizmie wielodotykowości. W innych przypadkach nie jest to konieczne. Decyzja zależy od programisty.

## Funkcja wielodotykowości w systemach poprzedzających wersję 2.2

W wersji 2.2 Androida wprowadzono kilka istotnych zmian do mechanizmu działania wielodotykowości. Wspomnialiśmy w poprzednim punkcie o uznaniu pewnych stałych za przestarzałe oraz o wprowadzeniu nowych. W tej wersji systemu pojawiły się również dwie nowe metody — `getActionMasked()` oraz `getActionIndex()` — ułatwiające określenie wskaźnika oraz indeksu wykorzystywanych w danym działaniu. Za pomocą tych metod możemy podmienić kod z listingu 25.12 fragmentem z listingu 25.13.

**Listing 25.13.** Przykładowy kod pozwalający na określenie wartości działania

---

```
int action = event.getActionMasked();
int ptrIndex = event.getActionIndex();
int ptrId = event.getPointerId(ptrIndex);
```

---

Kod ten jest o wiele prostszy od zaprezentowanego na listingu 25.12. Zwrócić jednak uwagę, że w działaniu trzeba uwzględnić następujące zmienne: `ACTION_DOWN`, `ACTION_UP`, `ACTION_MOVE`, `ACTION_CANCEL`, `ACTION_OUTSIDE`, `ACTION_POINTER_DOWN` lub `ACTION_POINTER_UP` (wartości kolejno od 0 do 6). Jeżeli chcemy wykorzystać tego typu rozwiązanie, możemy po prostu odjąć 5, w przypadku gdy metoda `getActionMasked()` przekaże wartość większą od 4. Możemy również pomóczyć się z dwiema dodatkowymi wartościami.

Jak już wcześniej wspomnialiśmy, zanim zaczniemy tworzyć własną maskę, widoczną na przykład na listingu 25.12, musimy pamiętać, że począwszy od wersji 2.2 Androida, stałe `ACTION_POINTER_ID_MASK` oraz `ACTION_POINTER_ID_SHIFT` zostały uznane za przestarzałe, a na ich miejsce wprowadzono stałe `ACTION_POINTER_INDEX_MASK` oraz `ACTION_POINTER_INDEX_SHIFT`, definiujące dokładnie te same wartości. Ponieważ te zaktualizowane stałe nie są rozpoznawane

przez starsze wersje systemu, pozostańmy lepiej przy tworzeniu stałych posiadających wartości, odpowiednio, 0x0000ff00 oraz 0x00000008, ponieważ będą one właściwie interpretowane w każdej wersji systemu.

## Obsługa map za pomocą dotyku

Aplikacje przeznaczone do pracy z mapami również obsługują zdarzenia dotyku. Widzieliśmy już, że poprzez dotknięcie mapy możemy wywołać kontrolkę zmiany skali mapy, możemy też ją przesuwać. Są to wbudowane funkcje map. A w jaki sposób można wdrożyć inne pomysły wykorzystania dotykowości? Zademonstrujemy implementację kilku ciekawych rozwiązań związanych z mapami, między innymi funkcji pobierającej współrzędne geograficzne lokalizacji po jej dotknięciu na mapie. W ten sposób mamy dostęp do bardzo wielu przydatnych funkcji.

Jedną z głównych klas służących do obsługi map jest klasa `MapView`. Klasa ta, podobnie jak omawiana wcześniej klasa `View`, zawiera metodę `onTouchEvent()`, której jedynym argumentem jest obiekt `MotionEvent`. Możemy także użyć metody `setOnTouchListener()` wobec klasy `MapView` do skonfigurowania procedury obsługi metod zwrotnych, reagujących na dotknięcie. Innymi głównymi obiektami dla map jest zestaw nakładek `Overlay`, w tym takie klasy, jak `ItemizedOverlay` i `MyLocationOverlay`. Wszystkie wymienione obiekty zostały omówione w rozdziale 17. Klasy `Overlay` również posiadają metodę `onTouchEvent()`, chociaż jej sygnatura jest nieco inna od analogicznej metody używanej w standardowej klasie `View`. Dla klasy `Overlay` wygląda ona tak:

```
onTouchEvent(android.view.MotionEvent e, MapView mapView)
```

Możemy przesłonić metodę `onTouchEvent()`, jeśli chcemy wprowadzić inne formy obsługi map. Częściej spotykamy się z przesyaniem metod w klasie `Overlay` niż w klasie `MapView`, zatem w tym punkcie zajmiemy się tym zagadnieniem. Podobnie jak we wcześniej omówionych przypadkach, metoda `onTouchEvent()` klas `Overlay` obsługuje obiekty `MotionEvent`. Nawet w przypadku map zdarzenie `MotionEvent` przechowuje współrzędne X i Y obszaru dotkniętego przez użytkownika. Jest to tutaj niemal nieprzydatne, ponieważ przeważnie będziemy chcieli znać współrzędne geograficzne dotkniętego punktu, a nie współrzędne ekranu. Na szczęście istnieją rozwiązania tego problemu.

Klasa `MapView` została wyposażona w interfejs `Projection`, który zawiera metody przetwarzające piksele na obiekty `GeoPoint` i odwrotnie. Dostęp do tego interfejsu uzyskujemy poprzez wywołanie metody `MapView.getProjection()`. Po wprowadzeniu interfejsu `Projection` do konwersji możemy wykorzystać metody `fromPixels()` i `toPixels()`. Pamiętajmy, że klasa `Projection` jest przydatna jedynie wtedy, gdy mapa nie ulega zmianie w widoku. Wewnątrz metody `onTouchEvent()` możemy za pomocą metody `fromPixels()` przekonwertować wartości X i Y położenia na obiekt `GeoPoint`.

Przydatną i jednocześnie interesującą metodą klasy `Overlay` jest metoda `onTap()`, bardzo podobna do omówionej wcześniej metody `onTouch()`, różniącej się jednak pewnym kluczowym aspektem. Klasy `Overlay` nie posiadają metody `onTouch()`. Sygnatura metody `onTap()` została pokazana poniżej:

```
public boolean onTap(GeoPoint p, MapView mapView)
```

Oznacza to, że jeśli użytkownik dotknie mapy z nakładką `Overlay`, zostanie wywołana metoda `onTap()` wraz z obiektem `GeoPoint`, który określa współrzędne wskazanego miejsca. W ten sposób zaoszczędzimy mnóstwo czasu, gdyż nie będziemy musieli podejmować prób określenia dotkniętego miejsca na mapie. Nie musimy się już martwić o konwersję współrzędnych `X` i `Y` lokacji na współrzędne geograficzne. Zajmuje się tym system.

Przyjrzymy się teraz ponownie przykładowi z rozdziału 17., w którym wyświetliśmy mapę wraz z przyciskami pozwalającymi na jej przeglądanie w różnych trybach (satelitarny, uliczny, widok ruchu ulicznego oraz tryb standardowy). Do tego projektu dodamy możliwość uruchamiania trybu widoku ulicznego lokacji wskazanej na mapie. W tym celu musimy umieścić nakładkę `Overlay` w widoku `MapView`, a po dotknięciu nakładki `Overlay` zdarzenie to zostanie przekształcone na wskazaną lokalizację na mapie. Po przekształceniu w taki sposób lokalizacji uruchomimy intencję wywołującą tryb widoku ulicznego. Rozpoczniemy od utworzenia w środowisku Eclipse kopii aplikacji `MapViewDemo` z rozdziału 17. (listingi 17.2 i 17.3). Następnie wykorzystamy informacje z listingu 25.14 do zmodyfikowania metody `onCreate()` głównej klasy `Activity` oraz dodamy nową klasę, również umieszczoną na listingu 25.14, w pliku `ClickReceiver.java`. Zmiany w metodzie `onCreate()` zostały zaznaczone pogrubioną czcionką. Interfejs, widoczny na rysunku 17.3, nie ulegnie zmianie.

**Listing 25.14.** Dodawanie funkcji dotyku do aplikacji `MapViewDemo`

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.mapview);

    mapView = (MapView) findViewById(R.id.mapview);

    ClickReceiver clickRecvr = new ClickReceiver(this);
    mapView.getOverlays().add(clickRecvr);
    mapView.invalidate();
}

// Jest to plik ClickReceiver.java
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.util.Log;

import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;

public class ClickReceiver extends Overlay{
    private static final String TAG = "ClickReceiver";
    private Context mContext;

    public ClickReceiver(Context context) {
        context = context;
    }

    @Override
    public boolean onTap(GeoPoint p, MapView mapView) {
        Log.v(TAG, "Otrzymano kliknięcie w tym punkcie: " + p);

        if(mapView.isStreetView()) {

```

```

        Intent myIntent = new Intent(Intent.ACTION_VIEW, Uri.parse
            ("google.streetview:cbll=" +
            (float)p.getLatitudeE6() / 1000000f +
            "," + (float)p.getLongitudeE6() / 1000000f
            +"&cbp=1,180,,0,1.0"
            ));
        mContext.startActivity(myIntent);
        return true;
    }
    return false;
}
}

```

To wystarczy, aby uruchomić zmodyfikowaną wersję aplikacji — chyba że nie posiadamy aplikacji widoku ulicznego<sup>1</sup> (StreetView) na emulatorze lub w urządzeniu. Aplikacja StreetView została umieszczona w emulatorach CupCake (1.5) i Donut (1.6), a została usunięta z emulatora Éclair (2.0). Jeżeli brakuje nam programu StreetView na emulatorze, możemy wykorzystać rzeczywiste urządzenie wyposażone w tę funkcjonalność i przetestować na nim utworzoną aplikację. Osoby posiadające wyłącznie emulator mogą wykonać następujące czynności:

1. Skonfiguruj urządzenie AVD oparte na interfejsie Google API w wersji 1.6 lub 1.5.
2. Uruchom emulator za pomocą urządzenia AVD zdefiniowanego w punkcie 1.
3. Wykonaj polecenie adb pull /system/app/StreetView.apk, aby skopiować aplikację StreetView.apk z emulatora na dysk twardy stacji roboczej.
4. Skonfiguruj urządzenie AVD dla wersji interfejsu Google API, z którego zamierzasz korzystać.
5. Zatrzymaj emulator z punktu 2. i uruchom emulator z punktu 4.
6. Wykonaj polecenie adb install StreetView.apk na pliku .apk skopionym w punkcie 3.

Aplikacja StreetView powinna zostać zainstalowana na emulatorze, dzięki czemu nasza przykładowa aplikacja zadziała. W nowszych wersjach Androida nazwa pakietu tej aplikacji brzmi Street.apk, zatem być może uda nam się znaleźć wersję nowszą od umieszczonej w Androidzie 1.6.

Po uruchomieniu świeżo zmodyfikowanej aplikacji MapViewDemo wykonajmy zbliżenie pozwalające na zobaczenie ulic miasta. Kliknijmy przycisk *Ulica*, aby ulice obsługiwane przez aplikację StreetView (zdjęcia tych ulic są zamieszczone na przykład w bazie danych Google) zostały zaznaczone na niebiesko. Dotknijmy teraz jednej z ulic, a zostanie wywołana metoda onTap() klasy ClickReceiver, która z kolei skontaktuje za pomocą intencji aktywność aplikacji StreetView z dotkniętą lokalizacją. Jeżeli dotknijemy obszar mapy nieobsługiwany przez aplikację StreetView, pojawi się pusty ekran tej aplikacji wraz z komunikatem typu „nieprawidłowa panorama”. Oznacza to, że serwer Google nie może odnaleźć zdjęć znajdujących się w pobliżu wybranego miejsca. Kliknijmy przycisk cofania, aby wrócić do aplikacji obsługującej mapy, i sprawdzmy inną lokalizację. Jeżeli zajrzymy do okna LogCat, zauważymy, że zostały w nim zapisane współrzędne geograficzne dotkniętej lokacji. Zwrócmy uwagę, że obiekt GeoPoint definiuje szerokość i długość geograficzną danymi typu int, natomiast identyfikator URI aplikacji StreetView wymaga typu float.

<sup>1</sup> Tryb widoku ulicznego jeszcze nie jest dostępny w Polsce — *przyp. red.*

W naszej przykładowej aplikacji zdecydowaliśmy się na wysyłanie intencji zawierającej wspólnie geograficzne dotkniętej lokacji do aktywności aplikacji StreetView. Możemy sobie jednak wyobrazić również inne możliwości. Jeżeli znamy szerokość i długość geograficzną lokalizacji, możemy wykorzystać obiekt Geocoder do identyfikacji jej okolicy. Nic nie stoi na przeszkodzie, aby użyć informacji o lokalizacji do nawigacji typu zakręt po zakręcie. Istnieje możliwość zmierzenia odległości wskazanej lokalizacji od naszego bieżącego położenia. Możemy nawet zachować dane lokalizacji do późniejszego użytku.

## Gesty

Gesty są specjalnym przypadkiem zdarzenia dotyku. Pojęcie „gest” jest stosowane wobec różnorodnych czynności obsługiwanych przez system Android, począwszy od prostej sekwencji dotykowej, na przykład szarpnięcia lub ściśnięcia, aż do formalnych gestów klasy Gesture, które zostaną omówione w dalszej części rozdziału. Szarpnięcia, ściśnięcia, długie przyciśnięcia oraz gesty przewijania posiadają zdefiniowane zachowanie, aktywowane ściśle określonymi bodźcami. Większość osób wie, że szarpnięcie oznacza gest, w którym palec zostaje przyłożony do ekranu, dość szybko przesunięty w określonym kierunku i na zakończenie oderwany od wyświetlacza. Na przykład wykonanie tego gestu w aplikacji Galeria (ukazującej obrazy sekwencyjnie, od lewej do prawej strony) spowoduje, że kolejne obrazy przemkną przed naszymi oczami.

W niniejszym podrozdziale wykorzystamy wiedzę zdobytą na temat obiektów MotionEvent i rozszerzymy ją o gesty, korzystając z przykładowego gestu ściskania. To nie jest wcale takie trudne, jak mogłoby się wydawać. Gest ściskania nie jest jawnie obsługiwany w wersjach Androida starszych od 2.2, jeśli więc chcemy zaimplementować w nich ten ruch, musimy własnoręcznie napisać kod odczytujący obiekty zdarzeń oraz wykonujący odpowiednie działanie. Tym się właśnie zajmiemy. Począwszy od wersji 2.2 systemu, uzyskujemy kilka pomocnych funkcji, pozwalających na korzystanie z takich gestów jak ściskanie; poznamy je w dalszej części rozdziału.

Zaprezentujemy następnie kilka klas, pomocnych w definiowaniu innych gestów, na przykład szarpnięć i długich przyciśnięć. Stąd pozostałe tylko krok do wprowadzenia niestandardowych gestów, tj. gestów, które możemy sami zarejestrować, a których odtworzenie przez użytkownika w naszej aplikacji uruchomi określona czynność. Najpierw jednak pobawmy się gestem ściskania!

## Gest ściskania

Jednym z ciekawszych zastosowań wielodotykowości jest gest ściskania, wykorzystywany do zmiany skali obrazu. Mechanizm ten opiera się na koncepcji rozsuwania i ściskania palców: jeżeli dotknimy wyświetlacz dwoma palcami i je rozsunie my, aplikacja powinna zareagować powiększeniem obrazu, jeżeli natomiast je ściśnięmy, dany element zostanie zmniejszony. Gest ten jest najczęściej wykorzystywany w aplikacjach obsługujących obrazy, na przykład pokazujących mapy.

W celu zademonstrowania procesu implementacji gestu ściskania zmodyfikujemy poprzednią aplikację. Na listingu 25.15 widzimy nową wersję klasy ClickReceiver; reszta kodu pozostaje bez zmian. Zwróćmy uwagę, że ta aplikacja będzie działała na urządzeniach wyposażonych przynajmniej w wersję 2.2 Androida, co zostanie wyjaśnione po zaprezentowaniu listingu.

**Listing 25.15.** Kod Java implementujący gest ściskania

```
// Jest to plik ClickReceiver.java
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.util.FloatMath;
import android.util.Log;
import android.view.MotionEvent;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;

public class ClickReceiver extends Overlay {
    private static final String TAG = "ClickReceiver";
    private static final float ZOOMJUMP = 75f;
    private Context mContext;
    private boolean inZoomMode = false;
    private boolean ignoreLastFinger = false;
    private float mOrigSeparation;

    public ClickReceiver(Context context) {
        mContext = context;
    }

    @Override
    public boolean onTap(GeoPoint p, MapView mapView) {
        Log.v(TAG, "Otrzymano kliknięcie w tym punkcie: " + p);

        if(mapView.isStreetView()) {
            Intent myIntent = new Intent(Intent.ACTION_VIEW, Uri.parse
                ("google.streetview:cbll=" +
                (float)p.getLatitudeE6() / 1000000f +
                "," + (float)p.getLongitudeE6() / 1000000f
                +"&cbp=1,180,,0,1.0"
                ));
            mContext.startActivity(myIntent);
            return true;
        }
        return false;
    }

    public boolean onTouchEvent(MotionEvent e, MapView mapView) {
        Log.v(TAG, "W metodzie onTouchEvent, działaniem jest " + e.getAction());
        int action = e.getAction() & MotionEvent.ACTION_MASK;

        if(e.getPointerCount() == 2) {
            inZoomMode = true;
        }
        else {
            inZoomMode = false;
        }

        if(inZoomMode) {
            switch(action) {
                case MotionEvent.ACTION_POINTER_DOWN:
```

```

// Możemy rozpocząć nowy gest ściskania, więc przygotujmy się
mOrigSeparation = calculateSeparation(e);
break;
case MotionEvent.ACTION_POINTER_UP:
    // Kończymy gest ściskania, więc przygotujmy się
    // Ignoruje ostatni palec, gdyż tylko on
    // dotyka wyświetlacza
    ignoreLastFinger = true;
    break;
case MotionEvent.ACTION_MOVE:
    // Wykonujemy gest ściskania, więc decydujemy
    // o zmianie poziomu przybliżenia/oddalenia
    float newSeparation = calculateSeparation(e);
    if(newSeparation - mOrigSeparation > ZOOMJUMP) {
        // palce się rozeszły, przybliżamy
        mapView.getController().zoomIn();
        mOrigSeparation = newSeparation;
    }
    else if (mOrigSeparation - newSeparation > ZOOMJUMP) {
        // Palce zbliżyły się do siebie, oddalamy
        mapView.getController().zoomOut();
        mOrigSeparation = newSeparation;
    }
    break;
}
// Nie przekazujmy tych zdarzeń Androidowi, ponieważ
// zajmujemy się nimi
return true;
}
else {
    // W razie konieczności zeruje logikę przybliżania/oddalania
}

// Jeżeli tylko jeden palec jest przyłożony, odrzucamy zdarzenia,
// dopóki nie zostanie uniesiony
if(ignoreLastFinger) {
    if(action == MotionEvent.ACTION_UP)
        ignoreLastFinger = false;
    return true;
}

return super.onTouchEvent(e, mapView);
}

private float calculateSeparation(MotionEvent e) {
    float x = e.getX(0) - e.getX(1);
    float y = e.getY(0) - e.getY(1);
    return FloatMath.sqrt(x * x + y * y);
}
}

```

Do nakładki ClickReceiver dodaliśmy metodę zwrotną `onTouchEvent()`. Wewnątrz tej metody uzyskujemy każdy obiekt `MotionEvent`, który jest kierowany z ekranu dotykowego do widoku `MapView`. W większości przypadków po prostu przekazujemy je dalej. W ten sposób

przesuwanie mapy będzie możliwe w nieprzerwany sposób, będzie też można uruchomić funkcję StreetView poprzez stuknięcie. Jednak jeżeli na wyświetlaczu zostanie wykryte dotknięcie dwoma palcami, może to oznaczać rozpoczęcie gestu ściskania, więc przechodzimy w tryb zmiany skali. Jeżeli pojawiają się informacje o dotyku dwoma palcami, musimy zdecydować o dalszych operacjach, stąd instrukcja switch przy okazji zdarzenia działania.

Gdyby pojawiło się działanie ACTION\_POINTER\_DOWN (pamiętajmy, że pojawia się ono jedynie w przypadku wielodotykowości, która tutaj występuje, gdyż używamy dwóch palców), oznaczałoby to, że z dotyku jednym palcem użytkownik przeszedł na dwa palce. Po tym zdarzeniu możemy oczekwać gestu ściskania ze strony użytkownika. Aby określić, czy palce oddalają się od siebie, czy też zbliżają się do siebie, musimy zapamiętać odległość dzielącą je na początku gestu. Jest ona obliczana jako pierwiastek kwadratowy z sumy kwadratów wartości stanowiących różnice współrzędnych pomiędzy palcami — innymi słowy, stosujemy najwyklejsze twierdzenie Pitagorasa. Jesteśmy przekonani, że po przyłożeniu dwóch palców do ekranu obiekt zdarzenia będzie posiadał współrzędne zdefiniowane w indeksach 0 i 1, i nie ma znaczenia, jakie palce są wykorzystywane.

W przypadku wystąpienia zdarzenia ACTION\_POINTER\_UP jest to ostatnie zdarzenie przekazujące w wynikach informacje o dwóch palcach, zanim obiekty MotionEvent zaczynają przesyłać wyniki zawierające dane o jednym palcu. Będzie to oznaczało, że gest ściskania został zakończony. Jeżeli pozwolimy teraz systemowi obserwować zdarzenia wywoływane dotykiem jednego palca, aplikacja może się zacząć dziwnie zachowywać. Gdyby na przykład Android otrzymał zdarzenie ACTION\_UP z ostatniego palca, mógłby uznać je za stuknięcie i uruchomić funkcję StreetView, co nie byłoby pożądanym zachowaniem. Spodziewamy się, że użytkownik uniesie ostatni palec w momencie zakończenia gestu ściskania, zatem do tego czasu ignorujemy wszelkie inne zdarzenia. Dokonujemy tego za pomocą wartości true w atrybutie ignoreLastFinger, która będzie sprawdzana w momencie decydowania o losie zdarzeń.

Jeżeli otrzymaliśmy zdarzenie ACTION\_MOVE, palce mogą się przybliżać lub oddalać. Poprzez obliczenie nowej odległości dzielącej obydwa palce oraz porównanie jej ze starą wartością odległości możemy zdecydować, czy obraz ma zostać powiększony, pomniejszony, czy też jego rozmiar nie zmieni się. W przypadku zmiany stopnia przybliżenia musimy wyzerować starą wartość odległości. Użytkownik może nie odrywać palców od ekranu i ciągle oddalać lub przybliżać do siebie palce, więc nasza aplikacja musi odpowiednio reagować na te gesty. Jeżeli nie wykryjemy znaczącej zmiany w różnicy odległości pomiędzy palcami, będziemy ciągle otrzymywać zdarzenia, dopóki nie uzyskamy odpowiedniego dystansu lub użytkownik nie zakończy gestu ściskania.

Bez względu na przychodzące działanie, jeżeli znajdziemy się w trybie zmiany przybliżenia, otrzymujemy wartość true z metody onTouchEvent(). W ten sposób informujemy system o przetwarzaniu bieżącego zdarzenia oraz pozwalamy na przesyłanie następnych. Na końcu działania metody onTouchEvent() musimy zdecydować, czy zdarzenie ma zostać pozostałe Androidowi. Za pomocą zmiennej ignoreLastFinger uniemożliwiamy pozostawianie zdarzeń Androidowi, w przypadku gdy gest ściskania został zakończony, lecz ostatni palec ciągle jest przyłożony do wyświetlacza. Po uniesieniu tego palca, co zostanie oznajmione pojawieniem się działania ACTION\_UP, możemy wznowić przekazywanie zdarzeń systemowi. W ten sposób pozwalamy Androidowi zająć się gestami stuknięć i przeciągania, lecz samodzielnie przetwarzamy gest ściskania. Oczywiście, sami możemy zająć się wspomnianymi gestami we wnętrzu tej metody zwrotnej.

W czasie testowania tej aplikacji będziemy mogli w dalszym ciągu przesuwać mapę w różne strony oraz za pomocą stuknięcia uruchamiać tryb StreetView, jednak teraz gesty ściskania i rozciągania pozwolą nam na zwiększenie lub zmniejszenie skali mapy.

Zignorowaliśmy na moment możliwość dotykania ekranu trzema palcami, a następnie odejmowania jednego z nich, dzięki czemu pozostałyby dwa aktywne palce. Wiele urządzeń rozpoznaje maksymalnie tylko dwa jednocześnie przyłożone palce do ekranu, powinniśmy się jednak spodziewać, że coraz więcej urządzeń będzie rozpoznawało większą liczbę palców. Jeżeli chcemy wykorzystywać gest ściskania w aplikacji, która nie obsługuje map, będziemy musieli własnoręcznie zdefiniować tryb powiększania danych obiektów. Jeżeli na przykład na ekranie wyświetlamy obraz, który chcemy powiększać i zmniejszać w reakcji na gesty rozciągania i ściskania, będziemy musieli odpowiednio nim manipulować po wystąpieniu działania ACTION\_MOVE, tak jak to zostało wcześniej zaprezentowane. Niedługo zajmiemy się omówieniem podobnego przykładu.

Wspomnieliśmy wcześniej, że gest ściskania nie był jawnie obsługiwany aż do pojawienia się wersji 2.2 Androida, a chociaż omawiany kod działa również w tej wersji, warto wykorzystać nowe funkcje, pozwalające na łatwiejsze wprowadzenie tego gestu do aplikacji. Warto zauważyc, że klasa MapView, począwszy od wersji 2.2 Androida, posiada standardowo wbudowaną obsługę gestu ściskania; działa to bez żadnej dodatkowej obsługi, więc nie musimy odnosić się do żadnego kodu odpowiedzialnego za ten gest w nowszych wersjach systemu. Zanim przyjrzymy się natywnej obsłudze gestu ściskania, musimy najpierw omówić klasę, która jest dostępna od samego początku — GestureDetector.

## Klasy GestureDetector i OnGestureListener

Co prawda implementacja gestu ściskania nie wymagała dużego wysiłku, byłoby jednak miło, gdyby Android zaoferował jakiś pomoc w rozpoznawaniu najpowszechniejszych gestów. Od programisty wymagałoby to wtedy jedynie wprowadzenia w aplikacji odpowiedniej logiki, reagującej na dane gesty. Na szczęście Android posiada właśnie taki mechanizm, chociaż musieliśmy czekać aż do wersji 2.2 Androida na klasę umożliwiającą jawną obsługę gestu ściskania.

Pierwszą klasą tego typu jest obecna od samego początku istnienia Androida klasa GestureDetector, której zadaniem jest otrzymywanie obiektów MotionEvent oraz informowanie o sekwencji zdarzeń przypominającej jeden ze standardowych gestów. Wszystkie obiekty zdarzeń są przekazywane z metody zwrotnej do tej klasy, która z kolei wywołuje inne metody zwrotne po rozpoznaniu gestu, na przykład szarpnięcia lub długiego wcisnięcia. Musimy zarejestrować obiekt nasłuchujący dla metod zwrotnych klasy GestureDetector — i to właśnie w nim definiujemy logikę precyzującą działania, jakie mają zostać wykonane po wykryciu jednego ze standardowych gestów. Niestety, klasa ta nie rozpoznaje gestu ściskania; do tego celu musimy skorzystać z innej, nowej klasy, do której wkrótce dojdziemy.

Istnieje kilka sposobów utworzenia obiektu nasłuchującego. Pierwszym rozwiązaniem jest napisanie nowej klasy, w której zostanie zaimplementowany odpowiedni interfejs obiektu nasłuchującego, na przykład GestureDetector.OnGestureListener. Mamy do dyspozycji kilka abstrakcyjnych metod. Trzeba je zaimplementować dla każdej metody zwrotnej, która może wystąpić.

Drugą opcją jest zastosowanie jednej z prostych implementacji obiektu nasłuchującego i przesłonięcie odpowiednich metod zwrotnych. Na przykład klasa GestureDetector.SimpleOnGestureListener posiada wszystkie metody zwrotne zaimplementowane w taki sposób, że nie wykonują one żadnej operacji i przekazują wartość false. Wystarczy rozszerzyć tę

klasę i przesłonić kilka metod, które będą przetwarzane po wykryciu odpowiednich gestów. Po zostało metody będą zaimplementowane w domyślny sposób. To drugie rozwiązanie wydaje się bardziej perspektywiczne, nawet w przypadku przesłonięcia wszystkich metod zwrotnych, ponieważ jeżeli w kolejnych wersjach Androida do interfejsu zostaną dodane kolejne metody abstrakcyjne, dzięki takiej prostej implementacji będziemy mieli do dyspozycji ich domyślne formy, więc będziemy zabezpieczeni.

W wersji 2.2 Androida wprowadzono klasę `ScaleGestureDetector`, która służy do rozpoznawania gestu ściskania. Zamierzamy zademonstrować tę klasę wraz z odpowiednim obiektem nasłuchującym. W tym celu pokażemy przykładowy kod z odpowiednim rysunkiem. Rozszerzamy tu prostą implementację interfejsu `ScaleGestureDetector.SimpleOnScaleGestureListener` dla obiektu nasłuchującego. Na listingu 25.16 zamieściliśmy układ graficzny oraz kod Java aktywności `MainActivity`.

**Listing 25.16.** Układ graficzny oraz kod Java odpowiedzialne za wykrywanie gestu ściskania za pomocą klasy `ScaleGestureDetector`

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/layout" android:orientation="vertical"
    android:layout_width="fill_parent" android:layout_height="fill_parent" >

    <TextView android:text="Użyj gestu ściskania do zmiany rozmiaru obrazu."
        android:layout_width="fill_parent" android:layout_height="wrap_content" />

    <ImageView android:id="@+id/image" android:src="@drawable/icon"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:scaleType="matrix" />

</LinearLayout>

// Jest to plik MainActivity.java
import android.app.Activity;
import android.graphics.Matrix;
import android.os.Bundle;
import android.util.Log;
import android.view.MotionEvent;
import android.view.ScaleGestureDetector;
import android.widget.ImageView;

public class MainActivity extends Activity {
    private static final String TAG = "ScaleDetector";
    private ImageView image;
    private ScaleGestureDetector mScaleDetector;
    private float mScaleFactor = 1f;
    private Matrix mMatrix = new Matrix();
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        image = (ImageView) findViewById(R.id.image);
        mScaleDetector = new ScaleGestureDetector(this,
            new ScaleListener());
    }
}
```

```

    }

@Override
public boolean onTouchEvent(MotionEvent ev) {
    Log.v(TAG, "w metodzie onTouchEvent");
    // Przekazuje wszystkie zdarzenia klasie ScaleGestureDetector
    mScaleDetector.onTouchEvent(ev);

    return true;
}

private class ScaleListener extends
ScaleGestureDetector.SimpleOnScaleGestureListener {
    @Override
    public boolean onScale(ScaleGestureDetector detector) {
        mScaleFactor *= detector.getScaleFactor();

        // Upewniamy się, że obraz nie będzie za mały lub za duży
        mScaleFactor = Math.max(0.1f, Math.min(mScaleFactor, 5.0f));

        Log.v(TAG, "w metodzie onScale, współczynnik skali = " + mScaleFactor);
        mMMatrix.setScale(mScaleFactor, mScaleFactor);

        image.setImageMatrix(mMMatrix);
        image.invalidate();
        return true;
    }
}
}

```

Układ graficzny jest bardzo prosty. Zawarta w nim została kontrolka `TextView` wyświetlająca komunikat o wykorzystaniu gestu ścisania oraz widok `ImageView` ze standardową ikoną Androida. Wykonując gest ścisania, będziemy zmieniać rozmiar tej ikony. Oczywiście, w jej miejscu możemy wstawić dowolny obraz. Wystarczy umieścić odpowiedni plik obrazu w folderze `drawable` oraz zmienić atrybut `android:src` w pliku układu graficznego. Zwróciśmy uwagę na atrybut `android:scaleType` naszego obrazu. Za jego pomocą informujemy system, że do przeprowadzania operacji skalowania obrazu będziemy używać macierzy grafiki. Chociaż służy ona również do przesuwania obrazu, skupimy się teraz wyłącznie na procesie skalowania. Zauważmy także, że zdefiniowaliśmy największy dopuszczalny widok `ImageView`. Nie chcemy, aby w trakcie skalowania obrazu był on obcinany przez granice widoku `ImageView`.

Również sam kod nie jest skomplikowany. Wewnątrz metody `onCreate()` uzyskujemy odniesienie do obrazu i tworzymy klasę `ScaleGestureDetector`. Jedynymi czynnościami w metodzie zwrotnej `onTouchEvent()` są odbieranie wszelkich obiektów zdarzeń oraz przekazywanie wartości `true` w celu ciągłego otrzymywania nowych zdarzeń. W ten sposób klasa `ScaleGestureDetector` obserwuje wszystkie zdarzenia i może zadecydować, kiedy poinformować system o wykryciu gestu.

W obiekcie `ScaleListener` jest przeprowadzany proces skalowania. W klasie obiektu nasługującego znajdziemy trzy metody zwrotne: `onScaleBegin()`, `onScale()` oraz `onScaleEnd()`. Pierwsza i trzecia metoda nie są nam potrzebne, więc ich nie implementujemy.

We wnętrzu metody `onScale()` możemy wykorzystać przekazany wykrywacz do uzyskania wielu informacji na temat operacji skalowania. Współczynnik skalowania oscyluje przeważnie w granicach wartości 1. Oznacza to, że w przypadku ściskania palców wartość tego współczynnika wynosi nieco mniej niż 1, w czasie ich rozsuwania staje się ona nieznacznie większa od 1. Domyślna wartość zmiennej `mScaleFactor` wynosi 1, więc stopniowo, w zależności od ruchu palców, jest ona zmniejszana lub zwiększa. Wartość 1 współczynnika skalowania definiuje normalny rozmiar obrazu. W przeciwnym wypadku obraz zostaje pomniejszony lub powiększony, zależnie od tego, czy wartość tego współczynnika maleje, czy rośnie. Za pomocą eleganckiej kombinacji funkcji `min` i `max` nakładamy na ten współczynnik pewne ograniczenia. Zapobiegamy w ten sposób zbytniemu powiększeniu lub zmniejszeniu obrazu. Następnie wykorzystujemy zmienną `mScaleFactor` do przeskalowania macierzy grafiki i stosujemy taką zaktualizowaną macierz wobec obrazu. Wywołanie metody `invalidate()` wymusza proces ponownego rysowania obrazu na ekranie.

Jak widać, wkładamy tu o wiele mniej wysiłku niż w poprzednim przykładzie, w którym musielibyśmy własnoręcznie zajmować się obiektyami zdarzeń. Możemy teraz zająć się implementacją odpowiedniej logiki uruchamianej standardowym gestem. Praca z interfejsem `OnGestureListener` bardzo przypomina czynności przeprowadzane na interfejsie `ScaleListener`; jedyna różnica polega na obecności metod zwrotnych odpowiedzialnych za inne gesty.

Standardowe gesty są użyteczne, ale niekiedy programista chce móc obsłużyć własne gesty wewnętrz aplikacji. Być może należy umożliwić użytkownikom narysowanie gestu zaznaczenia, dzięki czemu aplikacja wykona jakąś operację. Potrzebne nam będą do tego niestandardowe gesty, którymi się teraz zajmiemy.

## Niestandardowe gesty

W ostatniej części tego rozdziału przyjrzymy się formalnym klasom typu `Gesture`. Zgodnie z definicją gestem nazywamy uprzednio zarejestrowany ruch po ekranie dotykowym, którego aplikacja oczekuje od użytkownika. Jeżeli użytkownik wykona taki gest w aplikacji, zacznie ona wykonywać operacje zdefiniowane dla tego ruchu. Potrzebne są nakładki wykrywające dany ruch, które przekazują informacje o wykryciu ruchu do głównej aktywności. Stosowanie gestów pozwala uprościć interfejsu użytkownika, gdyż przyciski oraz inne kontrolki stają się niepotrzebne i są wypierane przez szybkie ruchy palcami lub inne gesty. Mogą one stanowić również interesujące interfejsy gier. W tym punkcie pokażemy, w jaki sposób można rejestrować własne gesty i programować ich obsługę w aplikacji. Zwrócmy uwagę, że wszelkie klasy związane z gestami nie są w ogóle wykorzystywane w tym przykładzie; prezentujemy tutaj zupełnie inny zestaw klas.

## Aplikacja Gestures Builder

Zanim zaprezentujemy kod odpowiedzialny za implementację gestów, warto zapoznać się z dostępną w zestawie Android SDK aplikacją `Gestures Builder`, która pomoże w zrozumieniu koncepcji gestów. Aplikacja `Gestures Builder` tworzy plik gestów, zawierający bibliotekę gestów, i pozwala na zarządzanie nim. Włączmy emulator z poziomu środowiska Eclipse, przejdźmy do menu aplikacji i kliknijmy ikonę `Gestures Builder`. Ikona ta została pokazana na rysunku 25.4.

Jeżeli emulator nie zawiera aplikacji `Gestures Builder`, musimy utworzyć nowy projekt w środowisku Eclipse. Jest ona dostępna jako przykładowa aplikacja w katalogu zestawu Android SDK, mianowicie w folderze `platforms/<wersja>/samples/GestureBuilder` lub w katalogu `samples`.



Rysunek 25.4. Ikona aplikacji Gestures Builder

Tworzymy nowy projekt za pomocą opcji *Create project from existing sample*. Wybieramy odpowiednią wersję Androida, aby odblokować rozwijalne menu z dostępnymi przykładowymi aplikacjami, i wybieramy stamtąd aplikację GestureBuilder. Możemy następnie zainstalować tę aplikację w emulatorze.

Zobaczmy niemal puste okno. Kliknijmy przycisk *Add*. Aplikacja wyświetli monit o wprowadzenie nazwy; nazwa ta zostanie powiązana z gestem, który za chwilę zarejestrujemy. Będzie ona używana w kodzie w odniesieniu do tego gestu i w pewnym sensie posłuży nam jako nazwa polecenia. Gdy użytkownik wykona zakodowany gest, jego nazwa zostanie przekazana do metod, aby aplikacja potrafiła przetworzyć żądanie użytkownika. Nazwa powinna być rzecznikiem, na przykład „spiral” albo „checkmark”, ewentualnie może brzmieć jak polecenie, na przykład „fetch” albo „stop”. Naszym pierwszym gestem będzie zaznaczenie, zatem wpiszmy w polu *Name* nazwę **checkmark**. Narysujmy teraz w tym pustym oknie duży znak zaznaczenia, w emulatorze za pomocą myszy, a w urządzeniu — za pomocą palca. Jeżeli jesteśmy niezadowoleni z rezultatu, możemy spróbować ponownie; stary gest zniknie w momencie rozpoczęcia rysowania nowego. Gdy już będziemy zadowoleni z wyniku, kliknijmy przycisk *Done*. Powinien pojawić się ekran podobny do zaprezentowanego na rysunku 25.5.

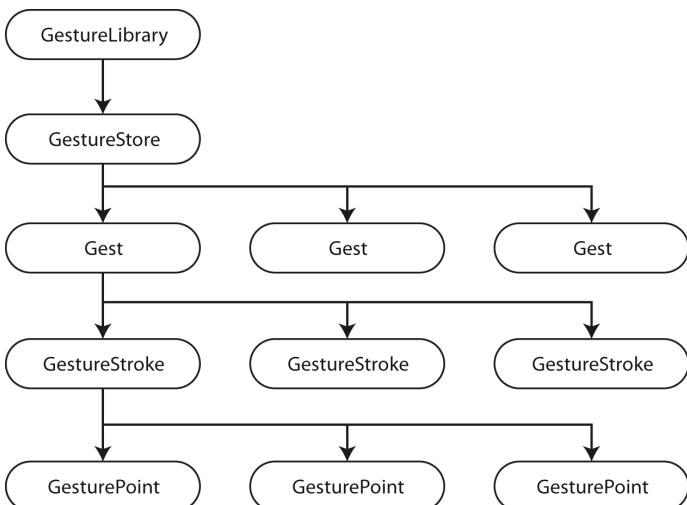


Rysunek 25.5. Gest zaznaczenia zapisany na karcie pamięci

Zwróćmy uwagę, że możemy narysować różne rodzaje gestów zaznaczania i wszystkie nazwać **checkmark**. Zarejestrujmy przynajmniej jeszcze jeden taki gest i nazwijmy go również **checkmark**; powinien w jakiś sposób różnić się od pierwotnego gestu, chociażby rozmiarem. Dodajmy również za pomocą przycisku *Add gesture* inne gesty, nadając im odmienne nazwy. Każdorazowe wcisnięcie przycisku *Done* powoduje dodanie kolejnego gestu do biblioteki.

Możemy spróbować utworzyć gest wielodotykowy poprzez narysowanie dwoma palcami równoległych linii imitujących znak równości. Funkcja ta nie działa i zostanie narysowana tylko jedna linia. Być może w następnych wersjach systemu będą obsługiwane gesty wielodotykowe — to znaczy gesty wymagające co najmniej dwóch palców.

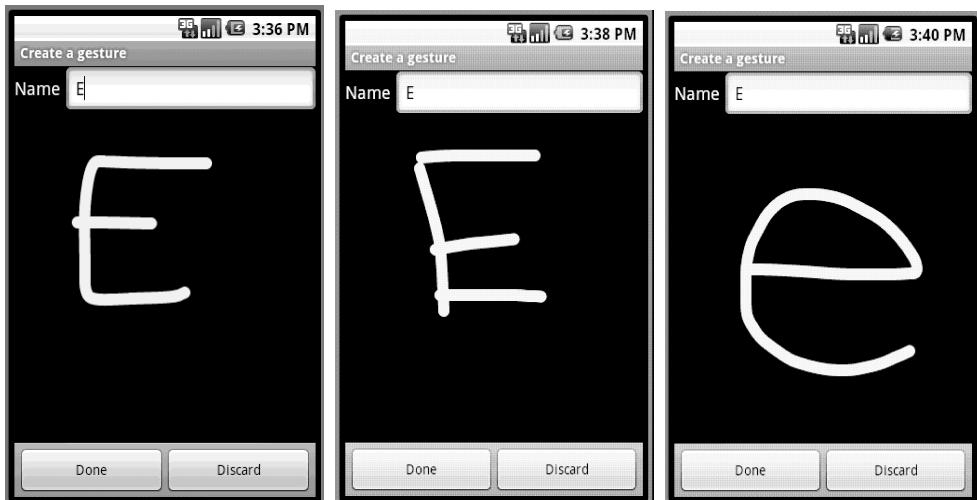
Każdy gest posiada nazwę i składa się z **gestów właściwych** (ang. *stroke*). Gestem właściwym jest sekwencja dotyku rozpoczynająca się od dotknięcia palcem ekranu, a kończąca na jego oderwaniu od wyświetlacza. Jak już wiemy, sekwencja dotyku jest tworzona przez zdarzenia MotionEvent. W analogiczny sposób gest właściwy jest tworzony przez punkty gestu. Gesty są przechowywane w **magazynie gestów**. **Biblioteka gestów** zawiera jeden taki magazyn. W Androidzie wszystkie te klasy można wykorzystać w kodzie. Na rysunku 25.6 został przedstawiony schemat powiązań pomiędzy poszczególnymi klasami gestów.



Rysunek 25.6. Struktura klas gestów

Chociaż do utworzenia gestu nie możemy używać funkcji wielodotykowości, istnieje sposób uwzględnienia wielu gestów właściwych w obrębie jednego gestu. Aby na przykład zdefiniować gest oznaczający literę *E*, potrzebujemy co najmniej dwóch gestów właściwych; jeden gest właściwy może definiować górny, boczny i dolny odcinek litery, a drugi gest właściwy połuży do narysowania jej środkowego odcinka. Możemy także najpierw narysować za pomocą jednego gestu właściwego pionową linię w literze *E*, a następnie trzy gesty właściwe wykorzystać do narysowania trzech linii poziomych. Istnieją różne metody narysowania litery *E* i na szczęście mamy możliwość zapisania ich wszystkich w bibliotece gestów. Zarejestrujmy gest odpowiedzialny za literę *E* na kilka różnych sposobów, ponieważ użytkownicy mogą korzystać z odmiennych technik rysowania tej litery, a chcemy, żeby aplikacja rozpoznała ten gest bez względu na sposób jego wykonania. Rysunek 25.7 przedstawia różne sposoby rejestracji liter E.

Utworzenie gestu składającego się z wielu gestów właściwych może stanowić nie lada wyzwanie na emulatorze. Jak już wspomnieliśmy, możemy ponownie narysować nowy gest na wcześniejszym geście, który zostanie usunięty. Skąd więc Android wie, kiedy rysujemy gest od nowa, a kiedy dodajemy jedynie nowy gest właściwy do istniejącego gestu? Android stosuje w tym celu



Rysunek 25.7. Różne sposoby rejestrowania litery „E”

wartość atrybutu `FadeOffset`, która jest podawana w milisekundach. Jeśli po jej przekroczeniu zaczniemy rysować gest, Android uzna, że cały proces należy przeprowadzić od początku. Domyślna wartość tego atrybutu wynosi 420 milisekund. Oznacza to, że jeśli podczas rysowania gestu uniesiemy palec na ponad 420 milisekund, system uzna, że skończyliśmy rysowanie tego gestu, i zostanie on zapamiętany w takim stanie. W rzeczywistym urządzeniu taki czas może wystarczyć do rozpoczęcia rysowania kolejnego gestu właściwego. W przypadku emulatora może nie być tak dobrze. Wszystko zależy od szybkości stacji roboczej.

Jeżeli aplikacja Gestures Builder stwarza problemy z akceptacją gestu składającego się z wielu gestów właściwych, możemy utworzyć własną wersję tej aplikacji i zmodyfikować domyślną wartość atrybutu `fadeOffset`. Opisaliśmy wcześniej sposób utworzenia projektu Gestures Builder w środowisku Eclipse. Postępujmy zgodnie z instrukcjami, a następnie przejdźmy do pliku `/res/layout/create_gesture.xml` i dodajmy atrybut `android:fadeOffset="1000"` do elementu `GestureOverlayView`. Wartość atrybutu `fadeOffset` zostanie powiększona do 1 sekundy (1000 milisekund). Możemy jednak wstawić tu dowolną wartość.

Poszukajmy miejsca, w którym są przechowywane gesty. Wiadomość typu `Toast` w aplikacji Gestures Builder informuje nas, że gesty są zapisywane w katalogu `/sdcard/gestures` (lub `/mnt/sdcard/gestures` od wersji 2.2 Androida). Skorzystajmy z perspektywy *File Explorer* w środowisku Eclipse lub z powłoki adb, aby odnaleźć folder `/sdcard` na emulatorze. Znajdziemy w nim plik `gestures`. Zwróćmy uwagę na jego niewielki rozmiar. Jest to plik binarny, zatem nie ma możliwości, aby go ręcznie edytować. Jeśli chcemy modyfikować jego zawartość, musimy otworzyć aplikację Gestures Builder. W trakcie tworzenia aplikacji obsługującej gesty musimy skopiować plik `gestures` do jej katalogu `/res/raw`. Dokonujemy tego za pomocą funkcji *File Copy*, znajdującej się w perspektywie *File Explorer*, lub za pomocą polecenia `adb pull`, co pozwala na skopiowanie pliku na dysk twardy, a stamtąd do projektu.

Mamy nie tylko możliwość dodawania nowych gestów za pomocą aplikacji Gestures Builder — poprzez długie kliknięcie możemy wywołać menu istniejącego gestu. Dzięki znajdującym się tu opcjom możemy zmienić nazwę gestu lub go usunąć. Nie można ponownie rejestrować zachowanego gestu, więc jeśli nam się nie podoba, musimy go usunąć i utworzyć nowy. Jak już

wspomnieliśmy, czasami pojawia się potrzeba zarejestrowania różnych odmian danego gestu i nadania im takiej samej nazwy. Nazwa gestu nie musi być niepowtarzalna, chociaż zalecane jest, aby identycznie nazwane gesty były do siebie podobne.

Teraz utworzymy przykładową aplikację obsługującą nasz nowy plik *gestures*. Utwórzmy nowy projekt Androida w środowisku Eclipse. Listing 25.17 zawiera zarówno plik XML układu graficznego, jak i kod Java aktywności.

**Listing 25.17.** Plik układu graficznego oraz kod Java aplikacji wykrywającej gesty

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Rysuj gesty, a ja odgadnę ich przeznaczenie."
    />

<android.gesture.GestureOverlayView
    android:id="@+id/gestureOverlay"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gestureStrokeType="multiple"
    android:fadeOffset="1000" />

</LinearLayout>

import java.util.ArrayList;
import android.app.Activity;
import android.gesture.Gesture;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.Prediction;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;

public class MainActivity extends Activity implements OnGesturePerformedListener {
    private static final String TAG = "Wykrywanie gestów";
    GestureLibrary gestureLib = null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
    gestureLib = GestureLibraries.fromRawResource(this, R.raw.gestures);
    gestureLib = GestureLibraries.fromFile("/sdcard/gestures");
    if (!gestureLib.load()) {
```

```

        Toast.makeText(this, "Nie można wczytać pliku /sdcard/gestures",
                      Toast.LENGTH_SHORT).show();
        finish();
    }
    // Przyjrzyjmy się bibliotece gestów, z którą będziemy pracować
    Log.v(TAG, "Funkcje biblioteki:");
    Log.v(TAG, "Styl ułożenia: " + gestureLib.getOrientationStyle());
    Log.v(TAG, "Typ sekwencji: " + gestureLib.getSequenceType());
    for( String gestureName : gestureLib.getGestureEntries() ) {
        Log.v(TAG, "Dla gestu " + gestureName);
        int i = 1;
        for( Gesture gesture : gestureLib.getGestures(gestureName) ) {
            Log.v(TAG, " " + i + ": ID: " + gesture.getID());
            Log.v(TAG, " " + i + ": Ilosc gestow wlasciwych: " +
                  gesture.getStrokesCount());
            Log.v(TAG, " " + i + ": Dlugosc gestu wlasciwego: " + gesture.getLength());
            i++;
        }
    }
    GestureOverlayView gestureView =
        (GestureOverlayView) findViewById(R.id.gestureOverlay);
    gestureView.addOnGesturePerformedListener(this);
}

@Override
public void onGesturePerformed(GestureOverlayView view, Gesture gesture) {
    ArrayList<Prediction> predictions = gestureLib.recognize(gesture);

    if (predictions.size() > 0) {
        Prediction prediction = (Prediction) predictions.get(0);
        if (prediction.score > 1.0) {
            Toast.makeText(this, prediction.name, Toast.LENGTH_SHORT).show();
            for(int i=0;i<predictions.size();i++)
                Log.v(TAG, "prognoza " + predictions.get(i).name +
                      " - wynik = " + predictions.get(i).score);
        }
    }
}
}

```

W tym przykładzie będziemy korzystać z tego samego pliku, który został zapisany przez aplikację Gestures Builder. Stosujemy w tym celu metodę `GestureLibraries.fromFile()` wewnętrznej metody `onCreate()`. Pokazujemy jednak również w komentarzach, w jaki sposób można uzyskać dostęp do pliku gestów stanowiącego część naszej aplikacji. Gdybyśmy wprowadzili metodę `fromRawResource()`, użylibyśmy argumentu przyjmującego postać zwykłego identyfikatora zasobów, natomiast plik gestów wstawilibyśmy do katalogu `/res/raw`.

Nasza aplikacja nie jest bardzo rozbudowana, lecz jej uruchomienie da nam większe pojęcie na temat zjawisk zachodzących w systemie podczas przetwarzania gestów. W momencie uruchamiania aplikacja wczytuje plik gestów i zapisuje informacje o znalezionych gestach. Wyświetla także wyniki prób dopasowania wzorcowego gestu do gestu narysowanego na ekranie. Włączmy aplikację wykrywającą gesty, przy oczywistym założeniu, że już uruchomiliśmy

aplikację Gestures Builder oraz zarejestrowaliśmy gesty w katalogu `/sdcard/gestures`. Popatrzymy na opis każdego gestu za pomocą identyfikatora, liczbę gestów właściwych oraz długości poszczególnych gestów właściwych.

Narysujmy gesty istniejące w bibliotece gestów. Następnie narysujmy takie, o których wiemy, że nie są obecne w tej bibliotece. Zobaczmy, co się dzieje w oknie *LogCat*. Zauważymy, że czasami nie są rozpoznawane gesty, które powinny zostać zidentyfikowane, oraz że czasami Android rozpoznaje opacznie gesty, przeważnie jednak właściwie odgaduje rysowane symbole. Zauważymy także, że w przypadku rozpoznania wprowadzonego gestu system wyświetla wyniki wszystkich gestów znajdujących się w bibliotece, a jeżeli nie rozpozna symbolu, nie uzyskujemy żadnego rezultatu.

Zobaczmy także, co się stanie, jeżeli będziemy zbytnio zwlekać z rysowaniem gestów właściwych, na przykład wolno rysując wcześniej opisaną literę *E*. Aplikacja pobierze dotychczas rysowane gesty właściwe i porówna je z biblioteką gestów, wskutek czego otrzymamy najprawdopodobniej błędny wynik, ewentualnie nie zostanie wyświetlony żaden rezultat. Wartość czasu przerwy pomiędzy rysowaniem poszczególnych gestów właściwych jest kontrolowana przez atrybut `FadeOffset`. W tym miejscu warto się chwilę zastanowić. Chcemy, aby Android zaczął analizować gesty tuż po zakończeniu procesu rysowania, jednak nie możemy sprawdzić, czy użytkownik skończył wprowadzać gest inaczej, niż czekając pewien czas. Zatem atrybut `FadeOffset` spełnia dwa zadania: pierwsza rola polega na kontrolowaniu czasu oczekiwania na wprowadzenie nowego gestu właściwego będącego częścią ogólnego gestu, a drugą funkcją tego atrybutu jest kontrola czasu oczekiwania na rozpoczęcie procesu porównywania wprowadzonych gestów z gestami umieszczonymi w bibliotece. Nadanie dużej wartości atrybutowi `FadeOffset` oznacza długi czas oczekiwania na rozpoczęcie procesu rozpoznawania gestów. Zbyt mała wartość tego atrybutu uniemożliwia rysowanie gestu składającego się z wielu gestów właściwych, ponieważ Android uzna rysowanie za zakończone przed wprowadzeniem wszystkich gestów właściwych. Czytelnik musi sam zadecydować, czy czas 420 milisekund jest wystarczający. Można również wprowadzić ustawienie preferencji, dzięki czemu użytkownicy sami dostosują czas oczekiwania do własnych potrzeb.

Skoro omawiamy temat gestów tworzonych przez wiele gestów właściwych, zwróćmy uwagę, że klasa `GestureOverlayView` posiada ustawienie pozwalające na ustalenie, czy aplikacja ma oczekwać tego typu gestów. W pliku XML tym atrybutem jest `android:gestureStrokeType`, a jego wartości to `single` (domyślna) lub `multiple`. Jeżeli chcemy wprowadzić możliwość stosowania gestów tworzonych przez większą liczbę gestów właściwych, musimy ustanowić ten atrybut. Możemy również skonfigurować go programowo za pomocą metody `setGestureStrokeType(int type)`, dla której argumentami są `GestureOverlayView.GESTURE_STROKE_TYPE_SINGLE` lub `GestureOverlayView.GESTURE_STROKE_TYPE_MULTIPLE`. Klasa `GestureOverlayView` zawiera również atrybuty XML i metody pozwalające na konfigurowanie kolorów i grubości linii.

Aby utworzyć własną aplikację obsługującą przetwarzanie gestów, musimy zadecydować, na które gesty będzie reagowała ta aplikacja, następnie musimy utworzyć bibliotekę tych gestów i zaimplementować — prawdopodobnie w klasie `Activity` — interfejs `onGesturePerformed` → `Listener` rozpoznający je i podejmujący odpowiednie działania.

Czy można sprawić, aby użytkownicy rejestrówali własne gesty? Na przykład mogą zechcieć stosować dla określonej operacji inny gest niż udostępniony w aplikacji. Jest to możliwe, jednak musimy sprawić, aby plik biblioteki gestów mógł być przez innych modyfikowany, natomiast rozsądną lokalizacją dla tego pliku jest karta SD. Całkiem łatwym procesem jest utworzenie nowego pliku biblioteki gestów, odczytywanie domyślnych gestów z pliku biblioteki

umieszczonego w aplikacji i zamiana tych domyślnych gestów na gesty dostosowane przez użytkownika. Możemy przejrzeć wspomnianą już implementację aplikacji Gestures Builder, aby poznać kod rejestratora gestów. Można również napisać aplikację Gestures Builder odpowiadającą na intencje, dzięki czemu w prosty sposób będzie wywoływana aktywność dodająca nowe gesty. Ewentualnie istnieje możliwość zarejestrowania wyłącznie gestów użytkownika w nowym, modyfikowalnym pliku biblioteki gestów, a następnie wczytania obydwu bibliotek do aplikacji — naszej i użytkownika. Wewnątrz metody `onGesturePerformed()` zostałaby najpierw zaimplementowana metoda `recognize()`, próbująca rozpoznawać gesty z biblioteki użytkownika, później z naszej. Aplikacja porównywałaby najbliższe przewidywania z obydwu bibliotek i na tej podstawie podejmowałaby decyzję co do wyboru gestu.

## Odbońniki

Poniżej prezentujemy odnośniki do materiałów, które pomogą Czytelnikowi zrozumieć zagadnienia omawiane w niniejszym rozdziale.

- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — znajdziemy tu zbiór projektów bezpośrednio związanych z książką. Prezentowane w tym rozdziale projekty zostały umieszczone w katalogu *ProAndroid3\_R25\_EkranyDotykowe*. W katalogu znajduje się również plik *Czytaj.TXT*, zawierający instrukcję importowania projektów do środowiska Eclipse.
- [http://www.ted.com/talks/jeff\\_han\\_demos\\_his\\_breakthrough\\_touchscreen.html](http://www.ted.com/talks/jeff_han_demos_his_breakthrough_touchscreen.html) — Jeff Han demonstruje swój interfejs użytkownika obsługujący wielodotykowość na konferencji TED w 2006 roku — świetny materiał.
- <http://android-developers.blogspot.com/2010/06/making-sense-of-multitouch.html> — wpis dotyczący wielodotykowości, w którym ukazany jest inny sposób implementacji klasy `GestureDetector` wewnątrz rozszerzenia widoku.

## Podsumowanie

W tym rozdziale zademonstrowaliśmy sposób obsługi ekranów dotykowych, począwszy od aplikacji rozpoznającej dotykanie jednym palcem, następnie zaś zajęliśmy się kwestią wielodotykowości. Wyjaśniliśmy zasadę działania funkcji obsługi dotyku w pracy z mapami, a także omówiliśmy niektóre pomocnicze klasy i metody ułatwiające przetwarzanie zdarzenia dotyku w aplikacji obsługującej mapy. W ostatnim podrozdziale przeanalizowaliśmy mechanizm gestów, umożliwiający wprowadzanie danych w nowy i prawdopodobnie prostszy sposób niż za pomocą klawiatury i innych kontrolek interfejsu użytkownika.



# Czujniki

Urządzenia obsługujące Androida często zawierają wbudowane czujniki sprzętowe, a Android posiada mechanizmy pozwalające na korzystanie z tych podzespołów. Praca z czujnikami może być ciekawa. Możliwość dokonywania pomiarów w świecie zewnętrznym oraz wykorzystywanie otrzymywanych danych w oprogramowaniu są niezmiernie emocjonujące. Mamy tu do czynienia z tym rodzajem doświadczenia programistycznego, którego nie zaznamy, pracując na standardowym komputerze, stojącym na biurku lub w serwerowni. Potencjał aplikacji korzystających z czujników jest olbrzymi i mamy nadzieję, że uda nam się zachęcić Czytelników do jego pełnej eksploatacji.

W niniejszym rozdziale zajmiemy się dostępną w Androidzie strukturą czujników. Wy tłumaczmy, czym są czujniki, w jaki sposób uzyskujemy ich odczyty, a także przeanalizujemy rodzaje i zastosowania otrzymywanych danych. Android posiada już kilka zdefiniowanych rodzajów czujników, należy jednak oczekwać pojawienia się zupełnie nowych kategorii tych elementów elektronicznych, które zostaną dołączone do architektury systemu.

## Czym jest czujnik?

W przypadku Androida czujnik jest wchodzącym w skład urządzenia podzespołem elektronicznym, pobierającym dane pochodzące ze środowiska zewnętrznego. Dane te są następnie przekazywane do aplikacji. Z kolei aplikacje wykorzystują odczyty czujników do informowania użytkownika o warunkach otoczenia, sterowania w grach, pracy w rzeczywistości rozszerzonej<sup>1</sup> lub do dostarczania użytecznych danych roboczych. Czujniki są jednokierunkowe — ich wyniki można wyłącznie odczytywać (wyjątkiem jest czujnik NCF, którym zajmiemy się w dalszej części rozdziału). To nieco upraszcza ich wykorzystywanie na potrzeby oprogramowania: trzeba skonfigurować obiekt nasłuchujący, odczytać dane pochodzące z czujników i następnie

<sup>1</sup> Rzeczywistość rozszerzona (ang. *Augmented Reality*) jest systemem umożliwiającym łączenie świata rzeczywistego z danymi generowanymi za pomocą komputerów. Najbardziej znanym przykładem AR jest wykorzystanie obrazu z kamery, na który nakłada się grafikę 3D generowaną w czasie rzeczywistym — przyp. red.

na bieżąco je przetwarzać. Podobne działanie do czujników posiadają odbiorniki systemu GPS. W rozdziale 17. pokazaliśmy, jak skonfigurować obiekty nasłuchujące wobec aktualizacji położenia uzyskiwanych z systemu GPS, tak aby aktualizacje położenia od razu były przetwarzane na informacje zrozumiałe dla użytkownika. Mimo że system GPS przypomina czujniki, nie stanowi on jednak części omawianej w tym rozdziale architektury.

Wśród dostępnych rodzajów czujników będziemy mieli do czynienia z takimi, jak:

- czujnik oświetlenia,
- czujnik zbliżeniowy,
- termometr,
- czujnik ciśnienia,
- żyroskop,
- akcelerometr,
- magnetometr,
- czujnik orientacji położenia,
- czujnik grawitacji (Android 2.3),
- czujnik przyśpieszenia liniowego (Android 2.3),
- czujnik wektora rotacji (Android 2.3),
- czujnik NFC (ang. *Near Field Communication* — komunikacja bliskiego pola; Android 2.3).

Czujnik NFC różni się od pozostałych. Zajmiemy się nim oddzielnie w dalszej części rozdziału, ponieważ uzyskujemy do niego dostęp w zupełnie inny sposób niż w przypadku pozostałych rodzajów czujników.

## Wykrywanie czujników

Nie powinniśmy jednak zakładać, że każde urządzenie będzie wyposażone we wszystkie wymienione rodzaje czujników. W rzeczywistości wiele urządzeń zostało wyposażonych tylko w część tych układów. Na przykład emulator Androida posiada wyłącznie akcelerometr. W jaki sposób możemy więc dowiedzieć się, które czujniki są dostępne w urządzeniu? Istnieją dwa sposoby — jeden bezpośredni, drugi pośredni.

Pierwszy sposób polega na zażądaniu od klasy `SensorManager` listy dostępnych czujników. W efekcie otrzymamy listę obiektów, dla których możemy ustawić obiekty nasłuchujące, a następnie pobrać z nich dane. Rozwiążanie to zostanie omówione w dalszej części rozdziału. Zakładamy tutaj, że aplikacja została już zainstalowana w urządzeniu; co jednak należy zrobić, w przypadku gdy urządzenie nie ma czujnika wymaganego przez nasz program?

W tym miejscu możemy wykorzystać drugie rozwiązanie. Możemy zdefiniować w pliku `AndroidManifest.xml` funkcje urządzenia niezbędne do działania aplikacji. Jeżeli nasz program wymaga obecności czujnika zbliżeniowego, możemy to określić w pliku manifeście za pomocą następującego wiersza:

```
<uses-feature android:name="android.hardware.sensor.proximity" />
```

W efekcie dana aplikacja zostanie zainstalowana wyłącznie w urządzeniu posiadającym czujnik zbliżeniowy, więc jej zainstalowanie będzie jednoznaczne z obecnością tego podzespołu.

## Jakie informacje możemy uzyskać na temat czujnika?

Chociaż stosowanie znaczników `uses-feature` w pliku manifeście gwarantuje nam wiedzę o obecności wymaganego czujnika w danym urządzeniu, to nie uzyskujemy za ich pomocą pełnych informacji na temat danego układu elektronicznego. Napiszemy więc prostą aplikację uzyskującą dane na temat czujnika. Na listingu 26.1 zostały zamieszczone układ graficzny aplikacji oraz kod Java klasy `MainActivity`.

### Uwaga!

Możemy pobrać wszystkie projekty omawiane w tym rozdziale. Na końcu rozdziału znajduje się adres URL witryny z utworzonymi projektami. W ten sposób będzie je można zimportować bezpośrednio do środowiska Eclipse.

**Listing 26.1.** Układ graficzny i kod Java aplikacji Lista czujników

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <ScrollView android:layout_width="fill_parent"
        android:layout_height="0dp"
        android:layout_weight="1" >
        <TextView android:id="@+id/text"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content" />
    </ScrollView>
</LinearLayout>

// Jest to plik MainActivity.java
import java.util.HashMap;
import java.util.List;
import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        TextView text = (TextView)findViewById(R.id.text);

        SensorManager mgr =
            (SensorManager) this.getSystemService(SENSOR_SERVICE);

        List<Sensor> sensors = mgr.getSensorList(Sensor.TYPE_ALL);

        StringBuilder message = new StringBuilder(2048);
        message.append("Czujniki znalezione w urządzeniu:\n");
    }
}
```

```
for(Sensor sensor : sensors) {  
    message.append(sensor.getName() + "\n");  
    message.append(" Typ: " +  
        sensorTypes.get(sensor.getType()) + "\n");  
    message.append(" Producent: " +  
        sensor.getVendor() + "\n");  
    message.append(" Wersja: " +  
        sensor.getVersion() + "\n");  
    message.append(" Rozdzielcość: " +  
        sensor.getResolution() + "\n");  
    message.append(" Maksymalny zasięg: " +  
        sensor.getMaximumRange() + "\n");  
    message.append(" Zasilanie: " +  
        sensor.getPower() + " mA\n");  
}  
text.setText(message);  
}  
  
private HashMap<Integer, String> sensorTypes =  
    new HashMap<Integer, String>();  
  
{  
    sensorTypes.put(Sensor.TYPE_ACCELEROMETER, "TYP_AKCELEROMETR");  
    sensorTypes.put(Sensor.TYPE_GYROSCOPE, "TYP_ŻYROSKOP");  
    sensorTypes.put(Sensor.TYPE_LIGHT, "TYP_OŚWIETLENIE");  
    sensorTypes.put(Sensor.TYPE_MAGNETIC_FIELD, "TYP_POLE_MAGNETYCZNE");  
    sensorTypes.put(Sensor.TYPE_ORIENTATION, "TYP_ORIENTACJA");  
    sensorTypes.put(Sensor.TYPE_PRESSURE, "TYP_CIĘNIENIE");  
    sensorTypes.put(Sensor.TYPE_PROXIMITY, "TYP_ODLEGŁOŚĆ");  
    sensorTypes.put(Sensor.TYPE_TEMPERATURE, "TYP_TEMPERATURA");  
    sensorTypes.put(Sensor.TYPE_GRAVITY, "TYP_GRAWITACJA");  
    sensorTypes.put(Sensor.TYPE_LINEAR_ACCELERATION,  
        "TYP_PRZYŚPIESZENIE_LINIOWE");  
    sensorTypes.put(Sensor.TYPE_ROTATION_VECTOR,  
        "TYP_WEKTOR_OBROTU");  
}  
}
```

---

Warto zwrócić uwagę, że w tej przykładowej aplikacji wprowadziliśmy kontrolkę `ScrollView`, ponieważ z dużym prawdopodobieństwem uzyskamy więcej wierszy, niż można by pomieścić na ekranie. Rozpoczynamy metodę `onCreate()` od uzyskania odniesienia do klasy `SensorManager`. Może istnieć tylko jedno takie odniesienie, więc odczytujemy je jako usługę systemową. Następnie wywołujemy metodę `getSensorList()`, aby uzyskać listę czujników. Zostają wyświetlane informacje dotyczące każdego dostępnego rodzaju czujnika. Na rysunku 26.1 widzimy rezultaty.

Powinniśmy wyjaśnić kilka spraw na temat uzyskanych informacji o czujnikach. Dzięki wartości `Typ` poznajemy podstawową kategorię czujnika, lecz nie uzyskujemy szczegółów na jego temat. Czujnik oświetlenia pozostaje czujnikiem oświetlenia, ale istnieją jego różne odmiany, w zależności od urządzenia. Na przykład rozdzielcość tego czujnika w jednym urządzeniu może być większa niż w innym. Gdy określamy potrzebę obecności czujnika w znaczniku `<uses-feature>`, nie potrafimy z góry określić, na jaką odmianę podzespołu natrafimy. Musimy więc wysłać do samego urządzenia zapytanie o dane czujnika i dostosować kod do otrzymanych wyników.



**Rysunek 26.1.** Wyniki uzyskane za pomocą aplikacji Lista czujników

Wartości uzyskiwane dla rozdzielczości i maksymalnego zasięgu będą podawane w jednostkach mierzonych przez dany czujnik. Wartość zasilania podawana jest w miliamperach (mA) i reprezentuje natężenie prądu pobieranego przez czujnik z baterii; im mniejsza wartość, tym lepiej.

Wiemy już teraz, jakie czujniki mamy do dyspozycji, zatem w jaki sposób możemy pobierać od nich dane? Jak już wcześniej wyjaśniliśmy, konfigurujemy obiekt nasłuchujący wobec przesyłanych danych. Przeanalizujmy teraz taką sytuację.

## Pobieranie zdarzeń generowanych przez czujniki

Czujniki zaczną przesyłać dane do aplikacji po zarejestrowaniu odczytującego je obiektu nasłuchującego. Gdy obiekt ten nie nasłuchuje, czujnik może zostać wyłączony, dzięki czemu oszczędzamy energię baterii, musimy więc się upewnić, że będziemy nasłuchiwać zdarzeń czujników jedynie wtedy, gdy będą potrzebne. Skonfigurowanie obiektu nasłuchującego nie jest w tym przypadku skomplikowaną czynnością. Przykładowo należy mierzyć poziom natężenia światła za pomocą czujnika oświetlenia. Na listingu 26.2 widzimy kod Java służący do obsługi tego zadania. Wykorzystamy w tym przykładzie układ graficzny zdefiniowany na listingu 26.1.

**Listing 26.2.** Kod Java aplikacji Monitor czujnika oświetlenia

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.TextView;
```

```
public class MainActivity extends Activity implements SensorEventListener {  
    private SensorManager mgr;  
    private Sensor light;  
    private TextView text;  
    private StringBuilder msg = new StringBuilder(2048);  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
  
        mgr = (SensorManager) this.getSystemService(SENSOR_SERVICE);  
  
        light = mgr.getDefaultSensor(Sensor.TYPE_LIGHT);  
  
        text = (TextView) findViewById(R.id.text);  
    }  
  
    @Override  
    protected void onResume() {  
        mgr.registerListener(this, light,  
            SensorManager.SENSOR_DELAY_NORMAL);  
        super.onResume();  
    }  
  
    @Override  
    protected void onPause() {  
        mgr.unregisterListener(this, light);  
        super.onPause();  
    }  
  
    public void onAccuracyChanged(Sensor sensor, int accuracy) {  
        msg.insert(0, sensor.getName() + " dokładność zmieniona: " +  
            accuracy + (accuracy==1?" (NISKA)":(accuracy==2?" (ŚREDNIA)":  
                "(WYSOKA)") + "\n");  
        text.setText(msg);  
        text.invalidate();  
    }  
  
    public void onSensorChanged(SensorEvent event) {  
        msg.insert(0, "Otrzymano zdarzenie czujnika: " + event.values[0] +  
            " jednostek SI: luks\n");  
        text.setText(msg);  
        text.invalidate();  
    }  
}
```

---

W tej przykładowej aplikacji ponownie uzyskujemy odniesienie do klasy `SensorManager`, jednak tym razem otrzymujemy informacje wyłącznie na temat czujnika oświetlenia. Następnie w metodzie `onResume()` naszej aktywności konfigurujemy obiekt nasłuchujący, a w metodzie `onPause()` następuje jego wyrejestrowanie. Nie chcemy zajmować się poziomami oświetlenia, gdy aplikacja nie znajduje się na pierwszym planie.

Wewnątrz metody `registerListener()` przekazywana jest wartość parametru określającego częstotliwość odświeżania odczytów czujnika. Parametr ten może przyjmować następujące wartości:

- `SENSOR_DELAY_NORMAL`,
- `SENSOR_DELAY_UI`,
- `SENSOR_DELAY_GAME`,
- `SENSOR_DELAY_FASTEST`.

Bardzo ważny jest wybór odpowiedniej wartości odświeżania. Niektóre czujniki są bardzo czułe i będą generować dużą liczbę wyników w krótkim czasie. Jeżeli wybierzemy wartość `SENSOR_DELAY_FASTEST`, możemy nawet przekroczyć możliwości ich przetwarzania przez aplikację. W zależności od operacji przeprowadzanych na odczytach czujnika istnieje możliwość, że pamięć urządzenia zostanie zapełniona w takim stopniu, iż proces jej oczyszczania okaże się zbyt mało wydajny, co będzie skutkowało widocznym zwolnieniem, a nawet przestojami w działaniu urządzenia. Z drugiej strony niektóre czujniki wymagają maksymalnej częstotliwości odczytów; dotyczy to szczególnie czujnika wektora obrotu.

Ponieważ w naszej aktywności zaimplementowaliśmy interfejs `SensorEventListener`, posiadamy dwie metody zwrotne związane ze zdarzeniami czujnika: `onAccuracyChanged()` oraz `onSensorChanged()`. Pierwsza metoda będzie nas informowała o zmianie dokładności czujnika (lub czujników, gdyż można ją wywołać wobec większej liczby tych układów elektronicznych). Wartość tego parametru może wynosić 0, 1, 2 lub 3, co oznacza dokładność, odpowiednio: nieniarodajną, niską, średnią lub wysoką. Niemiarodajna dokładność nie musi wcale oznaczać, że urządzenie jest popsułe; w ten sposób zazwyczaj wskazuje się potrzebę kalibracji czujnika. Druga metoda określa moment zmiany poziomu natężenia światła, dzięki czemu otrzymujemy zdarzenie zawierające informacje o nowej wartości (wartościach) pochodzącej z czujnika.

Obiekt `SensorEvent` zawiera kilka elementów, z których jeden stanowi tablicę wartości zmienoprzecinkowych. W przypadku czujnika oświetlenia tylko pierwsza wartość zmienoprzecinkowa ma znaczenie. Jest to wyrażona w luksach wartość natężenia oświetlenia zarejestrowana przez urządzenie. W naszej przykładowej aplikacji tworzymy wiadomości poprzez nakładanie nowych komunikatów na stare, a następnie ich wyświetlanie w kontrolce `TextView`. Najnowsze odczyty będą zawsze wyświetlane na górze ekranu.

Po uruchomieniu aplikacji (oczywiście w fizycznym urządzeniu, gdyż emulator nie posiada czujnika oświetlenia) zauważymy, że na początku nie będzie wyświetlany żaden wynik. Wystarczy jednak zmienić źródło światła padające na lewą górną część urządzenia. Najprawdopodobniej właśnie tam jest umieszczony czujnik oświetlenia. Jeżeli przyjrzymy się temu miejscu bardzo uważnie, zauważymy ciemniejszy punkt pod wyświetlaczem. To jest właśnie czujnik oświetlenia. Jeżeli zakryjemy go palcem, wartość natężenia światła prawdopodobnie bardzo się zmniejszy (chociaż może nie osiągnąć wartości 0). Powinny zacząć się pojawiać komunikaty informujące o zmianie poziomu oświetlenia.

#### Uwaga!

W momencie zakrycia czujnika oświetlenia możemy zaobserwować również włączenie podświetlenia klawiatury (jeżeli urządzenie jest wyposażone w taką funkcję). Wynika to z faktu, że system wykrył zmniejszoną ekspozycję na światło i podświetla klawisze, aby ułatwić korzystanie z telefonu.

## Problemy pojawiające się podczas uzyskiwania danych z czujników

Architektura czujników w Androidzie wiąże się z pewnymi problemami, których powinniśmy być świadomi. Ten aspekt obsługi czujników wcale nie jest prosty. W pewnych przypadkach problemy są bardzo łatwe do rozwiązań, w innych jest to niemożliwe lub bardzo trudne do osiągnięcia.

### Metoda `onAccuracyChanged()` zawsze zwraca tę samą wartość

Aż do wersji 2.2 Androida metoda zwrotna `onAccuracyChanged()` za każdym razem była wywoływana w momencie pojawienia się nowego odczytu i parametr dokładności przybierał wartość 3 (wysoka dokładność). Dobra jest dostosować zmiany dokładności danych czujnika, jednak niekiedy metoda ta będzie wywoływana za każdym razem, nawet jeżeli dokładność odczytu nie uległa zmianie.

### Brak bezpośredniego dostępu do wartości czujnika

Prawdopodobnie Czytelnik zdążył już zauważyc, że nie ma bezpośredniego sposobu na uzyskiwanie bieżącej wartości odczytu z czujnika. Jedynym rozwiązaniem jest implementacja obiektu nasłuchującego. Oznacza to, że nawet po ustawieniu tego obiektu nie mamy pewności, czy uzyskamy określone dane w interesującym nas przedziale czasowym. Dobrą stroną sytuacji jest to, że metoda zwrotna jest asynchroniczna i nie będzie blokowała nam głównego wątku interfejsu użytkownika w oczekiwaniu na dane z czujnika. Mimo to projektując aplikację, należy ją przygotować na to, iż dane z czujnika nie będą dostępne w zakładanym przez nas momencie.

Możliwe jest uzyskanie bezpośredniego dostępu do tych danych za pomocą natywnego kodu, a także funkcji JNI<sup>2</sup> Androida. W tym celu trzeba znać niskopoziomowe, natywne wywołania interfejsu sterownika interesującego nas czujnika, a ponadto wiedzieć, w jaki sposób skierować interfejs ponownie do systemu. Jest to więc możliwe do zrealizowania, ale niełatwwe zadanie.

### Wartości czujnika są zbyt wolno wysyłane

Nawet po wstawieniu wartości `SENSOR_DELAY_FASTEST` nie będziemy otrzymywać nowych wartości częściej niż co 20 ms (w zależności od urządzenia). Jeżeli potrzebujemy większej częstotliwości odświeżania danych, niż może nam zapewnić rzadko stosowane ustawienie `SENSOR_DELAY_FASTEST`, możemy — podobnie jak w poprzednim przypadku — wykorzystać natywny kod oraz interfejs JNI, będzie to jednak skomplikowane rozwiązanie.

### W Androidzie 2.1 czujniki wyłączają się wraz z ekranem

W wersji 2.1 Androida stwierdzono problemy z aktualizacjami odczytów czujników, spowodowane ich wyłączeniem w momencie wyłączenia wyświetlacza. Najwidoczniej ktoś uznał, że świetnym rozwiązaniem będzie uniemożliwienie wysyłania odczytów czujnika przy wyłączonym ekranie, nawet w przypadku ustawienia blokady przechodzenia urządzenia w stan uśpienia przez aplikację (która najprawdopodobniej korzysta z usługi). Główną przyczyną problemu jest wyrejestrowanie obiektu nasłuchującego w momencie wyłączenia ekranu. Istnieje kilka rozwiązań

<sup>2</sup> Technologia JNI (ang. *Java Native Interface*) stanowi mechanizm pozwalający na uruchamianie kodu napisanego w języku natywnym (np. C/C++) wewnętrz środowiska Java — przyp. tłum.

tego problemu. Przede wszystkim możemy skonfigurować czas wygaśnięcia wyświetlacza, dzięki czemu nie zostanie on wyłączony podczas otrzymywania aktualizacji odczytów. Zasadniczą wadą tego rozwiązania jest wysoki koszt energetyczny, co wiąże się z szybszym rozładowywaniem baterii. Aby ustawić czas wygaszania wyświetlacza, musimy wprowadzić fragment podobny do ukazanego poniżej, gdzie `myDelay` oznacza czas wyrażony w milisekundach:

```
Settings.System.putInt(getApplicationContext(),
    Settings.System.SCREEN_OFF_TIMEOUT, myDelay);
```

Wprowadzenie wartości -1 sprawi, że ekran nigdy nie zostanie wyłączony. Aplikacja będzie wymagała również przydzielenia odpowiedniego uprawnienia (`android.permission.WRITE_SETTINGS`) w pliku *AndroidManifest.xml*. Drugą wadą tego rozwiązania jest to, że ustawienie czasu wygaszania wyświetlacza ma charakter globalny. Jeśli jakaś aplikacja zmieni tę wartość, Android zmodyfikuje ją wszędzie. W rzeczywistości więc aplikacja powinna zapamiętać poprzednią wartość tego ustawienia i przywrócić ją w momencie końca pracy. Jednak nawet po wdrożeniu takiego rozwiązania możemy natrafić na problemy, ponieważ użytkownik po uruchomieniu aplikacji może się zastanawiać, dlaczego wyświetlacz nie zostaje wyłączony po pewnym czasie, przejść do panelu ustawień i wprowadzić zupełnie inną wartość wygaszania ekranu, następnie powrócić do aplikacji i dopiero wtedy ją zamknąć. Nie musimy wspominać, że po takich modyfikacjach ustawień ekran może zostać wyłączony w czasie pracy aplikacji, w efekcie czego przestanie ona otrzymywać odczyty czujnika.

### **Technika wyrejestrowania i zarejestrowania dotycząca ciągłych aktualizacji odczytów czujnika**

Jedna z metod zapewnienia aplikacji otrzymywania nieprzerwanego strumienia odczytów polega na zarejestrowaniu odbiorcy powiadomień związanych z wyłączeniem ekranu, następnie wyrejestrowaniu obiektu nasłuchującego zdarzenia czujnika i jego ponownym zarejestrowaniem w metodzie `onReceive()` klasy `BroadcastReceiver`. Rozwiązanie to okazało się skuteczne w niektórych urządzeniach pracujących pod kontrolą Androida w wersji 2.1, ale nie we wszystkich.

Ponieważ nasza aplikacja w normalnych warunkach zostaje zatrzymana w momencie wyłączenia wyświetlacza, najpierw musimy ustawić częściową blokadę przechodzenia urządzenia w stan wstrzymania (ang. *wake lock*), aby program pozostawał aktywny po wygaszeniu ekranu. W naszym przykładzie wykorzystujemy aktywność, jednak w rzeczywistej aplikacji najprawdopodobniej umieścilibyśmy kod obiektu nasłuchującego wewnętrz usługi. Na listingu 26.3 pokazaliśmy przykładową implementację tego pomysłu w postaci aktywności.

**Listing 26.3.** Rozwiązywanie problemów za pomocą wyłączenia obiektów `SensorListener`

---

```
package com.androidbook.sensor.accel;

// Jest to plik MainActivity.java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import android.app.Activity;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
```

```
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.os.Environment;
import android.os.PowerManager;
import android.os.PowerManager.WakeLock;
import android.provider.Settings;
import android.util.Log;

public class MainActivity extends Activity implements SensorEventListener {
    private static final String TAG = "AccelerometerRecordToFile";
    private WakeLock mWakelock = null;
    private SensorManager mMgr;
    private Sensor mAccel;
    private BufferedWriter mLog;
    final private SimpleDateFormat mTimeFormat =
        new SimpleDateFormat("HH:mm:ss - ");
    private int mSavedTimeout;
*
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mMgr = (SensorManager) this.getSystemService(SENSOR_SERVICE);

        mAccel = mMgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

        // Konfiguruje plik dziennika, w którym będą zapisywane informacje. Dodamy go,
        // w przypadku gdy nasza aktywność zostanie uruchomiona ponownie w środku
        // eksperymentu.
        try {
            String filename =
Environment.getExternalStorageDirectory().getAbsolutePath() +
            "/accel.log";
            mLog = new BufferedWriter(new FileWriter(filename, true));
        }
        catch(Exception e) {
            Log.e(TAG, "Nie mozna zainicjalizowac pliku dziennika");
            e.printStackTrace();
            finish();
        }

        PowerManager pwrMgr =
            (PowerManager) this.getSystemService(POWER_SERVICE);
        mWakelock = pwrMgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                                         "Accel");
        mWakelock.acquire();

        // Zapisuje bieżącą wartość czasu wygaszenia ekranu, a następnie
        // zmniejsza ją
        try {
            mSavedTimeout = Settings.System.getInt(getContentResolver(),
Settings.System.SCREEN_OFF_TIMEOUT);
```

```
}

catch(Exception e) {
    mSavedTimeout = 120000; //Domyślna wartość wynosi 2 minuty,
                           //jeśli nie możemy odczytać wartości bieżącej
}

Settings.System.putInt(getApplicationContext(),
    Settings.System.SCREEN_OFF_TIMEOUT, 5000); //5 sekund

}

public BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_SCREEN_OFF.equals(intent.getAction())) {
            writeLog("Ekran został wyłączony");
            // Wyrejestruje obiekt nasłuchujący i zarejestruje go ponownie.
            // Powinno być to konieczne wyłącznie w wersji 2.1 Androida, chociaż
            // nie zaszkodzi wstawić to rozwiązań również dla innych wersji systemu.

            mMgr.unregisterListener(MainActivity.this);
            mMgr.registerListener(MainActivity.this, mAccel,
                SensorManager.SENSOR_DELAY_NORMAL);

        }
    }
};

@Override
protected void onStart() {
    writeLog("rozpoczynanie...");
    mMgr.registerListener(this, mAccel,
        SensorManager.SENSOR_DELAY_NORMAL);

    IntentFilter filter = new IntentFilter(Intent.ACTION_SCREEN_OFF);
    registerReceiver(mReceiver, filter);

    super.onStart();
}

@Override
protected void onStop() {
    writeLog("zatrzymywanie...");
    mMgr.unregisterListener(this, mAccel);
    unregisterReceiver(mReceiver);
    try {
        mLog.flush();
    } catch (IOException e) {
        //Ignoruje wszelkie błędy występujące w pliku dziennika
    }
    super.onStop();
}

@Override
protected void onDestroy() {
    writeLog("zamykanie...");
    try {
        mLog.flush();
```

```

        mLog.close();
    }
    catch(Exception e) {
        // Ignoruje wszelkie błędy występujące w pliku dziennika
    }

    // Odczytuje pierwotną wartość czasu wygaśnięcia ekranu
    Settings.System.putInt(getContentResolver(),
        Settings.System.SCREEN_OFF_TIMEOUT, mSavedTimeout);

    mWakelock.release();

    super.onDestroy();
}

public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Ignoruje
}

public void onSensorChanged(SensorEvent event) {
    writeLog("Uzyskano zdarzenie czujnika: " + event.values[0] + ", " +
        event.values[1] + ", " + event.values[2]);
}

private void writeLog(String str) {
    try {
        Date now = new Date();
        mLog.write(mTimeFormat.format(now));
        mLog.write(str);
        mLog.write("\n");
    }
    catch(IOException ioe) {
        ioe.printStackTrace();
    }
}
}

```

Nie musimy się przejmować układem graficznym, gdyż ta przykładowa aplikacja będzie wyświetlała jedynie swoją nazwę. Nie interesują nas również uprawnienia, zatem na listingu 26.4 prezentujemy plik *AndroidManifest.xml*.

**Listing 26.4.** Plik *AndroidManifest.xml* aplikacji Monitor akcelerometru

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0" package="com.androidbook.sensor.accel">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">
        <activity android:name=".MainActivity"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>

```

```

</activity>
</application>
<uses-sdk android:minSdkVersion="3" />

<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_SETTINGS" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
</manifest>

```

Głównym zadaniem tego przykładowego kodu jest zapisywanie zdarzeń akcelerometru w pliku dziennika. Musimy umieścić częściową blokadę przechodzenia urządzenia w stan uśpienia wewnętrz metodę `onCreate()`, dzięki czemu aplikacja nie będzie wstrzymywana w momencie wyłączenia ekranu (blokady przechodzenia urządzenia w stan uśpienia zostały omówione w rozdziale 14.). Ustanawiamy również czas wygaśnięcia ekranu na 5 sekund, czyli wyświetlacz zostanie wyłączony dość szybko, jednak jego poprzednia wartość zostaje zapamiętana i będzie przywrócona w momencie wywołania metody `onDestroy()`.

**Uwaga!**

Modyfikujemy czas wygaśnięcia ekranu jedynie w celu sprawdzenia, co się stanie z obiektem nasłuchującym zdarzenia czujnika. W rzeczywistej aplikacji użytkowej nie stosowalibyśmy tego rozwiązania.

Konfigurujemy również odbiorcę `BroadcastReceiver`, który zostaje powiadomiony o wyłączeniu wyświetlacza. Zapisujemy tę informację w metodzie `onReceive()`, a następnie wprowadzamy omawiane obejście dla wersji 2.1 Androida, pozwalające na dalsze otrzymywanie aktualizacji odczytów z czujnika. W metodzie `onStart()` zostaje zarejestrowany obiekt nasłuchujący zdarzenia czujnika, a także odbiorca komunikatów. Obydwa te obiekty zostają wyrejestrowane w metodzie `onStop()`. Korzystamy z metod `onStart()` i `onStop()` zamiast `onResume()` i `onPause()`, ponieważ chcemy nasłuchiwać zdarzenia czujnika nawet po przejściu użytkownika do innej aktywności w trakcie działania programu.

Metoda `onDestroy()` zapewni oczyszczenie, opróżnienie i zamknięcie pliku dziennika. W przeciwnieństwie do poprzedniego przykładu, metoda `onAccuracyChanged()` nie wykonuje żadnej czynności. Dane zdarzeń są zapisywane do pliku dziennika w metodzie `onSensorChanged()`.

Taki wzór pracy z obiektem nasłuchującym zdarzenia czujnika powinniśmy stosować również w zwykłej, użytkowej aplikacji. Inaczej niż w przypadku poprzedniego przykładu, gdzie nie przejmowaliśmy się stanem wstrzymania urządzenia, najprawdopodobniej będziemy musieli wprowadzić blokadę przechodzenia urządzenia w stan wstrzymania, gwarantującą pobieranie zdarzeń przez aplikację nawet po wyłączeniu ekranu. Pamiętajmy, że jeżeli naszym systemem docelowym będzie Android 2.2 lub nowszy, nie powinniśmy się przejmować procesami rejestrowania i wyrejestrowywania obiektu nasłuchującego w odbiorcy `BroadcastReceiver`. Również Android w wersji 2.0 i starszych nie powinien sprawiać problemów.

Aplikacja, którą pokazaliśmy, jest bardzo ciekawa. Proponujemy, aby przetestować ją w następujący sposób:

1. Zainstaluj aplikację, następnie odłącz urządzenie od stacji roboczej, aby nie było ono połączone z kablem USB (czasami z tego powodu ekran pozostaje cały czas włączony pomimo wprowadzonych ustawień).

Kiedy aplikacja się uruchomi, możesz poruszać w przestrzeni urządzeniem, a po pięciu sekundach wyświetlacz zostanie wyłączony.

2. Poruszaj dalej urządzeniem, aby uruchamiać zdarzenia akcelerometru, a następnie po kilku sekundach odblokuj urządzenie i wciśnij przycisk cofania, aby zakończyć działanie aplikacji.

W głównym katalogu karty SD urządzenia znajdziemy plik dziennika, nazwany *accel.log*.

3. Podłącz ponownie urządzenie do stacji roboczej, następnie skopiuj plik *accel.log* i przejrzyj go.

Powinien być widoczny początkowy komunikat, wiele komunikatów o zdarzeniach, a następnie informacja *Ekran został wyłączone*. W zależności od urządzenia oraz wersji systemu tuż po nim mogą się pojawić kolejne komunikaty lub przerwa w napływie komunikatów do momentu odblokowania telefonu i zamknięcia aplikacji.

### **Technika pozostawiania włączonego ekranu przy ciągłych aktualizacjach odczytów czujnika**

Istnieje jeszcze jedno obejście problemu dotyczącego wersji 2.1 Androida. Przypomnijmy, że podstawowy problem polega na tym, że w niektórych urządzeniach czujniki zostają po prostu wyłączone w momencie wygaszenia ekranu. Rozwiążaniem zatem może być pozostawienie włączonego wyświetlacza. Na listingu 26.5 widzimy alternatywną wersję utworzonego w poprzednim przykładzie odbiorcy BroadcastReceiver. Różnica polega na tym, że tym razem w chwili wyłączenia wyświetlacza urządzenia ekran pozostanie włączony (chociaż przyciemniony). Dokonaliśmy także kilka mniejszych modyfikacji kodu. Czujnik będzie wysyłał odczyty do aplikacji nawet wtedy, gdy ekran będzie pozostawał przyciemniony. Jeżeli Czytelnik zamierza zaimportować gotowy projekt, nosi on nazwę *AccelerometerRecordToFileAlwaysOn*.

---

**Listing 26.5.** Pozostawienie włączonego wyświetlacza, nawet jeśli zostanie wyłączony przez użytkownika

---

```
// Dodajmy te obiekty do aktywności
private PowerManager mPwrMgr;
private WakeLock mTurnBackOn = null;
private Handler handler = new Handler();

// Dodajmy poniższe trzy wiersze do metody onCreate()
mPwrMgr = (PowerManager) this.getSystemService(POWER_SERVICE);
mWakeLock = mPwrMgr.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "Accel");
mWakeLock.acquire();

// Poniższym fragmentem zastępujemy klasę BroadcastReceiver w pliku MainActivity.java
public BroadcastReceiver mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        if (Intent.ACTION_SCREEN_OFF.equals(intent.getAction())) {
            writeLog("Ekran został wyłączony");
            // Z poziomu głównego wątku uruchamiamy ponownie ekran
            handler.post(new Runnable() {
                public void run() {
                    if(mTurnBackOn != null)
                        mTurnBackOn.release();
                }
            });
            mTurnBackOn = mPwrMgr.newWakeLock(
                PowerManager.SCREEN_DIM_WAKE_LOCK |
                PowerManager.ACQUIRE_CAUSES_WAKEUP,
                "AccelOn");
        }
    }
}
```

```

        mTurnBackOn.acquire();
    });
}

};

// Nie zapomnijmy dodać poniższego fragmentu w metodzie onDestroy()
if(mTurnBackOn != null)
    mTurnBackOn.release();

```

Jeżeli teraz po uruchomieniu aplikacji wciśniemy przycisk zasilania urządzenia w celu wyłączenia ekranu, wykryje ona to zdarzenie i za pomocą blokady przechodzenia w stan uśpienia ustawi wyświetlacz w trybie oszczędzania energii. W trakcie przeprowadzania tej czynności może się pojawić krótka przerwa pomiędzy kolejnymi odczytami czujnika, jednak jest to lepsze rozwiązanie od braku aktualizacji odczytów w czasie, gdy wyświetlacz pozostaje wyłączony. Zwrócmy uwagę, że wykorzystaliśmy procedurę obsługi do wystawienia obiektu Runnable z poziomu odbiorcy komunikatów. Dzięki temu nasz kod będzie przetwarzany w głównym wątku, co staje się istotne w momencie zwalniania blokady przechodzenia urządzenia w stan uśpienia w metodzie `onDestroy()`. Zakładanie i zwalnianie blokady muszą być przeprowadzane w tym samym wątku.

Zauważmy również, że w metodzie `onReceive()` odbiorcy komunikatów zwalniamy blokadę przechodzenia urządzenia w stan uśpienia, zanim pojawi się kolejna. Dokonujemy tego na wypadek kilkukrotnego wciśnięcia przycisku zasilania w trakcie rejestrowania odczytów czujnika. Liczba zwalnianych blokad musi odpowiadać liczbie zakładanych blokad, więc zwalniamy tę jedną jeszcze przed wystąpieniem następnej.

Skoro już wyjaśniliśmy, w jaki sposób należy pobierać dane z czujników, należy powiedzieć, co możemy z nimi zrobić. Jak już wcześniej stwierdziliśmy, w zależności od rodzaju czujnika wartości przekazywane w tablicy posiadają różnorodne znaczenie. W następnym podrozdziale zajmiemy się poszczególnymi typami czujników oraz znaczeniem generowanych przez nie odczytów.

## Interpretowanie danych czujnika

Potrafimy już uzyskiwać dane z czujników, czas więc zrobić z nimi coś sensownego. Musimy jednak pamiętać, że rodzaj otrzymywanych odczytów zależy od rodzaju czujnika. Niektóre czujniki są mniej skomplikowane od innych. W następnych punktach zajmiemy się opisem odczytów otrzymywanych z obecnie znanych nam rodzajów czujników. Wraz z nowymi generacjami urządzeń będą się z pewnością pojawiać również niespotykane do tej pory rodzaje czujników. Najprawdopodobniej sama architektura czujników w Androidzie pozostanie niezmieniona, więc omawiane w tym podrozdziale techniki powinny znaleźć zastosowanie także w przyszłych typach urządzeń.

## Czujniki oświetlenia

Czujnik oświetlenia stanowi jeden z najprostszych rodzajów tego typu układów elektronicznych — pokazaliśmy, jak go wykorzystać, w pierwszej przykładowej aplikacji omówionej w tym rozdziale. Czujnik generuje odczyty związane z wykrywanym poziomem natężenia światła. Wraz ze zmianą natężenia poziomu światła czujnik aktualizuje również swoje odczyty. Dane są wyrażane w luksach — luks jest jednostką natężenia światła w układzie SI. Aby poznać dokładną

definicję tej jednostki, możemy zająrzeć do podrozdziału „Odbońniki”, gdzie zostały umieszczone adresy URL zasobów zawierających bardziej szczegółowe informacje.

W przypadku tablicy wartości przechowywanej w obiekcie SensorEvent czujnik oświetlenia wykorzystuje tylko pierwszy element `values[0]`. Wartość ta jest zmiennoprzecinkowa i z technicznego punktu widzenia jej zakres sięga od 0 do maksymalnej wartości odbieranej przez dany czujnik. Piszymy „z technicznego punktu widzenia”, ponieważ w przypadku pomiarów prowadzonych w ciemności czujnik może wysyłać bardzo niewielkie wartości odczytów i w rzeczywistości nigdy nie przekazuje wartości równej 0.

Pamiętajmy również, że czujnik może przekazywać maksymalną wartość natężenia światła, która różni się dla poszczególnych rodzajów urządzeń. Z tego powodu definiowanie stałych związanych z oświetleniem w klasie SensorManager może się okazać nieprzydatne. Na przykład klasa ta zawiera stałą `LIGHT_SUNLIGHT_MAX`, której wartość zmiennoprzecinkowa wynosi 120 000. Kiedy jednak wcześniej wysyliśmy zapytanie do urządzenia, maksymalna wartość odczytu wynosiła 10 240, czyli zdecydowanie mniej od wartości tej stałej. Istnieje również inna stała, `LIGHT_SHADE`, o wartości 20 000, która przekracza maksymalną wartość osiąganą przez testowany czujnik. Musimy więc o tym pamiętać w trakcie pisania kodu wykorzystującego odczyty czujnika oświetlenia.

## Czujniki zbliżeniowe

Czujnik zbliżeniowy mierzy odległość dzielącą dany obiekt od urządzenia (w centymetrach) lub definiuje flagę określającą, czy dany obiekt jest blisko, czy daleko. Niektóre czujniki zbliżeniowe posiadają zakres wartości od 0 do maksimum wraz z wartościami pośrednimi, inne natomiast definiują wyłącznie wartości minimalną i maksymalną. Jeżeli maksymalny zakres czujnika jest równy jego rozdzielczości, to znaczy, że odczytuje on wyłącznie krańcowe wartości. Maksymalna wartość pewnych czujników wynosi 1.0, a innych — 6.0. Niestety, dopóki nie zainstalujemy i nie uruchomimy aplikacji, nie dowiemy się, jaki rodzaj czujnika został umieszczony w urządzeniu. Nawet jeśli umieścimy znacznik `<uses-feature>` w pliku `AndroidManifest.xml`, niewiele nam to pomoże. Nasza aplikacja powinna w elegancki sposób obsługiwać obydwa typy czujników zbliżeniowych, chyba że dokładniejsza wiedza o typie czujnika okaże się niezbędna.

Warto poznać interesujący szczegół dotyczący czujników zbliżeniowych: czasami czujnik ten jest częścią układu elektronicznego wspólnego z czujnikiem oświetlenia. Android jednak traktuje je jako logicznie oddzielne czujniki, jeśli więc będziemy potrzebować danych z obydwu rodzajów czujników, musimy utworzyć dla nich osobne obiekty nasłuchujące. Istnieje jeszcze jeden ciekawy fakt: w aplikacjach telefonicznych czujnik zbliżeniowy często jest stosowany do określania odległości głowy użytkownika od urządzenia. Jeżeli głowa znajdzie się wystarczająco blisko ekranu dotykowego, zostaje on zablokowany, aby żaden przycisk nie został przypadkowo wcisnięty przez ucho lub policzek podczas rozmowy.

Wśród kodów źródłowych utworzonych na potrzeby tego rozdziału znajdziemy prostą aplikację monitorującą czujnik zbliżeniowy, która w rzeczywistości jest zmodyfikowaną wersją monitora czujnika oświetlenia. W samej książce nie umieścimy tego kodu, jednak nic nie stoi na przeszkodzie, żeby samodzielnie go pobrać i z nim poeksperymentować.

## Termometry

Termometr przekazuje wyniki odczytu temperatury i również są to pojedyncze elementy w tablicy `values[0]`. Wartości są reprezentowane w stopniach Celsjusza. Wartości w skali Fahrenheita uzyskamy, mnożąc wartość wyrażoną w stopniach Celsjusza przez 9/5 i dodając do wyniku 32.

Na przykład 0 stopni Celsjusza (punkt zamarzania wody) w skali Fahrenheita przyjmuje wartość 32, natomiast 100 stopni Celsjusza (punkt wrzenia wody) to 212 stopni Fahrenheita.

W zależności od urządzenia termometr może zostać umieszczony w różnych miejscach, istnieje więc możliwość, że wynik mierzonej temperatury zakłóca ciepło generowane przez telefon. Przykładowo odczyty temperatury w pewnych urządzeniach są zakłócone przez ciepło powstające podczas pracy baterii. Powinniśmy o tym pamiętać podczas pisania aplikacji wykorzystujących termometry. Nie należy oczekwać, że termometr wbudowany w telefon będzie mierzył wyłącznie temperaturę powietrza otaczającego aparat.

Wśród projektów utworzonych na potrzeby tego rozdziału Czytelnik znajdzie jeden ukazujący sposób korzystania z termometru, zatytułowany `TemperatureSensor`.

## Czujniki ciśnienia

Co ciekawe, w czasie, gdy pisaliśmy niniejszą książkę, ten rodzaj czujników nie został jeszcze umieszczony w żadnym urządzeniu. Uważamy jednak, że kolejne generacje urządzeń mogą zostać wyposażone w barometryczne czujniki ciśnienia, pozwalające na przykład na pomiar wysokości. Nie należy mylić tego czujnika z funkcją ekranu dotykowego, który reaguje na nacisk palca, może określić jego siłę i generuje obiekt `MotionEvent`. Wykrywanie tego rodzaju oddziaływań mechanicznych zostało omówione w rozdziale 25., a służąca do tego architektura nie jest częścią omawianej w tym rozdziale struktury czujników.

Chociaż utworzenie aplikacji obsługujących czujniki ciśnienia przez proste skopiowanie i zmodyfikowanie zaprezentowanych do tej pory aplikacji monitorujących nie byłoby trudnym zadaniem, to jednak bez wiedzy, jakie jednostki będą stosowane w przypadku tych czujników, napisanie takiej aplikacji nie zdałoby się na wiele. Najwidoczniej programiści z firmy Google planują z wyprzedzeniem.

## Żyroskopy

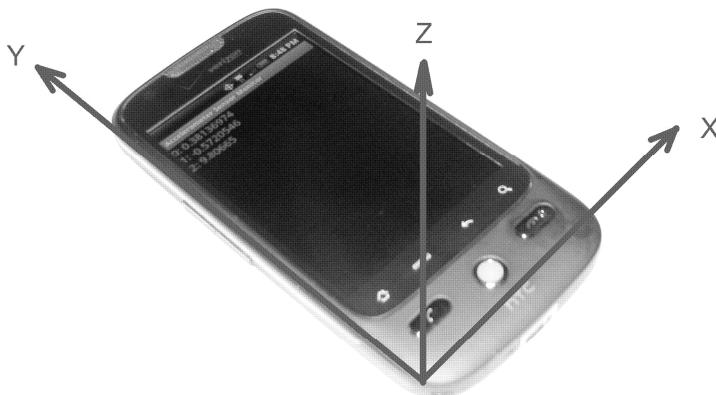
Żyroskopy stanowią bardzo ciekawą kategorię urządzeń, mierzącą skręt urządzenia w płaszczyźnie odniesienia. Inaczej mówiąc, za ich pomocą mierzymy prędkość obrotu telefonu w danej osi. Jeśli urządzenie nie będzie obracane, wartości odczytywane przez czujnik będą wynosić 0. W momencie obrotu smartfonu w dowolnym kierunku pojawią się niezerowe odczyty. Sam żyroskop nie poda nam wszystkich wymaganych danych. Niestety, podczas pracy z żyroskopami zawsze wkradną się jakieś błędy. Jednak w sprężeniu z akcelerometrami możemy określić ścieżkę ruchu urządzenia. Do powiązania odczytów pochodzących z obydwu czujników mogą służyć filtry Kalmana. Akcelerometry nie są zbyt dokładne w krótkich odcinkach czasowych, z kolei żyroskopy tracą ją wraz z upływem czasu, więc ich powiązanie ze sobą może nam zapewnić całkiem niezłą dokładność przez cały czas. Filtry Kalmana są bardzo skomplikowane, ale istnieje alternatywa zwana filtrami komplementarnymi, które są łatwiejsze do implementacji i generują całkiem poprawne wyniki. Wspomniane tu koncepcje wykraczają poza zakres książki.

Żyroskop przekazuje trzy wartości w tablicy wartości, opisujące kolejno punkty na osiach  $x$ ,  $y$  i  $z$ . Jednostką przekazywanych wartości są radiany na sekundę, reprezentują one szybkość obrotu urządzenia wokół danej osi. Jednym ze sposobów ich wykorzystania jest ich całkowanie po czasie w celu obliczenia zmiany kąta. W podobny sposób jest całkowana wartość prędkości liniowej po czasie w celu obliczenia odległości.

## Akcelerometry

Akcelerometry stanowią chyba najciekawsze z obecnie dostępnych rodzajów czujników. Za ich pomocą aplikacja może określić fizyczne ułożenie urządzenia w zależności od siły ciążenia, a dodatkowo wykrywać siły przesuwające to urządzenie. Dzięki takim informacjom programiści zyskują niespotykane dotąd możliwości, począwszy od nowej jakości sterowania w grach, a skończywszy na pracy w rzeczywistości rozszerzonej. Oczywiście, podstawowym zadaniem akcelerometru jest przekazanie do urządzenia informacji o zmianie jego ułożenia z orientacji pionowej na poziomą, i odwrotnie.

System współrzędnych w akcelerometrze działa następująco: oś *x* czujnika ma swój początek w lewym dolnym rogu urządzenia i jest skierowana w prawą stronę (patrząc od przodu urządzenia). Oś *y* również ma początek w lewym dolnym rogu urządzenia i jest skierowana w górę telefonu. Także punkt 0 osi *z* znajduje się w lewym dolnym rogu urządzenia i jest skierowany na zewnątrz, w taki sposób, że oddala się od urządzenia. Zostało to zobrazowane na rysunku 26.2.



Rysunek 26.2. System współrzędnych akcelerometru

System współrzędnych różni się od wykorzystywanego w układach graficznych i grafice dwuwymiarowej. W przypadku tamtych układów współrzędnych ich początek (0, 0) znajduje się w lewym górnym rogu ekranu, a wartości dodatnie osi *y* rosną w kierunku dolnym. Łatwo się pomylić podczas pracy z systemami współrzędnych w różnych układach odniesienia, należy więc zachować ostrożność.

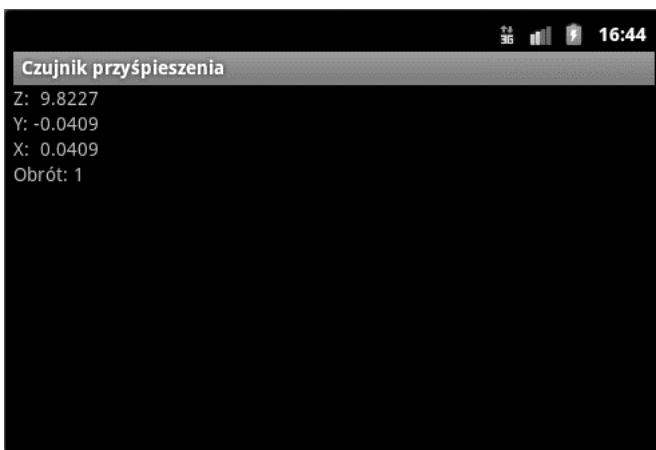
Jeszcze nic nie wspomnieliśmy o znaczeniu wartości przekazywanych przez akcelerometr, a więc co one oznaczają? Przyspieszenie jest mierzone w metrach na sekundę do kwadratu ( $\text{m/s}^2$ ). Przyspieszenie powodowane ziemską grawitacją wynosi  $9,81 \text{ m/s}^2$  i jest skierowane w dół, w stronę środka planety. Z punktu widzenia akcelerometru wartość siły ciążenia wynosi  $-9,81$ . Jeżeli urządzenie znajduje się w stanie spoczynku (nie porusza się) i jest ułożone na doskonale płaskiej, poziomej powierzchni, odczyty na osiach *x* i *y* przyjmą wartości 0, natomiast w osi *z*  $-+9,81$ . W rzeczywistości, zależnie od czułości i dokładności akcelerometru, wartości te nie będą doskonale odwzorowywać faktycznego stanu rzeczy, jednak będą stanowić wystarczająco dobre przybliżenie. W stanie spoczynku jedynie grawitacja będzie wpływać na urządzenie, a ponieważ jej wektor jest skierowany w dół (nasze urządzenie zaś leży płasko), nie będzie miała wpływu na osie *x* i *y*. W przypadku osi *z* będzie mierzona wartość siły działającej na urządzenie

oraz zostanie odjęta wartość siły ciążenia, więc 0 minus  $-9,81$  daje nam ostatecznie wartość  $+9,81$  — i tyle właśnie wynosi wartość siły przyłożonej do tej osi (element `values[2]` w obiekcie `SensorEvent`).

Wartości przesypane do aplikacji przez akcelerometr zawsze stanowią sumę sił działających na urządzenie minus wartość przyśpieszenia ziemskiego. Gdybyśmy unieśli w górę nasze ułożone doskonale płasko urządzenie, początkowo wartość mierzona dla osi *z* wzrosłaby, ponieważ zwiększylibyśmy oddziaływanie wbrew sile grawitacji. Gdy tylko przestaniemy unosić urządzenie, wartość sumaryczna działających sił powróci do wartości grawitacji. Gdyby urządzenie zostało upuszczone (czysto hipotetycznie — nie sprawdzajmy tego), zaczęłoby uzyskiwać przyśpieszenie w kierunku ziemi, a tym samym wartość odczytu w następnych momentach zmalałaby do zera.

Wyobraźmy sobie, że urządzenie z rysunku 26.2 obróćmy w taki sposób, aby było ułożone w trybie portretowym, pionowo. Oś *x* pozostaje bez zmian i wskazuje z lewej strony na prawą. Z kolei oś *y* zostaje ustawiona prostopadle do ziemi, a oś *z* zostaje skierowana w naszą stronę. Wartość osi *y* wynosi teraz  $+9,81$ , a osi *x* i *z* — po 0.

Co się stanie, jeśli obróćmy urządzenie do ułożenia poziomego, w trybie krajobrazowym, i w dalszym ciągu będziemy trzymać je pionowo, tj. ekran będzie się znajdował na wprost twarzy? Nietrudno zgadnąć, że osie *y* i *z* przybiorą wartości 0, a w osi *x* będzie działać siła równa  $+9,81$ . Taka sytuacja została zilustrowana na rysunku 26.3.



**Rysunek 26.3.** Wartości akcelerometru w trybie krajobrazowym, urządzenie ustawione w pionie

Gdy urządzenie znajduje się w stanie spoczynku lub porusza się z jednostajną prędkością, akcelerometry mierzą wyłącznie wartość grawitacji. Dla każdej osi odczyty akcelerometru stanowią składowe grawitacji w tej osi. Zatem za pomocą obliczeń trygonometrycznych możemy określić kąty oraz ułożenie urządzenia względem kierunku działania siły ciążenia. Oznacza to, że możemy się dowiedzieć, czy urządzenie jest ułożone w trybie portretowym, krajobrazowym, czy jakimś pośrednim. W rzeczywistości system korzysta właśnie z tego rozwiązania w czasie rozpoznawania orientacji ułożenia (tryb krajobrazowy lub portretowy). Zwróćmy jednak uwagę, że akcelerometry nie określają ułożenia urządzenia w odniesieniu do północy magnetycznej. Do tego właśnie służy magnetometr, który zostanie omówiony w dalszej części rozdziału.

## Akcelerometry a tryb wyświetletania

Akcelerometry jako takie są układami elektronicznymi, trwale przymocowanymi do obudowy urządzenia — i z tego powodu mają określone ułożenie względem reszty architektury telefonu, niezmieniające się w trakcie jego obracania. Odczyty wysyłane w wyniku ruchu będą oczywiście ulegały zmianom, jednak układ współrzędnych akcelerometrów jest oparty na urządzeniu i nie będzie w żaden sposób modyfikowany. Z kolei układ współrzędny ekranu zmienia się w zależności od trybu wyświetlania. Faktycznie, w zależności od ułożenia ekranu tryb portretowy może być obrócony nawet o 180 stopni. Analogiczna sytuacja dotyczy również trybu krajobrazowego.

Gdy nasza aplikacja odczytuje dane akcelerometru i ma właściwie modyfikować interfejs użytkownika, musi znać wartość obrotu urządzenia, aby móc wprowadzić niezbędne poprawki. W trakcie zmiany trybu z portretowego na krajobrazowy układ współrzędnych ekranu zostaje obrócony zgodnie z układem współrzędnych akcelerometrów. Aby tak się stało, aplikacja musi wykorzystać metodę `Display.getRotation()`, która została wprowadzona w wersji 2.2 Androida. Przekazywana wartość jest zwykłą liczbą całkowitą, nie symbolizuje jednak rzeczywistej wartości kąta obrotu. Mamy tu do czynienia z jedną z czterech następujących stałych: `Surface.ROTATION_0`, `Surface.ROTATION_90`, `Surface.ROTATION_180` lub `Surface.ROTATION_270`. Przyjmują one wartości kolejno: 0, 1, 2, 3. Wartości te informują aplikację, jak bardzo urządzenie zostało obrócone w stosunku do standardowego ułożenia wyświetlacza. Ponieważ nie wszystkie urządzenia obsługujące system Android w domyśle są ułożone w trybie portretowym, nie możemy z góry zakładać, że stała odpowiedzialna za ten tryb to `ROTATION_0`.

Nie wszystkie urządzenia przekazują wszystkie cztery wartości. W telefonie HTC Droid Eris obsługującym wersję 2.1 Androida metoda `Display.getOrientation()` (poprzedniczka metody `Display.getRotation()`, obecnie uznanej za przestarzałą) wyświetla tylko wartości 0 i 1, i to tyle. W zwykłym trybie portretowym przekazywana wartość wynosi 0. Jeżeli obróćmy urządzenie o 90 stopni w stronę przeciwną do kierunku ruchu wskazówek zegara, układ graficzny zostanie zmieniony i metoda `Display.getOrientation()` powróci z wartością 1. Jeżeli w trybie portretowym obróćmy urządzenie o 90 stopni zgodnie z kierunkiem ruchu wskazówek zegara, ekran pozostanie w trybie portretowym, a my otrzymamy wartość 0 z metody `Display.getOrientation()`.

W telefonie Motorola Droid pracującym pod kontrolą Androida 2.2 metoda `Display.getRotation()` powraca z wartościami 0, 1 lub 3. Wartość 2 nie jest przekazywana, co oznacza, że urządzenie nie pracuje w odwróconym trybie portretowym. Jest jednak pewien rozwarciający wynik: jeżeli odwróciemy urządzenie o 270 stopni w stronę przeciwną do kierunku ruchu wskazówek zegara (z pozycji domyślnej), metoda `Display.getRotation()` powróci z wartością 1 przy 90 stopniach i urządzenie przejdzie w tryb krajobrazowy, przy 180 stopniach cały czas pozostaje wartość 1 i tryb wyświetlania nie ulega zmianie, przy 270 stopniach układ graficzny zostaje odwrócony na odwrotny krajobrazowy, jednak metoda `Display.getRotation()` wciąż przekazuje wartość 1. Jeżeli obróćmy urządzenie z pozycji domyślnej o 90 stopni zgodnie z kierunkiem ruchu wskazówek zegara, metoda ta powróci z wartością 3. Ta pozycja wygląda dokładnie tak samo jak odwrócona pozycja z 270 stopni, zostaje jednak przekazana inna wartość w metodzie `Display.getRotation()`, w zależności od tego, jaką drogą do niej dotarliśmy.

## Akcelerometry i grawitacja

Do tej pory zajmowaliśmy się bardzo побieżnie kwestią zachowania odczytów akcelerometru w trakcie przemieszczania się urządzenia. Zastanówmy się teraz nad tym dokładniej.

Wszystkie siły działające na urządzenie zostaną wykryte przez akcelerometry. Jeżeli uniesiemy telefon, siła działająca w osi z będzie początkowo dodatnia, a jej wartość będzie większa niż +9,81. Jeżeli przesuniemy urządzenie w lewo, początkowa wartość wektora siły w osi x będzie ujemna.

Zależałoby nam teraz na oddzieleniu wartości wynikających z oddziaływania grawitacyjnego od pozostałych sił działających na urządzenie. Rozwiążanie jest całkiem proste i nosi nazwę filtra dolnoprzepustowego. Siły niebędące oddziaływaniem grawitacyjnym zazwyczaj niestopniowo wpływają na urządzenie. Inaczej mówiąc, jeżeli użytkownik potrąsi urządzeniem, pojawiające się siły są bardzo szybko rejestrowane przez akcelerometry. W związku z tym filtr dolnoprzepustowy usunie składową odpowiedzialną za same wstrząsy i pozostawi wyłącznie niezmienną składową, w tym przypadku przyspieszenie ziemskie. Zilustrujmy tę koncepcję na przykładzie. Interesujący nas projekt nosi nazwę **GravityDemo**. Na listingu 26.6 został umieszczony układ graficzny oraz kod Java.

#### **Listing 26.6.** Pomiar grawitacji za pomocą akcelerometrów

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <TextView android:id="@+id/text" android:textSize="20sp"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content" />
</LinearLayout>

// Jest to plik MainActivity.java.
import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends Activity implements SensorEventListener {
    private SensorManager mgr;
    private Sensor accelerometer;
    private TextView text;
    private float[] gravity = new float[3];
    private float[] motion = new float[3];
    private double ratio;
    private double mAngle;
    private int counter = 0;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        mgr = (SensorManager) this.getSystemService(SENSOR_SERVICE);
```

```
accelerometer = mgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

text = (TextView) findViewById(R.id.text);
}

@Override
protected void onResume() {
    mgr.registerListener(this, accelerometer,
        SensorManager.SENSOR_DELAY_UI);
    super.onResume();
}

@Override
protected void onPause() {
    mgr.unregisterListener(this, accelerometer);
    super.onPause();
}

public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Ignorujemy.
}

public void onSensorChanged(SensorEvent event) {
    // Wprowadzamy filtr dolnoprzepustowy, służący do uzyskania składowej grawitacji.
    // Pozostałością są składowe ruchu.
    for(int i=0; i<3; i++) {
        gravity[i] = (float) (0.1 * event.values[i] +
            0.9 * gravity[i]);
        motion[i] = event.values[i] - gravity[i];
    }

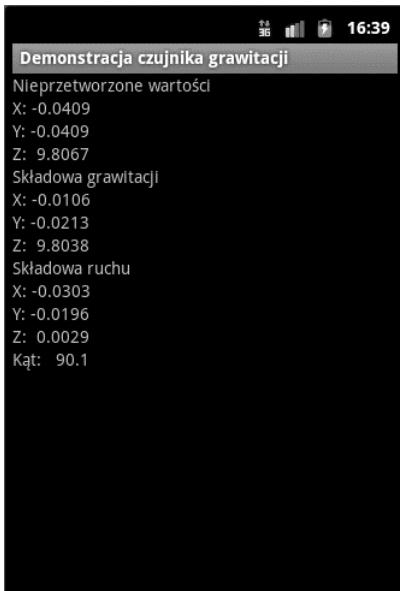
    // Zmienną ratio jest stosunek grawitacji przyłożonej w osi Y do standardowej wartości grawitacji.
    // Wartość ta powinna mieścić się w zakresie pomiędzy -1 i 1.
    ratio = gravity[1]/SensorManager.GRAVITY_EARTH;
    if(ratio > 1.0) ratio = 1.0;
    if(ratio < -1.0) ratio = -1.0;

    // Konwertuje radiany na stopnie, w trakcie kierowania się
    // w górę wartość zostaje przetworzona na ujemną.
    mAngle = Math.toDegrees(Math.acos(ratio));
    if(gravity[2] < 0) {
        mAngle = -mAngle;
    }

    // Wyświetla co dziesiątą wartość.
    if(counter++ % 10 == 0) {
        String msg = String.format(
            "Nieprzetworzone wartości\nX: %8.4f\nY: %8.4f\nZ: %8.4f\n" +
            "Składowa grawitacji\nX: %8.4f\nY: %8.4f\nZ: %8.4f\n" +
            "Składowa ruchu\nX: %8.4f\nY: %8.4f\nZ: %8.4f\nKąt: %8.1f",
            event.values[0], event.values[1], event.values[2],
            gravity[0], gravity[1], gravity[2],
            motion[0], motion[1], motion[2],
            mAngle);
        text.setText(msg);
    }
}
```

```
    text.invalidate();
    counter=1;
}
}
```

W wyniku uruchomienia powyższego kodu ujrzymy ekran zaprezentowany na rysunku 26.4. Poniższy zrzut ekranu został wykonany, kiedy urządzenie spoczywało płasko na stole.



**Rysunek 26.4.** Wartości grawitacji, ruchu i kąta

Aplikacja ta w większości przypomina wcześniejszy przykładowy program Monitor akcelerometru. Różnice pojawiają się w metodzie `onSensorChanged()`. Zamiast standardowego wyświetlania otrzymywanych wartości z tablicy próbujemy poznać składowe grawitacji i ruchu. Składową grawitacji uzyskujemy poprzez zsumowanie aktualnej i poprzedzającej wartości z tablicy grawitacji, przemnożonych przez współczynniki. Współczynniki te muszą po zsumowaniu dać wartość 1.0, a dobiera się je tak, że aktualną wartość z tablicy wartości grawitacji mnoży się przez mniejszy współczynnik, a poprzednią — przez większy (były suma tych współczynników była równa wartości 1.0). W naszym przykładzie wykorzystaliśmy współczynniki 0,9 i 0,1. Możemy również wypróbować inne wartości współczynników, na przykład 0,8 i 0,2. Tablica wartości grawitacji prawdopodobnie nie będzie zmieniała się tak szybko jak rzeczywiste odczyty czujnika. W ten sposób jednak zbliżamy się do rzeczywistych wartości. Do tego właśnie służy filtr dolnoprzepustowy. Wartości tabeli zdarzeń ulegają zmianom jedynie w przypadku sił poruszających urządzeniem, a my nie chcemy, aby były one mierzone jako część oddziaływania grawitacyjnego. W tabeli wartości grawitacji chcemy jedynie zarejestrować faktyczne odczyty siły ciążenia. Zastosowane tutaj obliczenia matematyczne nie sprawiają, że w magiczny sposób jest rejestrowane wyłącznie przyśpieszenie grawitacyjne, ale uzyskiwane wartości będą znacznie bliższe rzeczywistości niż nieprzetworzone dane.

Analizując kod, zwróciśmy również uwagę na tablicę wartości ruchu. Poprzez śledzenie różnicy pomiędzy nieprzetworzonymi wartościami zdarzeń a obliczonymi wartościami grawitacji mierzymy po prostu aktywne siły (niebędące przyspieszeniem ziemskim) działające na urządzenie. Jeżeli wartości w tablicy ruchu wynoszą zero lub są bliskie zeru, oznacza to, że urządzenie prawdopodobnie znajduje się w stanie spoczynku. Jest to bardzo przydatna informacja. W idealnym przypadku urządzenie poruszające się ze stałą prędkością również powinno generować wartości tabeli ruchu bliskie zeru, w rzeczywistości jednak są one większe.

## Używanie akcelerometrów do mierzenia kąta ułożenia urządzenia

Zanim przejdziemy dalej, chcielibyśmy zaprezentować jeszcze jedną cechę akcelerometrów. Jeżeli przypomnijmy sobie lekcje trygonometrii ze szkoły średniej, zauważymy, że cosinus kąta jest stosunkiem długości przyprostokątnej bliższej do tego kąta i przeciwprostokątnej. Jeśli weźmiemy pod uwagę kąt pomiędzy osią  $y$  a wektorem działania siły grawitacji, moglibyśmy zmierzyć wartość grawitacji działającej w kierunku  $y$ , a za pomocą funkcji  $\text{arcus cosinus}$  obliczyć kąt. Ta metoda znalazła zastosowanie w naszym kodzie. Musimy jednak w tym przypadku zmierzyć się z pewnym bałaganem związanym z architekturą czujników w Androidzie. W klasie `SensorManager` istnieją różne stałe definiujące grawitację, w tym również ziemską. Prawdopodobnie jednak mierzone wartości mogą przekraczać wartości określone przez te stałe. Za chwilę wyjaśnimy, co mamy na myśli.

Nasze urządzenie, pozostając w stanie spoczynku, powinno teoretycznie mierzyć wartość grawitacji równą wspominanej już wielokrotnie stałej, tak się jednak nieczęsto dzieje. W takim przypadku akcelerometr będzie prawdopodobnie dawał mniejsze lub większe odczyty od założonych. Zatem współczynnik grawitacji może być większy od 1 lub mniejszy od -1. Nasza funkcja `acos()` może zacząć w takim przypadku dawać niestabilne wyniki, zatem zamkamy otrzymywane wartości w zbiorze liczb z przedziału od -1 do 1. Jest to zakres kątowy równoważny wartościom od 0 do 180 stopni. Wygląda zachęcająco, ale w ten sposób nie uzyskamy kątów ujemnych, od 0 do -180 stopni. Żeby uzyskać takie ujemne wartości, korzystamy z kolejnego elementu tablicy wartości grawitacji, którym jest wartość  $z$ . Jeżeli wartość z grawitacji jest ujemna, oznacza to, że urządzenie jest skierowane wyświetlaczem w dół. W przypadku wszystkich wartości, w których urządzenie jest skierowane w dół, wprowadzamy znak minus, dzięki czemu zakres kątowy wynosi od -180 do +180 stopni, dokładnie tak, jak powinno być.

Nie bójmy się poeksperymentować z tą przykładową aplikacją. Zwróciśmy uwagę, że wartość kąta nachylenia urządzenia wynosi 90 stopni, gdy leży ono na stole, a 0 stopni (lub blisko tej wartości), gdy trzymamy je naprzeciwko twarzy. Jeżeli będziemy przechylali urządzenie jeszcze bardziej z pozycji płaskiej, wartość tego kąta zacznie przekraczać 90 stopni. Jeżeli będziemy je pochyłać coraz bardziej z pozycji 0 stopni, wartość kąta będzie przybierać wartości ujemne, aż w końcu urządzenie będzie ułożone wyświetlaczem do dołu i wartość tego kąta osiągnie wartość -90 stopni. Na koniec — pewnie zauważylismy w kodzie licznik kontrolujący częstotliwość aktualizacji ekranu. Ponieważ odczyty czujnika mogą być dość szybko odświeżane, postanowiliśmy pokazywać na wyświetlaczu zaledwie co dziesiąty wynik.

## Magnetometry

Magnetometr mierzy indukcję pola magnetycznego w otoczeniu i określa jej rozkład w osiach  $x$ ,  $y$  i  $z$ . Układ współrzędnych jest taki sam jak w przypadku akcelerometrów, więc możemy zastosować tu taki, jaki widać na rysunku 26.2. Jednostkami stosowanymi w magnetometrach są mikrotesle ( $\mu\text{T}$ ). Czujnik ten wykrywa ziemskie pole magnetyczne, dlatego też pozwala nam

określać kierunek północny. Magnetometr jest często nazywany również kompasem, nawet w znaczniku `<uses-feature>` jest stosowana nazwa `android.hardware.sensor.compass`. Magnetometr jest bardzo niewielkim i czulym urządzeniem, dlatego na jego odczyty wpływają pole magnetyczne generowane przez inne urządzenia znajdujące się w pobliżu, a w pewnym stopniu nawet układy znajdujące się w samym telefonie. Zatem wskazania magnetometru mogą być czasami niewiarygodne.

Wśród projektów przygotowanych specjalnie na potrzeby tego rozdziału umieściliśmy prostą aplikację `CompassSensor`, warto więc zainportować ją i trochę z nią poeksperymentować. Jeżeli zblizmy metalowe przedmioty do urządzenia w trakcie działania aplikacji, możemy zauważyc zmianę odczytów. Oczywiście, jeżeli blisko czujnika ustawimy magnes, odczyty na pewno ulegną zmianie, nie radzimy jednak tego robić, gdyż magnetometr może zostać rozkalibrowany.

Możemy się zastanawiać, czy istnieje możliwość wykorzystania magnetometru jako kompasu do wskazywania kierunku północnego. Odpowiedź brzmi: nie bezpośrednio. Chociaż magnetometr wykrywa strumienie magnetyczne otaczające urządzenie, jeżeli urządzenie nie będzie ułożone doskonale poziomo, jego odczyty jako kompasu będą niemiarodajne. Mamy jednak do dyspozycji akcelerometry, które informują nas o ułożeniu urządzenia względem osi stanowiącej kierunek działania siły grawitacji! Możemy więc wykorzystywać magnetometr jako kompas, będzie nam jednak do tego potrzebna pomoc akcelerometrów. Zobaczmy, jak się to robi.

## Współpraca akcelerometrów z magnetometrami

Klasa `SensorManager` zawiera pewne metody umożliwiające łączenie odczytów magnetometru z wartościami mierzonymi przez akcelerometr w celu określenia orientacji w przestrzeni. Jak niedawno wspomnieliśmy, nie możemy w tym celu wykorzystać samego magnetometru. W klasie `SensorManager` znajdziemy więc metodę `getRotationMatrix()`, pobierającą wartości akcelerometru i kompasu, a następnie przekazującą macierz danych pozwalających na określenie orientacji w przestrzeni.

Inna metoda tej klasy, `getOrientation()`, pobiera wspomnianą wcześniej tablicę obrotów i przetwarza ją na tablicę kierunkowości. Wartości podane w tej macierzy informują nas o obrocie urządzenia względem północy magnetycznej oraz o nachyleniu bocznym i wzdłużnym w stosunku do poziomu. Byłoby wspaniale, gdyby wszystkie te obliczenia były dokonywane automatycznie. Niestety, mechanizm ten stanowi wielkie wyzwanie, przynajmniej do wersji 2.2 Androida, gdzie jednym z problemów, i to wcale nie największym, jest brak ciągłości, w przypadku gdy trzymamy urządzenie naprzeciwko siebie, a następnie unosimy je nieznacznie w taki sposób, że spoglądamy nieco do góry na ekran. Owa nieciągłość polega na tym, że gdy tylko urządzenie przekroczy punkt 0 stopni (zgodnie z którym jeszcze znajdujemy się naprzeciwko urządzenia), magnetometr odwraca kierunki, co jest zupełnie nieintuicyjne. Na szczęście w wersji 2.3 Androida umieszczono kilka dodatkowych metod korygujących ten błąd (więcej informacji znajdziemy w punkcie „Czujniki wektora obrotu”). Jednak w międzyczasie powinniśmy również obsłużyć odczyty czujników w urządzeniach wyposażonych w wersje systemu starsze od 2.3.

## Czujniki orientacji w przestrzeni

Nadszedł czas na omówienie czujników orientacji. W poprzednim punkcie stwierdziliśmy, że można powiązać działanie akcelerometrów i magnetometrów w celu uzyskania odczytów dotyczących orientacji urządzenia, dzięki którym możemy dowiedzieć się, w jakim kierunku jest skierowany jego wyświetlacz. Takie samo zadanie wykonuje czujnik orientacji. W rzeczywistości

stanowi on połączenie akcelerometru i magnetometru na poziomie sterowników. Inaczej mówiąc, czujnik orientacji nie jest oddzielnym urządzeniem, ale oprogramowanie systemowe wiąże dwa wspomniane czujniki w taki sposób, że działają jak jeden układ elektroniczny.

**Uwaga!**

Nie wspominaliśmy o czujnikach orientacji aż do teraz, ponieważ zostały uznane za przestarzałe w wersji 2.2 Androida i nie zaleca się ich stosowania. Jednak są one bardzo przydatne, a do tego o wiele prostsze w użyciu od preferowanego rozwiązania, o czym się wkrótce przekonamy.

Niedawno stwierdziliśmy, że stosowanie preferowanego mechanizmu obliczania orientacji urządzenia w przestrzeni jest trudnym zadaniem. W następnym przykładzie porównamy wartości orientacji uzyskiwane z preferowanego rozwiązania z odczytami czujnika orientacji i przyjrzymy się różnicom.

Urozmaicimy nieco przykładową aplikację. Moglibyśmy po prostu wyświetlić wartości przekazywane przez czujniki, ale możemy je jeszcze wykorzystać w interesujący sposób. Wyobraźmy sobie, że stoiemy na ulicy w Jacksonville na Florydzie. Nasza aplikacja będzie nam pokazywać zdjęcia w trybie StreetView tego miasta, tak jakbyśmy tam byli, a czujnik orientacji posłuży nam do określenia kierunku, w jakim spoglądamy. Wraz ze zmianą kierunku orientacji telefonu będą się odpowiednio zmieniały widoki w trybie StreetView. Na listingu 26.7 widzimy układ graficzny i kod Java naszej przykładowej aplikacji, nazwanej VirtualJax.

**Listing 26.7.** Uzyskiwanie informacji o położeniu za pomocą czujników

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/main.xml -->
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
    <Button android:id="@+id/update" android:text="Aktualizuj wartości"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="doUpdate" />
    <Button android:id="@+id/show" android:text="Wyświetl moją pozycję!"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="doShow" android:layout_toRightOf="@+id/update" />
    <TextView android:id="@+id/preferred" android:textSize="20sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/update" />
    <TextView android:id="@+id/orientation" android:textSize="20sp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/preferred" />
</RelativeLayout>
```

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.content.Intent;
import android.hardware.Sensor;
```

```
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.net.Uri;
import android.os.Build;
import android.os.Bundle;
import android.view.View;
import android.view.WindowManager;
import android.widget.TextView;

public class MainActivity extends Activity implements SensorEventListener {
    private static final String TAG = "VirtualJax";
    private SensorManager mgr;
    private Sensor accel;
    private Sensor compass;
    private Sensor orient;
    private TextView preferred;
    private TextView orientation;
    private boolean ready = false;
    private float[] accelValues = new float[3];
    private float[] compassValues = new float[3];
    private float[] inR = new float[9];
    private float[] inclineMatrix = new float[9];
    private float[] orientationValues = new float[3];
    private float[] prefValues = new float[3];
    private float mAzmuth;
    private double mInclination;
    private int counter;
    private int mRotation;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        preferred = (TextView)findViewById(R.id.preferred);
        orientation = (TextView)findViewById(R.id.orientation);

        mgr = (SensorManager) this.getSystemService(SENSOR_SERVICE);

        accel = mgr.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        compass = mgr.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
        orient = mgr.getDefaultSensor(Sensor.TYPE_ORIENTATION);

        WindowManager window = (WindowManager)
            this.getSystemService(WINDOW_SERVICE);
        int apiLevel = Integer.parseInt(Build.VERSION.SDK);
        if(apiLevel < 8) {
            mRotation = window.getDefaultDisplay().getOrientation();
        }
        else {
            mRotation = window.getDefaultDisplay().getRotation();
        }
    }

    @Override
    protected void onResume() {
```

```
mgr.registerListener(this, accel,
    SensorManager.SENSOR_DELAY_GAME);
mgr.registerListener(this, compass,
    SensorManager.SENSOR_DELAY_GAME);
mgr.registerListener(this, orient,
    SensorManager.SENSOR_DELAY_GAME);
super.onResume();
}

@Override
protected void onPause() {
    mgr.unregisterListener(this, accel);
    mgr.unregisterListener(this, compass);
    mgr.unregisterListener(this, orient);
    super.onPause();
}

public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Ignorujemy
}

public void onSensorChanged(SensorEvent event) {
    // Musimy uzyskać dostęp do akcelerometru i kompasu,
    // zanim określmy wartości tablicy orientationValues
    switch(event.sensor.getType()) {
        case Sensor.TYPE_ACCELEROMETER:
            for(int i=0; i<3; i++) {
                accelValues[i] = event.values[i];
            }
            if(compassValues[0] != 0)
                ready = true;
            break;
        case Sensor.TYPE_MAGNETIC_FIELD:
            for(int i=0; i<3; i++) {
                compassValues[i] = event.values[i];
            }
            if(accelValues[2] != 0)
                ready = true;
            break;
        case Sensor.TYPE_ORIENTATION:
            for(int i=0; i<3; i++) {
                orientationValues[i] = event.values[i];
            }
            break;
    }

    if(!ready)
        return;

    if(SensorManager.getRotationMatrix(
        inR, inclineMatrix, accelValues, compassValues)) {
        // Uzyskaliśmy dobrą macierz obrotów

        SensorManager.getOrientation(inR, prefValues);

        mInclination = SensorManager.getInclination(inclineMatrix);
```

```
// Wyświetla co dziesiątą wartość
    if(counter++ % 10 == 0) {
        doUpdate(null);
        counter = 1;
    }
}

public void doUpdate(View view) {
    if(!ready)
        return;

    mAzmuth = (float) Math.toDegrees(prefValues[0]);
    if(mAzmuth < 0) {
        mAzmuth += 360.0f;
    }

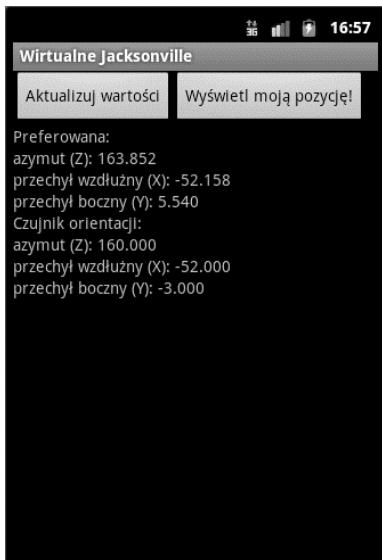
    String msg = String.format(
        "Preferowana:\nnazymut (Z): %7.3f \nprzechył wzdłużny (X): %7.3f\nprzechył
        boczny (Y): %7.3f",
        mAzmuth, Math.toDegrees(prefValues[1]),
        Math.toDegrees(prefValues[2]));
    preferred.setText(msg);

    msg = String.format(
        "Czujnik orientacji:\nnazymut (Z): %7.3f\nprzechył wzdłużny (X):
        ↪%7.3f\nprzechył
        boczny (Y): %7.3f",
        orientationValues[0],
        orientationValues[1],
        orientationValues[2]);
    orientation.setText(msg);

    preferred.invalidate();
    orientation.invalidate();
}

public void doShow(View view) {
    // google.streetview:cbll=30.32454,-81.6584&cbp=1,yaw,,pitch,1.0
    // yaw = wartość w stopniach, zgodnie ze wskazówkami zegara od bieguna północnego
    // W przypadku odchylenia (ang. yaw) możemy wykorzystać wartości mAzmuth
    // lub orientationValues[0].
    //
    // pitch = wartość w stopniach, przechył w górę lub dół. -90 oznacza spoglądanie w górę,
    // +90 to spoglądanie w dół,
    // nie biorąc pod uwagę faktu, że przechył wzdłużny (ang. pitch) nie jest poprawnie obliczany.
    Intent intent=new Intent(Intent.ACTION_VIEW, Uri.parse(
        "google.streetview:cbll=30.32454,-81.6584&cbp=1," +
        Math.round(orientationValues[0]) + ",,0,1.0"
    ));
    startActivity(intent);
    return;
}
}
```

Interfejs użytkownika stanowią dwa przyciski i para listingów zawierających odczyty czujników, z których na jednym są ukazane wartości generowane przez preferowaną metodę, a na drugim — wyniki z czujnika orientacji. Po uruchomieniu tej aplikacji powinniśmy ujrzeć ekran podobny do przedstawionego na rysunku 26.5.



Rysunek 26.5. Dwa sposoby określania orientacji w przestrzeni

Zanim przyjrzymy się wynikom, wyjaśnijmy, jakie jest zadanie tej aplikacji. W metodzie `onCreate()` przeprowadzamy takie same czynności jak w poprzednich przykładach: tworzymy odniesienia do widoków tekstowych, klasy `SensorManager` oraz trzech typów czujników, jakie chcemy wykorzystać: akcelerometru, kompasu i czujnika orientacji. Definiujemy również zmienną przechowującą wartość obrotu. Za chwilę dowiemy się po co.

W metodzie `onResume()` uruchamiamy czujniki, a w metodzie `onPause()` — wyłączamy je.

Podczas otrzymywania aktualizacji odczytów czujnika określamy, do której kategorii one należą, i rejestrujemy te wartości w lokalnych członkach: `accelValues`, `compassValues` lub `orientationValues`. Zauważmy, że moglibyśmy skopiować tablicę zdarzeń w celu zachowania lokalnych kopii odczytów; oznaczałoby to jednak ciągłe tworzenie obiektów, a to nie jest dobry pomysł. Tworzenie nowych obiektów i późniejsze ich usuwanie może być naprawdę kosztowne, jeśli chodzi o zużycie zasobów, dlatego ograniczamy się wyłącznie do aktualizowania już istniejących tablic.

Zwróćmy uwagę, że zanim zajmiemy się przetwarzaniem dalszej części kodu, sprawdzamy za pomocą operacji logicznej, czy posiadamy wartości zarówno w tablicy `accelValues`, jak i `compassValues`. Widzimy następnie wywołanie metody `getRotationMatrix()`, po której następuje wywołanie metody `getOrientation()`. Wprowadziliśmy również metodę `getInclination()`. Nie będziemy z niej korzystać, warto jednak wiedzieć, że reprezentuje ona kąt pomiędzy strumieniem magnetycznym a powierzchnią Ziemi. Im bliżej znajdujemy się któregoś z biegunów, tym większa jest przekazywana wartość kąta. Następnie tworzymy licznik, za pomocą którego będzie wyświetlana co dziesiąta aktualizacja wartości. Podobnie jak we wcześniejszych przykładach, mechanizm ten służy do zminimalizowania obciążenia interfejsu użytkownika, dzięki czemu aplikacja zyskuje na wydajności.

Wewnątrz metody `doUpdate()`, która może być wywoływana również za pomocą przycisku w interfejsie użytkownika, przeprowadzamy kilka obliczeń i wyświetlamy wyniki. W przypadku zalecanej metody pierwsza wartość, azymut, jest wyznaczana w radianach, w zakresie od ujemnej wartości pi do dodatniej wartości pi (czyli od -180 do 180 stopni). Wartości czujnika orientacji mieszczą się w zakresie od 0 do 360 stopni. Aby odczyty z obydwu rodzajów czujników były porównywalne, wzięliśmy pierwszą wartość z tablicy `prefValues`, przekształciliśmy radiany na stopnie i dodaliśmy 360, jeśli wartość była ujemna. Teraz wartości pozyskiwane z akcelerometru i magnetometru są porównywalne z odczytami czujnika orientacji. Pozostała część tej metody zapewnia obsługę wyświetlania tych wyników w interfejsie użytkownika.

Ostatnią metodą w naszej aplikacji jest `doShow()`. Uważamy, że jest naprawdę pasjonująca. W rozdziale 25. pokazaliśmy, w jaki sposób można przywołać aplikację StreetView za pomocą intencji. W tamtym rozdziale pominęliśmy część związaną z konfigurowaniem wartości odchylenia, która definiuje kierunek spoglądania użytkownika w przypadku wyświetlania obrazu. Przeanalizujemy teraz sposób przekazania wartości odchylenia, a także przechylu wzdużnego.

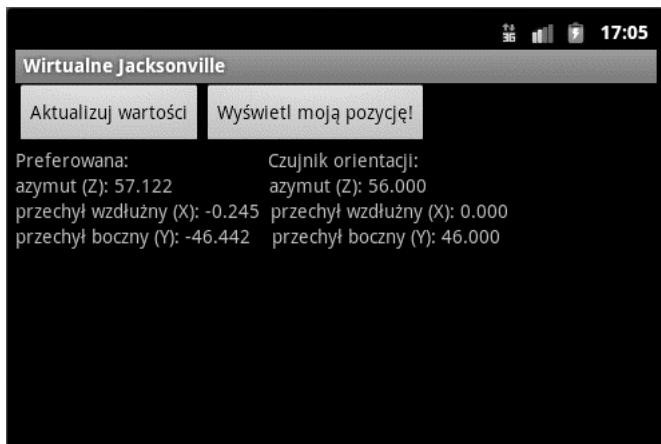
Jako długość i szerokość geograficzną wybraliśmy współrzędne miasta Jacksonville na Florydzie. Możemy wstawić tu oczywiście własne koordynaty. W przypadku odchylenia musimy przekazać wartość w stopniach liczoną od kierunku północnego (0 – 360), zatem możemy skorzystać z wartości zmiennej `mAzimuth` lub tablicy `orientationValues[0]`, przekształconej w liczbę całkowitą. Jeśli chodzi o przechyl wzdużny, moglibyśmy teoretycznie wykorzystać drugą wartość z którejś z tablic, a następnie dodać wartość 90. Jednak wydaje się, że aplikacja StreetView ma problemy z obsługą wartości przechylu wzdużnego innymi niż 0, przynajmniej w tej lokacji. Zatem na razie ustanawiamy wartość 0 przechylu wzdużnego. Jeżeli klikniemy przycisk *Wyświetl moją pozycję!*, zostanie uruchomiona aplikacja StreetView, a także wyświetlony obraz znajdujący się w kierunku, w którym spoglądamy. Jeżeli wciśniemy przycisk cofania, obróćmy się i ponownie klikniemy przycisk *Wyświetl moją pozycję!*, zostanie wyświetlony nowy obraz. Przyjrzymy się teraz rzeczywistym wartościom czujników.

Wartości generowane przez kod zgodny z zalecanym rozwiązaniem oraz przez czujnik orientacji wydają się identyczne albo bardzo do siebie zbliżone. Wartości pochodzące z tego drugiego mechanizmu wyglądają bardziej stabilnie, mamy tu również do czynienia z liczbami całkowitymi. Wydaje się, że jest naprawdę nieźle, ale to trochę pochopny wniosek. Jeżeli uniesiemy urządzenie nieco nad głowę i nachylimy je tak, żeby móc spoglądać na wyświetlacz, odczyty generowane obydwema metodami zaczną się od siebie znaczco różnić. Obróćmy teraz urządzenie, aby znalazło się w trybie krajobrazowym. Powinniśmy otrzymać wyniki przypominające widoczne na rysunku 26.6.

Co się stało? Wartości przechylu bocznego uzyskane w metodzie zalecanej i z odczytu czujnika orientacji uzyskały przeciwnie znaki. Problem polega na tym, że w obydwu mechanizmach stosowane są inne punkty odniesienia.

Nie wyjaśniliśmy jeszcze, co się dzieje, gdy urządzenie działa w trybie krajobrazowym, a nie portretowym. Jeżeli urządzenie jest ustawione naprzeciwko twarzy użytkownika w trybie krajobrazowym, akcelerometry nie zamieniają się miejscami, więc oś x zastępuje oś y i odwrotnie. W normalnych warunkach musielibyśmy w tym wypadku wprowadzić nieco przekształceń matematycznych, na szczęście klasa `SensorManager` zawiera pewną przydatną metodę — `remapCoordinateSystem()`. Może być ona wywołana pomiędzy momentem uzyskania macierzy obrotów a wywołaniem metody `getOrientation()`. Podstawową funkcją tej metody jest modyfikacja macierzy obrotów poprzez zamienienie osi układu współrzędnych. Sygnatura tej metody wygląda następująco:

```
public static boolean remapCoordinateSystem (float[] inR, int X, int Y, float[] outR)
```



**Rysunek 26.6.** Dane o orientacji urządzenia w przestrzeni otrzymane dwoma sposobami, urządzenie ustawione w trybie krajobrazowym

Przekazujemy w niej macierz obrotów oraz wartości określające sposób zamiany osi  $x$  i  $y$ , w wyniku czego otrzymujemy nową tablicę obrotów (`outR`), a także wartość logiczną, która wskazuje, czy proces przekształcania został zakończony powodzeniem. Wartości  $x$  i  $y$  są stałymi klasy `SensorManager`, takimi jak `AXIS_Z` lub `AXIS_MINUS_Y`.

Omawiane tu rozwiązanie zademonstrowaliśmy w przykładowej aplikacji *VirtualJaxWithRemap*, którą możemy pobrać wraz z innymi projektami.

## Deklinacja magnetyczna i klasa `GeomagneticField`

Chcielibyśmy poruszyć jeszcze jeden temat związany z orientacją w przestrzeni i określającymi ją urządzeniami. Dzięki kompasowi dowiemy się, gdzie znajduje się północ magnetyczna, nie wskaże nam on jednak rzeczywistej północy (geograficznej). Wyobraźmy sobie, że znajdujemy się na linii pomiędzy biegunem magnetycznym północnym a biegunem geograficznym północnym. Tworzyłyby one wtedy kąt 180 stopni. Im bardziej oddalilibyśmy się od obydwu biegunów, tym mniejszy kąt występowałby pomiędzy nimi. Różnica kątowa pomiędzy obydwoma typami biegunów zwana jest deklinacją magnetyczną. Wartość ta może być obliczona jedynie w odniesieniu do określonego punktu na powierzchni Ziemi. Oznacza to, że aby określić kierunek północy geograficznej, jeśli znamy kierunek północy magnetycznej, musimy jeszcze znać swoje położenie geograficzne. Na szczęście Android jak zwykle służy pomocą, tym razem przy użyciu klasy `GeomagneticField`.

Aby utworzyć obiekt klasy `GeomagneticField`, musimy przekazać jej współrzędne geograficzne. Zatem w celu określenia kąta deklinacji magnetycznej musimy znać położenie punktu odniesienia. Wymagana jest również znajomość godziny, w której będzie obliczana wartość deklinacji. Magnetyczny biegun północny zmienia swoje położenie w czasie. Po utworzeniu tego obiektu wywołujemy po prostu poniższą metodę, aby uzyskać kąt deklinacji (wyrażony w stopniach):

```
float declinationAngle = geoMagField.getDeclination();
```

Wartość zmiennej `declinationAngle` będzie dodatnia, jeżeli północ magnetyczna będzie się znajdowała na wschód od północy geograficznej.

## Czujniki grawitacji

Wraz z wersją 2.3 Androida wprowadzono czujniki grawitacji. W rzeczywistości nie jest to osobny układ elektroniczny. Mamy tu do czynienia z wirtualnym czujnikiem opartym na odczytach akcelerometrów. Tak naprawdę wykorzystywany jest tutaj opisany wcześniej mechanizm określania składowej grawitacji przez akcelerometry, w którym jest ona oddzielana od pozostałych sił działających na urządzenie. Nie możemy jednak modyfikować tego mechanizmu i musimy zaakceptować wszelkie współczynniki i przekształcenia dostępne w klasie tego czujnika. Był może w przyszłości ten wirtualny czujnik będzie wykorzystywał również inne sensory, na przykład żyroskop, do dokładniejszego pomiaru wartości grawitacji. Macierz wartości w przypadku tego czujnika przekazuje odczyty grawitacji w taki sam sposób, jak miało to miejsce w przypadku akcelerometrów.

## Czujniki przyśpieszenia liniowego

Podobnie jak miało to miejsce w przypadku czujnika grawitacji, czujniki przyśpieszenia liniowego są wirtualnymi sensorami reprezentującymi siły działające na urządzenie z wyłączoną składową grawitacji. Także i w tym przypadku ukazałismy wcześniej mechanizm pozwalający na uzyskanie tych wartości oraz usuwający z wyniku składową przyśpieszenia ziemskiego. Omawiany czujnik jeszcze bardziej ułatwia nam to zadanie. Był może w przyszłości również i ten sensor wykorzysta inne czujniki, na przykład żyroskop, do dokładniejszego mierzenia przyśpieszeń liniowych wpływających na urządzenie. Tablica wartości przedstawia wyniki w taki sam sposób jak w przypadku wskazań akcelerometrów.

## Czujniki wektora obrotu

Czujnik wektora obrotu przypomina wycofany już czujnik orientacji pod tym względem, że odczytuje orientację urządzenia w przestrzeni oraz podaje kąty zależne od punktu odniesienia wobec sprzętowego akcelerometru (rysunek 26.2). W trakcie pisania książki nie było jeszcze żadnych konkretnych informacji na temat tego czujnika. Prosimy zaglądać na naszą stronę ([www.androidbook.com](http://www.androidbook.com)), gdyż będziemy zamieszczać tam informacje na jego temat.

## Czujniki komunikacji bliskiego pola

Wraz z wprowadzeniem wersji 2.3 Androida uzyskaliśmy możliwość stosowania specjalnych terminali wykorzystujących pole NFC (ang. *Near Field Communication*). Przypominają one nieco terminale RFID (ang. *Radio Frequency Identification* — identyfikacja na częstotliwości radiowej), główna różnica polega na tym, że zasięg terminalu NFC wynosi 4 cale<sup>3</sup>. Oznacza to, że czujnik NFC musi bardzo zbliżyć się do terminalu (ang. *tag*), żeby móc go przeskanować. Terminali NFC mogą zostać zaprogramowane w taki sposób, aby przesyłyły dane tekstowe, identyfikatory URI oraz metadane, na przykład język, w jakim dana informacja jest przesyłana.

Zwróćmy uwagę, że technologia NFC nie jest nowością i w wielu krajach jest od lat znana i używana. W rzeczywistości w kilku państwach terminali kasowe wyposażone w terminalu NFC są dość powszechnie. Gdy terminal wykryje czujnik NFC, klient może dokonać transakcję za pomocą konta skojarzonego z identyfikatorem NFC. W internecie można znaleźć wiele filmów pokazujących użytkowników, którzy w ten sposób przeprowadzają transakcje płatnicze.

<sup>3</sup> Około 10 cm — przyp. tłum.

Reprezentanci firmy Google obiecuja, że pewnego dnia nasze portfele zostaną zastąpione przez telefony. Jest to istotnie kusząca wizja. Android pozwala przekształcić telefon w taki terminal wobec innego czytnika lub w czytnik wykrywający oraz skanujący terminale NFC.

W rzeczywistości istnieją trzy tryby działania mechanizmu NFC. Pierwszy tryb polega na odczytywaniu i zapisywaniu bezkontaktowych terminali. Drugim jest tryb emulacji karty. W tym trybie telefon pracujący pod kontrolą systemu Android może sam zachowywać się jak terminal NFC. Oczywistą zaletą tego rozwiązania jest możliwość zmiany zachowania urządzenia jako terminalu po wciśnięciu jednego przycisku. To właśnie dzięki temu trybowi nasz telefon może przekształcić się w portfel. Bez względu na rodzaj posiadanej karty kredytowej lub biletu nasze urządzenie może naśladować taki obiekt (oczywiście przy zastosowaniu wszelkich niezbędnych zabezpieczeń), dzięki czemu czytnik działa tak, jakby obsługiwał kartę kredytową, chociaż w rzeczywistości ma do czynienia z telefonem. Trzecim trybem mechanizmu NFC jest bezpośrednia, równorzędna komunikacja. W tym przypadku każde urządzenie jest równorzędne i nie są potrzebne terminale.

Wraz z wydaniem wersji 2.3.3 Androida możemy odczytywać terminale za pomocą urządzenia obsługującego ten system, podobnie jak ma to miejsce w terminalu kasowym z wcześniejszego przykładu, a także możemy zachowywać informacje w zapisywanych terminalach NFC. Jeżeli urządzenie użytkownika zostało poprawnie skonfigurowane, może przesyłać dane do innego urządzenia wyposażonego w czujnik NFC za pomocą protokołu P2P, zdefiniowanego przez firmę Google. W trakcie pisania książki nie była jeszcze dostępna funkcja emulacji karty lub, dokładniej, terminalu NFC. W istocie jest to bardzo trudne zadanie do wykonania, częściowo z powodu różnorodności architektur czujników NFC wprowadzanych do urządzeń. Nie wiadomo, kiedy tryb emulacji karty zostanie wprowadzony do pakietu SDK, wierzymy jednak, że kiedyś to nastąpi. W międzyczasie istnieje możliwość emulacji karty w pewnym zakresie na poziomie sterowników za pomocą zestawu Android NDK (ang. *Native Development Kit*). Zagadnienie to wykracza jednak poza zakres książki.

Oprócz przeprowadzania transakcji pieniężnych terminale NFC mogą spełniać również wiele innych zadań. Na przykład mogą być umieszczane w muzeum w pobliżu eksponatów i wysyłać adres URL do strony zawierającej multimedialne informacje na temat danego przedmiotu. Na przystankach autobusowych terminale mogą przechowywać rozkład jazdy interesującej nas linii. Przedsiębiorstwa mogą umieszczać w różnych miejscach terminale NFC umożliwiające łatwą rejestrację do swoich usług mobilnych. Być może zapomnimy o kluczach w hotelach, gdyż będziemy mogli otwierać drzwi urządzeniami wyposażonymi w NFC. Nawet produkty w sklepach mogą zostać zaopatrzone w terminale NFC, zawierające dokładniejsze informacje na ich temat, na przykład składniki i wartości odżywcze, parametry techniczne lub multimedialne reklamy.

## Aktywacja czujnika NFC

Obsługa czujnika NFC w Androidzie różni się od korzystania z pozostałych rodzajów sensorów. Nie stosujemy klasy `SensorManager`, tylko `NfcAdapter`. Zazwyczaj w urządzeniu dostępny jest tylko jeden czujnik NFC, zarządzający zapisywaniem informacji na terminalach i odczytywaniem ich treści, a także rozdzieleniem terminali pomiędzy aktywności. Adapter może być włączony lub wyłączony, natomiast w widoku *Ustawienia* znajdziemy opcje pozwalające na jego uaktywnianie lub wyłączanie. Opcje adaptera NFC można zazwyczaj znaleźć w zakładce *Sieci bezprzewodowe*. Jeżeli adapter jest włączony, po wykryciu terminalu nastąpi dość skomplikowany proces określający, która aktywność powinna otrzymać intencję informującą o obecności tego terminalu. Wszystko zależy od rodzaju danych przechowywanych przez terminal NFC, a tak-

że od obecności filtrów intencji dla zainstalowanych aplikacji w urządzeniu. Uwzględniana jest także jeszcze jedna informacja, mianowicie czy aktualnie pierwszoplanowa aktywność może otrzymywać terminale NFC. Niedługo zajmiemy się dokładniej tym zagadnieniem.

Aby uzyskać dostęp do adaptera, tworzymy najpierw za pomocą metody `getSystemService()` wystąpienie obiektu `NfcAdapter`. Następnie wywołujemy metodę `getDefaultAdapter()` w sposób zaprezentowany poniżej:

```
NfcManager manager = (NfcManager)
    context.getSystemService(Context.NFC_SERVICE);
NfcAdapter adapter = manager.getDefaultAdapter();
```

Otrzymamy w ten sposób singletonowy obiekt klasy `NfcAdapter`. Aby sprawdzić, czy klasa `NfcAdapter` jest aktywna, wprowadzamy metodę `isEnabled()`, która powraca z wartością logiczną określającą, czy technologia NFC została włączona w panelu *Ustawienia*. Nigdzie nie znaleźliśmy udokumentowanego sposobu na programowe włączanie i wyłączanie adaptera NFC. Jeżeli jest wyłączony, a dana aplikacja wymaga jego uruchomienia, musimy powiadomić użytkownika o konieczności ręcznego włączenia czujnika. Aby wyświetlić użytkownikowi właściwy widok ustawień, możemy skorzystać z następującego fragmentu kodu:

```
startActivityForResult(new Intent(
    android.provider.Settings.ACTION_WIRELESS_SETTINGS), 0);
```

Po przetworzeniu tego kodu Android otworzy odpowiedni widok ustawień i użytkownik będzie mógł włączyć adapter NFC. Metoda zwrotna `onActivityResult()` zostanie wywołana w chwili zamknięcia okna ustawień przez użytkownika. Pamiętajmy, że użytkownik może *nie* włączyć adaptera pomimo powiadomienia. Nasza aplikacja powinna być przygotowana również na ten scenariusz.

## Trasowanie terminali NFC

Nadszedł odpowiedni moment na omówienie różnych rodzajów technologii oraz terminali NFC. Mechanizm NFC nie jest ograniczony do jednego standardu. W rzeczywistości użytkownik może natrafić na kilka odmian terminali różniących się pomiędzy sobą. Terminali te nie posiadają takiej samej architektury, co oznacza, że Android musi zawierać dla każdego z nich oddzielną klasę. Jeśli zajrzymy do wnętrza pakietu `android.nfc.tech.package`, znajdziemy w nim kilka klas dotyczących różnych technologii terminali NFC, począwszy od klasy `MifareClassic`, poprzez `NfcV`, a skończywszy na `ISO-DEP`. Każdy rodzaj terminalu różni się od innych strukturą wewnętrzną, natomiast uzyskanie dostępu do zawartych w nich danych i manipulowanie nimi wymaga stosowania oddzielnych metod. Na szczęście Android został zaopatrzony w klasę `Tag` ułatwiającą komunikację NFC i za jej pomocą możemy utworzyć dowolny rodzaj terminalu. Po utworzeniu wystąpienia określonego terminalu NFC możemy przeprowadzać na nim dopuszczalne operacje. Oznacza to także, że należy wziąć pod uwagę kilka czynników przed wysłaniem terminalu do aktywności. Wyjaśnimy najpierw, w jaki sposób jest tworzona intencja terminalu NFC, dzięki czemu Czytelnik zrozumie mechanizm tworzenia odpowiednich filtrów intencji.

W trakcie przesyłania intencji zawierającej dane terminalu obiekt klasy `Tag` zawsze jest rozkładowany do pakietu dodatkowych danych intencji, a jego kluczem jest `EXTRA_TAG`. Jeżeli terminal zawiera informacje typu NDEF, zostaje dodana kolejna wartość z kluczem `EXTRA_NDEF_MESSAGES`. Ostatnim elementem dodatkowym może być identyfikator terminalu, którego klucz to `EXTRA_ID`. Dwie ostatnie wartości są opcjonalne i zależą od obecności danych w terminalu. Wszystkie intencje NFC są wysyłane za pomocą metody `startActivity()`. Zauważmy,

że tak naprawdę nie musimy nigdy uzyskiwać dostępu do adaptera, aby otrzymywać komunikaty NFC. Wiadomości z intencji będą przychodziły do aplikacji, tak samo jak wszelkie inne aplikacje przesyłane z różnych źródeł, tak długo, jak odpowiadają one filtrowi (filtrom) intencji.

### Uwaga!

Należy zwrócić uwagę, że technologia NFC dotyczy tylko urządzeń wyposażonych w czujnik NFC. Mechanizmy wymagane do tworzenia odpowiednich intencji zawierają funkcje nieobsługiwane przez pakiet SDK. Oznacza to, że samodzielne tworzenie testowej aktywności nadawczej jest bardzo trudne. W tym podrozdziale staramy się wyjaśnić mechanizmy rządzące tym systemem, dla których nie da się własnorzecznie napisać kodu. Oznacza to również, że aby rzeczywiście przetestować aplikację wykorzystującą mechanizm NFC, trzeba wykorzystać fizyczne urządzenie oraz terminal NFC. Być może kiedyś firma Google zaimplementuje odpowiednie funkcje w emulatorze lub w narzędziu DDMS.

W przypadku intencji terminalu wartość działania zależy od rodzaju informacji wykrytych w terminalu. Dla danej intencji istnieją trzy możliwe wartości działania:

1. ACTION\_NDEF\_DISCOVERED stanowi działanie w przypadku wykrycia bloku danych NDEF w terminalu. W takim przypadku Android szuka następnie obecności elementu NdefRecord w pierwszym obiekcie NdefMessage. Jeżeli elementem NdefRecord jest identyfikator URI lub rejestr SmartPoster, w polu danych intencji zostanie umieszczony ten identyfikator. Jeżeli z kolei zostanie wykryty rekord MIME, pole typu intencji zostanie zmodyfikowane do odpowiedniego typu MIME terminalu. System zacznie następnie szukać odpowiedniej aktywności dla tej intencji oraz właściwego algorytmu dopasowania intencji. Jeżeli nie zostanie znaleziona żadna aktywność, bieżąca intencja zostanie porzucona i Android spróbuje utworzyć następną intencję NFC.
2. ACTION\_TECH\_DISCOVERED jest działaniem podejmowanym, w przypadku gdy nie zostaną wykryte dane NDEF lub jeśli nie znajdziemy żadnej aktywności obsługującej ten format danych, lecz dostępna będzie technologia terminali. W tym przypadku Android dodaje metadane do intencji, za pomocą których zostanie uruchomiona odpowiednia technologia terminali. W terminalu NFC może zostać zaimplementowanych kilka różnych technologii, zwłaszcza że format Ndef bardziej przypomina wirtualny mechanizm. Android wyszukuje aktywność pasującą do intencji. Jeżeli zostanie znaleziona, prześlemy do niej intencję, w przeciwnym wypadku intencja ta zostanie porzucona i system wypróbuje trzeci rodzaj intencji NFC.
3. ACTION\_TAG\_DISCOVERED jest ostatnim działaniem definiowanym dla terminalu NFC. Jest ono podejmowane, gdy wszystkie pozostałe działania okażą się niedopasowane do aktywności. Intencja tego typu nie przenosi również danych ani typu MIME. Jeżeli ta intencja nie zostanie dopasowana do żadnej aktywności w urządzeniu, system NFC zaprzestaje prób i informacje o terminalu zostaną usunięte.

## Odbieranie terminali NFC

Bez względu na to, czy zdecydujemy się na utworzenie filtrów intencji za pomocą kodu, czy w pliku *AndroidManifest.xml*, musimy bardzo dobrze wiedzieć, czego szukamy, a filtry intencji przygotować z dużą ostrożnością. Jeżeli zdefiniujemy je zbyt rygorystycznie, aplikacja nie będzie powiadamiana o istotnych terminalach. Z kolei jeśli zdefiniujemy je niezbyt precyzyjnie, aplikacja zacznie otrzymywać komunikaty o terminalach, które nie są dla niej przeznaczone. W przypadku gdy nasza aplikacja otrzyma terminal dla niej nieprzeznaczony, być może w urządzeniu pojawią się problemy.

dzeniu istnieje inna aplikacja, dla której tego typu terminale są przeznaczone, jednak ta właściwa aplikacja nie otrzymała terminalu. Taka sytuacja może nastąpić, w przypadku gdy filtr intencji znalazł więcej niż jedną aplikację pasującą do terminalu. Wtedy system wyświetla monit, aby użytkownik wybrał właściwą aplikację. Może się zdarzyć, że użytkownik wybierze aplikację, dla której dany terminal nie jest przeznaczony. Istnieje więc kolejny powód, dla którego należy ostrożnie definiować filtry intencji dla terminali NFC: jeżeli użytkownik otrzyma monit o wybór aplikacji, z dużym prawdopodobieństwem przed podjęciem decyzji wyjdzie z zasięgu terminalu. Jeżeli możemy określić, jakiego typu dane terminali będą przetwarzane przez naszą aplikację, oznacza to, że możemy je bardzo dokładnie sprecyzować, na przykład za pomocą niestandardowego schematu identyfikatora URI lub własnego typu MIME.

Wybór filtru intencji zależy od rodzaju działania umieszczonego wewnętrz intencji terminalu NFC (zostało to omówione powyżej). Na listingu 26.8 zostało umieszczone przykładowy filtr intencji dla terminalu NDEF, który możemy umieścić w pliku *AndroidManifest.xml*.

#### **Listing 26.8.** Filtr intencji dla terminalu NDEF zawierającego typ MIME

---

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <data android:mimeType="type/subtype" />
</intent-filter>
```

---

Zamiast wartości type/subtype moglibyśmy oczywiście wskazać określony, poszukiwany przez nas typ MIME lub wprowadzić symbole wieloznaczne w przypadku akceptowania każdego typu lub podtypu. Możemy na przykład zdefiniować atrybut mimeType jako text/\*, dzięki czemu akceptowane byłyby wszystkie formaty tekstu. Nie musimy jednak definiować typu MIME dla terminalu NDEF. Jeżeli terminal ten posiada identyfikator URI, możemy utworzyć filtr intencji przypominający kod z listingu 26.9.

#### **Listing 26.9.** Filtr intencji dla terminalu NDEF zawierającego identyfikator URI

---

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>
    <data android:scheme="geo" />
</intent-filter>
```

---

W tym przykładzie definiujemy schemat geo, dzięki czemu po wykryciu terminalu zawierającego identyfikator rozpoczynający się od członu geo: zostanie uruchomiona nasza aktywność. Możemy stosować wszystkie atrybuty węzła <data> do określania, które dane terminalu NFC są oczekiwane przez naszą aktywność.

Jeżeli nasza aktywność wymaga terminali NFC utworzonych w określonej technologii, możemy skorzystać z filtru intencji zaprezentowanego na listingu 26.10. Może również zaistnieć sytuacja, w której zostanie wykryty terminal NDEF, lecz żadna aktywność nie będzie dopasowana do przetwarzania intencji NDEF\_DISCOVERED. Również w takim przypadku nasza aktywność może otrzymać tę intencję, dopóki jest ona zgodna z filtrem intencji. Inaczej mówiąc, jeżeli intencja terminalu zawierająca działanie NDEF\_DISCOVERED nie zostanie dostarczona do aktywności wyszukującej tego typu działania, zostanie wysłana do aktywności oczekującej terminalu utworzonego w określonej technologii.

**Listing 26.10.** Filtr intencji dla terminalu NFC zawierającego określona technologię

```
<intent-filter>
    <action android:name="android.nfc.action.TECH_DISCOVERED"/>
</intent-filter>
<meta-data android:name="android.nfc.action.TECH_DISCOVERED"
           android:resource="@xml/nfc_tech_filter" />
```

---

Zwróćmy uwagę, że wstawiliśmy teraz działanie definiujące poszukiwaną technologię, a zamiast węzła `<data>` wprowadziliśmy znacznik `<meta-data>`, który znajduje się poza znacznikiem `<intent-filter>`. Mamy również do czynienia z innymi znacznikami w tym węźle, które znajdują się w oddzielnym pliku, umieszczonym w katalogu `/res/xml`. Na listingu 26.11 demonstруjemy przykładowy plik `nfc_tech_filter.xml`.

**Listing 26.11.** Przykładowy plik XML zawierający filtr technologii NFC

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.NfcA</tech>
        <tech>android.nfc.tech.MifareUltralight</tech>
    </tech-list>
</resources>

<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
    <tech-list>
        <tech>android.nfc.tech.NfcB</tech>
        <tech>android.nfc.tech.Ndef</tech>
    </tech-list>
</resources>
```

---

Filtr ten definiuje dwa rodzaje terminali oczekiwanych przez naszą aplikację. Terminal NFC zazwyczaj zawiera listę obsługiwanych przez niego technologii. Jeżeli którykolwiek z elementów tej listy został wymieniony w filtrze z listingu 26.11, nasza aktywność uzyska dostęp do intencji tego terminalu.

Na listingu 26.11 pierwszy rodzaj terminalu określa technologie NfcA oraz MifareUltralight, w drugim zaś zdefiniowano: NfcB i Ndef. Możemy dodawać kolejne węzły `<resources>` do tego pliku w celu definiowania następnych terminali akceptowanych przez naszą aktywność. Uwzględniane tutaj technologie biorą swoje nazwy od nazw klas dostępnych w pakiecie `android.nfc.tech`, ale powinniśmy wpisywać jedynie te mechanizmy, które będą przydatne aktywności. Węzły potomne znacznika `<tech-list>` zawierają wszystkie technologie, które powinien posiadać terminal NFC, aby jego intencja pasowała do aktywności. Wszystkie technologie z danej listy muszą się znajdować na liście technologii obsługiwanych przez terminal NFC. Zatem lista technologii w filtrze może zawierać mniej elementów niż analogiczna lista w terminalu NFC, nie może jednak wystąpić odwrotna sytuacja. Kontynuując powyższy przykład, jeżeli w terminalu NFC znajdzie się wyłącznie wskazanie technologii Ndef, terminal ten nie zostanie przepuszczony przez żaden filtr i aktywność nie otrzyma jego intencji. Żadna z wymienionych list filtru intencji nie stanowi podzbioru na liście terminalu. Gdyby ten terminal zawierał technologie NfcA, NfcB oraz Ndef, okazałby się zgodny z drugą specyfikacją i zostałby przesłany do aktywności. Ta druga specyfikacja stanowi podzbior listy technologicznej terminalu NFC. Terminal ten byłby dopasowany, nawet gdyby zawierał dodatkowe technologie, niewymienione w filtrze intencji.

Ostatni filtr intencji, który może się przydać Czytelnikowi, został zaprezentowany na listingu 26.12. Charakteryzuje go uniwersalność. Oznacza to, że jeśli po wykryciu terminalu NFC nie znaleziono żadnej aktywności odbierającej terminale NDEF bądź zgodnej z określonymi w nim technologiami lub jeśli mamy do czynienia z nieznanym typem terminalu, zostanie utworzona intencja zawierająca działanie ACTION\_TAG\_DISCOVERED.

#### **Listing 26.12.** Filtr intencji dla nieznanego lub nieprzetwarzanego terminalu NFC

---

```
<intent-filter>
    <action android:name="android.nfc.action.TAG_DISCOVERED"/>
</intent-filter>
```

---

Zwróćmy uwagę, że dla tego filtru intencji nie zdefiniowano żadnego węzła <data> ani <meta-data>, ponieważ nie są przenoszone żadne dane w intencji oznaczonej działaniem ACTION\_TAG\_DISCOVERED. W normalnej sytuacji oznaczałoby to konieczność wprowadzenia znacznika <category>. Sprawa ma się jednak inaczej z intencjami terminali NFC. Stanowią one specjalny przypadek, zatem w filtrach intencji nie są wymagane tego typu terminale w celu dopasowania intencji. Jeżeli intencja otrzymuje działania ACTION\_TAG\_DISCOVERED, oznacza to, że system nie zdołał odnaleźć aktywności dla terminali NFC. W tym momencie każda aktywność przyjmująca to działanie otrzyma intencję tego terminalu. W większości standardowych operacji nigdy nie natrafimy na intencję znacznika ACTION\_TAG\_DISCOVERED, ponieważ większość terminali NFC będzie dopasowanych do kryteriów NDEF lub TECH.

Istnieje jeszcze jeden sposób, w jaki aktywność może otrzymać intencję terminalu NFC — zastosowanie systemu dyspozycji pierwszoplanowej. Jeżeli nasza aktywność znajduje się na pierwszym planie (co oznacza, że została uruchomiona metoda onResume() i użytkownik korzysta z tej aktywności), możemy utworzyć intencję oczekującą, tablicę filtrów intencji, tablicę list technologii, a następnie wprowadzić następujące wywołanie:

```
mAdapter.enableForegroundDispatch(this, pendingIntent,
    intentFiltersArray, techListsArray);
```

gdzie mAdapter jest adapterem NFC, a this stanowi odniesienie do naszej aktywności. Za pomocą tego wywołania skutecznie wystawiamy naszą aktywność przed wszystkie pozostałe aktywności i jeżeli którykolwiek z jej filtrów jest dopasowany do wykrytego terminalu NFC, to aktywność ta przetworzy terminal. Jeżeli aktywność nie otrzyma intencji terminalu z powodu niedopasowania, jej działanie będzie sprawdzane wobec pozostałych aktywności. Musimy wywołać tę metodę z poziomu wątku interfejsu użytkownika, a najlepiej tego dokonać w metodzie onResume() naszej aktywności. Wymagane byłoby również wprowadzenie następującego wywołania:

```
mAdapter.disableForegroundDispatch(this);
```

z poziomu metody zwrotnej onPause(), dzięki czemu nasza aktywność nie otrzyma intencji, której nie będzie mogła przetworzyć. Gdy aktywność w taki sposób otrzyma intencję, przekaże ją za pomocą metody zwrotnej onNewIntent().

Mamy tu do czynienia ze standardową intencją oczekującą. Tablica intentFiltersArray może stanowić zbiór potrzebnych nam obiektów IntentFilter, z których każdy definiuje określone działanie, a także, w razie potrzeby, dowolne dane lub typy MIME. Na listingu 26.13 widzimy przykładowy kod generujący filtr intencji dla obiektu Ndef, który następnie zostaje wstawiony do tablicy.

**Listing 26.13.** Kod filtru intencji dla technologii Ndef

```
IntentFilter ndef = new IntentFilter(NfcAdapter.ACTION_NDEF_DISCOVERED);
try {
    ndef.addDataType("text/*");
}
catch (MalformedMimeTypeException e) {
    throw new RuntimeException("fail", e);
}
intentFiltersArray = new IntentFilter[] {
    ndef,
};

```

---

Nie zapominajmy, że w tablicy filtrów intencji może się znaleźć wiele wystąpień obiektów IntentFilter, zawierających te same lub różne działania, a także posiadających dane lub ich pozbawione. To samo dotyczy wartości pól typów.

Obiekt techListsArray jest tablicą, której wartościami są inne tablice — listy zawierające nazwy klas obsługiwanych przez znacznik NFC. Możemy określić wiele list technologii. Zostało to zaprezentowane na listingu 26.14, który jest odpowiednikiem pliku zasobów ukazanego na listingu 26.11.

**Listing 26.14.** Kod tabeli zawierającej listy technologii

```
techListsArray = new String[][] {
    new String[] { NfcA.class.getName(),
                  MifareUltralight.class.getName() },
    new String[] { NfcB.class.getName(),
                  Ndef.class.getName() }
};
```

---

Jeśli po przeprowadzeniu omawianego procesu konfiguracji nasza aktywność uzyska dostęp do intencji terminalu NFC, spowoduje to uruchomienie metody zwrotnej `onNewIntent()` w celu odebrania terminalu. Z tego miejsca możemy odczytać dodatkową zawartość intencji, którą stanowią przechowywane informacje terminalu, co będzie tematem następnego podpunktu. Owszem, aby w dynamiczny sposób uzyskiwać dostęp do intencji terminalu NFC, trzeba włożyć dużo pracy, jednak z drugiej strony, jeżeli po uruchomieniu przez użytkownika tylko ta aktywność odbierała terminale, warto wykorzystać pokazane tu rozwiązanie. Zwróćmy jeszcze uwagę, że prawdopodobnie nie ma sensu jednokrotnie korzystanie z tej metody i umieszczanie filtrów intencji w pliku manifeście, jednak z technicznego punktu widzenia jest to możliwe.

## Odczytywanie terminali NFC

Jak już wcześniej sugerowaliśmy, odczytywanie terminali NFC jest dość skomplikowaną czynnością. Dokładniej mówiąc, sam proces dostarczania terminalu do aplikacji jest złożony. Wyjaśniając to na najbardziej podstawowym poziomie, w momencie wykrycia terminalu NFC system spróbuje określić aktywność, do której należy wysłać intencję tego terminalu. W przeciwieństwie do aktywności obsługujących pozostałe czujniki omawiane w tym rozdziale, aktywność przetwarzająca terminale NFC nie musi być uruchomiona w momencie ich wykrycia i z pewnością nie otrzyma informacji o terminalu za pomocą obiektu nasłuchującego. Powiadomiona aktywność otrzyma intencję, a to z kolei może oznaczać jej uruchomienie w celu przetworzenia danych terminalu.

Jedną z pierwszych kwestii rozważanych w procesie projektowania aplikacji otrzymującej i przetwarzającej intencje NFC jest konieczność obsługi fizycznego terminalu znajdującego się w otoczeniu urządzenia za pomocą interfejsu sprzętowego. Interfejs API generuje wywołania blokujące, co oznacza, że nie będą one przekazywane tak szybko, jak byśmy sobie tego życzyli, w związku z czym będziemy musieli uruchamiać metody terminalu w osobnym wątku.

Informacje terminalu NFC są umieszczone w pakiecie dodatkowych danych otrzymywanej intencji. Po odebraniu intencji możemy uzyskać dostęp do tych informacji za pomocą następującego fragmentu kodu:

```
Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
String[] techlists = tag.getTechLists();
```

Jeżeli zdefiniowany przez nas filtr intencji jest bardzo dokładny, będziemy już doskonale wieźć, jaki rodzaj terminalu otrzymaliśmy. Jeżeli jednak zdefiniowaliśmy większą liczbę akceptowanych technologii terminalu, możemy teraz sprawdzić listy tych technologii, aby poznać mechanizmy, z jakimi będziemy mieli do czynienia w terminalu. Każdy ciąg znaków na tej liście stanowi nazwę klasy przechowującą technologię obsługiwany przez wykryty terminal.

Jeżeli nasz terminal obsługuje klasę `android.nfc.tech.Ndef`, możemy wykorzystać poniższy kod do uzyskania bardziej bezpośredniego dostępu do danych NDEF:

```
NdefMessage[] ndefMsgs = intent.getParcelableArrayExtra
↳(NfcAdapter.EXTRA_NDEF_MESSAGES);
```

Theoretycznie możemy otrzymać wartość null, w przypadku gdy intencja nie będzie zawierała informacji NDEF. W przeciwnym wypadku nie powinno być problemu z analizą składni przesyłanych danych. Możemy odczytywać elementy klasy `NdefMessage` z poziomu intencji, zliczać je i w przypadku każdej z nich odbierać zawarte w nich obiekty `NdefRecord`.

Obiekty klasy `NdefRecord` są dość interesujące. Nie zaszkodzi, jeżeli Czytelnik zajrzy do specyfikacji technologii NFC, dostępnej pod adresem <http://www.nfc-forum.org/specs/>. Aby uzyskać do niej dostęp, trzeba zaakceptować postanowienia licencyjne NFC Forum. Jest to bezpłatny proces, należy jednak podać imię i nazwisko, adres, numer telefonu i adres e-mail. Alternatywnym rozwiązaniem jest aplikacja `NfcDemo` dostępna w pakiecie SDK Androida 2.3.3, w folderze z przykładowymi programami. Kod źródłowy tego przykładu został umieszczony na stronie <http://developer.android.com/resources/samples/NFCDemo/index.html>. Aplikacja ta odbiera intencje NFC i wyświetla zawartość klasy `NdefRecord` w widoku `ListView`. Sytuacja komplikuje się z powodu obecności kilku odmian klasy `NdefRecord`, które mogą zostać przesłane wraz z obiektami `NdefMessage`. Każda odmiana tej klasy ma inne zastosowanie. Na przykład odmiana `Text` przechowuje tekst w określonym języku. W odmianie `Uri` znajdziemy identyfikator URI. Ze wszystkich znanych rodzajów rekordów NDEF aplikacja `NfcDemo` obsługuje tylko trzy, z czego dwa przed chwilą omówiliśmy, a trzecim jest `SmartPoster`, któremu wkrótce przyjrzymy się uważniej.

Format klasy `NdefRecord` składa się z trzybitowego pola TNF (ang. *Type Name Format* — format nazwy typu), pola typu o zmiennej długości, pola identyfikatora o zmiennej długości oraz pola ładunku o zmiennej długości. A zatem mamy do czynienia z dwoma kategoriami pól. Pole TNF stanowi podstawowy typ tego obiektu i definiuje zawartość całego rekordu. Może być to na przykład bezwzględny rekord URI (`TNF_ABSOLUTE_URI`) lub oficjalny rekord RTD (`TNF_WELL_KNOWN`). Następne pole typu przechowuje dokładniejsze informacje na temat rodzaju rekordu w oparciu o wartość pola TNF. Jeżeli w polu TNF została zdefiniowana stała `TNF_WELL_KNOWN`, pole to będzie się składało ze stałych `RTD_*` klasy `NdefRecord`, takich jak `RTD_SMART_POSTER`. Jeżeli wartością pola TNF jest `TNF_ABSOLUTE_URI`, kolejnym polem typu będzie konstrukt BNF niezależnego identyfikatora URI, zdefiniowany w specyfikacji RFC 3986.

**Uwaga!**

Typ rekordu TNF\_UNCHANGED jest stosowany, w przypadku gdy ładunek komunikatu z powodu rozmiarów zostaje rozmieszczony w kilku obiektach NdefRecord. System automatycznie obsługuje tego typu podzielone obiekty NdefRecord, więc nie powinniśmy nigdy mieć do czynienia z wartością TNF\_UNCHANGED. Pakiet android.nfc łączy poszczególne elementy ładunku w jeden wielki obiekt NdefRecord.

Następnym polem obiektu NdefRecord jest jego identyfikator. Odczytywany rekord może posiadać identyfikator, chociaż nie jest to wymagane.

Na końcu mamy do czynienia z ładunkiem. Może to być dość duża tablica bajtowa, która posiada jednak wewnętrzną strukturę, zależną od rodzaju obiektu NdefRecord. Trzeba zwrócić uwagę na tę strukturę. W przypadku typu RTD URI pierwszy bajt tej tablicy reprezentuje początek identyfikatora URI. Na przykład wartość 1 oznacza `http://www.` — i od tego członu będzie rozpoczynał się każdy identyfikator URI znajdujący się we wnętrzu ładunku. W przypadku typu rekordu Text pierwszy bajt tabeli ładunku stanowi wartość kodowania tekstu (UTF-8 lub UTF-16), a także długość tablicy języka, występującej tuż po polu stanu kodowania. Za polem języka znajduje się właściwy tekst. W przypadku obiektów SmartPoster sprawa jest bardziej skomplikowana, gdyż każdy obiekt NdefRecord przechowuje obiekty NdefMessage, przechowujące z kolei więcej obiektów NdefRecord. Te ostatnie mogą zawierać rekord Title (zbudowany tak samo jak rekordy Text), rekord URI (nieróżniący się od omawianego wcześniej), rekord zalecanego działania, rekord rozmiaru, rekord ikony oraz rekord typu. Wartość zalecanego działania wskazuje na czynności, jakie aplikacja może wykonywać w przypadku danego obiektu SmartPoster. Zwrócić uwagę, że wartości zalecanego działania nie są wymieniane w dokumentacji klasy NdefRecord. Są one następujące:

- 1 UNKNOWN
- 0 DO\_ACTION
- 1 SAVE\_FOR\_LATER
- 2 OPEN\_FOR\_EDITING

Tylko od nas zależy, co z nimi zrobimy, chociaż oczywiście prawdopodobnie będziemy chcieli spróbować przeprowadzić na odczytywanym terminalu najważniejszą operację. Jeśli na przykład mamy do czynienia z formatem rekordu TNF\_WELL\_KNOWN, jego typem jest RTD\_SMART\_POSTER, a zalecane działanie przybiera wartość 0 (DO\_ACTION) i jest połączone z adresem URL strony WWW, najprawdopodobniej będziemy chcieli uruchomić przeglądarkę internetową i otworzyć stronę dostępną pod tym adresem. Rekord rozmiaru pozwala zdefiniować wielkość obiektu czekającego po drugiej stronie łączna. Jeżeli terminal odnosi się do pobieralnego pliku, w rekordzie rozmiaru może zostać określony jego rozmiar. W rekordzie ikony przechowywany jest obraz ikony, wykorzystywany przez urządzenie do jej wyświetlania wraz z tytułem oraz adresem URL.

Rekord typu nie jest tym samym co wartość formatu TNF czy typ klasy NdefRecord. Jest on wykorzystywany w terminalach SmartPoster, a w tym przypadku reprezentuje on typ MIME obiektu znajdującego się po drugiej stronie adresu URL. Urządzenie może nie obsługiwać danego typu MIME i w ten sposób zostanie uniemożliwione pobieranie tego obiektu.

Jedynym niezbędnym podrekordem terminalu SmartPoster jest rekord URI i tylko jedno jego wystąpienie może się znajdować w każdym terminalu. Możemy wypisać wiele rekordów Title, każdy przechowujący tekst w innym języku. Możliwe jest również zamieszczanie wielu rekordów ikony, pod warunkiem że każdy z nich posiada osobny typ MIME dla swojego formatu.

W przypadku wszystkich rodzajów terminali NFC, w tym również terminali NDEF, możemy zastosować poniższy fragment kodu, aby uzyskać wystąpienie danego typu terminali:

```
NfcA nfca = NfcA.get(tag);
```

Z pomocą tak utworzonego obiektu możemy uzyskać dostęp do określonych metod, najbardziej nadających się dla danego typu terminalu. W przypadku terminali `Ndef` i `NdefFormatable` klasy `NdefMessage` oraz `NdefRecord` okazują się bardzo przydatne do przetwarzania ich danych. Klasy pozostałynych rodzajów terminali posiadają odpowiednie metody umożliwiające obsługę tych terminali i zawartych w nich informacji. Mamy do dyspozycji odpowiednie metody do odczytu i zapisu danych na terminalu. Zwróćmy uwagę, że zapisywanie danych w terminalu nie jest tym samym co emulacja karty. Zapisywanie terminalu oznacza, że w pobliżu urządzenia znajduje się jakiś terminal, który można zmodyfikować (jeśli mamy odpowiednie uprawnienia). Emulacja karty jest oddzielnym procesem.

## Emulacja karty NFC

Emulacja karty oznacza imitowanie zachowania urządzenia wyposażonego w czujnik NFC jako terminal NFC przed odpowiednim czytnikiem. Oznacza to, że w określonym miejscu lokalnego urządzenia są przechowywane dane. Jeżeli w zasięgu tego urządzenia znajdzie się czytnik NFC, który zażąda dostępu do tych informacji, zostaną mu one udostępnione. Funkcja ta nie była jeszcze dostępna w czasie pisania tej książki, chociaż oczekujemy, że w końcu będzie możliwa z niej korzystać. Jeżeli emulacja karty jest Czytelnikowi naprawdę potrzebna, zalecamy zapoznanie się z dokumentacją sieciową, omawiającą przeprowadzenie tego procesu na najbardziej elementarnym poziomie urządzenia, tj. na poziomie zestawu NDK.

## Połączenia równorzędne (P2P) NFC

Zestaw Android SDK umożliwia w ograniczonym stopniu obsługę komunikacji P2P (ang. *peer-to-peer* — komunikacja równorzędna) pomiędzy dwoma urządzeniami za pomocą technologii NFC.

Funkcja ta nie jest pozbawiona wad, mianowicie korzystająca z niej aplikacja musi być uruchomiona i działać na pierwszym planie, a ponadto obsługiwać format NDEF. Być może pozostałe technologie będą w przyszłości obsługiwać komunikację P2P, na razie jednak pozwala na to jedynie format NDEF. Oznacza to także, że telefon musi być włączony, a aplikacja uruchomiona, aby móc w ten sposób nawiązać komunikację z innym urządzeniem.

Aby zaimplementować funkcję P2P, wykorzystamy metodę klasy `NfcAdapter` zwaną `enableForegroundNdefPush()`. Przyjmuje ona dwa parametry: aktywność oraz obiekt `NdefMessage`, które zostaną wysłane w momencie żądania danych przez inne urządzenie NFC. Podobnie jak w przypadku omówionego wcześniej systemu dyspozycji pierwszoplanowej, metoda ta powinna być wywoływana we wnętrzu metody `onResume()`, a wyłączana w metodzie `onPause()`. Obiekt `NdefMessage` może być dowolny, ale nasza aktywność powinna się znajdować na pierwszym planie podczas próby uzyskania dostępu do danych urządzenia przez czytnik. Firma Google stwierdziła, że urządzenie znajdujące się po drugiej stronie musi posiadać implementację protokołu wysyłania NDEF zawartego w pakiecie `com.android.npp`, co umożliwi nawiązywanie komunikacji z telefonem, jednak w momencie pisania książki nie było żadnych konkretnych informacji na ten temat. Będziemy jednak zamieszczać aktualizacje na naszej stronie.

Wyjaśnialiśmy wcześniej sposób wykorzystania węzła `uses-feature` zawierającego wpisy o czujnikach, dzięki czemu można się było upewnić, że dane urządzenie jest wyposażone w czujniki niezbędne do działania danej aplikacji jeszcze przed jej instalacją. Czujnik NFC nie stanowi w tym przypadku wyjątku. Powinniśmy umieścić następujący fragment w pliku `AndroidManifest.xml`, jeżeli chcemy, aby nasza aplikacja była instalowana jedynie na urządzeniach wyposażonych w czujnik NFC:

```
<uses-feature android:name="android.hardware.nfc" />
```

Lepiej upewnić się również, że plik `AndroidManifest.xml` zawiera odpowiednie uprawnienia, pozwalające aplikacji na uzyskanie dostępu do technologii NFC:

```
<uses-permission android:name="android.permission.NFC" />
```

## Testowanie technologii NFC za pomocą aplikacji NfcDemo

Omówiliśmy dużą część interfejsu technologii NFC, teraz jednak warto zadać pytanie: w jaki sposób możemy przetestować naszą aplikację? Jeśli chodzi o terminale NFC, być może udałoby się znaleźć w pobliżu jakieś wyposażone w nie obiekty. Nie powinno być to zbyt trudne w krajach, w których technologia ta zdała się już zadomowić. W Polsce może być o wiele trudniej. Możemy zakupić własne terminale NFC; na całym świecie można znaleźć kilku producentów sprzedających te podzespoły, a także odpowiednie oprogramowanie, umożliwiające zapisywanie na nich informacji. Niestety, narzędzie DDMS nie zostało wyposażone w funkcję przesyłania intencji wykrywanych terminali do emulatora. Przykładowa aplikacja `NfcDemo` została dołączona do wersji 2.3 Androida w czasach, gdy było dostępne wyłącznie działanie intencji `ACTION_TAG_DISCOVERED`. Wraz z wersją 2.3.3 Android się rozwinął, czego nie można powiedzieć o aplikacji `NfcDemo`. Można w niej znaleźć przydatne informacje na temat układów graficznych terminali NFC oraz o znaczeniu poszczególnych bajtów w tabelach danych terminali. Miejmy nadzieję, że wkrótce pojawi się zaktualizowana wersja tej aplikacji testowej, dostosowana do pracy z fizycznymi terminalami NFC oraz nowym systemem technologii NFC.

Jeżeli Czytelnik zdecyduje się na wczytanie przykładowej aplikacji `NfcDemo`, potrzebna mu będzie dodatkowa, zewnętrzna biblioteka. Plik tej biblioteki znajdziemy na stronie <http://code.google.com/p/guava-libraries/>. Po rozpakowaniu pliku ZIP uzyskamy dostęp do plików JAR. Należy zapisać w stacji roboczej plik `guava` (bez członu `gwt`). Trzeba utworzyć w środowisku Eclipse odniesienie do tego pliku; w tym celu należy kliknąć nazwę projektu prawym przyciskiem myszy, wybrać opcję `Build Path`, następnie `Configure Build Path` i otworzyć zakładkę `Libraries`. Następnie trzeba kliknąć opcję `Add External JARs`, wyszukać plik JAR, wybrać go i kliknąć przycisk `Open`. Pozostaje jeszcze ponowne skompilowanie projektu `NfcDemo`, trzeba więc kliknąć jego nazwę prawym przyciskiem myszy i wybrać opcję `Build Project`.

## Odbońniki

Znajdziemy tu łącza do materiałów pomocnych w zrozumieniu koncepcji zawartych w tym rozdziale:

- <ftp://ftp.helion.pl/przyklady/and3ta.zip> — znajdziemy tu listę projektów utworzonych specjalnie na potrzeby niniejszej książki. Projekty przeznaczone dla tego rozdziału zostały umieszczone w katalogu `ProAndroid3_R26_Czujniki`. Każdy z zawartych w nim projektów znajduje się w osobnym katalogu. W katalogu umieściliśmy również plik `Czytaj.TXT`, stanowiący dokładną instrukcję importowania projektów do środowiska Eclipse.

- <http://en.wikipedia.org/wiki/Lux> — informacje o jednostce natężenia światła — luksie.
- <http://android-developers.blogspot.com/2010/09/one-screen-turn-deserves-another.html> — wpis dotyczący zagadnienia obracania ekranu i poprawnego wyświetlania jego zawartości.
- [www.ngdc.noaa.gov/geomag/faqgeom.shtml](http://www.ngdc.noaa.gov/geomag/faqgeom.shtml) — znajdziemy tu informacje o geomagnetyzmie pochodzące od agencji NOAA.
- [www.youtube.com/watch?v=C7JQ7Rpwn2k](http://youtube.com/watch?v=C7JQ7Rpwn2k) — prezentacja Google TechTalk autorstwa Davida Sachsa, dotycząca akcelerometrów, żyroskopów i kompasów w odniesieniu do Androida.
- <http://stackoverflow.com/questions/1586658/combine-gyroscope-and-accelerometer-data> — przyjemny artykuł dotyczący łączenia odczytów pochodzących z żyroskopów i akcelerometrów w celu wykorzystania tych danych w aplikacjach.
- [www.nfc-forum.org/specs](http://www.nfc-forum.org/specs) — oficjalna strona specyfikacji technologii NFC.
- [www.slideshare.net/tdeLazzari/architecture-and-development-of-nfc-applications](http://www.slideshare.net/tdeLazzari/architecture-and-development-of-nfc-applications) — bardzo szczegółowa prezentacja autorstwa Thomasa de Lazzariego dotycząca technologii NFC.

## Podsumowanie

W niniejszym rozdziale przyjrzaliśmy się głównej architekturze czujników w Androidzie, a także funkcji komunikacji bliskiego pola — NFC. Zademonstrowaliśmy mechanizm odczytywania danych generowanych przez czujniki oraz sposoby ich przetwarzania. Teraz Czytelnik powinien być w stanie tworzyć świetne aplikacje współpracujące z nowoczesnymi urządzeniami w otaczającym świecie.



# Analiza interfejsu kontaktów

W rozdziale 4., dotyczącym dostawców treści, wymieniliśmy zalety wynikające z eksponowania danych za pomocą abstrakcji dostawcy treści. Udowodniliśmy również, że takie wyodrębnione dane są dostępne w postaci zbioru adresów URL, które inne obiekty mogą odczytywać, wysyłać do nich zapytania, a także aktualizować oraz wstawiać lub usuwać w ich wnętrzu własne informacje. Wspomniane adresy URL oraz ich kuryści stanowią podstawę interfejsu API dostawcy treści.

Jednym z takich interfejsów API dostawców treści jest interfejs kontaktów służący do pracy z danymi kontaktowymi. W Androidzie kontakty są przechowywane w bazie danych i eksponujemy je za pomocą dostawcy treści, którego uprawnienie rozpoznaje się od segmentu:

```
content://com.android.contacts
```

Dokumentacja zestawu Android SDK wymienia wiele różnych kontraktów zawieranych przez tego dostawcę kontaktów za pomocą różnorodnych interfejsów i klas Java, które są przechowywane w pakiecie:

```
android.provider.ContactsContract
```

W czasie tworzenia aplikacji natrafimy na wiele klas, pomocnych w wysyłaniu zapytań, odczytywaniu, aktualizowaniu oraz wstawianiu kontaktów do lub z bazy kontaktów, których nadzorem kontekstem jest `ContactsContract`. Podstawowa dokumentacja omawiająca zastosowanie interfejsu API kontaktów jest dostępna na stronie systemu Android:

<http://developer.android.com/resources/articles/contacts.html>

Główny punkt wejściowy interfejsu, czyli `ContactsContract`, nosi właściwą nazwę, ponieważ klasa ta definiuje kontrakt pomiędzy klientami kontaktów a dostawcą i zabezpieczeniem bazy kontaktów.

W tym rozdziale omówimy pojęcie kontraktu dość szczegółowo, nie wymienimy jednak każdego najdrobniejszego detalu. Interfejs kontaktów jest bardzo rozbudowany, a jego korzenie sięgają naprawdę daleko. Gdy jednak poświęcimy mu uwagę, po kilku tygodniach zauważymy, że jego zasadnicza struktura wcale nie jest taka skomplikowana. Właśnie na tej strukturze chcielibyśmy się skupić najbardziej i wyjaśnić jej podstawowe mechanizmy, co umożliwi Czytelnikowi jej zrozumienie w czasie potrzebnym na przeczytanie rozdziału.

## Koncepcja konta

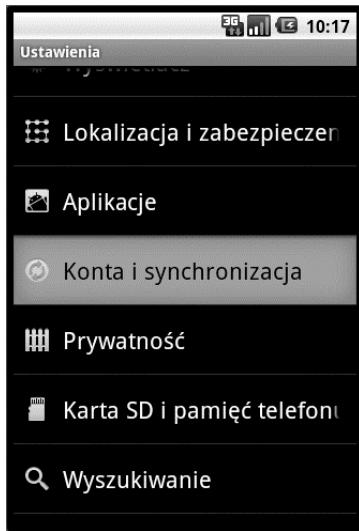
Wszystkie kontakty w systemie Android działają w kontekście konta. Czym jest konto? Jeśli na przykład korzystamy ze skrzynki pocztowej umieszczonej na serwerze Google, to znaczy, że posiadamy konto Google. Jeżeli umieściliśmy swoje dane w serwisie Facebook i jesteśmy jego użytkownikami, staliśmy się posiadaczami konta Facebook.

Nawet jeśli korzystamy tylko z poczty e-mail na serwerze Google, te same nazwa użytkownika i hasło dają nam dostęp do pozostałych usług tej firmy, zatem nasze konto pocztowe nie jest ograniczone wyłącznie do skrzynki pocztowej. Jednak niektóre rodzaje kont są ograniczone do jednego rodzaju usługi, czego przykładem jest konto pocztowe typu POP (ang. *Post Office Protocol* — protokół węzłów pocztowych). W przypadku urządzenia mobilnego możemy zarejestrować wiele różnych usług opartych na korzystaniu z konta.

Część z takich kont, na przykład konta Google, Facebook lub firmowe konto Microsoft Exchange, możemy ustanowić z poziomu widoku *Konta i synchronizacja*, dostępnego w aplikacji *Ustawienia*. Więcej informacji dotyczących kont znajdziemy w instrukcji użytkownika systemu Android. W podrozdziale „Odnośniki” zamieściliśmy do niej adres URL.

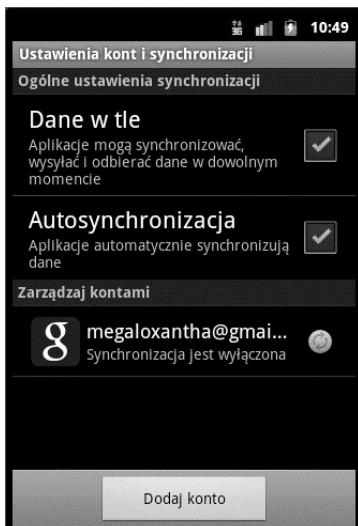
## Szybki przegląd ekranów związanych z kontami

Aby ułatwić zrozumienie natury kontaktów, przejrzyjmy najpierw kilka ekranów z nimi związanych, które znajdziemy w emulatorze. Rozpoczniemy od ekranu aplikacji *Ustawienia*, pokazanego na rysunku 27.1.



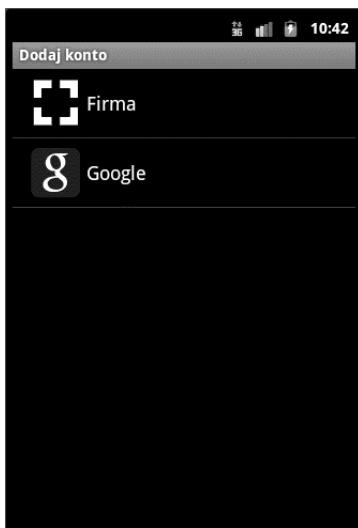
Rysunek 27.1. Wywoływanie widoku ustawień Konta i synchronizacja

Po zaznaczeniu elementu *Konta i synchronizacja* zostanie wyświetlony ekran *Ustawienia kont i synchronizacji*, który widzimy na rysunku 27.2. Znajdziemy tutaj, obok kilku opcji związanych z kontami, również listę kont powiązanych z urządzeniem.



Rysunek 27.2. Ustawienia kont i synchronizacji

Na rysunku 27.2 interesuje nas głównie lista dostępnych kont. W celach ćwiczeniowych kliknijmy przycisk *Dodaj konto*, dzięki czemu ujrzymy listę dostępnych kont, które możemy skonfigurować lub dodać (rysunek 27.3).

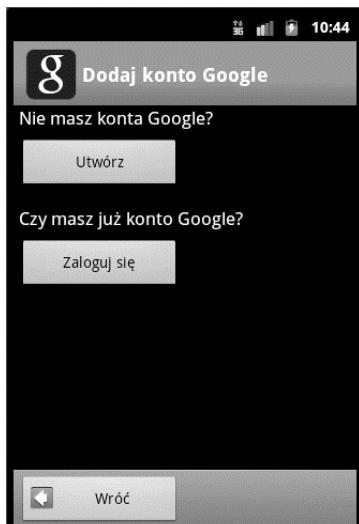


Rysunek 27.3. Lista kont, które możemy skonfigurować

Lista ta będzie miała różną zawartość w zależności od rodzaju urządzenia oraz dostępnych elementów. Na rysunku 27.3 zaprezentowaliśmy listę dostępnych kont dla wersji 2.3 Androida zainstalowanej na emulatorze, gdzie docelowy jest 9. poziom interfejsów Google API. Jeżeli pobrano samą podstawową wersję zestawu SDK, nie będzie można wybrać interfejsów Google API

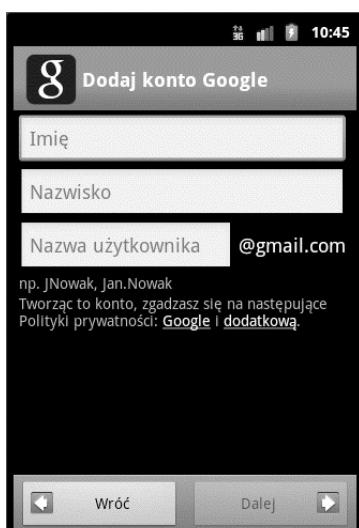
dla emulatora, zatem nie będzie można również skonfigurować konta Google, które znajduje się na liście widocznej na rysunku 27.3. Oznacza to również, że lista dostępnych kont zależy od wersji systemu Android, producenta urządzenia oraz operatora sieci lub dostawcy usług.

Ponadto, w zależności od dostawcy, liczba kont i rodzaje pól wymaganych do ich konfiguracji mogą się różnić. Jeśli na przykład wybierzemy w emulatorze konto Google, otrzymamy możliwość utworzenia nowego konta lub zalogowania się do już istniejącego (rysunek 27.4).



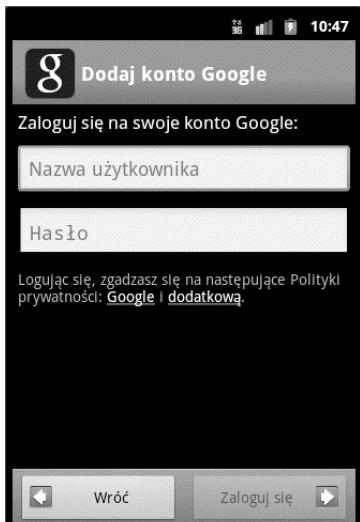
Rysunek 27.4. Dodawanie konta Google

Jeśli klikniemy przycisk *Utwórz*, pojawią się pola wymagane do założenia nowego konta Google, co zostało ukazane na rysunku 27.5.



Rysunek 27.5. Tworzenie konta Google

Na rysunku 27.5 widzimy pola wymagane do utworzenia nowego konta Google, jeśli użytkownik jeszcze go nie posiada. Jak już stwierdziliśmy, liczba i treść pól mogą się różnić w zależności od rodzaju konta. Teraz pokażemy, w jaki sposób wprowadzić ustawienia dla już istniejącego konta Google. W tym przypadku cały proces konfiguracji ogranicza się do zalogowania się na swoje konto, tak jak widać na rysunku 27.6.



**Rysunek 27.6.** Logowanie się na istniejące konto Google

Skoro zademonstrowaliśmy już podstawy dotyczące kont oraz sposób ich umieszczania w urządzeniu, wyjaśnijmy, dlaczego konta pełnią taką ważną rolę dla kontaktów.

## Związek pomiędzy kontami a kontaktami

Kontakty zarządzane przez użytkownika są powiązane z określonym kontem. Inaczej mówiąc, każde zarejestrowane na urządzeniu konto może przechowywać dużą liczbę związanych z nim kontaktów. Konto jest właścicielem zbioru kontaktów albo jest ono nadzorcze w stosunku do danego kontaktu. Równie dobrze konto może nie zawierać żadnego kontaktu.

Konto jest definiowane za pomocą dwóch ciągów znaków: jego nazwy oraz typu. W przypadku konta Google mamy do czynienia z nazwą użytkownika skrzynki pocztowej Gmail, a typem konta jest com.google. Oczywiście, typ konta musi być niepowtarzalny w obrębie całego urządzenia. W zakresie danego typu konta jego nazwa również musi być jedyna w swoim rodzaju. Typ i nazwa tworzą razem konto, a natychmiast po jego utworzeniu możemy zająć się wstawianiem do niego kontaktów.

## Wyliczanie kont

Zasadniczo interfejs kontaktów obsługuje kontakty przechowywane wewnątrz różnych kont. Samo tworzenie kont zachodzi poza interfejsem kontaktów, zatem opis możliwości związanych z pisaniem własnych dostawców kont oraz synchronizowania z nimi kontaktów wykracza poza zakres tego rozdziału. Sam proces konfiguracji kont jest nieistotny dla niniejszego rozdziału. Jeżeli jednak chcemy dodać kontakt lub listę kontaktów, musimy wiedzieć, jakie konta są dostępne

w urządzeniu. Możemy zastosować kod z listingu 27.1 do wyświetlenia rodzajów kont oraz ich wymaganych właściwości (nazwa i typ konta). Kod z listingu 27.1 generuje nazwy i typy kont w zależności od zmiennego kontekstu.

#### **Listing 27.1.** Kod umożliwiający wyświetlenie listy kont

---

```
public void listAccounts(Context ctx)
{
    AccountManager am = AccountManager.get(ctx);
    Account[] accounts = am.getAccounts();
    for(Account ac: accounts)
    {
        String acname=ac.name;
        String actype = ac.type;
        Log.d("accountInfo", acname + ":" + actype);
    }
}
```

---

Oczywiście, aby uruchomić kod z listingu 27.1, w pliku manifeście musi zostać umieszczone odpowiednie uprawnienie, widoczne na listingu 27.2.

#### **Listing 27.2.** Uprawnienie pozwalające na odczytywanie zawartości kont

---

```
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
```

---

Kod z listingu 27.1 spowoduje wyświetlenie mniej więcej następującej informacji:

---

Adres-e-mail-serwisu-google:com.google

---

W tym przypadku przyjęliśmy, że posiadamy skonfigurowane tylko jedno konto (Google). Jeżeli jest ich więcej, wszystkie zostaną wyświetcone w podobny sposób.

Zanim zajmiemy się bardziej szczegółowo kontaktami, zastanówmy się, w jaki sposób użytkownicy tworzą kontakty za pomocą aplikacji fabrycznej umieszczonej w systemie Android.

## Aplikacja Kontakty

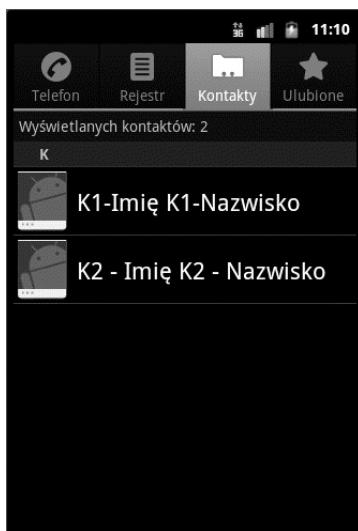
Jeżeli producent urządzenia, na przykład Motorola, lub operator (przykładowo Verizon<sup>1</sup>) nie przewidzieli własnej aplikacji zarządzającej kontaktami, z pomocą przychodzi system Android i jego domyślna aplikacja. Znajdziemy ją bez trudu na liście aplikacji dostępnych w urządzeniu, jej dokumentacja również została zamieszczona w instrukcji obsługi systemu Android.

## Wyświetlanie kontaktów

Po uruchomieniu aplikacji Kontakty pierwszym z wyświetlonych ekranów będzie lista kontaktów (rysunek 27.7). Zasadniczo kontaktem są dane dotyczące osoby, którą znamy w kontekście konta, np. Gmail. Jeżeli posiadamy wiele kont, lista z rysunku 27.7 zostanie wypełniona pochodzącyymi z nich kontaktami. Spoglądając na ten ekran, nie dowiemy się, z którym kontem jest

---

<sup>1</sup> Verizon Communications, Inc. — amerykański dostawca usług telekomunikacyjnych — przyp. red.



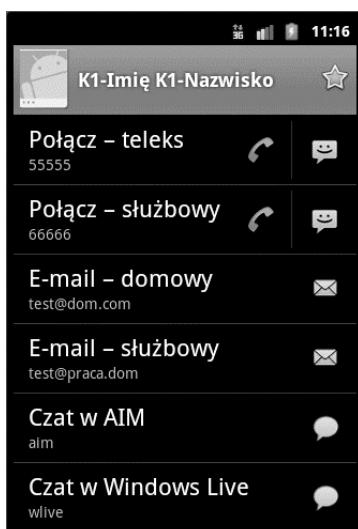
**Rysunek 27.7.** Wyświetlanie zebranych kontaktów

powiązany dany kontakt. System postara się nie powielać takich samych kontaktów pochodzących z różnych kont, chyba że zostanie to jawnie dozwolone. W następnym podrozdziale zajmiemy się tą heurystyką „podobnych kontaktów”.

Odnosząc się do sytuacji z rysunku 27.7, założyliśmy, że są dostępne dwa kontakty, które są alfabetycznie uszeregowane.

## Wyświetlanie szczegółów kontaktu

Jeżeli klikniemy jeden z kontaktów widocznych na ekranie z rysunku 27.7, zostaną wyświetlane jego szczegóły, widoczne na rysunku 27.8.



**Rysunek 27.8.** Szczegóły kontaktu

Na rysunku 27.8 zaprezentowaliśmy różne rodzaje informacji, przechowywane przez kontakt. Widzimy na nim również, jakie działania może przeprowadzić aplikacja zarządzająca kontaktami na danym kontakcie w zależności od liczby wypełnionych w nim pól. W przypadku niektórych pól możemy wykonać połączenie telefoniczne i wysłać wiadomość tekstową, a inne pozwalają na wysłanie wiadomości e-mail lub rozmowę przez komunikator internetowy.

## Edytowanie szczegółów kontaktu

Przyjrzyjmy się teraz, w jaki sposób możemy edytować (lub utworzyć nowy) kontakt zaprezentowany na rysunku 27.8. Dokonujemy tego poprzez wcisnięcie przycisku *Menu* i wybranie opcji *Edytuj kontakt* lub *Nowy kontakt*. Zostanie wyświetlony ekran zilustrowany na rysunku 27.9.



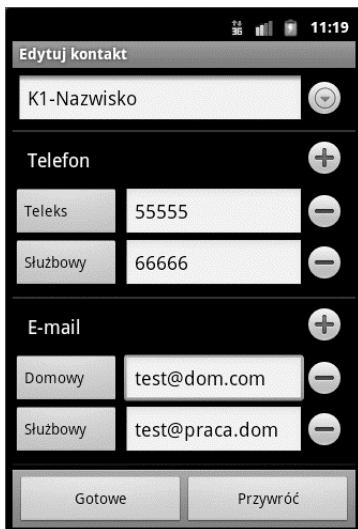
Rysunek 27.9. Edycja kontaktu

W górnej części zaprezentowanego na rysunku 27.9 ekranu *Edytuj kontakt* ujrzymy nazwę konta, w ramach którego dany kontakt jest modyfikowany lub tworzony. W przypadku omawianego kontaktu jedynym kontem jest telefon, co oznacza, że nie ma dla niego skonfigurowanego żadnego konta serwerowego (np. Google) oraz że mamy do czynienia z kontem lokalnym. Rzeczywiście, w bazie kontaktów nazwa i typ konta przyjmują w tym przypadku wartości null.

Firma Google usilnie zaleca utworzenie przynajmniej jednego konta na jej serwerze, zanim urządzenie pracujące pod kontrolą systemu Android zostanie uaktywnione, bez względu na to, czy korzystamy z telefonu, czy tabletu.

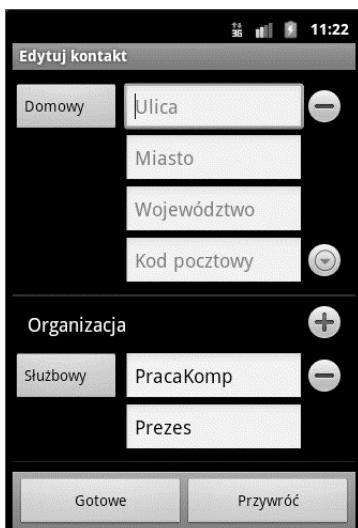
Jak jednak widać, możemy tworzyć kontakt bez konieczności łączenia go z określonym kontem, w takich zaś przypadkach ekran widoczny w momencie tworzenia takiego kontaktu wygląda tak jak na rysunku 27.9.

Patrząc na rysunek 27.9, zauważymy, że tuż za wskaźnikiem typu konta (*Tylko telefon* itd.) znajduje się miejsce na zdjęcie powiązane z kontaktem, a następnie zestaw pól. Rysunek 27.10 prezentuje następne pola, widoczne po przewinięciu ekranu.



**Rysunek 27.10.** Więcej pól edycji

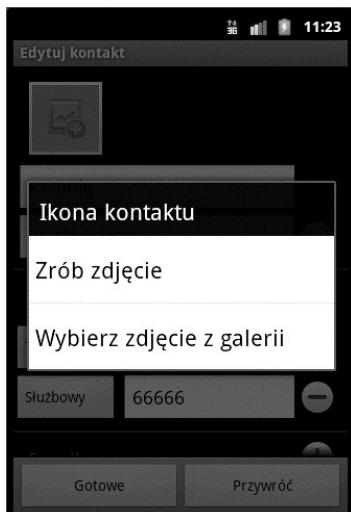
Jak widać na rysunku 27.10, istnieje możliwość przypisania do kontaktu różnych rodzajów numerów telefonów i adresów e-mail. Czytelnik zastanawia się pewnie również, czy w kontaktach możemy umieszczać własne pola zawierające niestandardowe dane (widoczne na rysunku 27.10 pola *Telefon* oraz *E-mail* stanowią znane, predefiniowane typy pól. Być może ktoś chciałby wstawić mniej oczekiwane formaty danych. To właśnie mamy na myśli, pisząc „niestandardowe”). Interfejs API kontaktów pozwala na wprowadzenie tego typu pól, co zostało zaprezentowane na rysunku 27.11, gdzie dodaliśmy dane adresowe do kontaktu.



**Rysunek 27.11.** Edytowanie niestandardowych danych kontaktowych

## Umieszczanie zdjęcia powiązanego z kontaktem

Możemy wprowadzić również zdjęcie dotyczące danego kontaktu. Na rysunku 27.12 widzimy okno ustawień zdjęcia, które zostanie otwarte po kliknięciu ukazanego na rysunku 27.9 pola zarezerwowanego na fotografię (pierwsza strona szczegółowych danych kontaktu).



Rysunek 27.12. Edycja zdjęcia kontaktu

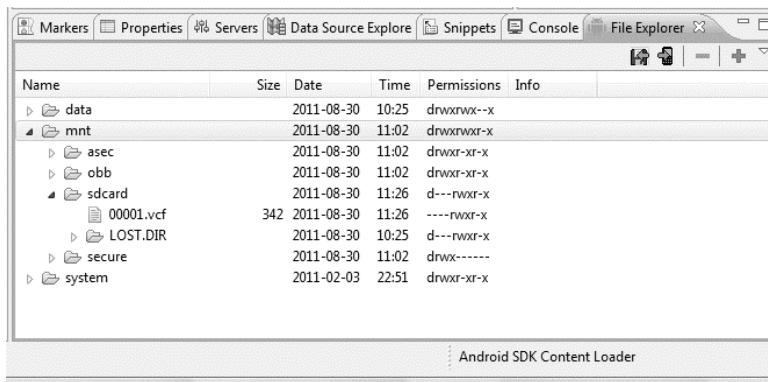
## Eksportowanie kontaktów

Zakończmy przegląd aplikacji zarządzającej kontaktami zapoznaniem się z mechanizmem eksportowania kontaktów na kartę SD. Funkcja ta pozwala nam między innymi na przeglądanie rodzajów danych przechowywanych w kontakcie oraz sprawdzenie, w jaki sposób są prezentowane w formie tekstuowej (rysunek 27.13).



Rysunek 27.13. Eksportowanie kontaktów

Po wyeksportowaniu kontaktów na kartę SD możemy przejrzeć jej zawartość za pomocą wtyczki ADT środowiska Android. Na rysunku 27.14 widzimy jeden z eksportowanych plików *.vcf* w perspektywie *File Explorer* środowiska Eclipse.



**Rysunek 27.14.** Informacje kontaktowe umieszczone na karcie SD

Możemy skopiować widoczny na rysunku 27.14 plik *.vcf* z urządzenia do stacji roboczej za pomocą ikon widocznych w prawym górnym rogu zakładki *File Explorer*. Zawartość pliku *.vcf* dla dwóch kontaktów widocznych na rysunku 27.8 będzie wyglądała tak jak zaprezentowano na listingu 27.3.

**Listing 27.3.** Eksportowane kontakty w formacie VCF

```
BEGIN:VCARD
VERSION:2.1
N:C1-Nazwisko;C1-Imię;;
FN:C1-Imię C1-Nazwisko
TEL;TLX:55555
TEL;WORK:66666
EMAIL;HOME:test@dom.com
EMAIL;WORK:test@praca.com
ORG:PracaKomp
TITLE:Prezes
ORG:Inna praca
TITLE:Prezes
URL:www.com
NOTE:Uwaga1
X-AIM:aim
X-MSN:wlive
END:VCARD

BEGIN:VCARD
VERSION:2.1
N:C2-Nazwisko;C2-Imię;;
FN:C2-Imię C2-Nazwisko
END:VCARD
```

## Różne typy danych kontaktowych

Z pomocą dotychczas prezentowanych rysunków pokazaliśmy, w jaki sposób można dodawać różne rodzaje informacji do kontaktu. Na listingu 27.4 zaprezentowaliśmy listę typów danych zdefiniowanych w interfejsie kontaktów (w nowszych wersjach systemu może się ona rozrastać; prezentujemy wersję zgodną z Androidem 2.3).

**Listing 27.4.** Standardowe typy danych kontaktowych

---

```
email  
event  
groupmembership  
im  
nickname  
note  
organization  
phone  
photo  
relation  
SipAddress  
structuredname  
structuredpostal  
website
```

---

Każdy rodzaj danych, na przykład `email` czy `structuredpostal` (przechowujący kod pocztowy), posiada własny zestaw pól. Skąd możemy wiedzieć, jaki kształt przybierają te pola? Są one zdefiniowane w pomocniczych klasach, znajdujących się w pakiecie:

`android.provider.ContactsContract.CommonDataKinds`

Dokumentacja tego pakietu jest dostępna pod adresem:

<http://developer.android.com/reference/android/provider/ContactsContract.CommonDataKinds.html>

Na przykład klasa `CommonDataKinds.Email` definiuje pola ukazane na listingu 27.5.

**Listing 27.5.** Rodzaje pól w adresie typu e-mail kontaktu

---

Adres e-mail

Typ poczty e-mail: `type_home`, `type_work`, `type_other`, `type_mobile`

Etykieta: do obsługi poczty `type_other`

---

Skoro znamy już podstawowe pojęcia i narzędzia wymagane do korzystania z kontaktów i kont, przejdźmy do właściwych szczegółów interfejsu kontaktów.

## Analiza kontaktów

Jak już stwierdziliśmy, kontakt jest przypisany do konta. Każde konto zawiera własny zbiór kontaktów. Z kolei na każdy kontakt przypada zestaw elementów danych (na przykład adres e-mail, numer telefonu, imię i nazwisko czy kod pocztowy). Co więcej, Android przedstawia

zbiorczy widok nieprzetworzonych kontaktów, które umieszcza na liście kontaktów po sprawdzeniu, czy spełniają określone kryteria. Użytkownik widzi taką zbiorczą listą kontaktów w momencie uruchomienia aplikacji Kontakty (rysunek 27.8).

Sprawdzimy teraz, w jaki sposób dane powiązane z kontaktami są przechowywane w różnych rodzajach tabel. Wyjaśnienie reguł rządzących tymi tabelami oraz powiązanymi z nimi widokami stanowi klucz do zrozumienia działania interfejsu kontaktów.

## Badanie treści bazy danych SQLite

Jednym ze sposobów zrozumienia i analizy bazodanowych tablic kontaktów jest pobranie treści bazy danych z urządzenia czy emulatora i przejrzenie jej za pomocą eksploratora SQLite.

Aby pobrać bazę kontaktów, skorzystamy z widocznej na rysunku 27.14 perspektywy *File Explorer*, gdzie przejdziemy do następującego katalogu, znajdującego się w emulatorze:

*/data/data/com.android.providers.contacts/databases*

W zależności od wersji systemu nazwa bazy danych może się nieznacznie różnić, powinna jednak brzmieć *contacts.db*, *contacts2.db* lub podobnie.

Theoretycznie wystarczy otworzyć bazę danych za pomocą odpowiedniego narzędzia. Wykrylibyśmy jednak problem związany z jej otwieraniem — większość narzędzi ulegało zawieszeniu. Kłopot polega na niestandardowych schematach zestawiania danych zdefiniowanych przez system Android dla takich działań jak porównywanie numerów telefonów.

Najwidoczniej w przypadku bazy danych SQLite niestandardowe schematy zestawiania są kompilowane jako część dystrybucji silnika SQLite. Jeżeli nie posiadamy bibliotek DLL kompilowanych wraz z dystrybucją systemu Android, eksploratory ogólnego przeznaczenia nie będą w stanie poprawnie odczytywać bazy danych. Ponieważ narzędzia te wykorzystują biblioteki DLL systemu Windows do otwierania bazy danych SQLite utworzonej w systemie Android opartym na rdzeniu Linuksa, ich działania kończą się niepowodzeniem. Poza tym wersja silnika SQLite dla systemu Windows nie posiada schematów zestawiania, które są zdefiniowane jako niezbędny element bazy kontaktów.

Trochę się nam jednak poszczęściło, gdyż w programie SQLite Explorer znalazły się błąd umożliwiający przeglądanie tabel, chociaż nie pozwala to na wyświetlanie schematu bazy danych. Możemy mieć więcej szczęścia z płatnymi aplikacjami. Jeżeli Czytelnika interesują inne alternatywy, poniżej zamieszczamy odnośnik do listy istniejących eksploratorów baz danych SQLite:

<http://www.sqlite.org/cvstrac/wiki?p=ManagementTools>

Jeżeli Czytelnik jest naprawdę dociekliwy, może dowiedzieć się więcej o schematach zestawiania z naszego artykułu *Exploring Contacts db*, dostępnego pod adresem <http://www.androidbook.com/item/3582>.

Jeśli Czytelnikowi nie uda się eksplorować bazy danych, nie wszystko jednak jest stracone, ponieważ w tym rozdziale umieściliśmy listę wszystkich najważniejszych tabel. Zaczniemy najpierw od analizy tak zwanych nieprzetworzonych kontaktów.

## Nieprzetworzone kontakty

Przypominamy, że kontakty widoczne po uruchomieniu aplikacji Kontakty są nazywane kontaktami zbiorczymi. Na każdy kontakt zbiorczy składa się zbiór tak zwanych nieprzetworzonych kontaktów. Kontakt zbiorczy stanowi jedynie widok zestawu podobnych do siebie

nieprzetworzonych kontaktów. Aby zrozumieć koncepcję kontaktów zbiorczych, najpierw musimy przeanalizować pojęcie nieprzetworzonego kontaktu oraz przechowywanych przez niego danych. Zajmiemy się więc w pierwszej kolejności nieprzetworzonymi kontaktami.

Zestaw kontaktów przypisany do konta jest w rzeczywistości nazywany zestawem nieprzetworzonych kontaktów. Każdy nieprzetworzony kontakt definiuje określony aspekt osoby znanej użytkownikowi w kontekście danego konta. W przeciwieństwie do kontaktu nieprzetworzonego kontakt zbiorczy może być dostępny poza granicami danego konta i w konsekwencji jest ogólnie wykorzystywany w urządzeniu.

Relacja pomiędzy kontem a jego zbiorem nieprzetworzonych kontaktów jest utrzymywana w tabeli nieprzetworzonych kontaktów. Listing 27.6 prezentuje strukturę tej tabeli w bazie kontaktów.

---

**Listing 27.6.** Definicja tabeli nieprzetworzonych kontaktów

```
CREATE TABLE raw_contacts
(_id INTEGER PRIMARY KEY AUTOINCREMENT,
is_restricted INTEGER DEFAULT 0,
account_name STRING DEFAULT NULL,
account_type STRING DEFAULT NULL,
sourceid TEXT,
version INTEGER NOT NULL DEFAULT 1,
dirty INTEGER NOT NULL DEFAULT 0,
deleted INTEGER NOT NULL DEFAULT 0,
contact_id INTEGER REFERENCES contacts(_id),
aggregation_mode INTEGER NOT NULL DEFAULT 0,
aggregation_needed INTEGER NOT NULL DEFAULT 1,
custom_ringtone TEXT
send_to_voicemail INTEGER NOT NULL DEFAULT 0,
times_contacted INTEGER NOT NULL DEFAULT 0,
last_time_contacted INTEGER,
starred INTEGER NOT NULL DEFAULT 0,
display_name TEXT,
display_name_alt TEXT,
display_name_source INTEGER NOT NULL DEFAULT 0,
phonetic_name TEXT,
phonetic_name_style TEXT,
sort_key TEXT COLLATE PHONEBOOK,
sort_key_alt TEXT COLLATE PHONEBOOK,
name_verified INTEGER NOT NULL DEFAULT 0,
contact_in_visible_group INTEGER NOT NULL DEFAULT 0,
sync1 TEXT, sync2 TEXT, sync3 TEXT, sync4 TEXT )
```

---

Najważniejsze pola zostały wyróżnione pogrubioną czcionką. Podobnie jak w przypadku pozostałych tabel systemu Android, znajdziemy tu kolumnę `_ID`, niepowtarzalnie definiującą nieprzetworzony kontakt. Pola `account_name` oraz `account_type` wspólnie definiują konto tego kontaktu (dokładniej nieprzetworzonego kontaktu). Pole `sourceid` wskazuje sposób określania nieprzetworzonego kontaktu w nadrzednym kontekście. Założymy na przykład, że chcemy dowiedzieć się, w jaki sposób jest zdefiniowany identyfikator nieprzetworzonego kontaktu wewnątrz konta pocztowego Google. W tym przypadku zazwyczaj pole to przechowuje identyfikator poczty e-mail użytkownika.

Kolumna `contact_id` odnosi się do kontaktu zbiorczego, którego częścią jest dany nieprzetworzony kontakt. Kontakt zbiorczy wskazuje przynajmniej jeden kontakt (lub więcej), który w istocie prezentuje tę samą osobę wykrytą na różnych kontaktach.

Pole `display_name` przechowuje wyświetlana nazwę kontaktu. Jest to przede wszystkim pole tylko do odczytu. Jego wartość jest ustanawiana przez wyzwalacze na podstawie informacji umieszczanych w tabeli danych (która zostanie omówiona w następnym punkcie).

Pola zawierające człon `sync` są wykorzystywane przez konto do synchronizowania kontaktów pomiędzy urządzeniem a kontem serwerowym, na przykład pocztą Gmail.

Chociaż do analizy tych pól wykorzystywaliśmy narzędzia obsługujące silnik SQLite, istnieje więcej sposobów na ich przeglądanie. Zalecanym rozwiązaniem jest wykorzystanie definicji klas zadeklarowanych w interfejsie `ContactsContract`. Aby przebadać kolumny należące do tabeli nieprzetworzonych kontaktów, powinniśmy przejrzeć dokumentację klasy `ContactsContract`.

→RawContact.

Rozwiązanie to posiada swoje zalety i wady. Istotną zaletą jest możliwość zapoznania się z polami, które są opublikowane i akceptowane przez zestaw Android SDK. Kolumny bazodanowe mogą być dodawane lub usuwane bez konieczności modyfikowania interfejsu publicznego. Zatem gdybyśmy chcieli bezpośrednio manipulować kolumnami bazodanowymi, możemy, ale nie musimy na nie natrafić. Z kolei korzystając z publicznych definicji tych kolumn, jesteśmy zawsze zabezpieczeni.

Należy jednak wspomnieć, że dokumentacja omawianej klasy zawiera mnóstwo innych stałych pomieszanych z nazwami kolumn; nawet my w trakcie opracowywania książki pogubiliśmy się w ich ogromie. Ta olbrzymia liczba definicji klasy daje wrażenie złożoności interfejsu API, podczas gdy w rzeczywistości 80 procent jego dokumentacji omawia stałe tych kolumn oraz umożliwiające do nich dostęp identyfikatory URI.

Gdy w następnych podrozdziałach będziemy ćwiczyć stosowanie interfejsu kontaktów, zaczniemy wykorzystywać stałe zdefiniowane w dokumentacji klasy zamiast bezpośrednich nazw kolumny. Mimo to uważamy, że bezpośrednia eksploracja tabel jest najszybszym sposobem pozwalającym na zrozumienie interfejsu kontaktów.

Zastanówmy się teraz, w jaki sposób są przechowywane dane związane z kontaktem, na przykład adres e-mail lub numer telefonu.

## Tabela danych

Jak już wspomnieliśmy podczas omawiania definicji tabeli nieprzetworzonych kontaktów, taki kontakt (w tym rozczarowującym znaczeniu) stanowi wyłącznie identyfikator wskazujący, z jakim kontem jest związany. Większość informacji stanowiących zawartość kontaktu jest przechowywana nie w tabeli nieprzetworzonych kontaktów, lecz w tabeli danych. Każdy element danych, taki jak adres e-mail lub numer telefonu, posiada własne pole w tabeli danych. Wszystkie te wiersze zawierające dane są powiązane z nieprzetworzonym kontaktem za pomocą jego identyfikatora, który stanowi jedną z kolumn tabeli danych oraz jest głównym identyfikatorem w tabeli nieprzetworzonych kontaktów.

Tabela danych składa się z szesnastu standardowych kolumn, w których może być przechowywanych sześćnaście dowolnych punktów danych danego elementu, na przykład adresu e-mail. Na listingu 27.7 widzimy strukturę tabeli danych.

**Listing 27.7.** Definicja tabeli danych kontaktów

```
CREATE TABLE data
(_id INTEGER PRIMARY KEY AUTOINCREMENT,
package_id INTEGER REFERENCES package(_id),
mimetype_id INTEGER REFERENCES mimetype(_id) NOT NULL,
raw_contact_id INTEGER REFERENCES raw_contacts(_id) NOT NULL,
is_primary INTEGER NOT NULL DEFAULT 0,
is_super_primary INTEGER NOT NULL DEFAULT 0,
data_version INTEGER NOT NULL DEFAULT 0,
data1 TEXT,data2 TEXT,data3 TEXT,data4 TEXT,data5 TEXT,
data6 TEXT,data7 TEXT,data8 TEXT,data9 TEXT,data10 TEXT,
data11 TEXT,data12 TEXT,data13 TEXT,data14 TEXT,data15 TEXT,
data_sync1 TEXT, data_sync2 TEXT, data_sync3 TEXT, data_sync4 TEXT )
```

---

Najważniejsze kolumny tabeli kontaktów zostały wyróżnione pogrubioną czcionką. Tak jak mogliśmy się spodziewać, pole `raw_contact_id` stanowi odniesienie do nieprzetworzonego kontaktu, z którym jest związana dana informacja.

Pole `mimetype_id` określa typ MIME danych wejściowych i wskazuje jeden z typów zdefiniowanych na listingu 27.4. Kolumny od `data1` do `data15` stanowią standardowe tablice przechowujące ciągi znaków, w których możemy umieszczać wszelkie niezbędne informacje, zgodne z danym typem MIME. Także tutaj pola typu sync obsługują synchronizację kontaktów. Tabela zawierająca informacje o typie MIME identyfikatorów została zaprezentowana na listingu 27.8.

**Listing 27.8.** Definicja tabeli wyszukiwania typów MIME

```
CREATE TABLE mimetypes
(_id INTEGER PRIMARY KEY AUTOINCREMENT,
mimetype TEXT NOT NULL)
```

---

Podobnie jak ma to miejsce w tabeli nieprzetworzonych danych, analiza kolumn tabeli danych jest możliwa dzięki dokumentacji pomocniczej klasy `ContactsContract.Data`.

Chociaż możemy rozpoznawać kolumny za pomocą definicji tej klasy, nie zapoznamy się w ten sposób z zawartością pól od `data1` do `data15`. W tym celu trzeba poznać definicje różnorodnych klas umieszczonych w przestrzeni nazw `ContactsContract.CommonDataKinds`.

Poniżej prezentujemy przykłady niektórych zawartych w niej klas:

- `ContactsContract.CommonDataKinds.Email`,
- `ContactsContract.CommonDataKinds.Phone`.

W istocie dla każdego typu danych wymienionego na listingu 27.4 istnieje jedna klasa ze wspomnianej przestrzeni nazw. W ostatecznym rozrachunku wszystkie elementy podległe klasy `CommonDataKinds` wskazują, które ze standardowych pól danych (`data1` – `data15`) są wykorzystywane oraz w jakim celu.

## Kontakty zbiorcze

Ostatecznie dochodzimy do wniosku, że kontakt i związane z nim dane w sposób jednoznaczny są przechowywane w tabeli nieprzetworzonych kontaktów i tabeli danych. Z drugiej strony kontakt zbiorczy jest nieco bardziej heurystyczny i już nie tak bardzo jednoznaczny.

Gdy natrafimy na kontakt, który jest taki sam dla różnych kont, chcielibyśmy, żeby jego nazwa pojawiała się na liście tylko raz. W tym celu Android umieszcza wszystkie podobne kontakty w widoku przeznaczonym tylko do odczytu. Takie zbiorcze kontakty są przechowywane w tabeli zwanej kontaktami. Aby zapełnić lub modyfikować taką tabelę kontaktów zbiorczych, Android wykorzystuje wiele wyzwalaczy wobec tabeli nieprzetworzonych kontaktów i tabeli danych.

Zanim przejdziemy do omówienia mechanizmów zbierania kontaktów, spojrzymy najpierw na definicję tabeli kontaktów (listing 27.9).

#### **Listing 27.9.** Definicja tabeli kontaktów zbiorczych

---

```
CREATE TABLE contacts
(_id INTEGER PRIMARY KEY AUTOINCREMENT,
name_raw_contact_id INTEGER REFERENCES raw_contacts(_id),
photo_id INTEGER REFERENCES data(_id),
custom_ringtone TEXT,
send_to_voicemail INTEGER NOT NULL DEFAULT 0,
times_contacted INTEGER NOT NULL DEFAULT 0,
last_time_contacted INTEGER,
starred INTEGER NOT NULL DEFAULT 0,
in_visible_group INTEGER NOT NULL DEFAULT 1,
has_phone_number INTEGER NOT NULL DEFAULT 0,
lookup TEXT,
status_update_id INTEGER REFERENCES data(_id),
single_is_restricted INTEGER NOT NULL DEFAULT 0)
```

---

Najważniejsze pola zostały oznaczone pogrubioną czcionką. Żaden klient nie aktualizuje bezpośrednio tej tabeli. Po dodaniu nieprzetworzonego kontaktu wraz z jego współlistniejącymi danymi Android sprawdza pozostałe nieprzetworzone kontakty pod kątem podobieństwa. Po wykryciu powtarzającego się wpisu jego identyfikator stanie się również identyfikatorem nowego nieprzetworzonego kontaktu. Nie zostanie wprowadzony żaden nowy wpis do tabeli kontaktów zbiorczych. W przypadku braku obecności duplikatu zostanie utworzony nowy kontakt zbiorczy, a jego identyfikator będzie przypisany również do nowego nieprzetworzonego kontaktu.

Do określenia, czy dwa nieprzetworzone kontakty są podobne, Android stosuje następujący algorytm:

1. Dwa nieprzetworzone kontakty posiadają takie same nazwy.
2. Wyrazy tworzące nazwy kontaktów są takie same, ulożone są jednak w innej kolejności: „pierwszy ostatni”, „pierwszy, ostatni” lub „ostatni, pierwszy”.
3. Rozpoznawane są zdrobnienia imion, na przykład „Robercik” jest zdrobnieniem imienia „Robert”.
4. Jeśli jeden z nieprzetworzonych kontaktów zawiera tylko dane o imieniu lub nazwisku, zostanie włączony mechanizm poszukiwania innych atrybutów, na przykład numeru telefonu lub adresu e-mail, i jeżeli będą one takie same, kontakty zostaną ze sobą powiązane.
5. Jeśli jeden z nieprzetworzonych kontaktów nie będzie zawierał imienia ani nazwiska, podobnie jak w punkcie 4., spowoduje to również włączenie procesu wyszukiwania innych atrybutów.

Ponieważ mamy tu do czynienia z metodami heurystycznymi, niektóre kontakty mogą zostać ze sobą powiązane przypadkowo. W takim wypadku aplikacje klienckie muszą zawierać mechanizm rozdzielający te kontakty. W czasie przeglądania instrukcji obsługi systemu Android dowiemy się, że domyślna aplikacja Kontakty umożliwia rozdzielanie przypadkowo połączonych kontaktów.

Możemy również uniemożliwić agregację kontaktów poprzez ustanowienie odpowiedniego trybu agregacji podczas wstawiania nieprzetworzonego kontaktu. Listing 27.10 zawiera spis dostępnych trybów agregacji.

#### **Listing 27.10.** Stałe trybu agregacji

---

```
AGGREGATION_MODE_DEFAULT  
AGGREGATION_MODE_DISABLED  
AGGREGATION_MODE_SUSPENDED
```

---

Pierwsza opcja jest oczywista; jest to domyślny tryb agregacji.

W drugim trybie (DISABLED) nieprzetworzony kontakt nie podlega procesowi agregacji. Nawet jeśli kontakt został już dołączony do tabeli kontaktów zbiorczych, zostanie z niej usunięty i otrzyma nowy identyfikator.

Po wybraniu trzeciej opcji (SUSPENDED), nawet jeśli właściwości kontaktu ulegną zmianie, co jest równoznaczne z jego odrzuceniem z bieżącego zbioru kontaktów, będzie on ciągle powiązany z jego zbiorczym odpowiednikiem.

Z powyższego akapitu wynika, że kontakt zbiorczy jest niestabilny. Założymy, że posiadamy unikatowy, nieprzetworzony kontakt, w którym umieszczone są imię i nazwisko. Jeśli dane te nie dublują się z danymi żadnego innego nieprzetworzonego kontaktu, zostanie przydzielony do oddzielnego kontaktu zbiorczego. Identyfikator takiego zbiorczego kontaktu będzie przechowywany w tabeli nieprzetworzonego kontaktu wraz z jego danymi.

Założymy, że zmienimy nazwisko w tym nieprzetworzonym kontakcie w taki sposób, że zduplicuje on zestaw innych powiązanych ze sobą kontaktów. W takim przypadku nasz zmodyfikowany, nieprzetworzony kontakt zostanie odłączony od pierwotnego kontaktu zbiorczego i przeniesiony do innego obiektu tego typu. Identyfikator pierwotnego kontaktu zbiorczego zostanie natomiast całkowicie porzucony i nie będzie już więcej pasował do żadnego kontaktu, ponieważ w rzeczywistości będzie to sam identyfikator bez żadnych dodatkowych danych.

Zatem kontakt zbiorczy jest niestabilny. Przechowywanie przez dłuższy czas identyfikatora takiego zbiorczego kontaktu nie ma wielkiego sensu.

Android oferuje pewne wyjście z tej kłopotliwej sytuacji, mianowicie pozwala na stosowanie pola `lookup` w tabelach kontaktów zbiorczych.

Takie pole wyszukiwania pozwala na agregację (konkatenację) konta z niepowtarzalnym identyfikatorem kontaktu w przypadku każdego nieprzetworzonego kontaktu. Informacja ta jest jeszcze bardziej kodyfikowana, dzięki czemu można ją wysłać w postaci adresu URL w celu odczytania identyfikatora najnowszego dołączonego kontaktu. Android wykorzystuje klucz wyszukiwania i znajduje wszelkie identyfikatory nieprzetworzonych kontaktów, które są przechowywane dla tego klucza. Następnie za pomocą algorytmu najlepszego dopasowania określa najodpowiedniejszy (lub być może nowy) identyfikator kontaktu zbiorczego.

Skoro jawnie analizujemy bazę kontaktów, przyjrzymy się dwóm bazowanym widokom, które mogą nam się przydać.

## **view\_contacts**

Pierwszym ze wspomnianych widoków jest `view_contacts`. Chociaż mamy do dyspozycji tabelę przechowującą zbiorcze kontakty (tabela kontaktów), interfejs API nie pozwala na uzyskanie bezpośredniego dostępu do tej tabeli. Zamiast tego do przeglądania kontaktów zbiorczych służy właśnie widok `view_contacts`. Gdy wysyłamy zapytanie oparte na identyfikatorze URI `ContactsContract.Contacts.CONTENT_URI`, otrzymamy kolumny, które bazują na widoku `view_contacts`. Definicja tego widoku została umieszczona na listingu 27.11.

**Listing 27.11.** Widok umożliwiający odczytywanie kontaktów zbiorczych

---

```
CREATE VIEW view_contacts AS

SELECT contacts._id AS _id,
       contacts.custom_ringtone AS custom_ringtone,
       name_raw_contact.display_name_source AS display_name_source,
       name_raw_contact.display_name AS display_name,
       name_raw_contact.display_name_alt AS display_name_alt,
       name_raw_contact.phonetic_name AS phonetic_name,
       name_raw_contact.phonetic_name_style AS phonetic_name_style,
       name_raw_contact.sort_key AS sort_key,
       name_raw_contact.sort_key_alt AS sort_key_alt,
       name_raw_contact.contact_in_visible_group AS in_visible_group,
       has_phone_number,
       lookup,
       photo_id,
       contacts.last_time_contacted AS last_time_contacted,
       contacts.send_to voicemail AS send_to_voicemail,
       contacts.starred AS starred,
       contacts.times_contacted AS times_contacted, status_update_id

FROM contacts JOIN raw_contacts AS name_raw_contact
ON(name_raw_contact_id=name_raw_contact._id)
```

---

Zwróćmy uwagę, że widok ten łączy tabelę kontaktów z tabelą nieprzetworzonych kontaktów, a wspólnym mianownikiem jest tu identyfikator zbiorczego kontaktu.

## **contact\_entities\_view**

Kolejny przydatny widok łączy tabelę nieprzetworzonych kontaktów z tabelą danych. Dzięki niemu możemy odczytywać jednocześnie wszystkie elementy danych określonego, nieprzetworzonego kontaktu, a nawet dane będące częścią wielu nieprzetworzonych kontaktów składających się na jeden kontakt zbiorczy. Na listingu 27.12 widzimy definicję widoku tych encji.

**Listing 27.12.** Widok encji kontaktów

---

```
CREATE VIEW contact_entities_view AS

SELECT raw_contacts.account_name AS account_name,
       raw_contacts.account_type AS account_type,
       raw_contacts.sourceid AS sourceid,
       raw_contacts.version AS version,
       raw_contacts.dirty AS dirty,
```

```
raw_contacts.deleted AS deleted,
raw_contacts.name_verified AS name_verified,
package AS res_package,
contact_id,
raw_contacts.sync1 AS sync1,
raw_contacts.sync2 AS sync2,
raw_contacts.sync3 AS sync3,
raw_contacts.sync4 AS sync4,
mimetype, data1, data2, data3, data4, data5, data6, data7, data8,
data9, data10, data11, data12, data13, data14, data15,
data_sync1, data_sync2, data_sync3, data_sync4,
raw_contacts._id AS _id,
is_primary, is_super_primary,
data_version,
data._id AS data_id,
raw_contacts.starred AS starred,
raw_contacts.is_restricted AS is_restricted,
groups.sourceid AS group_sourceid

FROM raw_contacts LEFT OUTER JOIN data
    ON (data.raw_contact_id=raw_contacts._id)
LEFT OUTER JOIN packages
    ON (data.package_id=packages._id)
LEFT OUTER JOIN mimetypes
    ON (data.mimetype_id=mimetypes._id)
LEFT OUTER JOIN groups
    ON (mimetypes.mimetype='vnd.android.cursor.item/group_membership'
        AND groups._id=data.data1)
```

---

Identyfikatory URI wymagane do uzyskania dostępu do tego widoku znajdziemy w klasie `ContactsContract.RawContacts.RawContactsEntity`.

## Praca z interfejsem kontaktów

Do tej pory omawialiśmy podstawowy mechanizm działania interfejsu kontaktów poprzez analizowanie jego tabel i widoków. Korzystając ze zdobytej wiedzy, utworzymy teraz kilka przykładowych programów. Chociaż możemy bez problemu posiąkać się kodami zawartymi na listinguach, na końcu rozdziału umieściliśmy odnośnik do plików zawierających gotowe projekty.

### Eksploracja kont

Rozpoczniemy ćwiczenia od napisania programu wyświetlającego listę dostępnych kont.

Do tego zadania będą nam potrzebne następujące pliki:

- `TestContactsDriverActivity.java` — główna aktywność sterująca, o której będziemy wspominać kilkakrotnie w dalszej części rozdziału. Aktywność ta zawiera zestaw elementów menu przywołujących poszczególne przykłady.
- `DebugActivity.java` — podstawowa klasa aktywności sterującej, ukrywająca kilka szczegółów implementacji, których znajomość nie jest wymagana do zrozumienia koncepcji interfejsu kontaktów.

- *debug\_activity\_layout.xml* — plik układu graficznego wymagany przez aktywność debugującą, przechowywany w podkatalogu */res/layout*.
- *AccountFunctionTester.java* — klasa Java, która po kliknięciu elementu menu wyświetla (poprzez aktywność sterującą) listę dostępnych kont w emulatorze lub urządzeniu.
- *BaseTester.java* — bazowa klasa aplikacji *AccountsFunctionTester*, ukrywająca szczegóły koordynacji pomiędzy główną aktywnością sterującą a pozostałymi funkcjami testowymi (każdy z prezentowanych przykładów został zaimplementowany w postaci oddzielnej funkcji testowej, dzięki czemu kod odzwierciedlający każdą z omawianych koncepcji został umieszczony w osobnym pliku).
- *IReportBack.java* — interfejs implementowany przez klasę *DebugActivity*, przekazywany klasie *BaseTester*. Interfejs ten pozwala dziedziczącym funkcjom testowym na wyświetlanie raportów lub komunikatów debuggera na ekranie za pomocą aktywności *DebugActivity*.
- *main\_menu.xml* — plik menu obsługujący wszystkie demonstrowane przez nas przykłady.
- *AndroidManifest.xml* — niezbędny plik manifest.

Zaprezentujemy teraz po kolej każdy z wymienionych plików. Rozpoczniemy od pliku menu.

## Plik menu

Plik menu z listingu 27.13 musi nosić nazwę *main\_menu.xml* oraz zostać umieszczony w podkatalogu *res/menu* naszego projektu.

**Listing 27.13.** Plik głównego menu naszego projektu

---

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">
        <item android:id="@+id/menu_show_accounts"
              android:title="Konta" />

        <item android:id="@+id/menu_da_clear"
              android:title="wyczyść" />
    </group>
</menu>
```

---

Na tym etapie umieszczamy w pliku tylko dwa elementy menu. W trakcie tworzenia dalszych przykładów będziemy wstawać do niego kolejne obiekty. Pierwszy element menu pozwala na wyświetlanie listy dostępnych kont, drugą opcję jest natomiast przydatna funkcjonalność ogólnego przeznaczenia, służąca do usuwania komunikatów debugowania lub informacji pochodzących z testowej aktywności sterującej.

## Pliki związane z funkcjami testującymi koncepcje kont

Po umieszczeniu pliku menu we właściwym miejscu przyjrzymy się plikom związanym z implementacją kodu, które będą wywoływanie w odpowiedzi na kliknięcie elementu menu *Konta* z listingu 27.13.

### IReportBack.java

Pierwszym z tego typu plików jest *IReportBack.java*, zaprezentowany na listingu 27.14.

**Listing 27.14.** IReportBack.java

```
//IReportBack.java
public interface IReportBack
{
    public void reportBack(String tag, String message);
    public void reportTransient(String tag, String message);
}
```

---

Interfejs ten jest kontraktem dla dziedziczących klientów, umożliwiającym im wysyłanie komunikatów informacyjnych oraz debugowania. Miejsce i sposób wyświetlania tych komunikatów nie należą do zadań klientów.

**BaseTester.java**

Wszystkie funkcje testowe będą posiadały dostęp do interfejsu *IReportBack.java*, dzięki czemu będą mogły generować komunikaty po ich wywołaniu za pomocą elementu menu. Gwarantuje nam to bazowa klasa dla wszystkich prezentowanych funkcji, zwana *BaseTester*. Jej kod źródłowy został zademonstrowany na listingu 27.15.

**Listing 27.15.** Kod źródłowy klasy BaseTester

```
public class BaseTester
{
    protected IReportBack mReportTo;
    protected Context mContext;
    public BaseTester(Context ctx, IReportBack target)
    {
        mReportTo = target;
        mContext = ctx;
    }
}
```

---

Klasa *BaseTester* przechowuje interfejs *IReportBack* oraz odniesienie do kontekstu (zazwyczaj jest nim nadrędzona klasa sterująca). Te dwie zmienne są wykorzystywane przez pochodne funkcje testowe.

**AccountsFunctionTester.java**

Zaproponujemy teraz pierwszą z omawianych funkcji testowych, *AccountsFunctionTester*, której kod umieszczono na listingu 27.16.

**Listing 27.16.** Klasa AccountsFunctionTester

```
public class AccountsFunctionTester extends BaseTester
{
    private static String tag = "tc>";
    public AccountsFunctionTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }
    public void testAccounts()
    {
```

```
    AccountManager am = AccountManager.get(this.mContext);
    Account[] accounts = am.getAccounts();
    for(Account ac: accounts)
    {
        String acname=ac.name;
        String actype = ac.type;
        this.mReportTo.reportBack(tag,acname + ":" + actype);
    }
}
```

Kod widoczny na listingu 27.16 jest raczej nieskomplikowany. Na początku rozdziału omówiliśmy zagadnienie kont oraz mechanizm wyświetlania ich w postaci listy. Kod z listingu 27.16 pobiera jedynie nazwę oraz typ każdego konta, a następnie wywołuje interfejs raportujący, dzięki któremu zostaną wyświetlane wyniki. Dopóki istnieje aktywność sterująca, która może wywołać metodę `testAccounts()`, powyższy kod może przekazywać nazwę i typ konta. Zastanówmy się teraz nad klasami związanymi z aktywnością sterującą.

## **Klasy aktywności sterujących**

Zajmiemy się najpierw podstawową klasą aktywności sterującej. Aktywność ta wykonuje następujące zadania:

- Zapewnienie widoku tekstowego, w którym będą wyświetlane komunikaty. W tym celu będzie wykorzystywany układ graficzny `debug_activity_layout`.
  - Wprowadzenie menu umożliwiającego wywoływanie poszczególnych funkcji testowych. Aktywność sterująca będzie przyjmowała wartości identyfikatora zasobu menu, otrzymywanego (poprzez konstruktor) z pochodnych klas. Zakładamy następnie, że istnieje predefiniowany element menu `menu_da_clear`, czyszczący zdefiniowany w pliku układu graficznego widok tekstowy. Ta klasa bazowa wyświetla również nazwę zaznaczonego elementu menu w polu tekstowym układu graficznego debuggera.

Skoro już znamy przeznaczenie tej aktywności, możemy przyjrzeć się plikowi `DebugActivity.java`, zaprezentowanemu na listingu 27.17.

## DebugActivity.java

**Listing 27.17.** Definicja klasy DebugActivity

```
public abstract class DebugActivity extends Activity
implements IReportBack
{
    //Najpierw spełniamy wymagania pochodnych klas
    protected abstract boolean onMenuItemSelected(MenuItem item);

    //Zmienne prywatne, konfigurowane za pomocą konstruktora
    private static String tag=null;
    private int menuId = 0;

    public DebugActivity(int inMenuId, String inTag)
    {
        tag = inTag;
        menuId = inMenuId;
    }
}
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.debug_activity_layout);
}
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(menuId, menu);
    return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item){
    appendMenuItemText(item);
    if (item.getItemId() == R.id.menu_da_clear){
        this.emptyText();
        return true;
    }
    return onOptionsItemSelected(item);
}
private TextView getTextView(){
    return (TextView)this.findViewById(R.id.text1);
}
protected void appendMenuItemText(MenuItem menuItem){
    String title = menuItem.getTitle().toString();
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + title);
}
protected void emptyText(){
    TextView tv = getTextView();
    tv.setText("");
}
private void appendText(String s){
    TextView tv = getTextView();
    tv.setText(tv.getText() + "\n" + s);
    Log.d(tag,s);
}
public void reportBack(String tag, String message)
{
    this.appendText(tag + ":" + message);
    Log.d(tag,message);
}
public void reportTransient(String tag, String message)
{
    String s = tag + ":" + message;
    Toast mToast = Toast.makeText(this, s, Toast.LENGTH_SHORT);
    mToast.show();
    reportBack(tag,message);
    Log.d(tag,message);
}
```

---

Oprócz metod umożliwiających wyświetlanie komunikatów w widoku tekstowym mamy tu do czynienia z metodą `reportTransient()` interfejsu `IReportBack`, który służy do wyświetlania informacji za pomocą kontrolki `Toast`.

### **debug\_layout\_activity.xml**

Widoczny na listingu 27.18 plik *debug\_layout\_activity.xml* musi zostać umieszczony w podkatalogu */res/layout*.

**Listing 27.18.** Plik układu graficznego debuggera — *debug\_layout\_activity.xml*

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/text1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Tutaj pojawia się tekst debugowania"
    />
</LinearLayout>
```

---

### **TestContactsDriverActivity.java**

Listing 27.19 prezentuje główną aktywność sterującą, która koordynuje działanie poszczególnych elementów menu z wywoływaniem odpowiednich metod, umieszczonych w funkcjach testowych.

**Listing 27.19.** Główna aktywność sterująca

---

```
public class TestContactsDriverActivity
extends DebugActivity
implements IReportBack
{
    public static final String tag="Test kontaktów";
    AccountsFunctionTester accountsFunctionTester = null;

    public TestContactsDriverActivity()
    {
        super(R.menu.main_menu,tag);
        accountsFunctionTester = new AccountsFunctionTester(this,this);
    }
    protected boolean onMenuItemSelected(MenuItem item)
    {
        Log.d(tag,item.getTitle().toString());
        if (item.getItemId() == R.id.menu_show_accounts)
        {
            accountsFunctionTester.testAccounts();
            return true;
        }
        return true;
    }
}
```

---

Kod tej aktywności sterującej jest czytelny i przejrzysty, ponieważ umieściliśmy większość jej funkcji w klasie bazowej.

Pierwszą sprawą, którą należy zauważyc na listingu 27.19, jest sposób przekazywania zasobu menu zdefiniowanego na listingu 27.13 (*main\_menu.xml*) do bazowej aktywności debugera. Aktywność ta łączy następnie menu w całość.

Drugim mechanizmem wartym zaobserwowania jest sposób wykorzystywania funkcji testowych przez aktywność sterującą. W kodzie z listingu 27.19 zademonstrowaliśmy jedynie funkcję testową dla kont. Wraz z postępami prac nad projektem będziemy dodawać kolejne funkcje testowe. Sposób ich stosowania jest zawsze taki sam.

### **Plik manifest**

Na listingu 27.20 zamieściliśmy zawartość pliku manifestu i tym samym kończymy prezentowanie wszystkich niezbędnych plików.

---

**Listing 27.20.** Plik manifest przykładowego programu

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.contacts"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"
        android:label="Test kontaktów">
        <activity android:name=".TestContactsDriverActivity"
            android:label="Test kontaktów">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="5" />
    <uses-permission android:name="android.permission.GET_ACCOUNTS"/>
</manifest>
```

---

### **Uruchomienie programu**

Listing 27.21 zawiera spis plików wymaganych do skompilowania i uruchomienia naszego prostego przykładu.

---

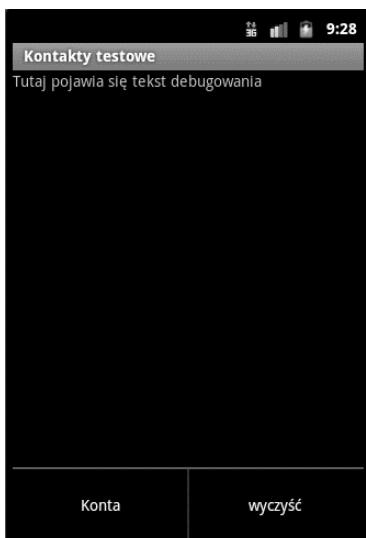
**Listing 27.21.** Kompletny spis plików tworzących pierwszą aplikację testową

---

```
IReportBack.java
BaseTester.java
AccountsFunctionTester.java
DebugActivity.java
TestContactsDriverActivity.java
/res/menu/main_menu.xml
/res/layout/debug_layout_activity.xml
AndroidManifest.xml
```

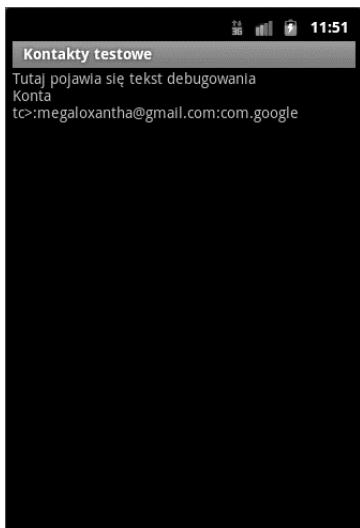
---

Jeśli po skompilowaniu i uruchomieniu kodu zawartego w tych plikach klikniemy element menu znajdujący się w głównej aktywności sterującej, zobaczymy ekran zilustrowany na rysunku 27.15.



Rysunek 27.15. Główna aktywność sterująca z dołączonym menu

Menu pokazane na rysunku 27.15 zawiera dwie opcje. Opcja *wyczyść* stanowi standardowy element menu, zdefiniowany w bazowej klasie aktywności debugowania, za pomocą którego wyzerujemy zawartość widoku tekstowego. Opcja *Konta* spowoduje wyświetlenie listy kont dostępnych w urządzeniu. Przekonajmy się, co się stanie po jego kliknięciu. Pojawi się ekran widoczny na rysunku 27.16.



Rysunek 27.16. Główna aktywność sterująca, prezentująca spis dostępnych kontaktów

Testowana przez nas wersja emulatora zawierała tylko jedną konfigurację konta — firmy Google, dlatego jego nazwa została wyświetlona przez naszą aplikację.

## Badanie kontaktów zbiorczych

W kolejnym przykładowym programie zaprezentujemy rozwiązanie pozwalające na badanie kontaktów zbiorczych. Zademonstrujemy w nim trzy kwestie dotyczące tego rodzaju kontaktów:

- Pokażemy, jak przejrzeć wszystkie wypełnione pola poprzez wykorzystanie identyfikatora URI, dzięki któremu można odczytać zawartość kontaktów zbiorczych.
- Wyświetlimy spis wszystkich zbiorczych kontaktów.
- Przedstawimy wszystkie pola przekazywane przez kursor na podstawie identyfikatora URI wyszukiwania.

W celu odczytywania kontaktów musimy umieścić następujące uprawnienie w pliku manifeście (jego kod pokazano na listingu 27.20):

`android.permission.READ_CONTACTS`

Do przetestowania tego przykładu wymagane też będą następujące nowe pliki (obok plików utworzonych wcześniej):

- `Utils.java`,
- `URIFunctionTester.java`,
- `AggregatedContactFunctionTester.java`,
- `AggregatedContact.java`.

Każdy z tych plików zostanie omówiony w dalszej części rozdziału.

Musimy także zmodyfikować następujące pliki, będące częścią poprzedniego przykładu:

- `main_menu.xml`,
- `TestContactsDriverActivity.java`.

Nieco dalej w tym podrozdziale wskażemy, jakie zmiany należy wprowadzić do wymienionych plików.

Ponieważ w omawianej przez nas funkcji pojawiają się dostawcy treści, identyfikatory URI oraz kursorы, zebraliśmy kilka metod użytkowych w pliku `Utils.java`, widocznym na listingu 27.22.

---

### Listing 27.22. Funkcje użytkowe pozwalające na pracę z kursorami

---

```
public class Utils
{
    public static String getColumnValue(Cursor cc, String cname)
    {
        int i = cc.getColumnIndex(cname);
        return cc.getString(i);
    }

    protected static String getCursorColumnNames(Cursor c)
    {
        int count = c.getColumnCount();
        StringBuffer cnamesBuffer = new StringBuffer();
        for (int i = 0; i < count; i++)
        {
            cnamesBuffer.append(cc.getColumnName(i));
            if (i < count - 1)
                cnamesBuffer.append(",");
        }
        return cnamesBuffer.toString();
    }
}
```

---

```

        for (int i=0;i<count;i++)
    {
        String cname = c.getColumnName(i);
        cnamesBuffer.append(cname).append(' ');
    }
    return cnamesBuffer.toString();
}
}

```

---

Pierwsza funkcja, `getColumnName()`, powraca z wartością kolumny, pobierając jej nazwę z bieżącego wiersza kursora. Bez względu na podstawowy typ danych kolumny funkcja ta przekazuje tę wartość w postaci ciągu znaków.

Druga funkcja jest bardzo przydatna. Pobiera ona dowolny kursor i przekazuje osobną listę wszystkich jego kolumn. Jest to użyteczne zwłaszcza w przypadku badania nowych identyfikatorów URI pod kątem rodzajów pól, które określają. Chociaż można udokumentować te kolumny w kodzie Java, wspomniana metoda ich odkrywania w czasie działania aplikacji może się przydać w pewnych sytuacjach.

Ponieważ ten i następne przykłady wykorzystują koncepcję wysyłania identyfikatora URI i odbierania kursora za pomocą aktywności, umieściliśmy funkcje realizujące te zadania w bazowej klasie `URIFunctionTester`. Na listingu 27.23 zamieściliśmy kod źródłowy tej klasy, po czym opisaliśmy każdą dostępną w niej metodę.

#### **Listing 27.23.** Klasa bazowa, umożliwiająca analizowanie funkcji związanych z identyfikatorami URI

---

```

public class URIFunctionTester extends BaseTester
{
    protected static String tag = "tc>";
    public URIFunctionTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }
    protected Cursor getACursor(String uri,String clause)
    {
        // Uruchamia kwerendę
        Activity a = (Activity)this.mContext;
        return a.managedQuery(Uri.parse(uri), null, clause, null, null);
    }

    protected Cursor getACursor(Uri uri,String clause)
    {
        // Uruchamia kwerendę
        Activity a = (Activity)this.mContext;
        return a.managedQuery(uri, null, clause, null, null);
    }
    protected void printCursorColumnNames(Cursor c)
    {
        this.mReportTo.reportBack(tag,Utils.getCursorColumnNames(c));
    }
}

```

---

Funkcja `getACursor()` pobiera identyfikator URI albo w postaci ciągu znaków, albo jako obiekt identyfikatora URI wraz z opartą na ciągu znaków klauzulą `where`, a następnie przekazuje kurSOR. W omawianych przykładach często wyświetlamy nazwy kolumn pochodzących z otrzymywanego kurSora, utworzyliśmy więc metodę `printCursorColumnNames()`, która z kolei wykorzystuje klasę `Utils` do analizowania zawartości kurSora i uzyskiwania nazw jego kolumn.

Każdy wiersz przekazywany przez kurSOR kontaktu będzie posiadał pewną liczbę pól. W naszym przykładzie interesują nas tylko niektóre z nich. Wyraźliśmy tę koncepcję w kolejnej klasie, zwanej `AggregatedContact`, która została ukazana na listingu 27.24.

**Listing 27.24.** Kilka pól pochodzących z kontaktu zbiorczego

---

```
public class AggregatedContact
{
    public String id;
    public String lookupUri;
    public String lookupKey;
    public String displayName;

    public void fillinFrom(Cursor c)
    {
        id = Utils.getValue(c, "_ID");
        lookupKey = Utils.getValue(c, ContactsContract.Contacts.LOOKUP_KEY);
        lookupUri = ContactsContract.Contacts.CONTENT_LOOKUP_URI + "/" + lookupKey;
        displayName = Utils.getValue(c, ContactsContract.Contacts.DISPLAY_NAME);
    }
}
```

---

Kod z listingu 27.24 wcale nie jest skomplikowany. Wykorzystaliśmy tu kurSOR do wczytania interesujących nas pól. Zaprezentujemy teraz na listingu 27.25 klasę `AggregatedContactFunctionTester`, która pomoże nam wypełnić zadania ustalone na początku tego podrozdziału.

**Listing 27.25.** Kod umożliwiający testowanie kontaktów zbiorczych

---

```
public class AggregatedContactFunctionTester extends UriFunctionTester
{
    public AggregatedContactFunctionTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }
    /*
     * Pobiera kurSOR ze wszystkich kontaktów
     * Bez klauzuli where
     * Nie stosujmy tego w przypadku dużego zbioru
     */
    private Cursor getContacts()
    {
        // Uruchamia kwerendę
        Uri uri = ContactsContract.Contacts.CONTENT_URI;
        String sortOrder = ContactsContract.Contacts.DISPLAY_NAME
            + " COLLATE LOCALIZED ASC";
        Activity a = (Activity) this.mContext;
        return a.managedQuery(uri, null, null, null, sortOrder);
    }
}
```

```
}

/*
 * Wykorzystuje powyższą metodę getContacts
 * do utworzenia spisu kolumn zawartych w kurorze
 */
public void listContactCursorFields()
{
    Cursor c = null;
    try
    {
        c = getContacts();
        int i = c.getColumnCount();
        this.mReportTo.reportBack(tag, "Liczba kolumn:" + i);
        this.printCursorColumnNames(c);
    }
    finally
    {
        if (c!= null) c.close();
    }
}

/*
 * Przy użyciu kurwora wypełnionego kontaktami
 * zostają wyświetlane nazwy kontaktów
 * wraz z ich kluczami wyszukiwania
 */
private void printLookupKeys(Cursor c)
{
    for(c.moveToFirst();!c.isAfterLast();c.moveToNext())
    {
        String name=this.getContactName(c);
        String lookupKey = this.getLookupKey(c);
        String luri = this.getLookupUri(lookupKey);
        this.mReportTo.reportBack(tag, name + ":" + lookupKey);
        this.mReportTo.reportBack(tag, name + ":" + luri);
    }
}

/*
 * Wykorzystuje funkcję getContacts()
 * do uzyskania kurwora i wyświetlenia wszystkich
 * nazw kontaktów wraz z kluczami ich wyszukiwania
 * Stosuje funkcję printLookupKeys()
 */
public void listContacts()
{
    Cursor c = null;
    try
    {
        c = getContacts();
        int i = c.getColumnCount();
        this.mReportTo.reportBack(tag, "Liczba kolumn:" + i);
        this.printLookupKeys(c);
    }
```

```
        }
    finally
    {
        if (c!= null) c.close();
    }
}

/*
 * Funkcja użytkowa odczytująca
 * klucz wyszukiwania z kurSORA kontaktu
 */
private String getLookupKey(Cursor cc)
{
    int lookupkeyIndex = cc.getColumnIndex(ContactsContract.Contacts.LOOKUP_KEY);
    return cc.getString(lookupkeyIndex);
}

/*
 * Funkcja użytkowa odczytująca
 * wyświetlaną nazwę z kurSORA kontaktu
 */
private String getContactName(Cursor cc)
{
    return Utils.getColumnValue(cc, ContactsContract.Contacts.DISPLAY_NAME);
}

/**
 * Konstruuje identyfikator wyszukiwania na podstawie
 * identyfikatora URI kontaktów i klucza wyszukiwania
 */
private String getLookupUri(String lookupkey)
{
    String luri = ContactsContract.Contacts.CONTENT_LOOKUP_URI + "/" + lookupkey;
    return luri;
}

/**
 * Wykorzystuje identyfikator URI wyszukiwania
 * do odczytania pojedynczego kontaktu zbiorczego
 */
private Cursor getASingleContact(String lookupUri)
{
    // Uruchamia kwerendę
    Activity a = (Activity)this.mContext;
    return a.managedQuery(Uri.parse(lookupUri), null, null, null, null);
}

/*
 * Funkcja sprawdzająca, czy identyfikator URI stworzony za pomocą identyfikatora
 * URI wyszukiwania zwraca kurSOR zawierający inny zestaw kolumn.
 * Jak można było się spodziewać, zwracany jest podobny kurSOR
 * zawierający podobny zbiór kolumn.
 */

```

```

public void listLookupUriColumns()
{
    Cursor c = null;
    try
    {
        c = getContacts();
        String firstContactLookupUri = getFirstLookupUri(c);
        printLookupUriColumns(firstContactLookupUri);
    }
    finally
    {
        if (c!= null) c.close();
    }
}

public void printLookupUriColumns(String lookupuri)
{
    Cursor c = null;
    try
    {
        c = getASingleContact(lookupuri);
        int i = c.getColumnCount();
        this.mReportTo.reportBack(tag, "Liczba kolumn:" + i);
        int j = c.getCount();
        this.mReportTo.reportBack(tag, "Liczba wierszy:" + j);
        this.printCursorColumnNames(c);
    }
    finally
    {
        if (c!=null)c.close();
    }
}

/*
 * Pobiera listę kontaktów
 * Wyszukuje pierwszy kontakt
 * Przekazuje wartość null, jeśli nie znajdzie żadnego kontaktu
 */
private String getFirstLookupUri(Cursor c)
{
    c.moveToFirst();
    if (c.isAfterLast())
    {
        Log.d(tag,"Brak wierszy, z których mozna pobrac pierwszy kontakt");
        return null;
    }
    //Znaleziono wiersz
    String lookupKey = this.getLookupKey(c);
    String luri = this.getLookupUri(lookupKey);
    return luri;
}

/*
 * Pobiera listę kontaktów
 * Wyszukuje pierwszy kontakt i zwraca go

```

```
* w formie obiektu AggregatedContact
*/
protected AggregatedContact getFirstContact()
{
    Cursor c=null;
    try
    {
        c = getContacts();
        c.moveToFirst();
        if (c.isAfterLast())
        {
            Log.d(tag,"Brak kontaktow");
            return null;
        }
        //Znaleziono kontakt
        AggregatedContact firstcontact = new AggregatedContact();
        firstcontact.fillinFrom(c);
        return firstcontact;
    }
    finally
    {
        if (c!=null) c.close();
    }
}
```

---

Główne funkcje publiczne zostały wyróżnione pogrubioną czcionką. Przeznaczenie każdej z nich zostało wyjaśnione w przypisany do niej komentarzu. Po utworzeniu tej funkcji testowej dodajmy elementy menu z listingu 27.26 do pliku menu (*/res/menu/main\_menu.xml*).

---

**Listing 27.26.** Elementy menu związane z funkcją testową kontaktów zbiorczych

---

```
<item android:id="@+id/menu_show_contact_cursor"
      android:title="kursor kontaktów" />

<item android:id="@+id/menu_show_contacts"
      android:title="kontakte" />

<item android:id="@+id/menu_show_single_contact_cursor"
      android:title="kursor pojedynczego kontaktu" />
```

---

Możemy wstawić je w dowolnym miejscu pliku *main\_menu.xml*, proponujemy jednak umieszczenie ich w początkowej części kodu, dzięki czemu nowsze funkcje będą wyświetlane na początku menu. Po dodaniu opcji menu modyfikujemy aktywność sterującą w taki sposób, żeby przypominała kod z listingu 27.27.

---

**Listing 27.27.** Główna aktywność sterująca dostosowana do testowania kontaktów zbiorczych

---

```
public class TestContactsDriverActivity extends DebugActivity
implements IReportBack
{
    public static final String tag="TestContactsDriverActivity ";
    AccountsFunctionTester accountsFunctionTester = null;
```

```

AggregatedContactFunctionTester aggregatedContactFunctionTester = null;

public TestContactsDriverActivity()
{
    super(R.menu.main_menu,tag);
    accountsFunctionTester = new AccountsFunctionTester(this,this);
    aggregatedContactFunctionTester =
        new AggregatedContactFunctionTester(this,this);
}
protected boolean onMenuItemSelected(MenuItem item)
{
    Log.d(tag,item.getTitle().toString());
    if (item.getItemId() == R.id.menu_show_accounts)
    {
        accountsFunctionTester.testAccounts();
        return true;
    }
    if (item.getItemId() == R.id.menu_show_contact_cursor)
    {
        aggregatedContactFunctionTester.listContactCursorFields();
        return true;
    }
    if (item.getItemId() == R.id.menu_show_contacts)
    {
        aggregatedContactFunctionTester.listContacts();
        return true;
    }
    if (item.getItemId() == R.id.menu_show_single_contact_cursor)
    {
        aggregatedContactFunctionTester.listLookupUriColumns();
        return true;
    }
    return true;
}
}

```

Zwróćmy uwagę na trzy publiczne funkcje, które są wywoływane w wyniku wcisnięcia odpowiednich opcji menu:

- `listContactCursorFields()`,
- `listContacts()`,
- `listLookupUriColumns()`.

Omówimy działanie tych funkcji, opierając się na kodzie umieszczonym na listingu 27.26. Funkcja `listContactCursorFields` odczytuje całą listę kontaktów i wyświetla w kursorze nazwy kolumn. Identyfikatorem URI służącym do odczytywania wszystkich kontaktów jest `ContactsContract.Contacts.CONTENT_URI`.

W celu odczytania kursora przekazujemy ten identyfikator URI metodzie `managedQuery()`. Możemy przekazać wartość `null` w trakcie rzutowania kolumn, aby wyświetlić wszystkie kolumny. Chociaż nie jest to zalecane rozwiązanie, w naszym przypadku jest ono logiczne, gdyż chcemy poznać wszystkie kolumny przekazane przez identyfikator URI. Na listingu 27.28 widzimy spis wszystkich kolumn otrzymywanych dzięki temu identyfikatorowi.

**Listing 27.28.** Kolumny kurSORA przekazywane przez identyfikator URI dostawcy kontaktów

```
times_contacted;
contact_status;
custom_ringtone;
has_phone_number;
phonetic_name;
phonetic_name_style;
contact_status_label;
lookup;
contact_status_icon;
last_time_contacted;
display_name;
sort_key_alt;
in_visible_group;
_id;
starred;
sort_key;
display_name_alt;
contact_presence;
display_name_source;
contact_status_res_package;
contact_status_ts;
photo_id;
send_to voicemail;
```

---

Nasz przykładowy program wygeneruje listę tych kolumn zarówno w widoku programu, jak również w oknie *LogCat*. Skopiowaliśmy te pola z okna *LogCat* i sformatowaliśmy w sposób widoczny na listingu 27.28.

**Uwaga!**

Podczas pracy z dostawcami treści technika polegająca na korzystaniu z identyfikatora URI oraz wyświetlaniu przekazywanych kolumn może się okazać bardzo przydatna.

Po zapoznaniu się ze spisem kolumn za pomocą identyfikatora URI treści kontaktów zaznaczmy kilka z nich i sprawdźmy, jakie wiersze są dostępne. Kliknijmy w tym celu opcję menu *kontakte*, co spowoduje wywołanie funkcji *listContacts()*. Korzysta ona z tego samego identyfikatora URI, tym razem jednak wyświetla dla każdego kontaktu następujące kolumny:

- *display name*,
- *lookup key*,
- *lookup uri*.

Bierzemy te pola pod uwagę, ponieważ chcemy zobaczyć, jak wyglądają klucz wyszukiwania oraz identyfikator klucza wyszukiwania w porównaniu do informacji zawartych w części teoretycznej tego rozdziału. Interesuje nas zwłaszcza mechanizm uruchamiania identyfikatora URI wyszukiwania oraz typ otrzymywanego kurSORA. Kliknijmy w tym celu element menu *kurSOR pojedynczego kontaktu*. Zostanie wywołana funkcja *listLookupUriColumns()*. Pobierze ona pierwszy kontakt z listy kontaktów, a następnie wygeneruje identyfikator URI wyszukiwania dla tego kontaktu, po czym z niego skorzysta, a my poznamy wyniki.

Okazuje się, że wspomniana funkcja przekaże kurSOR zawierający takie same kolumny jak widoczne na listingu 27.28 — jedyna różnica polega na obecności tylko jednego wiersza

wskazującego kontakt, którego dotyczy ten klucz wyszukiwania. Zauważmy również, że wprowadziliśmy następującą definicję identyfikatora URI wyszukiwania:

`ContactsContract.Contacts.CONTENT_LOOKUP_URI`

Podczas dyskusji na temat identyfikatorów URI wyszukiwania stwierdziliśmy, że każdy tego typu obiekt symbolizuje zbiór połączonych ze sobą, nieprzetworzonych kontaktów. W takim przypadku powinniśmy się spodziewać, że otrzymamy zestaw takich samych nieprzetworzonych kontaktów. Powyższy test (listing 27.28) udowadnia nam jednak, że nie dostajemy kurSORA zawierającego nieprzetworzone kontakty, lecz kurSOR przechowujący kontakty zbiorcze.

#### Uwaga!

W efekcie wyszukiwania opartego na identyfikatorze wyszukiwania kontaktu otrzymujemy kontakt zbiorczy, a nie kontakt nieprzetworzony.

Kolejną istotną i ciekawą cechą procesu wyszukiwania kontaktu zbiorczego opartego na identyfikatorze wyszukiwania jest to, że nie zachodzi w sposób liniowy i że nie jest dokładny. Oznacza to, że system nie będzie poszukiwał trafienia dokładnie odpowiadającego kluczowi dopasowania. Zamiast tego Android analizuje składnię klucza wyszukiwania pod kątem tworzących go nieprzetworzonych kontaktów, następnie odnajduje identyfikator kontaktu zbiorczego, który jest zgodny z większością rekordów nieprzetworzonych kontaktów, i przekazuje rekord kontaktu zbiorczego utworzonego w ten sposób.

Jedną z konsekwencji tego mechanizmu jest brak możliwości publicznego przejścia od klucza wyszukiwania do nieprzetworzonych kontaktów, które stanowią jego składowe. Jedynym rozwiązaniem jest odnalezienie identyfikatora kontaktu związanego z kluczem wyszukiwania, a następnie uruchomienie obiektu URI nieprzetworzonego kontaktu wobec identyfikatora znalezionej kontaktu, dzięki czemu zostaną odczytane jego nieprzetworzone kontakty.

## Badanie nieprzetworzonych kontaktów

W następnym przykładowym programie przedstawimy rozwiązanie pozwalające na eksplorację nieprzetworzonych kontaktów. W tym ćwiczeniu postaramy się wykonać trzy zadania:

- Odkryć wszystkie przekazywane pola poprzez uruchomienie identyfikatora URI odczytującego nieprzetworzone kontakty.
- Wyświetlić wszystkie nieprzetworzone kontakty.
- Wygenerować listę nieprzetworzonych kontaktów tworzących kontakt zbiorczy.

W tym przykładzie będą potrzebne następujące nowe pliki:

- `RawContact.java`,
- `RawContactFunctionTester.java`.

Pliki te zostaną zaprezentowane w trakcie omawiania szczegółów przykładu. Będziemy musieli zaktualizować również następujące pliki pochodzące z poprzedniego przykładu:

- `main_menu.xml`,
- `TestContactsDriverActivity.java`.

W dalszej części podrozdziału zaprezentujemy również zmiany, jakie należy wprowadzić w powyższych plikach.

Plik z listingu 27.29, `RawContact.java`, pobiera kilka istotnych pól z tabeli nieprzetworzonych kontaktów.

**Listing 27.29.** Plik RawContact.java

```
public class RawContact
{
    public String rawContactId;
    public String aggregatedContactId;
    public String accountName;
    public String accountType;
    public String displayName;

    public void fillinFrom(Cursor c)
    {
        rawContactId = Utils.getColumnValue(c, "_ID");
        accountName = Utils.getColumnValue(c, ContactsContract.RawContacts.ACCOUNT_NAME);
        accountType = Utils.getColumnValue(c, ContactsContract.RawContacts.ACCOUNT_TYPE);
        aggregatedContactId = Utils.getColumnValue(c,
            ContactsContract.RawContacts.CONTACT_ID);
        displayName = Utils.getColumnValue(c, "display_name");
    }
    public String toString()
    {
        return displayName
            + "/" + accountName + ":" + accountType
            + "/" + rawContactId
            + "/" + aggregatedContactId;
    }
}
```

---

Aby móc przetestować zawarte w tym przykładzie funkcje, musimy do pliku *main\_menu.xml* dodać widoczne na listingu 27.30 elementy menu.

**Listing 27.30.** Opcje menu umożliwiające przetestowanie nieprzetworzonych kontaktów

```
<item android:id="@+id/menu_show_rc_all"
      android:title="wszystkie nieprzetworzone kontakty" />

<item android:id="@+id/menu_show_rc"
      android:title="nieprzetworzone kontakty" />

<item android:id="@+id/menu_show_rc_cursor"
      android:title="kursor nieprzetworzonych kontaktów" />
```

---

Każdy z tych elementów menu powoduje wywołanie trzech funkcji umieszczonej w pliku *RawContactFunctionTester.java*. Zawarty w tym pliku kod jest widoczny na listingu 27.31.

**Listing 27.31.** Testowanie nieprzetworzonych kontaktów

```
public class RawContactsFunctionTester
extends AggregatedContactFunctionTester
{
    public RawContactsFunctionTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }
}
```

```
public void showAllRawContacts()
{
    Cursor c = null;
    try
    {
        c = this.getACursor(getRawContactsUri(), null);
        this.printRawContacts(c);
    }
    finally
    {
        if (c!=null) c.close();
    }
}
public void showRawContactsForFirstAggregatedContact()
{
    AggregatedContact ac = getFirstContact();
    this.mReportTo.reportBack(tag, ac.displayName + ":" + ac.id);

    Cursor c = null;

    try
    {
        c = this.getACursor(getRawContactsUri(), getClause(ac.id));
        this.printRawContacts(c);
    }
    finally
    {
        if (c!=null) c.close();
    }
}
private void printRawContacts(Cursor c)
{
    for(c.moveToFirst();!c.isAfterLast();c.moveToNext())
    {
        RawContact rc = new RawContact();
        rc.fillinFrom(c);
        this.mReportTo.reportBack(tag, rc.toString());
    }
}
public void showRawContactsCursor()
{
    AggregatedContact ac = getFirstContact();
    this.mReportTo.reportBack(tag, ac.displayName + ":" + ac.id);

    Cursor c = null;

    try
    {
        c = this.getACursor(getRawContactsUri(),null);
        this.printCursorColumnNames(c);
    }
    finally
    {
        if (c!=null) c.close();
    }
}
```

```
private Uri getRawContactsUri()
{
    return ContactsContract.RawContacts.CONTENT_URI;
}
private String getClause(String contactId)
{
    return "contact_id = " + contactId;
}
}
```

---

Na listingu 27.32 znalazł się zaktualizowany kod aktywności sterującej, przejmującej od elementów menu proces wywoływanego funkcji publicznych pochodzących z testera funkcji nieprzetworzonych kontaktów.

**Listing 27.32.** Zaktualizowana aktywność sterująca, pozwalająca na testowanie nieprzetworzonych kontaktów

---

```
public class TestContactsDriverActivity extends DebugActivity
implements IReportBack
{
    //.....kontynuacja
    RawContactsFunctionTester rawContactFunctionTester = null;

    public TestContactsDriverActivity()
    {
        //.....kontynuacja
        rawContactFunctionTester = new RawContactsFunctionTester(this, this);
    }
    protected boolean onMenuItemSelected(MenuItem item)
    {
        //.....kontynuacja
        if (item.getItemId() == R.id.menu_show_single_contact_cursor)
        {
            aggregatedContactFunctionTester.listLookupUriColumns();
            return true;
        }
        //początek nowych wpisów
        if (item.getItemId() == R.id.menu_show_rc_cursor)
        {
            rawContactFunctionTester.showRawContactsCursor();
            return true;
        }
        if (item.getItemId() == R.id.menu_show_rc_all)
        {
            rawContactFunctionTester.showAllRawContacts();
            return true;
        }
        if (item.getItemId() == R.id.menu_show_rc)
        {
            rawContactFunctionTester.showRawContactsForFirstAggregatedContact();
            return true;
        }
        //koniec nowych wpisów
    }
}
```

---

```

        return true;
    }
}

```

---

Na powyższym listingu zaprezentowaliśmy jedynie nowe wiersze, które należy wstawić do aktywności sterującej, gdyż jest to jeden z aktualizowanych plików.

Podobnie jak zrobiliśmy w przypadku kontaktów zbiorczych, przyjrzyjmy się najpierw naturze identyfikatorów URI nieprzetworzonych kontaktów oraz przekazywanym przez nie wartościom. Sygnatura identyfikatora nieprzetworzonego kontaktu wygląda następująco:

```
ContactsContract.RawContacts.CONTENT_URI;
```

Jeżeli przyjrzymy się kodowi metody `showRawContactsCursor()`, zauważymy, że jest w nim wykorzystywany powyższy identyfikator nieprzetworzonego kontaktu, dzięki czemu znajdują wyświetlane pola kursora. Kliknijmy obiekt menu *kursor nieprzetworzonych kontaktów*. Zobaczmy, że kursor nieprzetworzonego kontaktu zawiera pola wypisane na listingu 27.33.

#### **Listing 27.33.** Pola kursora nieprzetworzonego kontaktu

---

```

times_contacted;
phonetic_name;
phonetic_name_style;
contact_id;version;
last_time_contacted;
aggregation_mode;
_id;
name_verified;
display_name_source;
dirty;
send_to voicemail;
account_type;
custom_ringtone;
sync4;sync3;sync2;sync1;
deleted;
account_name;
display_name;
sort_key_alt;
starred;
sort_key;
display_name_alt;
sourceid;

```

---

Skoro zapoznaliśmy się z kolumnami kursora nieprzetworzonych kontaktów, mogą nas zainteresować również wiersze tej tabeli. Kliknijmy teraz opcję menu *wszystkie nieprzetworzone kontakty*. Zostanie wywołana metoda `showAllRawContacts()`. Metoda ta będzie manipulowała kursorem bez użycia klauzuli WHERE (dzięki czemu uzyskamy dostęp do wszystkich wierszy) oraz utworzy obiekt RawContact dla każdego wiersza, po czym wyświetli wyniki. Lista nieprzetworzonych kontaktów zostanie ukazana w widoku aplikacji oraz w oknie LogCat.

Za pomocą widocznych na listingu 27.33 kolumn kursora sprawdźmy, czy możemy zmodyfikować kwerendę w taki sposób, aby odczytywać kontakty za pomocą danego identyfikatora kontaktów zbiorczych. Przetestujemy taką możliwość, klikając element menu *nieprzetworzone*

*kontakte*. Zostanie najpierw wyszukany pierwszy kontakt zbiorczy, a następnie wysłany identyfikator nieprzetworzonego kontaktu wraz z klauzulą WHERE, definiującą wartość kolumny contact\_id. Wyniki będą widoczne zarówno w interfejsie użytkownika, jak i w dzienniku LogCat.

Chociaż przejrzelismy już kontakty zbiorcze i nieprzetworzone kontakty, tak naprawdę nie odczytaliśmy jeszcze zawartości najważniejszych ich pól, na przykład adresu e-mail lub numeru telefonu. W następnym punkcie powiemy, jak można tego dokonać.

## Przeglądanie danych nieprzetworzonego kontaktu

W kolejnym programie ukażemy sposób przeglądania danych powiązanych z nieprzetworzonymi kontaktami. Spróbujemy teraz wykonać dwa zadania:

- Wykryć wszystkie przekazywane pola poprzez uruchomienie identyfikatora odczytującego dane nieprzetworzonych kontaktów.
- Odczytać dane przechowywane w zestawie kontaktów zbiorczych.

W tej przykładowej aplikacji pojawiają się następujące nowe pliki:

- ContactData.java,
- ContactDataFunctionTester.java.

Zawartość tych plików zostanie ukazana w trakcie omawiania tej aplikacji. Wymagana będzie również modyfikacja dwóch plików z poprzedniego przykładu:

- main\_menu.xml,
- TestContactsDriverActivity.java.

Zmiany, które trzeba wprowadzić w tych plikach, zostaną zaprezentowane w dalszej części rozdziału. Kod zawarty w pliku ContactData.java służy do pobrania reprezentacyjnego zestawu danych kontaktu. Na listingu 27.34 znajdziemy kod źródłowy tego pliku.

**Listing 27.34.** Plik ContactData.java

---

```
public class ContactData
{
    public String rawContactId;
    public String aggregatedContactId;
    public String dataId;
    public String accountName;
    public String accountType;
    public String mimetype;
    public String data1;

    public void fillinFrom(Cursor c)
    {
        rawContactId = Utils.getColumnValue(c, "_ID");
        accountName = Utils.getColumnValue(c, ContactsContract.RawContacts.ACCOUNT_NAME);
        accountType = Utils.getColumnValue(c, ContactsContract.RawContacts.ACCOUNT_TYPE);
        aggregatedContactId =
            Utils.getColumnValue(c, ContactsContract.RawContacts.CONTRACT_ID);
        mimetype = Utils.getColumnValue(c, ContactsContract.RawContactsEntity.MIMETYPE);
        data1 = Utils.getColumnValue(c, ContactsContract.RawContactsEntity.DATA1);
        dataId = Utils.getColumnValue(c, ContactsContract.RawContactsEntity.DATA_ID);
    }
}
```

---

```

public String toString()
{
    return data1 + "/" + mimetype
        + "/" + accountName + ":" + accountType
        + "/" + dataId
        + "/" + rawContactId
        + "/" + aggregatedContactId;
}

```

---

Sposób działania tego przykładu został zdefiniowany w pliku *ContactFunctionTester.java*. Jego kod jest widoczny na listingu 27.35.

#### **Listing 27.35.** Testowanie danych kontaktu

---

```

public class ContactDataFunctionTester extends RawContactFunctionTester
{
    public ContactDataFunctionTester(Context ctx, IReportBack target)
    {
        super(ctx, target);
    }
    public void showRawContactsEntityCursor()
    {
        Cursor c = null;
        try
        {
            Uri uri = ContactsContract.RawContactsEntity.CONTENT_URI;
            c = this.getACursor(uri,null);
            this.printCursorColumnNames(c);
        }
        finally
        {
            if (c!=null) c.close();
        }
    }
    public void showRawContactsData()
    {
        Cursor c = null;
        try
        {
            Uri uri = ContactsContract.RawContactsEntity.CONTENT_URI;
            c = this.getACursor(uri,"contact_id w (3,4,5)");
            this.printRawContactsData(c);
        }
        finally
        {
            if (c!=null) c.close();
        }
    }
    protected void printRawContactsData(Cursor c)
    {
        for(c.moveToFirst();!c.isAfterLast();c.moveToNext())
        {
            ContactData dataRecord = new ContactData();
            dataRecord.fillinFrom(c);

```

```
        this.mReportTo.reportBack(tag, dataRecord.toString());
    }
}
}
```

---

Do wywołania funkcji publicznych występujących w tej klasie potrzebne nam będą elementy menu z listingu 27.36, które należy dodać do pliku *main\_menu.xml*.

---

**Listing 27.36.** Opcje menu wymagane do testowania danych kontaktu

---

```
<item android:id="@+id/menu_show_rce_data"
      android:title="dane kontaktu" />
<item android:id="@+id/menu_show_rce_cursor"
      android:title="kursor encji kontaktu" />
```

---

Aktywność sterująca musi zostać zmodyfikowana zgodnie z zawartością listingu 27.37, gdyż w przeciwnym wypadku nie będzie reagowała na wciśnięcia wspomnianych elementów menu i nie będzie wywoływała funkcji klasy *ContactDataFunctionTester*.

---

**Listing 27.37.** Zaktualizowana aktywność sterująca, pozwalająca na testowanie danych kontaktu

---

```
public class TestContactsDriverActivity extends DebugActivity
implements IReportBack
{
    public static final String tag="TestContacts";
    ...inne funkcje testowe
    ...dodajmy poniższy wiersz na końcu pozostałych funkcji testowych
    ContactDataFunctionTester contactDataFunctionTester = null;

    public TestContactsDriverActivity()
    {
        ...dodajmy poniższy wiersz na końcu niniejszej funkcji
        contactDataFunctionTester = new ContactDataFunctionTester(this,this);
    }
    protected boolean onMenuItemSelected(MenuItem item)
    {
        ...odpowiada na pozostałe elementy menu
        ...dodajmy następujące wiersze
        if (item.getItemId() == R.id.menu_show_rce_cursor)
        {
            contactDataFunctionTester.showRawContactsEntityCursor();
            return true;
        }
        if (item.getItemId() == R.id.menu_show_rce_data)
        {
            contactDataFunctionTester.showRawContactsData();
            return true;
        }
        ...koniec nowych wierszy
        return true;
    }
}
```

---

Przeanalizujmy teraz powyższy kod i cały przykładowy program. Android ustanawia specjalny widok, RawContactEntity, służący do odczytywania danych z tabeli nieprzetworzonych kontaktów oraz z odpowiadających jej tabel danych, co zostało zaprezentowane w punkcie dotyczącym widoku Contact\_entities\_view. Identyfikator URI uzyskujący dostęp do tego widoku został zdefiniowany w pomocniczej klasie. Pełna ścieżka stałej tego identyfikatora została zaprezentowana na listingu 27.38.

#### **Listing 27.38.** Identyfikator treści nieprzetworzonego kontaktu

---

```
ContactsContract.RawContactsEntity.CONTENT_URI
```

---

Powyższy identyfikator jest wykorzystywany w omawianym programie do uzyskiwania informacji na temat przekazywanych pól. Spis tych pól otrzymamy po wciśnięciu opcji menu *kursor encji kontaktu*. Listing 27.39 prezentuje spis kolumn uzyskiwany po kliknięciu wspomnianego elementu menu.

#### **Listing 27.39.** Kolumny kurSORA encji kontaktu

---

```
data_version;
contact_id;
version;
data12;data11;data10;
mimetype;
res_package;
_id;
data15;data14;data13;
name_verified;
is_restricted;
is_super_primary;
data_sync1;dirty;data_sync3;data_sync2;
data_sync4;account_type;data1;sync4;sync3;
data4;sync2;data5;sync1;
data2;data3;data8;data9;
deleted;
group_sourceid;
data6;data7;
account_name;
data_id;
starred;
sourceid;
is_primary;
```

---

Po zapoznaniu się z tym zestawem kolumn możemy zawęzić wyniki przekazywane przez ten kurSOR poprzez sformułowanie klauzuli WHERE. Przykładowo po kliknięciu następnego elementu menu uzyskamy elementy danych, którym przypisano identyfikatory kontaktu o wartościach 3, 4 i 5. W tym celu wystarczyło dodać w kodzie następującą klauzulę WHERE:

"contact\_id in (3,4,5)"

i wysłać ją wraz z kurSorem. Dokładnie taką operację została powiązana z obiektem menu *dane kontaktu*. Po jego wciśnięciu zostaną wyświetlane takie informacje, jak imię i nazwisko oraz adres e-mail (element danych rozpoznajemy po jego typie MIME).

## Dodawanie kontaktu oraz szczegółowych informacji o nim

Dotychczas omawialiśmy jedynie odczytywanie kontaktów. Zajmijmy się teraz przykładowym programem pozwalającym na dodanie nowego kontaktu zawierającego imię, nazwisko, adres e-mail oraz numer telefonu.

Aby móc zapisywać informacje w kontakcie, trzeba dodać następujące uprawnienie w pliku manifeście (listing 27.20):

```
android.permission.WRITE_CONTACTS
```

Do przetestowania tego przykładu będziemy potrzebować nowego pliku:

- *AddContactFunctionTester.java*.

Ponadto musimy zmodyfikować następujące pliki, pochodzące z poprzednich przykładów:

- *main\_menu.xml*,
- *TestContactsDriverActivity.java*.

Plik *AddContactFunctionTester.java* pozwala na dodawanie kontaktu wypełnionego danymi. Na listingu 27.40 widzimy kod źródłowy tego pliku.

**Listing 27.40.** Dodawanie kontaktów zawierających szczegółowe informacje

---

```
Import android.provider.ContactsContract.Data;  
//...pozostałe instrukcje importu, które środowisko Eclipse może dodać za nas  
  
public class AddContactFunctionTester extends ContactDataFunctionTester  
{  
    public AddContactFunctionTester(Context ctx, IReportBack target)  
    {  
        super(ctx, target);  
    }  
    public void addContact()  
    {  
        long rawContactId = insertRawContact();  
        this.mReportTo.reportBack(tag, "RawcontactId:" + rawContactId);  
        insertName(rawContactId);  
        insertPhoneNumber(rawContactId);  
        showRawContactsDataForRawContact(rawContactId);  
    }  
    private void insertName(long rawContactId)  
    {  
        ContentValues cv = new ContentValues();  
        cv.put(Data.RAW_CONTACT_ID, rawContactId);  
        cv.put(Data.MIMETYPE, StructuredName.CONTENT_ITEM_TYPE);  
        cv.put(StructuredName.DISPLAY_NAME,"Gall Anonim_" + rawContactId);  
        this.mContext.getContentResolver().insert(Data.CONTENT_URI, cv);  
    }  
    private void insertPhoneNumber(long rawContactId)  
    {  
        ContentValues cv = new ContentValues();  
        cv.put(Data.RAW_CONTACT_ID, rawContactId);  
        cv.put(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE);  
        cv.put(Phone.NUMBER,"123 123 " + rawContactId);  
        cv.put(Phone.TYPE,Phone.TYPE_HOME);  
    }  
}
```

```

        this.mContext.getContentResolver().insert(Data.CONTENT_URI, cv);
    }
    private long insertRawContact()
    {
        ContentValues cv = new ContentValues();
        cv.put(RawContacts.ACCOUNT_TYPE, "com.google");
        cv.put(RawContacts.ACCOUNT_NAME, "satya.komatineni@gmail.com");
        Uri rawContactUri =
            this.mContext.getContentResolver()
                .insert(RawContacts.CONTENT_URI, cv);
        long rawContactId = ContentUris.parseId(rawContactUri);
        return rawContactId;
    }
    private void showRawContactsDataForRawContact(long rawContactId)
    {
        Cursor c = null;
        try
        {
            Uri uri = ContactsContract.RawContactsEntity.CONTENT_URI;
            c = this.getACursor(uri, "_id = " + rawContactId);
            this.printRawContactsData(c);
        }
        finally
        {
            if (c!=null) c.close();
        }
    }
}

```

Jedyną funkcją publiczną jest `addContact()`. W celu jej wywołania potrzebny nam będzie element menu zamieszczony na listingu 27.41.

#### **Listing 27.41.** Element menu pozwalający na dodawanie kontaktu

---

```
<item android:id="@+id/menu_add_contact"
      android:title="Dodaj kontakt" />
```

---

Powyższe dwa wiersze umieszczaćmy w pliku *main\_menu.xml*. Musimy zmodyfikować także aktywność sterującą w taki sposób, żeby uruchamiała metodę `addContact()` po wcisnięciu utworzonej przed chwilą opcji menu. Na listingu 27.42 prezentujemy kod źródłowy zmodyfikowanej aktywności sterującej (pamiętajmy, że mamy tu do czynienia nie z nowym plikiem, lecz z aktualizacją starego).

#### **Listing 27.42.** Zaktualizowana aktywność sterująca, pozwalająca na dodawanie kontaktów

---

```
public class TestContactsDriverActivity extends DebugActivity
implements IReportBack
{
    ...inne mechanizmy
    AddContactFunctionTester addContactFunctionTester = null;

    public TestContactsDriverActivity()
    {
```

```
...inne mechanizmy
    addContactFunctionTester = new AddContactFunctionTester(this, this);
}
protected boolean onMenuItemSelected(MenuItem item)
{
    ...inne mechanizmy
    if (item.getItemId() == R.id.menu_add_contact)
    {
        addContactFunctionTester.addContact();
        return true;
    }
    return true;
}
```

---

Jeżeli klikniemy teraz opcję menu *Dodaj kontakt*, kod zawarty na listingu 27.40 (funkcja te-stowa dodawania kontaktów) wykona następujące czynności:

1. Najpierw doda do predefiniowanego konta (korzystając z jego nazwy i typu) nieprzetworzony kontakt za pomocą metody `insertRawContact()`.
2. Pobierze identyfikator nieprzetworzonego kontaktu i wstawi w tabeli danych rekord imienia i nazwiska (metoda `insertName()`).
3. Znowu pobierze identyfikator nieprzetworzonego kontaktu i wstawi w tabeli danych rekord numeru telefonu (metoda `insertPhoneNumber()`).

Na listingu 27.40 widzimy aliasy kolumn wykorzystywane przez wymienione metody podczas wstawiania rekordów. Umieściliśmy je również na listingu 27.43, aby uprościć ich przeglądanie.

---

**Listing 27.43. Używanie aliasów kolumn w standardowych strukturach danych kontaktu**

---

```
cv.put(Data.RAW_CONTACT_ID, rawContactId);
cv.put(Data.MIMETYPE, StructuredName.CONTENT_ITEM_TYPE);
cv.put(StructuredName.DISPLAY_NAME, "Gall Anonim_" + rawContactId);

cv.put(Data.RAW_CONTACT_ID, rawContactId);
cv.put(Data.MIMETYPE, Phone.CONTENT_ITEM_TYPE);
cv.put(Phone.NUMBER, "123 123 " + rawContactId);
cv.put(Phone.TYPE, Phone.TYPE_HOME);

cv.put(RawContacts.ACCOUNT_TYPE, "com.google");
cv.put(RawContacts.ACCOUNT_NAME, "satya.komatineni@gmail.com");
```

---

Szczególnie istotne jest, aby pamiętać, że takie stałe, jak `Phone.TYPE` czy `Phone.NUMBER`, odnoszą się w rzeczywistości do nazw kolumn `data1` i `data2` umieszczonych w tabeli danych.

Aby w końcu ujrzeć dodany rekord, kliknijmy element menu *Dodaj kontakt*. Zostaną dodane i wyświetcone szczegółowe informacje tego rekordu, gdyż zostaną one odczytane za pomocą funkcji `showRawContactsDataForRawContact()`. Każde z pól danych będzie umieszczone w strukturze `ContactData`.

## Kontrola agregacji

W tej chwili dla Czytelnika powinno już być jasne, że klienci aktualizujące lub dodające kontakty nie modyfikują tabeli contact w jawnym sposobie. Tabela ta jest modyfikowana przez obiekty wyzwalające, które śledzą tabelę nieprzetworzonych kontaktów oraz tabelę danych.

Z kolei dodawane lub zmieniane nieprzetworzone kontakty oddziałują na kontakty zbiorcze znajdujące się w tabeli kontaktów. Niekiedy jednak powiązanie dwóch kontaktów ze sobą jest niekorzystne.

Możemy kontrolować proces łączenia nieprzetworzonych kontaktów poprzez ustalenie trybu agregacji w czasie ich tworzenia. Jak widać po umieszczonych na listingu 27.33 nazwach kolumn tabeli nieprzetworzonych kontaktów, zawiera ona pole aggregation\_mode. Wartości umieszczone w tym polu zostały wymienione na listingu 27.2 i objaśnione w punkcie „Kontakty zbiorcze”.

Możemy również zapewnić, by dwa kontakty pozostawały rozdzielone, poprzez umieszczenie odpowiednich wierszy w tabeli agg\_exceptions. Identyfikatory URI wymagane do wstawiania danych do tej tabeli są zdefiniowane w klasie ContactsContract.AggregationExceptions. Struktura tabeli agg\_exceptions została zaprezentowana na listingu 27.44.

**Listing 27.44.** Definicja tabeli zawierającej wyjątki agregacji

---

```
CREATE TABLE agg_exceptions
(_id INTEGER PRIMARY KEY AUTOINCREMENT,
type INTEGER NOT NULL,
raw_contact_id1 INTEGER REFERENCES raw_contacts(_id),
raw_contact_id2 INTEGER REFERENCES raw_contacts(_id))
```

---

Kolumna type może przechowywać jedną ze stałych wymienionych na listingu 27.45.

**Listing 27.45.** Typy agregacji definiowane w tabeli wyjątków agregacji

---

```
TYPE_KEEP_TOGETHER
TYPE_KEEP_SEPARATE
TYPE_AUTOMATIC
```

---

Definicje i zadania poszczególnych typów agregacji są dość zrozumiałe. Typ TYPE\_KEEP\_TOGETHER nie pozwala na rozdzielenie dwóch kontaktów. Z kolei wartość TYPE\_KEEP\_SEPARATE uniemożliwia połoczenie kontaktów. Ostatnia wartość, TYPE\_AUTOMATIC, wykorzystuje domyślny algorytm agregacji kontaktów.

Identyfikator URI umożliwiający umieszczanie, odczytywanie i aktualizowanie wierszy zawartych w tej tabeli wygląda następująco:

`ContactsContract.AggregationExceptions.CONTENT_URI`

Również stałe wykorzystywane wraz z definicjami pól w tej tabeli są dostępne w klasie `ContactsContract.AggregationExceptions`.

## Konsekwencje synchronizacji

Przez większość rozdziału zajmowaliśmy się wyłącznie manipulowaniem kontaktami w obrębie urządzenia. Zazwyczaj jednak konta i przypisane im kontakty są synchronizowane łącznie. Jeśli na przykład utworzyliśmy konto Google w telefonie obsługiwany przez system Android, wszystkie kontakty zostaną skopiowane z serwera do tego telefonu.

Za każdym razem, gdy w urządzeniu dodajemy nowy kontakt lub konto serwerowe, następi proces synchronizacji i odzwierciedlenia danego obiektu w obydwu miejscach.

Jednak w tym wydaniu książki nie zajęliśmy się omówieniem interfejsu synchronizowania ani mechanizmem jego działania. Jest to zagadnienie równie obszerne jak tematyka kontaktów. Znajomość działania interfejsu kontaktów znacznie pomaga w zrozumieniu interfejsu synchronizacji. Zalecamy więc zaglądanie na stronę [www.androidbook.com](http://www.androidbook.com), która jest dość regularnie aktualizowana.

Natura mechanizmu synchronizacji wpływa również na proces usuwania kontaktów z urządzenia. W czasie usuwania kontaktu za pomocą identyfikatora kontaktu zbiorczego zostaną wykasowane wszystkie związane z nim nieprzetworzone kontakty, a także elementy danych powiązane z tymi kontaktami. Jednak system jedynie oznacza te obiekty jako usunięte i dopiero w procesie przebiegającej w tle synchronizacji z serwerem zaznaczone kontakty zostaną trwale usunięte z urządzenia. Taka kaskada procesów kasowania występuje także na poziomie nieprzetworzonych kontaktów, gdzie zostają usunięte elementy danych powiązane z danym nieprzetworzonym kontaktem.

## Odbośniki

Zaprezentowane poniżej odnośniki umożliwiają Czytelnikowi dostęp do materiałów pomocniczych oraz rozszerzających zakres informacji zawartych w tym rozdziale. Ostatni z zamieszczonych adresów URL umożliwia pobranie projektów utworzonych specjalnie na potrzeby tego rozdziału.

- <http://www.google.com/support/mobile/bin/answer.py?answer=182077> — adres instrukcji obsługi Androida w wersji 2.3. Znajdziemy w niej informacje dotyczące aplikacji Kontakty, pozwalającej na zarządzanie kontaktami. Choć omówiliśmy podstawowe informacje związane z obsługą tej aplikacji, najważniejsza w tej kwestii pozostaje instrukcja obsługi. Czytelnik może w niej znaleźć informacje, które my mogliśmy przypadkowo przeoczyć.
- <http://www.google.com/help/hc/pdfs/mobile/AndroidUsersGuide-30-100.pdf> — instrukcja obsługi Androida w wersji 3.0.
- <http://developer.android.com/resources/articles/contacts.html> — ten adres URL prowadzi do artykułu, w którym omówiono sposób korzystania z interfejsu kontaktów. Jest to podstawowa dokumentacja dotycząca interfejsu kontaktów, stworzona przez firmę Google.
- <http://www.androidbook.com/item/3585> — zrozumienie koncepcji interfejsu kontaktów polega przede wszystkim na pojęciu struktury tworzących go tabel. Klasa `ContactsContract` stanowi jedynie cienką osłonę wokół podstawowej struktury tabel. Pod tym adresem można znaleźć informacje o różnorodnych strukturach tabel opracowanych przez autorów książki. Znajdziemy tam nazwy pól, ich typy, widoki kontaktów zbiorczych i tak dalej.

- <http://developer.android.com/reference/android/provider/ContactsContract.html> — dokumentacja Javadoc opisująca klasę wejściową opublikowanego kontraktu kontaktów. Bardzo przydatny adres dla osób zajmujących się pisaniem programów wykorzystujących interfejs kontaktów.
- <http://www.netmite.com/android/mydroid/2.0/packages/providers/ContactsProvider/> — z powodu niedostatku informacji dotyczących obsługi kontaktów być może Czytelnika zainteresuje kod źródłowy dostawcy kontaktów. Pod tym adresem znajdziemy stronę Netmite, gdzie zamieszczono kody źródłowe wszystkich plików tworzących dostawcę treści.
- <http://www.netmite.com/android/mydroid/2.0/packages/apps/Contacts/src/com/android/contacts> — podobnie jak w poprzednim przypadku, łącze to prowadzi do kodu źródłowego aplikacji Kontakty. Jeżeli Czytelnik chce poznać mechanizm tworzenia lub aktualizowania kontaktu zbiorczego, właśnie znalazł życzę złota.
- <http://www.androidbook.com/item/3537> — jeżeli Czytelnik przeglądał kody źródłowe dostępne pod dwoma powyższymi adresami, prawdopodobnie poczuł się nieco ogłuszony. Pod tym adresem znajdziemy więc podsumowanie danych tam zawartych, które być może komuś się przyda.
- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — pod tym adresem znajdziemy projekty utworzone specjalnie na potrzeby niniejszej książki. Interesujący nas katalog nosi nazwę *ProAndroid3\_R27\_Kontakty*.

## Podsumowanie

W niniejszym rozdziale zapoznaliśmy się ze strukturą kontaktów, dostępną w systemie Android. Możemy wykorzystać zawarte tu informacje do odczytywania lub aktualizowania kontaktów za pomocą publicznego interfejsu kontaktów.

Chociaż poświęciliśmy mnóstwo uwagi interfejsowi kontaktów, nie omówiliśmy pracy z dostawcami treści pracującymi w trybie wsadowym, w którym można dodawać lub usuwać kontakty. Zestaw Android SDK zawiera klasę `ContentProviderOperation` umożliwiającą przeprowadzenie procesów wstawiania, aktualizowania i usuwania kontaktów w trybie wsadowym, co pozwala na optymalizację działania systemu.

Tryb wsadowy staje się tym istotniejszy dla dostawców synchronizacji, im większa liczba kontaktów jest aktualizowana i dodawana. W przypadku kwerend oraz sporadycznych aktualizacji omówione w tym rozdziale rozwiązania są całkowicie wystarczające. Warto jednak co jakiś czas zaglądać na stronę [www.androidbook.com](http://www.androidbook.com).



## Wdrażanie aplikacji na rynek — Android Market i nie tylko

Stworzenie wspaniałej aplikacji, którą pokochają użytkownicy, to jedna sprawa. Należy też zadbać o wprowadzenie rozwiązania umożliwiającego jej szybkie znalezienie i pobranie. W tym właśnie celu firma Android zaprojektowała sklep Android Market. Za pomocą ikony umieszczonej po prawej stronie urządzenia użytkownicy mogą przejść wprost na stronę sklepu i przeglądać, wyszukiwać, oceniać oraz pobierać aplikacje. Użytkownicy mogą uzyskać dostęp do serwisu Android Market również z poziomu komputera stacjonarnego, chociaż pobierane pliki będą ostatecznie umieszczane w urządzeniu, a nie w stacji roboczej. Część aplikacji jest dostępna za darmo, a w przypadku płatnych wersji zostały wprowadzone mechanizmy płatnicze usprawniające szybki zakup.

Android Market jest dostępny nawet z poziomu intencji wewnętrz aplikacji, dzięki czemu użytkownik w łatwy sposób może znaleźć miejsce, z którego można pobrać składniki wymagane przez ten program. Na przykład po wydaniu nowej wersji aplikacji możemy pozwolić użytkownikowi dostać się bezpośrednio do lokacji, z której można pobrać lub zakupić ten plik. Android Market nie jest jednak jedynym miejscem, w którym można zaopatrzyć się w aplikacje; w internecie cały czas pojawiają się nowe kanały.

Android Market nie jest dostępny z poziomu emulatora (choć istnieją pewne nielegalne sposoby obejścia tego problemu). Stanowi to swego rodzaju utrudnienie dla programisty. Najlepszym rozwiązaniem jest własne urządzenie pozwalające na połączenie z Android Market. Sklep ten jest dostępny poprzez urządzenie Android Developer Phone, zablokowano w nim jednak dostęp do płatnych aplikacji. Jest to jedno z rozwiązań firmy Google chroniących te aplikacje przed piractwem.

W rozdziale tym zajmiemy się konfigurowaniem procesu publikowania aplikacji w sklepie Android Market, przygotowaniem jej do sprzedaży, uproszczeniem procesu wyszukiwania, pobierania i korzystania z niej przez użytkowników, sposobami zabezpieczania aplikacji przed piractwem, a na końcu zaprezentujemy kilka alternatywnych sposobów udostępnienia programów bez wykorzystania sklepu firmy Google.

## Jak zostać wydawcą?

Zanim umieścimy aplikację w sklepie Android Market, musimy zostać wydawcami. W tym celu należy utworzyć konto programisty (ang. *Developer Account*). Po zarejestrowaniu takiego konta będziemy mogli zamieszczać aplikacje w sklepie Android Market, gdzie będą wyszukiwane i pobierane przez użytkowników. Proces rejestracji konta programisty jest względnie prosty i stosunkowo tani.

Aby cokolwiek opublikować, potrzebne jest konto Google — na przykład konto pocztowe *gmail.com*. Następnie tworzymy tożsamość w sklepie Android Market. W tym celu otwieramy stronę <http://market.android.com/publish/signup>. Wprowadzamy tu imię i nazwisko programisty, adres e-mail, adres strony WWW oraz numer telefonu kontaktowego. Po zarejestrowaniu konta dane te będzie można zmienić. Następnie trzeba uiścić opłatę rejestracyjną. Zajmuje się tym system Google Checkout. Przeprowadzenie całej transakcji wymaga zalogowania się na konto Google.

Jedną z opcji dostępnych podczas procesu płatności jest *Zachowaj poufny charakter mojego adresu e-mail*. Oznosi się to do bieżącej transakcji pomiędzy programistą a usługą Google Android Market, dotyczącej „zakupu” praw wydawcy. Zaznaczenie tej opcji spowoduje ukrycie adresu e-mail przed usługą Google Android Market. Nie ma to nic wspólnego z ukrywaniem adresu e-mail przed potencjalnymi użytkownikami aplikacji. Wybór tej opcji nie ma wpływu na dostępność adresu e-mail dla osób kupujących aplikację. W dalszej części rozdziału rozwinie my ten temat.

Następnie zostanie wyświetlona umowa dotycząca dystrybucji produktów przez ich dewelopera za pośrednictwem usługi Android Market. Jest to legalny kontrakt zawierany pomiędzy programistą a firmą Google. Są w nim określone warunki dystrybucji aplikacji, pobierania i zwrotu płatności, wsparcia i obsługi technicznej, systemu oceniania, praw nabywcy, praw wydawcy i tak dalej. Więcej informacji na temat tych reguł znajdziemy w podrozdziale „Postępowanie zgodnie z zasadami”.

Po zaakceptowaniu umowy ujrzymy stronę znaną powszechnie jako konsola programisty (ang. *Developer Console*) — <http://market.android.com/publish/Home>.

## Postępowanie zgodnie z zasadami

Umowa dotycząca dystrybucji produktów przez ich dewelopera za pośrednictwem usługi Android Market (ang. *Android Market Developer Distribution Agreement* — AMDDA) zawiera mnóstwo reguł postępowania. Być może przed jej zaakceptowaniem należałoby zasięgnąć opinii radcy prawnego, w zależności od zakresu planowanych działań wewnętrz serwisu. W tym punkcie omówimy kilka kwestii wartych odnotowania.

- Aby korzystać ze sklepu Android Market, należy być programistą o nieposzlakowanej reputacji,. Oznacza to, że trzeba przejść przez omówiony powyżej proces rejestracji, zaakceptować umowę i przestrzegać jej zasad. Skutkiem złamania zasad może być zablokowanie dostępu do sklepu i usunięcie z niego naszych aplikacji.
- Możemy dystrybuować zarówno produkty bezpłatnie, jak i za opłatą. Umowa dopuszcza obie możliwości. W przypadku sprzedaży produktów musimy korzystać z takiego procesora płatności, jak na przykład Google Checkout. W momencie wydania platformy Android 2.0 jedną formą pobierania opłat pochodzących ze sklepu Android Market była właśnie usługa Google Checkout. Obecnie użytkownicy mogą

po prostu obciążyć swój rachunek telefoniczny podczas pobierania opłat, co zostało ogłoszone przez firmę T-Mobile w 2009 roku oraz przez firmę AT&T w 2010 roku. W październiku 2010 roku zapowiedziano integrację usługi PayPal z serwisem Android Market, lecz po ponad roku ciągłe brakuje takiej opcji. W przyszłości to podejście może jednak ulec zmianie.

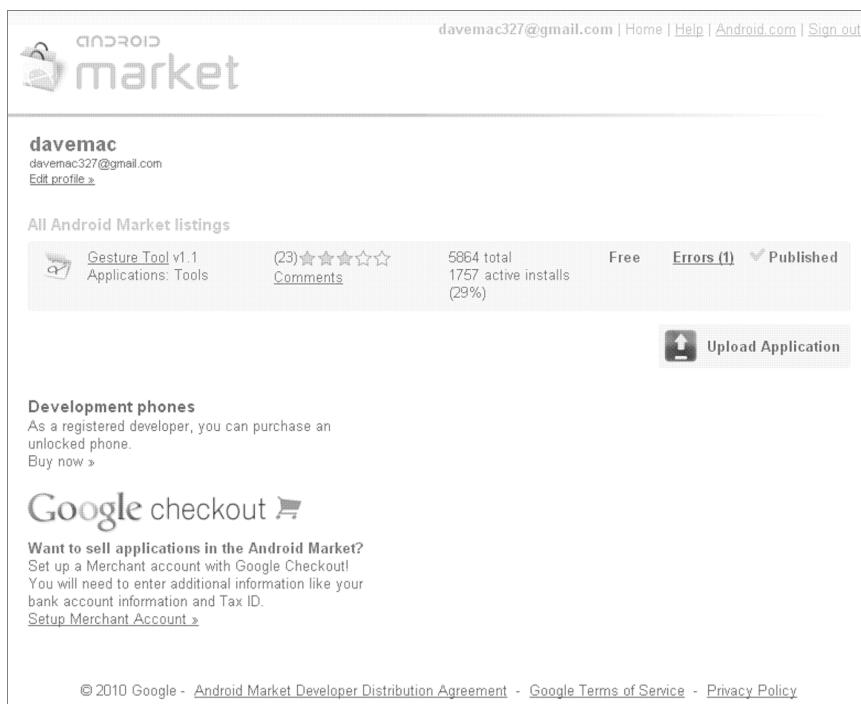
- W przypadku zakupu płatnych aplikacji jest pobierana opłata transakcyjna oraz ewentualnie opłata dla operatora sieci komórkowej, które zostaną odjęte od ceny produktu. W listopadzie 2011 roku opłata transakcyjna wynosi 30%, jeśli więc produkt kosztuje 10 dolarów, firma Google otrzymuje 3 dolary, a my 7 (pod warunkiem że nie dochodzą do tego opłaty dla operatora).
- Do programisty należy obowiązek odprowadzania należnego podatku do właściwych organów podatkowych. Podczas ustanawiania konta handlowego definiuje się odpowiednie wysokości podatku dotyczące zakupu produktu przez osoby znajdujące się w innych rejonach. Usługa Google Checkout pobierze odpowiedni podatek w zależności od tego, w jaki sposób została skonfigurowana. Opłata ta zostanie wysłana do programisty, który musi ją odprowadzić do urzędu skarbowego. Dodatkowe informacje na temat sprzedaży usług i licencji w Polsce można znaleźć pod adresem [http://www.vat.pl/sprzedaz\\_licencji\\_ebooki\\_oprogramowanie\\_firma\\_w\\_internecie\\_1052.php](http://www.vat.pl/sprzedaz_licencji_ebooki_oprogramowanie_firma_w_internecie_1052.php).
- Istnieje możliwość umieszczania w serwisie Android Market darmowej wersji demonstracyjnej aplikacji, posiadającej opcję odblokowania wszystkich funkcji pełnej wersji po wniesieniu opłaty; opłata musi być jednak uiszczena poprzez autoryzowany procesor płatności. Nie możemy skierować użytkowników darmowej aplikacji do innego procesora płatności w celu pobrania opłaty za odblokowanie wszystkich funkcji programu. Zabronione jest również pobieranie opłaty za prenumerowanie aplikacji dystrybuowanych poprzez sklep Android Market. Opłaty za usługi zasadniczo są nawet zalecane, gdyż pomagają chronić aplikację przed piractwem oraz zwiększać obroty twórców oprogramowania. Oznacza to jednak, że nie możemy sprzedawać takiej wersji aplikacji poprzez Android Market. Być może zostanie to zmienione w przyszłości. Pomyślimy o tym w następujący sposób: jeżeli chcemy zarabiać poprzez serwis Android Market, firma Google pragnie mieć swój udział w tych zarobkach.
- W lutym 2011 roku firma Google zapowiedziała wprowadzenie wewnętrzaplakacyjnych mikrotransakcji. Jest to dodatkowy pakiet SDK pozwalający na pobieranie opłat za cyfrowe towary lub dodatkową zawartość programu. Taką zawartością może być nowa broń lub pakiet poziomów w grze bądź pliki graficzne czy muzyczne. Zapłata za takie dodatki wygląda tak samo jak proces kupna aplikacji, co oznacza, że użytkownicy mogą obciążyć swój rachunek telefoniczny.
- Jeżeli nasza aplikacja wymaga od użytkownika posiadania konta w jakimś serwerze sieciowym, z którego można korzystać za dodatkową opłatą abonentową, serwer ten może pobierać tę opłatę w dowolny sposób. Taki sposób odłączenia opłaty abonentowej od aplikacji jest zgodny z umową AMDA — pod warunkiem że bezpłatna wersja aplikacji nie kieruje użytkowników na jej stronę domową. Nie byłoby jednak łatwiej rozprowadzać aplikację z tego samego serwera, na którym znajduje się usługa?
- Okazuje się, że istnieje możliwość wprowadzania alternatywnych sposobów przetwarzania płatności wnoszonych w formie darowizny przeznaczonej na rozwój darmowej wersji aplikacji, nie można jednak wewnątrz programu wprowadzać żadnych bodźców motywujących do uiszczenia takiej opłaty.

- Zwruty płatności są nieprzyjemną stroną serwisu Android Market. Początkowo użytkownikom przysługiwały 24 godziny na zażądanie zwrotu pieniędzy za zakupioną aplikację. Następnie czas ten został wydłużony do 48 godzin. Natomiast w grudniu 2010 roku został zmieniony na 15 minut! Kwadrans ten jest liczony od samego momentu zakupu, a nie od chwili zakończenia pobierania aplikacji. Zdarzały się nawet przypadki, gdy użytkownik nie zdążył jeszcze pobrać aplikacji, a okno czasowe zagwarantowane na zwrot pieniędzy zakończyło działanie. Co dziwne, umowa AMDA nie została zaktualizowana pod tym kątem i ciągle widnieje w niej wpis o 48 godzinach przysługujących na zwrot pieniędzy. Zwruty pieniędzy nie przysługują użytkownikom, którzy mogą przeglądać aplikacje przed ich zakupem. Dotyczy to również dzwonków i tapet. Usługa Google Checkout pozwala jednak deweloperom na zwrot pieniędzy nawet po wyznaczonym terminie, użytkownicy mogą więc mimo wszystko odzyskać pieniądze. Deweloperzy jednak nie mają ochoty na własnoręczne oddawanie pieniędzy.
- Wymagane jest, aby programista zagwarantował odpowiednią pomoc techniczną dotyczącą produktu. Jeżeli pomoc ta nie zostanie zapewniona, użytkownicy mogą żądać zwrotu pieniędzy. Będzie to się wiązać z kosztami dla dewelopera, zwłaszcza że prawdopodobnie będą w to wliczone również koszty manipulacyjne.
- Użytkownicy mają prawo do nieograniczonej liczby ponownych instalacji aplikacji pobranych ze sklepu Android Market. W przypadku przywrócenia ustawień fabrycznych urządzenia użytkownik będzie mógł pobrać wszelkie zakupione aplikacje bez konieczności ponownego płacenia za nie.
- Wydawca gwarantuje ochronę prywatności i praw użytkowników. Dotyczy to ochrony (na przykład zabezpieczania) wszelkich danych, które mogą zostać zebrane podczas użytkowania aplikacji. Zmiana warunków dotyczących ochrony danych użytkownika jest dopuszczalna jedynie w przypadku wyświetlenia odpowiedniej umowy i zaakceptowania jej przez użytkownika.
- Aplikacja nie może konkurować ze sklepem Android Market. Firma Google nie pozwala na umieszczenie aplikacji umożliwiających sprzedaż innych produktów poza sklepem Android Market, co jest równoznaczne z ominięciem procesora płatności. Nie oznacza to wcale, że nie można sprzedawać tej aplikacji innymi kanałami, lecz że ta aplikacja, po jej umieszczeniu w sklepie Android Market, nie może umożliwiać sprzedaży produktów znajdujących się poza tą usługą.
- Umieszczone produkty zostaną objęte systemem oceniania. Oceny mogą być przydzielane na podstawie udzielanej pomocy technicznej, szybkości instalacji i odinstalowania aplikacji, szybkości zwrotu kosztów oraz (lub) jako tak zwana ocena ogólna dewelopera (ang. *Developer Composite Score*). Jest ona szacowana na podstawie ocen wystawianych dla poprzednich aplikacji i może wpływać na ocenę przyszłych produktów. Z tego powodu istotne jest, aby wydawać aplikacje wysokiej jakości, nawet jeśli są darmowe. Nie jesteśmy pewni, czy ocena ogólna dewelopera w ogóle istnieje, jeśli jednak tak — nie mamy w nią wglądu.
- Poprzez sprzedaż aplikacji w sklepie Android Market udzielamy użytkownikowi „niewyłącznej i ogólnoświatowej licencji na odtwarzanie, prezentowanie i użytkowanie Produktu w Urządzeniu”. Jednak nic się nie stanie, jeśli napiszemy własne warunki umowy licencyjnej (ang. *End User License Agreement* — EULA), zastępujące powyższe stwierdzenie. Należy taką umowę umieścić na własnej stronie WWW lub zagwarantować jakiś inny sposób jawnego zaprezentowania jej użytkownikom.

- Firma Google wymaga przestrzegania cech marki Android. Do tych cech zaliczają się ograniczenia w wykorzystywaniu słowa „Android”, jak również ikony robota, znaku towarowego oraz kroju pisma. Informacje na ten temat znajdziemy pod adresem <http://www.android.com/branding.html>.

## Konsola programisty

Konsola programisty jest naszą stroną docelową, pozwalającą na kontrolowanie aplikacji umieszczonych w serwisie Android Market. Z poziomu konsoli programisty możemy zakupić urządzenie ADP (ang. *Android Developer Phone* — telefon programisty systemu Android), skonfigurować konto handlowe w usłudze Google Checkout (dzięki czemu możemy naliczać opłatę za aplikację), publikować aplikacje oraz przeglądać informacje na temat opublikowanych programów. Możemy także edytować takie szczegóły konta, jak imię i nazwisko programisty, adres e-mail, adres strony WWW oraz numer telefonu. Konsola programisty została zaprezentowana na rysunku 28.1.



**Rysunek 28.1.** Konsola programisty pozwalająca uzyskać dostęp do serwisu Android Market

Obecnie istnieją trzy rodzaje telefonów testowych obsługujących system Android: Android Developer Phone, Google Nexus One oraz Google Nexus S. Android Developer Phone (ADP) był przez długi czas jedynym telefonem pozwalającym na testowanie programowanych aplikacji. Jest to specjalne urządzenie, zaprojektowane przede wszystkim dla programistów aplikacji dla systemu Android. Jest to profesjonalny telefon, posiadający odblokowane wszystkie funkcje, niezależny od operatorów telefonii komórkowej. Akceptuje wszystkie rodzaje kart SIM, a w jego wyposażeniu znajduje się karta pamięci 1 GB, aparat fotograficzny, wysuwana klawiatura i system GPS. Pisząc o *odblokowanych* funkcjach, mamy na myśli możliwość wykonania każdej

czynności w urządzeniu, łącznie z instalacją nowej wersji oprogramowania sprzętowego i systemu Android, nie tylko aplikacji. Chociaż możemy instalować nowe wersje oprogramowania sprzętowego, fabryczna wersja tego urządzenia jest wyposażona w system Android 1.6.

Być może Czytelnik pamięta czasy wydania pierwszego telefonu firmy Google, jakim jest Nexus One. Chociaż był zaprojektowany we współpracy z firmą HTC, z powodu słabych wyników sprzedaży zaprzestano produkcji tego telefonu, a następnie przerobiono go na telefon testowy, pozwalający użytkownikom na sprawdzanie działania tworzonych aplikacji. Został w tej roli bardzo dobrze przyjęty, więc firma Google zwiększyła zamówienia na produkcję tego modelu. Specyfikacja techniczna telefonu Nexus One robi wrażenie. Został on zaopatrzony w czujnik zbliżenia, czujnik oświetlenia, akcelerometr oraz kompas. Bez większego problemu obsługuje system Android 2.2 i jeszcze przez pewien czas nie powinien mieć problemu z nowszymi wersjami systemu. Specyfikacja tego telefonu jest dostępna pod adresem <http://www.htc.com/us/support/nexus-one-google/tech-specs/>. Wśród wad należy wymienić brak obsługi technologii 3G, co zmusza do korzystania z innych form sieci bezprzewodowych (AT&T, 2G lub EDGE).

W grudniu 2010 roku firma Google rozpoczęła sprzedaż telefonu Nexus S, którego jednak nie można zamówić z poziomu konsoli programisty. Został po raz pierwszy udostępniony w sklepie Best Buy (USA) oraz w Carphone Warehouse (Wielka Brytania) i może zostać zakupiony bez umowy z operatorem sieci komórkowej. Urządzenie to jest jeszcze szybsze i bardziej zaawansowane od telefonu Nexus One, posiada wbudowany żyroskop, czujnik NFC, a także aparat umieszczony z przodu. Jest obsługiwany przez system Android 2.3 (Gingerbread). Specyfikacja, wraz w filmami demonstracyjnymi, jest dostępna pod adresem [www.google.com/nexus/#](http://www.google.com/nexus/#).

Jeżeli chcemy testować nowe wersje oprogramowania sprzętowego lub sam system Android, potrzebne nam będzie urządzenie testowe. Z trzech dostępnych rodzajów telefonów najlepszym wyborem jest Nexus S. Jednym z pozytywnych aspektów telefonów z grupy Nexus jest pierwszeństwo w aktualizowaniu ich do nowej wersji systemu. Jest to jeden z powodów, dla których warto wybrać telefon Nexus zamiast zwykłego telefonu związanego z operatorem. Jeżeli chcemy jedynie pisać aplikacje, a nie modyfikować w jakiś sposób system Android, powiniem nam wystarczyć zwykły telefon. Każde urządzenie obsługujące system Android może zostać podłączone do stacji roboczej w celach projektowych i testowych. Musimy jednak zwracać uwagę na specyfikację techniczną. Nie wszystkie telefony mogą zostać zaktualizowane do najnowszej wersji systemu, dotyczy to zwłaszcza mniej wydajnych modeli.

Jeśli nie skonfigurujemy konta handlowego za pomocą usługi Google Checkout, nie będziemy mogli pobierać opłat za produkty umieszczone w sklepie Android Market. Ustanowienie takiego konta nie jest skomplikowaną czynnością. Wystarczy kliknąć odpowiednie łącze w konsoli programisty, wypełnić formularz aplikacji, zaakceptować warunki korzystania z usługi i to będzie wszystko. Należy mieć przygotowany pod ręką numer karty kredytowej. Informacje o karcie kredytowej są wprowadzane w celu uiszczenia zwrotu zapłaty, w przypadku gdy na koncie Google Checkout nie ma wystarczającej ilości środków. Możemy także wprowadzić dane konta bankowego, dzięki czemu zyski ze sprzedaży aplikacji będą przenoszone na to konto. Zwróćmy uwagę, że usługa Google Checkout obsługuje nie tylko sklep Android Market. Nie powinniśmy więc się zdziwić, jeśli pojawi się informacja o opłacie transakcyjnej za sprzedaż pochodząą spoza sklepu Android Market. Wspomniana opłata transakcyjna wynosząca 30% ceny aplikacji jest obliczona dla sklepu Android Market. Istnieją również dodatkowe opłaty transakcyjne dla sprzedaży przeprowadzanych poza tą witryną, niezależne od wspomnianej powyżej opłaty.

Prawdopodobnie najczęściej wykorzystywanymi funkcjami konsoli programisty będą umieszczanie i monitorowanie aplikacji. W dalszej części rozdziału zajmiemy się procesem publikowania aplikacji w sklepie. W kwestii monitoringu otrzymujemy narzędzia pozwalające obserwować całkowitą liczbę pobrań aplikacji oraz liczbę użytkowników, którzy zainstalowali aplikację. Widoczna jest ogólna ocena programu w zakresie od 0 do 5 gwiazdek, a także liczba osób, które wystawiły ocenę. Z poziomu konsoli programisty możemy ponownie opublikować aplikację — na przykład jej aktualizację — lub wycofać ją ze sklepu. Ta ostatnia czynność nie usuwa aplikacji z urządzeń, a nawet nie musi jej usuwać z serwerów Google, dotyczy to zwłaszcza płatnych aplikacji. Użytkownik, który zapłacił za aplikację, a następnie ją odinstalował, lecz nie zażądał zwrotu kosztów, ma prawo do jej ponownego zainstalowania, nawet jeśli została ona wycofana z obiegu. Program przestaje być naprawdę dostępny dla użytkowników jedynie w przypadku złamania zasad firmy Google. W marcu 2011 roku firma Google dodała w konsoli programisty zestawienia i wykresy pozwalające na obserwowanie statystyk aplikacji w zależności od wersji systemu operacyjnego, rodzaju urządzenia, a także krajów i języków.

Oprócz oceniania aplikacji użytkownicy mogą również pozostawiać komentarze. Dla własnego dobra powinniśmy jak najczęściej czytać komentarze dotyczące naszej aplikacji, aby móc szybko rozwijać zauważone problemy. Wraz z komentarzem dostępne są ocena aplikacji, nazwa użytkownika oraz data umieszczenia komentarza. Niestety, nie możemy bezpośrednio odpowiadać na komentarze ani nawet umieszczać komentarzy pod komentarzami użytkowników. W ekstremalnych przypadkach, gdy treść komentarza jest wyjątkowo szkodliwa lub niecenzuralna, możemy zgłosić to obsłudze firmy Google, dostępnej pod adresem <http://market.android.com/support/>.

Możemy również przeglądać komunikaty o błędach wygenerowane przez aplikację oraz sprawdzać, w których momentach ulegała zawieszeniu lub zostawała niespodziewanie zamknięta. Na rysunku 28.2 widzimy ekran raportów o błędach aplikacji.

Freezes		Crashes	
0 new	0 reports	1 new	0 reports/week
0 old	0 reports	0 old	0 reports/week

© 2010 Google - Android Market Developer Distribution Agreement - [Google Terms of Service](#) - [Privacy Policy](#)

Rysunek 28.2. Ekran Application Error Reports

Przeglądając szczegóły raportu, możemy dotrzeć do śladu stosu, w którym nastąpiła awaria, jak również uzyskać takie informacje, jak rodzaj urządzenia, na którym działała aplikacja, oraz data awarii. Podobnie jednak jak ma to miejsce w przypadku komentarzy, nie możemy skomunikować się z użytkownikiem, którego aplikacja uległa awarii, aby poznać dalsze szczegóły lub

powrócić mu wyjść z opresji. Pozostaje nam nadzieję, że tacy użytkownicy napiszą na nasz adres e-mail lub zostawią informację na stronie aplikacji. W przeciwnym wypadku jesteśmy zdani na siebie i musimy samodzielnie wykryć przyczynę problemu oraz spróbować ją naprawić.

Dostępna jest jeszcze jedna funkcja konsoli programisty, która może się okazać przydatna — odnośnik do materiałów pomocniczych. Przycisk *Help* znajduje się w prawym górnym rogu ekranu. Kliknięcie go spowoduje otwarcie witryny pomocy, zawierającej porządną dokumentację dotyczącą serwisu Android Market, a także forum, na którym możemy poszukać odpowiedzi na dręczące nas pytania i umieszczać własne porady. To na forum znajdziemy informacje dotyczące najnowszych zasad zwrotu pieniędzy, problemów i zażeleń. Jeżeli forum nie okaże się przydatne, znajdziemy tu odnośnik do pomocy technicznej (*Contacting Support*), za pomocą którego możemy wysłać wiadomość wprost do przedstawicieli firmy Google.

Omówiliśmy niektóre z przydatnych funkcji konsoli programisty, jednak Czytelnik z pewnością nie może się doczekać najprzydatniejszej części rozdziału — omówienia procesu umieszczania aplikacji w serwisie Android Market. Dzięki temu procesowi użytkownicy będą mogli znaleźć i pobrać aplikację. Zanim jednak przejdziemy do tego etapu, musimy powiedzieć, w jaki sposób odpowiednio przygotować nasz program do pobierania i sprzedaży.

## Przygotowanie aplikacji do sprzedaży

Po utworzeniu kodu aplikacji, ale jeszcze przed jej umieszczeniem w sklepie Android Market, należy przeprowadzić kilka czynności przygotowawczych. Poświęcimy im teraz trochę uwagi i omówimy kolejne czynności, które trzeba wykonać.

## Testowanie działania na różnych urządzeniach

Bardzo istotne jest, aby przy lawinowej produkcji coraz to nowych urządzeń pracujących pod kontrolą systemu Android, z których każde zawiera potencjalnie odmienną konfigurację sprzętową, utworzona aplikacja została przetestowana pod kątem działania na docelowych telefonach. W idealnym przypadku programista ma dostęp do każdego rodzaju telefonu, który chce przetestować. Jest to dość kosztowna propozycja. Innym dobrym rozwiązaniem jest wykorzystanie urządzeń AVD dla każdego rodzaju telefonu poprzez utworzenie odpowiedniej konfiguracji sprzętowej, a następnie uruchomienie i przetestowanie takiego urządzenia na emulatorze. Niektórzy producenci urządzeń udostępniają pakiety Androida specyficzne dla danego telefonu, warto zatem przeglądać witryny sieciowe danej firmy. Zestaw Android SDK posiada klasę *Instrumentation* oraz program *UI/Application Exerciser Monkey*, usprawniające proces testowania. Wymienione narzędzia umożliwiają automatyzację testowania, nie musimy więc marnować czasu na powtarzanie tych samych czynności. Przed rozpoczęciem testowania powinniśmy usunąć zbędne artefakty z kodu oraz z folderu */res*. Chcemy przecież, aby aplikacja była jak najmniejsza oraz jak najszybsza przy minimalnym zużyciu pamięci.

## Obsługa różnych rozmiarów ekranu

W momencie wydania środowiska Android SDK 1.6 programiści zaczęli się borykać z nowymi rozmiarami wyświetlaczy. Aby uruchomić aplikację na nowym, mniejszym ekranie, należy ustawić swoisty element *<supports-screens>* jako element potomny węzła *<manifest>* w pliku *AndroidManifest.xml*. Bez wprowadzenia tego znacznika definiującego obsługę małych ekranów przez aplikację nie będzie ona dostępna w sklepie Android Market dla urządzeń posiadają-

cych niewielkie wyświetlacze. Oznacza to oczywiście, że aplikacja musi zostać skompilowana wobec środowiska Android SDK co najmniej w wersji 1.6. Jeśli chcemy, aby program działał w urządzeniach obsługujących starsze wersje zestawu Android SDK, musimy się upewnić, że nie będzie korzystał z żadnego interfejsu API wprowadzonego w wersji 1.6 lub nowszej oprogramowania. Należy następnie przetestować aplikację zarówno na urządzeniach AVD imitujących starsze telefony, jak i reprezentujących nowsze urządzenia. W celu obsługi różnych rozmiarów ekranu prawdopodobnie będziemy musieli utworzyć alternatywne pliki zasobów w podkatalogu `/res`. Na przykład w przypadku dodatkowej obsługi niewielkich wyświetlaczów oprócz plików znajdujących się w katalogu `/res/layout` trzeba będzie umieścić odpowiedniki tych plików w katalogu `/res/layout-small`. Nie oznacza to, że musimy tworzyć również odpowiedniki tych plików w katalogach `/res/layout-large` i `/res/layout-normal`, gdyż jeśli Android nie znajdzie takiego specyficznego katalogu, jak na przykład `/res/layout-large`, wykorzysta zasoby dostępne w katalogu `/res/layout`. Pamiętajmy również, że możemy tworzyć kombinacje kwalifikatorów dla plików zasobów — na przykład katalog `/res/layout-small-land` może zawierać układy graficzne dla małych ekranów zorientowanych w trybie poziomym. Omówiliśmy to zagadnienie w rozdziale 6. Obsługa małych wyświetlaczów oznacza prawdopodobnie również utworzenie alternatywnych wersji obiektów rysowanych, takich jak ikony. W przypadku tych obiektów może również zastępować potrzeba utworzenia alternatywnych katalogów zasobów, odpowiadających rozdzielczości ekranu oraz jego rozmiarowi.

Oczywiście, pod względem rozmiarów ekranu tablety podążają w przeciwnym kierunku i w ich przypadku stosujemy wartość `xlarge`. Ten sam znacznik `<supports-screens>` służy do definiowania aplikacji uruchamianej na bardzo dużym ekranie, natomiast wprowadzonym atrybutem jest `android:xlargeScreens`. W niektórych przypadkach możemy mieć do czynienia z aplikacją obsługiwana wyłącznie przez tablety, wtedy należy przypisać wartość `false` wszystkim pozostałym rozmiarom ekranu.

## Przygotowanie pliku `AndroidManifest.xml` do umieszczenia w sklepie Android Market

Plik `AndroidManifest.xml` prawdopodobnie powinien zostać troszeczkę zmodyfikowany przed umieszczeniem go w sklepie Android Market. Domyslnie narzędzia ADT środowiska Eclipse wstawiają atrybut `android:icon` do znacznika `<application>`, a nie do znaczników `<activity>`. Jeżeli istnieje możliwość uruchomienia większej liczby aktywności, powinniśmy dla każdej z nich ustanowić oddzielną ikonę, aby użytkownik mógł je łatwiej rozróżniać. Ciągle jednak musi być określona jedna ikona w znaczniku `<application>`, która posłuży jako domyślna ikona dla wszystkich aktywności nieposiadających własnej ikony. Aplikacja posiadająca atrybut `android:icon` wyłącznie wewnętrz węzła `<activity>` będzie bezproblemowo działała w urządzeniach oraz na emulatorze, jednak podczas umieszczania aplikacji na serwerze usługa Android Market przeszukuje znacznik `<application>` pod kątem informacji o ikonie. Przesyłanie aplikacji zostaje przerwane również w przypadku, gdy nazwa zastosowanego pakietu rozpoczyna się od ciągów znakowych `com.google`, `com.android`, `android` lub `com.example`, mamy jednak nadzieję, że nie zostały one zastosowane w aplikacjach Czytelników.

Istnieje również wiele innych kwestii związanych z kompatybilnością, które należy wziąć pod uwagę podczas testowania aplikacji na różnych konfiguracjach sprzętowych. Niektóre urządzenia posiadają aparat fotograficzny, inne nie mają klawiatury fizycznej, jeszcze inne mają manipulator kulkowy zamiast klawiszy nawigacyjnych. W razie potrzeby zastosujmy znaczniki `<uses-configuration>` i `<uses-feature>` w pliku `AndroidManifest.xml` do zdefiniowania

wymagań sprzętowych i programowych naszej aplikacji. Android Market wymusi te wymagania i uniemożliwi wyświetlanie naszej aplikacji urządzeniom nieposiadającym odpowiedniej konfiguracji. Zwróćmy uwagę, że mamy tu do czynienia z innymi znacznikami niż `<uses-permission>` pliku *AndroidManifest.xml*. Chociaż urządzenie użytkownika może być wyposażone w aparat fotograficzny, nie jest wcale powiedziane, że użytkownik zechce przydzielić naszej aplikacji dostęp do tego aparatu. Jednocześnie zadeklarowanie uprawnienia aplikacji do korzystania z aparatu wcale nie musi oznaczać, że obecność tej funkcji jest wymagana przez aplikację. W większości przypadków będziemy umieszczać obydwa znaczniki w pliku *AndroidManifest.xml*, aby określić wymóg obecności aparatu fotograficznego oraz uprawnienie korzystania z aparatu w razie potrzeby. Jednak nie wszystkie funkcje wymagają uprawnień, więc w naszym najlepszym interesie leży zdefiniowanie opcji potrzebnych naszej aplikacji.

Istnieje jeszcze jedna zasadnicza różnica pomiędzy węzłami `<uses-permission>` a `<uses-feature>`. Za pomocą tego drugiego znacznika możemy określić, czy dana funkcja jest konieczna do działania aplikacji lub czy nasz program może się bez niej obejść. To znaczy, że mamy do dyspozycji atrybut `android:required`, któremu możemy przypisać wartość `true` lub `false` (domyślna jest wartość `true`). Na przykład program może wykorzystywać funkcje sieci Bluetooth, jeżeli jest dostępna, będzie jednak działał równie dobrze bez nich. Zatem możemy wstawić w pliku manifeście następujący wiersz:

```
<uses-feature android:name="android.hardware.bluetooth" android:required="false" />
```

Wewnątrz kodu aplikacji powinniśmy wywołać klasę `PackageManager` w celu określenia, czy technologia Bluetooth jest dostępna, czego możemy dokonać za pomocą poniższego fragmentu:

```
boolean hasBluetooth = getPackageManager().hasSystemFeature(  
    PackageManager.FEATURE_BLUETOOTH);
```

Następnie możemy wykonać odpowiednią czynność, w zależności od obecności funkcji Bluetooth. Dokumentacja Androida w tym miejscu jest dość niejednoznaczna. Jeżeli zajrzymy do informacji o znaczniku `<uses-feature>` w konsoli programisty, nie znajdziemy tak wielu funkcji jak w dokumentacji klasy `PackageManager`, w której zostały zdefiniowane stałe `FEATURE_*` dla każdej dostępnej funkcji.

Znacznik `<uses-configuration>` jest nieco inny. Definiuje on rodzaje wymaganych elementów urządzenia, takich jak rodzaj klawiatury, ekran dotykowy czy mechanizm sterowania. Jednak w przeciwieństwie do węzła `<uses-feature>` nie wprowadzamy tutaj niezależnych wyborów, lecz tworzymy kombinacje konfiguracji wymaganych przez naszą aplikację. Jeżeli na przykład nasz program wymaga obecności kontrolera ruchu w postaci podkładki kierunkowej lub manipulatora kulkowego oraz ekranu dotykowego (wykorzystującego rysik lub palec), możemy zdefiniować dwa następujące wiersze:

```
<uses-configuration android:reqFiveWayNav="true" android:reqTouchScreen="stylus" />  
<uses-configuration android:reqFiveWayNav="true" android:reqTouchScreen="finger" />
```

## Lokalizacja aplikacji

Jeśli aplikacja będzie wykorzystywana w innych krajach, możemy rozważyć proces jej lokalizacji. Z technicznego punktu widzenia jest to względnie prosta czynność. Znalezienie osoby odpowiedzialnej za przeprowadzenie tego procesu to zupełnie inna sprawa. Z technicznego punktu widzenia tworzymy po prostu kolejny folder w katalogu `/res` — na przykład `/res/values-fr` przechowujący francuską wersję pliku *strings.xml*. Następnie bierzemy plik *strings.xml*, tłumaczymy wartości typu `string` na nowy język i zachowujemy tak zmodyfikowany plik w nowym folderze

zasobów, nie zmieniając jednocześnie nazwy tego pliku. W przypadku innych rodzajów zasobów — na przykład obiektów rysowanych lub menu — stosujemy dokładnie taką samą technikę. Obrazy i kolory mogą lepiej spełniać swoje zadanie, jeśli będą przystosowane do odmiennych krajów i kultur. Z tego właśnie powodu nie warto stosować prawdziwych nazw zasobów kolorów. W internetowej dokumentacji dotyczącej kolorów często można natrafić na taki zapis:

```
<color name="solid_red">#f00</color>
```

Oznacza on, że w takim kodzie lub innym pliku zasobów odnosimy się do koloru za pomocą jego rzeczywistej nazwy, w naszym przypadku jest to `solid_red`. Aby dostosować kolor do innego kraju lub kultury, najlepiej stosować nazwy kolorów typu `accent_color1` lub `alert_color`. W Wielkiej Brytanii odpowiedniejszy może być kolor czerwony, podczas gdy w Hiszpanii swoje zadanie będzie lepiej spełniał jakiś odcień żółci. Ponieważ nazwa `alert_color` nie określa zastosowanego koloru, jego zmiana nie jest już tak bardzo dezorientująca. Jednocześnie możemy zaprojektować przyjemny schemat kolorystyczny, zawierający barwy bazowe i ich odcienie, mając jednocześnie pewność, że właściwe kolory zostały użyte we właściwych miejscach.

W niektórych krajach opcje menu powinny zostać zmienione poprzez dodanie lub usunięcie pewnych elementów, ewentualnie można wprowadzić inną organizację menu, w zależności od miejsca stosowania aplikacji. Pliki menu są zazwyczaj przechowywane w katalogu `/res/menu`. Jeżeli natrafimy na taką sytuację, prawdopodobnie będzie lepiej, jeśli umieścimy wszystkie tekstowe ciągi znaków w pliku `strings.xml` lub innym pliku przechowywanym w podkatalogu `/res/values`, a następnie będziemy wszędzie odnosić się do tych zasobów za pomocą ich identyfikatorów. Zmniejszymy w ten sposób znacznie niebezpieczeństwo pominienia tłumaczenia tekstu w jakimś zapomnianym pliku zasobów. Tłumaczenie jest wtedy zatem ograniczone do plików znajdujących się w podkatalogu `/res/values`.

## Przygotowanie ikony aplikacji

Osoby kupujące oraz użytkownicy będą wyraźnie widzieć ikonę oraz etykietę aplikacji zarówno w sklepie Android Market, jak i — po jej pobraniu — w urządzeniu. Powinniśmy poświęcić szczególną uwagę utworzeniu jak najlepszej ikony i etykiety swojej aplikacji oraz jej aktywności. W razie potrzeby możemy je również zlokalizować. Nie zapominajmy również o ewentualnym dostosowaniu rozmiarów ikon do rozmiarów ekranu. Przyglądajmy się dzielom innych wydawców, szczególnie zaś aplikacjom należącym do tej samej kategorii co nasza. Chcemy, aby nasza aplikacja była widoczna, musimy więc unikać wtapiania się w tłum. Jednocześnie musimy pamiętać, że ikona i etykieta aplikacji muszą harmonizować z ikonami innych aplikacji zainstalowanych w urządzeniu użytkownika. Użytkownik nie może zastanawiać się nad przeznaczeniem aplikacji, której ikona wskazuje zupełnie inną funkcję.

Podczas tworzenia dowolnego obrazu wykorzystywanego w aplikacji, zwłaszcza jej ikony, musimy brać pod uwagę gęstość ekranu w docelowym urządzeniu. Gęstość jest definiowana jako liczba pikseli na cal. Mały ekran zazwyczaj oznacza również małą gęstość, wobec czego mniej pikseli składa się na jednostkę odległości, podczas gdy większe wyświetlacze często posiadają dużą gęstość. W przypadku ekranu posiadającego niewielką gęstość odpowiedni rozmiar ikony powinien składać się z mniejszej liczby pikseli, najczęściej o wymiarach  $36 \times 36$ . W przypadku wyświetlacza o większej gęstości najprawdopodobniej utworzymy ikonę o wymiarach  $72 \times 72$ . Ikona dla ekranu o średniej gęstości przybiera wartości  $48 \times 48$ , a przy bardzo dużych gęstościach może posiadać rozmiary  $96 \times 96$ .

## Problemy związane z zarabianiem pieniędzy na aplikacjach

Podczas ustalania ceny za aplikację trzeba rozważyć pewne kwestie. Czy utworzyć dwie wersje tej samej aplikacji — darmową i płatną — wymagające oddzielnej obsługi i zarządzania nimi? A może korzystamy ze wspólnego kodu bazowego i stosujemy jakąś technikę, dzięki której wiadomo, że została uiszczena opłata za program? Bez względu na zastosowane rozwiązańe, w jaki sposób chronimy aplikację przed jej kopiowaniem i instalowaniem na innych urządzeniach przez nieupoważnione do tego osoby? Z powodu ograniczonych zabezpieczeń stosowanych w telefonach oraz zdolności pewnych osób do omijania tych środków ostrożność zarządzanie niezawodnymi technologiami ochrony przed nieuprawnionym kopiowaniem jest niezwykle trudne.

Jednym z rozwiązań utrzymywania pojedynczego kodu bazowego, pozwalającego na umieszczenie oddzielnych trybów (bezpłatnego i płatnego), jest wykorzystanie możliwości klasy `PackageManager`:

```
this.getPackageManager().checkSignatures(mainAppPkg, keyPkg)
```

Metoda ta porównuje sygnatury dwóch pakietów oraz przekazuje wartość `PackageManager`.  
→`SIGNATURE_MATCH`, jeżeli obydwa pakiety istnieją i są identyczne. Nazwy pakietów muszą się różnić dla każdej aplikacji współistniejącej w sklepie Android Market, ale to nic nie szkodzi. W naszym kodzie, jeżeli chcemy zadecydować o udostępnieniu wszystkich funkcji aplikacji, możemy wywołać tę metodę i wprowadzić nazwę pakietu aplikacji głównej lub aplikacji od-blokowującej. Ten drugi program ustanawiamy następnie jako płatny. Jeżeli użytkownik zakupi aplikację od-blokowującą i pobierze ją na urządzenie, główna aplikacja otrzyma dopasowanie sygnatury i od-blokuje dodatkowe funkcje. Nieco mniej przyjemnym rozwiązaniem wykorzystującym pojedynczy kod bazowy jest wprowadzenie systemów oznaczających kolejne wersje kodu źródłowego, które skonfiguruja odpowiednie współdzielenie wspólnych elementów, oraz stworzenie skryptów tworzących darmowe i płatne wersje aplikacji.

Kolejnym rozwiązaniem umożliwiającym zarabianie na aplikacjach jest wprowadzenie wewnętrznych reklam. Istnieje mnóstwo okazji, aby dołączyć reklamy do aplikacji. Dwoma powszechnie występującymi możliwościami są AdMob i AdSense. Proces polega przede wszystkim na wstawieniu ich pakietów SDK do naszej aplikacji, określeniu odpowiedniego miejsca i czasu, w którym będą wyświetlane reklamy i dodaniu uprawnienia INTERNET do programu (dzięki czemu wyświetlone reklamy będą pobierane z internetu). Na koniec będziemy otrzymywać pieniądze za każde kliknięcie reklamy. Sama aplikacja może być bezpłatna, dzięki czemu łatwiej będzie ją umieścić w serwisie Android Market, a do tego nie musimy się tak bardzo przejmować kwestią piractwa. Wielu programistów przyznaje, że zarabia w ten sposób przyzwoite pieniądze.

Kolejną nową funkcją jest wprowadzona w lutym 2011 roku waluta klienta (ang. *Buyer's Currency*). Przedtem klienci — użytkownicy — musieli płacić w walucie sprzedawcy, co mogło stanowić problem dla osób mających kłopoty z wymianą waluty sprzedawcy na swoją własną. Oznaczało to również, że sprzedawca mógł tak naprawdę ustalić tylko jedną cenę dla użytkowników na całym świecie. Teraz, gdy sprzedawca może ustalić cenę dla danego kraju, może ją nie tylko podnosić lub obniżać w zależności od rejonu geograficznego, lecz także komfort użytkownika staje się wyraźnie większy.

## Kierowanie użytkowników z powrotem do sklepu

W systemie Android wprowadzono nowy schemat identyfikatorów URI, ułatwiający wyszukiwanie aplikacji w sklepie Android Market — `market://`. Jeśli na przykład chcemy skierować użytkowników do sklepu w celu znalezienia potrzebnego składnika lub dokupienia dodatkowej

aplikacji odblokowującej nowe funkcje naszej aplikacji, powinniśmy wprowadzić następujący kod, w którym w miejsce MY\_PACKAGE\_NAME wprowadzamy nazwę naszego pakietu:

```
Intent intent = new Intent(Intent.ACTION_VIEW,
    Uri.parse("market://search?q=pname:MY_PACKAGE_NAME"));
startActivity(intent);
```

Z pomocą tego kodu zostanie uruchomiona aplikacja Market, która wyświetli użytkownikowi nazwę tego pakietu. Użytkownik może wtedy pobrać lub zakupić aplikację. Zwróćmy uwagę, że powyższy schemat nie działa w standardowej przeglądarce WWW. Możemy wyszukiwać za pomocą nazwy pakietu (pname), a także szukać wydawcy za pomocą wyrażenia market://search?q=pub:\\"Fname Lname\\" lub przy użyciu dowolnego pola publicznego sklepu Android Market (nazwa aplikacji, nazwa wydawcy oraz opis aplikacji), wpisując market://search?q=<querystring>.

Jeżeli Czytelnik połączy zdobytą wiedzę z technikami omówionymi w poprzednim podręczniku, powinien umieć napisać kod, który będzie próbował odnaleźć pakiet odblokowujący w danym urządzeniu. Jeśli pakiet nie zostanie znaleziony, możemy wyświetlić monit i zapytać użytkownika, czy nie chce go pobrać z internetu. W przypadku odpowiedzi twierdzącej aplikacja wywoła intencję otwierającą usługę Android Market i automatycznie uruchamiającą stronę, na której będzie można pobrać lub zakupić aplikację odblokowującą.

## Usługa licencyjna systemu Android

Architektura aplikacji tworzonych dla systemu Android powoduje, niestety, że stanowią one łatwy cel dla piratów. Istnieje możliwość kopiowania tych aplikacji oraz rozsyłania tych kopii do innych urządzeń. W jaki sposób możemy więc sprawić, aby użytkownicy, którzy nie zakupili naszej aplikacji, nie mogli jej uruchomić? Nasze wymagania spełnia utworzona przez firmę Google biblioteka LVL (ang. *License Verification Library* — biblioteka weryfikująca licencje).

Jeżeli dana aplikacja została pobrana za pomocą usługi Android Market, musi istnieć kopia aplikacji Android Market w tym urządzeniu. Dodatkowo aplikacja ta musiała wprowadzić uprawnienia pozwalające na odczytywanie wartości przechowywanych w urządzeniu, takich jak nazwa konta Google użytkownika, numer IMSI oraz inne informacje. Począwszy od wersji 1.5 Androida, aplikacja Android Market została zmodyfikowana w taki sposób, że reaguje na żądania weryfikacji licencji pochodzące od aplikacji. W tym celu wywołujemy bibliotekę LVL z poziomu aplikacji, biblioteka ta nawiązuje łączność z aplikacją Android Market, z kolei program Android Market łączy się z serwerami firmy Google i nasza aplikacja otrzymuje odpowiedź wskazującą, czy użytkownik urządzenia posiada licencję na korzystanie z naszego kodu. Posiadamy kontrolę nad ustaleniami definiującymi zachowanie aplikacji w przypadku braku dostępu do sieci. Pełny opis implementacji biblioteki LVL znajdziemy pod adresem <http://developer.android.com/guide/publishing/licensing.html>.

Powinniśmy mieć jednak świadomość, że mechanizm LVL jest narażony na ataki hakerów. Jeżeli ktoś wie, gdzie znaleźć wartość przekazywaną w wyniku wywołania biblioteki LVL, i jeśli uzyska dostęp do naszego pliku .apk, może rozłożyć aplikację na czynniki pierwsze i odpowiednio ją zmodyfikować. Jeżeli zastosujemy oczywiste rozwiązanie polegające na wykorzystaniu instrukcji switch po otrzymaniu odpowiedzi z mechanizmu LVL, które służy do określenia zachowania aplikacji w zależności od otrzymanej wartości, haker może po prostu wymusić przekazanie tej pożąanej wartości i w tym momencie zabezpieczenia takiej aplikacji przestają istnieć. Z tego właśnie powodu twórcy Androida zalecają jak największe zagmatwianie kodu,

aby ukryć logikę odpowiedzialną za sprawdzanie wartości zwracanej przez bibliotekę LVL. Możemy sobie wyobrazić, że jest to dość skomplikowana procedura.

Wraz z wersją 2.3 Androida firma Google wprowadziła pewną formę obsługi takiego gmatwania kodu w postaci funkcji ProGuard. Jeżeli ustalimy docelową wersję systemu na co najmniej 2.3, nasza aplikacja automatycznie otrzyma plik *proguard.cfg*. Poprzez skonfigurowanie funkcji ProGuard w tym pliku możemy zmusić narzędzia ADT do gmatwania kodu w momencie komplilowania wersji rynkowej pliku *.apk*. Jeżeli wolimy tworzyć aplikacje za pomocą narzędzia *ant*, możemy je również skonfigurować w taki sposób, aby została w nim wykorzystana funkcja ProGuard do gmatwania kodu. Aby włączyć tę możliwość, musimy wprowadzić właściwość *proguard.config* w pliku *default.properties*, który umieszczamy w tej samej lokacji co plik *proguard.cfg*. W trakcie przeprowadzania procesu gmatwania funkcja ProGuard wygeneruje plik *mapping.txt* wraz z plikiem *.apk*. Musimy pozostawić ten plik, gdyż jest on niezbędny do odwrócenia procesu gmatwania stosu w aplikacji.

## Przygotowanie pliku .apk do wysłania

Aby przygotować swoją aplikację do wysłania — to znaczy utworzyć w tym celu plik *.apk* — trzeba postąpić zgodnie z poniższym algorytmem (został on szczegółowo omówiony w rozdziale 10.):

1. Jeżeli jeszcze tego nie zrobileś, utwórz certyfikat produktu, za pomocą którego podpiszesz swoją aplikację.
2. Jeżeli aplikacja wykorzystuje mapy, zamień klucz API MAP w pliku *AndroidManifest.xml* na klucz API MAP produktu. Jeżeli tego nie zrobisz, użytkownicy nie będą widzieli map.
3. Eksportuj aplikację poprzez kliknięcie prawym przyciskiem myszy nazwy projektu w oknie *Package explorer* środowiska Eclipse, wybranie opcji *Android Tools/Export Unsigned Application Package* oraz dobranie odpowiedniej nazwy pliku. Warto nadać temu plikowi tymczasową nazwę, ponieważ po uruchomieniu aplikacji *zipalign* w punkcie 5. będzie trzeba wprowadzić nazwę pliku wyjściowego, która będzie stanowić nazwę naszego pliku *.apk*.
4. Uruchom aplikację *jarsigner* wobec naszego nowego pliku *.apk*, aby podpisać go za pomocą utworzonego w punkcie 1. certyfikatu produktu.
5. Uruchom aplikację *zipalign* wobec pliku *.apk*, aby dopasować nieskompresowane dane do granic pamięci, dzięki czemu uzyskasz lepszą wydajność po uruchomieniu aplikacji. To właśnie teraz wprowadza się ostateczną nazwę pliku *.apk* naszej aplikacji.
6. Obecnie w środowisku Eclipse możesz wykorzystać opcję *Export Signed Application Package*, wykorzystującą kreator do przeprowadzenia etapów 3., 4. i 5.

## Wysyłanie aplikacji

Wysyłanie aplikacji jest prostym procesem, wymagającym jednak nieco przygotowań. Przed rozpoczęciem wysyłania musimy ustawić kilka parametrów oraz podjąć pewne decyzje. Niżej podrozdział został poświęcony omówieniu tych przygotowań i decyzji. Gdy już wszystko będzie gotowe, przejdziemy do konsoli programisty i wybierzemy opcję *Upload Application*. Zostaniemy poproszeni o wprowadzenie wielu informacji dotyczących naszej aplikacji, usługa Market przebadzie aplikację oraz dostarczone dane i w końcu nasz program zostanie przygotowany do publikacji w sklepie Android Market.

W poprzednim podrozdziale omówiliśmy proces przygotowania pliku .apk do wysłania. Przyjęciążenie uwagi klientów wymaga z naszej strony szczyty marketingu. Musimy stworzyć dobry opis aplikacji oraz jej przeznaczenia, konieczne też będzie wykonanie zrzutów ekranu, aby użytkownicy wiedzieli, że nie kupują kota w worku.

Jednym z pierwszych elementów, o jakie zostaniemy poproszeni podczas wysyłania aplikacji, są zrzuty ekranu. Najprostszym sposobem ich uzyskania jest zastosowanie narzędzia DDMS. Uruchamiamy środowisko Eclipse, następniełączamy aplikację na emulatorze lub w rzeczywistym urządzeniu i przełączamy perspektywę na widoki DDMS i Device. Wewnątrz widoku Device wybieramy urządzenie, na którym uruchomiliśmy aplikację, i klikamy przycisk *Screen Capture* (symbolizuje go ikona małego obrazu umieszczona w prawym górnym rogu ekranu) lub wybieramy go z menu *View*. Jeżeli pojawi się taka możliwość, wybierzmy 24-bitowy kolor. Android Market przekonwertuje zrzuty ekranu na skompresowane pliki JPEG; początkowa wartość 24 bitów przyniesie lepsze rezultaty niż początkowa wartość 8 bitów. Wybierzmy takie zrzuty, które przy ukazywaniu oryginalności aplikacji prezentują jednocześnie jej funkcje. Musimy wprowadzić co najmniej dwa zrzuty ekranu, maksymalnie zaś możemy zamieścić ich osiem.

Następną kwestią jest wygenerowana ikona aplikacji w wysokiej rozdzielczości. Możemy w tym celu wykorzystać projekt tej samej ikony, która zostanie zastosowana w urządzeniu, ale musi ona posiadać wymiary 512×512 pikseli. Jest to jeden z wymogów usługi Android Market.

Możemy umieścić również grafikę promującą, jej rozmiar będzie jednak mniejszy od zrzutu ekranu. Chociaż taka grafika stanowi jedynie dodatek, warto ją również opublikować. Nigdy nie wiadomo, kiedy zostanie wyświetlona; bez niej nie będziemy pewni, co (jeżeli cokolwiek) pojawi się na jej miejscu. Jednym z miejsc, w którym pojawia się grafika promocyjna, jest górna część ekranu wyświetlającego szczegóły aplikacji w sklepie Android Market.

Kolejnym opcjonalnym rodzajem obrazu jest grafika wymieniająca cechy aplikacji, o rozmiarze 1024×500 pikseli. Grafika ta będzie wykorzystywana w sekcji *Polecane* serwisu Android Market, więc powinniśmy się naprawdę postarać, aby ten obraz wyglądał jak najlepiej.

Ostatnim elementem graficznym związanym z naszą aplikacją jest opcjonalny plik wideo, który możemy zamieścić w serwisie YouTube. Adres do tego pliku możemy umieścić w opisie aplikacji.

Android Market poprosi następnie o informację tekstową dotyczącą aplikacji, która będzie wiadoma dla klientów, włącznie z tytułem, opisem oraz tekstem promującym. Możliwość wprowadzenia tekstu promującego stanie się dostępna jedynie po umieszczeniu grafiki promującej. Możemy opublikować tekst w wielu językach, gdyż nasza aplikacja będzie dostępna na całym świecie. Grafika promująca może być umieszczona w sklepie Android Market wyłącznie jednorazowo, zatem jeśli zrzuty ekranu wyglądają inaczej w różnych ustawieniach regionalnych, powinniśmy rozważyć umieszczenie ich w innym miejscu dostępnym dla klientów, na przykład na stronie domowej. Takie podejście może w przyszłości ulec zmianie.

Jeśli napisaliśmy własną wersję umowy EULA, powinniśmy w opisie zamieścić odnośnik do niej, dzięki czemu użytkownicy zapoznają się z jej treścią przed pobraniem aplikacji. Należy взять pod uwagę, że klienci prawdopodobnie będą chcieli wykorzystać funkcję wyszukiwania aplikacji, zatem najlepiej byłoby umieścić w tekście odpowiednie słowa kluczowe, dzięki czemu znacząco wzrośnie prawdopodobieństwo natrafienia na naszą aplikację przez osoby szukające zapewnianych przez nią funkcji. W końcu warto również umieścić krótki komentarz wraz z adresem e-mail na wypadek pojawienia się problemów z aplikacją. Bez tej prostej zachęty użytkownicy mogą częściej wystawiać negatywne opinie, a taka negatywna opinia naprawdę ogranicza możliwość rozwiązywania problemu w porównaniu do wymiany informacji z użytkownikiem, który natrafił na jakiś błąd.

Jedną z wad omówionego wcześniej mechanizmu wsparcia technicznego jest brak rozróżnienia pomiędzy wersjami aplikacji. Jeżeli wersja 1. oprogramowania otrzymała negatywne opinie, a my wydaliśmy wersję 2. pozbawioną wszystkich usterek poprzedniczki, recenzje dotyczące wersji 1. nie znikają, a klienci nie wiedzą, której wersji dotyczą te opinie. Po wydaniu zaktualizowanej aplikacji jej ocena (liczba gwiazdek) również nie zostaje wyzerowana. Częściowo z tego powodu firma Google zaczęła implementować pole tekstowe *Najnowsze zmiany*, gdzie możemy umieścić listę zmian wprowadzonych w nowej wersji aplikacji. To właśnie tutaj możemy stwierdzić, że jakiś błąd został naprawiony lub wymienić nowe funkcje.

Dostępne jest także oddzielne pole na tekst promujący, pozwalające na wstawienie 80 znaków. Po wyświetleniu naszej aplikacji na samej górze listy w sklepie Android Market tam właśnie są umieszczone grafika promująca oraz tekst promujący. Wypełnienie tych pól informacjami jest naprawdę dobrym pomysłem.

Jednym z obowiązków wydawcy aplikacji jest ujawnienie w opisie tej aplikacji wymaganych przez nią uprawnień. Mamy na myśli te same uprawnienia, które są definiowane za pomocą znaczników `<uses-permission>` w pliku *AndroidManifest.xml*. Kiedy użytkownik pobiera aplikację na urządzenie, system sprawdza plik *AndroidManifest.xml* i przed zakończeniem instalacji pyta użytkownika o wszystkie zamieszczone w nim uprawnienia. Również dobrze możemy umieścić je w opisie aplikacji. W przeciwnym wypadku ryzykujemy otrzymanie negatywnych opinii od użytkowników zaskoczonych faktem, że aplikacja wymaga uprawnień, których nie zamierzali jej przyznać. Nie wspominamy nawet o zwrotach kosztów, które zaniżają ogólną ocenę dewelopera. Podobnie jak w przypadku uprawnień, jeśli aplikacja wymaga określonego typu ekranu, aparatu fotograficznego lub jakiejkolwiek innej funkcji urządzenia, odpowiednie informacje powinny zostać zamieszczone w opisie programu. Najlepszym rozwiązaniem jest nie tylko zamieszczenie wymaganych przez aplikację uprawnień i funkcji, lecz także opis sposobu ich wykorzystania. Powinniśmy z góry odpowiedzieć na pytanie, dlaczego taka aplikacja wymaga obecności funkcji X.

W trakcie wysyłania aplikacji musimy wybrać jej rodzaj i określić kategorię. Ponieważ z upływem czasu wartości te ulegają zmianom, nie będziemy ich wymieniać, wystarczy przejść do ekranu *Upload Application*, żeby ujrzeć dostępne możliwości.

Następnym etapem jest ustalenie ceny aplikacji. Domyslnie aplikacja jest darmowa, a żeby to zmienić, trzeba posiadać skonfigurowane konto handlowe w usłudze Google Checkout. Wybór odpowiedniej ceny dla aplikacji wcale nie jest taki łatwy, chyba że posiadamy wyjątkowo rozwinięty zmysł marketingowca, a nawet wtedy nietrudno o pomyłkę. Zbyt wysokie opłaty mogą zniechęcać użytkowników, poza tym zwiększąją odczuwalne dla wydawcy skutki zwracania kosztów ludziom, którzy uznali aplikację za niewartą swojej ceny. Z kolei zbyt niskie ceny mogą również zniechęcić ludzi, którzy uzajmują, że taka aplikacja jest niskobudżetowa.

Android Market posiada opcję włączenia ochrony przed kopiowaniem po wysłaniu aplikacji. Serwis ten automatycznie wyposaży naszą aplikację w ten mechanizm, powinniśmy jednak pamiętać, że rozwiązanie to nieco bardziej obciąża pamięć urządzenia. Nie jest ono również niezawodne i nie gwarantuje całkowitego zabezpieczenia aplikacji. Ponieważ funkcja ochrony przez kopiowaniem znajduje się na etapie wycofywania, powinniśmy wziąć pod uwagę alternatywne lub dodatkowe sposoby ochrony programu, na przykład opisaną wcześniej usługę licencyjną systemu Android.

Pod koniec 2010 roku firma Google wprowadziła schemat oceniania aplikacji. Jego zadaniem jest określenie przedziału wiekowego grupy docelowej użytkowników. Niestety, połowa grup wiekowych dotyczy nastolatków. Do wyboru mamy oceny: *Dla wszystkich, Niska dojrzałość*.

**Średnia dojrzałość i Wysoka dojrzałość.** Wybór odpowiedniej grupy wiekowej zależy od treści aplikacji oraz od ilości tej treści. Firma Google zdefiniowała zasady związane z położeniem geograficznym oraz umieszczaniem lub publikowaniem lokacji. Najlepiej samemu przejrzeć te zasady, znajdziemy je pod adresem [www.google.com/support/androidmarket/bin/answer.py?hl=en&answer=188189](http://www.google.com/support/androidmarket/bin/answer.py?hl=en&answer=188189).

Jedną z ostatnich decyzji, jakie należy podjąć, jest wybór regionów geograficznych oraz operatorów telefonii komórkowej, dla których nasza aplikacja będzie widoczna. Poprzez wybór opcji *All* aplikacja będzie dostępna na całym świecie. Czasami jednak należy ograniczyć dystrybucję do danego regionu geograficznego lub operatora. W zależności od funkcji oferowanych przez aplikację wymagane jest nierzaz ograniczenie liczby krajów, w których aplikacja będzie dopuszczona do obrotu, ze względu na konieczność przestrzegania prawa eksportowego Stanów Zjednoczonych. Ograniczenie dystrybucji aplikacji do określonych operatorów jest konieczne, gdy wystąpią problemy z kompatybilnością z urządzeniami lub niezgodność z zasadami danego operatora. Aby przejrzeć listę dostępnych operatorów, klikamy w konsoli programisty nazwę wybranego kraju, dzięki czemu zostanie wyświetlony spis dostępnych operatorów w danym państwie. Zaznaczenie opcji *All* spowoduje również, że dana aplikacja będzie dostępna dla wszelkich kolejnych państw i operatorów, którym kiedyś firma Google udostępni serwis Android Market — w tym celu nie będą potrzebne żadne działania programisty.

Chociaż profil programisty zawiera informacje kontaktowe, możemy wprowadzić inne dane podczas wysyłania każdej aplikacji. Usługa Market monituje o wprowadzenie adresu strony WWW, adresu e-mail oraz numeru telefonu, służących jako informacje kontaktowe związane z daną aplikacją. W celu umożliwienia obsługi klientów musimy wypełnić przynajmniej jedno z wymienionych pól, nie jest konieczne jednak wprowadzenie danych do wszystkich trzech elementów. Podawanie tutaj prywatnego adresu e-mail nie jest najlepszym pomysłem, tak samo jak nie chcielibyśmy najprawdopodobniej podawać tutaj prywatnego numeru telefonu. Gdy będziemy zarabiać miliony dolarów na sprzedaży naszej aplikacji, raczej zatrudnimy kogoś odbierającego telefony i odpowiadającego na e-maile od użytkowników. Jeżeli od razu założymy adres e-mail do celów obsługi pomocy technicznej, nie będziemy mieli później problemu z rozdzieleniem poczty osobistej i wiadomości od użytkowników.

Po podjęciu wszystkich omówionych decyzji musimy naszej aplikacji nadać atest przestrzegania polityki treści w usłudze Android Market dla programistów (nie jest zbyt rygorystyczna), a także drugi atest — umożliwiający eksportowanie programu poza granice Stanów Zjednoczonych. Aplikacje udostępniane poprzez serwis Android Market podlegają prawu eksportowemu Stanów Zjednoczonych, ponieważ serwery Google są umieszczone w tym kraju. Dotyczy to nawet aplikacji utworzonych w innym państwie oraz sytuacji, kiedy zarówno programista, jak i jego użytkownicy znajdują się poza USA. Nie zapominajmy, że zawsze możemy dystrybować aplikację innymi kanałami. Gdy już wprowadzimy wszystkie niezbędne informacje i wyślemy zdjęcie, możemy w końcu wcisnąć przycisk *Save*. Nasza aplikacja zostanie wtedy przygotowana do wysłania w świat.

W końcu możemy opublikować naszą aplikację poprzez wciśnięcie przycisku *Publish*. Android Market sprawdzi publikowany program, zwłaszcza pod kątem daty wygaśnięcia certyfikatu aplikacji. Jeżeli cały proces przebiegnie pomyślnie, nasz kod stanie się dostępny do pobrania przez innych użytkowników. Gratulacje!

## Korzystanie ze sklepu Android Market

Z poziomu urządzeń Android Market jest dostępny już od pewnego czasu, natomiast mniej więcej od lutego 2011 roku można również korzystać z niego poprzez internet. Wydawcy nie mają żadnej kontroli nad działaniem sklepu, mogą co najwyżej wstawić ciekawy tekst i zrzuty ekranu do opisu umieszczonej aplikacji. Zatem pod tym względem o komfort użytkownika musi zadbać sama firma Google. Za pomocą urządzenia użytkownik może przeszukiwać bazę danych przy zastosowaniu słowa kluczowego, przeglądać najczęściej pobierane aplikacje (płatne oraz darmowe), zalecane programy lub nowości oraz całe kategorie. Po znalezieniu odpowiedniej aplikacji użytkownik może ją od razu zaznaczyć, co spowoduje wyświetlenie ekranu szczegółów programu, na którym można wybrać opcję jego instalacji lub kupna. Wybór opcji kupna uruchomi usługę Google Checkout, gdzie zostanie przeprowadzona finansowa część transakcji. Pobrań aplikacja pojawi się na urządzeniu użytkownika wśród pozostałych programów.

Interfejs użytkownika w wersji internetowej serwisu Android Market (<http://market.android.com>) wygląda praktycznie tak samo jak w urządzeniach, został jednak dostosowany do rozmiarów monitora. Jedna z zasadniczych różnic polega na konieczności wprowadzenia nazwy użytkownika konta Google, aby móc przeglądać wersję sieciową serwisu Android Market. W ten sposób firma Google łączy działania użytkownika w internetowej wersji serwisu Android Market z działaniami użytkownika na urządzeniu. Oznacza to, że po pierwsze, kiedy użytkownik korzysta z wersji sieciowej serwisu, Android Market ma dostęp do informacji, które aplikacje są zainstalowane w należącym do użytkownika urządzeniu. Po drugie, w trakcie zakupów pobierana aplikacja będzie wysyłana wprost do urządzenia użytkownika, nawet jeśli została zakupiona poprzez stację roboczą.

Witryna Android Market daje możliwość przeglądania pobranych aplikacji w katalogu *Moje zamówienia*. Można tu oglądać wszystkie zarówno zainstalowane aplikacje, jak i zakupione programy, nawet po ich usunięciu (najczęściej zostają usunięte wyłącznie z powodu braku miejsca w urządzeniu). Oznacza to, że użytkownik może usunąć płatną aplikację i ponownie ją zainstalować w innym terminie bez ponoszenia dodatkowych kosztów. Oczywiście, po wybraniu opcji zwrotu kosztów dana aplikacji nie będzie wyświetlana w katalogu *Moje zamówienia*. W folderze tym nie będą również pokazywane darmowe aplikacje po ich wykasowaniu. Lista aplikacji umieszczonych w tym katalogu jest powiązana z kontem Google obsługiwany przez urządzenie. Oznacza to, że możemy zmienić urządzenie bez obaw o utratę zakupionych aplikacji. Pamiętajmy jednak o tym, że jeśli posiadamy kilka tożsamości na serwerach Google, zakupione wcześniej aplikacje możemy pobrać jedynie z konta, na którym za nie zapłaciliśmy. Podczas przeglądania aplikacji w katalogu *Moje zamówienia* wszelkie nowe wersje programów będą zaznaczone oraz gotowe do uaktualnienia.

Android Market filtry aplikacje dostępne dla określonych użytkowników. Proces ten jest przeprowadzany na wiele sposobów. W niektórych krajach dostępne są wyłącznie bezpłatne wersje programów z powodu różnorakich wymogów prawa handlowego, które nie odpowiadają firmie Google w danym państwie. Firma Google bardzo stara się przezwyciężyć te przeszkody, aby płatne aplikacje były dostępne na całym świecie. Do tego czasu użytkownicy w niektórych krajach mogą cieszyć się jedynie darmowymi aplikacjami. Osoby posiadające urządzenia pracujące pod kontrolą starszej wersji systemu Android nie mają dostępu do aplikacji obsługiwanych przez nowsze wersje zestawu Android SDK. Użytkownicy korzystający z urządzeń niespełniających wymagań sprzętowych danej aplikacji (definiowanych w znacznikach `<uses-feature>` pliku *AndroidManifest.xml*) również nie będą widzieć takich programów. Na przykład aplikacje nieobsługujące małych wyświetlaczów nie będą widziane w sklepie Android Market przez

użytkowników posiadających urządzenia z takimi właśnie ekranami. Taka filtracja została wprowadzona głównie po to, aby uchronić użytkowników przed pobraniem aplikacji, która nie będzie działać w ich telefonach.

Jeżeli kupujemy aplikację w innych krajach, na etapie transakcji może nastąpić przewalutowanie, co zazwyczaj oznacza dodatkową opłatę, chyba że sprzedawca ustalił cenę w naszej walucie. Aplikacje są w rzeczywistości kupowane z kraju sprzedawcy za pośrednictwem usługi Google Checkout. Sklep Android Market wyświetla przybliżoną kwotę, lecz w rzeczywistości opłaty mogą być nieco inne w zależności od momentu przeprowadzenia transakcji oraz użytych procesorów płatności. Osoby kupujące mogą zauważać, że ich konto zostaje obciążone symboliczną opłatą (na przykład 1 dolara) w czasie przeprowadzania transakcji. Firma Google upewnia się w ten sposób, że wprowadzone informacje o płatności są poprawne, a wspomniana opłata w rzeczywistości nie zostanie uiszczena.

Istnieje w internecie kilka stron, które stanowią odzwierciedlenie witryny Android Market. Użytkownicy mogą tam wyszukiwać aplikacje, przeglądać kategorie oraz czytać informacje na temat programów bez konieczności posiadania urządzenia. Rozwiążanie to działa na zasadzie filtrowania przez Android Market konfiguracji urządzenia oraz regionu geograficznego, w którym znajduje się użytkownik. Nie ma jednak możliwości pobrania w ten sposób aplikacji na urządzenie. Przykładami takich lustrzanych witryn są [www.androlib.com](http://www.androlib.com), [www.androidzoom.com](http://www.androidzoom.com) oraz [www.cyrket.com](http://www.cyrket.com).

## Alternatywy dla serwisu Android Market

Serwis Android Market nie jest jedynym graczem na rynku. Nie istnieje żaden przymus korzystania z tej usługi. Powinniśmy brać pod uwagę również inne kanały dystrybucji, nie tylko po to, aby udostępniać aplikacje użytkownikom w niektórych krajach, lecz również po to, aby korzystać z innych metod pobierania opłat oraz możliwości zarabiania.

Istnieją sklepy z aplikacjami zupełnie niezależne od witryny Android Market. Przykładami mogą być [www.andappstore.com](http://www.andappstore.com), [slideme.org](http://slideme.org), [www.getjar.com](http://www.getjar.com) i [www.androidgear.com](http://www.androidgear.com). Serwis Amazon również uruchamia własną wersję sklepu Android App Store. Możemy na tych stronach wyszukiwać, przeglądać, czytać informacje o aplikacjach, a także pobierać je zarówno z poziomu telefonu, jak i przeglądarki. Witryny te nie muszą żądać przestrzegania zasad firmy Google, wliczając w to również opłaty transakcyjne oraz formy płatności. Te oddzielne sklepy akceptują takie procesory płatności, jak na przykład PayPal. Zostaje w nich również pominięte ograniczenie dotyczące rejonu geograficznego i konfiguracji sprzętowej. Niektóre ze sklepów oferują instalację klienta Android, inne zaś dokonują jego wstępnej instalacji w urządzeniu. Użytkownicy mogą po prostu otworzyć przeglądarkę WWW w urządzeniu i wyszukać w internecie daną aplikację; po jej zapisaniu na karcie pamięci system będzie posiadał informacje niezbędne do jej zainstalowania. Pobrany plik .apk jest traktowany jak aplikacja systemu Android. Jeżeli użytkownik kliknie w przeglądarce w historii pobranych plików nazwę takiego pliku (nie należy mylić tego mechanizmu z omówionym wcześniej katalogiem *Moje zamówienia*), zostanie zapytany, czy chce zainstalować pobraną aplikację. Taka swoboda oznacza, że programista może ustanawiać własne metody pobierania aplikacji przez użytkowników, nawet z domowej strony WWW, oraz zastosować wybrane przez siebie metody płatności. Nadal jednak trzeba wypełniać zobowiązania podatkowe wobec urzędu skarbowego.

Chociaż takie alternatywne metody dystrybuowania aplikacji nie są ograniczane przez zasady firmy Google, nie oferują również tak wysokiego poziomu ochrony kupca, jak ma to miejsce

w sklepie Android Market. Istnieje możliwość, że użytkownik zakupi z takiego źródła aplikację, która nie będzie działać na jego urządzeniu. Osoba kupująca musi również zapewnić sobie tworzenie kopii zapasowych aplikacji na wypadek jej przypadkowej utraty lub w momencie zmiany urządzenia.

Te inne kanały dystrybucji pozwalają nam zarabiać na sprzedaży każdego egzemplarza aplikacji, podobnie jak ma to miejsce w przypadku usługi Android Market. Możemy także implementować w ich obszarze alternatywne mechanizmy uiszczenia zapłaty. Oczywiście, możemy również wprowadzać reklamy do aplikacji i zarabiać w ten sposób. Nie ma także przeciwwskazań co do umieszczenia tych mechanizmów wewnętrz aplikacji. Na przykład serwis PayPal zawiera odpowiednią bibliotekę dostosowaną do systemu Android (warto zajrzeć na stronę <http://www.x.com>). W ten sposób umożliwiamy użytkownikom zakup dodatków, nowej treści lub płatnych aktualizacji wprost z poziomu aplikacji. W ten sam sposób można uzyskiwać datki. Możemy nawet zaimplementować mobilny sklepik, w którym byłby wykorzystywany mechanizm PayPal.

Pamiętajmy, że firma Google nie zabrania wydawcom jednocześnie sprzedaży aplikacji w wielu różnych sklepach i w usłudze Android Market. Zatem w celu zwiększenia efektywności powinniśmy wziąć pod uwagę wszystkie możliwości.

## Odbośniki

Poniżej prezentujemy odnośniki do materiałów, które mogą pomóc w zrozumieniu koncepcji omówionych w niniejszym rozdziale:

- <http://developer.android.com/guide/topics/manifest/manifest-intro.html> — jest to strona poradnika programisty poświęcona plikowi *AndroidManifest.xml*, zawierająca opisy zastosowania znaczników `<supports-screens>`, `<uses-configuration>` oraz `<uses-feature>`.
- [http://developer.android.com/guide/practices/screens\\_support.html](http://developer.android.com/guide/practices/screens_support.html) — strona poradnika programisty zawierająca informacje o tak zwanej obsłudze wielu ekranów. Znajdziemy tu wiele przydatnych informacji na temat korzystania z różnorodnych rozmiarów i gęstości wyświetlaczy.
- [http://developer.android.com/guide/practices/ui\\_guidelines/icon\\_design.html](http://developer.android.com/guide/practices/ui_guidelines/icon_design.html) — strona poradnika programisty zawierająca wskazówki dotyczące projektowania ikon. Umieszczono tu interesujące informacje na temat tworzenia ikon wpływających na jakość naszej aplikacji.
- <http://android-developers.blogspot.com/2010/09/securing-android-lvl-applications.html> oraz <http://android-developers.blogspot.com/2010/09/proguard-android-and-licensing-server.html> — dwa wpisy dotyczące metod stosowania biblioteki LVL w celu zapobieżenia piractwu.
- <http://developer.android.com/guide/market/billing/index.html> — dokumentacja dotycząca wbudowanego modułu pobierania opłat.

## Podsumowanie

Teraz możemy już podbić cały świat naszymi aplikacjami utworzonymi dla systemu Android! Pokazaliśmy, w jaki sposób należy przygotować siebie oraz swoją aplikację, jak należy ją opublikować oraz umożliwić użytkownikom jej wyszukanie, pobranie i użytkowanie.

# **Koncepcja fragmentów oraz inne pojęcia dotyczące tabletów**

Aż do teraz zajmowaliśmy się mechanizmami wspólnymi dla wszystkich wersji systemu Android. Zdumiewające, że minęły zaledwie dwa lata od zaprezentowania Androida w urządzeniach dostępnych na rynku, a już nastął świt nowej ery urządzeń wykorzystujących ten system — tabletów. Interfejs użytkownika dostępny w wersji 3.0 Androida został od podstaw zaprojektowany na potrzeby tabletów. Na szczęście nie oznacza to wcale, że musimy zignorować całą dotychczas zdobytą wiedzę i rozpocząć cały proces nauki od początku. W rzeczywistości *wszystko*, cze- go się do tej pory dowiedzieliśmy, okaże się przydatne w procesie pisania aplikacji przeznaczonych dla tabletów. Wraz z wersją 3.0 Androida zostały zaprezentowane nowe koncepcje oraz funkcje, których opanowanie jest niezbędne do posługiwania się bardzo dużymi (`xLarge`) ekranami tabletów. Większość aplikacji napisanych dla wcześniejszych wersji systemu będzie działała na tabletach, jednak mogą się pojawić kłopoty z ich optymalizacją. Jest to pierwszy rozdział tej książki, w którym zajmujemy się objaśnieniem nowych pojęć i funkcji.

Jedną z nowych rdzennych klas systemu Android 3.0 jest klasa `Fragment`, zawierająca kilka klas potomnych. W tym rozdziale zapoznamy się z koncepcją fragmentu, jego budową oraz powiązaniami z architekturą aplikacji, a także sposobami jego wykorzystania. Dzięki fragmentom możemy przeprowadzać teraz wiele czynności, które wcześniej były bardzo trudne do zaimplementowania. Interesujący jest również fakt, że fragmenty mogą zostać wykorzystane w aplikacjach dla starszych wersji Androida, ponieważ firma Google wydała zestaw SDK zawierający architekturę fragmentów działającą z tymi systemami. Zatem nawet jeśli nie jesteśmy zainteresowani tworzeniem aplikacji dla tabletów, możemy stwierdzić, że fragmenty ułatwią nam życie nawet w urządzeniach, które nie są wyposażone w ekran o wysokiej rozdzielczości.

Zacznijmy od koncepcji fragmentów.

## Czym jest fragment?

W pierwszym podrozdziale wyjaśnimy, czym są fragmenty i do czego służą. Najpierw jednak Czytelnik powinien przyswoić sobie odpowiednie podstawy, aby zrozumieć, po co w ogóle została zaprojektowana koncepcja fragmentów. Jak już powiedzieliśmy, aplikacje systemu Android wykorzystują aktywności w urządzeniach wyposażonych w niewielkie ekrany do prezentowania danych oraz różnorodnych funkcji użytkownikowi, a każda taka aktywność posiada w miarę proste, wyraźnie zdefiniowane przeznaczenie. Przykładowo za pomocą aktywności możemy wyświetlać listę kontaktów umieszczonych w książce adresowej. Inna aktywność może pozwalać na pisanie wiadomości e-mail. Aplikacja składa się z zestawu tego typu aktywności, zgrupowanych w większe jednostki w celu spełnienia bardziej złożonego zadania, na przykład zarządzania kontem pocztowym poprzez odczytywanie i wysyłanie wiadomości. Jest to dobre rozwiązanie w przypadku urządzeń wyposażonych w niewielkie gabarytowo ekrany, jeżeli jednak mamy do czynienia z bardzo dużym wyświetlaczem (co najmniej 10 cali), do dyspozycji będziemy mieć miejsce pozwalające na wykonywanie większej liczby czynności. Być może przydatna okaże się możliwość przeglądania listy wiadomości w skrzynce wiadomości przychodzących i jednocześnie podglądu zaznaczonego listu w osobnym oknie. Ewentualnie aplikacja może wyświetlać listę kontaktów i w tym samym czasie prezentować szczegółowy widok zaznaczonego kontaktu.

Jako programiści aplikacji dla Androida wiemy, że możemy osiągnąć ten cel poprzez zdefiniowanie jeszcze jednego układu graficznego, przeznaczonego dla bardzo dużych ekranów, zawierającego kontrolki ListView i inne rodzaje widoków. Poprzez „jeszcze jeden układ graficzny” mamy na myśli dodatkowy układ graficzny, występujący równorzędu z analogicznymi układami, zdefiniowanymi dla urządzeń posiadających mniejsze wyświetlacze. Oczywiście, niezbędne okaże się zaprojektowanie oddzielnego układu graficznego dla trybu portretowego i krajobrazowego. W przypadku bardzo dużego ekranu oznacza to dość dużo widoków dla etykiet, pól, obrazów i innych obiektów, które należy rozmieścić i odpowiednio zaprogramować. Nasuwa się myśl, że przydałby się jakiś sposób pogrupowania tych elementów i utworzenia dla nich wspólnej logiki, dzięki czemu dane elementy składowe aplikacji mogłyby być wykorzystywane w aplikacjach dla ekranów o różnych rozmiarach oraz dla różnych urządzeń, minimalizując w ten sposób pracę programisty. Dlatego właśnie została zaprojektowana koncepcja fragmentów.

Fragment można uznać za swego rodzaju podaktywność. Rzeczywiście, semantyka fragmentu bardzo przypomina semantykę aktywności. Zawiera on podobną hierarchię widoków oraz analogiczny cykl życia. Fragmenty mogą nawet reagować w taki sam sposób na wciśnięcia przycisku cofania jak aktywności. Jeżeli Czytelnik zastanawia się, czy w ten sposób można jednocześnie umieścić wiele aktywności na ekranie tabletu, to oznacza, że jest na dobrej drodze. Jednak ponieważ utrzymywanie więcej niż jednej uruchomionej aktywności w danej aplikacji może powodować nieporządek, zadanie to zostało przypisane właśnie fragmentom. Oznacza to, że fragmenty są przechowywane wewnętrz aktywności. Mogą one przebywać jedynie we wnętrzu kontekstu aktywności; fragment nie może bez niej istnieć. Współlegają one z pozostałymi elementami aktywności, co oznacza, że *nie* musimy konwertować całego interfejsu użytkownika, aby móc korzystać z fragmentu. Możemy utworzyć wcześniejszy układ graficzny aktywności i wykorzystać fragment odnoszący się do tylko jednego elementu interfejsu użytkownika.

W porównaniu do aktywności fragmenty zachowują się jednak inaczej, gdy mamy do czynienia z zachowywaniem i odczytywaniem stanu. Struktura fragmentu zawiera kilka funkcji, które pozwalają na o wiele łatwiejsze zapisywanie i wczytywanie jego stanu, niż ma to miejsce w przypadku aktywności.

To, kiedy należy wprowadzić fragmenty, zależy od kilku czynników, które teraz omówimy.

## Kiedy należy stosować fragmenty?

Jednym z głównych powodów stosowania fragmentów jest możliwość wykorzystywania elementów interfejsu użytkownika oraz jego funkcji w różnych urządzeniach oraz dla różnych rozmiarów ekranu. Stwierdzenie to dotyczy zwłaszcza tabletów. Pomyślmy, ile rzeczy naraz może się działać na ekranie tabletu. W tym przypadku mamy do czynienia raczej z komputerem biurkowym niż telefonem, a wiele aplikacji biurowych posiada wielopanelowy interfejs użytkownika. Zgodnie z tym, co powiedzieliśmy wcześniej, możemy w jednym momencie wyświetlić na ekranie listę elementów oraz szczegółowy opis jednego z nich. Łatwo to zobrazować w trybie krajobrazowym, gdzie lista może się znajdować po lewej stronie ekranu, a szczegółowy opis — po prawej. Co się jednak stanie, w przypadku gdy użytkownik obróci urządzenie do trybu portretowego i ekran stanie się węższy i wyższy? Być może będziemy teraz chcieli, aby lista znalazła się w górnej części wyświetlacza, a szczegóły w jego dolnej części. Co jednak zrobić, w przypadku gdy wyświetlacz będzie zbyt mały na jednocześnie wyświetlanie dwóch elementów? Czy najlepszym rozwiązaniem nie byłoby oddzielenie aktywności listy od aktywności szczegółów, lecz w taki sposób, aby mogły współdzielić logikę wykorzystywaną w tej samej aplikacji, ale podczas obsługi dużych ekranów? Mamy nadzieję, że Czytelnik odpowiedział twierdząco. Także w tym przypadku fragmenty okazują się pomocne.

Cofnijmy się do przykładu ze zmianą orientacji ekranu. Wiemy, że w przypadku pisania kodu obsługującego zmiany, które zachodzą w aktywności podczas obrotu urządzenia, prawdziwym utrapieniem okazuje się zapisywanie bieżącego stanu aktywności oraz jego przywracanie po odtworzeniu aktywności w nowym trybie. Czy nie podobałoby się nam, gdyby aktywność składała się z elementów, które byłyby utrzymywane w trakcie zmian trybu orientacji, dzięki czemu uniknęlibyśmy tego całego chaosu związanego z usuwaniem i odtwarzaniem aktywności? Oczywiście, że bardzo by nam się podobało. Od tego są fragmenty.

Wyobraźmy sobie teraz, że użytkownik korzysta z naszej aktywności i przeprowadza jakąś czynność. Założmy, że interfejs użytkownika uległ zmianie w obrębie tej aktywności i użytkownik chce cofnąć się o jeden albo dwa ekranы, a może nawet trzy. W klasycznej aktywności wciśnięcie przycisku cofania uniemożliwi użytkownikowi powrót do danej aktywności. W przypadku fragmentów każde wciśnięcie tego przycisku spowoduje cofnięcie o jeden fragment w ich stosie i użytkownik cały czas będzie mógł korzystać z bieżącej aktywności.

Pomyślmy teraz o interfejsie użytkownika, w którym zmianie ulega ogromny zakres treści; chcielibyśmy, aby ten proces przebiegał w elegancki sposób, jak na dopracowaną aplikację przystało. Także tutaj fragmenty okażą się pomocne.

Skoro już mniej więcej wiemy, czym jest fragment oraz do czego może nam się przydać, zajrzyjmy nieco głębiej w jego strukturę.

## Struktura fragmentu

Jak już wspomnieliśmy, fragment przypomina nieco podaktywność: posiada jasno określone przeznaczenie i niemal zawsze prezentuje interfejs użytkownika. Jednak aktywność stanowi klasę podrzędną w stosunku do kontekstu, natomiast fragment jest rozszerzeniem klasy `Object`, będącej częścią pakietu `android.app`. Fragment *nie* stanowi rozszerzenia aktywności. Jednak, podobnie jak ma to miejsce w przypadku aktywności, klasa `Fragment` (lub jej elementy podrzędne) będzie zawsze rozszerzana w celu przesłonięcia jej zachowania.

Fragment może posiadać hierarchię widoków służących do nawiązywania kontaktu z użytkownikiem. Hierarchia ta nie różni się od innych hierarchii widoków pod tym względem, że może zostać utworzona (rozwinięta) za pomocą specyfikacji układu graficznego w pliku XML lub implementacji w kodzie Java. Jeżeli ma być widziana przez użytkownika, taka hierarchia musi zostać dołączona do hierarchii widoków aktywności nadrzędnej, czym wkrótce zajmiemy się dokładniej. Obiekty tworzące hierarchię widoków fragmentów nie różnią się od widoków wykorzystywanych w innych rejonach Androida. Zatem cała wiedza zdobyta na temat widoków znajduje również zastosowanie w przypadku fragmentów.

Oprócz hierarchii widoków fragment zawiera także pakiet służący jako argumenty inicjalizacyjne. Analogicznie do aktywności, fragment może zostać automatycznie zachowany, a następnie wczytany przez system. W trakcie wczytywania fragmentu zostaje wywołany domyślny konstruktor (na przykład nieposiadający argumentów), następnie odczytany pakiet argumentów wobec nowego fragmentu. Kolejne metody zwrotne fragmentu uzyskują dostęp do tych argumentów, które mogą zostać wykorzystane do przywrócenia poprzedniego stanu. Z tego powodu koniecznie musimy:

- upewnić się, że istnieje domyślny konstruktor klasy fragmentu;
- dodać pakiet argumentów tuż po utworzeniu nowego fragmentu, dzięki czemu następne metody skonfigurują nasz fragment we właściwy sposób, a system w razie konieczności bezbłędnie go wczyta.

Aktywność w danej chwili może posiadać kilka aktywnych fragmentów, a jeżeli jeden fragment został wymieniony na inny, cały proces wymiany zostanie zapisany w stosie drugoplanowym. Stos ten jest zarządzany przez powiązany z aktywnością menedżer fragmentów. To właśnie za jego pomocą definiowane jest zachowanie fragmentów po wciśnięciu przycisku cofania. Menedżer fragmentów zostanie omówiony w dalszej części tego rozdziału. Teraz Czytelnikowi wystarczy wiedzieć, że fragment doskonale „wie”, z którą aktywnością jest powiązany, dzięki czemu może bez problemu nawiązać komunikację z menedżerem fragmentów. Fragment może również uzyskać dostęp do zasobów poprzez aktywność.

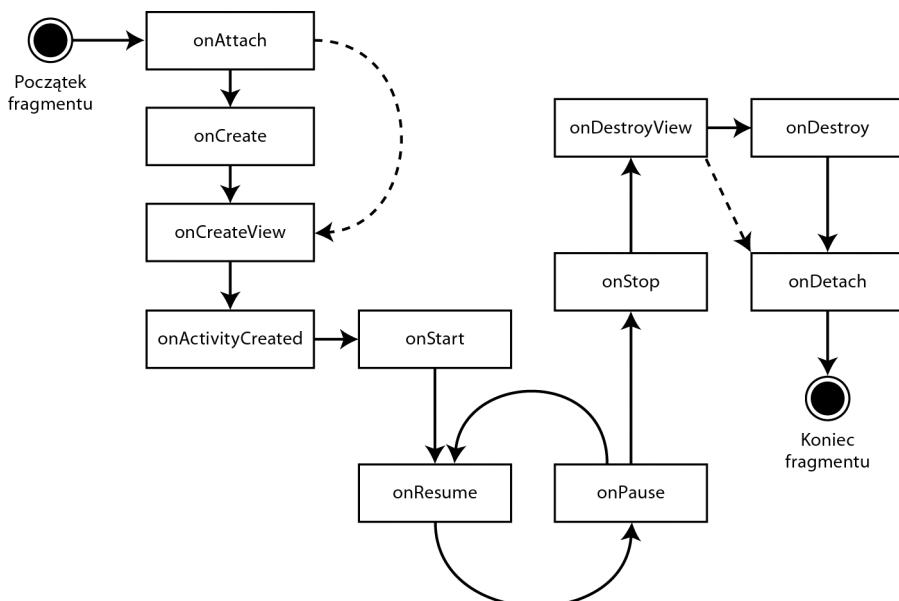
Ponieważ istnieje możliwość zarządzania fragmentem, zawiera on pewne dane identyfikacyjne, w tym znacznik oraz identyfikator. Dane te przydają się podczas wyszukiwania fragmentu, dzięki czemu może być wielokrotnie wykorzystywany.

Również analogicznie do aktywności, podczas odtwarzania pakietu można zapisać jego stan w obiekcie pakietu, który zostaje przekazany do metody zwrotnej `onCreate()`. Taki zachowany pakiet jest również przekazywany do metod `onInflate()`, `onCreateView()` i `onActivity→Created()`. Zwrócmy uwagę, że nie jest to ten sam pakiet, który jest wykorzystywany jako argument inicjalizacji. To właśnie w tym pakiecie będziemy najprawdopodobniej zapisywać stan fragmentu, a nie wartości wykorzystywane do jego inicjalizacji.

## Cykł życia fragmentu

Zanim zaczniemy testować fragmenty w przykładowych aplikacjach, musimy koniecznie zapoznać się z ich cyklem życia. Dlaczego? Cykl życia fragmentu jest bardziej skomplikowany od cyklu życia aktywności i jest bardzo ważne, aby zrozumieć, kiedy można wykonać daną operację na fragmencie. Na rysunku 29.1 przedstawiliśmy cykl życia fragmentu.

Jeżeli porównamy go z rysunkiem 2.15 (na którym pokazaliśmy cykl życia aktywności), dostrzeżemy kilka różnic, najczęściej związanych z oddziaływaniem pomiędzy fragmentem a aktywnością.



**Rysunek 29.1.** Cykl życia fragmentu

Fragment jest bardzo uzależniony od aktywności, do której jest przypisany, i może przejść przez kilka etapów cyklu życia, podczas gdy aktywność w tym samym czasie znajduje się ciągle na jednym etapie.

Na samym początku tworzy się wystąpienie fragmentu. Istnieje on teraz jako obiekt w pamięci. Prawdopodobnie pierwszą czynnością będzie dodanie argumentów inicjalizacji do tego obiektu. Stwierdzenie to jest najbardziej prawdziwe w przypadku odtwarzania wcześniej zachowanego stanu fragmentu. W momencie odtwarzania stanu fragmentu zostaje przywołyany domyślny konstruktor, po czym następuje dołączenie pakietu argumentów inicjalizacji. W przypadku tworzenia nowego wystąpienia fragmentu warto zapoznać się z kodem z listingu 29.1 — zaprezentowaliśmy w nim bardzo przydatny schemat metody fabrykującej obiekty wewnętrz definiacji klasy MyFragment.

#### **Listing 29.1.** Tworzenie wystąpienia fragmentu za pomocą statycznej metody fabrykującej

---

```

public static MyFragment newInstance(int index) {
    MyFragment f = new MyFragment();
    Bundle args = new Bundle();
    args.putInt("index", index);
    f.setArguments(args);
    return f;
}

```

---

Z perspektywy klienta otrzymuje on nowe wystąpienie fragmentu poprzez wywołanie statycznej metody newInstance() zawierającej jeden argument. Otrzymuje z powrotem utworzoną instancję obiektu, natomiast argument inicjalizacji został zdefiniowany w pakiecie argumentów fragmentu. Jeżeli ten fragment zostanie zapisany i odtworzony w późniejszym terminie, system przeprowadzi bardzo podobny proces, angażujący domyślny konstruktor i przyłączający argumenty

inicjalizacji. W tym konkretnym przypadku moglibyśmy zdefiniować sygnaturę metody (lub metod) `newInstance()`, która przyjmowałaby właściwą liczbę i typ argumentów, a następnie poprawnie zbudowałaby pakiet argumentów. Jest to jedyne zadanie metody `newInstance()`. Występujące po niej metody zwrotne prowadzą pozostałą część procesu konfiguracji fragmentu.

## Metoda zwrotna `onInflate()`

Następnym procesem, jaki *mogliby* nastąpić, jest rozwinięcie widoku układu graficznego. Jeżeli nasz fragment jest zdefiniowany przez znacznik `<fragment>` w rozwijanym układzie graficznym (najczęściej ma to miejsce w momencie wywołania metody `setContentView()` wobec głównego układu graficznego aktywności), w naszym fragmencie zostałaby wywołana osobna metoda zwrotna `onInflate()`. Zostają w niej przekazane interfejs `AttributeSet`, atrybuty znacznika `<fragment>` oraz pakiet atrybutów zachowanego stanu. Jeżeli dany fragment został odtworzony oraz została wcześniej zapisany jakiś stan w metodzie `onSaveInstanceState()`, we wspomnianym pakiecie są przechowywane wartości tego stanu. Oczekujemy po metodzie `onInflate()`, że będą odczytywane wartości stanu oraz zostaną one zachowane na później. W rzeczywistości na tym etapie istnienia fragmentu jest jeszcze za wcześnie, żeby móc cokolwiek zrobić z interfejsem użytkownika. Fragment nie został jeszcze nawet powiązany z aktywnością. Ale to będzie właśnie następna czynność, jaką zostanie na nim przeprowadzona.

### Uwaga!

Do rejestru błędów został wprowadzony defekt numer 14796, który wynika z rozbieżności pomiędzy dokumentacją metody `onInflate()` a tym, co faktycznie zachodzi w systemie Honeycomb. Zgodnie z dokumentacją metoda `onInflate()` jest zawsze wywoływana przed metodą `onAttach()`. W rzeczywistości po ponownym uruchomieniu aktywności metoda ta może zostać wywołana po metodzie `onCreateView()`. Jest już wtedy za późno na umieszczenie wartości w pakiecie i wywołanie metody `setArguments()`. Więcej informacji na ten temat znajdziemy pod adresem <http://code.google.com/p/android/issues/detail?id=14796>. Z tego samego powodu metoda zwrotna `onInflate()` nie została umieszczona na schemacie z rysunku 29.1; zbyt trudno przewidzieć, kiedy zostanie wywołana.

## Metoda zwrotna `onAttach()`

Metoda zwrotna `onAttach()` zostaje przywołana po przyłączeniu fragmentu do aktywności. Jeżeli chcemy wykorzystać odniesienie do aktywności, możemy uzyskać do niego dostęp. Taką aktywność możemy wykorzystać przynajmniej do uzyskania informacji o otaczającej aktywności. Możemy również stosować aktywność w postaci kontekstu dla innych czynności. Warto odnotować fakt, że klasa `Fragment` zawiera metodę `getActivity()`, która w razie potrzeby zawsze będzie przekazywała aktywność dołączoną do naszego fragmentu. Pamiętajmy, że przez cały cykl życia pakiet argumentów inicjalizacji będzie dostępny za pomocą metody `getArguments()` fragmentu. Jednak po przyłączeniu fragmentu do aktywności nie będziemy mogli już wywołać tej metody, zatem możemy dodawać argumenty inicjalizacji jedynie na samym początku.

## Metoda zwrotna `onCreate()`

Następnym etapem jest metoda `onCreate()`. Przypomina ona analogiczną metodę aktywności, różnica polega na tym, że nie powinniśmy w jej wnętrzu umieszczać kodu zależnego od obecności hierarchii widoków aktywności. Chociaż nasz fragment może być już powiązany z aktywnością, nie zostaliśmy jeszcze powiadomieni o zakończeniu działania metody `onCreate()` aktywności. Dopiero zbliżamy się do tego momentu. Jeżeli posiadamy pakiet argumentów

zachowanego stanu, zostanie on przekazany tej metodzie. Metoda ta może jako pierwsza utworzyć wątek drugoplanowy, pobierający dane wymagane przez fragment. Kod fragmentu jest przetwarzany w wątku interfejsu użytkownika i nie chcemy, aby były w nim przeprowadzane również operacje wejścia-wyjścia lub sieciowe. W rzeczywistości logicznym rozwiązaniem jest przygotowanie danych za pomocą wątku pobocznego. To właśnie w nim powinny występować wszystkie wywołania blokujące. Musimy później połączyć się w jakiś sposób z tymi danymi, istnieją na to jednak odpowiednie sposoby.

**Uwaga!**

Jednym ze sposobów wczytania danych w wątku pobocznym jest zastosowanie klasy `Loader`. Zabrakło nam miejsca na jej opis w książce, zapraszamy jednak do przejrzenia naszej oficjalnej strony WWW w celu zapoznania się z niezbędnymi informacjami.

### Metoda zwrotna `onCreateView()`

Kolejną metodę zwrotną stanowi `onCreateView()`. Spodziewamy się po niej, że przekaże naszemu fragmentowi hierarchię widoków. Wśród przekazywanych argumentów znajdziemy tutaj klasę `LayoutInflater` (służącą do rozwijania układu graficznego danego fragmentu), klasę nadziedną `ViewGroup` (na listingu 29.2 noszącą nazwę `container`) oraz pakiet zachowanego stanu (jeżeli istnieje). Jest bardzo istotne, aby zwrócić tutaj uwagę, że nie powinniśmy dodać hierarchii widoku do przekazywanego widoku potomnego `ViewGroup`. Powiązanie to nastąpi automatycznie później. Mamy do dyspozycji klasę nadziedną, dzięki czemu możemy ją wykorzystać wraz z metodą `inflate()` klasy `LayoutInflater`, chociaż w razie konieczności możemy samodzielnie sprawdzić tę klasę. Najprawdopodobniej jednak, jeżeli przyłączymy w tej metodzie zwrotnej hierarchię widoków fragmentu do klasy nadziednej, pojawią się wyjątki. Na listingu 29.2 prezentujemy przykład operacji, jaką można wykonać w tej metodzie.

#### **Listing 29.2.** Utworzenie hierarchii widoków fragmentu w metodzie `onCreateView()`

```
@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
    View v = inflater.inflate(R.layout.details, container, false);
    TextView text1 = (TextView) v.findViewById(R.id.text1);
    text1.setText(myDataSet[ getPosition() ] );
    return v;
}
```

Widzimy tutaj, w jaki sposób możemy uzyskać dostęp do układu graficznego wykorzystywanego wyłącznie przez ten fragment i rozwinąć go do widoku, który zostanie następnie przekazany obiektowi wywołującemu. Takie rozwiązanie posiada kilka zalet. Możemy zawsze utworzyć hierarchię widoków za pomocą kodu, jednak poprzez rozwinięcie układu graficznego z pliku XML wykorzystujemy możliwości technologii wyszukiwania zasobów. W zależności od konfiguracji urządzenia lub, dokładniej mówiąc, od aktualnie używanego urządzenia zostanie wybrany najwłaściwszy plik układu graficznego. Możemy wtedy uzyskać dostęp do określonego widoku przechowywanego w układzie graficznym; w naszym przypadku — pola `text1` klasy `TextView`. Jeszcze raz powtarzamy: nie należy dodać w tej metodzie zwrotnej widoku danego fragmentu do klasy nadziednej. Na listingu 29.2 widzimy, że wykorzystujemy pojemnik w wywołaniu metody `inflate()`, przekazujemy jednak również wartość `false` w parametrze `attachToRoot`.

## Metoda zwrotna onActivityCreated()

Zbliżamy się do momentu, w którym użytkownik będzie mógł oddziaływać na fragment. Następną metodą cyklu życia fragmentu jest `onActivityCreated()`. Zostaje ona wywołana po zakończeniu działania metody zwrotnej `onCreate()`. Mamy teraz pewność, że hierarchia widoków aktywności, w tym również hierarchia widoków fragmentu (jeśli została wcześniej przekazana), jest gotowa i dostępna. To właśnie na tym etapie wprowadzamy ostatnie poprawki w interfejsie, zanim oddamy go w ręce użytkownika. Jest to szczególnie istotne w przypadku odtwarzania aktywności i jej fragmentów z zachowanego stanu. Teraz właśnie do aktywności zostają również dołączone pozostałe fragmenty.

## Metoda zwrotna onStart()

Kolejna metoda cyklu życia fragmentu to `onStart()`. Na tym etapie fragment jest już widoczny dla użytkownika. Nie rozpoczęliśmy jednak jeszcze interakcji z użytkownikiem. Metoda ta jest powiązana z metodą zwrotną `onStart()` aktywności. W ten sposób możemy wstawić część operacji uprzednio umieszczanych w metodzie `onStart()` aktywności do metody `onStart()` fragmentu, gdyż to właśnie w nim znajdują się składniki interfejsu użytkownika.

## Metoda zwrotna onResume()

Ostatnią metodą zwrotną występującą przed rozpoczęciem interakcji użytkownika z fragmentem jest `onResume()`. Metoda ta jest powiązana z metodą `onResume()` aktywności. Po jej powrocie użytkownik może już korzystać z fragmentu. Jeśli na przykład nasz fragment zawiera podgląd widoku aparatu fotograficznego, uruchomimy go prawdopodobnie w metodzie `onResume()` tego fragmentu.

Dotarliśmy w końcu do punktu, w którym nasza aplikacja — ku radości użytkownika — szczerze rozpoczęła działanie. Po pewnym czasie użytkownik zechce zakończyć pracę z programem albo poprzez wcisnięcie przycisku cofania, albo przycisku ekranu startowego, albo ewentualnie poprzez uruchomienie innej aplikacji. Podobnie jak w przypadku aktywności, następuje tu sekwencja zdarzeń postępujących w kierunku przeciwnym do omówionej powyżej.

## Metoda zwrotna onPause()

Pierwszą metodą zwrotną anulującą fragment jest `onPause()`. Jest ona powiązana z metodą `onPause()` aktywności; podobnie jak ma to miejsce w przypadku aktywności, jeżeli fragment zawiera odtwarzacz multimedii lub jakiś inny współdzielony obiekt, możemy wstrzymać, zatrzymać lub odebrać wyniki jego działania właśnie poprzez tę metodę. Mamy tu do czynienia z takimi samymi zasadami „dobrego wychowania” co w przypadku aktywności: muzyka nie powinna być odtwarzana, kiedy użytkownik rozmawia przez telefon. Istnieje możliwość przejścia fragmentu od metody `onPause()` z powrotem do metody `onResume()`.

## Metoda zwrotna onStop()

Kolejna metoda zwrotna anulująca fragment to `onStop()`. Jest ona powiązana z metodą `onStop()` aktywności i pełni podobną funkcję. Zatrzymany fragment może przejść wprost do metody `onStart()`, skąd wiedzie bezpośrednia droga do metody `onResume()`.

## Metoda zwrotna onDestroyView()

Jeżeli nasz fragment ma zostać zamknięty lub jeśli trzeba zapisać jego stan, następną metodą zwrotną występującą na ścieżce jego anulowania jest `onDestroyView()`. Zostanie ona wywołana po odłączeniu hierarchii widoków z danego fragmentu, utworzonej w metodzie `onCreateView()`.

## Metoda zwrotna onDestroy()

Następnie mamy do czynienia z metodą `onDestroy()`. Zostaje ona wywołana, w przypadku gdy fragment przestaje być potrzebny. Zwróćmy uwagę, że jest on nadal dołączony do aktywności i ciągle można go „odnaleźć”, na niewiele się już jednak przyda.

## Metoda zwrotna onDetach()

Ostatnią metodą zwrotną cyklu życia fragmentu jest `onDetach()`. Po jej wywołaniu fragment przestaje być powiązany z aktywnością, nie posiada już hierarchii widoków i wszystkie związane z nim zasoby zostają zwolnione.

## Stosowanie metody setRetainInstance()

Być może Czytelnik zwrócił uwagę na połączenia oznaczone przerywaną linią na rysunku 29.1. Jedną z bardziej interesujących cech fragmentu jest możliwość zadeklarowania, że nie chcemy całkowicie pozbywać się go w przypadku odtwarzania aktywności, dzięki czemu nasze fragmenty mogą się pojawiać później. Taki fragment pojawia się więc wraz z wywołaniem metody `setRetainInstance()`, która przyjmuje wartość logiczną. Możemy sobie obrazowo przedstawić, że wartość ta może znaczyć: „Tak, ten fragment ma istnieć w trakcie odtwarzania aktywności” lub „Nie, ten fragment zostanie usunięty i utworzymy od początku nowy fragment”. Najlepszym miejscem na wywołanie tej metody jest wewnętrzne metody `onCreate()` fragmentu.

Jeśli parametr przyjmie wartość `true`, oznacza to, że chcemy przechować obiekt fragmentu w pamięci i nie mamy zamiaru tworzyć nowego obiektu od podstaw. Jeżeli jednak aktywność zostaje usunięta i odtworzona, musimy odłączyć fragment od starego obiektu i podłączyć go do nowej aktywności. Wniosek z tego taki, że jeżeli wartość przechowywanego wystąpienia wynosi `true`, w rzeczywistości nie będziemy całkowicie usuwać instancji fragmentu, zatem nie będziemy musieli tworzyć nowego fragmentu. Jednak wszystkie pozostałe metody zwrotne zostaną wywołane. Połączenia zaznaczone przerywaną linią oznaczają, że możemy podczas wychodzenia pominąć metodę `onDestroy()` oraz — na etapie ponownego podłączania fragmentu do aktywności — metodę `onCreate()`. Ponieważ aktywność jest naprawdopodobniej odtwarzana z powodu zmian konfiguracyjnych, metody zwrotne fragmentu powinny się opierać na założeniu, że takie zmiany nastąpiły i należy podjąć odpowiednie działania. Takim działaniem może być na przykład rozwinięcie układu graficznego tworzącego nową hierarchię widoków w metodzie `onCreateView()`. Do tego może posłużyć przykładowo kod zamieszczony na listingu 29.2. Jeżeli postanowimy skorzystać z funkcji przechowywania instancji, być może powinniśmy pominąć wstawienie części logiki inicjalizacji w metodzie `onCreate()`, ponieważ nie będzie ona wywoływana zawsze w taki sam sposób jak pozostałe metody zwrotne.

## Przykładowa aplikacja ukazująca cykl życia fragmentu

Nic nie pozwala bardziej docenić omawianej koncepcji, jak zademonstrowanie jej na działającym przykładzie. Utworzmy aplikację prezentującą w działaniu wszystkie omówione powyżej metody zwrotne. Jeden fragment będzie zawierał listę dzieł Szekspira; po kliknięciu któregoś

tytułu zostanie wyświetlony obszerny cytat z tej sztuki w osobnym fragmencie. Aplikacja ta działa na tablecie zarówno w trybie portretowym, jak i krajobrazowym. Skonfigurujemy następnie ten przykład w taki sposób, aby działał na mniejszym ekranie, dzięki czemu Czytelnik nauczy się rozdzielać fragmenty tekstowe na aktywności. Rozpoczniemy od układu graficznego aktywności w trybie krajobrazowym, zaprezentowanym na listingu 29.3 i zilustrowanym na rysunku 29.2.

**Uwaga!**

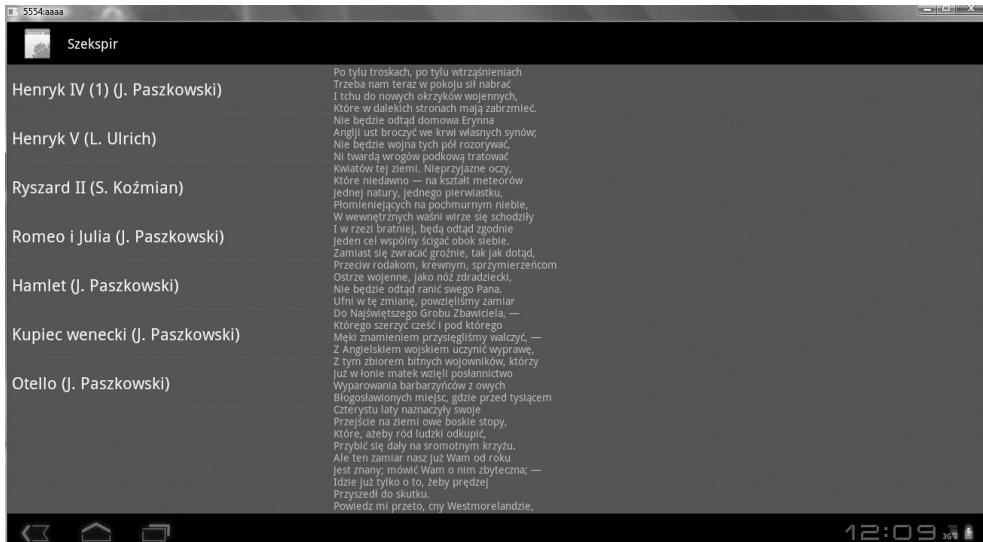
Na końcu rozdziału zamieściliśmy adres URL, pod którym znajdziemy listę projektów utworzonych na potrzeby rozdziału. Możemy je zimportować bezpośrednio do środowiska Eclipse.

**Listing 29.3.** Układ graficzny aktywności w trybie krajobrazowym

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout-land/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment class="com.androidbook.fragments.bard.TitlesFragment"
        android:id="@+id/titles" android:layout_weight="1"
        android:layout_width="0px"
        android:layout_height="match_parent" />

    <FrameLayout
        android:id="@+id/details" android:layout_weight="2"
        android:layout_width="0px"
        android:layout_height="match_parent" />
</LinearLayout>
```



**Rysunek 29.2.** Interfejs użytkownika przykładowej aplikacji ukazującej zastosowanie fragmentów

Powyższy układ graficzny nie różni się zbytnio od układów graficznych prezentowanych w poprzednich rozdziałach: jest on ułożony poziomo, z dwoma elementami umieszczonymi z lewej i z prawej strony. Mamy jednak do czynienia z nowym znacznikiem — <fragment>, w którym znajdziemy nowy atrybut noszący nazwę class. Pamiętajmy, że fragment nie jest widokiem, zatem układ graficzny fragmentu nieco się różni od innych układów graficznych. Należy również pamiętać, że znacznik ten stanowi jedynie wypełniacz w tym układzie graficznym.

Pozostałe atrybuty fragmentu wyglądają podobnie jak atrybuty widoku i pełnią analogiczne funkcje. Atrybut class znacznika fragmentu definiuje rozszerzoną klasę dla listy tytułów. Oznacza to, że musimy rozszerzyć klasę Fragment w celu zaimplementowania logiki, a w znaczniku <fragment> musi się znaleźć nazwa tej rozszerzonej klasy. Fragment posiada własną hierarchię widoków, która zostanie w późniejszym okresie samoistnie utworzona. Następnym znacznikiem jest FrameLayout, a nie kolejny znacznik <fragment>. Dlaczego? Wyjaśnimy to dokładniej w dalszej części rozdziału, na razie jednak Czytelnik powinien wiedzieć, że będziemy wprowadzać pewne modyfikacje tekstu i wymieniać fragmenty między sobą. Znacznik FrameLayout służy jako pojemnik widoku przechowujący bieżący fragment tekstu. W przypadku fragmentu przechowującego tytuły istnieje jeden (i tylko jeden) fragment, o który trzeba zadbać; nie ma żadnego zamieniania miejscami i żadnych innych modyfikacji. W przypadku obszaru wyświetlającego szekspirowski poemat mamy do czynienia z kilkoma fragmentami.

Kod klasy MainActivity został umieszczony na listingu 29.4.

#### **Listing 29.4.** Kod źródłowy klasy MainActivity

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.app.Fragment;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.content.Intent;
import android.content.res.Configuration;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;

public class MainActivity extends Activity {
    public static final String TAG = "Szekspir";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        Log.v(TAG, "w metodzie onCreate aktywnosci MainActivity");
        super.onCreate(savedInstanceState);
        FragmentManager.enableDebugLogging(true);
        setContentView(R.layout.main);
    }

    @Override
    public void onAttachFragment(Fragment fragment) {
        Log.v(TAG, "w metodzie onAttachFragment aktywnosci MainActivity. Id fragmentu = "
                + fragment.getId());
        super.onAttachFragment(fragment);
    }

    @Override
```

```
public void onStart() {
    Log.v(TAG, "w metodzie onStart aktywnosci MainActivity");
    super.onStart();
}

@Override
public void onResume() {
    Log.v(TAG, "w metodzie onResume aktywnosci MainActivity");
    super.onResume();
}

@Override
public void onPause() {
    Log.v(TAG, "w metodzie onPause aktywnosci MainActivity");
    super.onPause();
}

@Override
public void onStop() {
    Log.v(TAG, "w metodzie onStop aktywnosci MainActivity");
    super.onStop();
}

@Override
public void onSaveInstanceState(Bundle outState) {
    Log.v(MainActivity.TAG, "w metodzie onSaveInstanceState aktywnosci
    ↪MainActivity");
    super.onSaveInstanceState(outState);
}

@Override
public void onDestroy() {
    Log.v(TAG, "w metodzie onDestroy aktywnosci MainActivity");
    super.onDestroy();
}

public boolean isMultiPane() {
    return getResources().getConfiguration().orientation
        == Configuration.ORIENTATION_LANDSCAPE;
}

/**
 * Funkcja pomocnicza ukazująca szczegóły zaznaczonego elementu albo poprzez
 * wyświetlanie fragmentu znajdującego się w bieżącym interfejsie użytkownika, albo
 * poprzez rozpoczęcie nowej aktywności, w której będą one wyświetlane.
 */
public void showDetails(int index) {
    Log.v(TAG, "w metodzie showDetails(" + index + ") aktywnosci MainActivity");

    if (isMultiPane()) {
        // Sprawdza, który fragment jest wyświetlany, w razie potrzeby zamienia go.
        DetailsFragment details = (DetailsFragment)
            getFragmentManager().findFragmentById(R.id.details);
        if (details == null || details.getShownIndex() != index) {
            // Tworzy nowy fragment w celu ukazania szczegółów zaznaczenia.
        }
    }
}
```

```
details = DetailsFragment.newInstance(index);

// Przeprowadza operację zastąpienia istniejącego
// fragmentu fragmentem umieszczonym wewnątrz ramki.
Log.v(TAG, "tuz przed uruchomieniem operacji FragmentTransaction...");  
FragmentTransaction ft  
        = getSupportFragmentManager().beginTransaction();  
ft.setTransition(  
        FragmentTransaction.TRANSIT_FRAGMENT_FADE);  
//ft.addToBackStack("details");  
ft.replace(R.id.details, details);  
ft.commit();  
}  
  
} else {  
    // W przeciwnym wypadku musimy uruchomić nową aktywność  
    // wyświetlającą fragment zawierający okno dialogowe z zaznaczonym tekstem.  
    Intent intent = new Intent();  
    intent.setClass(this, DetailsActivity.class);  
    intent.putExtra("indeks", index);  
    startActivity(intent);  
}  
}
```

Mamy do czynienia z bardzo prostą aktywnością. Jedynym powodem umieszczenia wszystkich metod zwrotnych w kodzie źródłowym jest możliwość wyświetlania komunikatów w dzienniku. Gdyby nie ten fakt, jedną wymaganą metodą byłaby `onCreate()`, z kolei metodami pomocniczymi są `isMultiPane()` oraz `showDetails()`. Trudno byłoby wprowadzić prostszą metodę od ukazanej tu `onCreate()`. Służy ona tutaj wyłącznie do uruchomienia trybu debugowania menedżera fragmentów oraz ustanowienia widoku treści w postaci układu graficznego z listingu 29.3. Aby zdefiniować tryb wielopanelowy (na przykład w celu umieszczania obok siebie wielu fragmentów), wykorzystujemy jedynie położenie wyświetlacza. Jeżeli wyświetlacz jest ułożony w trybie krajobrazowym, możemy korzystać z wielu paneli równocześnie; w trybie portretowym jest to niemożliwe. Na koniec zauważmy, że pomocnicza metoda `showDetails()` służy do zdefiniowania sposobu wyświetlania szczegółów zaznaczonego tekstu. Indeks definiuje w tym przypadku pozycję na liście tytułów. Jeżeli aplikacja pracuje w trybie wielopanelowym, wykorzystamy fragment do wyświetlania tekstu. Fragment ten został nazwany `DetailsFragment` i do jego utworzenia (wraz z indeksem) stosujemy metodę fabrykującą. Kod klasy `DetailsFragment` został zaprezentowany na listingu 29.5. Później jeszcze powrócimy do metody `showDetails()`.

**Listing 29.5.** Kod źródłowy klasy DetailsFragment

```
import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;
import android.util.AttributeSet;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
```

```
public class DetailsFragment extends Fragment {

    private int mIndex = 0;

    public static DetailsFragment newInstance(int index) {
        Log.v(MainActivity.TAG, "w metodzie newInstance(" +
            index + ") klasy DetailsFragment");

        DetailsFragment df = new DetailsFragment();

        // Dostarcza indeks w postaci argumentu.
        Bundle args = new Bundle();
        args.putInt("indeks", index);
        df.setArguments(args);
        return df;
    }

    public static DetailsFragment newInstance(Bundle bundle) {
        int index = bundle.getInt("indeks", 0);
        return newInstance(index);
    }

    @Override
    public void onInflate(AttributeSet attrs, Bundle savedInstanceState)
    {
        Log.v(MainActivity.TAG,
            "w metodzie onInflate klasy DetailsFragment. Interfejs AttributeSet
            zawiera:");
        for(int i=0; i<attrs.getAttributeCount(); i++)
            Log.v(MainActivity.TAG, " " + attrs.getAttributeName(i) +
                " = " + attrs.getAttributeValue(i));
        super.onInflate(attrs, savedInstanceState);
    }

    @Override
    public void onAttach(Activity myActivity) {
        Log.v(MainActivity.TAG,
            "w metodzie onAttach klasy DetailsFragment; aktywnoscia jest: " +
            myActivity);
        super.onAttach(myActivity);
    }

    @Override
    public void onCreate(Bundle myBundle) {
        Log.v(MainActivity.TAG,
            "w metodzie onCreate klasy DetailsFragment. Pakiet zawiera:");
        if(myBundle != null) {
            for(String key : myBundle.keySet()) {
                Log.v(MainActivity.TAG, " " + key);
            }
        }
        else {
            Log.v(MainActivity.TAG, " Obiekt myBundle jest pusty");
        }
        super.onCreate(myBundle);

        mIndex = getArguments().getInt("indeks", 0);
    }
}
```

```
}

public int getShownIndex() {
    return mIndex;
}

@Override
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
    Log.v(MainActivity.TAG,
        "w metodzie onCreateView klasy DetailsFragment. pojemnik = " +
        container);

    // Nie wiążemy tego fragmentu z niczym za pomocą obiektu pompującego.
    // Android zajmuje się za nas przyłączaniem fragmentów.
    // Pojemnik jest jedynie przepuszczany, więc wiemy dzięki niemu,
    // dokąd trafi hierarchia widoków.
    View v = inflater.inflate(R.layout.details, container, false);
    TextView text1 = (TextView) v.findViewById(R.id.text1);
    text1.setText(Shakespeare.DIALOGUE[ mIndex ] );
    return v;
}

@Override
public void onActivityCreated(Bundle savedInstanceState) {
    Log.v(MainActivity.TAG,
        "w metodzie onActivityCreated klasy DetailsFragment. Klasa savedInstanceState
        zawiera:");
    if(savedInstanceState != null) {
        for(String key : savedInstanceState.keySet()) {
            Log.v(MainActivity.TAG, " " + key);
        }
    }
    else {
        Log.v(MainActivity.TAG, " Klasa savedInstanceState jest pusta");
    }
    super.onActivityCreated(savedInstanceState);
}

@Override
public void onStart() {
    Log.v(MainActivity.TAG, "w metodzie onStart klasy DetailsFragment");
    super.onStart();
}

@Override
public void onResume() {
    Log.v(MainActivity.TAG, "w metodzie onResume klasy DetailsFragment");
    super.onResume();
}

@Override
public void onPause() {
    Log.v(MainActivity.TAG, "w metodzie onPause klasy DetailsFragment");
    super.onPause();
}
```

```
@Override
public void onSaveInstanceState(Bundle outState) {
    Log.v(MainActivity.TAG,
        "w metodzie onSaveInstanceState klasy DetailsFragment");
    super.onSaveInstanceState(outState);
}

@Override
public void onStop() {
    Log.v(MainActivity.TAG, "w metodzie onStop klasy DetailsFragment");
    super.onStop();
}

@Override
public void onDestroyView() {
    Log.v(MainActivity.TAG,
        "w metodzie onDestroyView klasy DetailsFragment, widok = " +
        getView());
    super.onDestroyView();
}

@Override
public void onDestroy() {
    Log.v(MainActivity.TAG, "w metodzie onDestroy klasy DetailsFragment");
    super.onDestroy();
}

@Override
public void onDetach() {
    Log.v(MainActivity.TAG, "w metodzie onDetach klasy DetailsFragment");
    super.onDetach();
}
}
```

---

Klasa `DetailsFragment` jest w istocie równie nieskomplikowana. Jedynym powodem dużych rozmiarów kodu jest wprowadzenie instrukcji służących do wyświetlania informacji w dzienniku. Gdyby nie były nam one potrzebne, w powyższym fragmencie pozostawilibyśmy jedynie metody `newInstance()`, `getShowIndex()`, `onCreate()` oraz `onCreateView()`. Teraz Czytelnik już wie, w jaki sposób utworzono wystąpienie tego fragmentu. Ważna jest informacja, że wystąpienie tego fragmentu jest tworzone w kodzie, ponieważ układ graficzny definiuje pojemnik `ViewGroup` (dokładniej — obiekt `FrameLayout`), do którego trafi fragment przechowujący szczegółowe informacje. Ponieważ w przeciwieństwie do fragmentu zawierającego tytuły omawiany fragment nie jest zdefiniowany w pliku układu graficznego aktywności, musimy tworzyć wystąpienia fragmentów za pomocą kodu.

Aby utworzyć nowy fragment przechowujący szczegółowe informacje, stosujemy metodę `newInstance()`. Jak już wcześniej stwierdziliśmy, ta metoda fabrykująca przywołuje domyślny konstruktor, a następnie ustanawia pakiet argumentów za pomocą wartości indeksu. Po uruchomieniu metody `newInstance()` fragment przechowujący szczegóły może odczytać wartości indeksu w dowolnej metodzie zwrotnej poprzez odniesienie do pakietu argumentów za pomocą metody `getArguments()`. Dla naszej wygody możemy zapisać w metodzie `onCreate()` wartość indeksu pochodzącą z pakietu argumentów, dokładniej zaś w polu członkowskim klasy `DetailsFragment`.

Możemy się zastanawiać, dlaczego po prostu nie wprowadziliśmy wartości `mIndex` w metodzie `newInstance()`. Wynika to z faktu, że system poza naszym wzrokiem odtworzy fragment za pomocą domyślnego konstruktora. Następnie zostanie wykorzystany pakiet argumentów do przywrócenia poprzedniego stanu. Android nie wykorzysta metody `newInstance()`, więc jedynym pewnym sposobem ustanowienia wartości zmiennej `mIndex` jest odczytanie jej z pakietu argumentów i wprowadzenie jej w metodzie `onCreate()`. Metoda złożona `getShowIndex()` odczytuje wartości tego indeksu. Do opisania pozostała nam już tylko metoda `onCreateView()`, której zrozumienie również nie stanowi wielkiego wyzwania.

Zadaniem metody `onCreateView()` jest przekazywanie hierarchii widoków do fragmentu. Pamiętajmy, że na podstawie konfiguracji chcemy uzyskać wszystkie możliwe rodzaje układów graficznych dotyczące tego fragmentu, zatem najczęściej spotykana czynnością jest spożytkowanie pliku układu graficznego tego fragmentu. W naszej przykładowej aplikacji takim plikiem jest `details.xml`, który definiujemy za pomocą zasobu `R.layout.details`. Zawartość pliku `details.xml` została umieszczona na listingu 29.6.

**Listing 29.6.** Plik układu graficznego `details.xml` zdefiniowany dla fragmentu przechowującego szczegółowe informacje

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/details.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ScrollView android:id="@+id/scroller"
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView android:id="@+id/text1"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </ScrollView>
</LinearLayout>
```

---

W przypadku naszej przykładowej aplikacji możemy stosować ten sam układ graficzny dla trybu krajobrazowego i portretowego podczas wyświetlania szczegółowych informacji. Jest to układ przeznaczony nie dla aktywności, lecz wyłącznie do wyświetlania tekstu fragmentu. Ponieważ może być on uznany za domyślny układ graficzny, możemy umieścić go w katalogu `res/layout`, gdzie zostanie znaleziony i zastosowany, nawet jeśli wyświetlacz znajduje się w trybie krajobrazowym. Podczas wyszukiwania pliku układu graficznego służącego do wyświetlania szczegółów system sprawdza najpierw katalogi ściśle powiązane z konfiguracją urządzenia, powróci jednak do katalogu `res/layout`, jeśli nigdzie indziej nie znajdzie pliku `details.xml`. Oczywiście, jeżeli chcemy zaprojektować inny układ graficzny fragmentu dla trybu krajobrazowego, możemy zdefiniować osobny plik `details.xml` i umieścić go w katalogu `/res/layout-land`. Nic nie stoi na przeszkodzie, żeby eksperymentować z różnymi plikami `details.xml`.

W momencie wywołania metody `onCreate()` fragmentu przechowującego szczegóły system wybierze i rozwinię układ graficzny z odpowiedniego pliku `details.xml`, do którego wstawi tekst z klasy `Shakespeare`. Nie zamieścimy tu całego kodu klasy `Shakespeare`, lecz jedynie jego część (listing 29.7), aby ułatwić Czytelnikowi zrozumienie jego działania. Pełny kod źródłowy znajdziemy w gotowym projekcie, do którego adres został umieszczony w podrozdziale „Odbońniki” na końcu rozdziału.

**Listing 29.7.** Kod źródłowy klasy Shakespeare

```
public class Shakespeare {  
    public static String TITLES[] = {  
        "Henryk IV (1) (J. Paszkowski)",  
        "Henryk V (L. Ulrich)",  
        "Ryszard II (S. Koźmian)",  
        "Romeo i Julia (J. Paszkowski)",  
        "Hamlet (J. Paszkowski)",  
        "Kupiec wenecki (J. Paszkowski)",  
        "Otello (J. Paszkowski)"  
    };  
    public static String DIALOGUE[] = {  
        "Po tylu troskach, po tylu wtrąśnieniach,1\n...  
...i tak dalej..."
```

---

Zatem obecnie nasza hierarchia widoków we fragmencie zawierającym szczegółowe informacje przechowuje tekst z wybranej sztuki. Fragment ten jest już przygotowany do wyświetlenia. Możemy teraz wrócić do metody `showDetails()`, aby zająć się omówieniem klasy `FragmentTransaction`.

## Klasy FragmentTransaction i drugoplanowy stos fragmentów

Kod w metodzie `showDetails()`, który służy do wstawiania nowego fragmentu przechowującego szczegółowe informacje (ukazany ponownie na listingu 29.8), wygląda dość prosto, wykonuje jednak mnóstwo zadań. Warto poświęcić trochę czasu na wyjaśnienie, co tu się dzieje i dlaczego tak jest. Jeżeli nasza aktywność działa w trybie wielopanelowym, chcemy zaprezentować fragment przechowujący szczegóły obok fragmentu zawierającego listę. Być może nasza aplikacja wyświetla już ten pierwszy fragment, co oznacza, że stał się widoczny dla użytkownika. W każdym przypadku identyfikator zasobu `R.id.details` jest przeznaczony dla pojemnika `FrameLayout` aktywności, co jest widoczne na listingu 29.3. Jeżeli fragment przechowujący szczegóły znajduje się wewnątrz układu graficznego, z powodu braku własnego identyfikatora otrzyma właśnie wspomniany identyfikator. Aby więc dowiedzieć się, czy w układzie graficznym znajduje się jakiś fragment, możemy wysłać zapytanie do metody `findFragmentById()` menedżera fragmentów. Zostanie przekazana albo wartość `null`, jeśli układ graficzny jest pusty, albo informacje na temat bieżącego fragmentu. Możemy wtedy zadecydować, że powinniśmy umieścić nowy fragment w układzie graficznym, ponieważ układ graficzny może być pusty lub może być w nim umieszczony układ graficzny reprezentujący szczegóły nieodpowiedniego tytułu. Po podjęciu decyzji o utworzeniu i wykorzystaniu nowego fragmentu wywołujemy w tym celu metodę fabrykującą. Możemy teraz wstawić nowy fragment, który zostanie zaprezentowany użytkownikowi.

**Listing 29.8.** Przykład transakcji fragmentu

```
public void showDetails(int index) {  
    Log.v(TAG, "w metodzie showDetails(" + index + ") aktywnosci MainActivity");  
  
    if (isMultiPane()) {
```

---

<sup>1</sup> Przekład J. Paszkowskiego — przyp. tłum.

```

// Sprawdza, który fragment jest wyświetlany, podmienia go w razie potrzeby.
DetailsFragment details = (DetailsFragment)
    getFragmentManager().findFragmentById(R.id.details);
if (details == null || details.getShownIndex() != index) {
    // Tworzy nowy fragment, służący do wyświetlania szczegółów wybranego elementu.
    details = DetailsFragment.newInstance(index);

    // Przeprowadza transakcję i zamienia dowolny
    // fragment na fragment umieszczony w ramce.
    Log.v(TAG, "tuz przed uruchomieniem operacji FragmentTransaction...\"");
    FragmentTransaction ft
        = getFragmentManager().beginTransaction();
    ft.setTransition(
        FragmentTransaction.TRANSIT_FRAGMENT_FADE);
    //ft.addToBackStack("details");
    ft.replace(R.id.details, details);
    ft.commit();
}
// Reszta została pominięta w celu zaoszczędzenia miejsca.
}

```

Kluczową koncepcją, którą trzeba zrozumieć, jest to, że fragment musi się znaleźć w pojemniku widoków, zwany również grupą widoków. Wynika to częściowo z faktu, że fragment sam w sobie nie jest widokiem. Klasa ViewGroup zawiera takie elementy, jak układy graficzne i ich pochodne klasy. To właśnie dlatego wybraliśmy klasę FrameLayout do pliku *main.xml* stanowiącego układ graficzny aktywności. Nasz fragment przechowujący szczegóły zostanie umieszczony w pojemniku FrameLayout. Gdybyśmy zamiast tego zdefiniowali jeszcze jeden węzeł <fragment> w pliku układu graficznego aktywności, nie moglibyśmy przeprowadzić wymaganej operacji zamiany. To właśnie za pomocą klasy przeprowadzamy zamianę fragmentów. W czasie transakcji zamieniamy miejscami dowolny fragment umieszczony w ramce z nowym fragmentem przechowującym szczegóły. Moglibyśmy rozwiązać to w inny sposób, mianowicie lokalizując identyfikator zasobu — kontrolki TextView, która przechowuje tekst szczegółów, i wprowadzając w niej nowy tekst dla nowej wybranej sztuki Szekspira. Istnieje jednak jeszcze jeden fakt dotyczący fragmentów, przemawiający za korzystaniem z klasy FragmentTransaction.

Jak wiemy, aktywności są poukładane na stosie i w trakcie zagłębiania się w aplikację okazuje się, że często na stosie znajduje się kilka jednocześnie uruchomionych aktywności. Po wcisnięciu przycisku cofania aktywność znajdująca się na wierzchu stosu jest z niego usuwana, a jej miejsce zajmuje następna w kolejce aktywność, która zostaje wznowiona. Proces ten jest przeprowadzany wzdłuż całego stosu aż do poziomu ekranu startowego.

Stanowiło to znakomite rozwiązanie w przypadku prostych aktywności, teraz jednak omawiamy takie, które zawierają po kilka jednocześnie działających fragmentów. Ponadto, skoro możemy coraz bardziej zagłębiać się w aplikację bez konieczności opuszczania aktywności znajdującej się na wierzchu stosu, trzeba było rozwinąć koncepcję korzystania z przycisku cofania w taki sposób, aby uwzględnić także fragmenty. W istocie fragmenty wymagają zastosowania tej koncepcji nawet bardziej niż proste aktywności. Gdy mamy do czynienia z kilkoma fragmentami oddziałyującymi ze sobą równocześnie wewnętrz aktywności i następuje jednocośne przejście do nowej treści, które dotyczy wszystkich tych fragmentów, to wcisnięcie przycisku cofania powinno sprawić, że fragmenty te razem zostaną cofnięte o jeden etap. Aby zapewnić, że to cofnięcie obejmie wszystkie fragmenty w ramach danej aktywności, utworzono klasę FragmentTransaction — zapewnia ona koordynację tego procesu.

Należy pamiętać, że drugoplanowy stos fragmentów nie jest wymagany we wnętrzu aktywności. Możemy zaprogramować działanie przycisku cofania w taki sposób, żeby obejmował on wyłącznie aktywność, a nie fragmenty. Jeżeli nie zdefiniujemy drugoplanowego stosu fragmentów, wcisnięcie przycisku cofania spowoduje usunięcie bieżącej aktywności ze stosu i powrót do wcześniejszej aktywności. Jeśli zaś Czytelnik postanowi wykorzystać możliwości oferowane przez stos fragmentów, powinien na listingu 29.8 usunąć znaki komentarza z wiersza `ft.addToBackStack("details")`. W tym konkretnym przypadku zamieściliśmy w kodzie parametr znacznika w postaci ciągu znaków `details`. Znacznik ten powinien stanowić ciąg znaków symbolizujący stan fragmentów w momencie przeprowadzania transakcji. Możemy w kodzie modyfikować stos drugoplanowy za pomocą wartości znacznika, co umożliwia usunięcie pewnych wpisów czy też uniknięcie innych. Powinniśmy nadawać przemyślane nazwy znacznikom transakcji, aby móc je później łatwiej znajdować.

## Przejścia i animacje zachodzące podczas transakcji fragmentu

Jedną z wyjątkowo eleganckich cech transakcji fragmentu jest możliwość zilustrowania zmiany starego fragmentu na nowy za pomocą przejść i animacji. Nie mamy tu do czynienia z animacjami omawianymi w rozdziałach 16. i 20. Animacje przedstawione w tym rozdziale są o wiele prostsze i nie wymagają zaawansowanej wiedzy o grafice. Warto wykorzystać jedno z przejść pozwalających na dodanie efektów specjalnych podczas zmiany starego fragmentu na nowy. W ten sposób nasza aplikacja stanie się bardziej elegancka, a zmiany fragmentów nabiorą płynności. Jedną z pozwalających na to metod jest widoczna na listingu 29.8 `setTransition()`. Mamy jednak do dyspozycji również kilka innych przejść. W naszym przykładzie skorzystaliśmy z efektu zanikania, możemy jednak wprowadzić metodę `setCustomAnimations()` do opisania innych efektów specjalnych, na przykład wsuwania się jednego fragmentu z prawej strony ekranu, a drugiego — z lewej. Niestandardowe animacje wykorzystują nowe definicje obiektów animacji, a nie stare. Stare pliki animacji zawierają takie znaczniki, jak `<translate>`, podczas gdy w nowych stosowane są znaczniki typu `<objectAnimator>`. Stare pliki animacji znajdują się w katalogu `/data/res/anim` w odpowiednim miejscu platformy Android SDK (na przykład `platforms/android-11` dla systemu Honeycomb). Znajdziemy tu również nowe pliki umieszczone w katalogu `/data/res/animator`. Kod przejścia może wyglądać następująco:

```
ft.setCustomAnimations(android.R.animator.fade_in, android.R.animator.fade_out);
```

Powoduje on, że stary fragment zanika, podczas gdy nowy stopniowo się pojawia. Pierwszy parametr odnosi się do nowego fragmentu, a drugi do fragmentu usuwanego. Warto przejrzeć katalog `animator`, aby zapoznać się z innymi domyślnymi animacjami. Jeżeli chcemy utworzyć własną animację, w dalszej części rozdziału znajdziemy sekcję poświęconą animatorowi obiektów. Bardzo ważna jest również informacja, że wywołanie przejścia musi nastąpić przed wywołaniem metody `replace()`, w przeciwnym wypadku zostanie ono zignorowane.

Używanie animatora obiektów do programowania efektów specjalnych widocznych podczas zmiany fragmentów może być zabawne. W klasie `FragmentTransaction` znajdziemy jeszcze dwie metody, z którymi powinniśmy się zapoznać: `hide()` i `show()`. Parametrem w przypadku obydwu tych metod jest fragment. Zadania, jakie metody te realizują, wynikają z ich nazw. W przypadku fragmentu powiązanego z kontenerem widoków powodują one jego ukrywanie lub wyświetlanie w interfejsie użytkownika. W ten sposób fragment nie zostaje usunięty z menedżera fragmentów, musi być jednak związany z kontenerem widoków, aby metody miały wpływ na jego widoczność. Jeżeli fragment nie zawiera hierarchii widoków lub hierarchia ta nie jest powiązana z hierarchią wyświetlanych widoków, metody te okażą się bezużyteczne.

Po zdefiniowaniu efektów specjalnych występujących w czasie transakcji fragmentu musimy określić również główną czynność. W naszym przypadku zamieniamy fragment znajdujący się w ramce na nowy fragment przechowujący szczegóły. Posłużymy się tutaj metodą `replace()`. Stanowi ona ekwiwalent wywołania metody `remove()` wobec dowolnych fragmentów znajdujących się w ramce, a następnie metody `add()` wobec nowego fragmentu zawierającego szczegóły, co oznacza, że w razie potrzeby możemy po prostu wywoływać zamiast niej metody `remove()` i `add()`.

Ostatnim działaniem, jakie musimy podjąć podczas transakcji fragmentu, jest jej zatwierdzenie. Metoda `commit()` nie powoduje natychmiastowego wykonania czynności, lecz raczej ustalenie jej harmonogramu na czas, gdy wątek interfejsu użytkownika zostanie przygotowany.

Teraz już Czytelnik powinien rozumieć, dlaczego zmiana treści w pojedynczym fragmencie jest taka kłopotliwa. Zmieniamy tu nie tylko tekst; możemy w trakcie przejścia wstawić specjalne efekty graficzne. Istnieje również możliwość zapisania szczegółów przejścia w transakcji fragmentu, dzięki czemu proces ten może zostać później odwrócony. Ostatnie zdanie może wydawać się nieco niezrozumiałe, dlatego Czytelnikowi należy się wyjaśnienie.

Nie mamy tu do czynienia z transakcją w dosłownym tego słowa znaczeniu. Gdy usuwamy transakcje fragmentów ze stosu drugoplanowego, nie cofamy zmian w danych, które mogły zostać wprowadzone. Jeżeli zmiany zostały wprowadzone w obrębie aktywności, na przykład w trakcie tworzenia transakcji fragmentów w stosie drugoplanowym, wcisnięcie przycisku cofania nie sprawi, że zmienione wartości danych zostaną przywrócone do stanu początkowego. Cofamy się jedynie poprzez widoki interfejsu użytkownika, podobnie jak miało do miejsca w przypadku aktywności, teraz jednak dotyczy to fragmentów. Ponieważ fragmenty są zapisywane i odczytywane w taki, a nie inny sposób, wewnętrzny stan fragmentu odczytanego z atrybutów zachowanego stanu będzie zależeć od wartości zachowanych we fragmencie oraz od sposobu ich odczytania. Zatem fragmenty mogą wyglądać tak jak wcześniej, nie można jednak będzie tego powiedzieć o aktywności, chyba że w trakcie odczytywania stanu fragmentów będziemy odczytywać również stan aktywności.

W naszym przykładzie pracujemy tylko z jednym pojemnikiem widoków i wprowadzamy tylko jeden fragment przechowujący szczegóły. W przypadku bardziej złożonych interfejsów użytkownika możemy manipulować pozostałymi fragmentami za pomocą transakcji. W rzeczywistości zajmujemy się tylko rozpoczęciem transakcji, co oznacza, że zamieniamy stary fragment przechowujący szczegóły w ramce na nowy fragment, określamy animację przejścia oraz zatwierdzamy przeprowadzenie tego procesu. Oznaczyliśmy jako komentarz część kodu, w której transakcja jest dodawana do stosu drugoplanowego, można jednak usunąć z tego fragmentu znaki komentarza i tym samym dołączyć ją do stosu.

## Klasa FragmentManager

Klasa `FragmentManager` jest składnikiem obsługującym fragmenty przechowywane w aktywności. Zaliczają się do nich również fragmenty przechowywane w stosie drugoplanowym oraz niepołączone fragmenty. Wyjaśnimy to. Fragmenty powinny być tworzone wyłącznie w kontekście aktywności. Dzieje się to albo poprzez rozwinięcie układu graficznego aktywności, albo poprzez bezpośrednie utworzenie wystąpienia obiektu, co zostało ukazane na listingu 29.1. W tym drugim przypadku fragment zostaje zazwyczaj dołączony do aktywności za pomocą transakcji, natomiast w każdym wypadku uzyskujemy dostęp i zarządzamy fragmentami poprzez klasę `FragmentManager`.

Metodę `getFragmentManager()` wykorzystujemy wobec aktywności lub wobec przyłączonego fragmentu, aby uruchomić menedżer fragmentów. Z listingu 29.8 wiemy, że to właśnie z poziomu menedżera fragmentów uzyskujemy dostęp do transakcji. Poza tym za pomocą menedżera możemy odczytać identyfikator fragmentu, jego znacznik lub kombinację pakiet atrybutów – klucz, by w ten sposób znaleźć dany fragment.

W tym celu mamy do dyspozycji metody pobierające `findFragmentById()`, `findFragmentByTag()` oraz `getFragment()`. Ta ostatnia może zostać wykorzystana razem z metodą `putFragment()`, która również pobiera pakiet atrybutów, klucz oraz wstawiany fragment. Będziemy mieli najprawdopodobniej do czynienia z pakietem `savedState`, a metoda `putFragment()` będzie wstawiona w metodzie zwrotnej `onSaveInstanceState()`, dzięki czemu zostanie zachowany stan bieżącej aktywności (lub innego fragmentu). Metoda `getFragment()` może prawdopodobnie zostać wywołana w metodzie `onCreate()`, aby miała związek z metodą `putFragment()`, chociaż — jak już zostało wcześniej omówione — w przypadku fragmentu pakiet atrybutów jest również dostępny dla pozostałych metod zwrotnych.

Oczywiście, nie możemy stosować metody `getFragmentManager()` wobec fragmentów, które nie zostały podłączone do aktywności. Prawdziwe jest jednak również stwierdzenie, że możemy dołączyć fragment do aktywności w taki sposób, że nie będzie jeszcze widoczny dla użytkownika. Jeżeli zdecydujemy się na to rozwiązanie, naprawdę powinniśmy dołączyć znacznik zawierający określony ciąg znaków do fragmentu, dzięki czemu nie będziemy mieli później problemów z jego wyszukiwaniem. Prawdopodobnie wykorzystamy w tym celu następującą metodę klasy `FragmentTransaction`:

```
public FragmentTransaction add (Fragment fragment, String tag)
```

W rzeczywistości możemy otrzymać fragment, który nie eksponuje hierarchii widoków. Może się to okazać przydatne, jeśli zechcemy zawrzeć określona logikę w taki sposób, aby móc ją dołączyć do aktywności, lecz jednocześnie pozostawić jej pewną dozę autonomii, odgradzającą ją od cyklu życia aktywności i pozostałych fragmentów. Gdy aktywność przechodzi przez cykl odtwarzania wynikający ze zmiany konfiguracji urządzenia, taki fragment niebędący częścią interfejsu użytkownika może pozostawać w dużej części nietknity, podczas gdy sama aktywność zostaje usunięta i na jej miejsce wkracza nowa. Takie rozwiązanie jest interesującą opcją dla metody `setRetainInstance()`.

Menedżer fragmentów obsługuje również stos drugoplanowy. Podczas transakcji fragmentów system umieszcza fragmenty na tym stosie, podczas gdy menedżer fragmentów może je stamtąd usuwać. Zazwyczaj dokonujemy tego przy użyciu identyfikatora lub znacznika fragmentu, równie dobrze możemy jednak w tym celu wprowadzić pozycję w stosie lub po prostu usunąć fragment znajdujący się na wierzchu.

Na koniec należy stwierdzić, że menedżer fragmentów zawiera metody ułatwiające proces debugowania, w tym takie jak umożliwiające umieszczenie komunikatów w oknie *LogCat* (metoda `enableDebugLogging()`) lub umieszczenie bieżącego stanu menedżera fragmentów w strumieniu (metoda `dump()`). Zwróćmy uwagę, że na listingu 29.4 włączyliśmy tryb debugowania menedżera fragmentów w metodzie `onCreate()`.

## Ostrzeżenie dotyczące stosowania odniesień do fragmentów

Nadszedł czas, aby powrócić do dyskusji na temat cyklu życia fragmentu, argumentów i pakietów z zachowanymi stanami. System może zachować jeden z fragmentów w wielu różnych sytuacjach. Oznacza to, że w momencie, gdy trzeba będzie odczytać dany fragment, może go nie

być w pamięci. Z tego właśnie powodu ostrzegamy przed uznaniem zmiennego odniesienia do fragmentu za obiekt, który zachowuje ważność przez długi czas. Jeżeli fragmenty są wymieniane w pojemniku widoków za pomocą transakcji, każde odniesienie do poprzedniego fragmentu będzie teraz wskazywać fragment, który prawdopodobnie znajduje się w stosie drugoplanowym. Ewentualnie fragment może zostać odłączony od hierarchii widoków aktywności w trakcie zmiany konfiguracji urządzenia, na przykład w momencie zmiany pozycji urządzenia. Bądźmy ostrożni.

Jeżeli Czytelnik zamierza przechowywać odniesienie do fragmentu, powinien wiedzieć, kiedy może ono zostać zachowane. W przypadku konieczności jego znalezienia należy wykorzystać jedną z metod pobierania, zawartą w menedżerze fragmentów. Jeżeli chcemy koniecznie zatrzymać odniesienie do fragmentu, na przykład podczas odtwarzania aktywności w trakcie zmiany konfiguracji, możemy zastosować metodę `putFragment()` wraz z odpowiednim pakietem atrybutów. W przypadku zarówno aktywności, jak i fragmentów takim pakietem jest `savedState` wykorzystywany w metodzie `onSaveInstanceState()` oraz ponownie pojawiający się w metodzie `onCreate()` (lub innych wcześnie występujących metodach zwrotnych cyklu życia fragmentu). Prawdopodobnie Czytelnik nigdy nie będzie bezpośrednio przechowywać odniesienia do fragmentu w pakiecie argumentów; jeżeli ktoś poczuje jednak taką pokusę, powinien to bardzo dokładnie przemyśleć.

Innym mechanizmem pozwalającym na uzyskanie dostępu do określonego fragmentu jest wysłanie zapytania zawierającego jego identyfikator lub znacznik. Uprzednio omówione metody pobierające pozwalają na odczytanie w ten sposób fragmentów z menedżera, co oznacza, że posiadamy możliwość zachowania takiego znacznika lub identyfikatora, dzięki czemu możemy za ich pomocą uzyskać dostęp do danego fragmentu, co jest alternatywą metod `putFragment()` i `getFragment()`.

## Klasa ListFragment i węzeł <fragment>

Aby nasza aplikacja zyskała pełną funkcjonalność, musimy zająć się jeszcze kilkoma zagadnieniami. Pierwszym z nich jest klasa `TitlesFragment`. To właśnie ona zostaje utworzona za pomocą pliku `layout.xml` naszej aktywności. Znacznik `<fragment>` gra rolę wypełniacza, w którym zostanie umieszczony omawiany fragment. Nie ma tu zdefiniowanej hierarchii widoków tego fragmentu. Kod klasy `TitlesFragment` został umieszczony na listingu 29.9. Klasa ta służy do wyświetlania listy tytułów.

**Listing 29.9.** Kod klasy `TitlesFragment`

---

```
import android.app.Activity;
import android.app.ListFragment;
import android.os.Bundle;
import android.util.AttributeSet;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.ListView;

public class TitlesFragment extends ListFragment {
    private MainActivity myActivity = null;
    int mCurCheckPosition = 0;
```

```
@Override
public void onInflate(AttributeSet attrs, Bundle savedInstanceState) {
    Log.v(MainActivity.TAG,
        "w metodzie onInflate klasy TitlesFragment. Obiekt AttributeSet zawiera:");
    for(int i=0; i<attrs.getAttributeCount(); i++) {
        Log.v(MainActivity.TAG, " " + attrs.getAttributeName(i) +
            " = " + ("id".equals(attrs.getAttributeName(i))?
                Integer.toHexString(attrs.getAttributeIntValue(i, -1)):
                attrs.getAttributeValue(i)));
    }
    super.onInflate(attrs, savedInstanceState);
}

@Override
public void onAttach(Activity myActivity) {
    Log.v(MainActivity.TAG,
        "w metodzie onAttach klasy TitlesFragment; aktywnosc: " + myActivity);
    super.onAttach(myActivity);
    this.myActivity = (MainActivity)myActivity;
}

@Override
public void onCreate(Bundle myBundle) {
    Log.v(MainActivity.TAG,
        "w metodzie onCreate klasy TitlesFragment. Pakiet zawiera:");
    if(myBundle != null) {
        for(String key : myBundle.keySet()) {
            Log.v(MainActivity.TAG, " " + key);
        }
    }
    else {
        Log.v(MainActivity.TAG, " Obiekt myBundle jest pusty");
    }
    super.onCreate(myBundle);
}

@Override
public View onCreateView(LayoutInflater myInflater,
    ViewGroup container, Bundle myBundle) {
    Log.v(MainActivity.TAG,
        "w metodzie onCreateView klasy TitlesFragment. Pojemnikiem jest: "
        + container);
    return super.onCreateView(myInflater, container, myBundle);
}

@Override
public void onActivityCreated(Bundle savedInstanceState) {
    Log.v(MainActivity.TAG,
        "w metodzie onActivityCreated klasy TitlesFragment. Ob. savedInstanceState zawiera:");
    if(savedInstanceState != null) {
        for(String key : savedInstanceState.keySet()) {
            Log.v(MainActivity.TAG, " " + key);
        }
    }
    else {
        Log.v(MainActivity.TAG, " Obiekt savedInstanceState jest pusty");
    }
}
```

```
super.onActivityResult(savedInstanceState);

// Zapełnia listę tytułami pochodzącymi ze statycznej tabeli.
setListAdapter(new ArrayAdapter<String>(getActivity(),
    android.R.layout.simple_list_item_1,
    Shakespeare.TITLES));

if (savedInstanceState != null) {
    // Odczytuje ostatni stan sprawdzanej pozycji.
    mCurCheckPosition = savedInstanceState.getInt("curChoice", 0);
}

// Uzyskuje dostęp do widoku ListView klasy ListFragment i aktualizuje go.
ListView lv = getListView();
lv.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
lv.setSelection(mCurCheckPosition);

// Aktywność zostaje utworzona, fragmenty stają się dostępne.
// Fragment przechowujący szczegóły zostaje zapełniony.
myActivity.showDetails(mCurCheckPosition);
}

@Override
public void onStart() {
    Log.v(MainActivity.TAG, "w metodzie onStart klasy TitlesFragment");
    super.onStart();
}

@Override
public void onResume() {
    Log.v(MainActivity.TAG, "w metodzie onResume klasy TitlesFragment");
    super.onResume();
}

@Override
public void onPause() {
    Log.v(MainActivity.TAG, "w metodzie onPause klasy TitlesFragment");
    super.onPause();
}

@Override
public void onSaveInstanceState(Bundle outState) {
    Log.v(MainActivity.TAG, "w metodzie onSaveInstanceState klasy TitlesFragment");
    super.onSaveInstanceState(outState);
    outState.putInt("curChoice", mCurCheckPosition);
}

@Override
public void onListItemClick(ListView l, View v, int pos, long id) {
    Log.v(MainActivity.TAG,
        "w metodzie onListItemClick klasy TitlesFragment. pozycja = "
        + pos);
    myActivity.showDetails(pos);
    mCurCheckPosition = pos;
}
```

```
@Override
public void onStop() {
    Log.v(MainActivity.TAG, "w metodzie onStop klasy TitlesFragment");
    super.onStop();
}

@Override
public void onDestoryView() {
    Log.v(MainActivity.TAG, "w metodzie onDestoryView klasy TitlesFragment");
    super.onDestoryView();
}

@Override
public void onDestory() {
    Log.v(MainActivity.TAG, "w metodzie onDestory klasy TitlesFragment");
    super.onDestory();
}

@Override
public void onDetach() {
    Log.v(MainActivity.TAG, "w metodzie onDetach klasy TitlesFragment");
    super.onDetach();
    myActivity = null;
}
}
```

---

Podobnie jak wcześniej większość zamieszczonego tu kodu jest zbędna z punktu widzenia działania aplikacji i służy jedynie do przechowania instrukcji wyświetlających informacje w dzienniku, dzięki czemu będzie wiadomo, kiedy fragment przechodzi do określonego etapu cyklu życia. W przeciwieństwie do klasy DetailsFragment, w tym fragmencie metoda onCreateView() nie posiada specjalnego przeznaczenia. Wynika to z faktu, że rozszerzamy klasę ListFragment, która już zawiera widok ListView. Domyślne ustawienia metody onCreateView() w klasie ListView powodują przekazanie widoku kontrolki ListView. Właściwe operacje są przeprowadzane dopiero na etapie wywołania metody onActivityCreated(). Do tego czasu możemy być pewni, że zostanie utworzona hierarchia widoków aktywności wraz z hierarchią aktywności fragmentu. Identyfikatorem zasobu dla kontrolki ListView jest android.R.id.list1. Żeby jednak uzyskać do niej odniesienie, należy wywołać metodę getListView() wewnętrz metody onActivityCreated(). Ponieważ jednak klasa ListFragment nie jest tym samym co kontrolka ListView, nie podłączamy adaptera bezpośrednio do widoku ListView. Musimy zamiast tego użyć metody setListAdapter() klasy ListFragment. Ponieważ została skonfigurowana hierarchia widoków aktywności, możemy bezpiecznie powrócić do aktywności, aby wywołać metodę showDetails().

Na tym etapie cyklu życia aktywności dodaliśmy adapter do widoku listy, odczytaliśmy bieżącą pozycję (jeżeli powróciliśmy z etapu przywracania, spowodowanego na przykład zmianą położenia urządzenia) oraz zaprogramowaliśmy aktywność (w metodzie showDetails()), aby wprowadziła tekst związany z odpowiednim tytułem sztuki szekspirowskiej.

Klasa TitlesFragment również posiada obiekt nasłuchujący listy, zatem gdy użytkownik zaznaczy inny tytuł, system wywoła metodę zwrotną onListItemClick() i zmieni tekst na odpowiadający danej sztuce, znowu za pomocą metody showDetails().

Kolejnym elementem różniącym ten fragment od wcześniej omawianego fragmentu przechowującego szczegóły jest zapisywanie stanu w pakiecie (wartości wskazującej bieżącą pozycję na liście) oraz odczytywanie jej w metodzie `onCreate()` podczas zamykania i odtwarzania fragmentu. W przeciwnieństwie do fragmentu przechowującego szczegóły, który jest wymieniany w kontrolce `FrameLayout` układu graficznego aktywności, mamy tu do czynienia z tylko jednym fragmentem przechowującym tytuły. Jeśli więc następuje zmiana konfiguracji i nasz fragment przechodzi przez operację zapisywania i odczytywania, chcemy zapamiętać bieżącą pozycję. W przypadku fragmentów przechowujących szczegóły odtwarzamy je bez konieczności zapamiętywania poprzedniego stanu.

## Wywoływanie odrębnej aktywności w razie potrzeby

Istnieje część kodu, o której jeszcze nie wspominaliśmy — chodzi o fragment metody `showDetails()`. Kod ten przydaje się wtedy, gdy urządzenie znajdujące się w trybie portretowym wyświetla fragment przechowujący szczegóły, który wymiarami nie odpowiada fragmentowi przechowującemu tytuły. Potraktujmy to jako problem, chociaż w przypadku tabletów nie musimy się tym martwić. Ponieważ jednak technika fragmentów staje się dostępna również w starszych wersjach Androida, będziemy mogli korzystać z nich zarówno w telefonach, jak i w tabletach. Oznacza to, że całkiem często będziemy się spotykać z tym, że parametry ekranu uniemożliwią dogodne wyświetlenie fragmentu, który w normalnej sytuacji byłby umieszczony obok innych fragmentów. W takiej sytuacji musimy uruchomić oddzielną aktywność, służącą do wyświetlania tego fragmentu. W naszym przykładzie postanowiliśmy zaimplementować w ten sposób aktywność przechowującą szczegóły; jej kod znajdziemy na listingu 29.10.

**Listing 29.10.** Wyświetlanie nowej aktywności, jeśli określony fragment nie mieści się na ekranie

```
// Jest to plik DetailsActivity.java
import android.app.Activity;
import android.content.res.Configuration;
import android.os.Bundle;
import android.util.Log;

public class DetailsActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        Log.v(MainActivity.TAG, "w metodzie onCreate klasy DetailsActivity");
        super.onCreate(savedInstanceState);

        if (getResources().getConfiguration().orientation
            == Configuration.ORIENTATION_LANDSCAPE) {
            // Jeżeli ekran znajduje się w trybie krajobrazowym, oznacza to,
            // że klasa MainActivity wyświetla zarówno
            // tytuły, jak i tekst, zatem mniejsza aktywność jest
            // już niepotrzebna. Zignorujmy ją i pozwólmy klasie MainActivity
            // zająć się wszystkimi zadaniami.
            finish();
            return;
        }

        if(getIntent() != null) {
            // Jest to inny sposób utworzenia wystąpienia fragmentu
```

```
// przechowującego szczegóły.  
        DetailsFragment details =  
            DetailsFragment.newInstance(getIntent().getExtras());  
        getFragmentManager().beginTransaction()  
            .add(android.R.id.content, details)  
            .commit();  
    }  
}  
}
```

---

Warto przyjrzeć się kilku interesującym aspektom tego kodu. Przede wszystkim jest on naprawdę prosty do implementacji. Dokonujemy prostego określenia trybu orientacji urządzenia i jeżeli znajduje się ono w trybie portretowym, wstawiamy fragment przechowujący szczegóły do oddzielnej aktywności. W trybie krajobrazowym aktywność `MainActivity` może wyświetlać obydwa fragmenty obok siebie, zatem nie ma potrzeby pokazywania dodatkowej aktywności. Czytelnik może się zastanawiać, po co chcielibyśmy w ogóle tworzyć tę aktywność w trybie krajobrazowym. Odpowiedź jest prosta: nie chcemy. Jeżeli jednak aktywność przechowująca szczegóły została utworzona w trybie portretowym i użytkownik obróci urządzenie do trybu krajobrazowego, zostanie ona uruchomiona ponownie z powodu zmiany konfiguracji. Otrzymaliśmy więc dodatkową aktywność w trybie krajobrazowym. W tym momencie jedynym rozsądnym rozwiązaniem okazuje się zakończenie tej aktywności i przekazanie klasie `MainActivity` wszystkich zadań.

Kolejnym interesującym aspektem aktywności przechowującej szczegóły jest brak możliwości ustawienia głównego widoku treści za pomocą metody `setContentView()`. W jaki więc sposób zostaje utworzony interfejs użytkownika? Jeżeli przyjrzymy się uważnie wywołaniu metody `add()` w transakcji fragmentu, zauważymy, że pojemnik widoków, do którego dodajemy dany fragment, został określony jako zasób `android.R.id.content`. Jest to główny pojemnik widoków aktywności, zatem jeśli dołączamy do niego hierarchię widoków fragmentów, oznacza to, że hierarchia ta staje się jedną hierarchią widoków w aktywności. Do utworzenia nowego fragmentu (na przykład przyjmującego pakiet w postaci argumentu) wykorzystaliśmy tutaj dokładnie taką samą klasę `DetailsFragment` co wcześniej, lecz wprowadziliśmy inną metodę `newInstance()`, a następnie dołączliśmy go po prostu do głównego poziomu hierarchii widoków aktywności. W ten sposób fragment zostaje wyświetlony we wnętrzu tej aktywności.

Z punktu widzenia użytkownika ogląda on teraz jedynie widok zawierający fragment, w którym jest przechowywany tekst sztuki Szekspira. Jeżeli będzie chciał przejść do innego tytułu, musi wcisnąć przycisk cofania, co spowoduje powrót do głównej aktywności (przechowującej wyłącznie fragment z tytułami). Alternatywnym rozwiązaniem jest obrót urządzenia i przejście do trybu krajobrazowego. Wtedy w aktywności przechowującej szczegóły zostanie wywołana metoda `finish()`, co spowoduje jej zamknięcie i pojawiienie się odtworzonej aktywności głównej.

Kiedy urządzenie znajduje się w trybie portretowym i jeśli w głównej aktywności nie wyświetlamy fragmentu przechowującego szczegóły, należy utworzyć osobny plik układu graficznego `main.xml`, którego zawartość została zaprezentowana na listingu 29.11.

---

**Listing 29.11.** Układ graficzny aktywności dla trybu portretowego

---

```
<?xml version="1.0" encoding="utf-8"?>  
<!-- Jest to plik res/layout/main.xml -->  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"
```

---

```

        android:layout_width="match_parent"
        android:layout_height="match_parent">
<fragment class="com.androidbook.fragments.bard.TitlesFragment"
        android:id="@+id/titles"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>

```

---

Oczywiście, ten układ graficzny można zmodyfikować w dowolny sposób. W celach demonstracyjnych służy on jedynie do wyświetlania fragmentu przechowującego tytuły. Bardzo dobrze się stało, że klasa tego fragmentu nie wymaga dużej ilości kodu do przetwarzania zmian konfiguracji urządzenia.

Ostatnim elementem, jaki chcemy dołączyć w tym przykładzie, jest plik *AndroidManifest.xml*, zaprezentowany na listingu 29.12.

**Listing 29.12.** Plik AndroidManifest.xml

---

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0" package="com.androidbook.fragments.bard">
    <uses-sdk android:minSdkVersion="11" />

    <application android:icon="@drawable/icon"
        android:label="Szekspir">

        <activity
            android:name="com.androidbook.fragments.bard.MainActivity"
            android:label="Szekspir">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity
            android:name="com.androidbook.fragments.bard.DetailsActivity"
            android:label="Szekspir szczegółы">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>

```

---

Jest to standardowy plik manifest. Widzimy główną aktywność zawierającą kategorię LAUNCHER, dzięki czemu zostanie ona umieszczona na liście aplikacji urządzenia. Widzimy następnie oddzielną aktywność DetailsActivity ze zdefiniowaną kategorią DEFAULT. W ten sposób możemy uruchomić tę aktywność za pomocą kodu, nie zostanie ona jednak umieszczona na liście aplikacji.

## Trwałość fragmentów

W trakcie testowania omawianej aplikacji nie należy zapominać o obracaniu urządzenia (w emulatorze dokonamy tego za pomocą skrótu klawiaturowego *Ctrl+F11*). Zauważymy, że wraz z obrotem urządzenia obracać się będą również fragmenty. Jeżeli Czytelnik będzie śledzić komunikaty w oknie *LogCat*, zauważ, że aplikacja wygeneruje ich bardzo dużo. W szczególności należy zwrócić uwagę na te komunikaty wyświetlane w momencie obracania urządzenia, które dotyczą fragmentów; usuwana i odtwarzana jest nie tylko aktywność, lecz również fragmenty.

Dotychczas napisaliśmy jedynie niewielką ilość kodu do obsługi fragmentu przechowującego szczegóły. Kod ten służy do zachowywania bieżącej pozycji na liście tytułów w przypadku ponownego uruchomienia aktywności. W przypadku fragmentów przechowujących szczegóły nie musielibyśmy wprowadzać kodu obsługującego zmiany konfiguracji, ponieważ nie ma takiej potrzeby. Android sam obsługuje przechowywanie fragmentów znajdujących się w menedżerze, ich zachowywanie, a następnie odczytywanie w przypadku odtwarzania stanu aktywności. Czytelnik powinien mieć już świadomość, że fragmenty otrzymywane po zmianie konfiguracji najprawdopodobniej nie są tymi samymi fragmentami, które wcześniej znajdowały się w pamięci. Fragmenty te zostały zrekonstruowane. System zachował pakiet argumentów oraz informacje o typie fragmentu, a w przypadku każdego fragmentu przechowującego zapisane informacje o stanie zachował również pakiet z atrybutami zachowanego stanu, służące do późniejszego ich odtworzenia.

Komunikaty wyświetlane w oknie *LogCat* informują nas o fragmentach przechodzących przez cykl życia w synchronizacji z cyklem życia aktywności. Zauważymy, że fragment przechowujący szczegóły zostaje odtworzony, lecz system nie wywołuje ponownie metody *newInstance()*. Zamiast tego Android korzysta po prostu z domyślnego konstruktora, następnie dołącza do niego pakiet argumentów i rozpoczyna wywoływanie metod zwrotnych danego fragmentu. Dlatego tak ważne jest, aby nie umieszczać żadnego wymyślnego kodu w metodzie *newInstance()*, ponieważ w momencie odtwarzania fragmentu metoda ta zostanie pominięta.

Czytelnik powinien już także docenić możliwość wielokrotnego użytkowania fragmentów w różnych miejscach. Fragment przechowujący tytuły jest wykorzystywany w dwóch różnych układach graficznych, jeśli jednak przyjrzymy się jego kodowi, zauważymy, że atrybuty umieszczone w tych plikach nie mają większego znaczenia. Moglibyśmy utworzyć dwa zupełnie różniące się od siebie układy graficzne, a kod tego fragmentu wyglądałby dokładnie tak samo. To samo można powiedzieć o fragmencie przechowującym szczegóły. Został on wprowadzony do głównego układu graficznego trybu krajobrazowego oraz w aktywności przechowującej szczegóły. Także i w tym przypadku układy graficzne mogą się od siebie znacznie różnić, a kod fragmentu przechowującego szczegóły nie uległby zmianom. Również kod aktywności przechowującej szczegóły był bardzo prosty.

Do tej pory analizowaliśmy dwa typy fragmentów: klasę bazową *Fragment* oraz jej podkласę *ListFragment*. Przejdziemy teraz do kolejnego elementu klasy *Fragment*, jakim jest klasa podzielona *DialogFragment*.

## Fragmenty wyświetlające okna dialogowe

W rozdziale 8. omówiliśmy mechanizm okien dialogowych w wersjach systemu starszych od 3.0. Wraz z wersją 3.0 Androida wprowadzono nowy sposób pracy z oknami dialogowymi, oparty na fragmentach. Spodziewamy się, że omówiony w rozdziale 8. protokół zarządzanych okien dialogowych zostanie wyparty przez rozwiążanie omówione poniżej.

W tym podrozdziale Czytelnik dowie się, w jaki sposób wykorzystywać fragmenty do wyświetlania prostego okna dialogowego oraz niestandardowego okna dialogowego, ukazującego tekst zachęty.

## Podstawowe informacje o klasie DialogFragment

Zanim zademonstrujemy przykładowe aplikacje wyświetlające okno dialogowe zachęty oraz alert, chcielibyśmy najpierw zapoznać Czytelnika z teoretycznymi podstawami stosowania fragmentów wyświetlających okna dialogowe. Funkcjonalności związane z oknami dialogowymi w Androidzie 3.0 zostały zawarte w klasie `DialogFragment`. Stanowi ona podelement klasy `Fragment` i w istocie posiada właściwości fragmentu. Będzie to zatem nasza klasa bazowa, służąca do obsługi okien dialogowych. Gdy już utworzymy okno dialogowe pochodzące z tej klasy, na przykład:

```
public class MyDialogFragment extends DialogFragment { ... }
```

możemy wyświetlić taki fragment `MyDialogFragment` w postaci okna dialogowego przy użyciu transakcji fragmentu. Na listingu 29.13 umieściliśmy pseudokod obrazujący ten proces.

**Listing 29.13.** Wyświetlanie fragmentu przechowującego dialog

---

```
JakasAktywnosc
{
    //...pozostałe funkcje aktywności
    public void showDialog()
    {
        //konstruuje klasę MyDialogFragment
        MyDialogFragment mdf = MyDialogFragment.newInstance(arg1,arg2);
        FragmentManager fm = getFragmentManager();
        FragmentTransaction ft = fm.beginTransaction();
        mdf.show(ft,"znacznik-mojego-dialogu");
    }
    //...pozostałe funkcje aktywności
}
```

---

Zgodnie z listingu 29.13, aby wyświetlić fragment przechowujący okna dialogowe, należy wykonać następujące czynności:

1. Utworzenie fragmentu wyświetlającego okna dialogowe.
2. Uruchomienie transakcji fragmentu.
3. Wyświetlenie okna dialogowego za pomocą transakcji utworzonej na etapie 2.

Przyjrzyjmy się każdemu z wymienionych etapów.

## Utworzenie fragmentu wyświetlającego okna dialogowe

Podczas tworzenia fragmentu wyświetlającego okna dialogowe kierujemy się takimi zasadami jak w przypadku pozostałych typów fragmentów. Zalecanym wzorcem jest stosowanie metody fabrykującej, takiej jak `newInstance()`. W jej wnętrzu powinniśmy wykorzystać domyślny konstruktor, a następnie dodać pakiet z argumentami, zawierający przekazywane parametry. Metoda ta nie powinna służyć do innych zadań, ponieważ chcemy mieć pewność, że niczego nie pominiemy w procesie odtwarzania fragmentu z zachowanego stanu. W takim przypadku Android wywoła domyślny konstruktor i odtworzy za jego pomocą pakiet zawierający argumenty.

## Przesłanianie metody onCreateView

Podczas dziedziczenia fragmentu wyświetlającego okna dialogowe musimy przesłonić jedną lub dwie metody w celu wprowadzenia hierarchii widoków tego okna dialogowego. Pierwszą możliwością jest przesłanie metody onCreateView() i uzyskanie widoku. Drugą opcję stanowi przesłanie metody onCreateDialog() i otrzymanie obiektu Dialog (takiego, jaki był tworzony przez klasę AlertDialog.Builder).

Na listingu 29.14 zaprezentowaliśmy przykład przesłania metody onCreateView().

**Listing 29.14.** Przesłanianie metody onCreateView() klasy DialogFragment

---

```
MyDialogFragment
{
    ...inne funkcje
    public View onCreateView(LayoutInflater inflater,
                           ViewGroup container, Bundle savedInstanceState)
    {
        //Tworzy widok poprzez rozwinięcie wybranego układu graficznego
        View v =
            inflater.inflate(R.layout.prompt_dialog,container,false);

        //Możemy zlokalizować widok i wprowadzić wartości
        TextView tv = (TextView)v.findViewById(R.id.promptmessage);
        tv.setText(this.getPrompt());

        //Możemy wprowadzić metody zwrotne dla przycisków
        Button dismissBtn = (Button)v.findViewById(R.id.btn_dismiss);
        dismissBtn.setOnClickListener(this);

        Button saveBtn = (Button)v.findViewById(
            R.id.btn_save);
        saveBtn.setOnClickListener(this);
        return v;
    }
    ...inne funkcje
}
```

---

W kodzie z listingu 29.14 wczytujemy widok zdefiniowany przez układ graficzny. Następnie wyszukujemy dwa przyciski i określamy dla nich metody zwrotne. W bardzo podobny sposób tworzyliśmy wcześniej fragment przechowujący szczegóły. W przeciwieństwie jednak do przedstawionych wcześniej fragmentów, omawiany typ zawiera jeszcze jeden mechanizm pozwalający na utworzenie hierarchii widoków.

## Przesłanianie metody onCreateDialog

Alternatywą dla umieszczenia widoku w metodzie onCreateView() jest przesłanie metody onCreateDialog() i dostarczenie wystąpienia okna dialogowego. Na listingu 29.15 został zaprezentowany przykładowy kod takiego rozwiązania.

**Listing 29.15.** Przesłonięcie metody onCreateDialog() klasy DialogFragment

```
MyDialogFragment
{
    ...inne funkcje
    @Override
    public Dialog onCreateDialog(Bundle icicle)
    {
        AlertDialog.Builder b = new AlertDialog.Builder(getActivity());
        b.setTitle("Tytuł mojego okna dialogowego");
        b.setPositiveButton("OK", this);
        b.setNegativeButton("Anuluj", this);
        b.setMessage(this.getMessage());
        return b.create();
    }
    ...inne funkcje
}
```

W tym przykładzie wykorzystujemy konstruktor alertów do utworzenia przekazywanego okna dialogowego. Jest to skuteczne rozwiązanie w przypadku prostych okien dialogowych. Pierwsze rozwiązanie, polegające na przesłonięciu metody onCreateView(), jest równie proste i zapewnia o wiele większą swobodę.

## Wyświetlanie fragmentu przechowującego okna dialogowe

Po utworzeniu fragmentu wyświetlającego okno dialogowe wymagane będzie wykorzystanie transakcji fragmentu. Podobnie jak w przypadku pozostałych typów fragmentów, także i tutaj operacje przeprowadzane są za pośrednictwem transakcji.

Metoda show() przyjmuje transakcję w postaci parametru wejściowego. Widzimy to na listingu 29.13. Za pomocą transakcji metoda ta dodaje okno dialogowe do aktywności, a następnie zatwierdza tę transakcję. Metoda show() nie dodaje jednak transakcji do stosu drugoplanowego. Jeżeli chcemy, możemy najpierw dodać transakcję do stosu, a następnie przekazać ją metodzie show(). Metoda ta w przypadku fragmentu wyświetlającego okna dialogowego posiada następujące sygnatury:

```
public int show(FragmentTransaction transaction, String tag)
public int show(FragmentManager manager, String tag)
```

Pierwsza metoda show() wyświetla okno dialogowe poprzez dodanie tego fragmentu do przekazanej transakcji wraz z określonym znacznikiem. Przekazuje ona następnie identyfikator przeprowadzanej transakcji.

Druga metoda show() automatyzuje proces otrzymywania transakcji z menedżera transakcji. Jest to metoda skrócona. Jednak w przypadku korzystania z niej tracimy możliwość umieszczenia transakcji w stosie drugoplanowym. Jeżeli chcemy uzyskać kontrolę nad tym aspektem, musimy zastosować metodę o pierwszej z wymienionych sygnatur. Druga metoda okazuje się przydatna, gdy chcemy wyświetlić jedynie okno dialogowe i nie mamy innego powodu, aby w danym momencie przeprowadzać transakcję fragmentu.

Bardzo miłą implikacją okna dialogowego w postaci fragmentu jest fakt, iż menedżer fragmentów zarządza stanami w podstawowym zakresie. Jeśli na przykład urządzenie zostanie obrócone w chwili wyświetlanego okna dialogowego, zostanie ono odtworzone całkowicie bez naszego udziału.

Fragment wyświetlający okna dialogowe zawiera również metody pozwalające na kontrolowanie ramki, w której jest wyświetlany widok okna dialogowego, w tym takie właściwości, jak jej tytuł lub wygląd. Więcej opcji znajdziemy w dokumentacji klasy `DialogFragment`; łącze do tej dokumentacji zamieszczono na końcu rozdziału.

## Odwolanie fragmentu wyświetlającego okna dialogowe

Fragment wyświetlający okna dialogowe można odwołać na dwa sposoby. Pierwszym z nich jest jawne wywołanie metody `dismiss()` w odpowiedzi na wcisnięcie przycisku lub jakieś działanie na widoku okna dialogowego, co zostało ukazane na listingu 29.16.

**Listing 29.16.** Wywołanie metody `dismiss()`

---

```
if (someview.getId() == R.id.btn_Dismiss)
{
    //Wykorzystajmy jakieś metody zwrotne powiadamiające klientów
    //tego okna dialogowego o jego odwołaniu,
    //a następnie wywołajmy omawianą metodę.
    dismiss();
    return;
}
```

---

Metoda `dismiss()` usunie ten fragment z menedżera fragmentów, a następnie przeprowadzi odpowiednią transakcję. Jeżeli dany fragment znajduje się na stosie drugoplanowym, metoda ta spowoduje po prostu jego wycofanie, a na jego miejsce wejdzie poprzedni stan transakcji fragmentu. Bez względu na to, czy dostępny jest stos drugoplanowy, czy nie, wywołanie metody `dismiss()` poskutkuje wywołaniem standardowych metod zwrotnych usuwających fragment wyświetlający okna dialogowe, w tym również `onDismiss()`.

Należy zauważyć, że obecność metody `onDismiss()` wcale nie musi oznaczać wywołania metody `dismiss()`. Wynika to z faktu, iż metoda `onDismiss()` jest wywoływana również podczas zmiany konfiguracji urządzenia i z tego powodu nie nadaje się do określania czynności, jakie użytkownik przeprowadził na oknie dialogowym. Jeżeli okno dialogowe jest wyświetlane w trakcie zmiany trybu wyświetlania obrazu, we fragmencie zostanie wywołana metoda `onDismiss()`, nawet jeśli użytkownik nie wcisnął żadnego przycisku w tym oknie dialogowym. Zamiast tego powinniśmy prawdopodobnie zawsze polegać na jawnych zdarzeniach kliknięć przycisku znajdującego się w widoku okna dialogowego.

Jeżeli użytkownik wcisnie przycisk cofania w trakcie wyświetlania fragmentu zawierającego okno dialogowe, spowoduje to wywołanie metody zwrotnej `onCancel()` w tym fragmencie. Domyślnie system pozbędzie się fragmentu i nie będzie trzeba samodzielnie wywoływać metody `dismiss()`. Jeżeli jednak chcemy, aby aktywność wywołująca została powiadomiona o anulowaniu okna dialogowego, będziemy musieli w tym celu wprowadzić odpowiednią logikę do metody `onCancel()`. Na tym polega różnica pomiędzy metodami `onDismiss()` i `onCancel()`.

we fragmentach wyświetlających okna dialogowe. W przypadku metody `onDismiss()` cały czas nie będziemy mieli pewności, co spowodowało jej wywołanie. Mogliśmy również odnotować fakt, że fragment ten nie posiada metody `cancel()`, jedynie `dismiss()`, lecz — jak już stwierdziliśmy — w momencie wcisnięcia przycisku cofania system samodzielnie zajmuje się procesem jego anulowania czy też wycofania.

Alternatywnym sposobem wycofania fragmentu wyświetlającego okno dialogowe jest wprowadzenie innego fragmentu tego typu. Mechanizm wycofania starego fragmentu i wprowadzenia nowego różni się nieco od zwyczajnego wycofania bieżącego fragmentu. Na listingu 29.17 zamieściliśmy stosowny przykład.

**Listing 29.17.** Konfigurowanie dialogu przeznaczonego do stosu drugoplanowego

---

```
if (someview.getId() == R.id.btn_invoke_another_dialog)
{
    Activity act = getActivity();
    FragmentManager fm = act.getFragmentManager();
    FragmentTransaction ft = fm.beginTransaction();
    ft.remove(this);

    ft.addToBackStack(null);
    //Wartość null symbolizuje brak nazwy dla transakcji przeprowadzanej w stosie drugoplanowym

    HelpDialogFragment hdf =
        HelpDialogFragment.newInstance(R.string.helptext);
    hdf.show(ft, "NA POMOC");
    return;
}
```

---

W obrębie jednej transakcji usuwamy bieżący fragment i dodajemy nowy fragment wyświetlający okno dialogowe. Wizualnie poprzednie okno dialogowe znika, a w jego miejscu pojawia się nowe. Jeżeli użytkownik wcisnie przycisk cofania, nowe okno dialogowe zostanie wycofane i zostanie wyświetcone poprzednie okno dialogowe, ponieważ zachowaliśmy tę transakcję w stosie drugoplanowym. Jest to bardzo przydatny sposób wyświetlania na przykład okna dialogowego pomocy.

## Skutki wycofania okna dialogowego

Gdy dodajemy dowolny fragment do menedżera fragmentów, menedżer ten będzie zarządzał jego stanami. Oznacza to, że w trakcie zmiany konfiguracji urządzenia (wynikającego na przykład ze zmiany orientacji wyświetlacza) aktywność, wraz z fragmentami, zostanie uruchomiona ponownie. Zostało to zademonstrowane na przykładzie zawierającym cytaty z dzieł Szekspira.

Zmiana konfiguracji urządzenia nie wpływa na okna dialogowe, ponieważ są one również zarządzane przez menedżer fragmentów. Jednak niejawne zachowanie metod `show()` i `dismiss()` sprawia, że jeśli nie zachowamy ostrożności, dość łatwo możemy zgubić dany fragment wyświetlający okna dialogowe. Metoda `show()` automatycznie dodaje fragment do menedżera; z kolei metoda `dismiss()` automatycznie go stamtąd usuwa. Być może uzyskamy bezpośredni wskaźnik fragmentu przed jego wyświetleniem, nie będziemy jednak mogli dodać później tego fragmentu do menedżera za pomocą metody `show()`, ponieważ fragment może zostać dodany

do tego menedżera tylko jednorazowo. Może zaistnieć potrzeba odczytania tego wskaźnika poprzez odtworzenie aktywności. Jeżeli jednak chcemy pokazać, a następnie wycofać okno dialogowe, fragment zostanie niejawnie usunięty z menedżera fragmentów, co jest jednoznaczne z uniemożliwieniem jego odtworzenia i ponownego wskazania (ponieważ menedżer nie będzie posiadał informacji, że fragment ten istnieje).

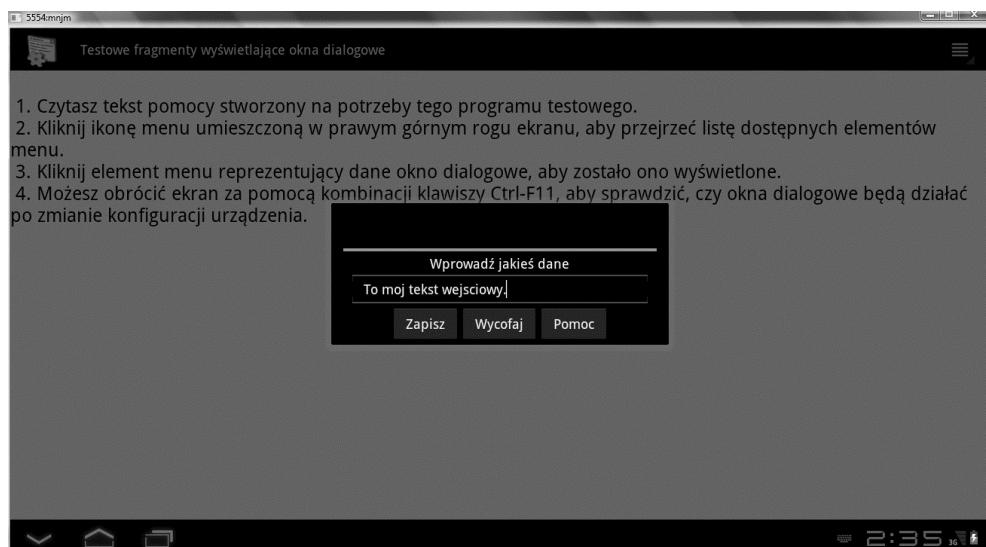
Jeżeli chcemy utrzymać stan okna dialogowego po jego wycofaniu, będziemy musieli go w jakiś sposób przechować na zewnątrz — albo w nadzędnej aktywności, albo we fragmencie niezwiązanym z oknami dialogowymi, który nie zostanie szybko usunięty.

## Przykładowa aplikacja wykorzystująca klasę DialogFragment

Utworzmy teraz przykładową aplikację, za pomocą której przedstawimy trzy koncepcje fragmentu wyświetlającego okna dialogowe. Przyjrzymy się również mechanizmowi komunikacji pomiędzy fragmentem a przechowującą go aktywnością. Aby tego dokonać, będziemy potrzebować pięciu plików:

- *MainActivity.java* jest główną aktywnością aplikacji. Będzie ona wyświetlała prosty widok zawierający tekst pomocniczy oraz menu, za pomocą którego będą uruchamiane okna dialogowe.
- *PromptDialogFragment.java* stanowi przykład fragmentu wyświetlającego okna dialogowe, w którym zostaje zdefiniowany osobny plik układu graficznego. Jest tu możliwa interakcja z użytkownikiem. Dostępne są trzy przyciski: *Zachowaj*, *Wycofaj* (na przykład służący do anulowania) oraz *Pomoc*.
- *AlertDialogFragment.java* to przykładowy fragment wyświetlający okna dialogowe, który wykorzystuje klasę *AlertDialogBuilder* do utworzenia okna dialogowego wewnętrz tego fragmentu. Mamy tu do czynienia z klasyczną metodą tworzenia okna dialogowego; możemy wykorzystać tu wiedzę nabycią podczas tworzenia zwykłych okien dialogowych.
- *HelpDialogFragment.java* jest bardzo prostym fragmentem wyświetlającym komunikat pomocy, umieszczony w zasobach aplikacji. Dana wiadomość zostaje określona w momencie tworzenia okna dialogowego. Obiekt ten jest wyświetlany zarówno z poziomu głównej aktywności, jak i fragmentu wyświetlającego okno zachęty.
- *OnDialogDoneListener.java* zawiera w sobie interfejs wymagany przez aktywność w celu otrzymywania komunikatów pochodzących z fragmentów. Stosując ten interfejs, stwierdzamy, że fragmenty nie muszą posiadać wielu informacji na temat aktywności wywołującej, wystarczy im informacja, że aktywność ta implementuje omawiany interfejs. W ten sposób wszystkie funkcje mogą się znajdować na swoich miejscach. Z punktu widzenia aktywności jest to bardzo popularny sposób otrzymywania komunikatów z fragmentów bez posiadania zbyt wielu informacji na ich temat.

Nasza przykładowa aplikacja zawiera trzy układy graficzne: dla głównej aktywności, dla fragmentu wyświetlającego okno zachęty oraz dla fragmentu wyświetlającego okno pomocy. Zaauważmy, że nie potrzebujemy układu graficznego dla fragmentu wyświetlającego alert, ponieważ jego utworzeniem zajmie się klasa *AlertDialogBuilder*. Po utworzeniu i uruchomieniu aplikacji zobaczymy ekran widoczny na rysunku 29.3.



Rysunek 29.3. Interfejs użytkownika przykładowej aplikacji — fragment przechowujący okno dialogowe

## Przykładowe okno dialogowe — klasa MainActivity

Przejdźmy do kodu źródłowego. Na listingu 29.18 znajduje się kod głównej aktywności.

**Listing 29.18.** Główna aktywność fragmentu wyświetlającego okno dialogowe

```
// Jest to plik MainActivity.java
import android.app.Activity;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.Toast;

public class MainActivity extends Activity
implements OnDialogDoneListener
{
    public static final String LOGTAG = "DialogFragmentDemo";
    public static final String ALERT_DIALOG_TAG = "ALERT_DIALOG_TAG";
    public static final String HELP_DIALOG_TAG = "HELP_DIALOG_TAG";
    public static final String PROMPT_DIALOG_TAG = "PROMPT_DIALOG_TAG";

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        FragmentManager.enableDebugLogging(true);
    }
}
```

```
@Override
public boolean onCreateOptionsMenu(Menu menu){
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item)
{
    if (item.getItemId() == R.id.menu_show_alert_dialog)
    {
        this.testAlertDialog();
        return true;
    }
    if (item.getItemId() == R.id.menu_show_prompt_dialog)
    {
        this.testPromptDialog();
        return true;
    }
    if (item.getItemId() == R.id.menu_help)
    {
        this.testHelpDialog();
        return true;
    }
    return true;
}

private void testPromptDialog()
{
    FragmentTransaction ft = getFragmentManager().beginTransaction();

    PromptDialogFragment pdf =
        PromptDialogFragment.newInstance("Wprowadź jakieś dane");

    pdf.show(ft, PROMPT_DIALOG_TAG);
}

private void testAlertDialog()
{
    FragmentTransaction ft = getFragmentManager().beginTransaction();

    AlertDialogFragment adf =
        AlertDialogFragment.newInstance("Komunikat alertu");

    adf.show(ft, ALERT_DIALOG_TAG);
}

private void testHelpDialog()
{
    FragmentTransaction ft = getFragmentManager().beginTransaction();

    HelpDialogFragment hdf =
        HelpDialogFragment.newInstance(R.string.help_text);
```

---

```

        hdf.show(ft, HELP_DIALOG_TAG);
    }

    public void onDialogDone(String tag, boolean cancelled,
                           CharSequence message) {
        String s = tag + " reaguje na: " + message;
        if(cancelled)
            s = tag + " zostało anulowane przez użytkownika";
        Toast.makeText(this, s, Toast.LENGTH_LONG).show();
        Log.v(LOGTAG, s);
    }
}

```

---

Kod głównej aktywności jest niezwykle prosty. W metodzie `onCreate()` ustanawiamy widok treści i włączamy debugowanie menedżera fragmentów. Widzimy następnie dwie metody związane z konfigurowaniem opcji menu. Wybór poszczególnych opcji menu powoduje wywołanie różnych metod o prostej budowie. Każda z tych metod wykonuje praktycznie takie samo zadanie: pozyskuje transakcję fragmentu, a następnie tworzy i wyświetla dany fragment. Zwrócmy uwagę, że każdy fragment posiada niepowtarzalny znacznik, który zostaje dostarczony metodzie `show()`. Znacznik ten zostaje powiązany z fragmentem w menedżerze, dzięki czemu możemy później lokalizować fragmenty. Fragmenty mogą również same określać wartość znacznika za pomocą metody `getTag()` klasy `Fragment`.

Ostatnią definicją metody w naszej głównej aktywności jest `onDialogDone()`, która jest częścią implementowanego interfejsu `OnDialogDoneListener`. Jak widać, omawiana metoda zwrotna zawiera znacznik wywołującego fragmentu, wartość logiczną wskazującą, czy fragment został anulowany, oraz komunikat. Dla naszych celów wystarczy wiedza, że komunikat zostaje wyświetlony w oknie `LogCat`; jest on również prezentowany użytkownikowi za pomocą kontrolki `Toast`.

## Przykładowe okno dialogowe — interfejs `OnDialogDoneListener`

Skoro pokazaliśmy, w jaki sposób ustalić moment zniknięcia okna dialogowego, utworzymy interfejs obiektu nasłuchującego implementowany przez obiekty, które wywołują okna dialogowe. Kod tego interfejsu został zaprezentowany na listingu 29.19.

### **Listing 29.19.** Interfejs obiektu nasłuchującego

---

```

// Jest to plik OnDialogDoneListener.java
/*
 * Interfejs standardowo implementowany przez aktywność,
 * dzięki czemu okno dialogowe może przesyłać komunikaty
 * o zdarzeniach.
 */
public interface OnDialogDoneListener {
    public void onDialogDone(String tag, boolean cancelled, CharSequence message);
}

```

---

Jak widać, mamy do czynienia z bardzo prostym interfejsem. Wybraliśmy tylko jedną metodę zwrotną dla tego interfejsu. Metoda ta koniecznie musi być zaimplementowana przez aktywność. Nasze fragmenty nie muszą posiadać informacji o szczegółach aktywności wywołującej, a jedynie o tym, że aktywność ta musi implementować interfejs `OnDialogDoneListener`.

Fragmenty mogą więc za pomocą tej metody zwrotnej komunikować się z aktywnością wywołującą. W zależności od przeznaczenia fragmentu w interfejsie tym może się znajdować wiele metod zwrotnych. Nasza przykładowa aplikacja rozdziela interfejs od definicji klas fragmentów. Aby ułatwić sobie zarządzanie kodem, możemy zamieścić interfejs obiektu nasłuchującego fragment wewnętrz samej definicji klasy fragmentu, a tym samym zapewnić sobie większą kontrolę nad synchronizacją obiektu nasłuchującego z fragmentem.

## Przykładowe okno dialogowe — klasa PromptDialogFragment

Przyjrzyjmy się teraz pierwszemu fragmentowi — `PromptDialogFragment`, którego układ graficzny oraz kod Java zostały razem umieszczone na listingu 29.20.

**Listing 29.20.** Układ graficzny i kod Java klasy `PromptDialogFragment`

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/prompt_dialog.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:padding="4dip"
    android:gravity="center_horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/promptmessage"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_marginLeft="20dip"
        android:layout_marginRight="20dip"
        android:text="Wprowadź tekst"
        android:layout_weight="1"
        android:layout_gravity="center_vertical|center_horizontal"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:gravity="top|center_horizontal" />

    <EditText
        android:id="@+id/inputtext"
        android:layout_height="wrap_content"
        android:layout_width="400dip"
        android:layout_marginLeft="20dip"
        android:layout_marginRight="20dip"
        android:scrollHorizontally="true"
        android:autoText="false"
        android:capitalize="none"
        android:gravity="fill_horizontal"
        android:textAppearance="?android:attr/textAppearanceMedium" />

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">

        <Button android:id="@+id/btn_save"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0"
```

```
        android:text="Zachowaj">
    </Button>

    <Button android:id="@+id/btn_dismiss"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:text="Wycofaj">
    </Button>

    <Button android:id="@+id/btn_help"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:text="Pomoc">
    </Button>

</LinearLayout>
</LinearLayout>

// Jest to plik PromptDialogFragment.java
import android.app.Activity;
import android.app.DialogFragment;
import android.app.FragmentTransaction;
import android.content.DialogInterface;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class PromptDialogFragment
extends DialogFragment
implements View.OnClickListener
{
    private EditText et;

    public static PromptDialogFragment
newInstance(String prompt)
{
    PromptDialogFragment pdf = new PromptDialogFragment();
    Bundle bundle = new Bundle();
    bundle.putString("zachęta",prompt);
    pdf.setArguments(bundle);

    return pdf;
}

@Override
public void onAttach(Activity act) {
    // Jeżeli aktywność, do której dołączylismy, nie
    // posiada zaimplementowanego interfejsu OnDialogDoneListener,
```

```
// poniższy wiersz spowoduje wyświetlenie wyjątku
// ClassCastException. Jest to najwcześniejszy etap,
// w którym możemy przetestować zachowanie aktywności.
OnDialogDoneListener test = (OnDialogDoneListener)act;
super.onAttach(act);
}

@Override
public void onCreate(Bundle icicle)
{
    super.onCreate(icicle);
    this.setCancelable(true);
    int style = DialogFragment.STYLE_NORMAL, theme = 0;
    setStyle(style,theme);
}

public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle icicle)
{
    View v = inflater.inflate(R.layout.prompt_dialog, container,
        false);

    TextView tv = (TextView)v.findViewById(R.id.promptmessage);
    tv.setText(getArguments().getString("zachęta"));

    Button dismissBtn = (Button)v.findViewById(R.id.btn_dismiss);
    dismissBtn.setOnClickListener(this);

    Button saveBtn = (Button)v.findViewById(R.id.btn_save);
    saveBtn.setOnClickListener(this);

    Button helpBtn = (Button)v.findViewById(R.id.btn_help);
    helpBtn.setOnClickListener(this);

    et = (EditText)v.findViewById(R.id.inputtext);
    if(icicle != null)
        et.setCharSequence(icicle.getCharSequence("wejście"));
    return v;
}

@Override
public void onSaveInstanceState(Bundle icicle) {
    icicle.putCharSequence("wejście", et.getText());
    super.onPause();
}

@Override
public void onCancel(DialogInterface di) {
    Log.v(MainActivity.LOGTAG, "w metodzie onCancel () fragmentu PDF");
    super.onCancel (di);
}

@Override
public void onDismiss(DialogInterface di) {
    Log.v(MainActivity.LOGTAG, "w metodzie onDismiss() fragmentu PDF");
    super.onDismiss(di);
```

```

}

public void onClick(View v)
{
    OnDialogDoneListener act = (OnDialogDoneListener) getActivity();
    if (v.getId() == R.id.btn_save)
    {
        TextView tv =
            (TextView) getView().findViewById(R.id.inputtext);
        act.onDialogDone(this.getTag(), false, tv.getText());
        dismiss();
        return;
    }
    if (v.getId() == R.id.btn_dismiss)
    {
        act.onDialogDone(this.getTag(), true, null);
        dismiss();
        return;
    }
    if (v.getId() == R.id.btn_help)
    {
        FragmentTransaction ft =
            getFragmentManager().beginTransaction();
        ft.remove(this);

        // W tym przypadku chcemy wyświetlić tekst pomocy
        // i po zakończeniu powrócić do poprzedniego okna dialogowego.
        ft.addToBackStack(null);
        //Wartość null oznacza brak nazwy dla transakcji stosu drugoplanowego.

        HelpDialogFragment hdf =
            HelpDialogFragment.newInstance(R.string.help1);
        hdf.show(ft, MainActivity.HELP_DIALOG_TAG);
        return;
    }
}
}

```

Układ graficzny okna dialogowego zachęty nie różni się od wcześniej tworzonych układów graficznych. Widzimy w nim kontrolkę `TextView` zawierającą tekst zachęty, kontrolkę `EditText`, w której użytkownik wprowadza własne dane, oraz trzy przyciski, zapewniające obsługę, kolejno, zachowania danych wejściowych, wycofania (na przykład anulowania) fragmentu wyświetlającego okna dialogowe oraz wyświetlania okna dialogowego pomocy.

Kod klasy `PromptDialogFragment` na początku niczym się nie różni od innych wcześniej utworzonych fragmentów. Znalazła się tu statyczna metoda `newInstance()` służąca do tworzenia nowych obiektów, a w jej wnętrzu wywołujemy domyślny konstruktor i generujemy pakiet argumentów, który zostaje następnie dołączony do tego obiektu. Następnie Czytelnik zapewne spostrzegł coś nowego w metodzie `onAttach()`. Chodzi o to, aby się upewnić, że aktywność dołączająca posiada zaimplementowany interfejs `OnDialogDoneListener`. Aby to sprawdzić, rzutujemy aktywność przekazywaną do interfejsu `OnDialogDoneListener`. Jeżeli interfejs nie jest zaimplementowany w aktywności, zostanie wyświetlony wyjątek `ClassCastException`. Moglibyśmy spróbować obejść ten wyjątek i wprowadzić bardziej eleganckie rozwiązanie, zależy nam jednak na utrzymaniu jak najmniejszej złożoności kodu.

Następnie umieściliśmy metodę zwrotną `onCreate()`. Zgodnie z powszechnym trendem związanym z pracą z fragmentami nie tworzymy tutaj interfejsu użytkownika, lecz możemy zdefiniować styl okna dialogowego. Jest to rozwiązanie specyficzne dla fragmentów wyświetlających okna dialogowe. Możemy samodzielnie ustalić styl i motyw lub określić jedynie styl i wprowadzić motyw o wartości 0 (zero), aby pozostawić systemowi swobodę w kwestii jego doboru.

W metodzie `onCreateView()` tworzymy hierarchię widoków danego fragmentu wyświetlającego okno dialogowe. Podobnie jak w przypadku pozostałych typów fragmentów, nie dodajemy hierarchii widoków do przekazywanego pojemnika widoków (na przykład poprzez ustawienie wartości `false` w parametrze `attachToRoot`). Następnie konfigurujemy metody zwrotne przycisków i wstawiamy tekst zachęty do okna dialogowego, które zostało pierwotnie przekazane do metody `newInstance()`. Na końcu sprawdzamy, czy poprzez pakiet zachowanych stanów nie są przekazywane wartości. Wynikałoby z tego, że fragment został odtworzony, najprawdopodobniej w wyniku zmiany konfiguracji, oraz że użytkownik mógł już wprowadzić jakiś tekst. Jeśli tak jest, musimy zapewnić kontrolkę `EditText` informacjami wprowadzonymi przez użytkownika. Pamiętajmy, że z powodu zmiany konfiguracji mamy do czynienia z innym obiektem widoku w pamięci, zatem musimy go zlokalizować i ustawić odpowiedni tekst. Kolejną metodą zwrotną jest `onSaveInstanceState()`; to właśnie w niej zapisujemy w pakiecie dowolny bieżący tekst wprowadzony przez użytkownika.

Metody zwrotne `onCancel()` i `onDismiss()` zaprezentowaliśmy jedynie z powodu możliwości zapisywania informacji w oknie dziennika, więc bez problemu zauważymy, kiedy zostaną one uruchomione w trakcie cyklu życia fragmentu.

Ostatnia metoda we fragmencie wyświetlającym okno zachęty jest przeznaczona do obsługi przycisków. Po raz kolejny uzyskujemy odniesienie do otaczającej aktywności i rzutujemy ją na interfejs, który jest przez nią implementowany. Jeżeli użytkownik wcisnął przycisk *Zapisz*, pobieramy wpisany tekst i wywołujemy metodę zwrotną interfejsu `onDialogDone()`. Jak zostało wcześniej указанie, metoda ta pobiera nazwę znacznika fragmentu, wartość logiczną wskazującą, czy dany fragment został anulowany, oraz komunikat, który w tym przypadku jest tekstem wprowadzonym przez użytkownika.

Następnie wywołujemy metodę `dismiss()`, aby usunąć fragment wyświetlający okno dialogowe. Pamiętajmy, że metoda ta nie tylko wizualnie usuwa fragment przed oczu użytkownika, lecz również wycofuje go z menedżera fragmentów, więc fragment ten staje się zupełnie niedostępny. Jeżeli zostanie wciśnięty przycisk *Wycofaj*, ponownie wywołujemy metodę zwrotną interfejsu, tym razem niezawierającą komunikatu, i wywołujemy metodę `dismiss()`. Z kolei jeśli użytkownik wciśnie przycisk *Pomoc*, nie chcemy w rzeczywistości utracić fragmentu wyświetlającego okno dialogowe, więc przeprowadzamy nieco odmienną operację. Została ona wcześniej omówiona. Aby zapamiętać okno zachęty, do którego będzie można wrócić później, musimy utworzyć transakcję fragmentu usuwającą to okno i dodającą okno pomocy za pomocą metody `show()`; powinniśmy je umieścić w stosie drugoplanowym. Zwrócmy także uwagę na sposób utworzenia fragmentu wyświetlającego okno pomocy za pomocą odniesienia do identyfikatora zasobu. Oznacza to, że fragment ten może być użyty wraz z dowolnym tekstem umieszczonym w aplikacji.

## Przykładowe okno dialogowe — klasa `HelpDialogFragment`

Niebawem zaprezentujemy kod fragmentu wyświetlającego okno pomocy, najpierw jednak opiszemy mechanizm jego działania. Utworzyliśmy transakcję fragmentu powodującą przejście od fragmentu wyświetlającego okno zachęty do fragmentu przechowującego okno pomocy

i umieściliśmy ją na stosie drugoplanowym. W wyniku tego fragment wyświetlający okno zupy zniknie z pojemnika widoków, ciągle jest jednak dostępny w menedżerze fragmentów oraz z poziomu stosu drugoplanowego. Na jego miejscu pojawi się nowy fragment wyświetlający okno pomocy, w którym zawarto widoczny dla użytkownika tekst pomocy. W momencie wycofania omawianego fragmentu jego wpis zostanie usunięty ze stosu, w wyniku czego zniknie (zarówno przed oczu użytkownika, jak również z poziomu menedżera fragmentów) i zostanie przywrócony fragment wyświetlający okno zupy. Cała operacja jest w rzeczywistości banalna do przeprowadzenia. Kod zamieszczony na listingu 29.21 jest niezwykle prosty, a jednocześnie niesamowicie skuteczny; działa bezbłędnie nawet wtedy, gdy podczas wyświetlania okna dialogowego zmieni się tryb wyświetlania.

**Listing 29.21.** Układ graficzny i kod Java klasy HelpDialogFragment

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik /res/layout/help_dialog.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:padding="4dip"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/helpmessage"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:layout_marginLeft="20dip"
        android:layout_marginRight="20dip"
        android:text="Tekst pomocy"
        android:layout_weight="1"
        android:layout_gravity="center_vertical|center_horizontal"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:gravity="top|center_horizontal" />

    <Button android:id="@+id/btn_close"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0"
        android:text="Zamknij">
    </Button>

</LinearLayout>

// Jest to plik HelpDialogFragment.java
import android.app.DialogFragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.TextView;

public class HelpDialogFragment
    extends DialogFragment
    implements View.OnClickListener
{
```

```
public static HelpDialogFragment  
newInstance(int helpResId)  
{  
    HelpDialogFragment hdf = new HelpDialogFragment();  
    Bundle bundle = new Bundle();  
    bundle.putInt("zasób pomocy", helpResId);  
  
    hdf.setArguments(bundle);  
  
    return hdf;  
}  
  
@Override  
public void onCreate(Bundle icicle)  
{  
    super.onCreate(icicle);  
    this.setCancelable(true);  
    int style = DialogFragment.STYLE_NORMAL, theme = 0;  
    setStyle(style,theme);  
}  
  
public View onCreateView(LayoutInflater inflater,  
    ViewGroup container,  
    Bundle icicle)  
{  
    View v = inflater.inflate(R.layout.help_dialog, container,  
        false);  
  
    TextView tv = (TextView)v.findViewById(R.id.helpmessage);  
    tv.setText(getActivity().getResources()  
        .getText(getArguments().getInt("zasób pomocy")));  
  
    Button closeBtn = (Button)v.findViewById(R.id.btn_close);  
    closeBtn.setOnClickListener(this);  
    return v;  
}  
  
public void onClick(View v)  
{  
    dismiss();  
}
```

---

Mamy tu do czynienia z kolejnym fragmentem wyświetlającym okno dialogowe, jeszcze prostszym od prezentowanego wcześniej. Zadaniem tego fragmentu jest wyświetlanie tekstu pomocy. Na układ graficzny składa się kontrolka `TextView` i przycisk *Zamknij*. Kod Java powinien wyglądać już znajomo dla Czytelnika. Znajdziemy w nim metody `newInstance()` służącą do utworzenia nowego fragmentu, który wyświetla okno pomocy, `onCreate()` pozwalającą na zdefiniowanie stylu i motywu okna dialogowego, a także `onCreateView()` generującą hierarchię widoków. W naszym przypadku potrzebny jest nam zasób typu `string`, którym posłużymy się do zapełnienia kontrolki `TextView`, zatem uzyskujemy dostęp do zasobów z poziomu aktywności i wybieramy identyfikator zasobu przekazany w metodzie `newInstance()`. Na końcu metoda `onCreateView()` ustanawia procedurę obsługi kliknięcia przycisku *Zamknij*. W tym przypadku w czasie zamykania okna system nie wykonuje niczego nadzwyczajnego.

Istnieją dwa sposoby wywoływania tego fragmentu: z poziomu aktywności oraz poprzez fragment wyświetlający okno zachęty. Jeżeli wybierzemy pierwsze rozwiązanie, późniejsze wycofanie tego fragmentu poskutkuje jego usunięciem ze szczytu stosu i wyświetleniem głównej aktywności znajdującej się pod spodem. Jeżeli wyświetlimy ten fragment z poziomu fragmentu wyświetlającego okno zachęty, jego wycofanie spowoduje wycofanie transakcji fragmentu (ponieważ fragment ten był częścią transakcji umieszczonej w stosie drugoplanowym) i usunięcie okna pomocy wraz z jednoczesnym przywróceniem fragmentu wyświetlającego okno zachęty. W efekcie użytkownik ponownie ujrzy okno zachęty.

## Przykładowe okno dialogowe — klasa AlertDialogFragment

Pozostał nam do zaprezentowania ostatni fragment wyświetlający okno dialogowe, mianowicie okno dialogowe alertu. Chociaż możemy utworzyć go podobnie jak w przypadku wcześniej zaprezentowanego fragmentu wyświetlającego okno pomocy, istnieje alternatywne rozwiązanie, pozwalające na wykorzystanie starszej struktury klasy AlertDialogBuilder. Rozwiązanie to okazywało się skuteczne w wielu poprzednich wersjach systemu Android. Listing 29.22 zawiera kod źródłowy fragmentu wyświetlającego okno alertu.

**Listing 29.22.** Kod Java klasy AlertDialogFragment

```
import android.app.AlertDialog;
import android.app.Dialog;
import android.app.DialogFragment;
import android.content.DialogInterface;
import android.os.Bundle;

public class AlertDialogFragment
    extends DialogFragment
    implements DialogInterface.OnClickListener
{
    public static AlertDialogFragment
    newInstance(String message)
    {
        AlertDialogFragment adf = new AlertDialogFragment();
        Bundle bundle = new Bundle();
        bundle.putString("komunikat alertu",message);
        adf.setArguments(bundle);

        return adf;
    }

    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        this.setCancelable(true);
        int style = DialogFragment.STYLE_NORMAL, theme = 0;
        setStyle(style,theme);
    }

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState)
    {
        AlertDialog.Builder b =

```

```
new AlertDialog.Builder(getActivity());
b.setTitle("Uwaga!!!");
b.setPositiveButton("OK", this);
b.setNegativeButton("Anuluj", this);
b.setMessage(this.getArguments().getString("komunikat alertu"));
return b.create();
}

public void onClick(DialogInterface dialog, int which)
{
    OnDialogDoneListener act = (OnDialogDoneListener) getActivity();
    boolean cancelled = false;
    if (which == AlertDialog.BUTTON_NEGATIVE)
    {
        cancelled = true;
    }
    act.onDialogDone(getTag(), cancelled, "Alert odwołany");
}
}
```

---

W przypadku tego fragmentu nie potrzebujemy układu graficznego, ponieważ klasa `AlertDialog` zapewnia jego utworzenie. Czytelnik zauważa, że ten fragment jest tworzony tak jak pozostałe, jednak zamiast metody zwrotnej `onCreateView()` stosujemy w tym przypadku metodę `onCreateDialog()`. Implementujemy albo metodę `onCreateView()`, albo `onCreateDialog()`, nigdy obydwie naraz. Element przekazywany przez metodę `onCreateDialog()` nie jest widokiem, lecz oknem dialogowym. Od tego miejsca możemy zacząć wykorzystywać informacje przedstawione w rozdziale 8., aby utworzyć okno dialogowe w standardowy sposób. Różnica polega na tym, że w celu uzyskania dostępu do parametrów okna dialogowego powinniśmy wziąć pod uwagę pakiet parametrów. W naszej przykładowej aplikacji wykorzystujemy go do zaprezentowania komunikatu alertu, nie ma jednak przeszkód, aby uzyskać dostęp do innych parametrów pakietu.

Zwróćmy także uwagę, że w przypadku tego typu fragmentu wyświetlającego okno dialogowe wymagana jest implementacja interfejsu `DialogInterface.OnClickListener` w klasie fragmentu, co oznacza, że nasz fragment musi implementować metodę zwrotną `onClick()`. Metoda ta zostanie wywołana, jeżeli użytkownik zacznie w jakiś sposób wpływać na okno dialogowe. Po raz wtóry uzyskujemy odniesienie do uruchomionego okna dialogowego oraz wskazanie, który przycisk został wciśnięty. Podobnie jak poprzednio, nie możemy polegać na metodzie `onDismiss()`, ponieważ może ona zostać wywołana wskutek zmiany konfiguracji urządzenia.

## Przykładowe okno dialogowe — główny układ graficzny

Aby nasza aplikacja była kompletna, musimy wstawić również układ graficzny głównej aktywności. Odpowiedni kod został zaprezentowany na listingu 29.23.

**Listing 29.23.** Główny układ graficzny

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/main.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:gravity="fill"
    >
<TextView android:id="@+id/textViewId"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/white"
    android:text="@string/help_text"
    android:textColor="@android:color/black"
    android:textSize="25sp"
    android:scrollbars="vertical"
    android:scrollbarStyle="insideOverlay"
    android:scrollbarSize="25dip"
    android:scrollbarFadeDuration="0"
    />
</LinearLayout>
```

Po uruchomieniu aplikacji należy przetestować wszystkie opcje w różnorodnych trybach wyświetlania obrazu. Spróbujmy obrócić urządzenie w momencie wyświetlania fragmentów. Czytelnikowi powinno się spodobać, że okna dialogowe obracają się wraz z resztą obrazu oraz że nie trzeba zbytnio przejmować się kodem odpowiedzialnym za zachowywanie i odczytywanie fragmentów w trakcie zmian konfiguracji.

Mamy nadzieję, że kolejnym elementem, jaki Czytelnik doceni, jest łatwość nawiązywania komunikacji pomiędzy fragmentami a aktywnością. Oczywiście, aktywność zawiera odniesienia (lub może je uzyskać) do wszystkich istniejących fragmentów, posiada więc możliwość uzyskania dostępu do metod eksponowanych przez te fragmenty. Nie jest to jedyny sposób komunikowania się fragmentów z aktywnością. Możemy zawsze zastosować metody pobierające wobec menedżera fragmentów w celu odczytania wystąpienia zarządzanego fragmentu, a następnie odpowiednio rzutować takie odniesienie i bezpośrednio wywołać metodę wobec tego fragmentu. Stopień odizolowania fragmentów od siebie za pomocą interfejsów oraz poprzez aktywności lub utworzenia sieci zależności pomiędzy fragmentami zależy od złożoności aplikacji oraz stopnia jej wykorzystania.

## Inne formy komunikowania się z fragmentami

Zademonstrowaliśmy elegancki sposób komunikowania się fragmentów pomiędzy sobą, polegający na definiowaniu i wykorzystywaniu interfejsu służącego do implementacji metod zwrotnych z fragmentów wprost w aktywności wywołującej. Nie jest to jedyny mechanizm komunikowania się fragmentów ze sobą. Ponieważ menedżer fragmentów ma dostęp do informacji na temat wszystkich fragmentów dołączonych do bieżącej aktywności, aktywność ta lub zawarty w niej fragment mogą zażądać takich informacji dotyczących dowolnego innego fragmentu przy użyciu uprzednio opisanych metod pobierających.

Po uzyskaniu odniesienia do fragmentu aktywność lub jej fragment mogą go w odpowiedni sposób rzutować, a następnie spowodować bezpośrednie wywołanie metod wobec tej aktywności lub jej fragmentu. W ten sposób fragmenty mogą uzyskać więcej informacji na temat innych fragmentów, niż byłoby to wymagane w zwykłych przypadkach. Nie należy jednak zapominać, że w tym przypadku aplikacja jest uruchomiona na urządzeniu mobilnym, zatem niekiedy zastosowanie pewnych uproszczeń jest uzasadnione. Wycinek kodu z listingu 29.24 ukazuje nam bezpośredni sposób komunikacji pomiędzy dwoma fragmentami.

**Listing 29.24.** Bezpośrednia komunikacja pomiędzy fragmentami

```
FragmentOther fragOther =  
    (FragmentOther)getFragmentManager().findFragmentByTag("other");  
fragOther.callCustomMethod( arg1, arg2 );
```

---

Na listingu 29.24 nie wprowadziliśmy żadnego interfejsu. Omawiany fragment bezpośrednio posiada informacje na temat klasy oraz dostępnych metod drugiego fragmentu. Takie rozwiązanie nie jest kłopotliwe, ponieważ fragmenty te mogą być częścią tej samej aplikacji. Poza tym fakt, że niektóre fragmenty mają dostęp do informacji na temat innych fragmentów, może być łatwiejszy do zaakceptowania.

## **Stosowanie metod startActivity() i setTargetFragment()**

Wspólną cechą fragmentów i aktywności jest możliwość uruchomienia aktywności za pomocą fragmentu. Fragment zawiera metody `startActivity()` oraz `startActivityForResult()`. Działają one podobnie jak w przypadku aktywności: w momencie przekazania wyniku zostanie wywołana metoda `onActivityResult()` wobec fragmentu uruchamiającego daną aktywność.

Istnieje jeszcze jeden mechanizm komunikacji, z którym Czytelnik powinien się zapoznać. Gdy dany fragment ma zostać uruchomiony przez inny fragment, wywołujący fragment może zostać powiązany z fragmentem wywoływanym. Zostało to zademonstrowane na listingu 29.25.

**Listing 29.25.** Konfiguracja mechanizmu komunikacji fragmentu wywołującego z docelowym fragmentem

```
mCalledFragment = new CalledFragment();  
mCalledFragment.setTargetFragment(this, 0);  
fm.beginTransaction().add(mCalledFragment, "work").commit();
```

---

Za pomocą tych kilku wierszy utworzyliśmy nowy obiekt `CalledFragment`, ustawiliśmy w bieżącym fragmencie wywoływany fragment jako fragment docelowy oraz za pomocą mechanizmu transakcji dodaliśmy ten wywoływany fragment do menedżera fragmentów oraz do aktywności. Po uruchomieniu wywoływanego fragmentu będzie mógł on wywołać metodę `getTargetFragment()`, która przekaże odniesienie do fragmentu wywołującego. Za pomocą tej metody wywoływany fragment może korzystać z metod fragmentu wywołującego, a nawet uzyskać bezpośredni dostęp do jego składowych widoku. Na listingu 29.26 zademonstrowaliśmy przykładowy kod, w którym wywoływany fragment wprowadza tekst bezpośrednio do interfejsu użytkownika znajdującego się we fragmencie wywołującym.

**Listing 29.26.** Komunikacja fragmentu docelowego z fragmentem wywołującym

```
TextView tv = (TextView)  
getTargetFragment().getView().findViewById(R.id.text1);  
tv.setText("Ustawiony z poziomu wywoływanego fragmentu");
```

---

## Tworzenie niestandardowych animacji za pomocą klasy ObjectAnimator

We wcześniejszej części rozdziału zaprezentowaliśmy w zarysie sposób niestandardowego animowania fragmentów. Pokazaliśmy kod niestandardowej animacji, dzięki której fragment wyświetlający szczegóły zniknął, a na jego miejscu pojawił się nowy fragment wyświetlający szczegóły. Stwierdziliśmy również, że w pakiecie Android SDK do dyspozycji pozostaje tylko kilka standardowych animacji, jednak nie wszystkie działają zgodnie z oczekiwaniami. W tym podrozdziale pokażemy, w jaki sposób można tworzyć własne, niestandardowe animacje, dzięki czemu Czytelnik będzie mógł wstawiać interesujące przejścia pomiędzy fragmentami.

Mechanizm implementowania niestandardowych animacji fragmentów został umieszczony w klasie `ObjectAnimator`. W rzeczywistości jest to ogólna funkcja Androida, która może być stosowana nie tylko w przypadku fragmentów, lecz również obiektów klasy `View`. W tym podrozdziale będziemy zajmować się wyłącznie animacją fragmentów, lecz omawiane tu zasady w równym stopniu stosuje się również do innych rodzajów obiektów. Animator obiektu pobiera obiekt i animuje go od stanu początkowego do stanu końcowego w określonym przedziale czasowym. Przedział ten jest definiowany w milisekundach. Istnieją procedury określające zachowanie obiektu w tym czasie. Procedury te noszą nazwę interpolatorów.

Jeżeli wyobrażymy sobie przejście od stanu początkowego do stanu końcowego jako prostą, interpolator będzie wyznaczał „położenie” animacji na tej prostej w dowolnym momencie tego danego czasu. Jedną z najprostszych procedur jest interpolator liniowy (ang. *linear interpolator*); dzieli on nasz odcinek na równe części i „przeskakuje” po nich w takich samych przedziałach czasowych. W wyniku tego obiekt porusza się z punktu początkowego do punktu końcowego ze stałą prędkością, bez początkowego przyśpieszenia i końcowego hamowania.

Domyślnym interpolatorem jest `accelerate_decelerate`, który wprowadza na początku animacji płynne przyśpieszenie, a na jej końcu — płynne hamowanie. Najciekawsza jest informacja, że interpolator może przekroczyć punkt końcowy na naszej prostej, a następnie cofnąć się. Na tym polega działanie interpolatora przekraczającego (ang. *overshoot interpolator*). Mamy również do czynienia z interpolatorem drgającym (ang. *bounce interpolator*), który porusza się z punktu początkowego do punktu końcowego, jednak po dotarciu do punktu końcowego wraca kilkakrotnie do punktu początkowego, aby ostatecznie zatrzymać się w punkcie końcowym.

Interpolator wpływa na wymiary obiektu. W przypadku stosowanych wcześniej animatorów `fade_in` oraz `fade_out`, tym wymiarem był parametr alfa fragmentu (tj. przezroczystość obiektu). Animator `fade_in` zmieniał wartość parametru alfa fragmentu od wartości (0) do (1), z kolei animator `fade_out` modyfikował wartość parametru alfa drugiego fragmentu od wartości (1) do (0). Jeden fragment przechodził ze stanu całkowitej przezroczystości do stanu zupełniej widoczności, podczas gdy w drugim fragmencie następował odwrotny proces.

Poza wzrokiem użytkownika animator obiektu znajduje główny widok fragmentu i regularnie wywołuje metodę `setAlpha()`, za każdym razem nieznacznie zmieniając wartość parametru. Częstotliwość powtórzeń wywołań zależy od interpolatora. Interpolator liniowy wprowadza wywołania w równych odstępach czasowych. Interpolator `accelerate_decelerate` najpierw ustanawia mniejsze wartości określonego parametru na jednostkę czasu i stopniowo je zwiększa, co daje wrażenie przyśpieszenia, natomiast pod koniec odwraca ten proces, przez co wydaje się, że następuje spowolnienie animacji danego wymiaru obiektu.

Wymiarami może być wiele spośród wartości, które można pobrać i ustawić wewnątrz klasy View. W rzeczywistości do pracy z przetwarzanym widokiem animator obiektu wykorzystuje mechanizm refleksji. Jeżeli zechcemy zdefiniować animację obrotu, Android wywoła metodę `setRotation()` wobec danego obiektu (lub jego widoku). Animator pobiera wartości początkową i końcową, a następnie wykorzystuje je do przetworzenia obiektu w danych granicach. Jeżeli nie zostanie zdefiniowana wartość początkowa, zostanie wprowadzona metoda pobierająca, mająca na celu uzyskanie bieżącej wartości obiektu. Zobaczmy, jak to się ma do naszych fragmentów.

Jedyną metodą w klasie `FragmentTransaction` definiującą niestandardową animację jest `setCustomAnimations()`, która pobiera dwa parametry — identyfikatory zasobów:

- Pierwszy parametr określa zasób animatora dla fragmentu wprowadzanego do pojemnika widoków.
- Drugi parametr definiuje zasób animatora dla fragmentu opuszczającego pojemnik widoków.

Obydwa animatory nie muszą być ze sobą nawet powiązane, najlepiej jednak by było, gdyby wizualnie do siebie pasowały. Innymi słowy, jeżeli jeden fragment zanika, drugi mógłby stopniowo pojawić się na miejscu poprzednika. Jeżeli jeden fragment wysuwa się z prawego brzegu ekranu, drugi może wsunąć się z lewej strony.

Zasoby animatora można znaleźć w folderze zestawu SDK, w katalogu związanym z właściwą platformą, a dalej w podkatalogu `/data/res/Animator`. To właśnie tu znajdziemy używane wcześniej pliki `fade_in.xml` i `fade_out.xml`. Możemy utworzyć również własne zasoby. Jeżeli zdecydujemy się na ten krok, najlepiej umieścić taki plik w katalogu `/res/Animator` naszego projektu. Plik ten w razie potrzeby można dodać ręcznie. Przyjrzymy się przykładowi umieszczonemu na listingu 29.27, gdzie widzimy prosty lokalny plik animatora (`slide_in_left.xml`).

---

**Listing 29.27.** Niestandardowy animator powodujący wysuwanie się obiektu z lewej strony ekranu

---

```
<?xml version="1.0" encoding="utf-8" ?>
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:interpolator="@+id/accelerate_decelerate"
    android:valueFrom="-1280"
    android:valueTo="0"
    android:valueType="floatType"
    android:propertyName="x"
    android:duration="2000" />
```

---

W tym pliku zasobów wykorzystano nowy znacznik, `objectAnimator`, wprowadzony w wersji 3.0 Androida. Podstawowa struktura pliku powinna jednak wyglądać znajomo dla Czytelnika. Mamy tu do czynienia z grupą atrybutów typu `android:` określających czynność, jaką chcemy wykonać. W przypadku animatora obiektów musimy zdefiniować kilka elementów. Pierwszym z nich jest interpolator. Lista dostępnych typów interpolatorów została umieszczona w zasobie `android.R.interpolator`. Dzięki wiedzy o nazwach zasobów zorientujemy się, że atrybut interpolatora stanowi odpowiednik pliku umieszczonego w katalogu zestawu SDK, w folderze właściwej platformy, dokładniej zaś w katalogu `/data/res/interpolator`, a nazwa tego pliku to `accelerate_decelerate.xml`.

Atrybut `android:propertyName` definiuje wymiar, jaki stanie się podstawą animacji. W tym przypadku zamierzamy przeprowadzić animację w kierunku osi X. Jeżeli przyjrzymy się metodzie `setX()` klasy `View`, zauważymy, że wprowadzonym parametrem jest wartość zmienno-

przecinkowa. Z tego właśnie powodu atrybut android:valueType posiada zdefiniowaną wartość floatType. Wartość atrybutu android:duration wynosi 2000, czyli 2 sekundy. Prawdopodobnie jest to zbyt długi czas dla standardowej aplikacji, w tym przypadku jednak chcemy zdążyć zobaczyć, co się dzieje w trakcie animacji. Dwa niewymienione jeszcze atrybuty, android:→valueFrom oraz android:valueTo, posiadają wartości odpowiednio: -1280 i 0. Zostały one określone w taki sposób, ponieważ dany fragment ma się znajdować w punkcie 0 na końcu animacji. Oznacza to, że po zakończeniu animacji lewa krawędź fragmentu będzie się znajdować przy lewej krawędzi pojemnika widoku. Ponieważ chcemy, aby nasz fragment wysunął się z lewej strony ekranu, musimy zadeklarować rozpoczęcie animacji od tej strony, a wartość -1280 wydaje się wystarczająco duża. Jak zapewne Czytelnik się domyśla, animator obiektu wyauważającego się z prawej strony wyglądałby niemal identycznie jak ten zaprezentowany na listingu 29.27, z tą różnicą, że atrybut android:valueFrom przyjąłby wartość 0, a w atrybucie android:valueTo wprowadzilibyśmy bardzo dużą wartość dodatnią, na przykład 1280.

Podczas korzystania z animatora obiektów zauważymy, że wartości większości wymiarów posiadają typ floatType, chociaż czasami będziemy wybierać również typ intType. Wystarczy spojrzeć na typ wartości parametru wymaganego przez metodę ustawiającą. To właśnie dzięki niej animator obiektu posiada tak wielki potencjał. Tak naprawdę nie jest ważne, skąd pochodzi metoda ustawiająca. Oznacza to, że możemy dodać do obiektu własny wymiar, a jego animację obsłuży klasa animatora. Zadaniem programisty jest jedynie dostarczenie metody ustawiającej i wprowadzenie atrybutów do pliku zasobu. Resztę pracy wykona animator. Jedyną wadą tego systemu jest to, że w przypadku pominięcia atrybutu valueFrom w pliku XML animator obiektu wykorzysta metodę pobierającą do określenia wartości początkowej obiektu. Metoda pobierająca musi wtedy przekazać właściwy typ wartości danego wymiaru.

Być może Czytelnika zainteresuje również możliwość jednoczesnego animowania kilku wymiarów. W tym celu wykorzystujemy znacznik <set> wokół większej liczby znaczników <object →Animator>. Na listingu 29.28 przedstawiliśmy plik zasobu animatora (*slide\_out\_down.xml*), który przeprowadza animację wzdłuż osi Y i jednocześnie modyfikuje parametr alfa obiektu.

**Listing 29.28.** Niestandardowy animator przeprowadzający animację w osi X oraz w wymiarze alfa

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
<objectAnimator
    android:interpolator="@android:interpolator/accelerate_cubic"
    android:valueFrom="0"
    android:valueTo="1280"
    android:valueType="floatType"
    android:propertyName="y"
    android:duration="2000" />
<objectAnimator
    android:interpolator="@android:interpolator/accelerate_cubic"
    android:valueFrom="1"
    android:valueTo="0"
    android:valueType="floatType"
    android:propertyName="alpha"
    android:duration="2000" />
</set>
```

Znacznik `<set>` jest odpowiednikiem klasy `ObjectAnimator` w Androidzie, jednak w strukturze XML znacznik ten posiada tylko jeden atrybut — `android:ordering`. Dopuszczalnymi wartościami tego atrybutu są domyślna `together`, umożliwiająca równolegle animowanie wymiarów, oraz `sequential`, definiująca kolejność występowania animacji zgodnie z kolejnością ich umieszczenia w pliku XML.

## Odbońniki

Poniżej zamieszczamy odnośniki do zasobów zawierających dodatkowe informacje na tematy przedstawione w tym rozdziale:

- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — znajdziemy tu listę wszystkich projektów utworzonych na potrzeby niniejszej książki. Właściwe pliki są umieszczone w katalogu o nazwie `ProAndroid3_R29_Fragmenty`. Zamieściliśmy w nim także plik `Czytaj.TXT`, stanowiący dokładną instrukcję importowania projektów do środowiska Eclipse. Dołączliśmy również projekty ukazujące zastosowanie pakietu kompatybilności fragmentów (*Fragment Compatibility SDK*) w starszych wersjach systemu Android.
- `ApiDemos` — wśród przykładów umieszczonych w zestawie Android SDK znajdziemy projekt `ApiDemos`. Umieszczono w nim kilka aplikacji wykorzystujących koncepcję fragmentów, które mogą nam pomóc w jej zrozumieniu.
- <http://developer.android.com/guide/topics/fundamentals/fragments.html> — artykuł z poradnika programisty, w którym omówiono koncepcję fragmentów.
- <http://android-developers.blogspot.com/2011/02/android-30-fragments-api.html> — wpis stanowiący wstęp do nauki posługiwania się fragmentami.
- <http://android-developers.blogspot.com/2011/02/animation-in-honeycomb.html> — wpis wprowadzający do nowej struktury animacji oraz do zastosowań animatora obiektów.

## Podsumowanie

W niniejszym rozdziale zaprezentowaliśmy zupełnie nową klasę `Fragment`, wprowadzoną w wersji 3.0 Androida, a także jej podklasy i klasy pokrewne, związane z menedżerem oraz transakcjami. Fragmenty stanowią zupełnie nowe, potężne rozwiązanie, pozwalające na organizację funkcjonalności oraz powiązanych z nią interfejsów użytkownika. Chociaż fragmenty zostały zaprojektowane specjalnie dla tabletów, dostępne będą również w przypadku urządzeń wyposażonych w niewielkie wyświetlacze. Będą pomocne w rozdzielaniu logiki działania aplikacji na niewielkie, proste w użytkowaniu składowe, które będą wykorzystywane, przenoszone i zarządzane na wcześniej niedostępne sposoby. Przedstawiliśmy jedną z ciekawszych nowych funkcji Androida — mowa tu o animatorze obiektów, który w prosty sposób może zmodyfikować przejścia pomiędzy fragmentami.

W następnym rozdziale zajmiemy się kolejnym istotnym aspektem aplikacji tworzonych z myślą o tabletach, czyli klasą `ActionBar`.

## Analiza klasy ActionBar

Klasa `ActionBar` jest nowym interfejsem API wprowadzonym w wersji 3.0 Androida. Pozwala ona na modyfikowanie paska tytułowego aktywności. W starszych wersjach systemu pasek tytułowy aktywności przechowywał wyłącznie jej nazwę.

Wraz z rozwojem systemu Android przejmuje on coraz więcej wzorców interfejsu użytkownika wykorzystywanych w komputerach biurowych. W aplikacji biurowej mamy do czynienia z paskiem tytułowym, paskiem menu oraz z pewną liczbą przycisków na pasku narzędzi. Implementacja interfejsu `ActionBar` stanowi odzwierciedlenie takiej struktury paska tytułowego i menu, spotykanej w komputerach stacjonarnych.

Interfejs `ActionBar` opiera się zwłaszcza na modelu paska tytułowego i paska menu spotykanego w przeglądarkach WWW. Został on zaprojektowany w taki sposób, aby programiści mogli uwzględniać w aplikacjach schematy nawigacji znane właśnie z przeglądarek.

Kluczowym zadaniem paska narzędzi jest zagwarantowanie użytkownikowi błyskawicznego dostępu do najczęściej wykorzystywanych narzędzi bez konieczności przeszukiwania menu opcji lub menu kontekstowych.

**Uwaga!**

We współczesnej literaturze informatycznej dogodny dostęp do działań zwany jest często afordancją, co odnosi się do wygodnego odkrywania czy też wykonywania działań. Na końcu rozdziału zamieściliśmy kilka adresów URL kierujących do informacji na temat afordancji.

W trakcie lektury tego rozdziału Czytelnik zapozna się z następującymi informacjami o pasku działania:

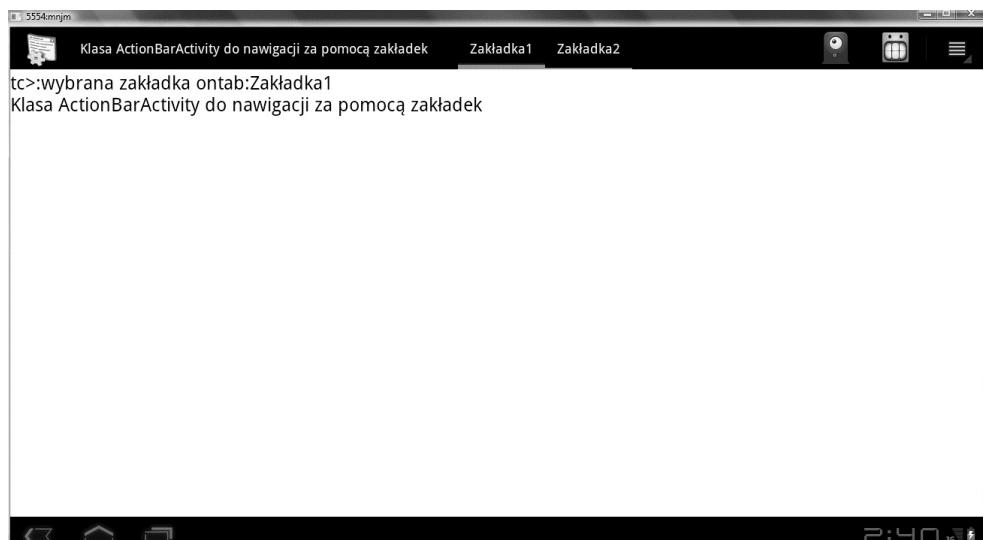
- Pasek działania jest częścią aktywności i zachowuje się zgodnie z jej cyklem życia.
- Pasek działania przyjmuje jedną z trzech postaci: paska zakładek, paska wyświetlającego listę oraz standardowego paska działania. W dalszej części rozdziału zaprezentujemy wygląd i zachowanie każdego z tych trybów.
- Sposób, w jaki obiekty nasłuchujące zakładki umożliwiają interakcję z paskiem zakładek.

- Sposób, w jaki adaptery obiektów typu Spinner oraz obiekty nasłuchujące listy są wykorzystywane do współdziałania z paskiem wyświetlającym listy.
- Mechanizm oddziaływanego przycisku ekranu startowego oraz paska działania na strukturę menu.
- Sposób umieszczania przycisków wewnętrz paska stanu oraz ich obsługiwania.

Zademonstrujemy te pojęcia poprzez utworzenie trzech różnych aktywności. Każda z nich połuży do zaprezentowania innego rodzaju paska działania. W ten sposób Czytelnik zyska możliwość przetestowania zachowania tej klasy w każdym z trzech trybów. Najpierw przyjrzymy się jednak elementom paska działania widocznym dla użytkownika.

## Anatomia klasy ActionBar

Na rysunku 30.1 został zaprezentowany typowy pasek działania, uruchomiony w trybie wyświetlanego zakładek.



**Rysunek 30.1.** Aktywność z paskiem zakładek

Jest to zrzut ekranu z działającej przykładowej aplikacji, która została omówiona w dalszej części tego rozdziału. Pasek działania widoczny na rysunku 30.1 składa się z pięciu części. Są to (licząc od lewej do prawej):

- **Obszar przycisku ekranu startowego.** Przycisk widoczny w lewej górnej części paska działania jest czasami nazywany przyciskiem ekranu startowego (ang. *home icon*). Przypomina on nieco kontekst nawigacji z przeglądarek WWW, w których kliknięcie przycisku strony startowej spowoduje jej uruchomienie. Przekonamy się później, że kliknięcie tego przycisku spowoduje wysłanie metody zwrotnej do opcji menu posiadającej identyfikator android.R.id.home.
- **Obszar tytułu.** W tym obszarze umieszczamy tytuł aplikacji bądź aktywności.

- **Obszar zakładek.** W obszarze zakładek znajduje się lista zdefiniowanych zakładek. Zawartość tego obszaru może być bardzo różnorodna. Jeżeli pasek działania pracuje w trybie wyświetlania zakładek, będą tu umieszczane zakładki. Jeśli natomiast zdefiniowano tryb wyświetlania listy, znajdziemy tutaj listę rozwijalnych elementów. W standardowym trybie obszar ten pozostaje pusty.
- **Obszar paska narzędzi.** Tuż za obszarem zakładek znajduje się obszar paska narzędzi, zawierający niektóre elementy menu dostępne w postaci przycisków. W przykładowej aplikacji zaprezentujemy, w jaki sposób można tutaj umieszczać wybrane opcje menu.
- **Obszar menu.** To ostatni obszar paska działania. Jest to pojedynczy, standardowy przycisk wyświetlający opcje menu. Po jego kliknięciu zostanie rozwinięte menu. Menu to może różnie wyglądać lub wyświetlać się w różnych miejscach, zależnie od rozmiaru urządzenia.

Aktywność z rysunku 30.1 oprócz paska działania zawiera również widok tekstowy debugowania, w którym są wyświetlane informacje dotyczące wykonanych działań. Działania te mogą być wynikiem kliknięcia którejś z zakładek, przycisku ekranu startowego, opcji menu działania lub opcji właściwego menu.

Zastanówmy się teraz nad sposobem implementacji trzech wymienionych rodzajów paska działania: paska zakładek, paska wyświetlającego listę oraz standardowego paska działania. Skoro zaprezentowaliśmy rzut ekranu aplikacji z paskiem działania w trybie paska zakładek, zajmijmy się najpierw tym przypadkiem.

## Aktywność paska działania wyświetlającego zakładki

Chociaż zamierzamy utworzyć trzy różne aktywności, z których każda przechowuje inny rodzaj paska działania, będą one posiadały wiele wspólnych cech.

- Wszystkie zawierają ten sam widok debugowania, dzięki czemu możemy monitorować wykonywane przez nie działania.
- Wszystkie posiadają wspólny przycisk strony startowej.
- Każda aktywność posiada tytuł.
- Wszystkie posiadają te same przyciski w obszarze paska narzędzi.
- Wszystkie posiadają wspólne menu opcji.

Podstawowa różnica między trzema omawianymi trybami paska działania leży w sposobie jego konfiguracji. W naszej przykładowej aplikacji umieścimy wspólny kod w klasie bazowej i pozwolimy każdej pochodnej aktywności (w tym również aktywności paska zakładek) na skonfigurowanie paska działania.

Dość trudno jest objąść działanie tych współdzielonych plików bez kontekstu przynajmniej jednej aktywności zawierającej pasek działania. Zaprezentujemy więc najpierw w pierwszym punkcie te wspólne pliki oraz sposób ich wykorzystania przez aktywność paska zakładek, a następnie dodamy do tego projektu aktywności przechowujące pozostałe dwa rodzaje pasków menu.

Poniżej zamieściliśmy listę plików, które będą potrzebne do utworzenia tego ćwiczeniowego projektu. Wymieniliśmy tutaj wszystkie wspólne pliki oraz pliki wymagane do wprowadzenia paska zakładek. Widzimy na tej liście wiele plików, ponieważ umieszczamy wspólny kod w klasach bazowych. W ten sposób zmniejszymy liczbę plików dodawanych podczas tworzenia dwóch kolejnych przykładowych aplikacji. Przy nazwach poszczególnych plików podano również numery odpowiednich listingów.

- *DebugActivity.java* — klasa bazowa aktywności generująca widok debugowania zaprezentowany na rysunku 30.1 (listing 30.2).
- *BaseActionBarActivity.java* — wywodzi się z klasy *DebugActivity* i zawiera wspólną logikę nawigacji (na przykład odpowiedzi na współdzielone działania, w tym zmiany pomiędzy trybami wyświetlania paska działania; listing 30.3).
- *IReportBack.java* — interfejs stanowiący nośnik komunikacyjny pomiędzy aktywnością debugowania a różnymi obiektami nasłuchującymi paski działania (listing 30.1).
- *BaseListener.java* — bazowa klasa obiektu nasłuchującego, współpracująca z klasą *DebugActivity* oraz różnorodnymi działaniami wywoływanymi z poziomu paska działania. Pełni rolę bazowej klasy dla obiektów nasłuchujących paska zakładek oraz paska wyświetlającego listę (listing 30.4).
- *TabNavigationActionBarActivity.java* — wywodzi się z klasy *BaseActionBarActivity* i definiuje pasek zakładek. Została tutaj umieszczona większość kodu odnoszącej się do paska zakładek (listing 30.6).
- *TabListener.java* — klasa ta jest potrzebna do dodawania zakładki do paska zakładek. Tutaj również generowane są odpowiedzi na kliknięcia zakładek. W naszym przypadku wyświetlany jest po prostu komunikat debugowania poprzez klasę *BaseListener* (listing 30.5).
- *AndroidManifest.xml* — zostają tu zdefiniowane wywoływane aktywności (listing 30.13).
- *layout/main.xml* — plik układu graficznego aktywności *DebugActivity*. Wszystkie trzy klasy paska działań wywodzą się z klasy *DebugActivity*, więc współdzielą one również ten układ graficzny (listing 30.7).
- *menu/menu.xml* — zestaw elementów menu pozwalający na przetestowanie interakcji z paskiem działania. Również ten plik jest współdzielony pomiędzy wszystkimi aktywnościami paska działania (listing 30.9).

## Implementacja bazowych klas aktywności

Wiele bazowych klas wykorzystuje interfejs *IReportBack*. Przedstawiliśmy go już w poprzednich rozdziałach. Jego przeznaczenie nie ulega tu zmianie. Prezentujemy go ponownie na liście 30.1, aby Czytelnik nie musiał go szukać w poprzednich rozdziałach.

---

### Listing 30.1. *IReportBack.java*

---

```
//IReportBack.java
package com.androidbook.actionbar;

public interface IReportBack
{
    public void reportBack(String tag, String message);
    public void reportTransient(String tag, String message);
}
```

---

Klasa implementująca ten interfejs pobiera komunikat i wyświetla go na ekranie. Może to być na przykład komunikat debugowania. Dokonujemy tego za pomocą metody *reportBack()*. Taką samą rolę pełni metoda *reportTransient()*, w tym przypadku jednak użytkownik widzi komunikat wyświetlony w kontrolce *Toast*.

W naszym przykładzie klasą implementującą interfejs IReportBack jest DebugActivity. Kod źródłowy tej aktywności został umieszczony na listingu 30.2.

#### **Listing 30.2.** Klasa DebugActivity zawierająca widok debugowania

---

```
//DebugActivity.java
package com.androidbook.actionbar;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public abstract class DebugActivity
    extends Activity
    implements IReportBack
{
    //Potrzebuję tego najpierw pochodne klasy
    protected abstract boolean
        onMenuItemSelected(MenuItem item);

    //Zmienne prywatne, ustanowione przez konstruktor
    private static String tag=null;
    private int menuId = 0;
    private int layoutid = 0;
    private int debugTextviewId = 0;

    public DebugActivity(int inMenuId,
        int inLayoutId,
        int inDebugTextviewId,
        String inTag)
    {
        tag = inTag;
        menuId = inMenuId;
        layoutid = inLayoutId;
        debugTextviewId = inDebugTextviewId;
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(this.layoutid);

        //Poniższy fragment jest potrzebny do przewijania
        //widoku tekstowego
        TextView tv = this.getTextView();
        tv.setMovementMethod(
            ScrollingMovementMethod.getInstance());
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu){
        super.onCreateOptionsMenu(menu);
        MenuInflater inflater = getMenuInflater();
        inflater.inflate(menuId, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item){
        appendMenuItemText(item);
    }
```

```
if (item.getItemId() == R.id.menu_da_clear){
    this.emptyText();
    return true;
}
boolean b = onMenuItemSelected(item);
if (b == true)
{
    return true;
}
return super.onOptionsItemSelected(item);
}
protected TextView getTextView(){
    return
        (TextView)this.findViewById(this.debugTextViewId);
}
protected void appendMenuItemText(MenuItem menuItem){
    String title = menuItem.getTitle().toString();
    appendText("MenuItem:" + title);
}
protected void emptyText(){
    TextView tv = getTextView();
    tv.setText("");
}
protected void appendText(String s){
    TextView tv = getTextView();
    tv.setText(s + "\n" + tv.getText());
    Log.d(tag,s);
}
public void reportBack(String tag, String message)
{
    this.appendText(tag + ":" + message);
    Log.d(tag,message);
}
public void reportTransient(String tag, String message)
{
    String s = tag + ":" + message;
    Toast mToast =
        Toast.makeText(this, s, Toast.LENGTH_SHORT);
    mToast.show();
    reportBack(tag,message);
    Log.d(tag,message);
}
}//eof-class
```

---

Podstawowym zadaniem tej bazowej klasy jest wyświetlenie aktywności zawierającej widok debugowania. Widok ten służy do prezentowania komunikatów przesyłanych przez metodę `reportBack()`. Aktywność ta będzie nadziedziona w stosunku do aktywności pasków działania.

## **Wprowadzenie jednolitego zachowania klas ActionBar**

Mamy jeszcze więcej okazji do refaktoryzacji kodu pochodzącego z aktywności dziedziczących i umieszczenia go w nowej klasie bazowej, nazwanej `BaseActionBarActivity`.

Głównym zadaniem tej klasy jest zagwarantowanie wspólnego kodu, przetwarzanego w odpowiedzi na kliknięcia elementów menu. Elementy te posłużą nam do zmiany trzech aktywności reprezentujących tryby wyświetlania paska działania. Po zmianie trybu będziemy mogli przetestować kontrolującą go aktywność.

Aktywność `BaseActionBarActivity` widzimy na listingu 30.3.

**Listing 30.3.** Wspólna klasa bazowa dla aktywności przechowującej paski działania

```
// BaseActionBarActivity.java
package com.androidbook.actionbar;
//
// Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public abstract class BaseActionBarActivity
    extends DebugActivity
{
    private String tag=null;
    public BaseActionBarActivity(String inTag)
    {
        super(R.menu.menu,
              R.layout.main,
              R.id.textViewId,
              inTag);
        tag = inTag;
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        TextView tv = this.getTextView();
        tv.setText(tag);
    }
    protected boolean onMenuItemSelected(MenuItem item)
    {
        //Reaguje na przycisk strony startowej
        if (item.getItemId() == android.R.id.home) {
            this.reportBack(tag,"Ikona startowa wcisnięta");
            return true;
        }

        //Wspólne zachowanie służące do wywoływania równorzędnych aktywności
        if (item.getItemId() == R.id.menu_invoke_tabnav){
            if (getNavMode() ==
                ActionBar.NAVIGATION_MODE_TABS)
            {
                this.reportBack(tag,
                               "Już przebywamy w aktywności paska zakładek");
            }
            else {
                this.invokeTabNav();
            }
            return true;
        }
        if (item.getItemId() == R.id.menu_invoke_listnav){
```

```
    if (getNavMode() ==  
        ActionBar.NAVIGATION_MODE_LIST)  
    {  
        this.reportBack(tag,  
                      "Już przebywamy w aktywności paska wyświetlającego listę");  
    }  
    else{  
        this.invokeListNav();  
    }  
    return true;  
}  
if (item.getItemId() == R.id.menu_invoke_standardnav){  
    if (getNavMode() ==  
        ActionBar.NAVIGATION_MODE_STANDARD)  
    {  
        this.reportBack(tag,  
                      "Już przebywamy w aktywności standardowego paska działania");  
    }  
    else{  
        this.invokeStandardNav();  
    }  
    return true;  
}  
return false;  
}  
private int getNavMode(){  
    ActionBar bar = this.getActionBar();  
    return bar.getNavigationMode();  
}  
private void invokeTabNav(){  
    Intent i = new Intent(this,  
                          TabNavigationActionBarActivity.class);  
    startActivity(i);  
}  
  
//Odkomentujmy poniższe ciała metod  
//wraz z implementowaniem kolejnych aktywności  
  
private void invokeListNav(){  
    //Intent i = new Intent(this,  
    //                      ListNavigationActionBarActivity.class);  
    //startActivity(i);  
}  
private void invokeStandardNav(){  
    //Intent i = new Intent(this,  
    //                      StandardNavigationActionBarActivity.class);  
    //startActivity(i);  
}  
}//eof-class
```

---

Jeżeli przeanalizujemy kod zapewniający obsługę elementów menu, zauważymy, że sprawdza on aktywność, która ma zostać przywołana w wyniku kliknięcia elementu menu. Jeżeli jest to bieżąca aktywność, Android wyświetla odpowiedni komunikat w widoku debugowania i pozostawia tę aktywność uruchomioną.

Ta bazowa aktywność upraszcza również kod pochodnych aktywności pasków działania, w tym również aktywności paska zakładek.

## Implementacja obiektu nasłuchującego zdarzeń z zakładek

Zanim przejdziemy do pracy z paskiem zakładek, musimy najpierw wprowadzić obiekt nasłuchujący zakładki. Obiekt ten pozwala na reagowanie na kliknięcia poszczególnych zakładek. Klasa tego obiektu wywodzi się z bazowego obiektu nasłuchującego, służącego do zapisywania komunikatów w dzienniku o działaniach na zakładkach. Na listingu 30.4 zaprezentowaliśmy bazowy obiekt nasłuchujący, który wykorzystuje interfejs IReportBack do wyświetlania komunikatów.

**Listing 30.4.** Wspólny obiekt nasłuchujący utworzony z myślą o aktywnościach przechowujących paski działań

---

```
//BaseListener.java
package com.androidbook.actionbar;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class BaseListener
{
    protected IReportBack mReportTo;
    protected Context mContext;
    public BaseListener(Context ctx, IReportBack target)
    {
        mReportTo = target;
        mContext = ctx;
    }
}
```

---

Omawiana klasa bazowa przechowuje odniesienie do implementacji interfejsu IReportBack oraz do aktywności, która jest wykorzystywana w postaci kontekstu. W naszym przypadku to aktywność DebugActivity implementuje ten interfejs oraz odgrywa rolę kontekstu.

Skoro utworzyliśmy już bazowy obiekt nasłuchujący, możemy zająć się obiektem nasłuchującym zakładki. Odpowiedni kod umieściliśmy na listingu 30.5.

**Listing 30.5.** Obiekt nasłuchujący zakładki w oczekiwaniu na działania

---

```
// TabListener.java
package com.androidbook.actionbar;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class TabListener extends BaseListener
implements ActionBar.TabListener
{
    private static String tag = "tc>";
    public TabListener(Context ctx,
                      IReportBack target)
    {
        super(ctx, target);
    }
}
```

---

```
public void onTabReselected(Tab tab,
                           FragmentTransaction ft)
{
    this.mReportTo.reportBack(tag,
        "ponownie wybrana zakładka ontab:" + tab.getText());
}
public void onTabSelected(Tab tab,
                         FragmentTransaction ft)
{
    this.mReportTo.reportBack(tag,
        "wybrana zakładka ontab:" + tab.getText());
}
public void onTabUnselected(Tab tab,
                           FragmentTransaction ft)
{
    this.mReportTo.reportBack(tag,
        "usunięto zaznaczenie zakładki ontab:" + tab.getText());
}
```

---

Obiekt nasłuchujący zakładki służy wyłącznie do przekazywania do widoku debugowania (rysunek 30.1) informacji o wywołaniach następujących z poziomu zakładek.

## Implementacja aktywności przechowującej pasek zakładek

Po wprowadzeniu obiektu nasłuchującego zakładki przechodzimy w końcu do tworzenia aktywności przechowującej pasek zakładek. Jej kod zaprezentowano na listingu 30.6.

### **Listing 30.6.** Aktywność paska zakładek

---

```
// TabNavigationActionBarActivity.java
package com.androidbook.actionbar;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class TabNavigationActionBarActivity
extends BaseActionBarActivity
{
    private static String tag =
        "Klasa ActionBarActivity do nawigacji za pomocą zakładek";
    public TabNavigationActionBarActivity()
    {
        super(tag);
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        workwithTabbedActionBar();
    }
    public void workwithTabbedActionBar()
    {
        ActionBar bar = this.getActionBar();
        bar.setTitle(tag);
```

---

```

        bar.setNavigationMode(
            ActionBar.NAVIGATION_MODE_TABS);

        TabListener tl = new TabListener(this, this);

        Tab tab1 = bar.newTab();
        tab1.setText("Zakładka1");
        tab1.setTabListener(tl);
        bar.addTab(tab1);

        Tab tab2 = bar.newTab();
        tab2.setText("Zakładka2");
        tab2.setTabListener(tl);
        bar.addTab(tab2);
    }
}//eof-class

```

---

W kolejnych podpunktach omówimy teraz kod składający się na aktywność paska zakładek (listing 30.6). Zwrócimy Czytelnikowi uwagę na każdy aspekt pracy z paskiem zakładek. Rozpoczniemy od kodu uzyskującego dostęp do paska działań, który stanowi część aktywności.

## Uzyskanie instancji paska działania

Analizując listing 30.6, zwróćmy uwagę, że kod kontrolujący pasek działania wcale nie jest skomplikowany. Dostęp do paska działania danej aktywności uzyskujemy poprzez wywołanie metody `getActionBar()`. Przypomnijmy wygląd całego wiersza:

```
ActionBar bar = this.getActionBar();
```

Zgodnie z zaprezentowanym fragmentem kodu pasek działania stanowi właściwość aktywności i nie przekracza jej granic. Innymi słowy, nie możemy wykorzystać paska działania do kontroliowania wielu aktywności lub wpływu na nie.

## Tryby nawigacji paska działania

Po otrzymaniu instancji paska działań aktywności ustanawiamy jego tryb wyświetlania jako `ActionBar.NAVIGATION_MODE_TABS`. Poniżej prezentujemy cały wiersz kodu:

```
bar.setNavigationMode(
    ActionBar.NAVIGATION_MODE_TABS);
```

Dwa pozostałe tryby to:

```
ActionBar.NAVIGATION_MODE_LIST
ActionBar.NAVIGATION_MODE_STANDARD
```

Po skonfigurowaniu trybu wyświetlania zakładek widzimy wiele związków z nimi metod wewnątrz klasy `ActionBar`, z których możemy korzystać. W kodzie z listingu 30.6 wykorzystaliśmy te metody do dodania dwóch zakładek do paska działań. Do inicjalizacji tych zakładek użyliśmy z kolei obiektu nasłuchującego zakładki, zdefiniowanego na listingu 30.5.

Poniżej umieściliśmy krótki wycinek kodu z listingu 30.6 ukazujący sposób dodawania zakładki do paska zadań:

```
Tab tab1 = bar.newTab();
tab1.setText("Zakładka1");
```

```
tab1.setTabListener(t1);
bar.addTab(tab1);
```

Jeżeli zapomniemy wywołać metodę `setTabListener()` wobec zakładki dodawanej do paska działań, zostanie wyświetlony komunikat o błędzie wskazujący na brak obiektu nasłuchującego.

## Przewijalny układ graficzny zawierający widok debugowania

Po każdym kliknięciu zakładki w pasku działań obiekty nasłuchujące zakładki wysyłają komunikaty do widoku debugowania. Na listingu 30.7 został zaprezentowany układ graficzny klasy `DebugActivity`, w której znajduje się widok debugowania.

**Listing 30.7.** Plik układu graficznego aktywności debugowania

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/main.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:gravity="fill"
        >
<TextView android:id="@+id/textViewId"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:background="@android:color/white"
        android:text="Początkowa wiadomość tekstowa"
        android:textColor="@android:color/black"
        android:textSize="25sp"
        android:scrollbars="vertical"
        android:scrollbarStyle="insideOverlay"
        android:scrollbarSize="25dip"
        android:scrollbarFadeDuration="0"
        />
</LinearLayout>
```

---

Warto zwrócić uwagę na kilka elementów tego układu graficznego. Wprowadzamy biały kolor tła widoku tekstuowego. W ten sposób zrzuty ekranu będą nieco wyraźniejsze. Również w podobnym celu zwiększyliśmy rozmiar czcionki.

Konfigurujemy także widok tekstowy w taki sposób, aby można go było przewijać. Chociaż zazwyczaj do takich zastosowań używane są widoki `ScrollView`, widok tekstowy sam w sobie posiada dostępną funkcję przewijania. Oprócz uruchomienia funkcji przewijania w pliku XML musimy także wywołać metodę `setMovementMethod()` wobec tego widoku tekstuowego — tak jak zostało pokazane na listingu 30.8.

**Listing 30.8.** Wprowadzenie funkcji przewijania w widoku tekstowym

---

```
TextView tv = this.getTextView();
tv.setMovementMethod(
    ScrollingMovementMethod.getInstance());
```

---

Jest to wycinek kodu pobrany z klasy `DebugActivity` (listing 30.2).

W trakcie przewijania widoku tekstowego stwierdzimy również, że pasek przewijania pojawia się, a następnie zanika. Nie jest to zbyt wygodny wskaźnik ilości tekstu znajdującego się poza ekranem. Możemy wyłączyć zanikanie tego paska poprzez ustalenie wartości 0 dla jego parametru `android:scrollbarFadeDuration`. Parametr ten jest widoczny na listingu 30.7.

## Pasek działania a interakcja z menu

Chcemy także zaprezentować w tym przykładzie mechanizm interakcji paska działania z elementami menu. Musimy więc w tym celu skonfigurować również plik menu. Prezentujemy go na listingu 30.9.

**Listing 30.9.** Plik menu utworzony na potrzeby omawianego projektu

```
<!-- /res/menu/menu.xml -->
<menu
    xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Ta grupa korzysta z domyślnej kategorii. -->
    <group android:id="@+id/menuGroup_Main">

        <item android:id="@+id/menu_action_icon1"
            android:title="Ikona działania1"
            android:icon="@drawable/creep001"
            android:showAsAction="ifRoom"/>

        <item android:id="@+id/menu_action_icon2"
            android:title="Ikona działania2"
            android:icon="@drawable/creep002"
            android:showAsAction="ifRoom"/>

        <item android:id="@+id/menu_icon_test"
            android:title="Ikona testowa"
            android:icon="@drawable/creep003"/>

        <item android:id="@+id/menu_invoke_listnav"
            android:title="Wywołaj pasek listy"
            />
        <item android:id="@+id/menu_invoke_standardnav"
            android:title="Wywołaj pasek standardowy"
            />
        <item android:id="@+id/menu_invoke_tabnav"
            android:title="Wywołaj pasek zakładek"
            />
        <item android:id="@+id/menu_da_clear"
            android:title="wyczyszc" />
    </group>
</menu>
```

### Uwaga!

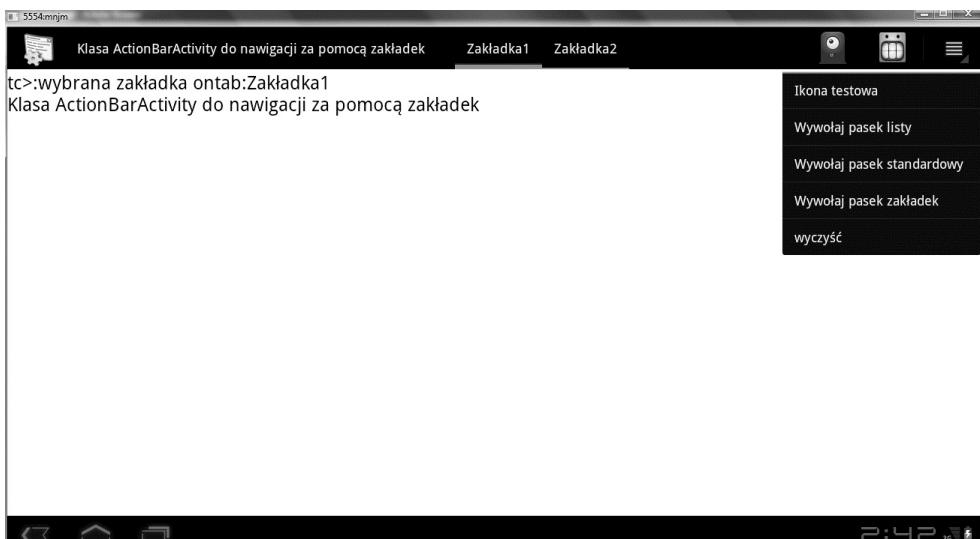
W pliku menu z listingu 30.9 wykorzystaliśmy trzy ikony (`creep001`, `002` i `003`) pochodzące ze strony [www.androidicons.com](http://www.androidicons.com). Zgodnie z informacjami zawartymi na tej stronie ikony te są objęte licencją Creative Commons 3.0.

W kolejnym podpunkcie przyjrzymy się nieco dokładniej utworzonemu właśnie menu.

## Wyświetlanie menu

Urządzenia pracujące pod kontrolą systemu Android w wersjach 2.3 i wcześniejszych często były wyposażone we wbudowany przycisk menu. Emulator działający w trybie dla wersji 3.0 systemu nie symuluje fizycznych przycisków powrotu do menu startowego, cofania lub menu, mogą być one jednak wciąż dostępne w niektórych urządzeniach.

Jak widzimy na rysunku 30.2, przyciski ekranu startowego i cofania są teraz przyciskami programowymi, umieszczonymi u dołu ekranu. Jednak przycisk menu jest wyświetlany w kontekście aplikacji, a dokładniej jako część paska działań, i jest umieszczony w prawym górnym rogu ekranu.



**Rysunek 30.2.** Aktywność wyświetlająca pasek zakładek i rozwinięte menu

Rysunek 30.2 przedstawia rozwinięte menu.

Należy zwrócić uwagę, że pasek menu nie musi wyświetlać ikon reprezentujących elementy menu. Nie należy liczyć na to, że w każdym przypadku wyświetlane ikony będą reprezentować elementy menu.

## Elementy menu jako działania

Jak już wspomnieliśmy na początku tego rozdziału, część elementów menu możemy zdefiniować w taki sposób, aby były wyświetlane bezpośrednio w pasku działania. Elementy te są reprezentowane przez znacznik `showAsAction`. Widzimy go na listingu 30.9. Dla przypomnienia prezentujmy ten fragment kodu na listingu 30.10.

### **Listing 30.10.** Atrybut elementu menu `showAsAction`

---

```
android:showAsAction="ifRoom"
```

---

Pozostałymi wartościami tego znacznika mogą być:

always  
never  
withText

Taki sam efekt możemy osiągnąć za pomocą klasy MenuItem:

```
menuItem.setShowAsAction(int actionEnum)
```

Dostępne są następujące wartości atrybutu actionEnum:

SHOW\_AS\_ACTION\_ALWAYS  
SHOW\_AS\_ACTION\_IF\_ROOM  
SHOW\_AS\_ACTION\_NEVER  
SHOW\_AS\_ACTION\_WITH\_TEXT

Ponieważ działania te są jedynie elementami menu, tak też się zachowują i wywołują metodę zwrotną onOptionsItemSelected() klasy danej aktywności.

Zauważmy na koniec, że w tym przykładzie wykorzystujemy sporą liczbę ikon. Możemy je zastąpić własnymi ikonami lub pobrać odpowiedni projekt, do którego łącze zostało zamieszczone na końcu rozdziału.

## Plik manifest Androida

Na listingu 30.11 została umieszczona zawartość pliku manifestu dla dotychczas utworzonej części projektu.

**Listing 30.11.** AndroidManifest.xml

---

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.actionbar"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"
        android:label="Demonstracja klasy ActionBar">
        <activity android:name=".TabNavigationActionBarActivity"
            android:label="Demonstracja paska działania: TabNav">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="11" />
</manifest>
```

---

Wartość parametru minSdkVersion wynosi 11 i odpowiada wersji 3.0 systemu.

## Badanie aktywności przechowującej pasek zakładek

Po skompilowaniu utworzonych plików i uruchomieniu aplikacji ujrzymy, widoczny na rysunku 30.1, pasek zakładek. Następnie po kliknięciu znajdującej się po prawej stronie ikony menu zostanie rozwinięte menu aplikacji, co zostało zilustrowane na rysunku 30.2.

Aplikacja została zaprojektowana w taki sposób, aby każde działanie na pasku działania zostało zapisane w widoku debugowania. Podczas testowania aplikacji Czytelnik może sprawdzić, co się stanie po wykonaniu następujących czynności:

- Jeżeli klikniemy przycisk strony startowej, ujrzymy w widoku tekstowym komunikat, że wcisnięto przycisk ekranu startowego.
- Jeżeli klikniemy zakładkę *Zakładka1*, pojawi się informacja, że *Zakładka1* została ponownie wybrana.
- Jeżeli klikniemy zakładkę *Zakładka2*, zostaną wyświetlane dwa komunikaty. Pierwszy z nich informuje, że *Zakładka1* schodzi z pierwszego planu, a z drugiego dowiadujemy się o kliknięciu *Zakładka2*. Komunikaty te zostają dostarczone przez zaprezentowane na listingu 30.5 obiekty nasłuchujące zakładki.
- Jeżeli klikniemy widoczne po prawej stronie przyciski działania, zauważymy, że zostaną wywołane powiązane z nimi elementy menu, a tym samym zostaną wstawione kolejne komunikaty w widoku tekstowym.
- Po rozwinięciu menu zobaczymy elementy menu służące do wywoływania innych aktywności, które pozwalają na zaprezentowanie pozostałych trybów wyświetlania paska działania. Musimy jednak z tym poczekać do zakończenia procesu tworzenia projektu. Do tego czasu klikanie tych elementów spowoduje jedynie wyświetlanie komunikatów w widoku tekstowym.

Na tym zakończymy implementację nie tylko aktywności wyświetlającej pasek zakładek, lecz również konfigurację podstawowej struktury, dzięki której utworzenie pozostałych aktywności okaże się znacznie łatwiejsze. Przejdzmy teraz do paska działań zdefiniowanego w trybie wyświetlania listy.

## Aktywność paska działania w trybie wyświetlania listy

Nasze bazowe klasy wykonują większą część pracy, zatem implementacja i przetestowanie aktywności przechowującej pasek działania w trybie wyświetlania listy nie powinny przysporzyć nam najmniejszych problemów. Do tego celu będą nam potrzebne następujące pliki:

- *SimpleSpinnerAdapter.java* — klasa ta jest wymagana do skonfigurowania paska wyświetlającego listę wraz z obiektem nasłuchującym. Zawarte są w niej wiersze wymagane przez rozwijalną listę (listing 30.12).
- *ListListener.java* — klasa ta pełni rolę obiektu nasłuchującego aktywności, która przechowuje testowany pasek działań. Musi ona zostać przekazana paskowi działania w momencie przejścia do trybu wyświetlania listy (listing 30.13).
- *ListNavigationActionBarActivity.java* — to właśnie tutaj zaimplementujemy aktywność paska działania wyświetlającego listę (listing 30.14).

Po utworzeniu tych trzech plików będziemy musieli zaktualizować kolejne dwa:

- *BaseActionBarActivity.java* — musimy usunąć znaki komentarza z kodu, który wywołuje aktywność przechowującą omawiany pasek działania (listing 30.3).
- *AndroidManifest.xml* — zdefiniujemy nową aktywność przechowującą pasek działania wyświetlany w trybie listy (listing 30.11).

## Utworzenie klasy SpinnerAdapter

Aby zainicjalizować pasek działania w trybie wyświetlania listy, potrzebne są nam dwa elementy:

- Adapter obiektu typu Spinner, który będzie zapełniał listę tekstem.
- Obiekt nasłuchujący listę, wywołujący metodę zwrotną po wybraniu dowolnego elementu listy.

Na listingu 30.12 widzimy klasę SimpleSpinnerArrayAdapter, która implementuje interfejs SpinnerAdapter. Jak już wspomnieliśmy, zadaniem tej klasy jest wyświetlenie listy elementów.

**Listing 30.12.** Utworzenie adaptera obiektu typu Spinner dla paska działań wyświetlającego listę

---

```
//SimpleSpinnerArrayAdapter.java
package com.androidbook.actionbar;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class SimpleSpinnerArrayAdapter
    extends ArrayAdapter<String>
    implements SpinnerAdapter
{
    public SimpleSpinnerArrayAdapter(Context ctx)
    {
        super(ctx,
            android.R.layout.simple_spinner_item,
            new String[]{"raz","dwa"});

        this.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);
    }
    public View getDropDownView(
        int position, View convertView, ViewGroup parent)
    {
        return super.getDropDownView(
            position, convertView, parent);
    }
}
```

---

Nie istnieje klasa bezpośrednio implementująca wymagany przez tę listę interfejs SpinnerAdapter. Utworzyliśmy zatem pochodną klasy ArrayAdapter i wprowadziliśmy do niej prostą implementację wspomnianego interfejsu. Na końcu rozdziału zamieściliśmy adres URL do dalszych materiałów poświęconych tego typu adapterom. Przejdzmy teraz do obiektu nasłuchującego listy.

## Utworzenie obiektu nasłuchującego listy

Jest to prosta klasa implementująca interfejs ActionBar.OnNavigationListener. Kod tej klasy został umieszczony na listingu 30.13.

**Listing 30.13.** Tworzenie obiektu nasłuchującego listy nawigacji

---

```
//ListListener.java
package com.androidbook.actionbar;
//
```

```
// Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class ListListener
extends BaseListener
implements ActionBar.OnNavigationListener
{
    public ListListener(
        Context ctx, IReportBack target)
    {
        super(ctx, target);
    }
    public boolean onNavigationItemSelected(
        int itemPosition, long itemId)
    {
        this.mReportTo.reportBack(
            "obiekt nasłuchujący listy", "ItemPosition:" + itemPosition);
        return true;
    }
}
```

---

Podobnie jak w przypadku obiektu nasłuchującego zakładki z listingu 30.5, dziedziczymy tu po klasie `BaseListener`, dzięki czemu możemy zapisywać komunikaty w widoku debugowania poprzez interfejs `IReportBack`.

## Konfigurowanie paska działania w trybie wyświetlania listy

Utworzyliśmy już wszystkie elementy potrzebne do skonfigurowania paska działania. Spójrzmy teraz na kod źródłowy aktywności przechowującej ten pasek działania w trybie wyświetlania listy, umieszczony na listingu 30.14. Klasa ta jest bardzo podobna do omawianej wcześniej aktywności przechowującej pasek zakładek.

---

**Listing 30.14.** Aktywność paska działania w trybie wyświetlania listy

---

```
// ListNavigationActionBarActivity.java
package com.androidbook.actionbar;
//
// Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class ListNavigationActionBarActivity
extends BaseActionBarActivity
{
    private static String tag=
        "Klasa ActionBarActivity";

    public ListNavigationActionBarActivity()
    {
        super(tag);
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
```

---

```

        super.onCreate(savedInstanceState);
        workwithActionBar();
    }
    public void workwithActionBar()
    {
        ActionBar bar = this.getActionBar();
        bar.setTitle(tag);
        bar.setNavigationMode(ActionBar.NAVIGATION_MODE_LIST);
        bar.setListNavigationCallbacks(
            new SimpleSpinnerArrayAdapter(this),
            new ListListener(this,this));
    }
}//eof-class

```

---

Istotna część kodu na listingu 30.14 została zaznaczona pogrubioną czcionką. Sam kod nie jest zbyt skomplikowany: w metodach zwrotnych paska działania konfigurujemy adapter obiektu typu Spinner oraz obiekt nasłuchujący jako listę.

## Zmiany w klasie BaseActionBarActivity

Po utworzeniu aktywności przechowującej pasek działań w trybie wyświetlania listy (listing 30.14) możemy cofnąć się i zmodyfikować kod klasy `BaseActionBarActivity` w taki sposób, żeby kliknięcie odpowiedniego elementu menu wywołało klasę `ListNavigationActionBarActivity`. Po usunięciu znaków komentarza z odpowiedniego fragmentu funkcji z listingu 30.3 kod ten będzie wyglądał dokładnie tak jak fragment zamieszczony na listingu 30.15.

**Listing 30.15.** Zmodyfikowany fragment kodu — wywoływanie aktywności przechowującej pasek działania w trybie wyświetlania listy

---

```

private void invokeListNav(){
    Intent i = new Intent(this,
        ListNavigationActionBarActivity.class);
    startActivity(i);
}

```

---

Efektem usunięcia znaków komentarza z tego fragmentu kodu będzie powiązanie elementu menu. W konsekwencji będzie można wywołać omawianą aktywność.

## Zmiany w pliku AndroidManifest.xml

Zanim będziemy mogli wywołać tę aktywność, musimy ją zarejestrować w pliku `AndroidManifest.xml`. Dodajmy kod widoczny na listingu 30.16 do manifestu z listingu 30.11, aby dokończyć proces rejestracji aktywności.

**Listing 30.16.** Rejestrowanie aktywności przechowującej pasek działania w trybie wyświetlania listy

---

```

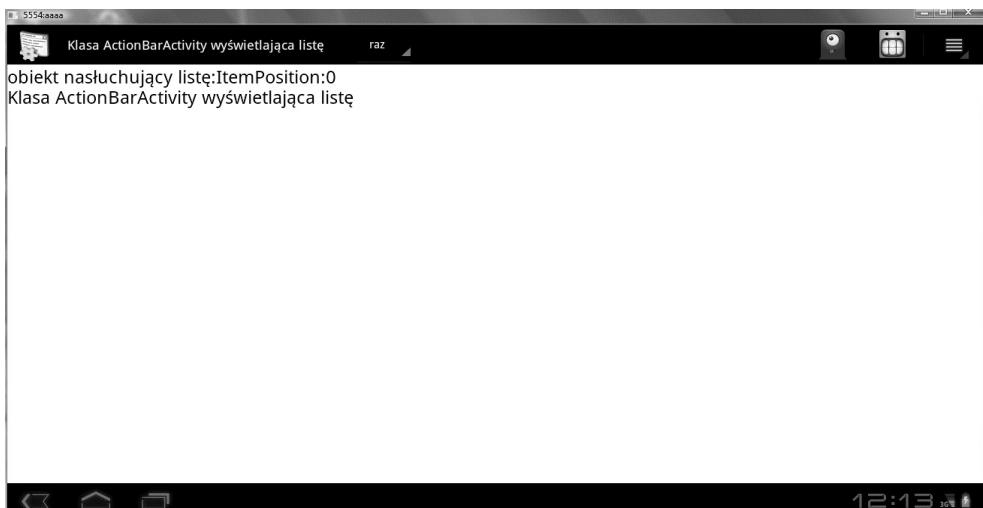
<activity android:name=".ListNavigationActionBarActivity"
    android:label="Demonstracja paska działania: ListNav">
</activity>

```

---

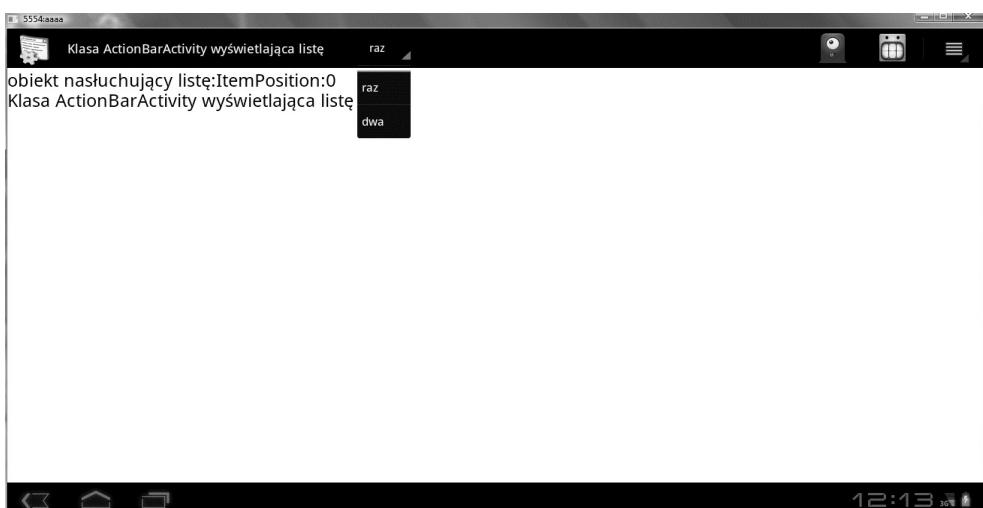
## Badanie aktywności zawierającej pasek działania w trybie wyświetlania listy

Po skompilowaniu dotychczas omówionych plików (nowych plików oraz zaktualizowanych wersji plików wymienionych na początku podrozdziału) oraz uruchomieniu programu ujrzymy pasek działania, zaprezentowany na rysunku 30.3.



Rysunek 30.3. Aktywność przechowująca pasek działania w trybie wyświetlania listy

Na rysunku 30.3 widzimy nierozwiniętą listę, znajdująjącą się tuż obok tytułu aktywności. W tym samym miejscu są umieszczane zakładki, jeśli pasek działania pracuje w trybie wyświetlania zakładek. Jeżeli klikniemy teraz element nazwany *raz*, lista zostanie rozwinięta. Widzimy to na rysunku 30.4.



Rysunek 30.4. Aktywność z widoczną rozwiniętą listą

Gdy porównamy tę aktywność do aktywności widocznej na rysunkach 30.1 i 30.2, stwierdzimy, że są one do siebie bardzo podobne. Różnica między nimi polega na tym, że w pierwszym przypadku na pasku działania mamy do czynienia z zakładkami, a w drugim — z listą. Zadaniem tych dwóch aktywności jest ukazanie Czytelnikowi istotnego podobieństwa procesów projektowania aplikacji dla Androida i stron WWW.

W przypadku witryny WWW możemy mieć do czynienia z wieloma stronami, każda z nich jednak wygląda podobnie do strony głównej. W naszym przypadku takim odpowiednikiem głównej strony jest klasa bazowa.

Chociaż wprowadziliśmy wiele aktywności w celu zademonstrowania pasków działania, wydaje się, że w wersji 3.0 systemu obiekty te lepiej się nadają do organizowania fragmentów wewnętrz jednej aktywności. Jeżeli jednak będziemy musieli wprowadzić wiele aktywności, warto się za stanowić nad zademonstrowanym przez nas wzorcem stosowania klasy bazowej jako głównej strony.

Zachowanie aktywności zawierającej pasek działań w trybie listy jest bardzo podobne do zachowania aktywności z paskiem działań w trybie paska zakładek. Różnica polega na efekcie kliknięcia elementu listy. Po każdym kliknięciu elementu listy zostanie wywołany obiekt naślu chujący listy, który z kolei wyświetli komunikat w widoku debugowania.

Po utworzeniu tych dwóch aktywności będziemy mogli je zmieniać za pomocą elementów menu.

Przejdźmy teraz do prostszej aktywności przechowującej standardowy pasek działania.

## Aktywność przechowująca standardowy pasek działania

W niniejszym podrozdziale zapoznamy się z funkcjonowaniem standardowego paska działania. Wprowadzimy nową aktywność, w której ustanowimy standardowy tryb nawigacji paska działania. Następnie przyjrzymy się wyglądowi i zachowaniu tego paska działania.

Podobnie jak miało to miejsce w przypadku klasy `ListNavigationActionBarActivity`, większość operacji jest wykonywana przez klasy bazowe, zatem implementacja i testowanie omawianej aktywności nie powinny nastręczać zbyt dużych problemów. Aby zaimplementować tę aktywność, potrzebny nam będzie następujący plik:

- `StandardNavigationActionBarActivity.java` — plik ten służy do skonfigurowania standardowego trybu wyświetlania paska działania (listing 30.17).

Po utworzeniu tego pliku musimy zmodyfikować następujące dwa pliki:

- `BaseActionBarActivity.java` — należy usunąć znaki komentarza wywołania aktywności wyświetlającej standardowy pasek działania po kliknięciu odpowiedniego elementu menu (zmiany zostały pokazane na listingu 30.18, a oryginalny kod — na listingu 30.3).
- `AndroidManifest.xml` — musimy zdefiniować tę nową aktywność w pliku manifeście (definicję tej aktywności widzimy na listingu 30.19, a należy ją umieścić w pliku `AndroidManifest.xml` umieszczonym na listingu 30.11).

Zajmiemy się teraz każdym z wymienionych plików.

## Aktywność przechowująca standardowy pasek działania

W przypadku pasków zakładek i pasków działania w trybie listy korzystaliśmy z odpowiednich obiektów nasłuchujących. Oprócz metod zwrotnych elementów menu nie wprowadzamy żadnego innego obiektu nasłuchującego standardowego paska działania. Nie musimy konfigurować tych metod zwrotnych, ponieważ są one automatycznie umieszczone przez środowisko SDK. W związku z tym konfiguracja standardowego paska działania jest niezwykle prosta.

Na listingu 30.17 znajdziemy kod źródłowy aktywności wyświetlającej standardowy pasek działania.

**Listing 30.17.** Aktywność przechowująca standardowy pasek działania

---

```
//StandardActionBarActivity.java
package com.androidbook.actionbar;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
public class StandardActionBarActivity
extends BaseActionBarActivity
{
    private static String tag=
        "Standardowa nawigacja za pomocą klasy ActionBarActivity";
    public StandardActionBarActivity()
    {
        super(tag);
    }
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        workwithStandardActionBar();
    }

    public void workwithStandardActionBar()
    {
        ActionBar bar = this.getActionBar();
        bar.setTitle(tag);
        bar.setNavigationMode(ActionBar.NAVIGATION_MODE_STANDARD);
        //Sprawdzamy, co się stanie, jeśli umieścimy zakładki
        attachTabs(bar);
    }
    public void attachTabs(ActionBar bar)
    {
        TabListener tl = new TabListener(this,this);

        Tab tab1 = bar.newTab();
        tab1.setText("Zakładka1");
        tab1.setTabListener(tl);
        bar.addTab(tab1);

        Tab tab2 = bar.newTab();
        tab2.setText("Zakładka2");
    }
}
```

---

```

        tab2.setTabListener(tl);
        bar.addTab(tab2);
    }
} //eof-class

```

---

Jedyną czynnością, jaką należy wykonać w celu zdefiniowania standardowego paska działania, jest wybór trybu nawigacji. Na listingu 30.17 wyróżniliśmy odpowiedni fragment kodu pogrubioną czcionką.

**Uwaga!** Na listingu 30.17 wprowadziliśmy również kod generujący zakładki, aby sprawdzić, co się z nimi stanie w standardowym trybie nawigacji. Nasze testy wykazały, że obecność tego fragmentu kodu nie generuje żadnych błędów, ale jest on po prostu ignorowany.

Zanim dowiemy się, jak wygląda standardowy pasek działania, musimy wprowadzić dwie zmiany do już istniejących plików.

## Zmiany w klasie BaseActionBarActivity

Po utworzeniu aktywności wyświetlającej standardowy pasek działania (listing 30.17) możemy się cofnąć i zmodyfikować klasę `BaseActionBarActivity` (listing 30.3) w taki sposób, aby omawiana aktywność była wywoływana przez odpowiedni element menu. Po usunięciu znaków komentarza funkcja widoczna na listingu 30.3 będzie wyglądała tak jak pokazano na listingu 30.18.

**Listing 30.18.** Fragment, z którego należy usunąć znaki komentarza, aby umożliwić wywoływanie aktywności przechowującej standardowy pasek działania

---

```

private void invokeStandardNav(){
    Intent i = new Intent(this,
        StandardNavigationActionBarActivity.class);
    startActivity(i);
}

```

---

Po usunięciu znaków komentarza z tego fragmentu element menu zostanie powiązany z kodem w sposób umożliwiający wywoływanie klasy `StandardNavigationActionBarActivity`.

## Zmiany w pliku AndroidManifest.xml

Zanim jednak będzie można wywołać tę aktywność, trzeba zarejestrować ją w pliku manifestie. Aby dokończyć proces rejestracji, dodajmy wiersze umieszczone na listingu 30.19 do pliku `AndroidManifest.xml`, widocznego na listingu 30.11.

**Listing 30.19.** Rejestrowanie aktywności wyświetlającej standardowy pasek działania

---

```

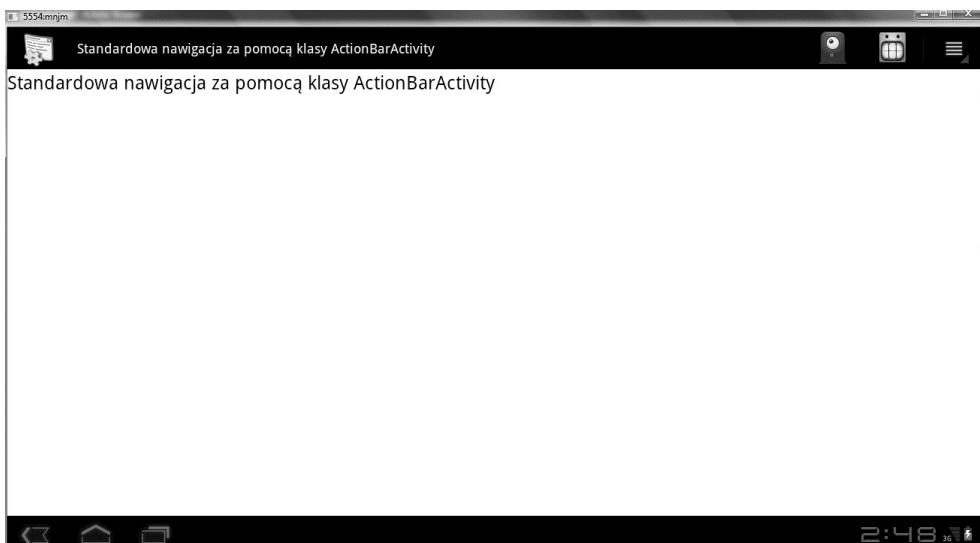
<activity android:name=".StandardNavigationActionBarActivity"
    android:label="Demonstracja paska działania: standardowa nawigacja">
</activity>

```

---

## Badanie aktywności przechowującej standardowy pasek działania

Gdy skompilujemy dotychczas omówione pliki (wymienione na początku podrozdziału) i uruchomimy aplikację, pierwsza widoczna aktywność będzie zawierała pasek zakładek (rysunek 30.1). Po kliknięciu ikony menu zobaczymy ekran zaprezentowany na rysunku 30.2. Z poziomu tego menu możemy wybrać opcję *Wywołaj pasek standardowy*, po czym pojawi się widoczna na rysunku 30.5 aktywność wyświetlająca standardowy pasek działania.



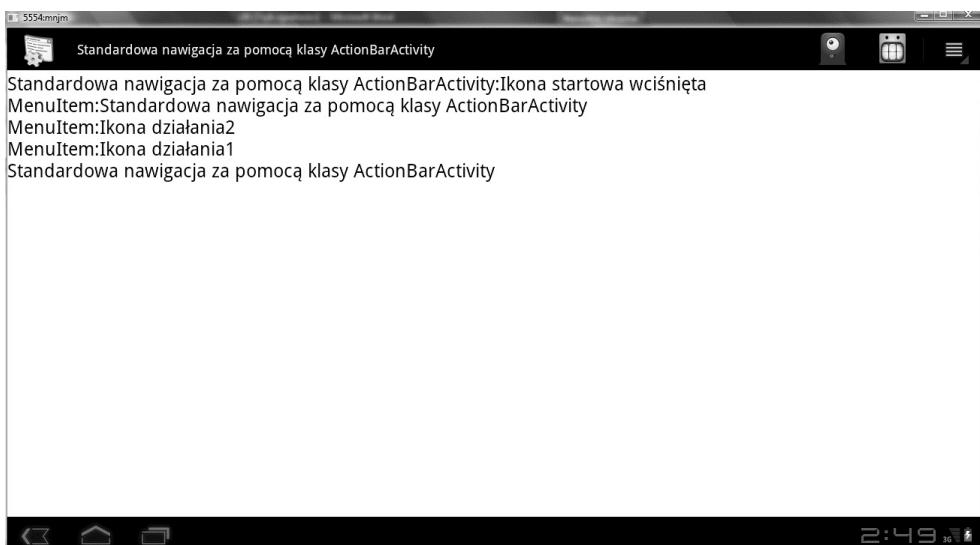
**Rysunek 30.5.** Aktywność zawierająca standardowy pasek działania

Pierwszą rzeczą przykuwającą uwagę Czytelnika będzie zapewne brak obszaru, w którym wcześniej były wyświetlane zakładki oraz lista. Jeżeli klikniemy teraz widoczne po prawej stronie przyciski działania, informacje o ich wywołaniu zostaną zapisane w widoku debugowania. Kliknijmy teraz przycisk ekranu startowego. Również w tym przypadku w widoku debugowania zostanie zapisana sygnatura wywołania. Po wykonaniu tych trzech kliknięć widok debugowania będzie wyglądał tak jak na rysunku 30.6.

## Odrośniki

Poniższe adresy URL okazały się bardzo przydatne w trakcie wyszukiwania materiałów do niższego rozdziału. Zawierają one również ogrom dodatkowych informacji. Dodatkowo ostatnie łącze prowadzi do strony, z której możemy pobrać gotowe projekty.

- Donald A. Norman, *The Design of Everyday Things*. Książka ta rozpowszechniła wspomnianą wcześniej koncepcję „wizualnej percepcji”, zwaną afordancją, w stosunku do dziedziny HCI (ang. *Human-Computer Interaction* — interakcja człowiek – komputer). Termin ten jest stosowany coraz częściej w literaturze poświęconej interfejsom użytkownika systemu Android. Omawiany w tym rozdziale pasek działania jest zachwalany jako jedna z głównych afordancji interfejsu użytkownika.



Rysunek 30.6. Reagowanie na zdarzenia zachodzące w pasku działania

- <http://en.wikipedia.org/wiki/Affordance> — definicja afordancji w Wikipedii.
- [www.androidbook.com/item/3624](http://www.androidbook.com/item/3624) — wyniki naszych badań związanych z paskiem działania systemu Android. Znajdziemy tutaj dalsze odnośniki, przykładowe kody, adresy do przykładowych aplikacji oraz rysunki reprezentujące różne tryby wyświetlania paska działania.
- <http://developer.android.com/reference/android/app/ActionBar.html> — dokumentacja klasy ActionBar.
- [www.androidbook.com/item/3627](http://www.androidbook.com/item/3627) — korzystanie z adaptera obiektu typu Spinner. Aby ustanowić tryb wyświetlania listy, musimy zrozumieć mechanizm działania listy rozwijalnej oraz obiektów typu Spinner. W tym krótkim artykule znajdziemy kilka przykładów oraz dalszych odnośników związanych ze stosowaniem obiektów typu Spinner w Androidzie.
- [www.androidicons.com](http://www.androidicons.com) — kilka ikon wykorzystywanych w naszym projekcie zostało zapożyczonych z tej witryny. Ikony te są objęte licencją Creative Commons 3.0.
- [www.androidbook.com/item/3302](http://www.androidbook.com/item/3302) — przyjemne dla oka układy graficzne. Zamieściliśmy pod tym adresem kilka przykładowych kodów źródłowych oraz informacji dotyczących prostych układów graficznych.
- <http://developer.android.com/reference/android/view/MenuItem.html> — dokumentacja klasy MenuItem. Znajdziemy tu informacje o dołączaniu elementów menu jako przycisków do paska działania.
- <http://developer.android.com/guide/topics/resources/menu-resource.html> — znajdziemy tu listę elementów języka XML dostępnych w procesie definiowania elementów menu jako przycisków paska działania.
- [ftp://ftp.helion.pl/przyklady/and3ta.zip](http://ftp.helion.pl/przyklady/and3ta.zip) — pod tym adresem znajdziemy projekty testowe utworzone specjalnie na potrzeby książki. Katalog zawierający projekty omawiane w niniejszym rozdziale nosi nazwę *ProAndroid3\_R30\_PasekDzialania*.

## Podsumowanie

Jak widać, paski działania nie stanowią wcale takiej tajemnicy. Są one znany paradygmatem, stosowanym podczas tworzenia aplikacji biurowych. Początkującym programistom zachowanie klasy, która w zależności od wyboru jednego z trzech trybów zyskuje odmienne właściwości, może się wydawać problematyczne. Nigdy nie wiadomo, czy posiadany podzbiór klas wykona zamierzone zadanie. Jednak różnica pomiędzy tymi trybami jest tak mała, że ich umieszczenie w obrębie jednej klasy naprawdę nie było złym pomysłem.

Wydaje się, że jedną z intencji wprowadzenia pasków działania było skierowanie procesu projektowania w stronę modelu nawigacji sieciowej.

Wydaje się także, że projektanci systemu Android starają się powiązać pasek działania z fragmentami w celu osiągnięcia pożądanej jednorodności interfejsu użytkownika. Jeżeli istnieje potrzeba wykorzystania w aplikacji kilku wymiennie używanych aktywności, zalecane jest rozważenie, czy taki efekt można osiągnąć, wprowadzając fragmenty, a nie dodatkowe aktywności. Fragmenty mają wiele udogodnień, zwłaszcza związanych z zarządzaniem ich stanem w przypadku obracania urządzenia, a tym samym zmiany jego konfiguracji. Aktywność przechowująca fragmenty utrzymuje stan pomiędzy zmianami konfiguracji. Fragmenty zostały omówione dokładniej w rozdziale 29.

W tym rozdziale zaprezentowaliśmy także jeden ze sposobów wprowadzania jednolitości projektowej za pomocą bazowej aktywności. Podobny efekt można również osiągnąć, wprowadzając delegację zamiast dziedziczenia. Istnieje także możliwość zapożyczenia znanych wzorców, stosowanych do tworzenia głównych stron witryn sieciowych, oraz sprawdzenia, jak w tej roli spisują się klasy środowiska Android SDK.

Funkcja paska działania została wprowadzona dopiero w wersji 3.0 Androida. Na bieżącą chwilę nie wiadomo, czy zostaną utworzone odpowiednie biblioteki przeznaczone dla starszych wersji systemu.

Istnieją również pewne rozbieżności pomiędzy dokumentacją a faktycznym stanem interfejsów API. W dokumentacji znalazło się stwierdzenie, że istnieją tylko trzy tryby wyświetlania paska działania, jednak w interfejsie można znaleźć dodatkowy tryb, zwany trybem rozwijalnego menu (ang. *dropdown navigation mode*). W trakcie testowania zachowywał się on dokładnie tak samo jak tryb wyświetlania listy, z tą jedną różnicą, że zniknął tytuł aktywności.

Możemy również za pomocą flag kontrolować elementy wyświetlane w pasku działania. Jest to dość proste rozwiązanie, zatem wystarczy zajrzeć do dokumentacji omawianego interfejsu API.

# Dodatkowe zagadnienia związane z wersją 3.0 systemu

Po przeczytaniu trzydziestu długich rozdziałów ciągle istnieją zagadnienia związane z wersją 3.0 systemu, których jeszcze nie zdążyliśmy poruszyć! W tym ostatnim rozdziale zajmiemy się tematyką usprawnień wprowadzonych do funkcji widżetów ekranu startowego oraz nowego interfejsu funkcji przeciągania.

W wersji 3.0 systemu wprowadzono znaczne usprawnienia funkcji widżetów. Dzięki nim możemy teraz dodawać na ekranie startowym widżety oparte na listach. Interfejs funkcji przeciągania jest z kolei całkowitą nowością. Za jego pomocą utworzymy bogate interfejsy użytkownika, przypominające budowę te spotykane w komputerach osobistych. Obydwa te zagadnienia omówimy bardzo szczegółowo.

## Widżety ekranu startowego oparte na listach

W rozdziale 22. omówiliśmy mechanizm działania widżetów w wersji 2.3 Androida i starszych. W wersji 3.0 systemu zostały wprowadzone znaczne modyfikacje tej funkcji; istnieje duże prawdopodobieństwo, że zmiany te zostaną również wprowadzone w kolejnej wersji systemu, zoptymalizowanej także pod kątem smartfonów.

Wartościowym wstępem do tego rozdziału byłoby powtórzenie sobie treści rozdziału 22., dzięki czemu łatwiej będzie dostrzec różnice pomiędzy starą a nową wersją widżetów. Jednak w tym rozdziale zamieściliśmy całościowy opis funkcji widżetów, zatem Czytelnik może się zapoznać z informacjami o nich nawet bez uprzedniego przeczytania rozdziału 22.

Jak wiemy ze wspomnianego rozdziału 22., rdzeniem widżetów ekranu startowego są widoki zdalne. Widżet ekranu startowego jest w istocie widokiem zdalnym rysowanym na ekranie startowym. Widok zdalny jest całkowicie odosobniony od mieszczących się w nim danych, podobnie jak strona WWW jest oddzielona od serwera.

W rozdziale 22. zamieściliśmy listę układów graficznych i widżetów, które mogą stanowić część widoku zdalnego. Widoki zbiorcze, takie jak listy czy siatki, nie były klasyfikowane jako elementy składowe widżetów w wersji 2.3 systemu. Sytuacja uległa zmianie w wersji 3.0, dzięki czemu uzyskujemy więcej możliwości na ekranie startowym. Wydanie 3.0 oferuje również miniaturową architekturę pozwalającą na asynchroniczne wczytywanie i wyświetlanie danych w tego typu zbiorczych widżetach. W wykorzystaniu tej właściwości przydadzą nam się nowe klasy i metody.

Zajmiemy się najpierw teoretycznym omówieniem tych zagadnień, a następnie, w celu utrwalenia przekazanej wiedzy, zademonstrujemy je na działających przykładach. Rozpoczniemy od omówienia nowych widoków zdalnych, dostępnych w wersji 3.0 systemu.

## Nowe widoki zdalne w wersji 3.0 systemu

W wersji 2.3 Androida dostępnych jest 13 układów graficznych i widżetów stanowiących elementy składowe widoku zdalnego:

- `AbsoluteLayout`,
- `FrameLayout`,
- `LinearLayout`,
- `RelativeLayout`,
- `AnalogClock`,
- `Button`,
- `Chronometer`,
- `ImageButton`,
- `ProgressBar`,
- `ViewFlipper`,
- `DateTimeView`,
- `ImageView`,
- `TextView`.

Część z tych układów graficznych i widoków (na przykład `AbsoluteLayout`) mogła zostać wycofana z użycia, powinniśmy więc najpierw to sprawdzić, zanim użyjemy którejś z wymienionych klas w kodzie. Być może Czytelnik zapyta, dlaczego powyższa lista jest taka istotna. Czy wykorzystujemy bezpośrednio klasy tych widoków bądź układów graficznych w procesie tworzenia widoku zdalnego?

Okazuje się, że klasa `RemoteViews` nie może zostać utworzona poprzez jawne przekazanie obiektów wymienionych powyżej klas. Niedopuszczalne jest również bezpośrednie dołączenie któregokolwiek z tych obiektów do tej klasy. Zamiast tego obiekt `RemoteViews` tworzy się poprzez przekazanie pliku układu graficznego do jego konstruktora. Powyższa lista jest o tyle istotna, że węzłami takiego pliku mogą być wyłącznie wymienione elementy.

Zaprezentowana poniżej lista stanowi poszerzony zbiór 16 elementów — układów graficznych, widżetów i widoków — dostępnych w wersji 3.0 Androida:

- `FrameLayout`,
- `LinearLayout`,
- `RelativeLayout`,

- AnalogClock,
- Button,
- Chronometer,
- ImageButton,
- ProgressBar,
- ListView,
- GridView,
- StackView,
- TextView,
- DateTimeView,
- ImageView,
- AdapterViewFlipper,
- ViewFlipper.

W kolejnych wersjach systemu do tej listy będą zapewne dodawane kolejne elementy. Głównym czynnikiem decydującym o tym, czy dany obiekt stanowi element składowy widoku zdalnego, jest jego obecność w interfejsie `RemoteViews.RemoteView`.

Mając tą wiedzę, wykorzystajmy środowisko Eclipse do sprawdzenia, które klasy uwzględnione w danym projekcie mogą być częścią widoku zdalnego. W tym celu:

1. Wprowadź w kodzie źródłowym instrukcję `import` odnoszącą się do interfejsu `RemoteView`.
2. Zaznacz nazwę tego interfejsu.
3. Kliknij nazwę interfejsu prawym przyciskiem myszy i przejdź do zakładki *References*.
4. Wybierz odniesienia do tego interfejsu.

Zostanie wyświetlona lista klas przechowywanych w interfejsie `RemoteView`.

## Praca z listami stanowiącymi część widoku zdalnego

W rozdziale 22. zajmowaliśmy się zestawem klas służących do tworzenia widżetów ekranu startowego. Podstawowymi klasami są w tym przypadku `AppWidgetProvider`, `AppWidgetManager`, `RemoteViews` oraz aktywność służąca do konfiguracji klasy `AppWidgetProvider` za pomocą początkowych parametrów.

Omówimy teraz w skrócie podstawową koncepcję działania widżetów ekranu startowego (pozostałe informacje przedstawione w tym podrozdziale staną się przez to nieco łatwiejsze do zrozumienia). Klasa `AppWidgetProvider` jest odbiorcą komunikatów wywoływanym co pewien czas, który jest z góry zdefiniowany w pliku konfiguracyjnym. Odbiorca ten następnie wczytuje wystąpienie klasy `RemoteViews` na podstawie pliku układu graficznego. Obiekt `RemoteViews` jest dalej przekazywany do odpowiedzialnej za jego wyświetlanie klasy `AppWidgetManager`.

Możemy ewentualnie poinformować system, że istnieje aktywność, która powinna zostać wywołana przed pierwszym umieszczeniem widżetu na ekranie startowym. Taka aktywność konfiguracyjna służy do ustawienia parametrów początkowych widżetu.

Możemy skonfigurować również zdarzenia `onClick` wobec zdalnych widoków widżetu, umożliwiając tym samym uruchomienie odpowiednich intencji opartych na zdarzeniach. Intencje te mogą wywoływać dowolne składniki, w tym takie jak wysyłanie komunikatów do odbiorcy `AppWidgetProvider`.

W ogólnej perspektywie są to jedyne czynności przeprowadzane w widżetach ekranu startowego. Resztę stanowią jedynie mechanizmy implementacji oraz różne wariacje omówionych podstawowych koncepcji.

Jednak aż do wersji 2.3 Androida widoki bazujące na liście nie były klasyfikowane jako elementy widoku zdalnego i nie istniał skuteczny mechanizm wypełniania widoków zdalnych przypominających listę. W wersji 3.0 systemu dodano następujące klasy, służące do obsługi widoków zdalnych w postaci listy:

- `RemoteViewsFactory` — klasa ta umożliwia zapełnienie widoku zdalnego w podobny sposób, jak adaptery listy zapełniają standardowe widoki listy. Jest to klasa otaczająca adapter widoku listy, dzięki czemu w sposób asynchroniczny dostarcza pojedyncze widoki zdalne do widoku zdalnego przyjmującego postać listy.
- `RemoteViewsService` — mamy tu do czynienia z usługą odpowiedzialną za przekazywanie klasy `RemoteViewsFactory` do obiektu `RemoteViews`. Zadaniem klasy `AppWidgetProvider` jest połączenie takiej usługi ze zdalnym widukiem w postaci listy. Dokonujemy tego poprzez dołączenie intencji wywołującej tę usługę do wspomnianego widoku zdalnego. Usługa ta umożliwia przedłużenie czasu istnienia procesu przechowującego klasę `AppWidgetProvider`. W przeciwnym wypadku po przekazaniu odbiorcy komunikatów proces ten zostanie odzyskany. W rozdziale 14. została omówiona symbiotyczna relacja pomiędzy odbiorcami komunikatów a długoterminowymi usługami.

W celu zapewnienia obsługi widoków zdalnych w postaci listy w wersji 3.0 systemu znalazły się następujące nowe metody interfejsu API:

- `RemoteViews.setPendingIntentTemplate()` — metoda ta pozwala na ustawienie szablonu intencji oczekującej wobec zdalnego widoku w celu reagowania na zdarzenia kliknięcie poszczególnych elementów listy. Zajmiemy się koncepcją „szablonu” po omówieniu pozostałych szczegółów.
- `RemoteViews.setOnClickListener()` — jest ona ustawiana wobec pojedynczych elementów listy i ściśle współpracuje z powyższą metodą.

Dzięki łącznemu wykorzystaniu tych dwóch dodatkowych metod możemy reagować na kliknięcia poszczególnych elementów listy zawartej w widoku zdalnym. Metody te zaprojektowano w taki sposób, aby należało ustawać jak najmniej intencji oczekujących.

W trakcie omawiania przykładowej aplikacji przeanalizujemy dokładnie wymienione klasy i metody. Poniżej pokazaliśmy, w jaki sposób możemy je zastosować podczas pracy z widokami listy umieszczonymi w widżecie ekranu startowego. Podczas przeglądania poszczególnych etapów warto zaglądać do ogólnego omówienia mechanizmu działania widżetów ekranu startowego (zawarliśmy je na początku podróżdziału):

1. **Przygotuj zdalny układ graficzny.** Utwórz odpowiedni układ graficzny, w którym znajdzie się widok listy. Zdalny układ graficzny nie różni się od standardowego układu, lecz możemy w nim umieścić jedynie widoki dozwolone dla widoku zdalnego. Zasada jest taka sama jak w przypadku wszelkich innych widżetów ekranu startowego (co zostało wyraźnie powiedziane w rozdziale 22.).

2. **Wczytaj zdalny układ graficzny.** W metodzie `onUpdate()` dostawcy widżetu wczytaj utworzony uprzednio składowy układ graficzny w postaci widoku zdalnego. Ten etap pracy także nie różni się od tworzenia standardowego widżetu. Połącz następnie zdalny widok z jego usługą, dzięki czemu widok listy zostanie zapełniony przez przekazaną klasę fabrykującą.
3. **Skonfiguruj usługę** `RemoteViewsService`. Zlokalizuj zdalny widok za pomocą jego identyfikatora i wprowadź intencję, która będzie wywoływała usługę widoku zdalnego. Usługa ta następnie przekazuje klasę `RemoteViewsFactory` do widoku listy, dzięki czemu zdalny widok zostaje zapełniony.
4. **Skonfiguruj klasę** `RemoteViewsFactory`. Usługa widoku zdalnego musi przekazać klasę `RemoteViewsFactory`, która będzie zapełniała widok listy.
5. **Skonfiguruj zdarzenia kliknięć.** Częścią procesu konfiguracji klasy `AppWidgetProvider` jest ustawienie szablonu intencji oczekującej `onClick`, dzięki czemu aplikacja będzie odpowiadała na tę intencję. Musisz jednak jednocześnie skonfigurować odpowiedzi na pojedyncze kliknięcie dla każdego elementu listy za pomocą klasy `RemoteViewsFactory`. Wynika to z faktu, że elementy listy w widoku zdalnym są zapełniane przez tę klasę fabrykującą.
6. **Odpowiedz na zdarzenia kliknięć.** Trzeba określić obiekt, który będzie odpowiadał na zdarzenia `onClick` ustanawiane wobec widoków listy. Odbiorcą tych zdarzeń może zostać klasa `AppWidgetProvider`. Musisz przygotować odbiorcę komunikatów, który będzie otrzymywał i zdarzenia `onClick` pochodzące ze zdalnych widoków i odpowiadał na nie.

Przyjrzyjmy się implementacji każdego etapu w przykładowym kodzie.

## Przygotowanie zdalnego układu graficznego

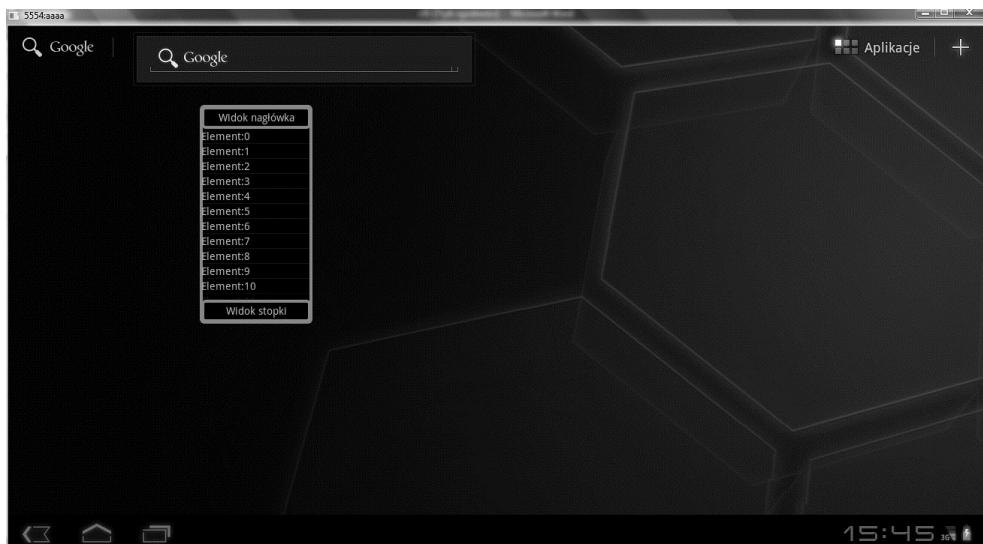
Jak zostało stwierdzone w poprzednim podpunkcie, układ graficzny widoku zdalnego pełniący rolę widżetu ekranu startowego może teraz przechowywać widok listy. Na listingu 31.1 został zaprezentowany przykładowy układ graficzny, możliwy do zastosowania w widoku zdalnym i zawierający widok listy.

**Listing 31.1.** Plik zdalnego układu graficznego zawierający kontrolkę ListView

```
<?xml version="1.0" encoding="utf-8"?>
<!-- /res/layout/test_list_widget_layout.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="150dp"
    android:layout_height="match_parent"
    android:background="@drawable/box1">
<TextView
    android:id="@+id/listwidget_header_textview_id"
    android:layout_width="fill_parent"
    android:layout_height="30dp"
    android:text="Widok nagłówka"
    android:background="@drawable/box1"
    android:gravity="center"
    android:layout_weight="0"/>
<FrameLayout
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:layout_gravity="center">
        <ListView android:id="@+id/listwidget_list_view_id"
            android:layout_width="match_parent"
            android:layout_height="match_parent"/>
        <TextView
            android:id="@+id/listwidget_empty_view_id"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:gravity="center"
            android:visibility="gone"
            android:textColor="#ffffffff"
            android:text="Widok zawierający puste rekordy"
            android:textSize="20sp" />
    </FrameLayout>
    <TextView
        android:id="@+id/listwidget_footer_textview_id"
        android:layout_width="fill_parent"
        android:layout_height="30dp"
        android:text="Widok stopki"
        android:background="@drawable/box1"
        android:gravity="center"
        android:layout_weight="0"/>
</LinearLayout>
```

Na listingu 31.1 każdy węzeł XML reprezentuje poprawny widok zdalny. Omawiany układ graficzny jest wyświetlany w taki sposób, że w postaci widżetu ekranu startowego będzie się prezentował tak jak na rysunku 31.1.



Rysunek 31.1. Ekran startowy zawierający widżet z widokiem listy

Na wzorzec układu graficznego z rysunku 31.1 składa się prosty nagłówek, część główna i stopka. Nagłówek i stopka posiadają stałą wysokość; w tym przypadku została ona ustalona na 30 dp. Chcemy ponadto, by wysokość części głównej była taka, aby zajmowała pozostałą część dostępnej wysokości widżetu. Osiągniemy to poprzez wprowadzenie wartości 0 w atrybutach `android:layout_weight` nagłówka i stopki. Dla części głównej wprowadzamy wartość 1 atrybutu `android:layout_weight` oraz `match_parent` w `android:layout_height`.

Węzeł `<framelayout>` stanowiący część główną widżetu wymaga krótkiego wyjaśnienia. Ze wszystkich elementów potomnych wybiera on tylko jeden jako widok. W tym przypadku będzie to `<listview>`, jeżeli lista będzie zawierała jakieś dane. W przypadku pustego widoku listy będzie stosowany pusty widok tekstowy. Konfigurujemy to w klasie `RemoteViewsFactory`.

W tym układzie znajdziemy także graficzny niestandardowy obiekt rysowany (`@drawable/box1`), służący do zaokrąglenia rogów. Listing 31.2 zawiera treść pliku `box1.xml`, który należy umieścić w podkatalogu `/res/drawable`.

### **Listing 31.2. Plik res/drawable/box1.xml**

---

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <stroke android:width="4dp" android:color="#888888" />
    <padding android:left="2dp" android:top="2dp"
        android:right="2dp" android:bottom="2dp" />
    <corners android:radius="4dp" />
</shape>
```

---

Skoro już mamy przykładowy układ graficzny widżetu ekranu startowego, zastanówmy się, w jaki sposób możemy go wczytać do widoku zdalnego.

## **Wczytanie zdalnego układu graficznego**

W przypadku widżetu ekranu startowego widok zdalny zostaje wczytany i wyświetlony za pomocą metody zwrotnej `onUpdate()` klasy `AppWidgetProvider`. Zaprezentowaliśmy takie rozwiązanie na listingu 31.3.

---

### **Listing 31.3. Wczytywanie zdalnego układu graficznego w metodzie onUpdate()**

```
public void onUpdate(Context context,
                     AppWidgetManager appWidgetManager,
                     int[] appWidgetIds)
{
    int N = appWidgetIds.length;
    for (int i=0; i<N; i++)
    {
        int appWidgetId = appWidgetIds[i];

        RemoteViews rv =
new RemoteViews(context.getPackageName(),
                R.layout.test_list_widget_layout);

        rv.setEmptyView(R.id.listwidget_list_view_id,
                      R.id.listwidget_empty_view_id);

        //Aktualizujemy wystąpienie tego widżetu
```

```
    appWidgetManager.updateAppWidget(appWidgetId, rv);
}
super.onUpdate(context, appWidgetManager, appWidgetIds);
}
```

---

Zwróćmy uwagę, że obiekt `RemoteViews` został skonstruowany za pomocą identyfikatora pliku układu graficznego, który definiuje cały widżet. Jest to ten sam układ graficzny co zaprezentowany na listingu 31.1. Pobieramy następnie wynikowy obiekt `RemoteViews` i ustanawiamy pusty widok dla określonego zasobu listy (definiowanego za pomocą identyfikatora) wewnątrz tego układu graficznego.

Na listingu 31.3 plik układu graficznego posiada identyfikator:

`R.layout.test_list_widget_layout`

Zasób widoku listy wewnątrz tego układu graficznego posiada identyfikator:

`R.id.listwidget_list_view_id`

Pusty widok definiowany dla tego zasobu posiada identyfikator:

`R.id.listwidget_empty_view_id`

Na listingu 31.4 identyfikatory te służą do konstruowania zdalnego widoku oraz ustawienia pustego widoku dla jednego z widoków listy.

---

#### **Listing 31.4. Wczytywanie zdalnych widoków**

---

```
RemoteViews rv =
new RemoteViews(context.getPackageName(),
    R.layout.test_list_widget_layout);
rv.setEmptyView(R.id.bdw_list_view_id,
    R.id.empty_view_id);
```

---

## **Konfigurowanie usługi `RemoteViewsService`**

Do tej pory udało nam się wczytać zdalne widoki w metodzie `onUpdate()` klasy `AppWidgetProvider`. Musimy teraz powiązać zdalny widok listy z usługą, która będzie mogła przekazać adaptera zapełniający listę danymi.

Dlaczego to musi być usługa? Dlaczego nie powiązaliśmy klasy fabrykującej bezpośrednio z widokiem listy?

Przyczyna jest taka, że klasa `AppWidgetProvider` jest odbiorcą komunikatów, zatem metoda `onUpdate()` dostawcy widżetów jest ograniczona czasowo przez tego odbiorcę. Aby uniknąć potencjalnych problemów, do zapełniania widoku listy w Androidzie 3.0 przeznaczono oddzielną usługę, wywodzącą się z pakietu `android.widget.RemoteViewsService`. Usługa ta jest odpowiedzialna za przekazywanie adaptera wykonującego operację zapełniania listy. Adapter musi należeć do typu `RemoteViewsService.RemoteViewsFactory`. Jest to procedura, która w schematyczny sposób dąży do ostatecznego powiązania widoku listy z klasą fabrykującą.

Na listingu 31.5 prezentujemy przykładowy sposób pisania kodu usługi widoku zdalnego oraz mechanizm przekazywania przez nią klasy fabrykującą.

**Listing 31.5.** Przykład usługi RemoteViewsService

---

```
public class TestRemoteViewsService
extends android.widget.RemoteViewsService
{
    @Override
    public RemoteViewsFactory onGetViewFactory(Intent intent)
    {
        return new TestRemoteViewsFactory(
            this.getApplicationContext(), intent);
    }
}
```

---

Zwróćmy uwagę na następujące cechy listingu 31.5:

- Musimy dziedziczyć po klasie RemoteViewsService.
- Musimy zdefiniować i otrzymać klasę fabrykującą RemoteViewsFactory. Wkrótce zajmiemy się tym zagadnieniem.

Skoro mamy do czynienia z usługą, dziedziczona klasa RemoteViewsService (w naszym przypadku TestRemoteViewsService) musi zostać również zadeklarowana w pliku manifeście. Stosowny przykład został pokazany na listingu 31.6.

**Listing 31.6.** Deklarowanie usługi RemoteViewsService w pliku manifeście

---

```
<!-- Usługa stosowana w klasie RemoteViews w procesie tworzenia widżetu zbiorczego -->
<service android:name=".TestRemoteViewsService"
    android:permission="android.permission.BIND_REMOTEVIEWS"
    android:exported="false" />
```

---

Po zdefiniowaniu takiej usługi zdalnych widoków możemy dołączyć ją do zdalnego widoku listy za pomocą kodu zamieszczonego na listingu 31.7 (przypominamy, że kod ten jest umieszczony w metodzie onUpdate() klasy AppWidgetProvider).

**Listing 31.7.** Powiązanie usługi RemoteViewsService z obiektem RemoteViewList

---

```
final Intent intent =
    new Intent(context, TestRemoteViewsService.class);
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
    appWidgetId);
intent.setData(
    Uri.parse(
        intent.toUri(Intent.URI_INTENT_SCHEME)));
rv.setRemoteAdapter(appWidgetId,
    R.id.listwidget_list_view_id, intent);
```

---

W kodzie z listingu 31.7 najpierw tworzymy jawną intencję poprzez powiązanie z nią klasy RemoteViewsService. Następnie wstawiamy do tej intencji dodatkowe dane, pozwalające na zidentyfikowanie widżetu, dla którego będzie wywoływana usługa. Kolejnym etapem jest przeprowadzenie tej dziwnej operacji odnoszenia się intencji do samej siebie — wczytania porcji danych do intencji za pomocą identyfikatora tej intencji. W ten sposób intencja staje się

niepowtarzalna, ponieważ dodatkowe informacje są dołączane do jej danych. Bez tych dodatkowych danych intencje nie są niepowtarzalne. Gdy już to uzyskamy, możemy dołączyć intencję do zdalnego widoku listy poprzez wywołanie metody `setRemoteAdapter()` i przekazanie jej identyfikatora widoku listy.

## Konfigurowanie klasy RemoteViewsFactory

Chociaż do procesu zapełniania listy wskazaliśmy usługę `RemoteViewsService`, ostatecznie to zadanie realizuje klasa `RemoteViewsFactory`. Aby zapełnić widok listy, najpierw trzeba zaimplementować ten interfejs (w rozdziale 6. znajdziemy informacje dotyczące kontrolek listy oraz adapterów listy).

Na listingu 31.8 widzimy sygnatury metod należących do klasy, która implementuje ten interfejs fabrykujący.

**Listing 31.8.** Kontrakt klasy `RemoteViewsFactory`

---

```
class TestRemoteViewsFactory
implements RemoteViewsService.RemoteViewsFactory
{
    public TestRemoteViewsFactory(Context context, Intent intent);
    public void onCreate();
    public void onDestroy();
    public int getCount();
    public RemoteViews getViewAt(int position);
    private void loadItemOnClickExtras(RemoteViews rv, int position);
    public RemoteViews getLoadingView();
    public int getViewTypeCount();
    public long getItemId(int position);
    public boolean hasStableIds();
    public void onDataSetChanged();
}
```

---

Zajmijmy się omówieniem tych metod oraz związanych z nimi wymagań. Zaczniemy od konstruktora.

## Konstruktor klasy `RemoteViewsFactory`

Konstruktor przyjmuje tutaj dwa argumenty (być może Czytelnik będzie miał do czynienia z inną klasą fabrykującą przyjmującą inne argumenty). W przypadku widżetów klasa ta jest tworzona przez usługę `RemoteViewsService` (co zostało pokazane na listingu 31.5), zatem kontekst stanowi tu dostawca widżetu, który sam w sobie jest odbiorcą komunikatów.

Drugim argumentem konstruktora jest intencja. Jest to ta sama intencją, która była wykorzystana w procesie wywołania usługi widoku zdalnego. Po utworzeniu tej intencji (listing 31.7) i dołączeniu jej do zdalnego widoku pozostawia ona zazwyczaj dodatkową wartość stanowiącą identyfikator widżetu.

Obydwie te wartości (kontekst oraz intencja) mogą być utrzymywane w konstruktorze jako zmienne lokalne, dzięki czemu kolejne metody będą mogły z nich korzystać. Przydatne zwłaszcza okazuje się pobranie identyfikatora widżetu z intencji i zachowanie go w formie zmiennej lokalnej.

### **Metoda zwrotna onCreate()**

Sygnaturą metody `onCreate()` jest:

```
Public void onCreate()
```

Zgodnie z wzorcem występującym wśród wielu składników Androida klasa `RemoteViewsFactory` zawiera metody `onCreate()` oraz `onDestroy()`. Dokumentacja sugeruje, że metoda `onCreate()` jest wywoływana przez klienckiego widoku zdalnego w momencie pierwszego utworzenia klasy. Zgodnie z dalszymi informacjami wspomniana klasa fabrykująca może być współdzielona przez wiele adapterów widoków zdalnych, w zależności od przekazanej intencji.

Jednak wzorzec ten nie odnosi się całkowicie do klasy `RemoteViewsFactory`, ponieważ — w przeciwieństwie do składowej aktywności lub składowej usługi — proces tworzenia tej klasy pozostaje całkowicie pod kontrolą programisty. Programista może przeprowadzić proces inicjalizacji w samym konstruktorze. Nie zostało w dokumentacji jasno stwierdzone, czy ten fabrykujący obiekt jest przechowywany w pamięci przez szkielet interfejsu widżetów w powiązaniu z intencją wywołującą usługę zdalnego widoku. Komunikaty dziennika wskazują, że metoda `onCreate()` jest z pewnością wywoływana. Mamy zatem możliwość inicjalizacji również tutaj, a nie wyłącznie w konstruktorze.

### **Metoda zwrotna onDestroy()**

Sygnatura metody `onDestroy()` to:

```
Public void onDestroy()
```

Metoda ta stanowi dopełnienie metody `onCreate()`. Zgodnie z dokumentacją zostaje ona wywołana w momencie odlączenia ostatniego adaptora widoku zdalnego powiązanego z danym obiektem (lub klasą fabrykującą). Nie jest jednak do końca jasne, kiedy ta metoda zostaje wywołana; nie zauważyliszyliśmy jej wywołania ani po usunięciu pojedynczego widżetu z ekranu startowego, ani po usunięciu ostatniego widżetu.

### **Metoda zwrotna getCount()**

Sygnaturą metody `getCount()` jest:

```
public int getCount()
```

Metoda `getCount()` przekazuje całkowitą liczbę elementów z widoku listy. Metoda ta bardzo przypomina analogiczną metodę spotykaną w adapterach listy, które zostały omówione w rozdziale 6.

### **Metoda zwrotna getViewAt()**

Metoda `getViewAt()` posiada następującą sygnaturę:

```
public RemoteViews getViewAt(int position)
```

Zadaniem tej metody jest przekazanie zdalnego widoku zgodnego z pozycją w widoku listy. Zazwyczaj w tej metodzie wczytujemy dostosowany układ graficzny dla tego widoku zdalnego na danej pozycji, a następnie wprowadzamy w tym widoku wartości, gdzie pozycja jest wskaźnikiem wczytywania odpowiednich danych. Na listingu 31.9 znajduje się przykład wczytywania pojedynczego układu graficznego dla elementu listy.

**Listing 31.9.** Wczytywanie układu graficznego dla pojedynczego elementu listy

```
RemoteViews rv =  
    new RemoteViews(  
        this.mContext.getPackageName(),  
        R.layout.list_item_layout);
```

---

Układ graficzny, do którego odnosimy się na listingu 31.9, może być zdefiniowany tak, jak zostało to zaprezentowane na listingu 31.10.

**Listing 31.10.** Układ graficzny utworzony na potrzeby pojedynczego elementu listy

```
<?xml version="1.0" encoding="utf-8"?>  
<TextView xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/textview_widget_list_item_id"  
    android:layout_width="fill_parent"  
    android:layout_height="wrap_content"  
    android:text="Tymczasowy tekst"  
/>
```

---

Po wczytaniu zdalnego widoku (listing 31.9) możemy przekazać go do wywołującego widoku listy, aby został narysowany. Możemy tu również ustawić reakcję tego konkretnego widoku listy na zdarzenie kliknięcia.

**Metoda zwrotna getLoadingView()**

Sygnatura metody `getLoadingView()` jest następująca:

```
public RemoteViews getLoadingView()
```

Metoda ta przekazuje niestandardowy widok wczytywania, który pojawia się pomiędzy wywołaniem metody `getViewAt(position)` a jej powrotem. Aby skorzystać z domyślnego widoku wczytywania, wprowadzamy tu wartość `null`.

**Metoda zwrotna getViewTypeCount()**

Sygnatura metody `getViewTypeCount()` jest następująca:

```
public int getViewTypeCount()
```

Jeżeli zdalny widok listy posiada tylko jeden typ widoku potomnego, metoda ta powróci z wartością 1. Jeżeli będzie dostępna większa liczba typów widoków, przekaże ona wartość równą liczbie dostępnych typów.

**Metoda zwrotna getItemId()**

Sygnatura metody `getItemId()` to:

```
public long getItemId(int position)
```

Metoda ta przekazuje właściwy identyfikator elementu znajdującego się na danej pozycji w widoku listy. Metoda ta bardzo przypomina analogiczną metodę spotykaną w adapterach listy, które zostały omówione w rozdziale 6.

### **Metoda zwrotna hasStableIds()**

Sygnaturą metody `hasStableIds()` jest:

```
public boolean hasStableIds()
```

Metoda ta powinna powracać z wartością `true`, w przypadku gdy identyfikator obiektu z metodą `getItemId()` wskazuje ten sam obiekt. Metoda ta bardzo przypomina analogiczną metodę spotykaną w adapterach listy, które zostały omówione w rozdziale 6.

### **Metoda zwrotna onDataSetChanged()**

Sygnatura metody `onDataSetChanged()` jest następująca:

```
public void onDataSetChanged()
```

Metoda ta jest wywoływana, gdy klasa `AppWidgetManager` otrzyma informację o zmianie wprowadzonej w widżecie, który przechowuje zdalny widok listy. Wywołanie menedżera widżetów dotrze w końcu do klasy fabrykującej w postaci metody `onDataSetChanged()`. W odpowiedzi musimy skonfigurować wykorzystywane dane, dzięki czemu pozostałe metody zwrotne, takie jak `getViewAt()` albo `getCount()`, będą mogły zareagować na nowe informacje. Zgodnie z dokumentacją dozwolone są w tym przypadku długotrwałe operacje służące do konfigurowania danych.

Na tym zakończymy dyskusję dotyczącą uwidocznienia widoku listy w widżecie. Zajmijmy się teraz zagadnieniem dołączenia zdarzeń kliknięcia do widoku listy, a nawet do jego widoków potomnych.

## **Konfigurowanie zdarzeń onClick**

Konfigurowanie zdarzeń `onClick` dla zdalnych widoków listy jest procesem dwuetapowym. Najpierw rejestrujemy zdarzenie `onClick` wobec widoku listy w metodzie `onUpdate()` dostawcy widżetów. Następnie przeprowadzamy tę samą czynność wobec każdego elementu potomnego tej listy w metodzie `getViewAt()` klasy fabrykującej.

Na początku pokażemy, w jaki sposób rejestrować zdarzenia kliknięć w głównym widoku listy. Po skonfigurowaniu takiego zdarzenia w widoku zdalnym musimy spowodować uruchomienie intencji po kliknięciu widoku zdalonego. Ponieważ dostawca widżetu jest odbiorcą komunikatów, może stać się on obiektem docelowym tej intencji. Musimy następnie wprowadzić odpowiednie zastrzeżenia w tym dostawcy widżetu, co pozwoli na zmodyfikowanie metody zwrotnej `onReceive()` w taki sposób, że będzie możliwa obsługa tej intencji.

Fragment kodu z listingu 31.11 pokazuje, w jaki sposób konfigurujemy intencję zdarzenia `onClick`, gdzie obiektem docelowym jest dostawca widżetu.

---

#### **Listing 31.11. Utworzenie intencji umożliwiającej samoistne wywołanie dostawcy widżetu**

---

```
Intent onListClickIntent =
    new Intent(context, TestListWidgetProvider.class);
```

---

Zwróćmy uwagę na sposób konfiguracji nazwy klasy dostawcy widżetu, która staje się docelowym składnikiem intencji. Intencja ta zostanie dostarczona do dostawcy widżetu. Dostawca ten odpowiada jednak również na inne intencje pochodzące z innych działań związanych z widżetem. Aby odróżnić tę intencję od pozostałych, musimy ustawić dla niej jawnie działanie. Na listingu 31.12 został ukazany przykład.

**Listing 31.12.** Definiowanie unikatowego działania w dostawcy widżetu dla zdarzenia kliknięcia

```
onListClickIntent.setAction(  
    TestListWidgetProvider.ACTION_LIST_CLICK);
```

---

Oczywiście, działanie `TestListWidgetProvider.ACTION_LIST_CLICK` jest niestandardowe i najlepiej je definiować jako część dostawcy widżetu `TestListWidgetProvider`.

Ponieważ kliknięcia mogą występować wśród wielu wystąpień tego widżetu, musimy wczytać identyfikator określonego widżetu jako dodatkowe dane intencji wywołującej. Odpowiednie rozwiązanie prezentujemy na listingu 31.13.

**Listing 31.13.** Wczytywanie identyfikatora widżetu do intencji zdarzenia `onClick`

```
onListClickIntent.putExtra(  
    AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
```

---

Intencja jest już niemal gotowa do ustanowienia wobec zdarzeń `onClick` w zdalnym widoku listy. Musimy przeprowadzić w niej jeszcze jedną czynność. Gdy intencje mają pojawić się dopiero później, stają się intencjami oczekującymi. W rozdziałach 5. (dotyczącym intencji) oraz 15. (w którym omówiliśmy menedżery alarmów) Czytelnik znajdzie więcej informacji na temat intencji oczekujących.

Intencja oczekująca nie uwzględnia żadnych definiowanych w niej dodatkowych danych, chyba że informacje te decydują o jej unikalności. Te dodatkowe dane intencji nie są jednak brane pod uwagę, jeżeli dana intencja jest unikatowa. Aby rozwiązać ten problem, musimy wprowadzić w intencji metodę `toUri()`.

Metoda ta pobiera wszystkie dodatkowe dane intencji i generuje długi ciąg znaków reprezentujący intencję. Na końcu tego ciągu znaków znajdują się dane dodatkowe intencji. Wprowadzenie tego ciągu znaków jako porcji danych jest równoznaczne z nadaniem intencji unikalności. Wynika to z faktu, że te dane zostaną pobrane przez intencję głównie w celu jej odróżnienia od innych intencji. Na listingu 31.14 widzimy przykład definiowania niepowtarzalnej intencji za pomocą metody `toUri()`.

**Listing 31.14.** Zastosowanie metody `toUri()`

```
onListClickIntent.setData(  
    Uri.parse(  
        onListClickIntent.toUri(Intent.URI_INTENT_SCHEME)));
```

---

Po zdefiniowaniu unikalności intencji uzyskujemy dostęp do niezbędnej intencji oczekującej, co zostało zademonstrowane na listingu 31.15.

**Listing 31.15.** Utworzenie intencji oczekującej na komunikat

```
PendingIntent onListClickPendingIntent =  
    PendingIntent.getBroadcast(context, 0,  
        onListClickIntent,  
        PendingIntent.FLAG_UPDATE_CURRENT);
```

---

Na listingu 31.15 flaga FLAG\_UPDATE\_CURRENT oznacza, że po znalezieniu podobnej intencji zostaną zaktualizowane jedynie jej dodatkowe dane. Nieco później, kiedy będziemy omawiać sposób wykorzystywania intencji oczekującej przez widoki zdalne, wyjaśnimy dokładniej, dlaczego wspomniany proces może się okazać konieczny.

Po utworzeniu intencji oczekującej, na przykład takiej, jaką zaprezentowaliśmy na listingu 31.15, możemy ustanowić reakcję na kliknięcie widoku listy. Wykorzystujemy tutaj metodę setPendingIntentTemplate() do ustanowienia relacji pomiędzy intencją oczekującą a widokiem listy. Na listingu 31.16 widzimy przykład zastosowania metody setPendingIntentTemplate().

#### **Listing 31.16.** Zastosowanie metody setPendingIntentTemplate()

---

```
RemoteViews rv;
rv.setPendingIntentTemplate(R.id.listwidget_list_view_id,
    onListClickPendingIntent);
```

---

Pierwszym argumentem na listingu 31.16 jest identyfikator widoku listy umieszczonego w głównym układzie graficznym (listing 31.1). Drugi argument stanowi intencja oczekująca, którą przygotowywaliśmy w kodzie na listingach od 31.11 do 31.14. Zwróćmy uwagę, że w kodzie z listingu 31.16 wywołujemy **szablon intencji oczekującej**. Skąd się wziął wyraz „szablon” (ang. *template*)?

Zgodnie z informacjami zawartymi w dokumentacji zestawu SDK twórcy systemu Android nie zalecają tworzenia osobnych intencji oczekujących dla każdego wiersza listy. Zamiast tego należy raczej utworzyć intencję oczekującą dla całej listy, a następnie — w odpowiedzi na kliknięcia poszczególnych elementów listy — po prostu przesyłać dodatkowe dane tej intencji. Zadanie to jest łatwiejsze, jeśli utworzymy jedną intencję oczekującą na poziomie listy, a następnie prześlemy ją ponownie wraz z różnymi danymi dodatkowymi. Dlatego właśnie intencja oczekująca z listingu 31.15 posiada flagę definiującą aktualizowanie jej dodatkowych danych.

Przyjrzyjmy się teraz, w jaki sposób dodatkowe dane są dostarczane z poszczególnych elementów zdalnego widoku listy. Jak można się było spodziewać, operacja ta jest przeprowadzana w tym samym miejscu, w którym są tworzone te elementy widoku zdalnego, a dokładniej w metodzie getViewAt() klasy fabrykującej (listing 31.9). Na listingu 31.17 przedstawiliśmy mechanizm dołączania intencji zawierających dodatkowe dane do elementu listy po jego kliknięciu.

#### **Listing 31.17.** Dołączenie intencji zawierającej dodatkowe dane do elementu listy po jego kliknięciu

---

```
//Wczytuje zdalny widok elementu listy
RemoteViews listItemRv;

//Pobiera zupełnie nową intencję
Intent ei = new Intent();

//Dodaje do intencji zdefiniowane przez nas dodatkowe dane
ei.putExtra("com.androidbook.widgets.unikatowe_dane_dodatkowe_w_postaci_ciagu_znakow",
    "Pozycja klikniętego elementu:" + position);

//Dodatkowe dane zostają ustawione wobec zdalnego widoku listy
listItemRv.setOnClickFillInIntent(R.id.textview_widget_list_item_id, ei);
```

---

Kluczową metodą na listingu 31.17 jest `setOnClickListener()`. Umożliwia nam ona dostarczanie nowej intencji zawierającej zdefiniowane dane dodatkowe, które zostaną wczytane. Dane te zostaną pobrane przez wewnętrzną strukturę i nałożone na szablon intencji oczekującej, który został skonfigurowany jako część zdarzenia `onClick`.

Na listingu 31.17 pobraliśmy jedynie tekst z bieżącego wiersza, nieco go zmodyfikowaliśmy i wstawiliśmy w postaci dodatkowych danych. Dzięki widocznemu na tym listingu kodowi po kliknięciu elementu listy będącej częścią widżetu pojawi się intencja, która zostanie przesłana wraz z dodatkowymi danymi do odbiorcy komunikatów. Przyjrzymy się więc, w jaki sposób należy przygotować takiego odbiorcę, aby odczytać dane zdefiniowane dla każdego elementu listy.

## Odpowiedź na zdarzenia onClick

W szablonie intencji oczekującej, który zdefiniowano dla widoku listy (listing 31.16), zauważymy dwie następujące kwestie:

- Przywoływanym składnikiem jest sam dostawca widżetu.
- Przyporządkowano określone działanie, specyficzne dla tego dostawcy widżetu.

Dostawca widżetu musi w odpowiedzi:

1. Zadeklarować w postaci ciągu znaków rozpoznawalne działanie.
2. Przesłonić metodę `onReceive()` i zareagować na działanie wymienione w punkcie 1.

Na listingu 31.18 ukazaliśmy sposób definiowania unikatowego działania w postaci stałej reprezentowanej przez ciąg znaków.

### **Listing 31.18.** Definicja niestandardowego działania

---

```
public static final String ACTION_LIST_CLICK =  
    "com.androidbook.homewidgets.listclick";
```

---

Na listingu 31.19 widzimy mechanizm przesyłania metody `onReceive()`. Pokazaliśmy sposób testowania działania intencji oraz wywołania metody `dealwithListAction()`. Na końcu tej metody musimy wywołać metodę `onReceive()` bazowej klasy dla wszystkich pozostałych działań. Jeżeli tego nie zrobimy, sam widżet nie będzie odbierał żadnych działań.

### **Listing 31.19.** Przesłanianie metody `onReceive()`

---

```
@Override  
public void onReceive(Context context, Intent intent)  
{  
    if (intent.getAction()  
        .equals(TestListWidgetProvider.ACTION_LIST_CLICK))  
    {  
        //Nie jest to jedno z działań widżetu  
        //Jest to specyficzne działanie, które zostało tutaj skierowane  
        dealwithListAction(context,intent);  
        return;  
    }  
  
    //Upewnijmy się, że zostanie wywołana  
    super.onReceive(context, intent);  
}
```

---

Na listingu 31.20 prezentujemy metodę `dealwithListAction()`, w której odczytujemy dodatkowe dane, załadowane do intencji w kodzie z listingu 31.17.

#### **Listing 31.20.** Odpowiadanie na zdarzenie onClick elementu listy

---

```
public void dealwithListAction(Context context, Intent intent)
{
    String clickedItemText =
        intent.getStringExtra(
            TestListAdapterProvider.EXTRA_LIST_ITEM_TEXT);
    if (clickedItemText == null)
    {
        clickedItemText = "Błąd";
    }
    clickedItemText =
        clickedItemText
        + "Został kliknięty element:"
        + clickedItemText;

    Toast t =
        Toast.makeText(context,clickedItemText,Toast.LENGTH_LONG);
    t.show();
}
```

---

Na listingu 31.20 odczytaliśmy dodatkowe dane za pomocą uprzednio utworzonej stałej i wprowadziliśmy obiekt typu `Toast`. Metoda ta jest przetwarzana w głównym wątku, musimy więc upewnić się, że nie będą w niej przeprowadzane długotrwałe operacje (aspekt ten został dokładnie omówiony w rozdziale 14., dotyczącym długoterminowych usług).

Zakończymy w ten sposób część teoretyczną dotyczącą nowych funkcji wprowadzonych w widżetach zawierających widoki listy. Przetestujmy je teraz w działającej, przykładowej aplikacji. Większa część zaprezentowanego dotychczas kodu pochodzi właśnie z tego projektu, zatem jego zrozumienie nie powinno stanowić problemu.

## **Działający przykład**

### **— testowy widżet ekranu startowego oparty na liście**

Prezentowany w tym punkcie przykładowy widżet posłuży do zilustrowania omawianych koncepcji dotyczących widżetów opartych na listach. Po utworzeniu tego projektu ujrzymy widżet wyświetlający listę, który będzie można przeciągać po ekranie startowym. W czasie takiego przeciągania zobaczymy 20 elementów listy wypełnionych przykładowym tekstem. Po kliknięciu jednego z tych elementów na ekranie startowym zostanie wyświetlony obiekt typu `Toast`, zawierający tekst z wybranego pola listy.

Poniżej wymieniliśmy wszystkie potrzebne pliki:

- `TestListAdapterProvider.java` jest główną klasą; mamy tu do czynienia z testowym dostawcą widżetów, który implementuje widżet zawierający (pośród innych widoków) widok listy (listing 31.21).
- `TestRemoteViewsFactory.java` jest klasą zawierającą zbiór elementów, które — po wczytaniu przez dostawcę widżetów — zostaną wyświetcone w widoku listy (listing 31.22).

- *TestRemoteViewsService.java* stanowi usługę widoku zdalnego, która tworzy wystąpienie klasy *TestRemoteViewsFactory* (listing 31.23).
- *Layout/test\_list\_widget\_layout.xml* to główny układ graficzny całego widżetu wczytanego przez dostawcę widżetów (listing 31.1).
- *Layout/list\_item\_layout.xml* jest plikiem układu graficznego zdefiniowanego dla widoku pojedynczego elementu listy. Układ ten jest wczytywany przez klasę fabrykującą (listing 31.10).
- *Drawable/box1.xml* stanowi prostą klasę pomocniczą, pozwalającą na zaokrąglenie roгów w głównym układzie graficznym widżetu (listing 31.2).
- *Xml/test\_list\_appwidget\_provider.xml* to plik metadanych, służący do definiowania widżetu w Androidzie (listing 31.24).
- *AndroidManifest.xml*, czyli plik konfiguracyjny aplikacji, w którym definiujemy dostawcę widżetów oraz usługę zdalnego widoku (listing 31.25).

## Utworzenie testowego dostawcy widżetów

Proces tworzenia widżetu ekranu startowego rozpoczynamy od utworzenia dostawcy widżetów dziedziczącego po klasie *AppWidgetProvider* oraz przeładowania jego metody *onUpdate()* w celu dostarczenia widżetowi widoku. Proces ten został dokładnie objaśniony w rozdziale 22. W tym przykładzie nasz dostawca został nazwany *TestListWidgetProvider*. Na listingu 31.21 został umieszczony jego kod źródłowy opatrzony komentarzami.

**Listing 31.21.** *TestListWidgetProvider.java*

---

```
package com.androidbook.homewidgets.listwidget;

//  
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów  
//

public class TestListWidgetProvider extends AppWidgetProvider  
{  
    private static final String tag = "TestListWidgetProvider";  
  
    public static final String ACTION_LIST_CLICK =  
        "com.androidbook.homewidgets.listclick";  
  
    public static final String EXTRA_LIST_ITEM_TEXT =  
        "com.androidbook.homewidgets.list_item_text";  
  
    public void onUpdate(Context context,  
                        AppWidgetManager appWidgetManager,  
                        int[] appWidgetIds)  
    {  
        Log.d(tag, "Wywołana metoda onUpdate");  
        final int N = appWidgetIds.length;  
        Log.d(tag, "Liczba widżetów: " + N);  
        for (int i=0; i<N; i++)  
        {  
            int appWidgetId = appWidgetIds[i];  
            updateAppWidget(context, appWidgetManager, appWidgetId);  
        }  
    }  
}
```

```
        }
        super.onUpdate(context, appWidgetManager, appWidgetIds);
    }

    public void onDeleted(Context context, int[] appWidgetIds)
    {
        Log.d(tag, "Wywołana metoda onDeleted");
        super.onDeleted(context, appWidgetIds);
    }

    public void onEnabled(Context context)
    {
        Log.d(tag, "Wywołana metoda onEnabled");
        super.onEnabled(context);
    }

    public void onDisabled(Context context)
    {
        Log.d(tag, "Wywołana metoda onDisabled");
        super.onEnabled(context);
    }

    private void updateAppWidget(Context context,
                                 AppWidgetManager appWidgetManager,
                                 int appWidgetId)
    {
        Log.d(tag, "Wywołana metoda onUpdate dla widżetu:" + appWidgetId);

        final RemoteViews rv =
new RemoteViews(context.getPackageName(),
                R.layout.test_list_widget_layout);

        rv.setEmptyView(R.id.listwidget_list_view_id,
                      R.id.listwidget_empty_view_id);

        // Określa usługę dostarczającą dane do
        // zbiorczego widżetu. Zwrócić uwagę, że musimy
        // wstawić obiekt appWidgetId za pomocą danych,
        // w przeciwnym wypadku zostanie zignorowany.
        final Intent intent =
            new Intent(context, TestRemoteViewsService.class);
        intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                      appWidgetId);

        intent.setData(
            Uri.parse(
                intent.toUri(Intent.URI_INTENT_SCHEME)));
    }

    rv.setRemoteAdapter(appWidgetId,
                        R.id.listwidget_list_view_id, intent);

    // Konfiguruje metodę zwracającą widok listy.
    // Wymagana jest unikatowa intencja oczekująca
    // dla tego identyfikatora widżetu. Intencja wysyła do samej siebie wiadomość,
    // która zostanie odebrana w metodzie onReceive.
```

```
Intent onListClickIntent =
    new Intent(context,TestListWidgetProvider.class);

//Ustanawia takie działanie, dzięki któremu odbiorca będzie mógł odróżnić je
//od pozostałych działań powiązanych z widżetami.
onListClickIntent.setAction(
    TestListWidgetProvider.ACTION_LIST_CLICK);

//Odbiorca ten obsługuje wszystkie wystąpienia widżetu,
//zatem musimy wiedzieć, dla którego wystąpienia
//przeznaczony jest ten komunikat.
onListClickIntent.putExtra(
    AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);

//Definiujemy unikatowość intencji w trakcie przygotowywania
//intencji oczekującej.
//Metoda toUri wczytuje dodatkowe dane jako
//część identyfikatora URI.
//Dane tej intencji nie są w ogóle używane, służą jedynie
//do definiowania unikatowości intencji oczekującej.
//Spójrzmy na metodę intent.filterEquals(), aby dowiedzieć się,
//w jaki sposób intencje są ze sobą porównywane.
onListClickIntent.setData(
    Uri.parse(
        onListClickIntent.toUri(Intent.URI_INTENT_SCHEME)));

//Musimy później dostarczyć tę intencję w postaci komunikatu
//do tego samego odbiorcy po
//kliknięciu zdalnego widoku.
final PendingIntent onListClickPendingIntent =
    PendingIntent.getBroadcast(context, 0,
        onListClickIntent,
        PendingIntent.FLAG_UPDATE_CURRENT);

//Ustanawiamy szablon intencji oczekującej dla
//elementu listy.
//Każdy widok w liście będzie musiał definiować wtedy
//własny zestaw danych dodawanych do
//tego szablonu, a następnie przesyłać taki
//ostateczny szablon.
//Przyjrzyjmy się, w jaki sposób metoda RemoteViewsFactory()
//konfiguruje każdy element w widoku listy.
//Przejrzyjmy także dokumentację metody RemoteViews.setFillIntent().
rv.setPendingIntentTemplate(R.id.listwidget_list_view_id,
    onListClickPendingIntent);

//Aktualizuje widżet
appWidgetManager.updateAppWidget(appWidgetId, rv);
}

@Override
```

```

public void onReceive(Context context, Intent intent)
{
    if (intent.getAction()
        .equals(TestListWidgetProvider.ACTION_LIST_CLICK))
    {
        //Nie jest to jedno z działań widżetu.
        //Jest to specyficzne działanie, które zostało tutaj skierowane.
        dealwithListAction(context,intent);
        return;
    }

    //Upewnijmy się, że zostanie wywołana.
    super.onReceive(context, intent);
}
public void dealwithListAction(Context context, Intent intent)
{
    String clickedItemText =
        intent.getStringExtra(
            TestListWidgetProvider.EXTRA_LIST_ITEM_TEXT);
    if (clickedItemText == null)
    {
        clickedItemText = "Błąd";
    }
    clickedItemText =
        clickedItemText
        + "Został wybrany element:"
        + clickedItemText;

    Toast t =
        Toast.makeText(context,clickedItemText,Toast.LENGTH_LONG);
    t.show();
}

}//eof-class

```

Po teoretycznym wstępie z poprzedniego punktu Czytelnik powinien zrozumieć wiele z zadań wykonywanych przez ten plik. Kod źródłowy został również bogato opatrzony komentarzami, podsumowującymi dotychczas przekazane informacje; poniżej jednak prezentujemy krótkie podsumowanie działania tej klasy:

1. Wczytujemy widok w metodzie `onUpdate()`.
2. Lokalizujemy zdalny widok listy iłączamy go do klasy fabrykującej za pomocą usługi.
3. Ustanawiamy zdarzenia `onClick` zdalnego widoku za pomocą szablonu intencji oczekującej.
4. Przesłaniamy metodę `onReceive()` i przetwarzamy niestandardowe działanie `onClick`.

Nie należy kompilować tego pliku przed załadowaniem do środowiska Eclipse pozostałych plików wymaganych do działania projektu, gdyż znajdują się w nim odniesienia do tych pozostałych plików.

## Utworzenie klasy fabrykującej

Na listingu 31.22 został zamieszczony kod źródłowy klasy fabrykującej, która wypełnia widok listy.

**Listing 31.22.** TestRemoteViews[BP3]Factory.java

---

```
package com.androidbook.homewidgets.listwidget;
//
//Za pomocą skrótu klawiaturowego Ctrl+Shift+O uzupełnimy instrukcje importów
//
class TestRemoteViewsFactory
    implements RemoteViewsService.RemoteViewsFactory
{
    private Context mContext;
    private int mAppWidgetId;
    private static String tag="TRVF";
    public TestRemoteViewsFactory(Context context, Intent intent)
    {
        mContext = context;
        mAppWidgetId =
            intent.getIntExtra(
                AppWidgetManager.EXTRA_APPWIDGET_ID,
                AppWidgetManager.INVALID_APPWIDGET_ID);

        Log.d(tag,"Utworzono klase fabrykujaca");
    }

    //Wywołana podczas pierwszego utworzenia klasy fabrykującej.
    //Jedna klasa fabrykująca może być współdzielona pomiędzy wieloma
    //obiektami RemoteViewsAdapter, w zależności od przekazanej intencji.
    public void onCreate()
    {
        Log.d(tag,"Wywolano metode onCreate dla identyfikatora widzetu:" + mAppWidgetId);
    }

    //Wywołana w momencie odłączenia ostatniego obiektu
    //RemoteViewsAdapter powiązanego z tą klasą fabrykującą.
    public void onDestroy()
    {
        Log.d(tag,"Wywolano metode onDestroy dla identyfikatora widzetu:" +
            mAppWidgetId);
    }

    //Całkowita liczba elementów listy.
    public int getCount()
    {
        return 20;
    }

    public RemoteViews getViewAt(int position)
    {
        Log.d(tag,"Wywolano metode getView:" + position);
        RemoteViews rv =
            new RemoteViews(
```

```
        this.mContext.getPackageName(),
        R.layout.list_item_layout);
    String itemText = "Element:" + position;
    rv.setTextViewText(
        R.id.textview_widget_list_item_id, itemText);

    this.loadItemOnClickExtras(rv, position);
    return rv;
}
private void loadItemOnClickExtras(RemoteViews rv, int position)
{
    Intent ei = new Intent();
    ei.putExtra(TestListWidgetProvider.EXTRA_LIST_ITEM_TEXT,
        "Pozycja klikniętego obiektu:" + position);
    rv.setOnClickListener(R.id.textview_widget_list_item_id, ei);
    return;
}

//Za jej pomocą możemy wykorzystać niestandardowy widok wczytywania,
//który pojawia się pomiędzy wywołaniem i powrotem metody
//getViewAt(int). W przypadku powrotu z wartością null,
//wykorzystany zostanie domyślny widok wczytywania.
public RemoteViews getLoadingView()
{
    return null;
}

//Liczba różnych rodzajów widoków
//tworzących tę listę.
public int getViewTypeCount()
{
    return 1;
}

//Wewnętrzny identyfikator obiektu
//na danej pozycji.
public long getItemId(int position)
{
    return position;
}

//Zostanie wstawiona wartość true, jeśli ten sam identyfikator
//odnosi się zawsze do tego samego obiektu.
public boolean hasStableIds()
{
    return true;
}

//Wywoływana w momencie uruchomienia metody notifyDataSetChanged()
//w adapterze widoku zdalnego. W ten sposób klasa RemoteViewsFactory
//może reagować na zmiany wprowadzone w danych poprzez aktualizowanie
//wszystkich wewnętrznych odniesień.
//Uwaga: można bezpieczne przeprowadzać zasobochłonne zadania
//wewnętrz tej metody, w synchroniczny sposób.
//W międzyczasie będą wyświetlane stare informacje
```

```
//wewnątrz widżetu.  
public void onDataSetChanged()  
{  
    Log.d(tag, "onDataSetChanged");  
}  
}
```

---

Większość powyższego kodu została już wcześniej omówiona. Klasa ta definiuje obecność dwudziestu wierszy. Układ graficzny każdego wiersza zostaje wczytany z pliku, a zawarty w nim tekst zostaje umiejscowiony w odpowiedniej pozycji. Następnie zostaje wczytany tekst z każdej pozycji do intencji działania onClick. Ten właśnie tekst zostaje umieszczony w kontrolce typu Toast.

## Kod usługi zdalnego widoku

Na listingu 31.23 zaprezentowaliśmy kod źródłowy usługi, która przekazuje klasę fabrykującą zdalnego widoku.

**Listing 31.23.** TestRemoteViewsService.java

---

```
package com.androidbook.homewidgets.listwidget;  
import android.content.Intent;  
  
public class TestRemoteViewsService  
extends android.widget.RemoteViewsService  
{  
    @Override  
    public RemoteViewsFactory onGetViewFactory(Intent intent)  
    {  
        return new TestRemoteViewsFactory(  
            this.getApplicationContext(), intent);  
    }  
}
```

---

## Główny plik układu graficznego widżetu

Główny plik układu graficznego definiujący wygląd widżetu na stronie startowej powinien zostać umieszczony w katalogu */res/layout/test\_list\_widget\_layout.xml* (przypominamy, że zawartość tego pliku została zaprezentowana na listingu 31.1). Do głównego układu graficznego zaliczamy również funkcję zaokrąglania rogów, która została umieszczona w osobnym pliku. Plik ten znajduje się w katalogu */res/drawable/box1.xml*, a jego definicję znajdziemy na listingu 31.2.

## Układ graficzny pojedynczych elementów listy

Plik ten definiuje wygląd pojedynczego elementu stanowiącego składnik listy. Plik ten musi zostać umieszczony w katalogu *layout/list\_item\_layout.xml*. Został on pokazany na listingu 31.10.

## Metadane dostawcy widżetów

Dostawca widżetów wymaga dołączenia pliku metadanych podczas jego definiowania w pliku manifeście. Plik ten umieszcza się w katalogu */res/xml/test\_list\_appwidget\_provider.xml*. Prezentujemy jego treść na listingu 31.24.

**Listing 31.24.** Plik zawierający informacje o widżecie

---

```
<!-- xml/test_list_widget_layout.xml -->
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="222dp"
    android:minHeight="222dp"
    android:updatePeriodMillis="1000000"
    android:initialLayout="@layout/test_list_widget_layout"
    android:label="Testowy widżet zawierający listę"
    >
</appwidget-provider>
```

---

W pliku metadanych ustalamy rozmiar widżetu oraz definiowaną w milisekundach częstotliwość wywołań metody `onUpdate`. Przypominamy, że plik tego typu został dokładniej omówiony w rozdziale 22.

## AndroidManifest.xml

Listing 31.25 zawiera kod pliku konfiguracyjnego aplikacji. Zaznaczyliśmy pogrubioną czcionką definicje dostawcy widżetów oraz usługi zdalnego widoku.

**Listing 31.25.** Plik AndroidManifest.xml

---

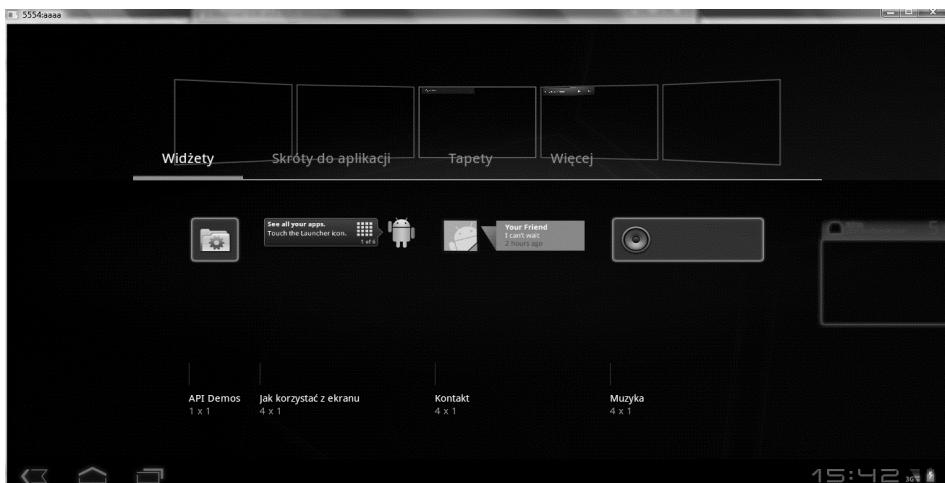
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.homewidgets.listwidget"
    android:versionCode="1"
    android:versionName="1.0.0">
    <application android:icon="@drawable/icon"
        android:label="Testowy widżet wyświetlający listę">
        <!--
        ****
        * Dostawca testowego widżetu wyświetlającego listę
        ****
        -->
        <receiver android:name=".TestListWidgetProvider">
            <meta-data android:name="android.appwidget.provider"
                android:resource="@xml/test_list_appwidget_provider" />
            <intent-filter>
                <action
                    android:name="android.appwidget.action.APPWIDGET_UPDATE" />
            </intent-filter>
        </receiver>
        <!-- Usługa dostarczająca obiekty RemoteViews do widżetu zbiorczego -->
        <service android:name=".TestRemoteViewsService">
            android:permission="android.permission.BIND_REMOTEVIEWS"
            android:exported="false" />
        </service>
        <uses-sdk android:minSdkVersion="11" />
    </application>
</manifest>
```

---

## Testowanie widżetu wyświetlającego listę

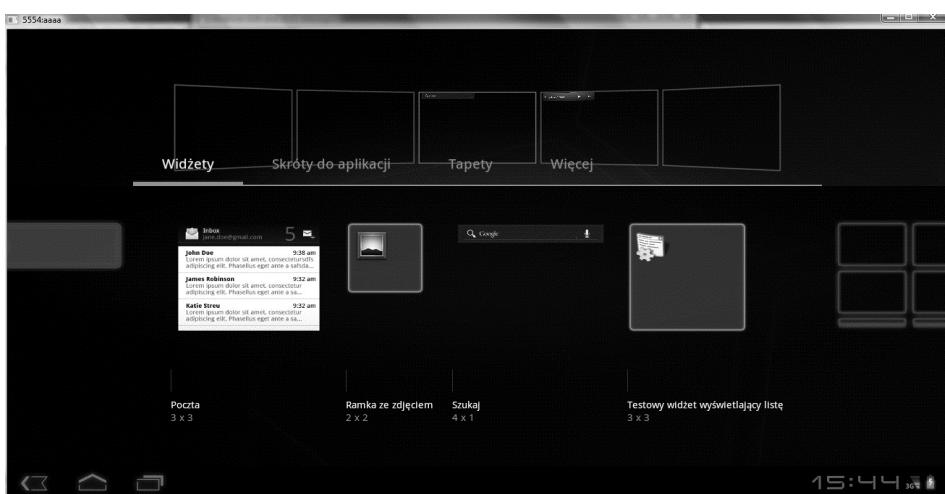
Po utworzeniu i wdrożeniu projektu w środowisku Eclipse otrzymamy komunikat o jego udanej instalacji. Ponieważ nie wprowadziliśmy do projektu żadnej aktywności, którą moglibyśmy uruchomić, domyślnie nie zobaczymy niczego nowego w oknie emulatora.

Aby zainstalować utworzony w tym przykładzie widżet, musimy najpierw zajrzeć do listy dostępnych widżetów. Kliknięcie ikony ekranu startowego spowoduje wyświetlenie okna dostępnych widżetów, co zostało zilustrowane na rysunku 31.2.



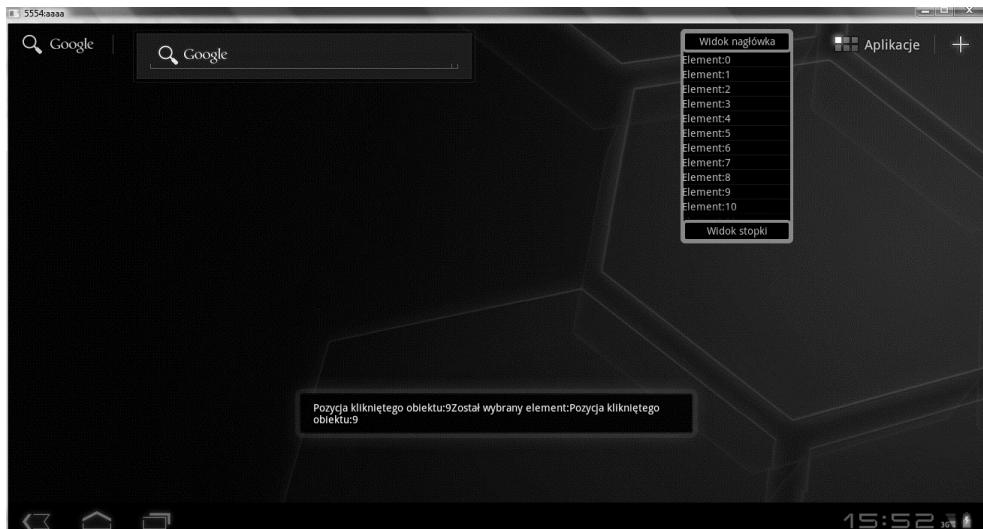
Rysunek 31.2. Lista widżetów

Nasz widżet nosi nazwę *Testowy widżet wyświetlający listę*, może więc znajdować się na samym końcu listy, co oznacza, że musimy przewinąć ekran widżetów w prawo. Widzimy to na rysunku 31.3.



Rysunek 31.3. Przewijanie ekranu w prawą stronę w poszukiwaniu utworzonego widżetu

Możemy teraz przeciągnąć *Testowy widżet wyświetlający listę* na dowolną stronę ekranu startowego. Po rozpoznaniu operacji przeciągania przez system klikamy przycisk ekranu startowego, aby go wyświetlić. W międzyczasie ujrzymy podstawową formę widżetu, którą zaprezentowaliśmy na rysunku 31.1. Jeżeli klikniemy któryś z elementów listy, pojawi się kontrolka *Toast* zawierająca odpowiedni komunikat związany z tym elementem (rysunek 31.4).



**Rysunek 31.4.** Obiekt *Toast* generowany wskutek kliknięcia elementu listy

Na tym zakończymy omówienie usprawnień wprowadzonych do widżetów w wersji 3.0 Androida. Przejdzmy teraz do zupełnie nowego interfejsu przeciągania.

## Funkcja przeciągania

Przed pojawieniem się wersji 3.0 systemu nie istniała możliwość bezpośredniej obsługi funkcji przeciągania elementów. W rozdziale 25. pokazaliśmy, w jaki sposób przesuwać dany widok po ekranie; dowiedzieliśmy się również, że można wykorzystać bieżące położenie przeciąganego obiektu, aby ustalić, czy pod nim znajduje się jakiś inny element. Po otrzymaniu zdarzenia *MotionEvent* oznaczającego oderwanie palca od wyświetlacza aplikacja może rozpoznać, czy jest to równoznaczne z upuszczeniem obiektu. Chociaż wprowadzenie takiej funkcji jest możliwe, zdecydowanie wygodniejszym rozwiązaniem byłaby bezpośrednia implementacja operacji przeciągania. Otrzymaliśmy ją właśnie w wersji 3.0 Androida.

### Podstawowe informacje o funkcji przeciągania w wersji 3.0 Androida

Na najbardziej ogólnym poziomie proces przeciągania rozpoczyna się od zadeklarowania widoku, w którym został on zainicjalizowany, następnie wszystkie zaangażowane w tę operację elementy śledzą zjawisko przesuwania obiektu, dopóki nie zostanie wykryte zdarzenie upuszczenia. Jeżeli zdarzenie upuszczenia zostanie wykryte przez widok, który je odbierze, cała operacja przeciągania zostanie zakończona sukcesem. Jeżeli żaden widok nie odbierze zdarzenia upuszczenia

bądź jeśli widok odbierający nie może go przyjąć, przyciąganie okaże się nieudane. Informacje o procesie przyciągania są przekazywane przez obiekt `DragEvent`, który jest przekazywany do wszystkich dostępnych obiektów nasłuchujących pod tym kątem.

Wewnątrz obiektu `DragEvent` znajdziemy deskryptory przechowujące mnóstwo informacji, które są zależne od inicjalizatora operacji przyciągania. Na przykład obiekt `DragEvent` może zawierać odniesienia do samego inicjalizatora, informacji o stanie, danych tekstowych, identyfikatorów URI oraz zasadniczo wszelkich innych elementów, jakie chcielibyśmy przekazać za pomocą sekwencji przyciągania.

Możemy przekazywać informacje, które w rezultacie umożliwiają dynamiczną komunikację pomiędzy widokami, jednak początkowe dane zawarte w obiekcie `DragEvent`, które zostają do niego dołączone w momencie utworzenia, nie ulegają potem zmianom. Oprócz danych obiekt `DragEvent` zawiera także wartość działania, definiującą bieżące zachowanie w sekwencji przyciągania, a także informacje o położeniu, określające lokalizację przyciąganego obiektu na ekranie.

Obiekt `DragEvent` może określić jedno z sześciu następujących działań:

- `ACTION_DRAG_STARTED` definiuje rozpoczęcie nowej sekwencji przyciągania.
- `ACTION_DRAG_ENTERED` oznacza przyciągnięcie obiektu do wnętrza określonego widoku.
- `ACTION_DRAG_LOCATION` określa przyciągnięcie obiektu do nowej lokalizacji na ekranie.
- `ACTION_DRAG_EXITED`, gdy obiekt zostaje przyciągnięty na zewnątrz określonego widoku.
- `ACTION_DROP` występuje wtedy, gdy użytkownik puszcza przyciągany obiekt. Decyzja, czy nastąpiło faktyczne zjawisko upuszczenia, należy do odbiorcy tego zdarzenia.
- `ACTION_DRAG_ENDED` informuje wszystkie obiekty nasłuchujące procesu przyciągania o zakończeniu sekwencji przyciągania. Metoda `DragEvent.getResult()` służy do określenia, czy upuszczenie obiektu zostało zakończone powodzeniem.

Może się wydawać, że powinniśmy skonfigurować obiekt nasłuchujący sekwencji przyciągania w każdym widoku będącym częścią tego procesu. W rzeczywistości możemy zdefiniować obiekt nasłuchujący wobec dowolnego elementu znajdującego się w aplikacji i będzie on otrzymywać wszystkie zdarzenia sekwencji przyciągania występujące we wszystkich widokach systemu. Możemy w ten sposób nieco skomplikować sytuację, ponieważ obiekt nasłuchujący zdarzenia przyciągania nie musi być powiązany ani z przyciąganym obiektem, ani nawet z docelowym miejscem. Obiekt nasłuchujący może zarządzać całą koordynacją sekwencji przyciągania.

Jeżeli przyjrzymy się przykładowemu projektowi dostępnemu w zestawie Android SDK, zauważymy, że w rzeczywistości obiekt nasłuchujący zostaje powiązany z kontrolką `TextView`, która nie ma nic wspólnego z faktyczną sekwencją przyciągania. Prezentowana przez nas aplikacja posiada obiekty nasłuchujące połączone z określonymi widokami. Każdy z takich obiektów nasłuchujących otrzymuje zdarzenie `DragEvent`, pojawiające się w sekwencji przyciągania. Oznacza to, że dany widok może otrzymać obiekt `DragEvent` i zignorować go, ponieważ dotyczy zupełnie innego widoku. Oznacza to także, że obiekt nasłuchujący musi posiadać jakiś mechanizm rozróżniania w kodzie, a obiekt `DragEvent` powinien posiadać wystarczającą ilość informacji do określania przeprowadzanych czynności.

Jeżeli obiekt nasłuchujący otrzyma zdarzenie `DragEvent` informujące jedynie o przyciąganiu nieznanego elementu oraz o jego współrzędnych (15, 57), to obiekt ten niewiele będzie mógł zrobić. O wiele przydatniejszy będzie obiekt `DragEvent`, który przekazuje dane definiujące prze-

ciągany element, określające jego położenie w pozycji (15, 57), jak również wskazujące, że mamy do czynienia z operacją kopiowania. Dodatkowo ta informacja jest zawarta w odpowiednim identyfikatorze URI. Jeśli zastosujemy taki obiekt DragEvent, to w momencie upuszczenia elementu będziemy posiadać wystarczająco wiele informacji, aby przeprowadzić proces kopiowania.

## Przykładowa aplikacja prezentująca funkcję przeciągania

Tworząc naszą przykładową aplikację, wykorzystamy podstawową koncepcję stosowaną w wersji 3.0 Androida — fragmenty. Pozwoli nam to udowodnić między innymi, że operacja przeciągania przekracza również granice fragmentów. Utworzmy zbiór kółek z lewej strony ekranu, a z prawej — docelowy kwadrat. Po uchwyceniu kółka za pomocą długiego kliknięcia zmieni ono kolor, a w trakcie przeciągania będziemy widzieć jego cień. Gdy kółko zostanie przeciągnięte nad kwadrat, zacznie on świecić. Jeżeli upuścimy teraz kółko w obszarze kwadratu, pojawi się komunikat, że doliczono kolejne upuszczenie do całkowitej puli zliczanych upuszczeń. Poza tym obszar kwadratu przestanie świecić, a oryginalne kółko powróci do swojego pierwotnego koloru.

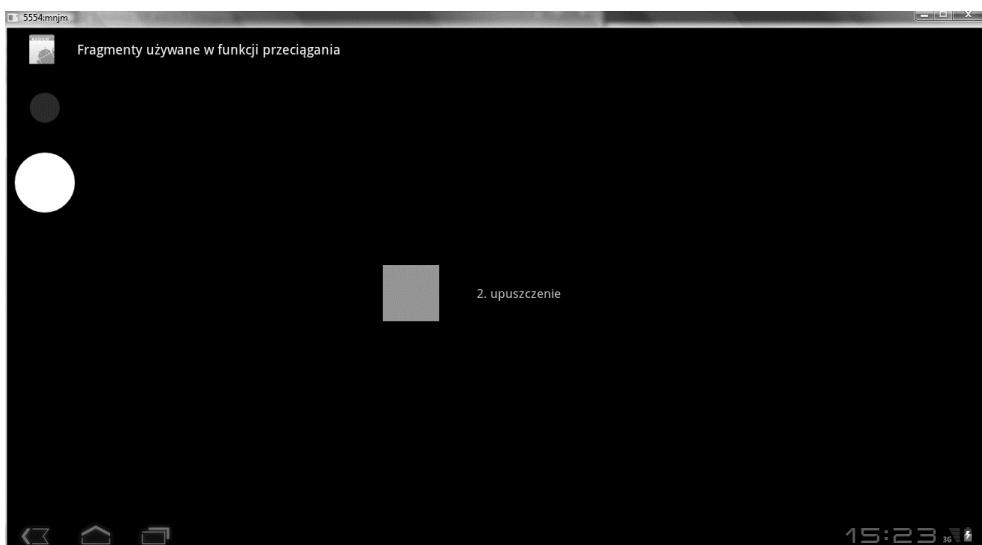
### **Lista plików**

Do utworzenia omawianej aplikacji będą potrzebne następujące pliki:

- *main.xml* stanowi główny układ graficzny aplikacji, w którym zostały umieszczone dwa fragmenty (listing 31.26).
- *palette.xml* jest układem graficznym fragmentu zawierającego kółka umieszczone po lewej stronie ekranu (listing 31.27).
- *dropzone.xml* stanowi układ graficzny fragmentu reprezentującego docelowy obszar po prawej stronie ekranu oraz komunikat o ilości upuszczeń (listing 31.28).
- *MainActivity.java* to najprostsza z możliwych aktywności. Ustanawia ona jedynie główny widok treści, a resztę operacji pozostawia fragmentom (listing 31.29).
- *Palette.java* jest kodem również nieskomplikowanego zbioru kółek. Służy on jedynie do rozwinięcia układu graficznego palety (listing 31.30).
- *DropZone.java* jest nieco bardziej złożonym kodem, ponieważ implementujemy w nim mechanizm upuszczania. Zostaje tu rozwinięty układ graficzny *dropzone.xml*, a następnie zaimplementowany obiekt nasłuchujący (listing 31.31).
- *Dot.java* stanowi klasę niestandardowego widoku obiektów, które będą przeciągane. Tutaj przechowujemy mechanizm inicjalizacji sekwencji przeciągania, obserwujemy zdarzenia przeciągania oraz rysujemy kółka (listing 31.32).
- *attrs.xml* definiuje dwa atrybuty XML wykorzystywane w układzie graficznym *palette.xml* do opisu atrybutu kółek (listing 31.33).
- *AndroidManifest.xml* jest głównym plikiem manifestem aplikacji (listing 31.34).
- *strings.xml* zawiera ciągi znaków wykorzystywane przez plik *AndroidManifest.xml* (listing 31.35).

## **Tworzenie układu graficznego przykładowej aplikacji ukazującej funkcję przeciągania**

Zanim przejdziemy do kodów źródłowych, zobaczymy na rysunku 31.5, jak prezentuje się nasza aplikacja.



Rysunek 31.5. Interfejs aplikacji demonstrującej działanie funkcji przeciągania

Na listingu 31.26 został zaprezentowany główny układ graficzny tworzący interfejs użytkownika, widoczny na rysunku 31.5. Podobnie jak w przykładach omówionych w rozdziale 29., układ ten składa się z prostego, głównego układu liniowego, w którym zostały poziomo rozmieszczone dwa fragmenty. Pierwszy fragment będzie przechowywał zbiór kółek, a drugi będzie naszą strefą upuszczania obiektów.

**Listing 31.26.** Plik głównego układu graficznego

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/main.xml -->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <fragment class="com.androidbook.drag.drop.demo.Palette"
        android:id="@+id/palette"
        android:layout_width="wrap_content"
        android:layout_height="match_parent" />

    <fragment class="com.androidbook.drag.drop.demo.DropZone"
        android:id="@+id/dropzone"
        android:layout_width="0px"
        android:layout_height="match_parent"
        android:layout_weight="1" />

</LinearLayout>
```

Plik układu graficznego, który definiuje wygląd fragmentu przechowującego kółka (listing 31.27), jest nieco bardziej interesujący. Ponieważ fragment jest reprezentowany przez ten układ graficzny, nie musimy wstawać w tym układzie graficznym znacznika fragmentu. Układ graficzny

zostanie rozwinięty i stanie się hierarchią widoków dla fragmentu przechowującego kółka. Same kółka zostały zdefiniowane niestandardowo, a dwa z nich zostały rozmiieszczone pionowo. Zwróćmy uwagę, że w definicji kółek znajdziemy dwa atrybuty XML (`dot:color` oraz `dot:radius`). Jak widać, atrybuty te określają kolor oraz promień kółka. Poruszmy ten temat ponownie przy okazji omawiania pliku `attrs.xml`. Pozostałe atrybuty stanowią standard definiowania widoków.

**Listing 31.27.** Plik układu graficznego `palette.xml`, definiujący wygląd kółek

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/palette.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:dot="http://schemas.android.com/apk/res/com.androidbook.drag.drop.demo"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <com.androidbook.drag.drop.demo.Dot android:id="@+id/dot1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="30dp"
        android:tag="Niebieskie kolko"
        dot:color="#ff1111ff"
        dot:radius="20dp"
    />
    <com.androidbook.drag.drop.demo.Dot android:id="@+id/dot2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:tag="Biale kolko"
        dot:color="#ffffffff"
        dot:radius="40dp"
    />
</LinearLayout>
```

---

Układ graficzny strefy upuszczania z listingu 31.28 jest również bardzo łatwy do zrozumienia. Widzimy w nim zielony kwadrat oraz umieszczony obok niego komunikat tekstowy. W tym właśnie miejscu będziemy upuszczać przeciągane kółka. Komunikat tekstowy będzie informował o liczbie upuszczonych kółek.

**Listing 31.28.** Plik układu graficznego `dropzone.xml`

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/layout/dropzone.xml -->
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <View android:id="@+id/droptarget"
        android:layout_width="75dp"
        android:layout_height="75dp"
        android:background="#00ff00"/>
    <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="0"/>
</LinearLayout>
```

---

```
        android:layout_gravity="center_vertical"
        android:background="#00ff00" />

<TextView android:id="@+id/dropmessage"
        android:text="0 kropek"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:paddingLeft="50dp"
        android:textSize="17sp" />

</LinearLayout>
```

---

Jak widzimy na listingu 31.29, mamy do czynienia z najprostszą możliwą aktywnością. Ustanawia ona jedynie nadrędny widok treści, który zostaje przyporządkowany głównemu układowi graficznemu (zawierającemu dwa fragmenty). Kod, który definiuje interesujące nas operacje, został zamieszczony w pozostałych plikach.

#### **Listing 31.29.** Plik głównej aktywności

---

```
package com.androidbook.drag.drop.demo;

// Jest to plik MainActivity.java
import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

---

Kod z listingu 31.30 również jest niezwykle prosty. Jest to kod fragmentu, a więc przesłaniamy w nim metodę onCreateView() w taki sposób, aby układ graficzny został rozwinięty w widok, który następnie zostaje przekazany.

#### **Listing 31.30.** Plik Palette.java

---

```
package com.androidbook.drag.drop.demo;

// Jest to plik Palette.java
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Palette extends Fragment {

    @Override
```

```
public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle icicle) {
    View v = inflater.inflate(R.layout.palette, container, false);
    return v;
}
```

### Odpowiedź na zdarzenie onDrag w strefie upuszczania

Do tego momentu skonfigurowaliśmy cały główny układ graficzny aplikacji. Na listingu 31.31 pokazaliśmy, w jaki sposób należy zorganizować docelowy obszar, w którym będą upuszczane obiekty.

**Listing 31.31.** Plik DropZone.java

```
        Log.v(DROPTAG, "Sekwencja przeciagania rozpoczęta w dropTarget");
        break;
    case DragEvent.ACTION_DRAG_ENTERED:
        Log.v(DROPTAG, "Sekwencja przeciagania wkroczyla do dropTarget");
        anim = ObjectAnimator.ofFloat((Object)v, "alpha", 1f, 0.5f);
        anim.setInterpolator(new CycleInterpolator(40));
        anim.setDuration(30*1000); // 30 sekund
        anim.start();
        break;
    case DragEvent.ACTION_DRAG_EXITED:
        Log.v(DROPTAG, "Sekwencja przeciagania opuscila dropTarget");
        if(anim != null) {
            anim.end();
            anim = null;
        }
        break;
    case DragEvent.ACTION_DRAG_LOCATION:
        Log.v(DROPTAG, "Sekwencja przeciagania kontynuuje w dropTarget: " +
                event.getX() + ", " + event.getY());
        break;
    case DragEvent.ACTION_DROP:
        Log.v(DROPTAG, "Upuszczenie obiektu w dropTarget");
        if(anim != null) {
            anim.end();
            anim = null;
        }

        ClipData data = event.getClipData();
        Log.v(DROPTAG, "Elementem danych jest " + data.getItemAt(0).getText());

        dropCount++;
        String message = dropCount + ". upuszczenie";
        dropMessage.setText(message);
        break;
    case DragEvent.ACTION_DRAG_ENDED:
        Log.v(DROPTAG, "Sekwencja przeciagania zakonczona w dropTarget");
        if(anim != null) {
            anim.end();
            anim = null;
        }
        break;
    default:
        Log.v(DROPTAG, "inne dzialanie w strefie upuszczania: " + action);
        result = false;
    }
    return result;
}
});
return v;
}
}
```

---

Docieramy w końcu do interesującej części kodu. Programując strefę upuszczania, musimy utworzyć rejon docelowy, do którego chcemy przeciągać kółka. Jak już wcześniej zauważyliśmy, w układzie graficznym zdefiniowaliśmy zielony kwadrat, obok którego znalazło się pole tekstowe.

Ponieważ strefa upuszczania również jest fragmentem, przesłaniamy metodę `onCreateView()` klasy `DropZone`. Pierwszą czynnością jest rozwinięcie układu graficznego strefy upuszczania, a następnie wydobycie odniesienia widoku kwadratowego obszaru (`dropTarget`) oraz komunikatu tekstowego (`dropMessage`). Możemy wtedy skonfigurować obiekt nasłuchujący sekwencji przeciągania nad ten obszar docelowy, dzięki czemu będziemy informowani o postępach procesu przeciągania.

Obiekt nasłuchujący umieszczony w rejonie docelowym zawiera jedną metodę zwrotną — `onDrag()`. Będzie ona otrzymywała odniesienie do widoku oraz obiekt `DragEvent`. Odniesienie to jest powiązane z widokiem znajdującym się w relacji z obiektem `DragEvent`. Zgodnie z tym, co wcześniej wspomnieliśmy, obiekt nasłuchujący nie musi być koniecznie połączony z widokiem, w którym będzie następować zdarzenie przeciągania, zatem to metoda zwrotna musi identyfikować widok, w którym pojawi się sekwencja przeciągania.

Jedną z pierwszych czynności, jaką prawdopodobnie chcielibyśmy przeprowadzić w dowolnej metodzie zwrotnej `onDrag()`, jest odczytanie działania pochodzącego z obiektu `DragEvent`. Poznamy w ten sposób stan sekwencji przeciągania. Przeważnie wystarczy nam odnotowanie informacji, czy zdarzenie przeciągania jest ciągle w toku. Nie musimy na przykład nic robić z działaniem `ACTION_DRAG_LOCATION`. Chcemy jednak zaimplementować odpowiednią logikę, uruchamiającą się w przypadku przeciągnięcia obiektu przez określone granice (działanie `ACTION_DRAG_ENTERED`), która przestanie być aktywna po przeciągnięciu tego obiektu na zewnątrz danego rejonu (`ACTION_DRAG_EXITED`) lub po upuszczeniu tego obiektu (`ACTION_DRAG_DROPPED`).

Stosujemy tu omówioną w rozdziale 29. klasę `ObjectAnimator`, w tym jednak przypadku służy nam ona do zdefiniowania cyklicznego interpolatora, modyfikującego przezroczystość obszaru docelowego. W ten sposób uzyskamy efekt pulsowania docelowego zielonego kwadratu. Ma to być efekt graficzny, zachęcający użytkownika do upuszczenia obiektu w tym rejonie. Skoro w określonych warunkach włączamy animację, musimy ją również wyłączyć w momencie upuszczenia obiektu lub zakończenia sekwencji przeciągania. Teoretycznie nie musimy kończyć animacji w przypadku działania `ACTION_DRAG_ENDED`, rozsądniej jednak byłoby to zrobić.

W przypadku tego konkretnego obiektu nasłuchującego będziemy otrzymywać jedynie działania `ACTION_DRAG_ENTERED` i `ACTION_DRAG_EXITED`, jeśli wystąpi interakcja z powiązany widokiem. Jak się również przekonamy, zdarzenia `ACTION_DRAG_LOCATION` będą występować tylko w obrębie docelowego widoku.

Ostatnim naprawdę interesującym etapem jest samo działanie `ACTION_DROP` (zwróćmy uwagę, że pominięty został człon `DRAG_`). Jeżeli działanie to nastąpiło w nasłuchiwanym obszarze, to znaczy, że użytkownik upuścił kółko w obszarze zielonego kwadratu. Ponieważ przez cały czas spodziewamy się, że obiekt zostanie upuszczony w tym obszarze, możemy po prostu odczytać dane z pierwszego elementu, a następnie wyświetlić je w oknie `LogCat`. W przypadku użytkowej aplikacji możemy zwrócić większą uwagę na obiekt `ClipData`, przechowywany w samym zdarzeniu przeciągania. Za pomocą jego właściwości możemy akceptować lub odrzucać zdarzenie upuszczenia.

Nadszedł właściwy moment na określenie wyniku przeciągania w metodzie `onDrag()`. W zależności od sytuacji możemy poinformować system, że obsłużyliśmy zdarzenie przeciągania (poprzez przekazanie wartości `true`), lub możemy tego nie robić (wartość `false`). Jeżeli wewnątrz obiektu zdarzenia przeciągania nie ma wymaganych danych, zdecydowanie należy przekazać wartość `false` i poinformować w ten sposób system, że zdarzenie nie zostało właściwie obsłużone.

Po umieszczeniu informacji dotyczących zdarzenia przeciągania w oknie *LogCat* zwiększymy wartość w liczniku zdarzeń upuszczania. Wartość ta jest aktualizowana w interfejsie użytkownika — i to tyle, jeśli chodzi o klasę *DropZone*.

Jeżeli spojrzymy na nią z ogólnej perspektywy, okaże się, że wcale nie jest taka skomplikowana. W rzeczywistości nie umieściliśmy tu kodu przetwarzającego zdarzenia *MotionEvent*. Nie musieliśmy nawet wprowadzać tu kodu sprawdzającego, czy sekwencja przeciągania ma miejsce. Otrzymujemy jedynie odpowiednie wywołania metod zwrotnych wraz z postępem sekwencji przeciągania.

## Konfigurowanie widoków obiektów stanowiących źródła sekwencji przeciągania

Zastanówmy się teraz, w jaki sposób zostały zorganizowane widoki związane z obiektami stanowiącymi źródła sekwencji przeciągania. Na początku zachęcamy do przejrzenia kodu zawartego na listingu 31.32.

**Listing 31.32.** Kod źródłowy niestandardowego widoku — kółko

---

```
package com.androidbook.drag.drop.demo;

// Jest to plik Dot.java
import android.content.ClipData;
import android.content.Context;
import android.content.res.TypedArray;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.util.Log;
import android.view.DragEvent;
import android.view.View;

public class Dot extends View
    implements View.OnDragListener
{
    private static final int DEFAULT_RADIUS = 20;
    private static final int DEFAULT_COLOR = Color.WHITE;
    private static final int SELECTED_COLOR = Color.MAGENTA;
    protected static final String DOTTAG = "DragDot";
    private Paint mNormalPaint;
    private Paint mDraggingPaint;
    private int mColor = DEFAULT_COLOR;
    private int mRadius = DEFAULT_RADIUS;
    private boolean inDrag;

    public Dot(Context context, AttributeSet attrs) {
        super(context, attrs);

        // Stosujemy ustawienia atrybutów z pliku układu graficznego.
        // Uwaga: mogą one ulegać zmianie pod wpływem zmiany konfiguracji,
        // na przykład obrotu wyświetlacza.
        TypedArray myAttrs = context.obtainStyledAttributes(attrs,
            R.styleable.Dot);
```

```

final int numAttrs = myAttrs.getIndexCount();
for (int i = 0; i < numAttrs; i++) {
    int attr = myAttrs.getIndex(i);
    switch (attr) {
        case R.styleable.Dot_radius:
            mRadius = myAttrs.getDimensionPixelSize(attr, DEFAULT_RADIUS);
            break;
        case R.styleable.Dot_color:
            mColor = myAttrs.getColor(attr, DEFAULT_COLOR);
            break;
    }
}
myAttrs.recycle();

// Konfiguruje malowane kolory
mNormalPaint = new Paint();
mNormalPaint.setColor(mColor);
mNormalPaint.setAntiAlias(true);

mDraggingPaint = new Paint();
mDraggingPaint.setColor(SELECTED_COLOR);
mDraggingPaint.setAntiAlias(true);

// Rozpoczyna sekwencję przeciągania, wywołaną długim kliknięciem na kółku.
setOnLongClickListener(lcListener);
setOnDragListener(this);
}

private static View.OnLongClickListener lcListener =
    new View.OnLongClickListener() {
    private boolean mDragInProgress;

    public boolean onLongClick(View v) {
        ClipData data =
            ClipData.newPlainText("DragData", (String)v.getTag());

        mDragInProgress =
            v.startDrag(data, new View.DragShadowBuilder(v),
                (Object)v, 0);

        Log.v((String) v.getTag(),
            "rozpoczęto proces przeciagania? " + mDragInProgress);

        return true;
    }
};

@Override
protected void onMeasure(int widthSpec, int heightSpec) {
    int size = 2*mRadius + getPaddingLeft() + getPaddingRight();
    setMeasuredDimension(size, size);
}

// Mechanizm przeciągania
public boolean onDrag(View v, DragEvent event) {
    String dotTAG = (String) getTag();
    // Martwimy się jedynie o zdarzenia przeciagania, jeżeli to kółka są przeciągane.
}

```

```
if(event.getLocalState() != this) {  
    Log.v(dotTAG, "To zdarzenie przeciagania nie jest przeznaczone dla nas");  
    return false;  
}  
boolean result = true;  
  
// Pobiera wymagane wartości zdarzenia.  
int action = event.getAction();  
float x = event.getX();  
float y = event.getY();  
  
switch(action) {  
case DragEvent.ACTION_DRAG_STARTED:  
    Log.v(dotTAG, "sekwencja przeciagania rozpoczęta. X: " + x + ", Y: " + y);  
    inDrag = true; // Wykorzystywana w poniższej metodzie draw() do zmiany koloru  
    break;  
case DragEvent.ACTION_DRAG_LOCATION:  
    Log.v(dotTAG, "sekwencja przeciagania kontynuowana... Poz.: " + x + ", " + y);  
    break;  
case DragEvent.ACTION_DRAG_ENTERED:  
    Log.v(dotTAG, "sekwencja przeciagania wkroczyła. Poz.: " + x + ", " + y);  
    break;  
case DragEvent.ACTION_DRAG_EXITED:  
    Log.v(dotTAG, "sekwencja przeciagania opuściła. Poz.: " + x + ", " + y);  
    break;  
case DragEvent.ACTION_DROP:  
    Log.v(dotTAG, "upuszczenie. Poz.: " + x + ", " + y);  
    // Przekazuje wartość false, ponieważ nie akceptujemy upuszczenia w rejonie startowym.  
    result = false;  
    break;  
case DragEvent.ACTION_DRAG_ENDED:  
    Log.v(dotTAG, "przeciąganie zakończone. Sukces? " + event.getResult());  
    inDrag = false; // Przywraca pierwotny kolor kółka  
    break;  
default:  
    Log.v(dotTAG, "inne działanie w sekwencji przeciagania: " + action);  
    result = false;  
    break;  
}  
return result;  
}  
  
// W tym miejscu rysujemy kółko oraz zmieniamy jego kolor,  
// jeśli jest ono przeciągane. Uwaga: zmiana koloru wpływa jedynie na  
// właściwe kółko, a nie na jego cień.  
public void draw(Canvas canvas) {  
    float cx = this.getWidth()/2 + getLeftPaddingOffset();  
    float cy = this.getHeight()/2 + getTopPaddingOffset();  
    Paint paint = mNormalPaint;  
    if(inDrag)  
        paint = mDraggingPaint;  
    canvas.drawCircle(cx, cy, mRadius, paint);  
    invalidate();  
}  
}
```

Kod klasy Dot w dużym stopniu przypomina zawartość klasy DropZone. Wynika to częściowo z faktu, że również tutaj otrzymujemy zdarzenia przeciągania. Konstruktor tej klasy przetwarza atrybuty definiujące właściwy promień i kolor kółek, a następnie konfiguruje dwa obiekty nasłuchujące, jeden wychwytujący długie kliknięcia, a drugi — zdarzenia przeciągania.

Fragment kodu dotyczący określania atrybutów jest szczególnie interesujący. Chcemy móc określić właściwości kółka w pliku układu graficznego, jednak niestandardowe atrybuty XML wymagają odpowiedniej konfiguracji w którymś miejscu. W naszym przypadku przeprowadzamy ją w pliku *attrs.xml*, umieszczonym w katalogu */res/values*. Definiujemy w tym pliku obiekt nadawania stylu Dot, a także dwa atrybuty: *color* i *radius*. Na listingu 31.33 została umieszczona zawartość pliku *attrs.xml*.

**Listing 31.33.** Plik attrs.xml definiujący nowe atrybuty dla klasy Dot

---

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/values/attrs.xml -->
<resources>
    <declare-styleable name="Dot">
        <attr name="color" format="color" />
        <attr name="radius" format="dimension" />
    </declare-styleable>
</resources>
```

---

Jak widać, nazwę obiektu można uzyskać ze znacznika. Nazwy (i typy) atrybutów są umieszczone w znacznikach podrzędnych. Poprzez określenie atrybutów w pliku *attrs.xml* możemy wykorzystać je w pliku układu graficznego, a także w kodzie Java. W tym drugim przypadku przeszukujemy wszystkie rozpoznawalne atrybuty, a po znalezieniu atrybutów *color* lub *radius* pobieramy ich wartość i przypisujemy do kółka. Jest to całkiem wygodny sposób definiowania wyglądu obiektów za pomocą atrybutów XML.

Podczas rysowania obiektu posłużymy się dwoma kolorami. Jeden kolor przypiszemy do kółka znajdującego się w stanie spoczynku. W momencie przeciągania, kiedy chcemy w jakiś sposób wyróżnić kółko, zmieniamy jego barwę na karmazynową.

Obiekt nasłuchujący długich kliknięć służy do określania początku sekwencji przeciągania. Jedynym sposobem rozpoczęcia procesu przeciągania jest kliknięcie kółka i przytrzymanie go. Uaktywnienie obiektu nasłuchującego długie kliknięcia spowoduje utworzenie nowego obiektu *ClipData* za pomocą ciągu znaków i znacznika kółka. Znacznik ten jest nazwą kółka zdefiniowaną w pliku układu graficznego. Istnieje kilka innych metod wprowadzenia danych do obiektu *ClipData*, dlatego też zalecamy zapoznanie się z dokumentacją tego obiektu.

Następna metoda jest kluczowa dla naszej aplikacji — *startDrag()*. To właśnie tutaj Android przejmie kontrolę i rozpoczęcie sekwencję przeciągania. Zwróćmy uwagę, że argumentem jest uprzednio utworzony obiekt *ClipData*, następnie cień przeciąganego obiektu, „lokalny stan” obiektu, a na końcu wartość zero.

Cień przeciąganego obiektu jest obrazem wyświetlany w trakcie trwania sekwencji przeciągania. W naszym przypadku zostaje on dołączony na etapie przeciągania, podczas gdy oryginalne kółko pozostaje na swoim miejscu. Domyslnym zachowaniem klasy *DragShadowBuilder* jest utworzenie cienia przypominającego oryginalny obiekt, więc wystarczy, że ją wywołamy i przekażemy do widoku. Możemy tutaj popuścić wodze fantazji i utworzyć dowolny widok cienia, jeśli jednak przesłonimy tę klasę, będziemy musieli jeszcze zaimplementować kilka dodatkowych klas.

Metoda `onMeasure()` przekazuje Androidowi informacje o wymiarach wykorzystywanego niestandardowego widoku. Musimy podać systemowi te dane, aby widok ten pasował do innych elementów widocznych na ekranie.

Znajdziemy tu także metodę zwrotną `onDrag()`. Jak już wspomnieliśmy, każdy dostosowany obiekt nasłuchujący może odbierać zdarzenia przeciągania. Wszystkie te obiekty odbierają na przykład działania `ACTION_DRAG_STARTED` oraz `ACTION_DRAG_ENDED`. Gdy więc pojawi się takie zdarzenie, musimy ostrożnie wykorzystywać zawarte w nim informacje. Ponieważ w naszej aplikacji umieściliśmy dwa kółka, musimy uważać, żeby każda czynność była przeprowadzana na tym właściwym.

Gdy obydwa kółka otrzymują działanie `ACTION_DRAG_STARTED`, tylko jedno z nich powinno zmienić kolor na karmazynowy. Aby określić, które kółko jest właściwe, należy porównać przekazane obiekty lokalnego stanu z własnym obiektem lokalnego stanu. Jeżeli przyjrzymy się fragmentowi, w którym definiowaliśmy obiekt lokalnego stanu, zauważymy, że przekazaliśmy do niego bieżący widok. Zatem po otrzymaniu obiektu lokalnego stanu możemy go porównać z naszym bieżącym stanem, aby określić, czy znajduje się w widoku rozpoczynającym sekwencję przeciągania.

Jeżeli obiekt znajduje się w innym widoku, w oknie `LogCat` zostanie wyświetlony komunikat, że sekwencja przeciągania nie jest przeznaczona dla tego widoku. Nastąpi również przekazanie wartości `false`, która informuje system, że zdarzenie nie zostanie tu obsłużone.

Jeżeli zdarzenie przeciągania trafi do właściwego widoku, należy pobrać z tego zdarzenia pewne wartości i najczęściej jedynie wyświetlić odpowiedni komunikat w oknie `LogCat`. Pierwszym wyjątkiem jest działanie `ACTION_DRAG_STARTED`. Jeżeli działanie to jest przeznaczone dla danego widoku, to znaczy, że kółko jest częścią sekwencji przeciągania. Wprowadzamy więc wartość logiczną w zmiennej `inDrag`, dzięki której metoda `draw()` będzie mogła później wyświetlić kółko w innym kolorze. Kolor jest zmieniony jedynie do czasu pojawienia się działania `ACTION_DRAG_ENDED`, po czym zostaje przywrócona pierwotna barwa kółka.

Jeżeli zostanie wywołane działanie `ACTION_DROP` wobec kółka, to znaczy, że użytkownik próbuje upuścić to kółko na inne kółko, być może w jego początkowym miejscu. Nie powinno mieć to żadnych konsekwencji, więc w tym przypadku należy przekazać po prostu wartość `false`.

Na koniec metoda `draw()` niestandardowego widoku definiuje środek naszego okręgu (w naszym przypadku przeciąganego kółka), a następnie rysuje go za pomocą odpowiedniego koloru. Dzięki metodzie `invalidate()` system zostaje poinformowany o modyfikacji widoku oraz konieczności ponownego narysowania interfejsu użytkownika. Za pomocą tej metody gwarantujemy bardzo szybkie zaktualizowanie interfejsu użytkownika o nowe elementy.

Na listingu 31.34 prezentujemy plik manifest tej aplikacji.

---

**Listing 31.34.** Plik `AndroidManifest.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidbook.drag.drop.demo"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk android:minSdkVersion="11" />
    <application android:icon="@drawable/icon"
```

---

```

        android:label="@string/app_name">
<activity android:name=".MainActivity"
          android:label="@string/app_name">
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

</application>
</manifest>

```

---

Na listingu 31.35 pokazaliśmy zawartość pliku *strings.xml*, w którym umieściliśmy ciągi znaków wykorzystywane przez plik manifest.

**Listing 31.35.** Plik strings.xml

---

```

<?xml version="1.0" encoding="utf-8"?>
<!-- Jest to plik res/values/strings.xml -->
<resources>
    <string name="app_name">Fragmenty używane w funkcji przeciągania</string>
</resources>

```

---

W ten sposób uzyskaliśmy już wszystkie pliki i informacje wymagane do skompilowania i wdrożenia naszej przykładowej aplikacji wykorzystującej funkcję przeciągania.

## Testowanie przykładowej aplikacji wykorzystującej funkcję przeciągania

Poniżej przedstawiamy przykładową zawartość okna *LogCat*, w którym są wyświetlane informacje pojawiające się po włączeniu naszej przykładowej aplikacji. Zwróćmy uwagę, że do identyfikacji komunikatów dotyczących niebieskiego kółka wykorzystujemy znacznik **Niebieskie kolko**, do określenia białego kółka stosujemy nazwę **Biale kolko**, a zdarzenia dotyczące strefy upuszczania są definiowane za pomocą nazwy **DropTarget**.

---

Biale kolko:	rozpoczeto proces przeciągania? true
Niebieskie kolko:	To zdarzenie przeciągania nie jest przeznaczone dla nas
Biale kolko:	sekwencja przeciągania rozpoczęta. X: 53.0, Y: 206.0
DropTarget:	Sekwencja przeciągania rozpoczęta w dropTarget
DropTarget:	Sekwencja przeciągania wkroczyła do dropTarget
DropTarget:	Sekwencja przeciągania kontynuuje w dropTarget: 29.0, 36.0
DropTarget:	Sekwencja przeciągania kontynuuje w dropTarget: 48.0, 39.0
DropTarget:	Sekwencja przeciągania kontynuuje w dropTarget: 45.0, 39.0
DropTarget:	Sekwencja przeciągania kontynuuje w dropTarget: 41.0, 39.0
DropTarget:	Sekwencja przeciągania kontynuuje w dropTarget: 40.0, 39.0
DropTarget:	Upuszczenie obiektu w dropTarget
DropTarget:	Elementem danych jest Biale kolko
ViewRoot:	Reporting drop result: true
Biale kolko:	przeciąganie zakonczone. Sukces? true
Niebieskie kolko:	To zdarzenie przeciągania nie jest przeznaczone dla nas
DropTarget:	Sekwencja przeciągania zakonczona w dropTarget

---

W tym konkretnym przypadku sekwencja przeciągania została rozpoczęta przez białe kółko. Po uruchomieniu całej procedury przez długie kliknięcie otrzymujemy wiadomość *rozpoczęto proces przeciagania?*.

Zwróćmy uwagę, że w trzech następnych linijkach pojawia się informacja o odebraniu działania ACTION\_DRAG\_STARTED. Okazuje się, że wywołanie metody zwrotnej nie było przeznaczone dla niebieskiego kółka. Jego adresatem nie był również obszar dropTarget.

Zauważmy następnie, że kolejne komunikaty opisują zachowanie sekwencji przeciągania w obszarze dropTarget, począwszy od działania ACTION\_DRAG\_ENTERED. Oznacza to, że kółko zostało przeciągnięte nad obszar zielonego kwadratu. Współrzędne X i Y są mierzone względem lewego górnego rogu tego widoku. Zatem pierwszy wpis sekwencji przeciągania przeprowadzanej w zielonym obszarze posiada współrzędne (x, y) = (29, 36), a upuszczenie nastąpiło w punkcie (40, 39). Jak widać, widok docelowy potrafi wydobyć nazwę znacznika definiującego białe kółko i umieścić ją w oknie LogCat.

Ponownie zwracamy uwagę, że wszystkie obiekty nasłuchujące odebrały działanie ACTION\_DRAG\_ENDED, ale jedynie białe kółko pozwoliło na wyświetlenie wyników za pomocą metody getResults().

Zachęcamy do eksperymentowania z tą przykładową aplikacją. Przeciągnijmy jedno kółko nad drugie, a nawet na jego oryginalną wersję. Dodajmy jeszcze jedno kółko w pliku *palette.xml*. Zwróćmy uwagę, że po wyjściu z zielonego obszaru pojawia się odpowiedni komunikat w oknie LogCat. Warto także odnotować fakt, że jeżeli upuścimy kółko gdziekolwiek poza zielonym obszarem, sekwencja przeciągania zostanie uznana za nieudaną.

## Odbońniki

- <http://developer.android.com/sdk/android-3.0-highlights.html> — wymieniono tu nowe funkcje wprowadzone w wersji 3.0 Androida, a także odnotowano informacje o zmianach wprowadzonych w widżetach ekranu startowego.
- [www.androidbook.com/item/3624](http://www.androidbook.com/item/3624) — nasze notatki robocze, powstałe na etapie zbierania materiałów do niniejszego rozdziału, dotyczące widżetów ekranu startowego dostępnych w wersji 3.0 Androida. Znajdziemy tu odnośniki do interfejsów API, fragmentów kodów, otwartych pytań oraz dalszych badań.
- [www.androidbook.com/item/3299](http://www.androidbook.com/item/3299) — nasze notatki robocze, powstałe na etapie zbierania materiałów do niniejszego rozdziału, dotyczące widżetów ekranu startowego dostępnych w wersji 2.2 Androida. Znajdziemy tu odnośniki do interfejsów API, fragmentów kodów, otwartych pytań oraz poprzednich badań.
- [www.androidbook.com/item/3637](http://www.androidbook.com/item/3637) — nasze dodatki dotyczące klasy RemoteViews, zaktualizowane o informacje związane z wersją 3.0 Androida, w tym również przykładowe kody, kolejne pytania, a także zewnętrzne i wewnętrzne odniesienia.
- <http://developer.android.com/guide/topics/appwidgets/index.html> — główne źródło informacji na temat poprzednich wersji widżetów ekranu startowego. Zwróćmy uwagę, że nie ma tu wzmianki na temat zmian wprowadzonych w wersji 3.0 Androida, zapoznamy się tu jednak z podstawowymi mechanizmami.
- <http://developer.android.com/reference/android/appwidget/AppWidgetManager.html> — dokumentacja bardzo istotnej klasy AppWidgetManager.

- <http://developer.android.com/reference/android/widget/RemoteViewsService.RemoteViewsFactory.html> — dokumentacja klasy RemoteViewsFactory.
- <http://developer.android.com/reference/android/widget/RemoteViews.html> — dokumentacja klasy RemoteViews.
- <http://developer.android.com/reference/android/widget/RemoteViewsService.html> — dokumentacja usługi RemoteViewsService.
- <ftp://ftp.helion.pl/przykłady/and3ta.zip> — łącze, dzięki któremu możemy pobrać projekty utworzone na potrzeby niniejszej książki. Nazwy katalogów zawierających te projekty to *ProAndroid3\_R31\_WidżetyLista* oraz *ProAndroid3\_R31\_TestPrzeciągania*.

## Podsumowanie

W rozdziale tym omówiliśmy dwa istotne usprawnienia wprowadzone w wersji 3.0 Androida: widżety ekranu startowego oparte na listach oraz interfejs przeciągania.

Najpierw dokładnie przeanalizowaliśmy modyfikacje wprowadzone w widżetach ekranu startowego. Pokazaliśmy, w jaki sposób wczytywać i zapełniać zdalne widoki zawierające listy poprzez wykorzystanie usługi zdalnych widoków oraz klasy fabrykującej. Dowiedzieliśmy się także, jak możemy konfigurować zdarzenia onClick oraz manipulować samą klasą AppWidgetProvider w celu otrzymywania tych zdarzeń oraz reagowania na nie. Zaprezentowany przez nas materiał pozwala na implementowanie bogatych, użytecznych widżetów ekranu startowego.

W podrozdziale poświęconym funkcji przeciągania zajęliśmy się wszelkimi dostępnymi nowinkami wykorzystywanymi w czasie sekwencji przeciągania, dzięki którym użytkownik ma do dyspozycji jeszcze większe możliwości, porównywalne z osiąganymi na komputerach stacjonarnych. Przekazaliśmy wiele informacji na temat źródeł, zdarzeń oraz obiektów docelowych sekwencji przeciągania.

Potencjał kryjący się w tych funkcjach jest ograniczony jedynie wyobraźnią Czytelników.



# Skorowidz

3GPP, 3rd Generation Partnership Project, 625

## A

AAC, Advance Audio Coding, 626

AAPR, Android Asset Packaging Tool, 69

abstrakcja, 472

adapter

ArrayAdapter, 203

ManateeAdapter, 218

ManateeAdapter, 218

niestandardowy, 218

SimpleCursorAdapter, 200

standardowy, 218

ADB, Android Debug Bridge, 83

ADP, Android Developer Phone, 1009

ADT, Android Development Tools, 38, 51

AGC, Automatic Gain Control, 625

agregacja, 970, 1001

akcelerometr, 924

mierzenie kątów, 930

pomiar grawitacji, 927

składowa ruchu, 929

tryb krajobrazowy, 925

tryb wyświetlania, 926

współpraca z magnetometrami, 931

współrzędne, 924

aktualizacja lokacji, 569

aktualizowanie danych, 138

aktualizowanie widoku, 742

aktywne foldery, 42

AllContactsLiveFolderCreatorActivity.java,  
725

AndroidManifest.xml, 723

BetterCursorWrapper.java, 732

folder kontaktów, 721

lista, 720

MyContactsProvider.java, 726

MyCursor.java, 731

rejestrowanie identyfikatora URI, 730

testowanie, 733

tworzenie folderu, 722

aktywność, 39, 59, 428

ACTION\_DIAL, 159

animacji widoku, 537

DetailsActivity, 1053

klienta nadawczego, 457

konfiguradora widżetu, 739

ListActivity, 209, 210

LocalSearchEnabledActivity, 794, 795

MainActivity, 375

MultiViewTestHarness, 669

NotesList, 72

OpenGL20MultiViewTestHarness, 705

paska działania, 1096

klasa bazowa, 1085

pliki projektu, 1081

paska działania wyświetlającego zakładki,

1081

paska zakładek, 1088

PreferenceActivity, 304, 312

przechowująca pasek działania, 1098, 1099

RegularActivity, 778

SearchActivity, 791, 792, 795

TestHandlersDriverActivity, 446

TestOpenGLMainDriver, 670

ViewAnimationActivity, 537

wyłączająca wyszukiwanie, 786

wyszukiwania

kod źródłowy, 804

wyszukiwania globalnego, 770

wyswietlająca pasek zakładek, 1092

wywołania wyszukiwania, 808

- aktywność
  - wywołująca klasę AsyncTask, 355
  - z paskiem zakladek, 1080
  - z rozwiniętą listą, 1098
- aktywny folder, live folder, 717
- alarm powtarzalny, 504
  - anulowanie, 506
  - kompilowanie kodu, 506
- alert odległościowy, 578
- aliasy kolumn w strukturach danych kontaktu, 1000
- AMDDA, Android Market Developer Distribution Agreement, 1006
- AMR, Adaptive Multi-Rate, 626
- analiza baz danych, 121
- analiza wbudowanych dostawców treści, 120
- Android Debug Bridge, 121
- Android Developer Phone, 1009
- Android Market, 319, 1005, 1022
  - alternatywy dla serwisu, 1023
  - katalog Moje zamówienia, 1022
- Android SDK, 32, 56
  - android.app, 44
  - android.bluetooth, 44
  - android.content, 44
  - android.content.pm, 44
  - android.content.res, 44
  - android.database, 45
  - android.database.sqlite, 45
  - android.gesture, 45
  - android.graphics, 45
  - android.graphics.drawable, 45
  - android.graphics.drawable.shapes, 45
  - android.hardware, 45
  - android.location, 45
  - android.media, 45
  - android.net, 45
  - android.net.wifi, 45
  - android.opengl, 46
  - android.os, 46
  - android.preference, 46
  - android.provider, 46
  - android.sax, 46
  - android.speech, 46
  - android.speech.tts, 46
  - android.telephony, 46
- android.telephony.cdma, 46
- android.telephony.gsm, 46
- android.text, 46
- android.text.method, 47
- android.text.style, 47
- android.utils, 47
- android.view, 47
- android.view.animation, 47
- android.view.inputmethod, 47
- android.webkit, 47
- android.widget, 47
- animacja, 517
  - klatek kluczowych, 524
  - poklatkowa, 518
    - dodawanie animacji do aktywności, 520
    - lista klatek, 521
    - planowanie, 518
    - środowisko testowe, 519, 522
    - tworzenie aktywności, 519
    - układ graficzny, 519
  - rotacyjna, 524, 531
  - skali, 524
    - definiowanie w pliku XML, 528
- translacyjna, 524, 531
- typu alfa, 524, 530
- układu graficznego, 523
  - kod aktywności, 527
  - środowisko testowe, 525
  - tworzenie aktywności, 525
  - widok ListView, 525
- w osi X oraz w wymiarze alfa, 1077
- widoku, 533
  - aktywność, 534
  - dodawanie animacji, 536
  - kod aktywności, 535
  - kod źródłowy aktywności, 537
  - metody preTranslate i postTranslate, 538
  - układ graficzny, 534
- widoku ListView, 540
- animator niestandardowy, 1076
- animowane przejścia, tweening, 42
- animowanie obiektów, 691
- animowanie trójkąta, 676
- animowanie widoku ListView, 528
- ANR, Application Not Responding, 351, 427
- API Google Maps, 52

- aplikacja  
adb, 123  
Android Debug Bridge, 121  
Android Hierarchy, 57  
BDayWidget, 747  
Browser, 38  
Contacts, 38  
Downloads, 367  
Gestures Builder, 898, 901  
HelloAndroidApp, 65  
Home, 38  
Hierarchy Viewer, 242  
J2EE, 68  
keytool, 318, 319  
Kontakty, 958  
Lista czujników, 909  
    kod Java, 909  
    układ graficzny, 909  
    wyniki, 911  
MapViewDemo, 549, 889  
Monitor akcelerometru, 918  
Monitor czujnika oświetlenia, 911  
NfcDemo, 950  
Notepad, 69  
    analiza kodu, 71  
    tworzenie, 70  
    uruchomienie, 69  
    wczytywanie, 70  
obsługująca mapy, 548  
obsługująca multimedia, 606  
Package Browser, 409  
Phone, 38  
poczta, 593  
przedstawiająca mapę świata, 157  
przeglądarki stron, 157  
rejestrująca, 631  
SDK Manager, 54  
SipDemo, 599  
sługa do przeciągania obiektów, 876  
sługa do tłumaczeń, 397  
    główny plik aplikacji, 399  
    plik AIDL usługi tłumacza, 399  
    plik AndroidManifest.xml, 403  
    plik usługi tłumaczenia, 402  
    plik z ciągami znaków, 399  
    plik z funkcją tłumaczenia, 402  
    plik z tablicami, 399  
układ graficzny, 398  
StreetView, 890  
Terminal, 56  
TouchDemo1, 863  
    główna aktywność, 866  
    interfejs użytkownika, 864  
    kod Java, 865  
    komunikaty, 867  
    układ graficzny, 863  
ukazująca zastosowanie fragmentów, 1034  
umożliwiająca połączenie telefoniczne, 157  
Ustawienia, 954  
Ustawienia wyszukiwania, 775  
wykrywająca gesty, 902  
wyświetlająca klawiaturę, 157  
zawierająca szczegółowe mapy, 157  
aplikacje Androida, 48  
architektura REST, 119  
argument  
    @avdname, 122  
    CHECK\_VOICE\_DATA\_PASS, 845  
    fill\_parent, 182  
    metody startSearch(), 839  
    minSdkVersion, 182  
ARM, Advanced RISC Machine, 39  
artefakty usługi i klienta, 395  
atrybut android  
    gravity, 231  
    layout\_gravity, 231  
    layout\_span, 234  
    padding, 235  
    permission, 334  
    prompt, 215  
    readPermission, 334  
    text, 181  
    writePermission, 334  
atrybut  
    FadeOffset, 901  
    id, 97  
    quantity, 102  
    queryAfterZeroResults, 822  
    REFERER, 396  
    ringtoneType, 309  
    searchSuggestIntentData, 822  
    showSilent, 309  
    suggestActionMsgColumn, 837

atrybuty

uprawnienia, 329

węzła danych, 160

AVD, Android Virtual Device, 51, 60, 65

## B

badanie aktywności zawierającej pasek

działania, 1098, 1102

badanie kontaktów zbiorczych, 980

elementy menu, 986

funkcje użytkowe, 980

główna aktywność, 986

klasa bazowa, 981

kolumny kurSORA, 988

pliki projektu, 980

testowanie kontaktów, 982

badanie nieprzetworzonych kontaktów, 989

opcje menu, 990

pliki projektu, 989

pola kurSORA, 993

przeglądanie danych, 994

testowanie danych kontaktu, 995

testowanie kontaktów, 990

baza danych contacts.db, 133

baza danych SQLite, 119, 125

bezpieczeństwo

certyfikat cyfrowy, 318

definiowanie uprawnień, 334

klucz prywatny, 318

klucz publiczny, 318

kontrolowanie dostępu do zasobów, 334

magazyn kluczy, 318

podpisywanie aplikacji, 318

przekazywanie uprawnień, 333

biblioteka

FreeType, 37, 38

języka C, 37

OpenGL, 37

Skia, 38

Surface Manager, 38

WebKit, 37, 38

biblioteka gestów, 900

biblioteka OpenGL, 649

animowanie trójkątów, 676

dodawanie trójkątów, 675

glClear, 658

glColor, 659

glDrawElements, 656

glFrustum, 661

gluLookAt, 660

glVertexPointer, 654

glViewport, 663

koncepcja widzenia, 660

ksztalty, 678

objętość widzenia, 661

OpenGLTestHarnessActivity, 667

podstawy rysowania, 654

RegularPolygon, 681

rozmiar ekranu, 663

rysowanie prostokąta, 679

rzutowanie ortograficzne, 662

rzutowanie perspektywiczne, 662

teksty, 678, 694

trójwymiarowa scena, 659

tworzenie trójkątów, 676

wielobok foremny, 681

biblioteka OpenGL ES, 649

proste ksztalty, 654

biblioteka OpenGL ES 2.0, 704

aktywność sterująca, 706

dostęp do jednostek cieniowania, 711

funkcje, 706

jednostki cieniące, 708

jednostki cieniące wierzchołki, 708

jednostki cieniące fragmenty, 708

renderowanie, 707

trójkąt, 711

źródła, 715

biblioteki Apache HTTP, 48

biblioteki Java AndroIDA, 32

biblioteki multimedialnych, 37

blokada przechodzenia w stan zatrzymania, 477, 486

błąd komplikacji we wtyczce ADT, 111

błędy w aplikacji, 81

BSD, Berkeley Software Distribution, 37

bufor koloru, 658

bufor nio, 655, 658, 666

## C

CA, certificate authority, 318

CAD, Computer Aided Design, 650

cal, 235

Centrum Oprogramowania Linuksa, 53

certyfikat  
 Android Debug, 321  
 cyfrowy, 318  
 debugowania, 320  
     okres ważności, 324  
 PKI, 412  
 podpisany samoistnie, 318  
 produktu, 1018  
 programistyczny, 320  
 testowy, 546  
 własny, 318  
 X.509, 318

**CHOICE\_MODE\_MULTIPLE**, 211

**CHOICE\_MODE\_NONE**, 211

**CHOICE\_MODE\_SINGLE**, 211

ciąg znaków, 115  
 ciężar, weight, 228  
 com.google.android.maps, 47  
 CRUB, Create, Read, Update, Delete, 77  
 cyfrowy podpis aplikacji, 322  
 cykl życia  
     aktywności, 446, 447  
     aplikacji, 78  
     dostawcy treści, 448  
     odbiorców komunikatów, 448  
     usługi, 448  
 czas życia procesu, 446  
 czas życia składnika, 446  
 częstotliwość próbkowania, 628  
 czujnik  
     aktualizacje odczytów, 915  
     interpretacja danych, 921  
     pobieranie zdarzeń, 911  
     pozostawianie włączonego ekranu, 920  
     rodzaje czujników, 908  
     rozwiązywanie problemów, 915  
     termometry, 922  
     trudne problemy, 914  
     wybór wartości odświeżania, 913  
     wykrywanie, 908  
 czujnik NFC, *Patrz* NFC, 907  
 czujniki  
     akcelerometry, *Patrz* akcelerometr, 924  
     ciśnienia, 923  
     grawitacji, 939  
     informacje o położeniu, 932

komunikacji bliskiego pola, 939  
 magnetometry, 930  
 orientacji w przestrzeni, 931  
 oświetlenia, 921  
 przyśpieszenia liniowego, 939  
 wektora obrotu, 939  
 zbliżeniowe, 922  
 żyroskopy, 923

## D

Dalvik VM, 33, 36  
 dane, 159, 167  
 dane dodatkowe, 161  
 dane nieprzetworzonego kontaktu, 994  
 dane typu Bundle, 568  
**DATABASE\_MODE\_2LINES**, 801  
**DATABASE\_MODE\_QUERIES**, 801  
 DDMS, Dalvik Debug Monitor Server, 82  
 Debug, 82  
 debugger `debug_layout_activity.xml`, 977  
 debugowanie, 293, 1090  
 debugowanie aplikacji, 82  
 definicja  
     nieregularnej tabeli, 232  
     tabeli danych kontaktów, 968  
     tabeli kontaktów zbiorczych, 969  
     tabeli nieprzetworzonych kontaktów, 966  
 definiowanie  
     działania w dostawcy widżetu, 1118  
     kontrolki GridView, 214  
     prostokąta o zaokrąglonych rogach, 109  
     wielokrotności, 102  
     zasobów typu Color, 105  
     zasobów typu color-drawable, 108  
     zasobów typu dimension, 106  
     zasobów typu string, 103  
 deklarowanie niestandardowego  
     uprawnienia, 328  
 deklinacja magnetyczna, 938  
 Developer Account, 1006  
 dip, 235  
 długie kliknięcie, 261  
 długoterminowa usługa, 473  
 długoterminowy odbiorca komunikatów, 476

- dodawanie  
  animacji do widoku, 536  
  elementów do menu, 249  
  funkcji dotyku, 889  
  identyfikatorów do kontrolek, 181  
  kontaktu, 998  
plików do magazynu multimediiów, 644  
plików dźwiękowych do silnika TTS, 854  
pliku do dostawcy treści, 137  
trójkątów za pomocą indeksów, 675  
typów do obiektu, 161  
znaczników, 553  
dokumentacja pakietu SDK, 412  
dostawca  
  BookProvider, 141  
    dodawanie książki, 150  
    usuwanie książki, 150  
    wyświetlanie listy książek, 151  
    zliczanie książek, 151  
  Contacts, 131  
  GPS\_PROVIDER, 577  
  lokalizacji, 567  
  MediaStore, 131  
  NETWORK\_PROVIDER, 577  
  PASSIVE\_PROVIDER, 577  
  pasywny, 567  
  położenia, 568  
  propozycji, 798  
    aktywność wyszukiwania, 803  
    główna aktywność, 810  
    manifest zawierający definicję, 802  
    pliki implementacji, 799  
    pole wyszukiwania lokalnego, 811  
    propozycja lokalna, 812  
    propozycje globalne, 813  
    użytkowanie, 810  
    włączanie, 812  
  wyniki wyszukiwania lokalnego, 811  
  zadania, 800  
propozycji niestandardowej  
  badanie metadanych, 821  
  identyfikatory URI, 818  
  implementacja, 813  
  implementacja aktywności  
    wyszukiwania, 824  
  klasa SearchActivity, 825  
  korzystanie, 831  
metadane wyszukiwania, 820  
plik manifest, 830  
przekazywanie kwerendy, 820  
wyniki, 832  
wywołanie SearchActivity, 827  
zakończenie aktywności, 829  
SearchRecentSuggestionsProvider, 806  
sieciowy, 567  
SimpleSuggestionProvider, 804  
  metadane wyszukiwania, 807  
SuggestUrlProvider, 814  
  implementacja klasy, 815  
  kod źródłowy, 815  
  pliki projektu, 814  
systemu GPS, 567  
treści, 41, 59, 119  
  analiza wbudowanych dostawców, 120  
architektura, 126  
baza danych, 139  
  dodawanie pliku, 137  
  identyfikator URI, 128  
  implementacja, 139  
  odczyt danych, 130  
  rejestrowanie dostawcy, 149  
  widżetu, widget provider, 736  
dostępność, 704  
dp, 235  
dynamiczne dane, 180  
działanie, 159, 167  
  ACTION\_DOWN, 862  
  ACTION\_DRAG\_ENDED, 1144  
  ACTION\_DRAG\_STARTED, 1144  
  ACTION\_GET\_CONTENT, 171  
  ACTION\_MOVE, 862, 894  
  ACTION\_NDEF\_DISCOVERED, 942  
  ACTION\_PICK, 169  
  ACTION\_POINTER\_DOWN, 894  
  ACTION\_POINTER\_UP, 894  
  ACTION\_SEARCH, 828, 833  
  ACTION\_TAG\_DISCOVERED, 942  
  ACTION\_TECH\_DISCOVERED, 942  
  ACTION\_UP, 862  
  ACTION\_VIEW, 828, 833  
CALL, 159  
Intent.ACTION\_CALL, 160  
Intent.ACTION\_DIAL, 160  
VIEW, 825

**E**

Eclipse, 38, 51  
 EditText, 184  
 edycja kontaktu, 960  
 ekran dotykowy, 861  
 ekran preferencji, 297  
 ekran startowy, 1110  
 eksploracja kont, 972  
 funkcje testujące, 973  
 główna aktywność sterująca, 977  
 plik manifest, 978  
 plik menu, 973  
 spis plików, 978  
 element Activity.managedQuery, 74  
 element Dot, 876  
 element nasłuchujący, 192  
 element SimpleCursorAdapter, 74  
 element TextView, 94  
 elementy składowe Androida, 68  
 AndroidManifest.xml, 68  
 anim, 68  
 assets, 68  
 drawable, 68  
 layout, 68  
 menu, 68  
 raw, 68  
 res, 68  
 src, 68  
 values, 68  
 xml, 68  
 emulacja karty NFC, 949  
 emulacja rozruchu urządzenia, 64  
 emulator Androida, 38  
 EULA, 1019  
 EXTRA\_EMAIL., 162  
 EXTRA SUBJECT, 162

**F**

filtr intencji, 160, 166, 943, 945  
 flaga  
 CONTEXT\_IGNORE\_SECURITY, 414  
 CONTEXT\_INCLUDE\_CODE, 414  
 CONTEXT\_RESTRICTED oznacza, 414  
 Menu.FLAG\_APPEND\_TO\_GROUP, 266  
 krawędzi, 869

nietrwałości, 485  
 Service.START\_REDELIVER, 485  
 Service.START\_STICKY, 485  
 trwałości, 485  
 folder  
 android\AVD, 65, 66  
 assets, 69  
 drawable, 897  
 raw, 69  
 res, 69, 74  
 res/layout-land, 240  
 res/layout-port, 240  
 res/layout-square, 240  
 układu graficznego res/layout, 240  
 values, 74  
 foldery wiadomości SMS, 592, 593  
 format dźwięku 3GPP, 625  
 format VCF, 963  
 fragment, 40, 1026  
 aplikacja ukazująca cykl życia, 1033  
 cykl życia, 1028  
 formy komunikowania, 1073  
 komunikacja pomiędzy fragmentami, 1074  
 metoda onAttach(), 1030  
 metoda onCreate(), 1030  
 metoda onCreateView(), 1031  
 metoda onDestroyView(), 1033  
 metoda onDetach(), 1033  
 metoda onInflate(), 1030  
 metoda onPause(), 1032  
 metoda onResUME(), 1032  
 metoda onStart(), 1032  
 metoda onStop(), 1032  
 metoda setRetainInstance(), 1033  
 metoda zwrotna onActivityCreated(), 1032  
 przechowujący dialog, 1055  
 przejścia i animacje, 1044  
 stosowanie, 1027  
 stosowanie odniesień, 1046  
 struktura, 1027  
 transakcja fragmentu, 1042  
 trwałość, 1054  
 tworzenie hierarchii widoków, 1031  
 wycofanie okna dialogowego, 1059  
 wyświetlanie okna dialogowego, 1054  
 wyświetlanie nowej aktywności, 1051

- fragment wyświetlający okna dialogowe, 1060  
główna aktywność, 1061  
główny układ graficzny, 1072  
interfejs użytkownika, 1061  
kod Java, 1064  
pliki projektu, 1060  
układ graficzny, 1064
- funkcja  
debugowania, 84  
getACursor(), 982  
getDistinctPendingIntent(), 510  
getEventsFromAnXMLFile, 111  
getExternalStorageDirectory(), 607  
getExtras, 161  
Install New Software..., 56  
java.util.AttributeSet, 110  
listContacts(), 988  
ProGuard, 1018  
przeciągania, 876, 1131  
    główna aktywność, 1136  
    interfejs aplikacji, 1134  
    lista plików, 1133  
    podstawowe informacje, 1131  
    przykładowa aplikacja, 1133  
    testowanie aplikacji, 1145  
    tworzenie układu graficznego, 1133  
    układ graficzny, 1134
- putExtras, 161  
setData(), 435  
StrictMode, 85, 89  
StrictMode w trybie debugowania, 86  
testThread(), 437  
TTS, 843  
type-to-search, 797  
wielodotykowości, 879, 887
- funkcje  
daty, 760  
kalendarza, 494  
użytkowe, 980
- G**
- geokodowanie w Androidzie, 559, 563  
geokodowanie w oddzielnym wątku, 563  
geolokalizacja, 559
- gest ścisania, 891  
kod Java, 892, 896  
ScaleGestureDetector, 896  
układ graficzny, 896
- gest zaznaczenia, 899  
gesty, 891  
    aplikacja, 902  
    biblioteka, 900  
    kod Java aplikacji, 902  
    magazynie, 900  
    rejestrowanie, 905  
    struktura klas, 900  
    układ graficzny aplikacji, 902
- gesty niestandardowe, 898  
gesty właściwe, 900, 904  
glClear  
    zerowanie koloru, 658
- glDrawElements  
    konceptja pasa, 657  
    konceptja wachlarza, 657  
    ksztalt, 657  
    kwadrat, 657  
    linia, 657  
    pas linii, 657  
    pętle linii, 657  
    punkt, 657  
    trójkąty, 657
- glFrustum  
    bliski punkt, 662  
    daleki punkt, 662  
    objętość widzenia, 662  
    ostrosłup widzenia, 662  
    promień, 662
- gluLookAt  
    orientacja aparatu, 661  
    punkt oczny, 660  
    punkt spoglądania, 661  
    punkt widoku, 661  
    wektor góry, 661  
    współrzędne świata, 660
- glVertexPointer  
    bryła okalająca, 655  
    interfejs API, 656  
    objętość okalająca, 655  
    współrzędne świata, 655

glViewport  
wziernik, 663  
głębia w obrazie dwuwymiarowym, 539  
Google Checkout, 1022  
Google Maps, 546  
Google Nexus One, 1009  
Google Nexus S. Android Developer Phone, 1009  
GPS, Global System Positioning, 545  
GPU, Graphical Processing Unit, 649  
grafika trójwymiarowa, 652  
grawitacja, gravity, 228  
grupa opcji, 193  
grupy widoków, 39  
GSM, Global System for Mobile Communication, 38

**H**

handheld, 32  
Hashimi Sayed Y., 23  
hasła storepass i keypass, 321  
hiperbola, 532  
historia Androida, 34

**I**

IANA, Internet Assigned Numbers Authority, 129  
IDE, Integrated Development Environment, 38  
identyfikator  
    Contacts.People.CONTENT\_URI, 131  
    predefiniowany, 526  
    public static, 74  
    treści nieprzetworzonego kontaktu, 997  
    układu graficznego, 527  
URI, 76, 128, 130, 332  
    klasa UriMatcher, 147  
    rozpoznawanie kolekcji, 140  
    rozpoznawanie URI, 147  
    wprowadzanie klauzuli WHERE, 134  
    wstawianie rekordów, 136  
URI propozycji, 815  
URI wyszukiwania, 815  
URI wyszukiwania kontaktów, 989  
zasobów, 97, 114  
    t1\_1\_en\_port, 115  
    t1\_enport, 114  
    t2, 115

testport\_port, 114  
teststring\_all, 114, 116  
zasobów R.menu.moje\_menu, 269  
IDialogProtocol, 288  
IETF, Internet Engineering Task Force, 597  
ikony akustyczne, 855  
implementacja  
    aktywności wyszukiwania, 824  
    bazowych klas aktywności, 1082  
    długoterminowej usługi, 483, 486  
    dostawców treści, 139  
    dostawcy treści BookProvider, 141  
    dostawcy widżetu, 751  
    interfejsu AIDL, 379  
    interfejsu AnimationListener, 541  
    interfejsu Parcelable, 386  
    klasy AlarmManagerService, 514  
    klasy Renderer, 664  
    klasy ReportStatusHandler, 439  
    klasy WorkerThreadRunnable, 438  
    ksztaltu RegularPolygon, 682, 683  
    metody delete, 147  
    metody getType(), 819  
    metody insert, 147  
    metody query, 146  
    metody update, 147  
    modeli widżetów, 753  
    niestandardowego dostawcy propozycji, 814  
    obiektu nasłuchującego zdarzeń, 1087  
    oświetlonego zielonego pokoju, 478  
    usługi lokalnej, 373  
        plik AndroidManifest.xml, 374  
        plik main.xml, 373  
        plik MainActivity.java, 373  
    usługi StockQuoteService2, 388  
        wątku roboczego, 438  
instalacja narzędzi ADT, 57  
instalowanie aktualizacji, 324  
instancja paska działania, 1089  
instancja widżetu, 738  
instrukcja awk, 123  
instrukcja create, 125  
instrukcja find, 123  
instrukcja grep, 123  
instrukcja insert, 137

intencja, intent, 40, 59, 155, 156  
  atrybuty dodatkowe, 161  
  dane, 159, 167  
  działanie, 159, 167  
  jawna nazwa klasy, 159  
  kategorie intencji, 168  
  mapa typu klucz – wartość, 159  
  MediaStore.ACTION\_IMAGE\_CAPTURE,  
    644  
  niejawnia, 159  
  oczekująca, pending intent, 172  
  PendingIntent, 495  
  przydzielanie do ich składników, 166  
  putExtras, 161  
  schemat danych, 167  
  ścieżka danych, 168  
  typ danych, 167  
  uprawnienia do danych, 168  
  VIEW, 160  
interakcja aktywności wyszukiwania lokalnego, 792  
interakcja aktywności z przyciskiem wyszukiwania, 777  
interfejs  
  ActionBar, 1079  
  AJAX Language, 396  
  API, 43, 653  
  API Google Maps, 396  
  API multimedii, 619  
  aplikacji obsługującej multimedia, 606  
  AttributeSet, 1030  
  ContactsContract, 967  
  createPackageName(), 413  
  DDMS, 571  
  DialogInterface, 276  
  DialogInterface.OnClickListener, 1072  
  EGL, 651  
  glDrawElements, 654  
  GLSurfaceView.Renderer, 663  
  Google AJAX Language API, 396  
  Google Directions, 569  
  graficzny  
    pojemnik, kontener, 176  
    układ graficzny, 176  
    widok, widżet, kontrolka, 176  
IDialogFinishedCallBack, 291

IDialogProtocol, 286, 288  
JDBC, 127  
kontaktów, 972  
modelu widżetu, 753  
nasłuchujący AnimationListener, 541  
obiektu nasłuchującego, 1063  
OnCheckedChangeListener, 193  
OnDialogDoneListener, 1063, 1067  
onItemClickListener, 208  
onLoadCompleteListener, 616  
pomiędzy OpenGL ES a Androidem, 663  
Projection, 888  
rejestratora wideo, 632  
Renderer, 664  
ResolveInfo, 164  
SensorEventListener, 913  
Shape, 682  
Tłumacz Google, 395, 397  
UI, 175, *Patrz* interfejs użytkownika  
UI Androida, 39  
UI Emulator Control, 591  
UI kontrolek DatePicker i TimePicker, 198  
UI odtwarzacza plików wideo, 620  
UI widoku RingtonePreference, 308  
użytkownika, 39  
  konstruowanie interfejsu za pomocą kodu oraz języka XML, 180  
  programowanie za pomocą kodu, 177  
  projektowanie interfejsu, 176  
  tworzenie interfejsu w pliku XML, 179  
  użytkownika silnika TTS, 845  
interpolator, 531, 532  
interpolator liniowy, linear interpolator, 1075  
interpolator accelerate\_interpolator, 532  
IPC, Inter-Process Communication, 37

## J

J2EE, Java 2 Platform Enterprise Edition, 58  
Java Development Kit, 52  
Java Standard Edition, 32  
jawna nazwa klasy, 159  
jawne przywołanie aktywności, 447  
JCP, Java Community Process, 651  
JDK, Java Development Kit, 52, 53, 56  
JDK, Java SE Development Kit, 51  
język AIDL, 368

język HTML, 91  
 język XML, 69  
 język XUL, 39  
 JIT, Just-In-Time Compiler, 36  
 JRE, Java Runtime Environment, 52  
 JSON, JavaScript Object Notation, 343  
 JVM, Java Virtual Machine, 32

**K**

kamera  
 zmiana ustawień, 672  
 karta SD, 601, 602  
 foldery, 605  
 źródło plików audio, 612  
 katalog  
 /res, 203  
 /res/layout/, 95  
 /res/menu, 269  
 /res/values, 1143  
 /res/xml, 297  
 /tools, 242  
 assets, 111  
 DCIM, 604  
 drawable-port, 240  
 drawable-square, 240  
 frameworks/base, 49  
 HOME, 53  
 layout, 69  
 layout-en, 114  
 Moje zamówienia, 1022  
 par klucz – wartość, 136  
 rawable-land, 240  
 tools, 55, 122  
 tools/android, 54  
 katalogi alternatywnych zasobów, 113  
 katalogi na karcie SD  
 DIRECTORY\_ALARMS, 605  
 DIRECTORY\_DCIM, 605  
 DIRECTORY\_DOWNLOADS, 605  
 DIRECTORY\_MOVIES, 605  
 DIRECTORY\_MUSIC, 605  
 DIRECTORY\_NOTIFICATIONS, 605  
 DIRECTORY\_PICTURES, 605  
 DIRECTORY\_PODCASTS, 605  
 DIRECTORY\_RINGTONES, 605

katalogi zasobów, 112  
 kategoria LAUNCHER, 304, 1053  
 kategorie aktywności  
 CATEGORY\_ALTERNATIVE, 165  
 CATEGORY\_BROWSABLE, 165  
 CATEGORY\_DEFAULT, 165  
 CATEGORY\_EMBED, 165  
 CATEGORY\_GADGET, 165  
 CATEGORY\_HOME, 165  
 CATEGORY\_LAUNCHER, 165  
 CATEGORY\_PREFERENCE, 165  
 CATEGORY\_SELECTED\_ALTERNATIVE,  
 165  
 CATEGORY\_TAB, 165  
 CATEGORY\_TEST, 165  
 kategorie intencji, 163, 168  
 klasa  
 AbstractRenderer, 664  
 abstrakcyjna, 472  
 AccountsFunctionTester, 974  
 ActionBar, 1079, 1080, 1084  
 Activity, 214, 262  
 AdapterView, 200, 201  
 AlarmManager, 748  
 AlarmManagerService, 514  
 AlertDialog, 1060  
 AlertDialogFragment, 1071  
 AllContactsLiveFolderCreatorActivity, 725  
 ALongRunningReceiver, 475  
 android.app.AlertDialog.Builder, 274  
 android.content.ContentProvider, 139  
 android.content.ContentResolver, 136  
 android.content.ContentValues, 136  
 android.graphics.Matrix, 537  
 android.location.Geocoder, 559  
 android.media.MediaPlayer, 601  
 android.os.Bundle, 161  
 android.os.Debug, 83  
 android.preference.Preference  
 atrbuty, 298  
 konfigurowanie projektu, 298  
 android.preference.PreferenceActivity, 297  
 android.util.Log, 81  
 android.view.LayoutInflater, 278  
 android.view.Menu, 247  
 android.view.SubMenu, 247

- klasa
- android.view.ViewGroup, 175
  - android.widget.ImageButton, 188
  - android.widgetListAdapter, 213
  - android.widget.RadioButton, 193
  - AndroidHttpClient, 349, 350
  - AnimatedSimpleTriangleRenderer, 677
  - Animation, 541
  - AnimationDrawable, 521
  - app\_name, 93
  - Application, 81
  - AppWidgetProvider, 742, 1107
  - ArrayAdapter, 202, 203, 208
  - AssetManager, 112
  - AsyncPlayer, 606, 617
  - AsyncTask, 351, 354, 357
  - AudioFormat, 629
  - AudioRecord, 626
  - AudioTrack, 618
  - BaseActionBarActivity, 1097, 1101
  - BaseAdapter, 218
  - BaseTester, 974
  - BDayWidgetModel, 752
  - BetterCursorWrapper, 732
  - BitmapFactory, 196
  - BookProviderMetaData, 139
  - BookTableMetaData, 140
  - Builder, 85
  - CamcorderProfile, 639
  - Camera, 42, 539
  - CameraProfile, 639
  - ClientCustPermMainActivity, 331
  - ComponentName, 162
  - ContactsContract.AggregationExceptions, 1001
  - ContentProvider, 141
  - ContentProviderOperation, 1003
  - ContentResolver, 137
  - ContentValues, 136, 137, 138
  - Context, 453
  - CustomHttpClient, 347
  - DatabaseHelper, 77
  - DebugActivity, 975, 1083
  - DeferWorkHandler, 436
  - DetailsFragment, 1037, 1050, 1052
  - DialogFragment, 1055, 1056, 1060
- Dot, 1143
- DownloadImageTask, 358
- DownloadManager, 362
- Drawable, 521
- ES20AbstractRenderer, 711
- Fragment, 1025, 1027
- FragmentManager, 1045
- FragmentTransaction, 1042, 1044, 1076
- GenericManagedAlertDialog, 291
- GenericPromptDialog, 292
- Geocoder, 560, 565
- GeomagneticField, 938
- GeoPoint, 559
- GestureDetector, 895
- GestureOverlayView, 904
- GLSurfaceView, 664, 667, 704
- GridViewActivity, 214
- GS20SimpleTriangleRenderer, 714
- HelpDialogFragment, 1068
  - kod Java, 1069
  - układ graficzny, 1069
- HttpActivity, 358
- HttpClient, 338
- HttpGet, 338
- HttpURLConnection, 349
- ImageView, 196
- Inflater, 221
- Intent, 40, 162
- IntentService, 469, 492
  - kod źródłowy, 470
  - rozszerzanie na odbiorcę komunikatów, 472
- ItemizedOverlay, 554, 557
- JetPlayer, 606, 617
- LayoutInflater, 278, 1031
- LightedGreenRoom, 476
- LinearLayout, 228
- ListActivity, 593
- ListFragment, 1047
- ListPreference, 296
- LocationManager, 567, 580, 583
- MainActivity, 867, 1035, 1061
- ManagedActivityDialog, 287, 288
- ManagedDialogsActivity, 287, 290
- ManateeAdapter, 221
- MapActivity, 548, 550

- MapController, 551  
MapView, 548, 888  
Matrix, 539  
MediaPlayer, 601, 610, 618  
MediaRecorder, 622  
MediaScanner, 647  
MediaStore, 640, 646  
MotionEvent, 861, 869, 873, 880  
MyContactsProvider, 726  
MyCursor, 731  
MyFragment, 1029  
MyLocationOverlay, 574  
    dostosowywanie, 577  
    zastosowanie, 574  
MySMSMonitor, 590  
NotePadProvider, 76  
NotesList, 73  
ObjectAnimator, 1075, 1139  
OnGestureListener, 895  
Overlay, 557  
PaintDrawable,, 108  
podstawowa  
    Activity, 427  
    BroadcastReceiver, 427  
    ContentProvider, 427  
    Service, 427  
PolygonRenderer, 691  
PrivActivity, 327  
PromptDialogFragment, 1064, 1067  
PromptListener, 279  
RadioGroup, 194  
RadioGroup.OnCheckedChangeListener,  
    193  
RegularPolygon, 681  
    animowanie obiektów, 691  
    calcArrays, 689  
    Constructor, 689  
    getAngleArrays, 689  
    getIndexBuffer, 689  
    getVertexBuffer, 689  
    getXMultiplierArray, 689  
    getYMultiplierArray, 689  
    renderowanie kwadratu, 689  
    rysowanie koła, 693  
RemoteViews, 741  
RemoteViewsFactory, 1114  
Renderer, 664  
ReportStatusHandler, 439  
ScaleGestureDetector, 896  
SearchActivity, 825, 828  
SearchRecentSuggestionsProvider, 813, 840  
SendAlarmOnceTester, 499  
SensorManager, 908, 930, 936  
Shakespeare, 1042  
SimpleCursorAdapter, 200, 201, 202, 208  
SimpleRectangleRenderer, 690  
SimpleSuggestionProvider, 799  
    kod źródłowy, 800  
    tryby bazodanowe, 801  
SimpleTriangleRenderer, 665  
SimpleTriangleRenderer2, 675, 676  
SipAudioCall, 599  
SipSession, 599  
SmsManager, 588  
SoundPool, 612, 617  
    maksymalna liczba próbek, 616  
    odtwarzanie dźwięku, 613  
    parametr SRC\_QUALITY, 616  
    strumień audio, 616  
Spannable, 224  
Spinner, 215  
SpinnerAdapter, 1095  
SQLiteQueryBuilder, 146, 149  
static final ints, 93  
StrictModeWrapper, 88  
System.out.println, 81  
TextToSpeech, 841  
TexturedSquareRenderer, 698  
ThreadGroup, 372  
TitlesFragment, 1047  
Toast, 293  
    debugowanie, 293  
UriMatcher, 146, 147  
Utils, 455  
VelocityTracker, 874  
ViewAnimation, 536  
ViewGroup, 1043  
widget.RadioGroup, 193  
WorkerThreadRunnable, 438  
XmlPullParser, 110  
klasy  
    aktywności sterującej, 975  
    android.view.View, 175

- klasy  
  bazowe aktywności  
    ActionBar, 1084  
    AndroidManifest.xml, 1093  
    showAsAction, 1092  
    SpinnerAdapter, 1095  
  służące do obsługi widoków zdalnych, 1108  
sterownika  
  DeferWorkHandler.java, 441  
  ReportStatusHandler.java, 441  
  Utils.java, 441  
  WorkerThreadRunnable.java, 441  
związane z menu, 248  
klatka kluczowa, 517  
klauzula select, 141  
klauzula WHERE, 134  
klient usługi IStockQuoteService, 382  
kliknięcie, 187  
klucz  
  API AJAX, 397  
  API MAP, 1018  
  interfejsu API mapy, 546  
  map-api, 321  
  prywatny, 318, 411  
  publiczny, 318, 411  
KML, Keyhole Markup Language, 572  
kod niestandardowy, 169  
kod odbiorcy, 454  
kod odbiorcy komunikatów, 465  
kod usługi zdalnego widoku, 1128  
kod źródłowy Androida, 48  
kod źródłowy Git, 48  
kody przycisków działania, 835  
kolumny kursora encji kontaktu, 997  
kolumny kursora propozycji, 823  
  definiowanie, 824  
Komatineni Satya, 23  
komentarze w kodzie., 140  
kompilacja, 501, 503, 506, 508, 511, 513  
kompilator JIT, 36  
kompilowanie jednostek cieniujących, 709  
kompilowanie kodu, 449  
kompilowanie zasobów, 98  
komunikat  
  ANR, 432, 450  
  testowy, 456  
  wysyłanie komunikatu, 454  
konfiguracja  
  alarmu, 493  
  kanalów, 629  
  strumieni audio, 855  
  uruchomieniowa, 63  
  urządzenia pionowa, *portrait*, 240  
  urządzenia pozioma, *landscape*, 240  
  urządzenia tryb kwadratowy, *square*, 240  
konfigurator widżetów, 739  
konfigurowanie  
  alertu odległościowego, 579  
  ciężaru, 230  
  klasy RemoteViewsFactory, 1114  
  konstruktora alertów, 278  
  menu za pomocą kodu, 255  
  obiektów nasłuchujących, 278  
  odbiorcy dla alarmu, 495  
  paska działania, 1096  
  powtarzalnego alarmu, 503  
  procedury obsługi kliknięcia, 188  
  usługi RemoteViewsService, 1112  
widoków obiektów, 1140  
wirtualnego urządzenia AVD, 64  
zasad ThreadPolicy, 85  
zasad VmPolicy funkcji, 86  
zdarzeń onClick, 1117  
źródła danych, 611  
konsola programisty, 1009  
konstruktor alertów, 278  
konstruktor klasy RemoteViewsFactory, 1114  
kontakty  
  agregacja, 1001  
  analiza, 964  
  dodawanie kontaktu, 998  
  aktywność sterująca, 999  
  edytowanie niestandardowych danych, 961  
  edytowanie szczegółów, 960  
  eksportowanie, 962  
  interfejs, 972  
  nieprzetworzone, 965, *Patrz także badanie nieprzetworzonych kontaktów*  
  odczytywanie kontaktów zbiorczych, 971  
  pobieranie bazy kontaktów, 965  
  standardowe typy danych, 964  
  synkronizacja, 1002  
  tabela danych, 967  
  testowanie danych  
  aktywność sterująca, 996

- testowanie nieprzetworzonych kontaktów aktywność sterująca, 992
- typy danych, 964
- umieszczanie zdjęć, 962
- widok contact\_entities\_view, 971
- widok view\_contacts, 971
- wyświetlanie, 958
- wyświetlanie szczegółów, 959
- zbiorcze, *Patrz także* badanie kontaktów zbiorczych
- kontekst wyszukiwania, 838
- kontener, container, 176
- konto dodawanie konta Google, 956
- lista kont, 958
- logowanie, 957
- odczytywanie zawartości, 958
- tworzenie konta Google, 956
- ustawienia kont i synchronizacji, 955
- wstawianie kontaktów, 957
- zarejestrowane na urządzeniu, 957
- konto programisty, Developer Account, 1006
- kontrakt klasy RemoteViewsFactory, 1114
- kontroler układu graficznego, 529
- kontrolka, control, 176
- AdapterView, 204
  - AnalogClock, 199
  - Button, 187
  - CheckBox, 190
  - Chronometer, 223
  - com.google.android.maps.MapView, 200
  - DatePicker, 197
  - DigitalClock, 199
  - EditText, 233
  - Gallery, 217
  - GridControl, 213
  - GridView, 213, 221
    - definiowanie kontrolki w pliku XML, 214
  - ImageButton, 188
  - ImageView, 195
  - ListView, 205
    - dodawanie elementów, 205
    - dodawanie innych kontrolek, 208
    - przyjmowanie danych, 207
    - reakcja na kliknięcie, 206
    - wyświetlanie wartości, 205
- listy, 205
- MapView, 200, 549
- ProgressBar, 223
- RadioButton, 192
- RatingBar, 223
- ScrollBar, 223
- Spinner, 215
- TableRow, 233
- TimePicker, 197
- ToggleButton, 190
- kontrolki
- Androida, 182
  - Androida 2.2, 467
  - AutoCompleteTextView, 185
  - daty i czasu, 197
    - AnalogClock, 199
    - DatePicker, 197
    - DigitalClock, 199
    - TimePicker, 197
  - ImageView, 196
  - przycisków, 187
    - Button, 187
    - CheckBox, 190
    - ImageButton, 188
    - RadioButton, 192
    - ToggleButton, 190
  - kontrolki tekstu, 183
    - AutoCompleteTextView, 185
    - EditText, 184
    - MultiAutoCompleteTextView, 186
    - TextView, 183
    - TextView, 181
  - widoku, 741
- koprocesor graficzny, 649
- kreator New Android Project, 598
- kryteria dopasowania, 166
- kSOAP2, 343
- kursor propozycji, 822
- kursor systemu Android, 133
- kwalifikatory konfiguracji, 113
- kwalifikatory zasobów, 241
- Gęstość pikseli na ekranie, 241
  - Język i region, 241
  - Klawiatura, 241
  - Orientacja ekranu, 241
  - Rodzaj tekstowych danych wejściowych, 241
  - Rozmiary ekranu, 241

kwalifikatory zasobów

Sterowanie przy braku klawiatury dotykowej, 241

Szersze/wyższe ekrany, 241

Typ ekranu dotykowego, 241

Wersja środowiska SDK, 241

kwerenda wyszukiwania, 833

## L

layout, *Patrz* układ graficzny

lista

aktywnych folderów, 720

animowanych klatek, 521

baz danych znajduje się w katalogu, 123

kodów przycisków działania, 835

kolumn, 823

kompletny, 273

pakietów, 408

preferencji, 297, 302

propozycji, 768

technologii, 946

układów graficznych, widżetów i widoków,

1106

widżetów, 1130

widżetów ekranu startowego, 738

ListActivity

odczytywanie danych, 212

listingi, 449, 489

lo, 56

localhost, 56

lokalizacja certyfikatu testowego, 547

lupa, 243

LVL, License Verification Library, 1017

## M

M3G, 652

macierz jednostkowa, 542

macierz transformacji, 539, 541

MacLean Dave, 23

magazyn gestów, 900

magazyn kluczny, 318, 319, 320

magazyn MediaStore, 645

magazyn multimedialny, 644

ManagedActivityDialog

klasa DialogRegistry, 289

mapa typu klucz – wartość, 159

mapy, 546

aplikacja, 548

nakładanie własnej warstwy, 553

obsługa za pomocą dotyku, 888

usługa Google Maps, 546

mapy projekcji, 149

MD5, 546

mechanizm przechowywania i dostępu, 120

Pliki, 120

Preferencje, 120

Sieć, 120

SQLite, 120

mechanizm refleksji, 87

mechanizm type-to-search, 797

menedżer

FrameLayout, 237

LinearLayout, 228

RelativeLayout, 235

TableLayout, 231

menedżer alarmów, 493, 449

aktywność do testowania ustawień, 500

alarm powtarzalny, 503

czas uruchomienia alarmu, 494

jednorazowe wysłanie alarmu, 498

konfiguracja, 493

konfigurowanie odbiorcy, 495

pierwszeństwo intencji, 512

praca z wieloma alarmami, 508

projekt, 497

testowanie scenariuszy, 501

trwałość, 515

twierdzenia, 515

układ graficzny, 502

ustawiania, 496

uzyskanie dostępu, 494

wersje alternatywne, 503

menedżer LinearLayout, 607

menedżer powiadomień, 463

menedżer telefonii, 594

menedżer układu graficznego, 227

FrameLayout, 228

LinearLayout, 228

RelativeLayout, 228

TableLayout, 228

- menu, 247  
aktywności, widoki i menu kontekstowe, 262  
aktywność, 254  
aktywności SearchInvokerActivity, 790  
alternatywne, 264  
dodatkowe znaczniki, 271  
dodawanie elementów, 255, 256  
dodawanie podmenu, 260  
drugorzędne, 256  
dynamiczne, 268  
grupy, 250  
konfiguracja, 255  
Konta i synchronizacja, 954  
kontekstowe, 261, 263  
modyfikowanie AndroidManifest.xml, 258  
odpowiedź na kliknięcie, 251, 257  
opcji, 249  
rejestrowanie widoku TextView, 263  
rozszerzone, 259  
standardowe, 255  
systemowe, 261  
środowisko testowe, 253  
tworzone za pomocą kodu Java, 268  
układ graficzny, 254  
w postaci ikon, 259  
wybór elementów, 251  
wykorzystanie intencji, 252  
zapelnianie menu, 266  
zdefiniowane w plikach XML, 268  
zmiana danych, 268  
metadane bazy danych, 139  
metadane dostawcy widżetów, 1128  
metadane wyszukiwania, 793, 807  
metoda  
    activity.onCreateContextMenu(), 262  
    addEarcon(), 855  
    addPreferencesFromResource(), 297  
    addSubMenu(), 261  
    animate(), 523  
    ArrayAdapter.createFromResource(), 217  
    callService(), 393  
    cancel(), 515  
    captureImage(), 644  
    commit(), 314  
    context.getSharedPreferences(), 757  
    createFromResource(), 203  
    DefaultHttpClient(), 347  
    detectAll(), 87  
    detectDiskReads(), 87  
    dismiss(), 1058, 1068  
    Display.getOrientation(), 926  
    Display.getRotation(), 926  
    distanceTo(), 569  
    doInBackground(), 353, 354, 356  
    doSpeak(), 845  
    doUpdate(), 937  
    draw(), 877  
    enableDefaults(), 87  
    enqueue(), 364  
    fabrykująca, 1029  
    findLocation(), 565  
    findPreference(), 312  
    fromRawResource(), 903  
    GestureLibraries.fromFile(), 903  
    GET, 340  
    getAccuracy(), 569  
    getAction(), 867, 886  
    getActionIndex(), 887  
    getActionMasked(), 887  
    getCheckedItemIds(), 213  
    getCheckedItemPositions(), 211  
    getCheckedRadioButtonId(), 195  
    getCount(), 150, 221  
    getDefaultEngine(), 856  
    getEdgeFlags(), 869  
    getExternalStoragePublicDirectory(), 607  
    getHttpClient(), 347, 348  
    getHttpContent(), 349  
    getInterpolation(), 532  
    getIntrinsicHeight(), 557  
    getIntrinsicWidth(), 557  
    getItemId(), 251  
    getLastNonConfigurationInstance(), 357  
    getMinBufferSize(), 629  
    getOrientation(), 931, 937  
    getPathSegments(), 146  
    getPointerCount(), 880  
    getPrefsToSave(), 754  
    getRotationMatrix(), 931  
    getSharedPreferences(), 303, 314  
    getString(), 303  
    getTag(), 1063

metoda  
    getText(), 208  
    getType(), 146, 819  
    getViewTypeCount(), 221  
    glDrawElements, 657  
    glVertexPointer, 654  
    handleMessage, 436  
    hasAccuracy(), 569  
    HTTP GET, 405  
    HTTP POST, 405  
    initCamera(), 634  
    initRecorder(), 637  
    insert(), 78, 204  
    invalidate(), 1144  
    isCancelled(), 356  
    isChecked(), 192  
    isEnabled(), 222  
    isLocationDisplayed(), 552, 575  
    isRouteDisplayed(), 552  
    LayoutInflater(), 278  
    LayoutInflater.from(), 278  
    ListView, 74  
    Log.d, 111  
    makeText(), 294  
    MenuBuilder.addIntentOptions, 267  
    mIndexBuffer, 658  
    MotionEvent.getAction(), 887  
    moveToFirst(), 133  
    newInstance(), 350  
    notifyDataSetChanged(), 204  
    nstartDrag(), 1143  
    obtain(), 873  
    obtainMessage(), 435  
    onAccuracyChanged(), 913, 914  
    onActivityCreated(), 1032  
    onActivityResult(), 568  
    onCheckedChanged(), 195  
    onClick(), 192, 873  
    onCreate(), 73, 79, 88, 206, 805, 903  
    onCreateContextMenu(), 264  
    onCreateDialog(), 284  
    onCreateOptionsMenu, 249, 265  
    onCreateView(), 1031, 1056, 1068  
    onDestroy(), 80, 446  
    onEnabled(), 753  
    onInflate(), 1030  
    onInit(), 856  
    onItemClick(), 208  
    onListItemClick(), 75  
    onMeasure(), 1144  
    onNewIntent(), 803, 805, 811  
        testowanie, 806  
    onOptionsItemSelected, 251, 252, 270  
    onPause, 446  
    onPostExecute(), 353  
    onPreExecute(), 353, 361  
    onPrepareDialog(), 284  
    onPrepareOptionsMenu, 268  
    onProgressUpdate(), 353  
    onProviderDisabled(), 573  
    onReceive(), 591, 753, 1120  
    onRestart(), 80  
    onResume(), 79, 80  
    onRetainNonConfigurationInstance(), 357  
    onSaveInstanceState(), 1047  
    onSearchRequested(), 789, 838  
    onSensorChanged(), 913, 919, 929  
    onStart(), 79  
    onStartCommand, 485  
    onStop(), 80, 446  
    onTouchEvent(), 862, 863, 877, 893  
    onUpdate(), 743, 744, 1111  
    penaltyDeath(), 85  
    PendingIntent.getActivity(), 172, 173  
    permitDiskReads(), 87  
    playSilence(), 856, 858  
    populate(), 557  
    postInvalidate(), 576  
    postTranslate, 538  
    PreferenceManager.getDefaultShared  
        → Preferences(this), 303  
    preTranslate, 538, 542  
    publishProgress(), 353  
    putFragment(), 1047  
    queueSound(), 616  
    recognize(), 905  
    recycle(), 873  
    registerListener(), 913  
    requestLocationUpdates(), 571, 573  
    respondToMenuItem(), 433  
    rotate, 539  
    scanFile(), 646

- scheduleDistinctIntents(), 511  
scheduleSameIntentMultipleTimes(), 510  
sendAlarmOnce(), 499  
sendBroadcast(), 173, 453, 454  
sendDataMessage(), 588  
sendMessage(), 435  
sendMessageDelayed(), 435  
sendMultipartTextMessage(), 589  
sendSmsMessage(), 587  
setAdapter(), 214  
setBounds(), 557  
setChecked(), 193  
setContentView(), 211, 214  
setData(), 435  
setDataSource(), 611, 612, 621  
setEdgeFlags(), 869  
setEngineByPackageName(), 856  
setEntries(), 313  
setGroupVisible, 250  
setImageResource(), 189  
setLatestEventInfo(), 466  
setListAdapter(), 214  
setMeasureAllChildren(), 239  
setOnCheckedChangeListener(), 193  
setOneShot(), 522  
setOnTouchListener(), 888  
setOnUtteranceCompletedListener(), 846  
setOptionText(), 303  
setPendingIntentTemplate(), 1119  
setRepeating(), 505  
setRetainInstance(), 1033  
setTargetFragment(), 1074  
showAllRawContacts(), 993  
sleep(), 432, 439  
sort(), 204  
speak(), 846  
startActivity(), 169, 1074  
startActivity(intent), 173  
startSearch(), 789, 839  
    appSearchData, 789  
    globalSearch, 789  
    initialQuery, 789  
    selectInitialQuery, 789  
startService(), 173, 372, 429, 469  
stop(), 441  
stopSelf, 485  
surfaceCreated(), 635  
testAccounts(), 975  
toUri(), 1118  
translate, 539  
tv.getText(), 225  
uiCallback.sendEmptyMessage(0), 565  
Utils.logThreadSignature(), 436, 439  
VelocityTracker.obtain(), 874  
zoomToSpan(), 558  
zwrotna getCount(), 1115  
zwrotna getItemId(), 1116  
zwrotna getLoadingView(), 1116  
zwrotna getViewAt(), 1115  
zwrotna getViewTypeCount(), 1116  
zwrotna hasStableIds(), 1117  
zwrotna onAttach(), 1030  
zwrotna onCreate(), 1030, 1115  
zwrotna onCreateView(), 1031  
zwrotna onDataSetChanged(), 1117  
zwrotna onDestory(), 1033, 1115  
zwrotna onDestoryView(), 1033  
zwrotna onDetach(), 1033  
zwrotna onDrag(), 1144  
zwrotna onInflate(), 1030  
zwrotna onPause(), 1032  
zwrotna onResume(), 1032  
zwrotna onStop(), 1032  
metody cyklu życia aktywności, 79  
metody główne  
    delete, 139  
    getType, 139  
    insert, 139  
    query, 139  
    update, 139  
metody pobierające, 1046  
metody uzyskiwania aktualizacji położenia, 574  
MFC, Microsoft Foundation Classes, 39  
mikrostopnie, 562  
mikrotesla,  $\mu$ T, 930  
milimetr, 235  
MIME, Multipurpose Internet Mail  
    Extensions, 127  
moduł HttpClient, 338, 343, 405  
monitorowanie zdarzeń animacji, 541  
motyw, 227  
MultiAutoCompleteTextView, 186

**N**

nagrywanie i odtwarzanie dźwięku, 622  
nakładanie tekstyury, 683  
nakładka ItemizedOverlay, 558  
narzędzia  
    AAPR, 69  
    ADT, 38, 56  
    DDMS, 1019  
    Developer Tools, 57  
    do usuwania błędów, 81  
    wątkowania, 430  
narzędzie  
    Abstract Window Toolkit, 32  
adb, 323  
AVD Manager, 83  
edytora manifestu, 327  
Export Unsigned Application Package, 321  
Hierarchy Viewer, 175, 242  
    ekran urządzeń, 243  
    tryb Pixel Perfect, 244  
    układ graficzny, 242  
jarsigner, 318  
keytool, 321, 547  
LogCat, 81, 82, 85, 867  
    wpisy, 870  
Swing, 32  
widżet Toast, 571  
zipalign, 322  
nazwa składnika, component name, 266  
NDK, Native Development Kit, 940  
NFC, Near Field Communication, 35, 939  
    aktywacja, 940  
    emulacja karty, 949  
    odbieranie terminali, 942  
    odczytywanie terminali, 946  
P2P, 949  
testowanie technologii, 950  
trasowanie terminali, 941  
tryby działania, 940  
wybór filtra intencji, 943  
numer portu 5554, 588

**O**

obiekt  
    addressContainer, 178  
    ApplicationInfo, 86

    AudioRecord, 629  
    AudioRecord., 628  
    Builder, 87  
    ClipData, 1143  
    Criteria, 568  
    criteriaIntent, 266  
    cursor, 130, 772  
    DatePicker, 273  
    DragEvent, 1132  
        ACTION\_DRAG\_ENDED, 1132  
        ACTION\_DRAG\_ENTERED, 1132  
        ACTION\_DRAG\_EXITED, 1132  
        ACTION\_DRAG\_LOCATION, 1132  
        ACTION\_DRAG\_STARTED, 1132  
        ACTION\_DROP, 1132  
    Drawable, 109  
    falseLayoutBottom, 873  
    FileDescriptor, 611  
    flight\_sort\_options\_values, 301  
    Geocoder, 891  
    GridView, 214  
    HttpClient, 349  
    HttpGet, 349  
    HttpParams, 349  
    HttpPost, 349  
    includeInGlobalSearch, 812  
    IntentService, 485  
    klasy Spinner, 216  
    Location, 568  
    ManateeAdapter, 221  
    map, 141  
    MediaController, 621  
    Menu, 261  
    Message, 435  
    MotionEvent, 862, 873, 877, 880, 891  
    MotionView, 862  
    nasłuchujący, 251, 897, 1087  
    nasłuchujący OnInitListener, 845  
    NdefMessage, 949  
    NdefRecord, 948  
    nio, 666  
    OnCheckedChangeListener, 192  
    Parcelable, 391  
    PendingIntent, 515  
    PreferenceCategory, 311  
    RadioButton, 195  
    RemoteViews, 466  
    SensorEvent, 913

- SensorListener, 915  
SharedPreferences, 314  
SipProfile, 599  
Spinner, 204, 216, 1095  
SubMenu, 260  
TextToSpeech, 845  
TimePicker, 273  
Toast, 294  
trueBtnTop, 867  
VelocityTracker, 875  
ViewHolder, 221  
wakelock, 478  
XmlPullParser, 110  
XmlResourceParser, 110  
obiekty nasłuchujące, 278  
obsługa map, 888  
obsługa wyjątków, 343  
obszar dropTarget, 1146  
ochrona zasobów i funkcji urządzenia, 325  
odbieranie terminali NFC, 942  
odbiorca BroadcastReceiver, 580, 858, 919  
odbiorca komunikatów, broadcast receiver, 453, 462, 467, 579  
alert odległościowy, 579  
definicja odbiorcy, 460  
długoterminowy, 467, 474  
opóźnienia czasowe, 461  
powiadomienia, 463  
powielanie, 460  
odbiorca pozaprocesowy, 462  
odbiorca przebywający we własnym procesie, 462  
odbiorca TestReceiver, 495  
odbiorca treści, 461  
odczytywanie danych w ListActivity, 210  
odległość pomiędzy dwoma obiektami, 569  
odpowiedź na wybór elementów, 251  
odpowiedź na zdarzenia onClick, 1120  
odpowiedź na zdarzenie onDrag w strefie upuszczania, 1137  
odstępy, 235  
odtwarzanie ciszy, 856  
odtwarzanie ikony akustycznej, 856  
odtwarzanie multimediów, 606  
AsyncPlayer, 617  
AudioTrack, 618  
interfejs API multimediów, 619  
JetPlayer, 617  
kod aplikacji, 607  
MediaPlayer, 618  
setDataSource, 611  
SoundPool, 612  
układ graficzny, 607  
ograniczenia klasy AnimationDrawable, 522  
OHA, 48  
okna alertów, 274  
projektowanie, 274  
okna dialogowe  
alertów, 273  
asynchroniczne, 273  
informujące, 273  
listy kompletacyjne, 273  
modalne, 281  
obiekt DatePicker, 273  
obiekt TimePicker, 273  
okna niezarządzane, 283  
okna zarządzane  
DialogRegistry, 289  
GenericManagedAlertDialog, 291  
GenericPromptDialog, 292  
IDialogProtocol, 288  
ManagedActivityDialog, 288  
ManagedDialogsActivity, 289, 290  
protokół, 283  
struktura, 286  
upraszczanie protokołu, 285  
pojedynczego wyboru, 273  
synchroniczne, 273  
wielokrotnego wyboru, 273  
zachęty, 276  
kod, 280  
obiekt nasłuchujący, 279  
plik XML układu graficznego, 277  
projektowanie, 276  
przeprojektowanie okna, 282  
tworzenie i wyświetlenie, 279  
okno  
Devices, 242  
Emulator Control, 576  
File Explorer, 604  
Hierarchy Viewer, 243  
kreatora New Android Project, 61

- okno  
Launch Options, 84  
LogCat, 82, 582, 629, 885, 988  
Package explorer, 1018  
terminalu, 52
- opcja  
Add External JARs, 404  
Add note, 73  
Android SDK and AVD Manager, 323  
Android Tools, 321  
Build Path/Configure Build Path., 404  
Create project from existing sample, 899  
Debug As/Android Application, 82  
Debugowanie USB, 82  
Export Signed Application Package, 323  
Launch from snapshot, 84  
QUEUE\_ADD, 845  
QUEUE\_FLUSH, 845  
Upload Application, 1018  
Wipe user data, 84
- Open Graphics Library, 650  
OpenCORE PacketVideo, 37  
OpenGL ES, 39  
OpenGL ES 2.0, Patrz biblioteka OpenGL ES 2.0  
OpenJDK, 52
- operacja  
setRotate, 542  
setScale, 542  
setSkew, 542  
setTranslate, 542
- operacje na macierzach, 541
- oprogramowanie integracyjne, middleware, 337
- optymalizacja aplikacji, 322
- Oracle/Sun JDK, 53
- organizowanie preferencji w kategorie, 310
- orientacja w przestrzeni, 936
- oś głębi, 660
- oświetlony zielony pokój, 478  
implementacja, 478
- P**
- P2P, peer-to-peer, 949
- pakiet  
.apk, 610  
android.app, 1027  
android.location, 559
- android.media, 601  
android.nfc, 948  
android.opengl.GLES20., 704  
android.provider, 120  
android.providers.Contacts, 132  
android.view.animation, 525  
com.svox.pico, 856  
java.nio, 666  
map, 545  
nio, 652  
OpenGL ES, 38  
R.java, 92
- pakiety, 407  
dokumentacja SDK, 412  
lista, 408  
nazwa, 408  
podpisywanie, 409  
specyfikacja, 407  
usuwanie, 409
- pakiety java.\*., 47
- para kluczy, 318
- parametr  
childLayout, 202  
from, 202  
Intent, 169  
odświeżania czujnika, 913  
requestCode, 169  
SRC\_QUALITY, 616  
to, 202  
uri, 137
- parser jSON, 342  
parser SOAP, 342  
parser XML, 342  
pary typu MIME, 129
- pasek działania, 1079, 1080  
interakcja z menu, 1091  
obszar menu, 1081  
obszar paska narzędzi, 1081  
obszar przycisku ekranu startowego, 1080  
obszar tytułu, 1080  
obszar zakładek, 1081  
tryb wyświetlania listy, 1094, 1096  
tryby nawigacji, 1089
- pasek zakładek  
badanie aktywności, 1093  
pasująca aktywność, 266

- PDU, Protocol Description Unit, 591  
perspektywa, 82  
perspektywa DDMS, 82, 603  
perspektywa Debug, 82  
pętla komunikatów, 282  
Phillips Dylan, 25  
Pico, 43  
pierwsza aplikacja, 60  
piksel, 235  
piksele niezależne od gęstości, 235  
piksele niezależne od skali, 235  
PKI, Public Key Infrastructure, 411  
planowanie bazy danych, 139  
plik  
.adt, 546  
.apk, 318, 407, 601  
.dex, 36  
.jar, 36  
.profile, 53  
\_has\_set\_default\_values.xml, 304  
AbstractRenderer.java, 672  
AccountsFunctionTester.java, 974  
AIDL usługi tłumacza, 399  
aktywności sterującej, 442  
AlarmIntentPrimacyTester.java, 513

- plik
- RawContact.java, 990
  - release.keystore, 319
  - scale.xml, 524
  - sdcard.img, 602
  - SDK Manager, 54
  - searchable.xml, 793
  - SimpleSuggestionProvider.java, 800
  - SimpleTriangleRenderer.java, 672
  - StandaloneReceiver.java, 462
  - strings.xml, 92, 181, 215, 302, 829, 1145
  - TestAppActivity.java, 420
  - TestBCRActivity.java, 456
  - TestContactsDriverActivity.java, 977
  - TestHandlersDriverActivity.java, 442
  - TestLibActivity.java, 417
  - TestListViewProvider.java, 1122
  - TestOpenGLMainDriverActivity.java, 672
  - TestReceiver.java, 456
  - TestRemoteViewsFactory.java, 1126
  - TestRemoteViewsService.java, 1128
  - TranslateService.java, 402
  - układu graficznego, 94, 444, 458
  - układu graficznego main.xml, 94
  - Utils.java, 455, 456, 462, 980
  - web.xml, 68
  - wewnętrzny, 137
  - XML, 98, 109, 268
  - XML animacji rotacyjnej, 531
  - XML preferencji, 302
  - XML zawierający definicje menu, 269
  - zasobów menu, 458
  - zawierający informacje o widżecie, 1129
  - zdalnego układu graficznego, 1109
  - ZIP, 449, 489
- pliki
- aplikacji służącej do tłumaczeń, 397
  - do testowania usług
  - implementacji dostawcy propozycji, 799
  - nieskompresowane, 98
  - parcelowane, 388
  - programu wyświetlającego listę kont, 972
  - projektu menedżera alarmów, 497
  - projektu TestBCR, 489
  - projektu testowego, 456
  - projektu z odbiorcą komunikatów, 490
- układów graficznych, 114
  - wideo, 619
  - widżetu urodzinowego, 746
- podmenu, 260
- podpis cyfrowy, 410
- podpisywanie aplikacji, 318
- podpisywanie plików, 411
- podpisywanie pliku .apk, 321
- podpisywanie pliku .jar, 318
- podręcznik referencyjny środowiska OpenGL ES, 653
- pojemnik, 176
- pojemnik ListView, 201, 221
- pole NFC, 939
- pole QSB, 771, 793
- polecenia powłoki, 124
- polecenie
  - #ls /system/bin, 123
  - adb, 83
  - adb devices, 121, 122
  - adb help, 122
  - android list avd, 122
  - find, 123
  - ipconfig, 56
  - ls, 123
  - ls -l, 123
  - ls /data/data, 123
  - sqlite> .tables, 125
  - sqlite>.exit, 125
- połączenia równorzędne (P2P) NFC, 949
- położenie, 568, 932
- położenie bieżące, 576
- położenie geograficzne, 559
  - aktualizacja danych, 569
  - LocationManager, 571
  - MyLocationOverlay, 574
- omięcie grawitacji, 927
- pomoc techniczna, 1012
- POP, Post Office Protocol, 954
- port#, 83
- powiadomienia, 464
  - kod odbiorcy komunikatów, 465
- powłoka ash, 123
- powłoka dostawcy widżetu, 743
- predefiniowanie identyfikatora, 97

- preferencje, 295  
 ekran preferencji, 297  
 kategorie, 311  
 klasa ListPreference, 296  
 lista preferencji, 297  
 magazyn danych, 306  
 programowe zarządzanie, 312  
 przechowywanie stanu aktywności, 313  
 widok CheckBoxPreference, 305  
 widok EditTextPreference, 307  
 widok RingtonePreference, 308  
 zagnieżdżenie elementów  
   PreferenceScreen, 310  
 zapisywanie stanu, 313  
 preferencje aplikacji, 295  
 preferencje pola wyboru, 305  
 preferencje RingtonePreference, 309  
 prefiks vnd, 129  
 procedura DeferWorkHandler, 433  
 procedura obsługi, handler, 431, 432  
   klasy sterownika, 441  
   menu, 445  
 proces w Androidzie  
   Activity, 427  
   BroadcastReceiver, 427  
   ContentProvider, 427  
   Service, 427  
 procesy, 407  
 program SQLite Explorer, 965  
 program testujący menedżer alarmów, 502  
 projekt bibliotek, 414, 420, 425  
   identyfikatory współdzielonych zasobów, 424  
   kod aktywności, 420  
   manifest, 419, 422  
   menu, 418, 422  
   twierdzenia, 414, 417  
   układ graficzny, 418, 421  
 projekt Provider, 48  
 protokół  
   odbiorcy komunikatów, 468  
   SIP,inicjalizacji sesji, 597  
   SOAP, 127  
   SSL, 37  
   zarządzanych okien dialogowych, 283, 287  
 przeciąganie obiektów, 876  
   kod Java, 876  
   układ graficzny, 876  
 przekroczenie limitu czasu, 344, 348  
 przekroczenie limitu czasu gniazda, 343  
 przekroczenie limitu czasu połączenia, 343  
 przesłonięcie kontrolki ListView, 209  
 przetwarzanie tekstu na mowę, 841  
   pętla przekazywania wyrażeń, 847  
   pliki dźwiękowe, 848  
   prędkość mowy, 842  
   silnik Pico, 842  
   śledzenie wyrażeń, 846  
   układ graficzny, 848  
   ustawienia silnika, 842  
   usunięcie z kolejki tekstu, 845  
   wyrażenie, 846  
   zapisywanie pliku dźwiękowego, 850  
 przycisk  
   Generate API Key, 547  
   Inspect Screenshot, 243  
   Install Selected, 55  
   Load View Hierarchy, 242  
   przełączania, 190  
   Publish, 1021  
   Screen Capture, 1019  
   wyszukiwania, 769, 777, 778  
 przyciski działania  
   definicja, 836  
   keycode, 836  
   kolumny, 837  
   queryActionMsg, 837  
   suggestActionMsg, 837  
   suggestActionMsgColumn, 837  
 punkt, 235

**Q**

- QEMU, 39  
 QSB, Quick Search Box, 769

**R**

- raporty o błędach aplikacji, 1011  
 refleksja, 87  
 reguły przydzielania intencji, 166  
 rejestracja upoważnienia, 126  
 rejestrator dźwięku, 642  
 rejestrator wideo, 632  
   aktywność, 632  
 AndroidManifest.xml, 639

rejestrator wideo  
 kod obsługujący wstrzymywanie, 633  
 kod przetwarzania, 636  
 metody zwrotne, 638  
 rejestrowanie aktualizacji lokacji, 569  
 rejestrowanie aktywności, 156  
 rejestrowanie dostawcy, 150  
 rejestrowanie multimedialnych, 621  
 analiza procesu rejestracji, 630  
 AudioRecord, 626  
 CAMCORDER, 625  
 MediaRecorder, 622  
 nieskompresowane dane audio, 630  
 rejestrowanie rozmowy, 625  
 VOICE\_RECOGNITION, 625  
 za pomocą intencji, 641  
 rejestrowanie odbiorcy komunikatów, 456  
 rejestrowanie widoku dla menu kontekstowego, 263  
 rejestry dziennika LogCat, 872  
 rekord, 136  
 renderowanie, 707  
 renderowanie kwadratu, 689  
 REST, REpresentational State Transfer, 119  
 RESTful, 41  
 RFID, Radio Frequency Identification, 939  
 RISC, Reduced Instruction Set Computer, 39  
 rodzaje adapterów, 204  
 ArrayAdapter<T>, 204  
 CursorAdapter, 204  
 ResourceCursorAdapter, 204  
 SimpleAdapter, 204  
 SimpleCursorAdapter, 204  
 rodzaje menu, 259  
 rodzaje zasobów, 99  
 ciągi znaków, 99  
 kolorowe obiekty rysowane, 100  
 kolory, 99  
 obrazy, 100  
 tablice ciągów znaków, 99  
 wielokrotności, 99  
 własne pliki XML, 100  
 własne, nieskompresowane pliki dodatkowe, 100  
 własne, nieskompresowane zasoby, 100  
 wymiary, 99

rozszczepianie klasy ContentProvider, 141  
 RPC, Remote Procedure Call, 368  
 RTP, Real-time Transport Protocol, 598  
 RTSP, Real-time Streaming Protocol, 598  
 rysowanie wielokąta, 693  
 rysowanie wielu figur geometrycznych, 699  
 rzutowanie obrazu trójwymiarowego, 659  
 glFrustum, 659  
 gluLookAt, 659  
 glViewport, 659

## S

schemat danych, 167  
 SD, Secure Digital, 601  
 SDK, Software Development Kit, 32  
 SDP, Session Description Protocol, 598  
 segment ścieżki, 135  
 sekwencja przeciągania, 1140  
 serwis Google Maps, 553  
 sieć  
 3G, 38  
 Bluetooth, 38  
 EDGE, 38  
 WiFi, 38, 364  
 silnik Pico, 842  
 silnik przetwarzania tekstu na mowę, 43  
 silnik renderujący prostokąt, 679  
 silnik SquareRender, 690  
 silnik TTS, 845, 850, 859  
 dostępność języka, 857  
 funkcje zaawansowane, 854  
 metody językowe, 857  
 odtwarzanie ciszy, 856  
 odtwarzanie ikony akustycznej, 856  
 SIP, Session Initiation Protocol, 585  
 skala mapy, 552  
 skalowanie, 528  
 sklep, Patrz Android Market  
 składnia odniesienia do zasobu, 95  
 skrót MD5 certyfikatu testowego, 547  
 skróty dla elementu menu, 272  
 skrzynka odbiorcza, 592  
 SMS, Short Messaging Service, 585  
 foldery, 593  
 monitorowanie wiadomości, 589  
 skrzynka odbiorcza, 592

- wiadomości przychodzące, 589  
 wysyłanie wiadomości, 586  
**SOAP**, Simple Object Access Protocol, 119  
**sp**, 235  
 specyfikacja JSR 239, 652  
 specyfikacja pakietu, 407  
 spis dostępnych kontaktów, 979  
 spis kwalifikatorów konfiguracji, 113  
 sprzedaż aplikacji, 1012  
     lokalizacja aplikacji, 1014  
     obsługa różnych rozmiarów ekranu, 1012  
     ponowne kierowanie do sklepu, 1016  
     przygotowanie ikony aplikacji, 1015  
     przygotowanie pliku .apk do wysłania, 1018  
     przygotowanie pliku  
         AndroidManifest.xml, 1013  
     testowanie działania, 1012  
     usługa licencyjna, 1017  
     ustalanie ceny, 1016  
**SSL**, Secure Sockets Layer, 37  
**stała**  
     CATEGORY\_SYSTEM., 261  
     FILL\_PARENT, 182  
     intent.ACTION\_SEARCH, 805  
     MATCH\_PARENT, 182  
     Menu.CATEGORY\_ALTERNATIVE, 266  
     Menu.CATEGORY\_SECONDARY, 248  
     Menu.CATEGORY\_SYSTEM, 248  
     Notes.CONTENT\_URI, 73  
**stałe trybu agregacji**, 970  
 stan uaktywnienia przycisku, 189  
 stan wcisnięty przycisku, 189  
 stan zatrzymania, 476, 477  
 stany aktywności, 80  
 stany wątku, 441  
 stos Java, 666  
 stos drugoplanowy, 1059  
 stos programowy, 337  
 stos programowy Android SDK, 33, 37  
**StrictMode**, 84  
 strona startowa Androida, 719  
 strona startowa z polem QSB, 769  
 struktura klas gestów, 900  
 struktura preferencji, 296  
 struktura składników, 428  
 strumienie audio, 855  
 styl ErrorText, 226  
 styl ErrorText.Danger, 226  
**style**, 224  
     dla fragmentów tekstu, 225  
     nadrzędne, 226  
     umieszczone dynamiczne, 225  
     umieszczone w widoku, 226  
     wykorzystywane w wielu widokach, 225  
**Sun JDK**, 52  
**superklasa aktywności**, 172  
**symbol #**, 123  
**synonim**, 141  
**system GPS**, 82  
**system operacyjny**  
     iPhone OS, 33  
     Linux, 52  
     Mac OS X, 52  
     Mobile Linux, 33  
     Moblin, 33  
     Symbian OS, 33  
     Windows 7, 52  
     Windows Mobile, 33  
     Windows Vista, 52  
     Windows XP, 52  
**szablon intencji oczekującej**, 1119

## Ś

- ścieżka danych**, 168  
**środowisko**  
     Android SDK, 52  
     chronionej pamięci, 78  
     Dalvik VM, 36  
     Eclipse, 51  
     Eclipse IDE for Java Developers, 53  
     IDE, 48  
     IDE Eclipse, 51  
     J2EE, 58, 338  
     Java ME, 652  
     JRE, 52  
     JVM, 32  
     OpenGL ES 2.0, 39  
     programowania, 51  
     projektowe, 89  
     testowe biblioteki OpenGL, 667  
     testowe do sprawdzania menu, 254

**T**

tabela contact, 1001  
tabela wyszukiwania, 968  
tabela zawierająca wyjątki agregacji, 1001  
tablica ciągów znaków, 101  
tablica dodanych elementów menu, 267  
tablica flight\_sort\_options, 301  
tagi do symulowania podmenu, 271  
technologia ARM, 39  
technologia M3G, 652  
technologia Ndef, 946  
technologia NFC, 939  
teksturowane koła, 703  
teksturowany kwadrat, 699  
teksturowany wielobok, 700  
tekstury, 694  
    glActiveTexture, 697  
    glBindTexture, 697  
    glGenTextures, 697  
    glTexCoordPointer, 697  
    glTexEnv, 697  
    glTexParameter, 697  
    GLUtils.texImage2D, 697  
proces obsługi, 695  
rysowanie, 698  
znormalizowane współrzędne, 694  
terminal  
    ACTION\_NDEF\_DISCOVERED, 942  
    ACTION\_TAG\_DISCOVERED, 942  
    ACTION\_TECH\_DISCOVERED, 942  
testowanie  
    animacji typu alfa, 530  
    danych kontaktu, 995  
    długoterminowych usług, 488  
    dostawcy BookProvider, 150  
    kontaktów zbiorczych, 982  
    nieprzetworzonych kontaktów, 990  
    odbiorców komunikatów, 489  
    pierwszeństwa intencji, 512  
    procedur obsługi, 442  
    technologii NFC, 950  
    ustawień alarmów, 500  
    widżetu wyświetlającego listę, 1130  
TextView, 183  
ThreadPolicy, 85  
transformacja widoku, 542  
trasowanie terminali NFC, 941

**tryb**

agregacji, 970  
debugowania, 82  
debugowania USB, 82  
MODE\_PRIVATE, 314  
MODE\_WORLD\_READABLE, 314  
MODE\_WORLD\_WRITEABLE, 314  
portretowy, 1052  
rozwijalnego menu, 1104  
ruchu ulicznego, 552  
usuwanie błędów, 82  
widoku ulic, 552  
wyszukiwania, 771  
TTS, Text To Speech, 841  
tworzenie  
    aktywnego folderu, 722  
    aplikacji Notepad, 70  
    cyfrowego podpisu, 411  
    dostawcy widżetów, 1122  
    fragmentu wyświetlającego okna  
        dialogowe, 1055  
    instancji widżetu, 742  
    intencji oczekującej, 495  
    intencji oczekującej na komunikat, 1118  
    intencji oczekujących, 511  
    interfejsu użytkownika w pliku XML, 179  
    interfejsu użytkownika za pomocą kodu, 177  
    kategorii preferencji, 311  
    klasy fabrykującej, 1126  
    klasy SoundPool, 616  
    komunikatu, 435  
    konfiguracji uruchomieniowej, 63  
    konta Google, 956  
    listy pakietów, 408  
    menu, 249  
    menu za pomocą plików XML, 268  
    niestandardowych adapterów, 218  
    niestandardowych animacji, 1075  
    obiektu nasłuchującego listy nawigacji, 1095  
    odbiorcy komunikatów, 455  
    odniesień do kontrolek, 182  
    odpowiedzi dla elementów menu, 270  
    odpowiedzi dla menu kontekstowego, 264  
    okna alertu, 275  
    okna dialogowego zachęty, 277  
    powiadomień, 465  
    pół wyboru, 190

projektu, 449  
 projektu bibliotek, 417  
 prostego odbiorcy, 454  
 tożsamości w sklepie, 1006  
 trójkątów, 676  
 urządzenia AVD, 67, 602  
 wystąpienia fragmentu, 1029  
 typy agregacji, 1001  
 typy danych, 167  
 typy MIME, 77, 128, 130, 265  
 typy treści, 129  
     application, 129  
     audio, 129  
     example, 129  
     image, 129  
     message, 129  
     model, 129  
     multipart, 129  
     text, 129  
     video, 129

**U**

Ubuntu, 52  
 układ graficzny, layout, 94, 95, 113, 176, 227  
     animacja, 523  
     dostosowanie do urządzenia, 239  
     optymalizacja, 242  
     tworzenie interfejsu użytkownika, 240  
     usuwanie błędów, 242  
 układ graficzny  
     aktywności LocalSearchEnabledActivity, 794  
     aktywności SearchActivity, 792  
     aktywności SearchInvokerActivity, 789  
     aktywności TestAlarmsDriverActivity, 502  
     aktywności TestOpenGLMainDriver, 671  
     aktywności wyszukiwania, 828  
     aplikacji MapViewDemo, 549  
     aplikacji odtwarzającej multimedia, 607  
     aplikacji rejestrującej, 631  
     aplikacji tłumaczącej, 398  
     dla aplikacji TouchDemo1, 863  
     do animacji poklatkowej, 519  
     do wywoływania klasy AsyncTask, 354  
 FrameLayout  
     widok ImageView, 239  
 klasy SearchActivity, 828

LinearLayout, 178, 228  
 ciężar, 228  
 grawitacja, 228  
 RelativeLayout  
     interfejs użytkownika, 237  
 TableLayout  
     kontrolka EditText, 234  
     nieregularna tabela, 233  
 trueLayoutTop, 871  
 usługi IStockQuoteService, 384  
 usługi StockQuoteService2, 389  
 widżetu, 749  
 zawierający widok debugowania, 1090  
 umowa EULA, 1019  
 upoważnienie, 127  
 uprawnienia, 325  
     atrybuty, 329  
     definiowanie uprawnień, 334  
     dla funkcji i zasobów, 326  
     identyfikatorów URI, 332  
     do danych, 168  
     niestandardowe, 326, 330  
     przekazywanie uprawnień, 333  
     w pliku AndroidManifest.xml, 325  
 uprawnienie  
     android.permission.ACCESS\_COARSE\_  
         →LOCATION, 567  
     android.permission.ACCESS\_FINE\_  
         →LOCATION, 567, 580  
     android.permission.INTERNET, 600, 610  
     android.permission.READ\_CONTACTS,  
         980  
     android.permission.READ\_PHONE\_  
         →STATE, 596  
     android.permission.RECORD\_AUDIO, 626  
     android.permission.USE\_SIP, 600  
     android.permission.WRITE\_EXTERNAL\_  
         →STORAGE, 853  
 uruchamianie aktywności, 162  
 uruchamianie emulatora, 83  
 Urząd Przydzielania Numerów  
     Internetowych, 129  
 urządzenia AVD, 60, 63, 65, 122  
 urządzenia typu handheld, 282  
 urządzenie Android Developer Phone, 1005

usługa, 59  
długoterminowa, 486, 488  
Google Maps, 553  
Google Maps JavaScript API, 569  
HTTP, 43, 337  
IStockQuoteService, 377, 380, 381, 382, 384  
LocationManager, 566, 567, 569, 571  
lokalna, 367, 369  
nietrwała, 484  
notowań giełdowych, 377  
RemoteViewsService, 1112, 1113  
RESTful, 48  
ServiceManager, 566  
StockQuoteService2, 390  
Test60SecBCRService, 474  
trwała, 485  
WakefullIntentService, 472  
zdalna, 367  
zorientowana na położenie, 43  
usługi  
definiowanie interfejsu, 376  
przekazywanie plików parcelowanych, 388  
przekazywanie typów danych, 385  
uruchamianie i zatrzymywanie, 478  
utworzenie i zamknięcie, 478  
usługi obsługujące język AIDL, 368, 376  
usługi w Androidzie, 368  
ustanawianie menedżera alarmów, 496  
ustawianie obrazu, 196  
usuwanie błędów, 82  
usuwanie danych, 138

## V

VoIP, Voice over Internet Protocol, 597

## W

validator licencji Android Market, Market License Validator, 52  
waluta klienta, Buyer's Currency, 1016  
wartość startOffset, 529  
wątek  
narzędzia, 429  
stan Dead, 441  
stan New thread, 441  
stan Not runnable, 441  
stan Runnable, 441

wątek główny  
aktywności, 428  
dostawcy treści, 429  
informacje o stanie, 439  
odbiorcy komunikatów, 429  
opóźnianie operacji, 432  
przetrzymywanie, 432  
usługi, 429  
wątek pojedynczy, 429  
wątek roboczy, 436  
testowanie, 442  
węzły, 427  
wiadomości e-mail, 593  
widoczność menu, 272  
widok, view, 39, 58, 176  
CheckBoxPreference, 305  
contact\_entities\_view, 971  
EditTextPreference, 307  
encji kontaktów, 971  
GridView, 200  
ImageView, 221, 520  
Konta i synchronizacja, 954  
ListPreference, 313  
ListView, 200, 201, 202, 525  
MapView, 554  
MapView wraz ze znacznikami, 557  
niestandardowy - kółko, 1140  
RemoteViews, 735  
RingtonePreference, 308  
definiowanie preferencji, 309  
interfejs UI, 308  
TextView, 95, 202, 263  
view\_contacts, 971  
zdalny, 1106  
widżet, widget, 176, 736  
cykl życia, 740  
definiowanie, 740  
definiowanie dostawcy, 747  
definiowanie rozmiaru, 748  
definiowanie w pliku manifest, 740  
definiowanie w pliku XML, 741  
implementacja abstrakcyjna modelu, 754  
implementacja aktywności konfiguratora, 761  
implementacja dostawcy, 751  
implementacja modeli, 753  
implementacja modelu stanów, 758

- interfejs modelu, 753  
ksztalt tla, 750  
metody zdarzeń zwrotnych, 745  
odinstalowanie pakietów, 745  
ograniczenia i rozszerzenia, 764  
tworzenie instancji, 742  
układ graficzny, 749  
układ graficzny formularza, 763  
Urodziny, 757  
usunięcie instancji widżetu, 745  
współdzielone preferencje, 755  
widżet ekranu startowego  
  AndroidManifest.xml, 1129  
  metadane dostawcy widżetów, 1128  
  pliki projektu, 1121  
  układ graficzny widżetu, 1128  
widżet RadioButton, 192  
widżet urodzinowy, 746  
widżety ekranu startowego, 42, 736, 1105  
  lista, 738  
  tworzenie instancji widżetu, 737  
wieloczęściowa metoda POST, multipart  
  POST, 341  
wielodotykowość, 879  
  kod Java, 880  
  układ graficzny, 880  
  wyniki narzędzia LogCat, 883  
  zastosowanie, 882  
wielokrotności, 101  
wielowątkowy moduł HttpClient, 346  
wirtualna maszyna, 33  
własne pliki zasobów XML, 109  
właściwość  
  colspan, 234  
  onClick, 192  
  orientation, 228  
włączanie widoków, 603  
wskaźnik do danych, 159  
współdzielenie danych, 412  
współdziedzone identyfikatory użytkownika, 412  
współrzędne tekstury, 694  
wtyczka ADT, 82, 546  
wtyczka Galaxy Tab, 52  
wyjątek IllegalArgumentException, 629  
wyjątki protokołowe, 343  
wyjątki transportowe, 343  
wykonywanie zdjęć, 643  
wymiarystyle, 235  
  Cale, 235  
  Milimetry, 235  
  Piksele, 235  
  Piksele niezależne od gęstości, 235  
  Piksele niezależne od skali, 235  
  Punkty, 235  
wysokość pojemnika ListView, 209  
wysyłanie aplikacji, 1018  
wysyłanie komunikatu, 454  
wysyłanie powiadomienia, 464  
wyszukiwanie  
  aktywność wyszukiwania, 773  
  aplikacja odpowiedzialna za ustawienia, 775  
  dostawcy propozycji, 772  
  dostęp do aktywności testowych, 785  
  globalne, 768  
  interakcja aktywności z przyciskiem, 780, 782  
  jawne wywoływanie, 787  
  kod źródłowy aktywności, 778  
  kursor propozycji, 772  
  menu aktywności, 784  
  metadane, 793  
  pliki aktywności, 778  
  pliki projektu, 777  
  pliki układu graficznego, 778  
  pole wyszukiwania, 769  
  propozycje wyszukiwania, 771, 772  
  propozycje zerowe, 771  
  SearchInvokerActivity, 789  
  tryb propozycji zerowych, 771  
  typy aktywności, 777  
  układ graficzny aktywności, 781  
  ustawienia, 775, 777  
  wywoływanie aplikacji, 774  
wyszukiwanie globalne, 769  
  dostawcy propozycji, 774  
wyszukiwanie lokalne, 769, 790, 811  
  aktywność, 795  
  pole wyszukiwania, 796  
  wyniki wyszukiwania, 796  
wyszukiwanie w Androidzie, 768  
wywołanie dostawcy widżetu, 1117  
wywołanie obiektu Cursor, 132  
wywoływanie ekranu startowego, 166  
wywoływanie usługi, 391  
wzorce identyfikatorów URI, 148

**X**

XUL, XML User Interface Language, 39

**Z**

zabezpieczenia, 317

  certyfikat cyfrowy, 318

  stosowanie uprawnień, 325

zabezpieczenia na granicach procesu, 324

zabezpieczenia środowiska wykonawczego, 324

zakładka

  Permissions, 328

  Virtual devices, 83

Window/Android SDK and AVD  
  Manager, 58

zasady postępowania w Android Market, 1006

zasoby, 40, 91

  a zmiany konfiguracji, 112

  alternatywne, 113

  Androida, 98

  childLayout, 203

  domyślne, 113

  identyfikatory zasobów, 114

  nieskompresowane, 111

  obrazów w języku XML, 107

  plurals, 101

  R.drawable.frame\_animation, 523

  typu Color, 104

  typu color w kodzie Java, 105

  typu color-drawable, 108

  typu color-drawable w kodzie Java, 108

  typu color-drawable w kodzie XML, 108

  typu dimension, 105

  typu dimension w kodzie Java, 106

  typu dimension w kodzie XML, 106

  typu drawable, 196

  typu image, 106

  typu image w środowisku Java, 107

  typu layout, 94

  typu string, 92, 95, 103

  typu string w języku XML, 104

  typu string w kodzie Java, 103

wielokrotność, 102

zastępowanie metody funkcją, 607

zaufany wydawca certyfikatów, certificate authority, CA, 318

zdalne wywołanie procedury, 368

zdalny układ graficzny, 1109

  wczytywanie, 1111

zdarzenia dotyku, 888

zdarzenie ACTION\_MOVE, 885

zintegrowane przeszukiwanie Androida, 42

zmienna

  ignoreLastFinger, 894

  PATH, 55

  systemowa PATH, 321

  środowiskowa JAVA\_HOME, 53

  środowiskowa PATH, 55

znacznik

  <activity>, 227

  <application>, 227

  <big>, 224

  <monospace>, 224

  <small>, 224

  <strike>, 224

  <sub>, 224

  <sup>, 224

  <uses-permissions>, 334

  accelerateInterpolator, 532

group, 268

ikony menu, 271

kategorii grupy, 271

menu, 268

showAsAction, 1092

uses-permission, 625

włączania menu, 272

wyłączania menu, 272

zaznaczania, 271

**ż**

żądanie metody GET, 340

żądanie metody POST, 340

żądanie typu MIME, 129