



Ćwiczenia
praktyczne



Marcin Lis

WYDANIE **IV**

Java

ĆWICZENIA PRAKTYCZNE

Realizuj sny na Javie!

- Pakiet JDK i początki programowania, czyli jak szybko stworzyć działający program
- Obiektowość w akcji, czyli na czym polega największa zaleta Javy
- Uniwersalna składnia, czyli jak najlepiej wykorzystać przenośność tego języka

Spis treści

Wstęp	7
O książce	8
Narzędzia	9
Wersje Javy	10
Rozdział 1. Krótkie wprowadzenie	11
Instalacja JDK	11
Tryb tekstowy	11
Instalacja w systemie Windows	13
Instalacja w systemie Linux	15
Pierwszy program	17
B-kod, kompilacja i maszyna wirtualna	19
Java a C++	20
Obiektowy język programowania	21
Struktura programu	22
Rozdział 2. Zmienne, operatory i instrukcje	23
Zmienne	23
Typy podstawowe	24
Deklarowanie zmiennych typów podstawowych	25
Nazewnictwo zmiennych	28
Typy odnośnikowe	29
Deklarowanie zmiennych typów odnośnikowych	29

Operatory	32
Operatory arytmetyczne	33
Operatory bitowe	39
Operatory logiczne	41
Operatory przypisania	42
Operatory porównania (relacyjne)	42
Operator warunkowy	43
Priorytety operatorów	44
Instrukcje	45
Instrukcja warunkowa if...else	45
Instrukcja wyboru switch	50
Pętla for	53
Pętla while	57
Pętla do...while	59
Rozszerzona pętla for	61
Rozdział 3. Tablice	63
Tworzenie tablic	63
Zapis i odczyt elementów	66
Operacje z użyciem pętli	68
Rozmiar tablicy	72
Rozdział 4. Obiekty i klasy	75
Metody	77
Konstruktory	86
Specyfikatory dostępu	94
Pakiety i typy klas	101
Dziedziczenie	102
Rozdział 5. Obsługa błędów oraz wyjątki	109
Błędy w programach	109
Instrukcja try...catch	114
Zgłaszanie wyjątków	117
Hierarchia wyjątków	119
Rozdział 6. Operacje wejścia-wyjścia	123
Wyświetlanie danych na ekranie	124
Wczytywanie danych z klawiatury	126
Nowe sposoby wprowadzania danych	135
Obsługa konsoli	139
Operacje na plikach	145

Rozdział 7. Aplety	153
Aplikacja a aplet	153
Pierwszy aplet	154
Jak to działa?	157
Cykl życia apletu	158
Kroje pisma (fonty)	158
Rysowanie grafiki	161
Kolory	168
Wyświetlanie obrazów	172
Rozdział 8. Interakcja z użytkownikiem	179
Obsługa myszy	180
Rysowanie figur (I)	183
Rysowanie figur (II)	188
Rysowanie figur (III)	190
Rozdział 9. Aplikacje z interfejsem graficznym	195
Tworzenie okna aplikacji	195
Budowanie menu	199
Wielopoziomowe menu	206
Okna dialogowe	209
Rozdział 10. Grafika i komponenty	215
Rysowanie elementów graficznych	215
Obsługa komponentów	217
Przyciski JButton	218
Pola tekstowe JTextField	219
Pola tekstowe JTextArea	221
Etykiety JLabel	224
Pola wyboru JCheckBox	226
Listy rozwijane JComboBox	228
„Prawdziwa” aplikacja	230

Wstęp

Chyba każdy, kto interesuje się informatyką, słyszał o Javie. Ten stosunkowo młody (w porównaniu z C++ czy PASCalem) język programowania wyjątkowo szybko zdobył bardzo dużą popularność i akceptację ze strony programistów na całym świecie. Początkowo wiele osób kojarzyło Javę tylko z apletami zawartymi na stronach WWW. To jednak tylko niewielka część zastosowań, która dziś straciła już nieco na znaczeniu. Tak naprawdę jest to doskonały obiektowy język programowania, mający różnorodne zastosowania — od krótkich apletów do poważnych aplikacji. Początki były jednak zupełnie inne.

Być może trudno w to obecnie uwierzyć, ale język ten, pierwotnie znany jako *Oak* (z ang. *dąb*), miał służyć jako narzędzie do sterowania tzw. urządzeniami elektronicznymi powszechnego użytku, czyli wszelkiego rodzaju telewizorami, magnetowidami, pralkami czy kuchenkami mikrofalowymi. Praktycznie — dowolnym urządzeniem, które posiadało mikroprocesor. I to pierwotne przeznaczenie nie jest współcześnie mniej istotne niż kiedyś. W dobie powszechnej komputeryzacji i podłączania do sieci rozmaitych urządzeń (w tym także wspomnianych lodówek i pralek) takie zastosowanie wręcz zwiększa, a nie zmniejsza atrakcyjność języka. Stąd też wywodzi się jedna z największych zalet Javy — jej przenośność, czyli możliwość uruchamiania jednego programu na wielu różnych platformach. Skoro bowiem

miała służyć do programowania dla tak wielu różnorodnych urządzeń, musiała być niezależna od platformy sprzętowo-systemowej. Ten sam program można będzie więc uruchomić, przynajmniej teoretycznie, zarówno na komputerze PC i Macintosh, jak i w Windowsie oraz Uniksie.

Historia *Oak* rozpoczęła się pod koniec 1990 roku. Język został opracowany jako część projektu o nazwie Green, rozpoczętego przez Patricka Naughtona, Mike'a Sheridana i Jamesa Goslinga w firmie Sun Microsystems. Język był opracowany już w 1991 roku, jednak do 1994 roku nie udało się go spopularyzować i prace nad projektem zostały zawieszone. Był to jednak czas gwałtownego rozwoju sieci Internet i okazało się, że *Oak* doskonale sprawdzałby się w tak różnorodnym środowisku, jakim jest globalna sieć. W ten oto sposób w 1995 roku światło dzienne ujrzała Java.

To, co stało się później, zaskoczyło chyba wszystkich, w tym samych twórców języka. Java niewiarygodnie szybko została zaakceptowana przez społeczność internetową i programistów na całym świecie. Niewątpliwie bardzo duży wpływ miała tu umiejętnie prowadzona kampania marketingowa producenta. Niemniej decydujące były z pewnością wyjątkowe zalety tej technologii. Java to bardzo dobrze skonstruowany język programowania, który programistom zwykle przypada do gustu już przy pierwszym kontakcie. W każdym razie o Javie mówią i piszą wszyscy, pojawiają się setki książek i stron internetowych, powstają w końcu napisane w niej programy. Obecnie to już dojrzała, choć wciąż rozwijana technologia, która wrosła na dobre w świat informatyki.

O książce

Celem niniejszej publikacji jest przedstawienie nie wszystkich aspektów programowania w Javie, ale jedynie pewnego wycinka tego zagadnienia. Omówione zostały podstawowe zasady programowania, zamieszczono przykłady tworzenia apletów, czyli programów osadzanych w stronach WWW, a także zaprezentowano podstawy tworzenia aplikacji z graficznym interfejsem oraz operacje wejścia-wyjścia. Niestety, ze względu na ograniczoną ilość miejsca nie można było przedstawić wielu ciekawych i bardziej zaawansowanych zagadnień, dlatego też będzie to raczej wycieczka po programowaniu w Javie niż

metodyczny kurs opisujący całość zagadnienia. Jak jednak wskazuje sam tytuł, ta publikacja to ćwiczenia, które mają pozwolić na szybkie zapoznanie się z podstawowymi konstrukcjami języka, niezbędnymi do rozpoczęcia programowania. Niniejsze ćwiczenia będą więc zarówno doskonałym podręcznikiem dla osób, które by szybko chciały poznać się ze strukturą języka, jak i uzupełnieniem bardziej metodycznego kursu, jakim jest np. publikacja *Praktyczny kurs Java* (<http://helion.pl/ksiazki/pkjav4.htm>). Zagadnienia zaawansowane, np. tworzenie aplikacji sieciowych lub bazodanowych, zostały też przedstawione w publikacji *Ćwiczenia zaawansowane* (<http://helion.pl/ksiazki/czjav2.htm>).

Narzędzia

Aby rozpocząć programowanie w Javie, niezbędne są odpowiednie narzędzia. Konkretnie — kompilator oraz maszyna wirtualna, która interpretuje skompilowane programy. Będziemy opierać się tu na pakiecie Java Development Kit (JDK). Można skorzystać z wersji sygnowanej przez oficjalnego producenta Javy — firmę Oracle¹ — lub rozwijanej na zasadach wolnego oprogramowania wersji OpenJDK. Wersja oficjalna dostępna jest pod adresami <http://www.oracle.com/java/>, <http://java.sun.com> (w tym przypadku nastąpi przekierowanie do domeny [oracle.com](http://www.oracle.com)) oraz <http://www.java.com>, a OpenJDK pod adresem <http://openjdk.java.net/>. Najlepiej korzystać z możliwie nowej wersji JDK, tzn. 1.7 (7.0), 1.8 (8.0) lub wyższej (o ile taka będzie dostępna), choć podstawowe przykłady będą działać nawet na bardzo wiekowej już wersji 1.1.

Przy ćwiczeniach omawiających tworzenie apletów można skorzystać z dowolnej przeglądarki internetowej obsługującej język Java lub też dostępnej w JDK aplikacji *appletviewer*. Większość obecnie dostępnych na rynku przeglądarek udostępnia Javę poprzez mechanizm wtyczek, umożliwiając zastosowanie najnowszych wersji JRE (ang. *Java Runtime Environment*), czyli środowiska uruchomieniowego. Należy z tej możliwości skorzystać.

¹ Pierwotny producent — Sun Microsystems — został zakupiony przez Oracle w 2009 roku. Tym samym obecnie to Oracle oficjalnie odpowiada za rozwój Javy.

Oprócz JDK będzie potrzebny jedynie dowolny edytor tekstowy (korzystającym ze środowiska Windows można polecić np. doskonały Notepad++) pozwalający na wpisywanie tekstu programów i zapisywanie ich w plikach na dysku. Istnieje co prawda możliwość używania zintegrowanych środowisk programistycznych, jednak dla osób początkujących lepszym zestawem byłby JDK i zwykły edytor tekstowy (najlepiej programistyczny), tak aby najpierw poznać dobrze sam język i jego właściwości, a dopiero później bardziej zaawansowane narzędzia (oczywiście jeśli ktoś woli pracę w środowiskach takich jak NetBeans czy Eclipse, może ich również z powodzeniem używać do nauki).

Wersje Javy

Pierwsza powszechnie wykorzystywana wersja Javy nosiła numer 1.1 (JDK 1.1 i JRE 1.1). Stosunkowo szybko pojawiła się jednak kolejna wersja, oznaczona numerem 1.2. Niosła ona ze sobą na tyle znaczące zmiany i usprawnienia, że nadano jej nazwę Platforma Java 2 (ang. *Java 2 Platform*). Tym samym wersja poprzednia została nazwana Platformą Java 1. W ramach projektu Java 2 powstały trzy wersje narzędzi JDK i JRE: 1.2, 1.3 i 1.4, a każda z nich miała od kilku do kilkunastu podwersji. Kolejnym krokiem w rozwoju projektu była wersja 1.5, która ze względów czysto marketingowych została przemianowana na 5.0. Następnie pojawiły się wersje 6.0 (czyli 1.6), 7 (czyli 1.7) oraz 8 (czyli 1.8). Podczas przygotowywania materiałów do niniejszej książki używane były wersje 6, 7 i 8. Jeśli stosowano konstrukcje języka dostępne wyłącznie w wersji 7 lub 8, było to oznaczane w opisach przykładów. Warto przy tym wspomnieć, że wewnętrzna numeracja narzędzi (widoczna np. w opcjach kompilatora *javac* i narzędzia uruchomieniowego *java*) wciąż bazuje na wcześniejszej, logicznej numeracji (czyli Java 6.0 jest tożsama z Java 1.6, Java 7 to inaczej Java 1.7, a Java 8 to inaczej Java 1.8).

1

Krótkie wprowadzenie

Instalacja JDK

Instalacja pakietu JDK, który umożliwi nam tworzenie aplikacji, nikomu nie powinna przysporzyć problemów, i to niezależnie od tego, czy wybrana zostanie wersja dla systemu Linux, Windows czy też dla innej platformy. JDK instaluje się bowiem podobnie jak każdą inną aplikację. Proces ten jest także opisany na stronach internetowych producenta. Podane niżej przykłady instalacji odnoszą się do wersji 1.5 – 1.8, dla systemów 32- i 64-bitowych.

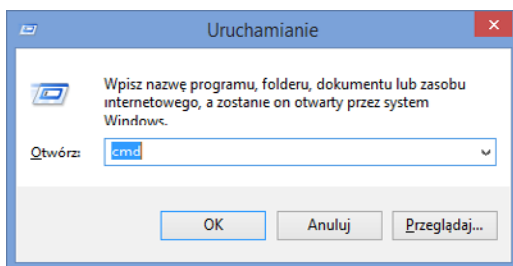
Tryb tekstowy

Kompilator zawarty w pakiecie JDK pracuje w trybie tekstowym, w takim też trybie będziemy uruchamiać pierwsze napisane przez nas programy ilustrujące cechy języka. Ponieważ w dobie systemów

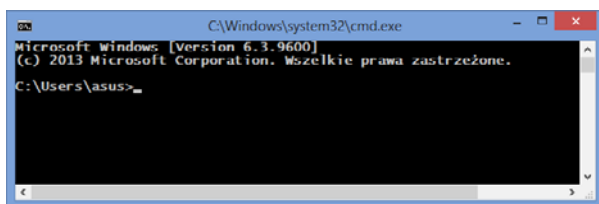
oferujących graficzny interfejs użytkownika z takiego trybu korzysta się coraz rzadziej, zobaczmy, jak uruchomić go w systemach Linux i Windows.

Jeśli pracujemy w systemie Windows, wciskamy na klawiaturze kombinację klawiszy Windows+R¹, w polu *Otwórz* (w niektórych wersjach systemu — *Uruchom*) wpisujemy `cmd` lub `cmd.exe`² i klikamy przycisk *OK* lub wciskamy klawisz *Enter* (rysunek 1.1). (W systemach XP i starszych, pole *Uruchom* jest też dostępne bezpośrednio w menu *Start*³). Można też w dowolny inny sposób uruchomić aplikację `cmd.exe`. Pojawi się wtedy okno wiersza poleceń (wiersza polecenia), w którym będzie można wydawać komendy. Jego wygląd dla systemów z rodziny Windows 8 został przedstawiony na rysunku 1.2 (w innych wersjach wygląda bardzo podobnie).

Rysunek 1.1.
*Uruchamianie
wiersza poleceń
w Windows 8*



Rysunek 1.2.
*Okno konsoli
(wiersza poleceń)
w systemie
Windows 8*



Jeśli pracujemy w Linuksie na konsoli tekstowej, nie trzeba oczywiście wykonywać żadnych dodatkowych czynności. Jeżeli jednak ko-

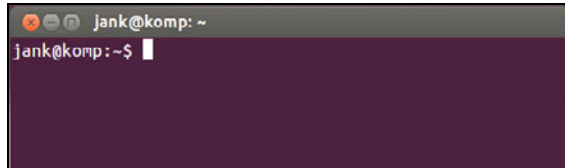
¹ Klawisz funkcyjny *Windows* jest też opisywany jako *Start*.

² W starszych systemach (Windows 98, Me) należy uruchomić aplikację `command.exe` (*Start/Uruchom/command.exe*).

³ W systemie Windows w wersjach Vista i 7 pole *Uruchom* standardowo nie jest dostępne w menu startowym, ale można je do niego dodać, korzystając z opcji *Dostosuj*.

rzystamy z interfejsu graficznego, należy uruchomić program terminala (konsoli). Jego przykładowy wygląd dla dystrybucji Ubuntu (wersja 13.10) został zaprezentowany na rysunku 1.3.

Rysunek 1.3.
*Wygląd konsoli
tekstowej
w dystrybucji
Ubuntu*



Dokładniejsze informacje o posługiwaniu się konsolą systemową można znaleźć w dokumentacji systemów operacyjnych.

Instalacja w systemie Windows

Wersja instalacyjna dla systemów z rodziny Windows jest dystrybuowana w postaci pliku o nazwie:

`jdk-wersja-windows-proc.exe`

np.:

`jdk-8-windows-i586.exe`
`jdk-1.7.0-windows-i586.exe`

Uruchomienie tego pliku spowoduje rozpoczęcie typowego procesu instalacji aplikacji. Uwaga: Java 8 oficjalnie nie współpracuje z systemami Windows poniżej wersji 7.

Po instalacji dobrze jest dopisać do zmiennej środowiskowej PATH ścieżkę dostępu do katalogu *bin* środowiska JDK. Aby zmienić ten parametr dla bieżącej sesji⁴ konsoli systemowej (po jej uruchomieniu w sposób opisany w poprzednim punkcie), należy wydać polecenie:

`path=%path%;ścieżka_dostępu`

np.:

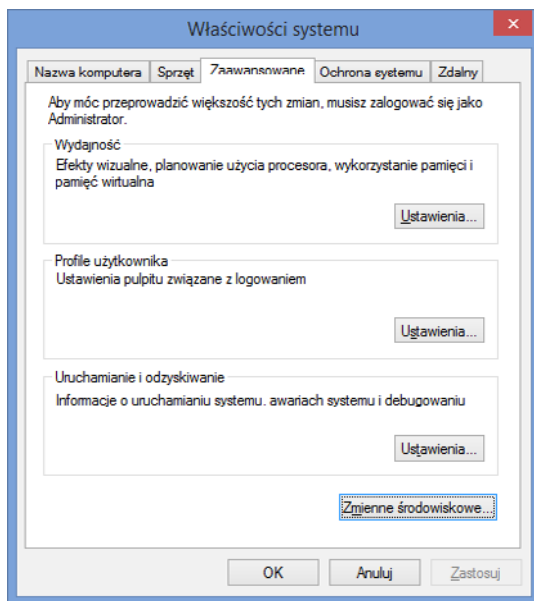
`path=%path%;c:\Program Files\java\jdk1.8.0\bin`

⁴ Ustawienie nie będzie obowiązywało ani w innych oknach konsoli, ani też po zamknięciu bieżącego okna.

zakładając, że mamy do czynienia ze standardową instalacją JDK w wersji 8 (rysunek 1.4). Aby dodać wspomnianą wartość na stałe (tak aby obowiązywała zawsze, w każdej sesji konsoli), należy uruchomić panel właściwości systemu (w sposób odpowiedni dla danej wersji Windows — typowo *Mój komputer/Właściwości*. Można też skorzystać ze skrótu klawiaturowego *Windows+Pause*), wybrać w nim zakładkę *Zaawansowane* (lub *Zaawansowane ustawienia systemu* zależnie od posiadanej wersji Windows) i kliknąć znajdujący się w niej przycisk *Zmienne środowiskowe* (rysunek 1.4).

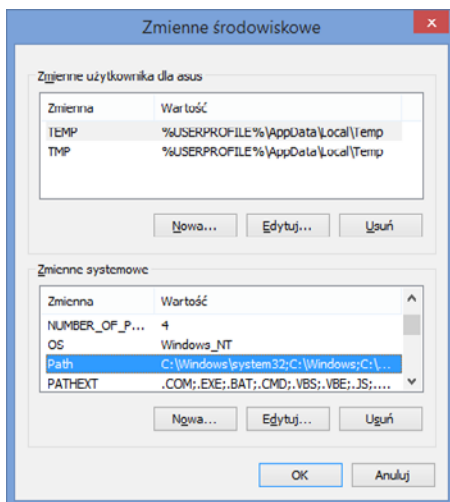
Rysunek 1.4.

Panel
właściwości
systemu

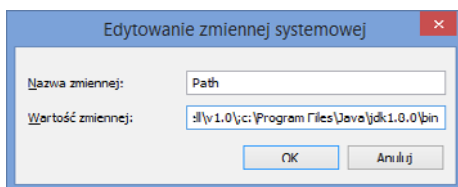


W oknie *Zmienne środowiskowe* w części zatytułowanej *Zmienne systemowe* należy odszukać wpis dotyczący zmiennej *Path* (rysunek 1.5), zaznaczyć go i kliknąć widoczny poniżej przycisk *Edytuj*. W kolejnym oknie (rysunek 1.6) możliwa będzie edycja wpisu. Do istniejących danych należy dopisać (po średniku) ścieżkę dostępu do katalogu *bin* środowiska JDK — podobnie jak w przykładzie przedstawionym wyżej. Uwaga: nie należy kasować istniejących wartości!

Rysunek 1.5.
*Okno zmiennych
środowiskowych
systemu*



Rysunek 1.6.
*Edycja
zawartości
zmiennej
systemowej*



Instalacja w systemie Linux

Do wyboru mamy instalację za pomocą dystrybucji źródłowej lub w postaci pakietu RPM. W pierwszym przypadku do dyspozycji jest plik o nazwie:

```
jdk-wersja-linux-proc.tar.gz
```

w której *wersja* jest numerem wersji dystrybucji, a *proc* — określeniem rodziny procesorów, np.:

```
jdk-8-linux-i586.tar.gz  
jdk-1.7.0-linux-i586.tar.gz
```

Należy go skopiować lub przenieść do wybranego katalogu na dysku (do katalogu, w którym ma być zainstalowana Java), wydając polecenie (kopiowanie):

```
cp ./jdk-1.8.0-linux-i586.tar.gz /usr/java/
```

lub (przenoszenie):

```
mv ./jdk-1.8.0-linux-i586.tar.gz /usr/java/
```

o ile katalogiem docelowym jest `/usr/java`.

Następnie należy przejść do tego katalogu, wydając komendę:

```
cd /usr/java
```

oraz rozpakować plik, wpisując polecenie:

```
tar xvfz jdk-1.8.0-linux-i586.tar.gz
```

Wtedy w bieżącym katalogu zostanie utworzony podkatalog o nazwie `jdk1.8.0`, w którym znajdują się pliki pakietu.

Aby usprawnić pracę z JDK, po instalacji warto do zmiennej środowiskowej `PATH` dodać ścieżkę do podkatalogu `bin` tego pakietu (np. `/usr/java/jdk1.8.0/bin/`, o ile JDK zostało zainstalowane w katalogu `/usr/java/jdk1.8.0/`) — nie będziemy wtedy musieli za każdym razem podawać pełnej ścieżki dostępu, aby uruchomić kompilator czy też inne narzędzie (rysunek 1.7). Aby wykonać tę operację dla bieżącej sesji w przypadku powłoki systemowej `bash`, wykonujemy polecenie:

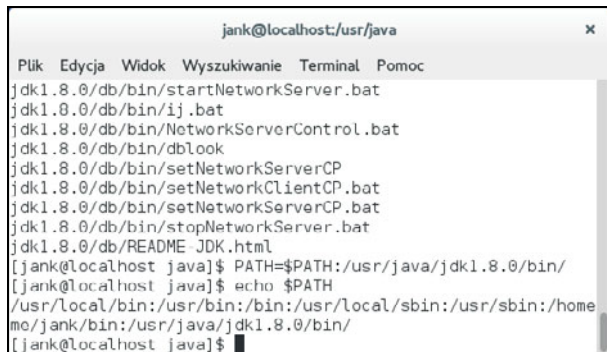
```
PATH=$PATH:/ścieżka_dostępu
```

np.:

```
PATH=$PATH:/usr/java/jdk1.8.0/bin/
```

Jeżeli zmiany mają być zapisane na stałe, należy odpowiednie modyfikacje wprowadzić do pliku `.bash_profile` znajdującego się w katalogu domowym danego użytkownika systemu.

Rysunek 1.7.
Modyfikacja
zmiennej
systemowej
`PATH`



W drugim przypadku (pakiet RPM) do dyspozycji mamy plik o nazwie:

```
jdk-wersja-linux-proc-rpm
```

np.:

```
jdk-8-linux-i586-rpm
```

Może on być instalowany przez użytkownika posiadającego uprawnienia administratora systemu (w dystrybucjach, które obsługują pakiety typu *rpm*, np. RedHat, Fedora). Instalacja odbywa się przez wydanie polecenia:

```
rpm -ivh jdk-8-linux-i586.rpm
```

Po jej zakończeniu pakiet JDK jest gotowy do użycia.

Pierwszy program

Zacznijmy tradycyjnie, tak jak w większości kursów dotyczących programowania. Napišemy prostą aplikację, której jedynym zadaniem będzie wyświetlenie na ekranie napisu.

Ć W I C Z E N I E

1.1 Aplikacja wyświetlająca tekst

Napisz program wyświetlający na ekranie dowolny napis.

```
class Main
{
    public static void main (String args[])
    {
        System.out.println ("Pierwszy program w Javie");
    }
}
```

1. Plik z tekstem programu zapisujemy w wybranym katalogu roboczym pod nazwą *Main.java*. W wierszu poleceń zmieniamy katalog bieżący na ten, w którym zapisaliśmy plik. Wydajemy więc polecenie:

```
cd dysk:\ścieżka_dostępu\nazwa_katalogu
```

np.:

```
cd c:\java\projekty
```

w systemie Windows lub:

```
cd ścieżka_dostępu/nazwa_katalogu
```

np.:

```
/home/jank/java/
```

w systemie Linux.



Wielkość liter ma znaczenie! I to zarówno w tekście programu, jak i w nazwie pliku. Jeśli się pomylimy, kompilacja się nie uda.

2. Nie będziemy w tej chwili analizować sposobu działania tej aplikacji, postaramy się natomiast zobaczyć wyniki jej działania na ekranie. Przystępujemy do kompilacji, pisząc (przy założeniu, że pakiet JDK jest już poprawnie zainstalowany w systemie, a zmienna systemowa PATH wskazuje ścieżkę dostępu do katalogu *bin* pakietu JDK⁵):

```
javac Main.java
```

3. Jeżeli kompilacja (bliższe wyjaśnienie tego terminu znajduje się w kolejnym podrozdziale) się powiedzie, na dysku powstanie nowy plik o nazwie *Main.class*. Aby uruchomić program, piszemy⁶:

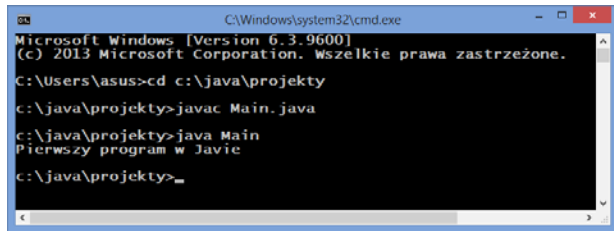
```
java Main
```

Efekt działania jest widoczny na rysunku 1.8. Zwróćmy uwagę, że w przypadku kompilacji podawaliśmy nazwę pliku z rozszerzeniem (*Main.java*), natomiast przy uruchamianiu rozszerzenie pominęliśmy (*Main*).

⁵ Jeżeli zmienna środowiskowa PATH nie zawiera wskazanej ścieżki dostępu, należy ją podać przy wywoływaniu kompilatora *javac*. Instrukcja kompilująca program wyglądałaby wtedy np. tak: `/usr/java/jdk1.8.0/bin/javac Main.java` lub tak: `c:\Program Files\java\jdk8\bin\javac Main.java`. W Linuksie może być też konieczne dodanie do nazwy kompilowanego pliku wskazania do katalogu bieżącego (o ile wskazanie tego katalogu nie znajduje się w zmiennej środowiskowej PATH). Wtedy zamiast *Main.java* trzeba użyć odwołania `./Main.java`.

⁶ Mają tu zastosowanie te same uwagi, które zostały podane w poprzednim przypisie.

Rysunek 1.8.
Efekt działania
pierwszego
programu
w Javie



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. Wszelkie prawa zastrzeżone.
C:\Users\asus>cd c:\java\projekty
c:\java\projekty>javac Main.java
c:\java\projekty>java Main
Pierwszy program w Javie
c:\java\projekty>_
```

B-kod, kompilacja i maszyna wirtualna

Każdy napisany program przed wykonaniem musi być przetłumaczony na język zrozumiały dla komputera (procesora), czyli poddany procesowi kompilacji. W przypadku takich języków jak C, C++ czy Pascal jest to zazwyczaj kompilacja do kodu natywnego procesora, tzn. kodu, który procesor jest w stanie bezpośrednio wykonać. W przypadku Javy tak jednak być nie może, ponieważ programy nie byłyby wtedy przenośne. Każdy procesor ma przecież inny zestaw instrukcji, a zatem program skompilowany dla jednego, nie byłby zrozumiały dla drugiego. Dlatego też w przypadku Javy efektem kompilacji jest kod pośredni, tzw. *b-kod* (ang. *b-code*, *byte-code*). Jest on zawarty w plikach z rozszerzeniem *class*. W naszym przypadku tekst programu (nazywany kodem źródłowym) z pliku *Main.java* został skompilowany do b-kodu, który zapisano w pliku *Main.class*.

Kompilacja została wykonana przez kompilator — program *javac*. Kompilator ten pracuje w wierszu poleceń i przyjmuje w postaci argumentu jeden plik lub więcej plików z kodem źródłowym. Zatem jego schematyczne wywołanie to:

```
javac nazwa_pliku.java
```

lub też, jeśli plików jest kilka:

```
javac plik1.java plik2.java plik3.java
```

Aby b-kod mógł zostać zrozumiany przez procesor, musi być ponownie przetłumaczony. Dokonuje tego tzw. maszyna wirtualna Javy, czyli interpreter b-kodu dla danego typu procesora (i systemu opera-

cyjnego). Wynika z tego, że wystarczy, aby dla każdego systemu powstała dedykowana mu maszyna wirtualna Javy (środowisko uruchomieniowe), a będzie można uruchomić w nim każdy program w Javie bez wykonywania dodatkowych modyfikacji (to oczywiście pewne uproszczenie, nie uruchomimy np. programu graficznego na konsoli tekstowej). W pierwszych wersjach Javy była to, niestety, również jedna z przyczyn jej powolności. Kod interpretowany jest bowiem wolniejszy od wykonywanego bezpośrednio na danym procesorze. Obecnie nie ma to już tak dużego znaczenia jak kiedyś, gdyż są dostępne doskonałe maszyny wirtualne, zawierające np. kompilatory JIT (ang. *Just In Time*), które w locie kompilują część (lub nawet całość) b-kodu do kodu natywnego procesora — niemniej Java będzie zazwyczaj wolniejsza⁷ niż czyste C czy C++⁸.

Java a C++

Osobom, które znają C++, Java zapewne od razu się spodoba. Choćby dlatego, że jej składnia jest do C++ bardzo podobna. Jest to jednak miejscami nieco złudne, gdyż ten sam zapis w C++ wcale nie musi oznaczać dokładnie tego samego w Javie. Mówiąc ściślej, syntaktyka jest podobna, ale semantyka inna. Trzeba jednak dodać, że bardzo szybko wychwytuje się te różnice i choć w początkowym okresie owe różnice mogą nieco przeszkadzać, w niedługim czasie przestają sprawiać jakikolwiek problem. Przedstawiony w ćwiczeniu 1.1 program w Javie wyświetlający na ekranie napis wyglądałby w C++ następująco:

```
#include <iostream>
int main (int argc, char **argv)
{
    cout << "Pierwszy program w Javie";
    return 0;
}
```

⁷ To nie zawsze musi być prawda. Współczesne środowiska uruchomieniowe mogą wykonywać wielokrotną optymalizację i kompilację do kodu natywnego, dostosowując strukturę wynikową do warunków danego systemu i przechowując prekompilowane fragmenty w pamięci. To może skutkować lepszą wydajnością niż w przypadku klasycznej kompilacji statycznej.

⁸ Obecnie nawet programy w C++ kompiluje się do kodu pośredniego. Tak jest np. w środowisku .NET.

Można by tu pominąć nieużywane argumenty funkcji `main`, pisząc:

```
#include <iostream>
int main ()
{
    cout << "Pierwszy program w Javie";
    return 0;
}
```

a oba programy wciąż będą funkcjonalnymi odpowiednikami. W przypadku Javy nie jest to już jednak możliwe. Argumenty funkcji `main` muszą być dokładnie takie jak w ćwiczeniu 1.1 (nie można ich pominąć).

Obiektowy język programowania

Java jest w pełni obiektowym (czy też zorientowanym obiektowo⁹) językiem programowania, tzn. programowanie odbywa się tutaj poprzez definiowanie obiektów i metod z nimi związanych. Cóż to jednak znaczy? Co to jest obiekt? Otóż w świecie rzeczywistym obiektem może być wszystko: drzewo, samochód, pies, telewizor. Tak samo w świecie komputerowym może to być np. okno czy punkt na ekranie lub też dowolny inny abstrakcyjny byt wymyślony przez programistę. Każdy obiekt ma swoje właściwości, które są opisywane przez pola obiektu oraz operujące na nim metody. Można powiedzieć, że obiekt przechowuje dane oraz wykonuje zaprogramowane czynności.

Klasa to opis obiektu. Mówi się, że obiekt jest wystąpieniem (inaczej instancją) danej klasy. Klasa jest też typem obiektu. Metody są to natomiast funkcje, które można wykonywać na danym obiekcie, np. obiekt punkt może mieć metodę `przesuń`, która ustali jego pozycję na ekranie.

Zapewne dla osób, które nie zetknęły się z programowaniem obiektowym, jest to całkowicie zagmatwane i mało zrozumiałe. Nie należy się tym jednak przejmować. Po kolei dojdziemy do wszystkiego.

⁹ Obecnie różnica między terminami „obiektowy” a „zorientowany obiektowo” (ang. *object oriented*) jest czysto akademicka i o ile kiedyś mogła mieć znaczenie, to współcześnie obu terminów można używać zamiennie (choć spotyka się też purystów twierdzących, że jest to absolutnie niedopuszczalne).

Struktura programu

Każdy program w Javie składa się ze zbioru klas. Na razie przyjmiemy zasadę, że najlepiej, aby klasa została zapisana w pliku o takiej samej nazwie jak nazwa tej klasy oraz o rozszerzeniu *java*. Tę zasadę będziemy stosować w pierwszej części książki (ta kwestia zostanie dokładniej wyjaśniona w rozdziale 4.). Wykonywanie programu rozpoczyna się od metody (funkcji) *main*, która musi być zadeklarowana jako publiczna i statyczna. Struktura programu w przypadku naszego pierwszego przykładu była następująca:

```
class Main
{
    public static void main (String args[])
    {
        System.out.println ("Pierwszy program w Javie");
    }
}
```

Jak widać, zadeklarowaliśmy klasę (słowo *class*) o nazwie *Main*. W ciele tej klasy (w jej wnętrzu), tzn. pomiędzy znakami nawiasu klamrowego: { i }, została zadeklarowana metoda *main*. Przed nią znajdują się słowa *public* i *static*, czyli jest spełniony warunek o jej publiczności i statyczności (o tym, co tak naprawdę to oznacza, będziemy mówić później). Wykonanie rozpocznie się zatem od tej metody. W ciele funkcji *main* znajduje się z kolei instrukcja (można by powiedzieć: polecenie) powodująca wypisanie na ekranie tekstu *Pierwszy program w Javie*. Na razie nasze programy ilustrujące cechy języka będziemy pisać właśnie w taki sposób, aby zawierały instrukcje w ciele (we wnętrzu) funkcji *main*, przyjmując na słowo, że tak właśnie ma być.

Warto zauważyć, że programy w Javie często formatuje się nieco inaczej:

```
class Main {
    public static void main (String args[]) {
        System.out.println ("Pierwszy program w Javie");
    }
}
```

Dla osób początkujących ta wersja z reguły jest mniej czytelna. To jednak kwestia czysto techniczna. Należy stosować tę wersję, która jest dla nas wygodniejsza (biorąc jednak pod uwagę, że praktycy częściej stosują powyższy sposób zapisu).

2

Zmienne, operatory i instrukcje

Zmienne

Zmienna jest to miejsce, w którym możemy przechowywać jakieś dane, np. liczby czy ciągi znaków. Każda zmienna musi mieć swoją nazwę, która ją jednoznacznie identyfikuje, a także typ informujący o tym, jakiego rodzaju dane można w niej przechowywać. Np. zmienna typu `int` przechowuje liczby całkowite, a zmienna typu `float` — liczby zmiennoprzecinkowe. Typy w Javie dzielą się na dwa rodzaje: *typy podstawowe* (ang. *primitive types*) oraz *typy odnośnikowe* (ang. *reference types*).

Typy podstawowe

Typy podstawowe dzielą się na:

- ❑ typy całkowitoliczbowe (ang. *integral types*),
- ❑ typy zmiennopozycyjne (rzeczywiste, ang. *floating-point types*),
- ❑ typ boolean,
- ❑ typ char.

Typy całkowitoliczbowe

Rodzina typów całkowitoliczbowych składa się z czterech typów:

- ❑ byte,
- ❑ short,
- ❑ int,
- ❑ long.

W przeciwieństwie do innych języków, takich jak np. C++, szczegółowo określono sposób reprezentacji tych danych. Niezależnie więc od tego, w jakim systemie pracujemy (16-, 32- czy 64-bitowym), dokładnie wiadomo, na ilu bitach zapisana jest zmienna danego typu. Wiadomo też, z jakiego zakresu może ona przyjmować wartości, nie ma więc dowolności, która w przypadku np. języka C mogła prowadzić do trudności przy przenoszeniu programów między różnymi platformami. W tabeli 2.1 zaprezentowano zakresy poszczególnych typów danych oraz liczbę bitów niezbędną do zapisania zmiennych danego typu.

Tabela 2.1. Zakresy dla typów arytmetycznych w Javie

Typ	Liczba bitów	Liczba bajtów	Zakres
byte	8	1	od -128 do 127
short	16	2	od -32 768 do 32 767
int	32	4	od -2 147 483 648 do 2 147 483 647
long	64	8	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807

Typy zmiennoprzecinkowe

Typy zmiennoprzecinkowe występują tylko w dwóch odmianach:

- ❑ float (pojedynczej precyzji),
- ❑ double (podwójnej precyzji).

Zakres oraz liczbę bitów i bajtów potrzebnych do zapisu tych zmiennych zaprezentowano w tabeli 2.2.

Tabela 2.2. Zakresy dla typów zmiennoprzecinkowych w Javie

Typ	Liczba bitów	Liczba bajtów	Zakres
float	32	4	od $-3,4\text{e}38$ do $3,4\text{e}38$
double	64	8	od $-1,8\text{e}308$ do $1,8\text{e}308$

Format danych float i double jest zgodny ze specyfikacją standardu ANSI/IEEE 754. Zapis $3,4\text{e}38$ oznacza $3,4 \times 10^{38}$.

Typ boolean

Jest to typ logiczny. Może on reprezentować jedynie dwie wartości: true (prawda) i false (fałsz). Może być wykorzystywany przy sprawdzaniu różnych warunków w instrukcjach if, a także w pętlach i innych konstrukcjach programistycznych, które zostaną przedstawione w dalszej części rozdziału.

Typ char

Typ char służy do reprezentacji znaków (liter, znaków przestankowych i innych), przy czym w Javie jest on 16-bitowy i zawiera znaki Unicode. Ponieważ tak naprawdę znaki są reprezentowane jako 16-bitowe kody liczbowe, typ ten zaliczany jest czasem do typów arytmetycznych.

Deklarowanie zmiennych typów podstawowych

Aby móc użyć w programie jakiejś zmiennej, najpierw trzeba ją zadeklarować, tzn. podać jej typ oraz nazwę. Ogólna deklaracja wygląda następująco:

```
typ_zmiennej nazwa_zmiennej;
```

Po takiej deklaracji zmienna jest już gotowa do użycia, tzn. można jej przypisywać różne wartości bądź też wykonywać na niej różne operacje, np. dodawanie. Przypisanie wartości zmiennej odbywa się za pomocą znaku (operatora) =. Pierwsze przypisanie wartości zmiennej nazywamy *inicjacją zmiennej* lub *inicjalizacją zmiennej*¹.

Ć W I C Z E N I E

2.1 Deklarowanie zmiennych

Zadeklaruj dwie zmienne całkowite i przypisz im dowolne wartości. Wyniki wyświetl na ekranie.

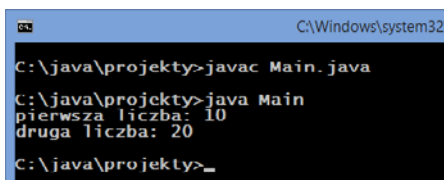
```
class Main {  
    public static void main (String args[]) {  
        int pierwszaLiczba;  
        int drugaLiczba;  
        pierwszaLiczba = 10;  
        drugaLiczba = 20;  
        System.out.println ("pierwsza liczba: " + pierwszaLiczba);  
        System.out.println ("druga liczba: " + drugaLiczba);  
    }  
}
```

Zostały tu zadeklarowane dwie zmienne typu `int` (umożliwiające przechowywanie liczb całkowitych): `pierwszaLiczba` i `drugaLiczba`. Następnie przypisano im dwie wartości całkowite: 10 i 20. Na zakończenie została użyta instrukcja² `System.out.println`, która pozwala wyprowadzić ciąg znaków na ekran (ciągi znaków ujęte w cudzysłowy zostały połączone z wartościami zmiennych za pomocą operatora +). Po skompilowaniu i uruchomieniu takiego programu zobaczymy więc na ekranie widok przedstawiony na rysunku 2.1.

¹ Choć w żargonie technicznym częściej używa się słowa „inicjalizacja”, z formalnego punktu widzenia prawidłowym terminem jest „inicjacja”. Wynika to jednak wyłącznie z tego, że termin ten pojawił się w języku polskim znacząco wcześniej niż kojarzona chyba wyłącznie z informatyką „inicjalizacja” (kalka z ang. *initialization*). Co więcej, np. w języku angielskim funkcjonują oba terminy (*initiation* oraz *initialization*) i mają nieco inne znaczenia. Nie wnikając jednak w niuanse językowe, można powiedzieć, że w przedstawionym znaczeniu oba te terminy mogą być (i są) używane zamiennie.

² W rzeczywistości jest to wywołanie statycznej metody `println`. Ten temat zostanie poruszony w dalszej części książki.

Rysunek 2.1.
Wynik działania
programu
z ćwiczenia 2.1



```
C:\Windows\system32
C:\java\projekty>javac Main.java
C:\java\projekty>java Main
pierwsza liczba: 10
druga liczba: 20
C:\java\projekty>_
```

Wartość zmiennej można również przypisać już w trakcie deklaracji, pisząc:

```
typ_zmiennej nazwa_zmiennej = wartość;
```

Można również zadeklarować wiele zmiennych danego typu, odzielając ich nazwy przecinkami. Część z nich może być też od razu zainicjowana:

```
typ_zmiennej nazwa1, nazwa2, nazwa3;
typ_zmiennej nazwa1 = wartość1, nazwa2, nazwa3 = wartość2;
```

Zmienne w Javie, podobnie jak w C czy C++, ale inaczej niż w Pascalu, można deklarować wedle potrzeb w dowolnym miejscu programu³.

Ć W I C Z E N I E

2.2 Jednoczesna deklaracja i inicjacja zmiennych

Zadeklaruj i jednocześnie zainicjuj dwie zmienne typu całkowitego. Wynik wyświetl na ekranie.

```
class Main {
    public static void main (String args[]) {
        int pierwszaLiczba = 10;
        int drugaLiczba = 20;
        System.out.println ("pierwsza liczba: " + pierwszaLiczba);
        System.out.println ("druga liczba: " + drugaLiczba);
    }
}
```

³ Oczywiście w dowolnym miejscu, w którym ze względów składniowych deklaracja jest dopuszczalna.

ĆWICZENIE

2.3 Deklarowanie zmiennych w jednym wierszu

Zadeklaruj w jednym wierszu kilka zmiennych typu całkowitego. Niektóre z nich zainicjuj.

```
class Main {  
    public static void main (String args[]) {  
        int pierwszaLiczba = 10, drugaLiczba = 20, i, j, k;  
        System.out.println ("pierwsza liczba: " + pierwszaLiczba);  
        System.out.println ("druga liczba: " + drugaLiczba);  
    }  
}
```

Nazewnictwo zmiennych

Przy nazywaniu zmiennych obowiązują pewne zasady. Otóż nazwa może się składać z wielkich i małych liter oraz cyfr, znaku podkreślenia i znaku dolara. Nie może się jednak zaczynać od cyfry. Można przy tym używać polskich znaków (a dokładniej — dowolnych znaków Unicode; zatem znaki niemieckie, portugalskie czy nawet chińskie są również dozwolone). Często jednak korzysta się wyłącznie ze znaków alfabetu łacińskiego (zależy to jednak od konwencji przyjętej w danym projekcie). Nazwa zmiennej powinna także odzwierciedlać funkcję pełnioną w programie. Jeżeli na przykład określa ona liczbę punktów w jakimś zbiorze, to najlepiej nazwać ją `liczbaPunktow` lub nawet `liczbaPunktowWZbiorze` (poprawne będą też nazwy `liczbaPunktów` i `liczbaPunktówWZbiorze`). Mimo że tak długa nazwa może wydawać się dziwna, jednak bardzo poprawia czytelność programu oraz ułatwia jego analizę. Naprawdę warto ten sposób stosować. Z reguły przyjmuje się też, co również jest bardzo wygodne, że nazwę zmiennej rozpoczynamy małą literą, a poszczególne człony tej nazwy (wyrazy, które się na nią składają) wielką literą — dokładnie tak jak w powyższych przykładach⁴.

⁴ Jest to zatem standard Lower Camel Case. Nie jest to jednak zapis obligatoryjny, można oczywiście stosować inne standardy.

Typy odnośnikowe

Typy odnośnikowe (ang. *reference types*) możemy podzielić na dwa umowne rodzaje:

- ❑ typy klasowe (ang. *class types*)⁵,
- ❑ typy tablicowe (ang. *array types*).

Zacznijmy od typów tablicowych. Tablice są to wektory elementów danego typu i służą do uporządkowanego przechowywania wartości tego typu. Mogą być jedno- bądź wielowymiarowe. Dostęp do danego elementu tablicy jest realizowany przez podanie jego indeksu, czyli miejsca w tablicy, w którym się on znajduje. Dla tablicy jednowymiarowej będzie to po prostu kolejny numer elementu, dla tablicy dwuwymiarowej trzeba już podać zarówno numer wiersza, jak i kolumny itd. Jeśli chcemy zatem przechować w programie 10 liczb całkowitych, najwygodniej będzie użyć w tym celu 10-elementowej tablicy typu `int`. Tablice zostaną bliżej omówione w rozdziale 3.

Typy klasowe pozwalają na tworzenie klas i deklarowanie zmiennych obiektowych. Zajmiemy się nimi w rozdziale 4.

Deklarowanie zmiennych typów odnośnikowych

Zmienne typów odnośnikowych deklarujemy podobnie jak w przypadku zmiennych typów podstawowych, tzn. pisząc:

```
typ_zmiennej nazwa_zmiennej;
```

lub:

```
typ_zmiennej nazwa_zmiennej_1, nazwa_zmiennej_2, nazwa_zmiennej_3;
```

Stosując taki zapis, inaczej niż w przypadku typów prostych, zadeklarowaliśmy jednak jedynie tzw. *odniesienie* (ang. *reference*) do obiektu, a nie sam byt, jakim jest obiekt! Takiemu odniesieniu przypisana jest domyślnie wartość pusta (`null`), czyli praktycznie nie możemy wykonywać na nim żadnej operacji. Dopiero po utworzeniu odpowiedniego obiektu w pamięci możemy powiązać go z tak zadeklarowaną zmienną. Jeśli zatem napiszemy np.:

```
int a;
```

⁵ Typy klasowe moglibyśmy podzielić z kolei na obiektowe i interfejsowe; są to jednak rozważania, którymi nie będziemy się w niniejszej publikacji zajmować.

będziemy mieli gotową do użycia zmienną typu całkowitego. Możemy jej przypisać np. wartość 10. Żeby jednak móc skorzystać z tablicy, musimy zadeklarować zmienną odnośnikową typu tablicowego, utworzyć obiekt tablicy i powiązać go ze zmienną. Dopiero wtedy będziemy mogli swobodnie odwoływać się do kolejnych elementów. Pisząc zatem:

```
int tablica[];
```

zadeklarujemy odniesienie do tablicy, która będzie mogła zawierać elementy typu `int`, czyli 32-bitowe liczby całkowite. Samej tablicy jednak jeszcze wcale nie ma. Przekonamy się o tym, wykonując kolejne ćwiczenia (osoby nieznające pojęcia tablicy mogą pominąć tę część materiału i powrócić do niej po zapoznaniu się z rozdziałem 3.).

Ć W I C Z E N I E

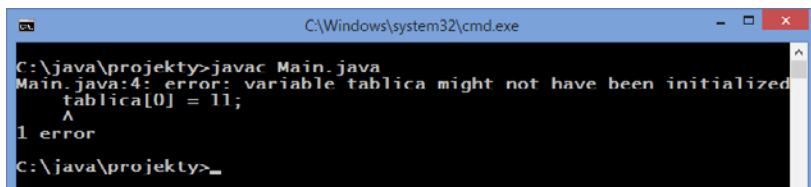
2.4 Deklarowanie tablicy

Zadeklaruj tablicę elementów typu całkowitego. Przypisz zerowemu elementowi tablicy dowolną wartość. Spróbuj skompilować i uruchomić program.

```
class Main {
    public static void main (String args[]) {
        int tablica[];
        tablica[0] = 11;
        System.out.println ("Zerowy element tablicy to: " + tablica[0]);
    }
}
```

Już przy próbie kompilacji kompilator wypisze na ekranie tekst: `variable tablica might not have been initialized`, informujący nas, że chcemy odwołać się do zmiennej, która prawdopodobnie nie została zainicjalizowana (rysunek 2.2). Widzimy też wyraźnie, że w razie wystąpienia błędu na etapie kompilacji otrzymujemy kilka ważnych i pomocnych informacji. Przede wszystkim jest to nazwa pliku, w którym wystąpił błąd (to ważne, gdyż program może składać się z bardzo wielu plików), numer wiersza w tym pliku oraz konkretne miejsce wystąpienia błędu. Na samym końcu kompilator podaje też całkowitą liczbę błędów.

Skoro jednak wystąpił błąd, należy go natychmiast naprawić. W tym przypadku oznacza to utworzenie tablicy i przypisanie jej zmiennej `tablica`.



Rysunek 2.2. Błąd kompilacji spowodowany niezainicjowaniem zmiennej *tablica*

ĆWICZENIE

2.5 Deklaracja i utworzenie tablicy

Zadeklaruj i utwórz tablicę elementów typu całkowitego. Przypisz zerowemu elementowi tablicy dowolną wartość. Spróbuj wyświetlić zawartość tego elementu na ekranie.

```
class Main {  
    public static void main (String args[]) {  
        int tablica[] = new int[10];  
        tablica[0] = 11;  
        System.out.println ("Zerowy element tablicy to: " + tablica[0]);  
    }  
}
```

Wyrażenie `new int[10]` oznacza utworzenie nowej, jednowymiarowej tablicy liczb typu `int` o rozmiarze 10 elementów. Ta nowa tablica została przypisana zmiennej odnośnikowej o nazwie `tablica`. Po takim przypisaniu możemy odwoływać się do kolejnych elementów tej tablicy, pisząc:

```
tablica[index]
```

Zapis `tablica[0] = 11;` oznacza więc: przypisz wartość 11 elementowi tablicy o indeksie 0.

Warto zwrócić szczególną uwagę, że elementy tablicy są numerowane od zera, a nie od 1. Oznacza to, że pierwszy element tablicy 10-elementowej ma indeks 0, a ostatni 9 (a nie 10!).

Co się jednak stanie, jeśli — nieprzyzwyczajeni do takiego sposobu indeksowania — odwołamy się do indeksu o numerze 10? Sprawdźmy to, wykonując ćwiczenie 2.6.

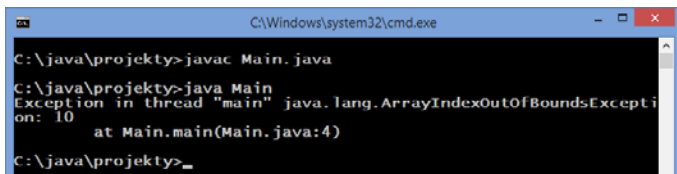
ĆWICZENIE

2.6 Odwołanie do nieistniejącego indeksu tablicy

Zadeklaruj i zainicjalizuj tablicę 10-elementową. Spróbuj przypisać elementowi o indeksie 10 dowolną liczbę całkowitą.

```
class Main {  
    public static void main (String args[]) {  
        int tablica[] = new int[10];  
        tablica[10] = 11;  
        System.out.println ("Dziesiąty element tablicy to: " + tablica[10]);  
    }  
}
```

Efekt działania kodu jest widoczny na rysunku 2.3. Wbrew pozorom nie stało się jednak nic strasznego. Wystąpił błąd, został on jednak obsłużony przez maszynę wirtualną Javy. Konkretnie został wygenerowany tzw. wyjątek i program standardowo zakończył działanie. Taki wyjątek możemy jednak przechwycić i tym samym zapobiec niekontrolowanemu zakończeniu aplikacji. Jest to jednak odrębny, aczkolwiek bardzo ważny temat — zajmiemy się nim nieco później. Godne uwagi jest to, że próba odwołania się do nieistniejącego elementu została wykryta i to odwołanie tak naprawdę nie wystąpiło! Program nie naruszył więc niezarezerwowanego dla niego obszaru pamięci.



Rysunek 2.3. Próba odwołania się do nieistniejącego elementu tablicy

Operatory

Poznaliśmy już zmienne, musimy jednak wiedzieć, jakie operacje możemy na nich wykonywać. Operacje wykonujemy za pomocą różnych operatorów, np. odejmowania, dodawania, przypisania itd. Są więc operatory⁶:

⁶ Można wydzielić również inne grupy, ale wykracza to poza ramy tematyczne niniejszej publikacji.

- ❑ arytmetyczne,
- ❑ bitowe,
- ❑ logiczne,
- ❑ przypisania,
- ❑ porównania.

Operatory arytmetyczne

Wśród tych operatorów znajdziemy standardowo działające:

- ❑ + — dodawanie,
- ❑ - — odejmowanie,
- ❑ * — mnożenie,
- ❑ / — dzielenie.

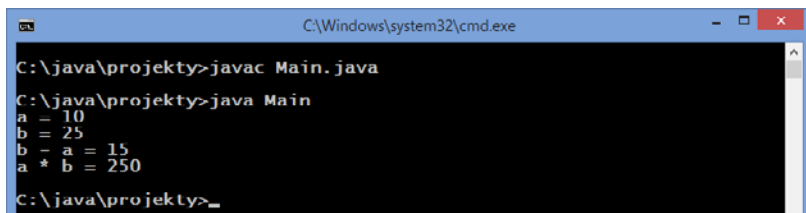
ĆWICZENIE

2.7 Operacje arytmetyczne na zmiennych

Zadeklaruj dwie zmienne typu całkowitego. Wykonaj na nich kilka operacji arytmetycznych. Wyniki wyświetl na ekranie.

```
class Main {  
    public static void main(String args[]) {  
        int a, b, c;  
        a = 10;  
        b = 25;  
        c = b - a;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("b - a = " + c);  
        c = a * b;  
        System.out.println("a * b = " + c);  
    }  
}
```

Najpierw zostały zadeklarowane trzy zmienne typu `int`: `a`, `b` i `c`. Dwie pierwsze otrzymały wartości liczbowe (10 i 25), natomiast trzecia — wartość wynikającą z odejmowania `b - a` (będzie to zatem 15). Następnie wartości zmiennych zostały wyświetlone na ekranie. W kolejnym kroku zmienna `c` otrzymała nową wartość wynikającą z mnożenia `a * b` i ta wartość (250) również została wyświetlona. Efekt działania programu został przedstawiony na rysunku 2.4.



```
C:\Windows\system32\cmd.exe
C:\java\projekty>javac Main.java
C:\java\projekty>java Main
a = 10
b = 25
b - a = 15
a * b = 250
C:\java\projekty>
```

Rysunek 2.4. Wykonywanie działań arytmetycznych na zmiennych

Do operatorów arytmetycznych należy również znak %, przy czym nie oznacza on obliczania procentów, ale dzielenie modulo (resztę z dzielenia). Np. wynik działania $12 \% 5$ wynosi 2, piątka mieści się bowiem w dwunastu dwa razy, pozostawiając resztę 2 ($5 * 2 = 10$, $10 + 2 = 12$).

Ć W I C Z E N I E

2.8 Dzielenie modulo

Zadeklaruj kilka zmiennych. Wykonaj na nich operacje dzielenia modulo. Wyniki wyświetl na ekranie.

```
class Main {
    public static void main(String args[]) {
        int a, b, c;
        a = 10;
        b = 25;
        c = b % a;
        System.out.println("b % a = " + c);
        System.out.println("a % 3 = " + a % 3);
        c = a * b;
        System.out.println("(a * b) % 120 = " + c % 120);
    }
}
```

Pierwsza część kodu jest tu taka sama jak w ćwiczeniu 2.8. Zmienną `c` została jednak przypisana wartość dzielenia modulo $b \% a$. Tym samym pierwszą jej wartością będzie 5 ($25 \% 10 = 5$, bo $2 * 10 = 20$, $20 + 5 = 25$). Wynik tego działania (wartość zmiennej `c`) został wyświetlony za pomocą znanej nam konstrukcji `System.out.println`. W kolejnej instrukcji na ekran została wyprowadzona wartość działania $a \% 3$, czyli 1 ($10 \% 3 = 1$, bo $3 * 3 = 9$, $9 + 1 = 10$). Następnie zmiennej `c` przypisano wartość mnożenia $a * b$ (tak samo jak w ćwiczeniu 2.7),

a na ekran został wyprowadzony wynik działania $c \% 20$ (czyli $(a * b) \% 120$). Wynik tego działania to 10 ($c = 250, 250 \% 120 = 10$, bo $2 * 120 = 240, 240 + 10 = 250$).

Kolejne operatory typu arytmetycznego to operator inkrementacji i dekrementacji. Operator inkrementacji (czyli zwiększania), którego symbolem jest ++, powoduje przyrost wartości zmiennej o jeden. Może występować w formie przyrostkowej (postinkrementacyjnej) bądź przedrostkowej (preinkrementacyjnej). Oznacza to, że jeśli mamy zmienną, która nazywa się np. x, forma przedrostkowa będzie wyglądać: ++x, natomiast przyrostkowa: x++.

Oba te wyrażenia zwiększą wartość zmiennej x o jeden, jednak nie są one równoważne. Otóż operacja x++ zwiększa wartość zmiennej po jej wykorzystaniu, natomiast ++x przed jej wykorzystaniem. Czasem takie rozróżnienie jest bardzo pomocne przy pisaniu programu.

Ć W I C Z E N I E

2.9 Operator inkrementacji

Przeanalizuj poniższy kod. Nie uruchamiaj programu, ale zastanów się, jaki ciąg liczb będzie wyświetlony. Następnie, już po uruchomieniu kodu, sprawdź swoje przypuszczenia.

```
class Main {
    public static void main (String args[]) {
        /*1*/ int x = 1, y;
        /*2*/ System.out.println (++x);
        /*3*/ System.out.println (x++);
        /*4*/ System.out.println (x);
        /*5*/ y = x++;
        /*6*/ System.out.println (y);
        /*7*/ y = ++x;
        /*8*/ System.out.println (++y);
    }
}
```

Dla ułatwienia poszczególne wiersze w programie zostały oznaczone kolejnymi liczbami (w tym celu użyto komentarzy blokowych złożonych ze znaków /* i */). Wynikiem działania tego programu będzie ciąg liczb 2, 2, 3, 3, 6. Dlaczego? Na początku zmienna x przyjmuje wartość 1. W drugim wierszu metody main występuje operator ++x, zatem najpierw jest ona zwiększana o jeden ($x = 2$), a dopiero potem

wyświetlana na ekranie. W wierszu o numerze 3 jest odwrotnie. Najpierw wartość zmiennej x jest wyświetlana ($x = 2$), a dopiero potem zwiększana o 1 ($x = 3$). W wierszu czwartym po prostu wyświetlamy wartość x ($x = 3$). W wierszu piątym najpierw zmiennej y jest przypisywana dotychczasowa wartość x ($x = 3, y = 3$), a następnie wartość x jest zwiększana o jeden ($x = 4$). W wierszu szóstym wyświetlamy wartość y ($y = 3$). W wierszu siódmym najpierw zwiększamy wartość x o jeden ($x = 5$), a później przypisujemy tę wartość zmiennej y ($y = 5$). W wierszu ostatnim, ósmym, zwiększamy y o jeden ($y = 6$) i wyświetlamy na ekranie.

Operator dekrementacji ($--$) działa analogicznie, z tym że zamiast zwiększać wartości zmiennych, zmniejsza je, oczywiście zawsze o jeden.

ĆWICZENIE

2.10 Operator dekrementacji

Zmień kod z ćwiczenia 2.9 tak, aby operator $++$ został zastąpiony operatorem $--$. Następnie przeanalizuj jego działanie i sprawdź, czy otrzymany wynik jest taki sam jak otrzymany na ekranie po uruchomieniu kodu.

```
class Main {
    public static void main (String args[]) {
        /*1*/ int x = 1, y;
        /*2*/ System.out.println (--x);
        /*3*/ System.out.println (x--);
        /*4*/ System.out.println (x);
        /*5*/ y = x--;
        /*6*/ System.out.println (y);
        /*7*/ y = --x;
        /*8*/ System.out.println (--y);
    }
}
```

Kod działa tu analogicznie do przedstawionego w ćwiczeniu 2.9, z tą różnicą, że wartość zmiennych jest zmniejszana. W związku z tym po uruchomieniu programu zostanie uzyskany ciąg liczb: 0, 0, -1, -1, -4.

Działania operatorów arytmetycznych na liczbach całkowitych nie trzeba chyba wyjaśniać, no może z dwoma wyjątkami. Otóż, co się stanie, jeżeli wynik dzielenia dwóch liczb całkowitych nie będzie

liczbą całkowitą? Odpowiedź na szczęście jest prosta — zostanie utracona część ułamkowa wyniku. Zatem wynikiem działania $7 / 2$ w arytmetyce liczb całkowitych będzie 3 („prawdziwym” wynikiem jest oczywiście 3,5, która to wartość zostaje zaokrąglona w dół do najbliższej liczby całkowitej, czyli trzech).

Ć W I C Z E N I E

2.11 Dzielenie liczb całkowitych

Wykonaj dzielenie zmiennych typu całkowitego. Sprawdź rezultaty w sytuacji, gdy rzeczywisty wynik jest ułamkiem.

```
class Main {
    public static void main(String args[]) {
        int a = 8, b = 3, c = 2;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("a / b = " + a / b);
        System.out.println("a / c = " + a / c);
        System.out.println("b / c = " + b / c);
    }
}
```

Zostały zadeklarowane i jednocześnie zainicjowane trzy zmienne typu `int`, a ich wartości wypisane na ekranie: `a = 8`, `b = 3` i `c = 2`. Następnie zostały wyświetlone wyniki dzielenia całkowitoliczbowego `a / b`, czyli 2 („prawdziwym” wynikiem jest 2 i dwie trzecie), `a / c`, czyli 4, i `b / c`, czyli 1 („prawdziwym” wynikiem jest 1,5).

Drugim problemem jest to, co się stanie, jeżeli przekroczymy zakres jakiejś zmiennej. Pamiętamy np., że zmienna typu `byte` jest zapisywana na 8 bitach i może przyjmować wartości od -128 do 127 (patrz tabela 2.1). Spróbujmy zatem przypisać zmiennej tego typu wartość 128 .

Ć W I C Z E N I E

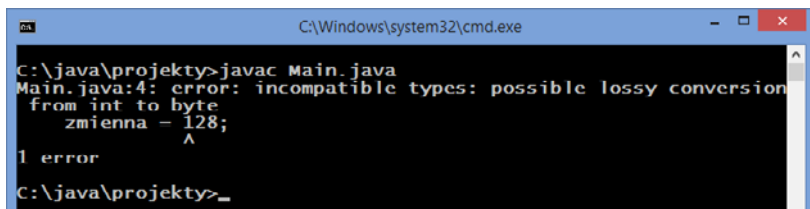
2.12 Przekroczenie zakresu w trakcie kompilacji

Zadeklaruj zmienną typu `byte`. Przypisz jej wartość 128 . Spróbuj wykonać kompilację otrzymanego kodu.

```
class Main {
    public static void main (String args[]) {
        byte zmienna;
```

```
    zmienna = 128;  
    System.out.println(zmienna);  
}  
}
```

Próbując wykonać kompilację kodu z ćwiczenia, szybko się przekonamy, że nie będzie to możliwe. Kompilator nie dopuści do przypisania zmiennej wartości wykraczającej poza zakres jej typu. Zobaczmy zatem komunikat o błędzie przedstawiony na rysunku 2.5.



Rysunek 2.5. Próba przekroczenia dopuszczalnej wartości zmiennej

Niestety, kompilator nie zawsze będzie w stanie wykryć tego typu błąd. Może się bowiem zdarzyć, że zakres przekroczymy w trakcie wykonywania programu. Co wtedy?

Ć W I C Z E N I E

2.13 Przekroczenie zakresu w trakcie działania kodu

Zadeklaruj zmienną typu `int`. Wykonaj operacje arytmetyczne przekraczające dopuszczalną wartość takiej zmiennej. Wynik wyświetl na ekranie.

```
class Main {  
    public static void main (String args[]) {  
        int wartosc = 2147483647;  
        int wartosc1 = wartosc + 1;  
        int wartosc2 = wartosc + wartosc;  
        System.out.println ("wartosc = " + wartosc);  
        System.out.println ("wartosc + 1 = " + wartosc1);  
        System.out.println ("wartosc + wartosc = " + wartosc2);  
    }  
}
```

Zmiennej `wartosc` została maksymalna wartość dla typu `int` (2147483647). Następnie została do niej dodana wartość 1, a wynik trafił do zmiennej pomocniczej `wartosc1`. W kolejnym kroku zmiennej `wartosc2` przypi-

sano wynik dodawania $\text{wartosc} + \text{wartosc}$, a na końcu wartości wszystkich zmiennych zostały wyświetlone na ekranie. Jaki zatem będzie wynik? Otóż po dodaniu 1 do wartości 2147483647 otrzymaliśmy -2147483648, natomiast w wyniku dodawania $2147483647 + 2147483647$ powstała wartość -2 (rysunek 2.6). Dlaczego? Otóż, jeżeli jakaś wartość przekracza dopuszczalny zakres swojego typu, jest „zawijana” do początku tego zakresu. Obrazowo zilustrowano to na rysunku 2.7.

Rysunek 2.6.

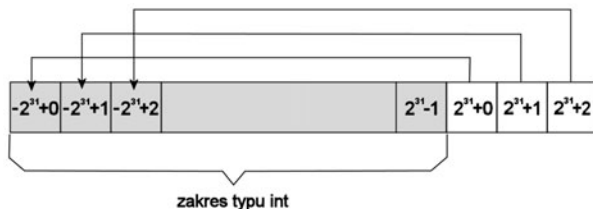
Wyniki operacji
przekraczających
dopuszczalne
wartości

```

C:\Windows\system32
C:\java\projekty>java Main
wartosc = 2147483647
wartosc + 1 = -2147483648
wartosc + wartosc = -2
C:\java\projekty>_
  
```

Rysunek 2.7.

Przekroczenie
dopuszczalnego
zakresu
dla typu int



Operatory bitowe

Operacje te, jak sama nazwa wskazuje, są dokonywane na bitach. Przypomnijmy zatem podstawowe wiadomości o systemach liczbowych. W systemie dziesiętnym, z którego korzystamy na co dzień, wykorzystywanych jest dziesięć cyfr — od 0 do 9, w systemie szesnastkowym dodatkowo litery od A do F, a w systemie ósemkowym cyfry od 0 do 7. W systemie dwójkowym będą zatem wykorzystywane jedynie dwie cyfry — 0 i 1. Kolejne liczby budowane są z odpowiednich symboli dokładnie tak samo jak w systemie dziesiętnym; zostało to przedstawione w tabeli 2.3. Widać wyraźnie, że np. 4 dziesiętnie to 100 dwójkowo, a 10 dziesiętnie to 1010 dwójkowo.

Tabela 2.3. Reprezentacja liczb w różnych systemach liczbowych

system dwójkowy	system ósemkowy	system dziesiętny	system szesnastkowy
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Na liczbach możemy dokonywać znanych ze szkoły operacji bitowych AND (iloczyn bitowy), OR (suma bitowa), NOT (negacja bitowa) oraz XOR (bitowa alternatywa wykluczająca). Symbolem operatora AND jest znak & (ampersand), operatora OR znak | (pionowa kreska), operatora NOT znak ~ (tylda), natomiast operatora XOR znak ^ (daszek, kareta). Oprócz tego można również wykonywać operacje przesunięć bitów. Zestawienie występujących w Javie operatorów bitowych zostało przedstawione w tabeli 2.4.

Tabela 2.4. Operatory bitowe w Javie

Operator	Symbol
AND	&
OR	
NOT	~
XOR	^
Przesunięcie bitowe w prawo	>>
Przesunięcie bitowe w lewo	<<
Przesunięcie bitowe w prawo z wypełnieniem zerami	>>>

Operatory logiczne

Argumentami operacji logicznych muszą być wyrażenia posiadające wartość logiczną, czyli true lub false (prawda i fałsz). Przykładowo wyrażenie `10 < 20` jest niewątpliwie prawdziwe (10 jest mniejsze od 20), zatem jego wartość logiczna jest równa true. W grupie tej wyróżniamy trzy operatory:

- ❑ logiczne AND (`&&`),
- ❑ logiczne OR (`||`),
- ❑ logiczne NOT, negacja (`!`).

Warto zauważyć, że w części przypadków stosowania operacji logicznych aby otrzymać wynik, wystarczy obliczyć tylko pierwszy argument⁷. Wynika to oczywiście z właściwości operatorów. Jeśli wynikiem obliczenia pierwszego argumentu jest bowiem wartość true, a wykonujemy operację OR, to niezależnie od stanu drugiego argumentu wartością całego wyrażenia będzie true. Podobnie przy stosowaniu operatora AND — jeżeli wartością pierwszego argumentu będzie false, to wartością całego wyrażenia również będzie false.

⁷ Ścisłej rzecz ujmując, może wystarczyć znajomość wartości jednego dowolnego argumentu.

Operatory przypisania

Operacje przypisania są dwuargumentowe i powodują przypisanie wartości argumentu znajdującego się z prawej strony do argumentu znajdującego się z lewej strony. Najprostszym operatorem tego typu jest oczywiście klasyczny znak równości. Zapis `liczba = 5` oznacza, że zmiennej `liczba` chcemy przypisać wartość 5. Oprócz tego mamy jeszcze do dyspozycji operatory łączące klasyczne przypisanie z innym operatorem arytmetycznym bądź bitowym. Zostały one zebrane w tabeli 2.5.

Tabela 2.5. Operatory przypisania i ich znaczenie w Javie

Argument 1	Operator	Argument 2	Znaczenie
x	=	y	x = y
x	+=	y	x = x + y
x	-=	y	x = x - y
x	*=	y	x = x * y
x	/=	y	x = x / y
x	%=	y	x = x % y
x	<<=	y	x = x << y
x	>>=	y	x = x >> y
x	>>>=	y	x = x >>> y
x	&=	y	x = x & y
x	=	y	x = x y
x	^=	y	x = x ^ y

Operatory porównania (relacyjne)

Operatory porównania, czyli relacyjne, służą oczywiście do porównywania argumentów. Wynikiem takiego porównania jest wartość logiczna `true` (jeśli jest ono prawdziwe) lub `false` (jeśli jest fałszywe). Zatem wynikiem operacji `argument1 == argument2` będzie `true`, jeżeli argumenty są sobie równe, lub `false`, jeżeli argumenty są różne. Czyli

4 == 5 ma wartość false, a 2 == 2 ma wartość true. Do dyspozycji mamy operatory porównania zawarte w tabeli 2.6.

Tabela 2.6. Operatory porównania w Javie

Operator	Opis
==	Jeśli argumenty są sobie równe, wynikiem jest true, w przeciwnym razie wynikiem jest false.
!=	Jeśli argumenty są różne, wynikiem jest true, w przeciwnym razie wynikiem jest false.
>	Jeśli argument prawostronny jest mniejszy od lewostronnego, wynikiem jest true, w przeciwnym razie wynikiem jest false.
<	Jeśli argument prawostronny jest większy od lewostronnego, wynikiem jest true, w przeciwnym razie wynikiem jest false.
>=	Jeśli argument prawostronny jest mniejszy od lewostronnego lub równy lewostronnemu, wynikiem jest true, w przeciwnym razie wynikiem jest false.
<=	Jeśli argument prawostronny jest większy od lewostronnego lub równy lewostronnemu, wynikiem jest true, w przeciwnym razie wynikiem jest false.

Operator warunkowy

Operator warunkowy ma następującą składnię:

```
warunek ? wartość1 : wartość2;
```

Wyrażenie takie przybiera *wartość1*, jeżeli warunek jest prawdziwy, lub *wartość2* w przeciwnym razie.

Ć W I C Z E N I E

2.14 Wykorzystanie operatora warunkowego

Użyj operatora warunkowego do zmodyfikowania wartości dowolnej zmiennej typu całkowitego (int).

```
class Main {  
    public static void main (String args[]) {  
        int x = 1, y;
```

```
y = (x == 1) ? 10 : 20;
System.out.println ("y = " + y);
}
}
```

W powyższym ćwiczeniu najważniejszy jest oczywiście wiersz:

```
y = (x == 1) ? 10 : 20;
```

który oznacza: przypisz zmiennej *y* wartość wyrażenia warunkowego. Jaka jest ta wartość? Jeżeli *x* jest równe 1, będzie to 10, w przeciwnym razie — 20. Ponieważ zmienną *x* zainicjalizowaliśmy wartością 1, wartością wyrażenia (która trafi do zmiennej *y*) będzie 10. A zatem na ekranie zostanie wyświetlony ciąg znaków *y* = 10. Użyty w instrukcji nawias okrągły nie jest konieczny i służy wyłącznie zwiększeniu czytelności kodu źródłowego. Równie poprawna byłaby instrukcja w postaci: *y* = *x* == 1 ? 10 : 20;.

Priorytety operatorów

Sama znajomość operatorów to jednak nie wszystko. Niezbędna jest jeszcze wiedza na temat tego, jaki mają one priorytet, czyli jaka jest kolejność ich wykonywania. Wiadomo na przykład, że mnożenie jest „silniejsze” od dodawania, zatem najpierw mnożymy, potem dodajemy. W Javie jest podobnie, siła każdego operatora jest ściśle określona. Przedstawiono to w tabeli 2.7⁸. Im wyższa pozycja w tabeli, tym wyższy priorytet operatora. Operatory znajdujące się na jednym poziomie (w jednym wierszu) mają ten sam priorytet.

Tabela 2.7. Priorytety operatorów w Javie

Grupa operatorów	Symbole
inkrementacja przyrostkowa	++, --
inkrementacja przedrostkowa, negacje	++, --, ~, !
mnożenie, dzielenie	*, /, %
przesunięcia bitowe	<<, >>, >>>

⁸ Tabela nie zawiera wszystkich operatorów występujących w Javie, a jedynie te omawiane w książce.

Tabela 2.7. Priorytety operatorów w Javie — ciąg dalszy

Grupa operatorów	Symbole
porównania	<, >, <=, >=
porównania	==, !=
bitowe AND	&
bitowe XOR	^
bitowe OR	
logiczne AND	&&
logiczne OR	
warunkowe	?
przypisania	=, +=, -=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, =

Instrukcje

Instrukcja warunkowa if...else

Bardzo często w programie niezbędne jest sprawdzenie jakiegoś warunku i w zależności od tego, czy jest on prawdziwy, czy fałszywy, wykonanie różnych instrukcji. Do takiego sprawdzania służy właśnie instrukcja warunkowa `if...else`. Ma ona ogólną postać:

```
if (wyrażenie warunkowe){  
    // instrukcje do wykonania, jeżeli warunek jest prawdziwy  
}  
else{  
    // instrukcje do wykonania, jeżeli warunek jest fałszywy  
}
```

Trzeba przy tym zaznaczyć, że *wyrażenie warunkowe*, inaczej niż w C i C++, musi dać w wyniku wartość typu boolean, tzn. `true` lub `false`.

Ć W I C Z E N I E

2.15 Użycie instrukcji warunkowej if...else

Wykorzystaj instrukcję warunkową if...else do stwierdzenia, czy wartość zmiennej typu całkowitego jest mniejsza od zera.

```
class Main {
    public static void main (String args[]) {
        int a = -10;
        if (a > 0){
            System.out.println ("Zmienna a jest większa od zera.");
        }
        else{
            System.out.println ("Zmienna a nie jest większa od zera.");
        }
    }
}
```

W instrukcji warunkowej if jest badany warunek $a > 0$, czyli czy wartość zmiennej a jest większa od 0. Jeśli warunek jest prawdziwy (wartość zmiennej a jest większa od 0), wykonywana jest instrukcja z bloku if, w przeciwnym razie (wartość zmiennej a nie jest większa od 0) wykonywana jest instrukcja z bloku else. Ponieważ początkowo do a została przypisana wartość -10, po skompilowaniu i uruchomieniu przykładu na ekranie pojawi się napis *Zmienna a nie jest większa od zera*. Zmiana wartości a na większą od 0 i ponowna kompilacja oraz uruchomienie spowoduje wyświetlenie drugiego napisu.

Spróbujmy teraz czegoś nieco bardziej skomplikowanego. Zajmijmy się klasycznym przykładem liczenia pierwiastków równania kwadratowego. Przypomnijmy, że jeśli mamy równanie o postaci $Ax^2 + Bx + C = 0$, to aby obliczyć jego rozwiązanie, liczymy tzw. deltę (Δ), która jest równa $B^2 - 4AC$. Jeżeli delta jest większa od zera, mamy dwa pierwiastki: $x_1 = \frac{-B + \sqrt{\Delta}}{2A}$ i $x_2 = \frac{-B - \sqrt{\Delta}}{2A}$. Jeżeli delta jest

równa zero, istnieje tylko jedno rozwiązanie — mianowicie $x = \frac{-B}{2A}$.

W przypadku trzecim, jeżeli delta jest mniejsza od zera, równanie takie nie ma rozwiązań w zbiorze liczb rzeczywistych.

Skoro jest tutaj tyle warunków do sprawdzenia, jest to doskonały przykład do potrenowania zastosowania instrukcji `if...else`. Aby nie komplikować zagadnienia, nie będziemy się w tej chwili zajmować wczytywaniem parametrów równania z klawiatury (ten temat zostanie omówiony w rozdziale 6.), ale podamy je bezpośrednio w kodzie.

Ć W I C Z E N I E

2.16 Pierwiastki równania kwadratowego

Wykorzystaj operacje arytmetyczne oraz instrukcję `if...else` do obliczenia pierwiastków równania kwadratowego o parametrach podanych bezpośrednio w kodzie programu.

```
class Main {
    public static void main (String args[]) {
        int parametrA = 1, parametrB = -1, parametrC = -6;

        System.out.println ("Parametry równania:");
        System.out.println ("A: " + parametrA + " B: " + parametrB +
            " C: " + parametrC);

        if (parametrA == 0){
            System.out.println ("To nie jest równanie kwadratowe: A = 0!");
        }
        else{
            double delta = parametrB * parametrB - 4 * parametrA * parametrC;
            if (delta < 0){
                System.out.println ("Delta < 0.");
                System.out.println
                    ("Brak rozwiązań w zbiorze liczb rzeczywistych");
            }
            else{
                double wynik;
                if (delta == 0){
                    wynik = - parametrB / (2 * parametrA);
                    System.out.println ("Rozwiązanie: x = " + wynik);
                }
                else{
                    wynik = (- parametrB + Math.sqrt(delta)) / (2 * parametrA);
                    System.out.print ("Rozwiązanie: x1 = " + wynik);
                    wynik = (- parametrB - Math.sqrt(delta)) / (2 * parametrA);
                    System.out.println (" , x2 = " + wynik);
                }
            }
        }
    }
}
```

Na początku kodu definiowane są zmienne określające parametry równania, a ich wartości wyświetlane są na ekranie. Następnie za pomocą pierwszej instrukcji warunkowej `if` badany jest stan parametru `A`, czyli zmiennej `parametrA`. Po stwierdzeniu, że parametr `A` jest równy 0, wyświetlana jest informacja, iż nie mamy do czynienia z równaniem kwadratowym. W przeciwnym razie wykonywane są instrukcje z bloku `else`. Jest w nim obliczana delta, a następnie wykonywana kolejna instrukcja `if` badająca, czy delta jest mniejsza od 0. Jeśli jest mniejsza, wyświetlana jest informacja o braku rozwiązań. W przeciwnym razie (delta większa od 0 bądź równa 0) w bloku `else` wykonywana jest kolejna instrukcja warunkowa badająca tym razem, czy delta jest równa 0. Gdy jest równa 0, jedyny wynik jest obliczany i wyświetlany na ekranie. W przeciwnym razie (delta większa od 0) obliczane są dwa pierwiastki, a wynik obliczeń wyświetlany jest na ekranie.

Jak łatwo zauważyć, instrukcję warunkową można zagnieżdżać, tzn. po jednym `if` może występować kolejne, po nim następne itd. Jednak jeżeli zapiszemy to w sposób podany w poprzednim ćwiczeniu, przy wielu zagnieżdżeniach otrzymamy bardzo nieczytelny kod. Aby temu zapobiec, można posłużyć się złożoną instrukcją warunkową `if...else if`. Zamiast tworzyć mniej wygodną konstrukcję, taką jak przedstawiona poniżej:

```
if (warunek1){  
    // instrukcje 1  
}  
else{  
    if (warunek2){  
        // instrukcje 2  
    }  
    else{  
        if (warunek3){  
            // instrukcje 3  
        }  
        else{  
            // instrukcje 4  
        }  
    }  
}
```

całość można zapisać dużo prościej i czytelniej w postaci:

```
if (warunek1){  
    //instrukcje 1  
}  
else if (warunek2){  
    //instrukcje 2  
}  
else if (warunek3){  
    //instrukcje 3  
}  
else{  
    //instrukcje 4  
}
```

Ć W I C Z E N I E

2.17 Zastosowanie instrukcji if...else if

Napisz kod obliczający pierwiastki równania kwadratowego o parametrach zadanych w programie. Wykorzystaj instrukcję if...else if.

```
class Main {  
    public static void main (String args[]) {  
        int parametrA = 1, parametrB = -1, parametrC = -6;  
  
        System.out.println ("Parametry równania:");  
        System.out.println  
            ("A: " + parametrA + " B: " + parametrB + " C: " + parametrC);  
  
        if (parametrA == 0){  
            System.out.println ("To nie jest równanie kwadratowe: A = 0!");  
        }  
        else{  
            double delta = parametrB * parametrB - 4 * parametrA * parametrC;  
            double wynik;  
  
            if (delta < 0){  
                System.out.println ("Delta < 0.");  
                System.out.println  
                    ("Brak rozwiązań w zbiorze liczb rzeczywistych");  
            }  
            else if (delta == 0){  
                wynik = - parametrB / (2 * parametrA);  
                System.out.println ("Rozwiązanie: x = " + wynik);  
            }  
            else{  
                wynik = (- parametrB + Math.sqrt(delta)) / (2 * parametrA);  
                System.out.print ("Rozwiązanie: x1 = " + wynik);  
            }  
        }  
    }  
}
```

```
        wynik = (- parametrB - Math.sqrt(delta)) / (2 * parametrA);
        System.out.println (" x2 = " + wynik);
    }
}
}
```

Instrukcja wyboru switch

Instrukcja switch pozwala w wygodny i przejrzysty sposób sprawdzać wiele warunków i wykonywać różny kod w zależności od tego, czy warunki są prawdziwe, czy fałszywe. Można nią zastąpić ciąg instrukcji if...else if. Jeżeli w kodzie mamy przykładową konstrukcję o schematycznej postaci:

```
if (wyrażenie == X){
    // instrukcja 1;
}
else if (wyrażenie == Y){
    // instrukcja2;
}
else if (wyrażenie == Z){
    // instrukcja3;
}
else{
    // instrukcja4;
}
```

to możemy ją zastąpić następująco:

```
switch (wyrażenie){
    case X:
        // instrukcja1;
        break;
    case Y:
        // instrukcja2;
        break;
    case Z:
        // instrukcja3;
        break;
    default:
        // instrukcja4;
}
```


Po kolei jest tu sprawdzane, czy wyrażenie przyjmuje jedną z wartości x , y i z . Jeżeli równość zostanie w jednym z przypadków stwierdzona, wykonywane są instrukcje po odpowiedniej klauzuli `case`. Jeżeli równość nie zostanie stwierdzona, wykonywane są instrukcje po słowie `default`. Instrukcja `break` powoduje wyjście z bloku `switch`. Wyrażeniem jest z reguły po prostu nazwa zmiennej. W wersjach Javy do 7 (1.7) wartościami wyrażenia (symbole x , y , z w kodzie) mogą być tylko wartości typu `char`, `byte`, `short` i `int`, natomiast w Javie 7 dodatkowo uwzględniany jest też typ `String` (czyli ciąg znaków).

ĆWICZENIE

2.18 Użycie instrukcji wyboru `switch`

Używając instrukcji `switch`, napisz program sprawdzający, czy wartość zadeklarowanej zmiennej jest równa 1, czy 10. Wyświetl na ekranie stosowny komunikat.

```
class Main {
    public static void main (String args[]) {
        int a = 10;
        switch (a){
            case 1 :
                System.out.println("a jest równe 1.");
                break;
            case 10:
                System.out.println("a jest równe 10.");
                break;
            default:
                System.out.println("a nie jest równe ani 1, ani 10.");
        }
    }
}
```



Jeżeli w którymś z przypadków `case` zapomnimy o słowie (instrukcji) `break`, wykonywanie instrukcji `switch` będzie kontynuowane aż do osiągnięcia pierwszej instrukcji `break` lub wykonania całego bloku `switch`. To może prowadzić do otrzymania niespodziewanych efektów. W szczególności zostanie wtedy wykonany blok instrukcji występujący po `default`. Może to być oczywiście efekt zamierzony, może też jednak powodować trudne do wykrycia błędy.

ĆWICZENIE

2.19 Efekt pominięcia instrukcji break

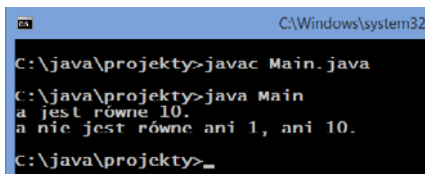
Zmodyfikuj kod z ćwiczenia 2.18, usuwając instrukcję `break`. Zaobserwuj, jak zmieniło się działanie programu.

```
class Main {
    public static void main (String args[]) {
        int a = 10;
        switch (a){
            case 1:
                System.out.println("a jest równe 1.");
            case 10:
                System.out.println("a jest równe 10.");
            default:
                System.out.println("a nie jest równe ani 1, ani 10.");
        }
    }
}
```

Po skompilowaniu i uruchomieniu kodu zobaczymy widok zaprezentowany na rysunku 2.8. Jak widać, według naszego programu zmienna `a` jest jednocześnie równa 10, jak i różna od 10. Bezpośrednim tego powodem jest oczywiście usunięcie słowa `break`. Instrukcja `switch` działa teraz w ten sposób, że najpierw wartość `a` jest porównywana z wartością 1 (pierwszy przypadek `case`). Równość nie jest stwierdzona, zatem następuje przejście do kolejnego bloku `case` (`case 10`). W tym bloku równość zostanie stwierdzona (`a = 10`), jest więc wykonywana instrukcja z tego bloku (`System.out.println("a = 10");`). Ponieważ jednak w bloku `case 10` nie ma instrukcji `break`, zostaną wykonane wszystkie instrukcje z innych bloków `case` znajdujących się za `case 10` (o ile takie występują) oraz z bloku `default`, chyba że w którymś z nich wystąpi jednak instrukcja `break`. W tym przypadku za `case 10` mamy tylko blok `default` i żadnej instrukcji `break`, zatem zostanie wykonana instrukcja `System.out.println("a nie jest równe ani 1, ani 10.");`. Warto samodzielnie sprawdzić, co się stanie w programie po przypisaniu zmiennej `a` na samym początku wartości 1 (zamiast 10).

Rysunek 2.8.

*Ilustracja błędu
z ćwiczenia 2.19*



```
C:\Windows\system32
C:\java\projekty>javac Main.java
C:\java\projekty>java Main
a jest równe 10.
a nie jest równe ani 1, ani 10.
C:\java\projekty>
```

Pętla for

Pętle w językach programowania pozwalają na wykonywanie powtarzających się czynności. Nie inaczej jest w Javie. Jeśli chcemy np. wypisać na ekranie 10 razy napis Java, możemy zrobić to, pisząc 10 razy `System.out.println("Java");`. Jeżeli jednak chcielibyśmy mieć już 150 takich napisów, to, pomijając oczywiście kwestię sensowności tej czynności, byłby to już problem. Na szczęście z pomocą przychodzi nam właśnie pętla. Pętla typu `for` ma następującą składnię:

```
for (wyrażenie początkowe; wyrażenie warunkowe; wyrażenie modyfikujące){  
    // instrukcje do wykonania  
}
```

wyrażenie początkowe jest stosowane do zainicjalizowania zmiennej używanej jako licznik liczby wykonanych pętli, *wyrażenie warunkowe* określa warunek, który musi być spełniony, aby dokonać kolejnego przejścia w pętli, *wyrażenie modyfikujące* jest zwykle używane do modyfikacji zmiennej będącej licznikiem.

Ć W I C Z E N I E

2.20 Budowa pętli for

Wykorzystując pętlę typu `for`, napisz program wyświetlający na ekranie 10 razy napis Java.

```
class Main {  
    public static void main (String args[]) {  
        for (int i = 1; i <= 10; i++){  
            System.out.println ("Java");  
        }  
    }  
}
```

Zmienna `i` to tzw. **zmienna iteracyjna**, której na początku przypisujemy wartość 1 (`int i = 1`). Następnie w każdym przebiegu (czyli iteracji) pętli jest ona zwiększana o jeden (`i++`) oraz wykonywana jest instrukcja `System.out.println ("Java");`. Wszystko to trwa, aż `i` osiągnie wartość 10 (`i <= 10`).

Wyrażenie modyfikujące jest zwykle używane do modyfikacji zmiennej iteracyjnej. Takiej modyfikacji możemy jednak dokonać również wewnątrz pętli. Struktura tego typu wygląda następująco:

```
for (wyrażenie początkowe; wyrażenie warunkowe;){  
    // instrukcje do wykonania  
    wyrażenie modyfikujące  
}
```

Ć W I C Z E N I E

2.21 Wyrażenie modyfikujące w bloku instrukcji

Zmodyfikuj pętlę typu for z ćwiczenia 2.20, tak aby wyrażenie modyfikujące znalazło się w bloku instrukcji.

```
class Main {  
    public static void main (String args[]) {  
        for (int i = 1; i <= 10;){  
            System.out.println ("Java");  
            i++;  
        }  
    }  
}
```

Zwróćmy uwagę, że mimo iż wyrażenie modyfikujące jest teraz wewnątrz pętli, średnik znajdujący się po `i <= 10` jest niezbędny! Jeśli o nim zapomnimy, kompilator zgłosi błąd.

Kolejną ciekawą możliwością jest połączenie wyrażenia warunkowego i modyfikującego.

Ć W I C Z E N I E

2.22 Łączenie wyrażenia warunkowego i modyfikującego

Napisz taką pętlę typu for, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym.

```
class Main {  
    public static void main (String args[]) {  
        for (int i = 1; i++ <= 10;){  
            System.out.println ("Java");  
        }  
    }  
}
```

Tu warto wspomnieć, że obecny przykład nie jest w pełni funkcjonalnym odpowiednikiem przykładów z ćwiczeń 2.20 i 2.21 (zmienna i wewnątrz pętli zmienia się w innym zakresie).

W podobny sposób jak w poprzednich przykładach możemy się pozbyć wyrażenia początkowego, które przeniesiemy przed pętlę. Schemat wygląda następująco:

```
wyrażenie początkowe;  
for (; wyrażenie warunkowe;){  
    //instrukcje do wykonania  
    wyrażenie modyfikujące  
}
```

Ć W I C Z E N I E

2.23 Wyrażenie początkowe przed pętlą

Zmodyfikuj pętlę typu for w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, a wyrażenie modyfikujące wewnątrz niej.

```
class Main {  
    public static void main (String args[]) {  
        int i = 1;  
        for (; i <= 10;){  
            System.out.println ("Java");  
            i++;  
        }  
    }  
}
```

Skoro zaszliśmy już tak daleko w pozbywaniu się wyrażeń sterujących, usuńmy również wyrażenie warunkowe. Jest to jak najbardziej możliwe!

Ć W I C Z E N I E

2.24 Pętla bez wyrażeń

Zmodyfikuj pętlę typu for w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, natomiast wyrażenia modyfikujące i warunkowe wewnątrz pętli.

```
class Main {  
    public static void main (String args[]) {  
        int i = 1;  
        for ( ; ; ){  
            System.out.println ("Java");  
            if (i++ >= 10) break;  
        }  
    }  
}
```

Przy stosowaniu tego typu konstrukcji pamiętajmy, że do prawidłowego funkcjonowania kodu niezbędne są oba średniki w nawiasie okrągłym występującym po `for`. Warto też zwrócić uwagę na zmianę kierunku nierówności. We wcześniejszych przykładach sprawdzaliśmy bowiem, czy `i` jest mniejsze od 10 bądź równe 10, a teraz — czy jest większe bądź równe. Dzieje się tak, ponieważ poprzednio sprawdzaliśmy, czy pętla ma być dalej wykonywana, natomiast obecnie — czy ma zostać zakończona. Wykorzystaliśmy też instrukcję `break`. Służy ona do natychmiastowego przerywania wykonywania pętli.

Przydatną instrukcją jest `continue`. Powoduje ona rozpoczęcie kolejnej iteracji (przebiegu) pętli, tzn. w miejscu jej wystąpienia wykonywanie bieżącej iteracji jest przerywane i rozpoczyna się kolejny przebieg.

ĆWICZENIE

2.25 Zastosowanie instrukcji `continue`

Napisz program wyświetlający na ekranie liczby od 1 do 20, które nie są podzielne przez 2. Skorzystaj z pętli `for` i instrukcji `continue`.

```
class Main {
    public static void main (String args[]) {
        for (int i = 1; i <= 20; i++){
            if (i % 2 == 0){
                continue;
            }
            System.out.println (i);
        }
    }
}
```

Przypomnijmy, że `%` to operator dzielenia modulo, tzn. daje on resztę z dzielenia. To oznacza, że jeśli wynikiem działania `i % 2` jest 0, to wartość `i` z pewnością jest parzysta (podzielna przez 2). Jest to badane w instrukcji warunkowej `if`. Zatem gdy warunek `i % 2 == 0` jest prawdziwy (`i` jest podzielne przez 2), wykonywana jest instrukcja `continue`. Tym samym natychmiast rozpoczyna się kolejna iteracja pętli i nie jest wykonywana instrukcja `System.out.println (i);`. Gdy warunek jest fałszywy (`i` nie jest podzielne przez 2), instrukcja wyświetlająca wartość `i` jest wykonywana, dzięki czemu na ekranie pojawiają się nieparzyste liczby z zakresu 1 – 20.

Oczywiście nic nie stoi na przeszkodzie, aby aplikację działającą w taki sam sposób jak w ćwiczeniu 2.25 napisać bez użycia instrukcji `continue`.

ĆWICZENIE

2.26 Liczby niepodzielne przez dwa

Zmodyfikuj kod z ćwiczenia 2.25 tak, aby nie było konieczności użycia instrukcji `continue`.

```
class Main {
    public static void main (String args[]) {
        for (int i = 1; i <= 20; i++){
            if (i % 2 != 0){
                System.out.println (i);
            }
        }
    }
}
```

Tym razem instrukcja wyświetlająca wartość `i` jest wykonywana tylko wtedy, gdy prawdziwy jest warunek `i % 2 != 0`, a więc wtedy, gdy reszta z dzielenia `i` przez 2 jest różna od zera (co oznacza, że wartość `i` jest nieparzysta). Dzięki takiej konstrukcji został osiągnięty taki sam efekt jak w ćwiczeniu 2.25, ale bez użycia instrukcji `continue`.

Pętla while

O ile pętla typu `for` służy zwykle do wykonywania liczby operacji znanej przed rozpoczęciem pętli (np. zapisanej w zmiennej), to w przypadku pętli `while` liczba ta nie jest zazwyczaj znana (np. wynika z wartości zwróconej przez funkcję). Nie jest to oczywiście obligatoryjny podział. Tak naprawdę obie można napisać w taki sposób, aby były swoimi funkcjonalnymi odpowiednikami. Ogólna konstrukcja pętli typu `while` jest następująca:

```
while (wyrażenie warunkowe){
    // instrukcje
}
```

Instrukcje są wykonywane dopóty, dopóki wyrażenie warunkowe jest prawdziwe. Oznacza to, że gdzieś w pętli musi wystąpić modyfikacja warunku bądź też instrukcja `break`. Inaczej będzie się wykonywała w nieskończoność!

Ć W I C Z E N I E**2.27 Budowa pętli while**

Używając pętli typu `while`, napisz program wyświetlający na ekranie 10 razy napis Java.

```
class Main {  
    public static void main (String args[]) {  
        int i = 1;  
        while (i <= 10){  
            System.out.println ("Java");  
            i++;  
        }  
    }  
}
```

Taka pętla działa, dopóki prawdziwy jest warunek `i <= 10`. Ponieważ początkową wartością `i` jest 1, a we wnętrzu pętli wartość tej zmiennej w każdym przebiegu jest zwiększana o 1, pętla (a tym samym instrukcja `System.out.println ("Java");`) zostanie wykonana dokładnie 10 razy.

Podobnie jak w przypadku pętli `for`, mamy dużą dowolność w kształtowaniu pętli `while`. Przykładowo wyrażenie modyfikujące wartość zmiennej kontrolującej przebiegi pętli może być z powodzeniem połączone z wyrażeniem warunkowym.

Ć W I C Z E N I E**2.28 Połączone wyrażenia**

Kod z ćwiczenia 2.27 zmodyfikuj tak, aby wyrażenie warunkowe zmieniało jednocześnie wartość zmiennej `i`.

```
class Main {  
    public static void main (String args[]) {  
        int i = 1;  
        while (i++ <= 10){  
            System.out.println ("Java");  
        }  
    }  
}
```


W tym miejscu należy zwrócić uwagę, że co prawda pętla daje taki sam efekt jak ta w ćwiczeniu 2.27, ale nie są one swoimi dokładnymi odpowiednikami. Tym razem zmienna i zmienia się w innym zakresie.

Ć W I C Z E N I E

2.29 Liczby nieparzyste i pętla while

Korzystając z pętli while, napisz program wyświetlający na ekranie liczby od 1 do 20 niepodzielne przez 2.

```
class Main {
    public static void main (String args[]) {
        int i = 1;
        while (i <= 20){
            if (i % 2 != 0)
                System.out.println (i);
            i++;
        }
    }
}
```

Pętla do...while

Istnieje jeszcze jedna odmiana pętli. Jest to do...while. Jej konstrukcja jest następująca:

```
do{
    // instrukcje;
}
while (warunek);
```

Oznacza to: wykonuj *instrukcje*, dopóki prawdziwy jest *warunek*.

Ć W I C Z E N I E

2.30 Budowa pętli do...while

Korzystając z pętli do...while, napisz program wyświetlający na ekranie 10 razy dowolny napis.

```
class Main {
    public static void main (String args[]) {
        int i = 1;
        do{
            System.out.println ("Java");
        }
```

```
    }  
    while (i++ <= 9);  
  }  
}
```

Choć w pierwszej chwili wydawać by się mogło, że to przecież to samo co zwykła pętla `while`, jest jednak pewna znacząca różnica. Otóż w przypadku pętli `do...while` warunek zakończenia jest sprawdzany dopiero po jej wykonaniu (w pętli `while` warunek jest sprawdzany przed wykonaniem instrukcji z wnętrza). To oznacza, że instrukcje są wykonane co najmniej jeden raz, nawet jeśli warunek jest na pewno fałszywy.

Ć W I C Z E N I E

2.31 Pętla `do...while` z fałszywym warunkiem

Zmodyfikuj kod z ćwiczenia 2.30 w taki sposób, aby wyrażenie warunkowe na pewno było fałszywe. Zaobserwuj wyniki działania programu.

```
class Main {  
    public static void main (String args[]) {  
        int i = 10;  
        do{  
            System.out.println ("Java");  
        }  
        while (i++ <= 9);  
    }  
}
```

W tym programie warunek zakończenia pętli (`i++ <= 9`) z pewnością jest fałszywy, przed wykonaniem pętli zmiennej `i` została bowiem przypisana wartość 10 (która oczywiście nie jest ani mniejsza od 9, ani równa 9, a więc warunek `i++ <= 9` z pewnością jest fałszywy). Mimo tego, gdy skompilujemy i uruchomimy program, na ekranie pojawi się jeden napis `Java`. To najlepszy dowód, że warunek został sprawdzony po wykonaniu instrukcji z wnętrza pętli, a nie przed ich wykonaniem.

Rozszerzona pętla for

Począwszy od wersji 1.5 (5.0), Java udostępnia jeszcze jeden rodzaj pętli. Jest ona nazywana pętlą *foreach* lub rozszerzoną pętlą *for* (ang. *enhanced for*) i pozwala na automatyczną iterację po kolekcji obiektów lub też po tablicy. Zostanie przedstawiona w kolejnym rozdziale, omawiającym tablice.

3

Tablice

Tworzenie tablic

Tablice to jedno z najprostszych struktur danych. Umożliwiają przechowanie uporządkowanego zbioru elementów danego typu. Tablicę można sobie wyobrazić tak, jak przedstawiono to na rysunku 3.1. Składa się ona z ponumerowanych kolejno komórek, przy czym należy wyraźnie zaznaczyć, że numeracja zaczyna się od 0. Czyli pierwsza komórka ma indeks 0. Każda komórka może przechowywać pewną porcję danych. Jakiego rodzaju będą to dane, określa typ tablicy. Jeśli zatem zadeklarujemy tablicę typu całkowitoliczbowego (int), będzie ona mogła zawierać liczby całkowite. Jeżeli będzie to natomiast typ znakowy (char), poszczególne komórki będą mogły zawierać różne znaki.

Rysunek 3.1.
*Schemat
struktury tablicy*



Zmienną typu tablicowego tworzy się podobnie jak zmienną typu prostego (taką jak `int`, `char` itp.), należy jednak do deklaracji dodać nawias kwadratowy przed nazwą lub po nazwie. Schematycznie wygląda to tak (pierwszy przypadek):

```
typ_tablicy nazwa_tablicy[];
```

lub tak (drugi przypadek):

```
typ_tablicy[] nazwa_tablicy;
```

Dawniej bardziej popularna była wersja z umieszczaniem nawiasu po nazwie zmiennej, obecnie coraz częściej stosuje się wersję drugą. Taka deklaracja nie spowoduje jednak utworzenia samej tablicy. Ponieważ w Javie tablice są obiektami (to pojęcie zostanie przedstawione w rozdziale 4.), konieczne jest użycie dodatkowej konstrukcji ze słowem `new`, schematycznie:

```
new typ_tablicy[liczba_elementów];
```

A zatem pełna instrukcja tworząca zmienną tablicową i przypisująca jej nowo powstałą tablicę będzie miała postać:

```
typ_tablicy nazwa_tablicy[] = new typ_tablicy[liczba_elementów];
```

lub:

```
typ_tablicy[] nazwa_tablicy = new typ_tablicy[liczba_elementów];
```

ĆWICZENIE

3.1 Tworzenie tablicy

Zadeklaruj i utwórz tablicę trójelementową dla liczb typu `int`.

```
class Main {  
    public static void main (String args[]) {  
        int tablica[] = new int[3];  
    }  
}
```

Tworzenie zmiennej tablicowej oraz tablicy może też być rozdzielone i odbywać się w różnych miejscach programu. Wtedy schemat postępowania byłby następujący:

```
typ_tablicy nazwa_tablicy[];  
/* tutaj mogą znaleźć się inne instrukcje */  
nazwa_tablicy = new typ_tablicy[liczba_elementów];
```

Ć W I C Z E N I E

3.2 Rozdzielenie deklaracji od utworzenia tablicy

Zadeklaruj zmienną tablicową. W osobnym miejscu programu utwórz odpowiadającą jej tablicę.

```
class Main {
    public static void main (String args[]) {
        int tablica[];
        // tutaj mogą wystąpić inne instrukcje programu
        tablica = new int[3];
        // powyższa instrukcja spowodowała powstanie tablicy
        // trójelementowej oraz przypisanie jej zmiennej tablica
    }
}
```

W obu powyższych ćwiczeniach powstała tablica trójelementowa, która może przechowywać liczby typu `int`. Była jednak zawsze pusta, tzn. nie zawierała żadnych wartości. Sposoby wypełniania tablicy danymi zostaną podane w podrozdziale „Zapis i odczyt elementów”. Istnieje jednak sposób na jednoczesne utworzenie tablicy i przypisanie jej poszczególnym komórkom wybranych wartości. Taka konstrukcja ma ogólną postać:

```
typ_tablicy nazwa_tablicy[] = {wartość1, wartość2, ... , wartośćn}
```

W takim przypadku nie podajemy rozmiaru tablicy, gdyż jest on bezpośrednio określony przez liczbę elementów wymienionych w nawiasie klamrowym.

Ć W I C Z E N I E

3.3 Jednoczesna deklaracja i inicjacja tablicy

Użyj jednej instrukcji do utworzenia tablicy i wypełnienia jej danymi.

```
class Main {
    public static void main (String args[]) {
        int tablica[] = {1, 8, 15, 2};
    }
}
```

Użyta instrukcja spowoduje powstanie 4-elementowej tablicy zawierającej w kolejnych komórkach liczby 1, 8, 15 oraz 2 i przypisanie jej zmiennej o nazwie `tablica`.

Zapis i odczyt elementów

W tablicach można zapisywać różne wartości, oczywiście możliwy jest też ich odczyt. Aby uzyskać dostęp do wartości zapisanej w danej komórce, należy podać jej numer (indeks) w nawiasie kwadratowym występującym za nazwą tablicy. Trzeba przy tym pamiętać, że indeksowanie tablicy zaczyna się od 0, co oznacza, że indeksem pierwszej komórki jest 0, a nie 1. Odczytajmy zatem dane z tablicy, takiej jak w ćwiczeniu 3.3, i wyświetlmy je na ekranie.

Ć W I C Z E N I E

3.4 Odczyt danych z tablicy

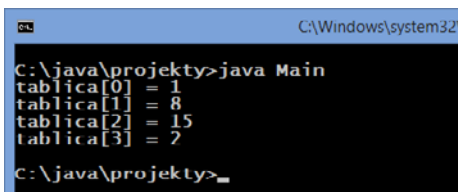
Do kodu z ćwiczenia 3.3 dodaj instrukcje odczytujące dane z tablicy i wyświetlające jej zawartość na ekranie.

```
class Main {  
    public static void main (String args[]) {  
        int tablica[] = {1, 8, 15, 2};  
        System.out.println("tablica[0] = " + tablica[0]);  
        System.out.println("tablica[1] = " + tablica[1]);  
        System.out.println("tablica[2] = " + tablica[2]);  
        System.out.println("tablica[3] = " + tablica[3]);  
    }  
}
```

Do wyświetlenia danych zapisanych w tablicy została użyta zwykła instrukcja `System.out.println`. Każde wywołanie powoduje pobranie wartości wskazanej komórki (o indeksach 0, 1, 2 i 3) i wyprowadzenie jej na ekran. Dzięki temu po uruchomieniu programu zobaczymy widok przedstawiony na rysunku 3.2.

Rysunek 3.2.

*Odczyt zawartości
tablicy*



```
C:\Windows\system32\  
C:\java\projekty>java Main  
tablica[0] = 1  
tablica[1] = 8  
tablica[2] = 15  
tablica[3] = 2  
C:\java\projekty>_
```

Może pojawić się pytanie: co zawiera tablica, która nie została zainicjowana ani wypełniona żadnymi danymi? Czyli co znajduje się w komórkach tablicy utworzonej w sposób przedstawiony w ćwiczeniu

3.1 lub 3.2. Odpowiedź jest prosta. Taka tablica jest pusta, czyli zawiera wartości domyślne dla danego typu. W przypadku typów arytmetycznych zawsze jest to wartość 0. Można to sprawdzić za pomocą kodu z ćwiczenia 3.5.

Ć W I C Z E N I E

3.5 Badanie pustej tablicy

Utwórz dwie 2-elementowe tablice. Jedna ma przechowywać wartości typu `int`, a druga — wartości typu `double`. Nie wypełniaj ich danymi. Sprawdź za to, jaka jest zawartość poszczególnych komórek.

```
class Main {  
    public static void main (String args[]) {  
        int tablica1[] = new int[2];  
        double tablica2[] = new double[2];  
        System.out.println("tablica1[0] = " + tablica1[0]);  
        System.out.println("tablica1[1] = " + tablica1[1]);  
        System.out.println("tablica2[0] = " + tablica2[0]);  
        System.out.println("tablica2[1] = " + tablica2[1]);  
    }  
}
```

Komórki tablicy mogą być wielokrotnie zapisywane i odczytywane. Przy czym każdy kolejny zapis w danej komórce kasuje jej poprzednią zawartość. Do zapisu stosujemy zwykły operator przypisania `=`. Konstrukcja w postaci:

```
tablica[indeks] = wartość;
```

oznacza: przypisz *wartość* komórce tablicy *tablica* wskazywanej przez *indeks*. Oczywiście przypisywana wartość musi być takiego typu, jakiego jest tablica (lub musi istnieć możliwość automatycznej konwersji między tymi typami, np. można przypisać wartość typu `short` do komórki tablicy typu `int`).

Ć W I C Z E N I E

3.6 Modyfikacja zawartości tablicy

Zapisz w tablicy dowolne wartości, a następnie zmodyfikuj je.

```
class Main {  
    public static void main (String args[]) {  
        int tablica[] = new int[2];
```

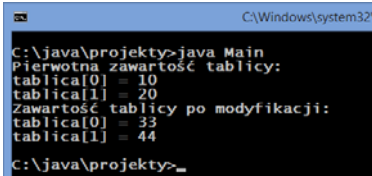
```
    tablica[0] = 10;
    tablica[1] = 20;
    System.out.println("Pierwotna zawartość tablicy:");
    System.out.println("tablica[0] = " + tablica[0]);
    System.out.println("tablica[1] = " + tablica[1]);

    tablica[0] = 33;
    tablica[1] = 44;
    System.out.println("Zawartość tablicy po modyfikacji:");
    System.out.println("tablica[0] = " + tablica[0]);
    System.out.println("tablica[1] = " + tablica[1]);
}
}
```

Najpierw powstała tablica liczb typu `int` o dwóch elementach. Do jej utworzenia została użyta konstrukcja przedstawiona w ćwiczeniu 3.1. Następnie obie komórki otrzymały przykładowe wartości całkowite (10 i 20), a zawartość tablicy została wyświetlona na ekranie. W dalszej części kodu wartości obu komórek zmodyfikowano. Pojawiły się w nich wartości 33 i 44, które również zostały wyświetlone, co jest widoczne na rysunku 3.3.

Rysunek 3.3.

*Odczyt
zawartości
tablicy przed
modyfikacją
i po niej*



```
C:\Windows\system32\cmd.exe
C:\java\projekty>java Main
Pierwotna zawartość tablicy:
tablica[0] = 10
tablica[1] = 20
Zawartość tablicy po modyfikacji:
tablica[0] = 33
tablica[1] = 44
C:\java\projekty>
```

Operacje z użyciem pętli

Do wypełniania tablic danymi oraz do ich odczytu najczęściej używa się pętli (przedstawionych w rozdziale 2.). Jest to wręcz niezbędne, gdyż trudno się spodziewać, aby można było „ręcznie” odczytać wartości z więcej niż kilkunastu czy kilkudziesięciu komórek. Wielkość tablicy nie musi też być z góry znana, może wynikać z danych używanych w trakcie działania programu. Z tablicami mogą współpracować dowolne rodzaje pętli. W niektórych przypadkach bardzo wygodna jest wspomniana na końcu rozdziału 2. pętla `foreach`. Zaczniemy jednak od zwykłej pętli `for`.

Ć W I C Z E N I E

3.7 Użycie pętli for do odczytu i zapisu tablicy

Użyj pętli for do zapisania w 10-elementowej tablicy 10 kolejnych liczb całkowitych. Następnie odczytaj zawartość i wyświetl ją na ekranie.

```
class Main {
    public static void main (String args[]) {
        int tablica[] = new int[10];

        for(int i = 0; i < 10; i++){
            tablica[i] = i + 1;
        }
        System.out.println("Zawartość tablicy:");
        for(int i = 0; i < 10; i++){
            System.out.println(tablica[i]);
        }
    }
}
```

Powstała 10-elementowa tablica liczb typu int. Skoro ma 10 elementów, to pierwszy jest indeks 0, a ostatni — 9. Dlatego zmienna sterująca pętli for, która wypełnia tablicę danymi, ma początkową wartość 0, a warunek zakończenia pętli to $i < 10$. Tym samym i zmienia się też od 0 do 9. W tablicy mamy jednak zapisać wartości od 1 do 10. Czyli komórka o indeksie 0 ma mieć wartość 1, o indeksie 1 — wartość 2 itd. A zatem wartość komórki ma być zawsze o 1 większa niż wartość indeksu (zmiennnej i). Dlatego instrukcja wewnątrz pętli ma postać:

```
tablica[i] = i + 1;
```

Druga pętla for służy tylko do wyświetlania danych zawartych w tablicy. Jej konstrukcja jest taka sama jak w pierwszym przypadku. Wewnątrz pętli znajduje się instrukcja wyświetlająca wartości kolejnych komórek.

W ćwiczeniu 3.7 wyświetlona została tylko zawartość tablicy bez informacji o indeksach poszczególnych komórek (inaczej niż w przykładach niekorzystających z pętli). Dzięki temu druga pętla miała bardzo czytelną postać. Jeśli jednak indeksy również miałyby się pojawić, napis wyprowadzany na ekran musiałby być składany z kilku członów, tak aby uwzględniał wartość zmiennej sterującej i . Zostało to pokazane w ćwiczeniu 3.8.

ĆWICZENIE

3.8 Wyświetlanie wartości indeksów

Zmień drugą pętlę for z ćwiczenia 3.7, tak aby oprócz zawartości komórek tablicy wyświetlane były również ich indeksy.

```
for(int i = 0; i < 10; i++){
    System.out.println("tablica[" + i + "] = " + tablica[i]);
}
```

ĆWICZENIE

3.9 Liczby w porządku malejącym

Użyj pętli for do wypełnienia tablicy kolejnymi liczbami od 20 do 1. Gotową zawartość tablicy wyświetl na ekranie.

```
class Main {
    public static void main (String args[]) {
        int tablica[] = new int[20];

        for(int i = 0; i < 20; i++){
            tablica[i] = 20 - i;
        }
        System.out.println("Zawartość tablicy:");
        for(int i = 0; i < 20; i++){
            System.out.println("tablica[" + i + "] = " + tablica[i]);
        }
    }
}
```

Kod tego ćwiczenia jest bardzo podobny do rozwiązań ćwiczeń 3.7 i 3.8. Tym razem tworzona jest tablica 20-elementowa, a więc w obu zastosowanych pętlach for zmienna iteracyjna i zmienia się od 0 (indeks pierwszej komórki) do 19 (indeks ostatniej komórki). Ponieważ w kolejnych komórkach mają się znaleźć wartości malejące od 20 do 1, wartości te są uzyskiwane w pierwszej pętli za pomocą działania $20 - i$.

Zamiast pętli for (która zwykle jest wygodniejsza) do obsługi tablicy można też użyć pętli while. Należy przy tym zwracać uwagę na warunek zakończenia, a także na stan zmiennej iteracyjnej, które nie są tak intuicyjne jak w przypadku pętli for.

ĆWICZENIE

3.10 Tablice i pętla while

Wykonaj zadanie z ćwiczenia 3.7, używając pętli typu while.

```
class Main {
    public static void main (String args[]) {
        int tablica[] = new int[10];
        int i = -1;
        while(i++ < 9){
            tablica[i] = i + 1;
        }
        System.out.println("Zawartość tablicy:");
        i = -1;
        while(i++ < 9){
            System.out.println("tablica[" + i + "] = " + tablica[i]);
        }
    }
}
```

Tablica została utworzona w standardowy sposób. Następnie powstała zmienna *i* o pierwotnej wartości -1 (będzie używana jako zmienna iteracyjna kontrolująca przebiegi pętli). Musi to być -1, gdyż wyrażenie warunkowe badające warunek zakończenia pętli (*i*++ < 9) jest jednocześnie wyrażeniem modyfikującym wartość zmiennej (zwiększa ją o 1). A zatem najpierw badany jest warunek *i* < 9, następnie, gdy *i* jest mniejsze od 9, jest zwiększane o 1, a dopiero potem jest wykonywana instrukcja z wnętrza pętli. Dlatego *i* wewnątrz pętli zmienia się w zakresie od 0 do 9. Gdyby pierwotną wartością *i* było 0, zmienna ta wewnątrz pętli zmieniałaby się od 1 do 10. Oczywiście można przygotować kod również w taki sposób, aby przy inicjacji *i* otrzymało wartość 0. Wystarczy zmodyfikować pętlę następująco:

```
int i = 0;
while(i < 10){
    tablica[i] = i + 1;
    i++;
}
```

W prosty sposób można by też wtedy uniknąć konieczności wykonywania operacji *i* + 1. To jednak pozostanie jako ćwiczenie do samodzielnego wykonania.

Do odczytu danych z tablic można użyć również rozszerzonej pętli *for* (nazywanej też pętlą *foreach*), której opis został pominięty w roz-

dziale 2. Jeśli mamy tablicę¹ zawierającą wartości pewnego typu, to do przejrzenia wszystkich jej elementów możemy użyć konstrukcji o postaci:

```
for(typ nazwa: tablica){  
    // instrukcje  
}
```

W takim przypadku w kolejnych przebiegach pętli `for` pod nazwa będzie podstawiana wartość kolejnego elementu.

Ć W I C Z E N I E

3.11 Wykorzystanie rozszerzonej pętli `for`

Zadeklaruj tablicę liczb typu `int` i wypełnij ją przykładowymi danymi. Następnie użyj rozszerzonej pętli `for` do wyświetlenia zawartości tablicy na ekranie.

```
class Main {  
    public static void main (String args[]) {  
        int tab[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
        System.out.println("Zawartość tablicy:");  
        for(int val: tab){  
            System.out.println(val);  
        }  
    }  
}
```

Tablica `tab` została zainicjowana kolejnymi liczbami od 1 do 10. W każdym przebiegu pętli `for` pod zmienną `val` jest podstawiana wartość kolejnego elementu tablicy. W pierwszym przebiegu jest to pierwszy element (o indeksie 0), w drugim — drugi element (o indeksie 1) itd. Pętla kończy się po osiągnięciu ostatniego elementu (o indeksie 9).

Rozmiar tablicy

Każda tablica posiada właściwość² `length`, która określa bieżącą liczbę komórek. Aby uzyskać tę informację, piszemy:

```
tablica.length
```

¹ A dokładniej: dowolny obiekt posiadający iterator. To jest jednak zagadnienie bardziej zaawansowane, które nie będzie opisywane w książce.

² Znaczenie tego terminu zostanie dokładniej omówione w rozdziale 4.

Przy tym dopuszczalny jest tylko odczyt. Czyli prawidłowa jest na przykład konstrukcja:

```
int rozmiar = tablica.length;
```

ale nieprawidłowy jest zapis:

```
tablica.length = 10;
```

Ta właściwość jest bardzo przydatna, gdyż (choć taka sytuacja nie ma miejsca w prezentowanych przykładach) rozmiar tablicy nie zawsze jest z góry znany.

Ć W I C Z E N I E

3.12 Odczyt rozmiaru tablicy

Utwórz tablicę o dowolnym rozmiarze. Odczytaj wartość właściwości `length` i wyświetl ją na ekranie.

```
class Main {
    public static void main (String args[]) {
        int tab[] = {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
            10, 9, 8, 7, 6, 5, 4, 3, 2, 1
        };
        System.out.println("Wielkość tablicy: " + tab.length);
    }
}
```

Ć W I C Z E N I E

3.13 Właściwość `length` i pętla `for`

Utwórz tablicę zawierającą pewną ilość liczb całkowitych. Zawartość tablicy wyświetl na ekranie za pomocą pętli `for`. Do określenia rozmiaru tablicy użyj właściwości `length`.

```
class Main {
    public static void main (String args[]) {
        int tab[] = {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
            10, 9, 8, 7, 6, 5, 4, 3, 2, 1
        };
        for(int i = 0; i < tab.length; i++){
            System.out.println("tab[" + i + "] = " + tab[i]);
        }
    }
}
```

Zasada odczytu danych w tym przykładzie jest taka sama jak w ćwiczeniu 3.7, z tą jedynie różnicą, że rozmiar tablicy jest określany za pomocą właściwości `length` (`tab.length`). Dzięki temu można np. dopisać dowolną ilość nowych danych w instrukcji inicjującej tablicę, a kod pętli `for` nie będzie wymagał żadnych zmian. Nowy rozmiar zostanie uwzględniony automatycznie.

Ć W I C Z E N I E

3.14 Właściwość `length` i pętla `while`

Wykonaj zadanie z ćwiczenia 3.13, używając pętli `while`.

```
class Main {
    public static void main (String args[]) {
        int tab[] = {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
            10, 9, 8, 7, 6, 5, 4, 3, 2, 1
        };
        int i = 0;
        while(i < tab.length){
            System.out.println("tab[" + i + "] = " + tab[i]);
            i++;
        }
    }
}
```

Zmienną sterującą przebiegiem pętli jest `i` — jej początkowa wartość to 0. Warunkiem zakończenia pętli jest `i < tab.length`, przy czym `i` jest zwiększane dopiero w ostatniej instrukcji. A zatem wartość zmiennej sterującej zmienia się od 0 do maksymalnego rozmiaru tablicy minus 1 (w momencie, gdy `i` będzie równe `tab.length`, nastąpi opuszczenie pętli). Zwiększanie zmiennej `i` może się też odbywać już w instrukcji wyświetlającej dane na ekranie, co jednak zmniejszyłoby czytelność kodu. Wtedy we wnętrzu pętli znajdowałby się tylko jeden wiersz kodu w postaci:

```
System.out.println("tab[" + i + "] = " + tab[i++]);
```

Oczywiście można też zastosować pomysł przedstawiony w ćwiczeniu 3.10 i modyfikować wartość `i` już w wyrażeniu warunkowym, odpowiednio dostosowując instrukcję wyświetlającą dane.

4

Obiekty i klasy

Każdy program w Javie składa się z klas. W naszych dotychczasowych przykładach była to tylko jedna klasa o nazwie `Main`. Widać więc, że klasa może zawierać kod wykonujący różne zadania. W rzeczywistości są to jednak opisy obiektów, a każdy obiekt jest instancją, czyli wystąpieniem danej klasy. Oznacza to, że klasa definiuje typ danego obiektu. Co to jest typ?

„Typ jest przypisany do zmiennej, wyrażenia lub innego bytu programistycznego (danej, obiektu, funkcji, procedury, operacji, metody, parametru, modułu, wyjątku, zdarzenia). Specyfikuje on rodzaj wartości, które może przybierać ten byt. (...) Jest to również ograniczenie kontekstu, w którym odwołanie do tego bytu może być użyte w programie”¹.

Zatem jeżeli typem zmiennej jest `int`, to może ona przyjmować wartości typu `int`, czyli liczby całkowite z danego przedziału. Nie można takiej zmiennej przypisać np. tablicy. Obrazowo można powiedzieć, że klasa to pewnego rodzaju plan, na podstawie którego w programie tworzone są obiekty; te z kolei mogą przechowywać dane i wykonywać różne zadania. Schematyczny szkielet klasy wygląda następująco:

¹ Kazimierz Subieta, *Wytwarzanie, integracja i testowanie systemów informatycznych*, PJWSTK, Warszawa 1997, s. 7.

```
class nazwa_klasy
{
    // treść klasy
}
```

W treści klasy definiujemy pola i metody. Pola służą do przechowywania danych (możemy je potraktować jak zmienne), metody — do wykonywania różnych operacji.

Żeby nie poprzestawać na suchych definicjach, które mogą wydawać się niejasne, stwórzmy jakiś konkretny przykład. Załóżmy, że chcemy w programie zapisać dane dotyczące punktów na płaszczyźnie. W tym celu powinniśmy stworzyć klasę Punkt. Każdy punkt powinien charakteryzować się dwiema współrzędnymi: x i y . Jeśli dla uproszczenia przyjmiemy, że współrzędne mogą być tylko całkowite (np. są to współrzędne pikseli na ekranie), to klasa ta powinna zawierać dwa pola typu `int`.

ĆWICZENIE

4.1 Tworzenie prostej klasy

Napisz kod klasy, w której można będzie przechowywać dane dotyczące punktów na płaszczyźnie.

```
class Punkt
{
    int x;
    int y;
}
```

Powstała tu definicja klasy o nazwie `Punkt` zawierająca dwa pola — x i y , które opisują położenie punktu na płaszczyźnie.

Gdy mamy do dyspozycji klasę, możemy zadeklarować zmienną typu tej klasy (a więc za pomocą klasy został zdefiniowany nowy typ danych). W tym przypadku byłyby to zmienna typu `Punkt`. Jest to bardzo proste:

```
Punkt nowyPunkt;
```

Jak widać, podajemy najpierw nazwę klasy, potem nazwę zmiennej, dokładnie tak samo, jak miało to miejsce w przypadku poznanych już wcześniej zmiennych typów podstawowych (`int`, `char` itp.). Z opisów podanych we wcześniejszych rozdziałach wiadomo, że w ten sposób zadeklarowaliśmy jedynie zmienną obiektową, która wska-

zuje obiekt typu `Punkt`. Inaczej mówiąc, zmienna taka jest referencją (wskazaniem, odniesieniem, z ang. *reference*) do obiektu typu `Punkt`². Aby móc skorzystać z obiektu takiego typu, trzeba go najpierw utworzyć. Używamy do tego operatora `new` w postaci:

```
new NazwaKlasy();
```

W omawianym przypadku będzie to wyglądało następująco:

```
Punkt nowyPunkt = new Punkt();
```

Można też najpierw zadeklarować referencję, a dopiero potem utworzyć obiekt i powiązać go z tą referencją:

```
Punkt nowyPunkt;  
nowyPunkt = new Punkt();
```

Warto zwrócić uwagę, że w `C++` jest inaczej! To znaczy już napisanie:

```
Punkt nowyPunkt;
```

spowodowałoby utworzenie obiektu typu `Punkt`, a nie tylko referencji do niego. Jeżeli ktoś jest przyzwyczajony do `C++`, powinien o tym pamiętać, gdyż jest to często popełniany błąd.

Obiekt, który powstał, zawiera dwa pola: `x` i `y`. Te pola często nazywamy *właściwościami obiektu*³. Mówimy zatem, że obiekt `nowyPunkt` zawiera właściwości `x` i `y`.

Metody

Metody, jak już wiemy, zawierają kod operujący na polach danej klasy bądź też na dostarczonych z zewnątrz danych (ogólniej — mogą wykonywać dowolnie zaprogramowane zadanie). Metody wywołujemy za pomocą operatora `.` (kropka), poprzedzając je nazwą zmiennej odnośnikowej. Wygląda to następująco:

```
nazwa_zmiennej.nazwa_metody(parametry metody);
```

² Często jednak zmienną obiektową (referencyjną) utożsamia się z obiektem. Takie uproszczenie będzie też czasem stosowane w książce. Zamiast mówić „zmienna `X` wskazująca obiekt typu `Y`”, można zatem powiedzieć po prostu „obiekt typu `Y`” lub nawet „obiekt `X`”. Nie jest to w pełni poprawne, ale w pewnych sytuacjach dopuszczalne.

³ To pewne uproszczenie: terminy „właściwość” i „pole” formalnie nie są tożsame.

W nawiasie okrągłym występującym po nazwie metody podajemy jej parametry (inaczej argumenty). W ten sposób możemy przekazać jej jakieś dane. W przypadku klasy `Punkt` z ćwiczenia 4.1 moglibyśmy stworzyć dwie metody, które zwracałyby współrzędną x i współrzędną y punktu.

Ć W I C Z E N I E

4.2 Pierwsze metody klasy `Punkt`

Do klasy `Punkt` dodaj metody podające współrzędną x oraz współrzędną y .

```
class Punkt
{
    int x, y;
    int pobierzX()
    {
        return x;
    }
    int pobierzY()
    {
        return y;
    }
}
```

Wewnątrz klasy oprócz pól x i y zostały umieszczone metody `pobierzX` i `pobierzY`. Pierwsza zwraca wartość pola x , a druga — wartość pola y . Wartości pól są zwracane za pomocą instrukcji `return` (instrukcja ta powoduje przerwanie wykonywania danej metody i, ewentualnie, zwrócenie przez nią wartości wymienionej po słowie `return`). Jeśli mając taką klasę, utworzymy nowy obiekt typu `Punkt` i zapiszemy go w zmiennej `punkt`:

```
Punkt punkt = new Punkt();
```

to aby uzyskać informację o wartości współrzędnej x , będziemy mogli napisać:

```
punkt.x;
```

lub też:

```
punkt.pobierzX();
```

np.:

```
int wspolrzecznaX = punkt.x;
int wspX = punkt.pobierzX();
```

Należy zwrócić uwagę, że po wywołaniu metody (zapis `punkt.pobierzX()` oznacza wywołanie metody `pobierzX` należącej do obiektu wskazywanego przez zmienną `punkt`) w miejsce wystąpienia tego wywołania jest podstawiana wartość zwrócona przez tę metodę za pomocą instrukcji `return`. To dlatego możliwe jest przypisanie `wspX = punkt.pobierzX()`.

Ć W I C Z E N I E

4.3 Ustawianie i pobieranie współrzędnych

Do klasy `Punkt` dodaj metodę ustawiającą współrzędne oraz metodę zwracającą współrzędne.

```
class Punkt
{
    int x, y;
    void ustawWspolrzedne(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    Punkt pobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
}
```

Przedstawiona klasa zawiera dwa pola typu `int` o nazwach `x` i `y`, które będą służyły do przechowywania współrzędnych danego punktu. Są w niej również dwie metody. Jedna zajmuje się ustawieniem pól danego obiektu, a druga pobiera te wartości. Metoda `ustawWspolrzedne` jest typu `void`⁴, co oznacza, że nie zwraca ona żadnej wartości. Ma natomiast dwa argumenty, `wspX` i `wspY`, oba typu `int`. W ciele (czyli wewnątrz) tej metody do pola `x` przypisywana jest wartość parametru `wspX`, a do pola `y` — wartość parametru `wspY`. Zatem po wykonaniu instrukcji (zakładając istnienie obiektu `punkt`):

```
punkt.ustawWspolrzedne (1, 10);
```

⁴ Zostało tu zastosowane często spotykane uproszczenie, w którym typ metody jest utożsamiany z typem zwracanym przez metodę. W rzeczywistości pojęcia te nie są tożsame.

pole `x` obiektu `punkt` przyjmie wartość 1, a pole `y` — wartość 10. Do pól danego obiektu odwołujemy się tak samo jak w przypadku metod, tzn. za pomocą operatora `.` (kropka).

Druga metoda — `pobierzPunkt` — zwraca odniesienie do obiektu typu `Punkt` (w uproszczeniu: obiekt typu `Punkt`). Można zatem napisać taką oto instrukcję:

```
Punkt innyPunkt = punkt.pobierzWspolrzedne();
```

W ciele tej metody najpierw jest tworzony nowy obiekt typu `Punkt`, a następnie odpowiednim polem tego obiektu są przypisywane pola `x` i `y` obiektu bieżącego. Obiekt `punkt` (dokładniej referencja do tego obiektu) jest zwracany instrukcją `return`.

Najwyższy czas wykorzystać tak stworzony kod w konkretnym przykładzie. W tym celu utworzymy dwa pliki. Jeden o nazwie (tak jak do tej pory) *Main.java* oraz drugi o nazwie *Punkt.java* (zgodnie z wcześniejszą przyjętą zasadą, że każdą klasę będziemy umieszczać w oddzielnym pliku o nazwie zgodnej z nazwą klasy⁵). Zawartością pliku *Punkt.java* będzie podany wyżej kod klasy `Punkt` (z ćwiczenia 4.3), a zawartością pliku *Main.java* — kod z ćwiczenia 4.4.

ĆWICZENIE

4.4

Wykorzystanie obiektu klasy `Punkt`

Napisz prosty program wykorzystujący obiekty klasy `Punkt`.

```
class Main {
    public static void main (String args[]) {
        Punkt punkt = new Punkt();
        Punkt pomocniczyPunkt;

        pomocniczyPunkt = punkt.pobierzWspolrzedne();

        System.out.println ("Przed ustawieniem współrzędnych:");
        System.out.println ("Współrzędna x = " + pomocniczyPunkt.x);
        System.out.println ("Współrzędna y = " + pomocniczyPunkt.y);

        punkt.ustawWspolrzedne (1, 10);
        pomocniczyPunkt = punkt.pobierzWspolrzedne();
    }
}
```

⁵ W tym przykładzie obie klasy mogłyby się znaleźć w jednym pliku, ponieważ obie są klasami pakietowymi. Ta kwestia zostanie jednak wyjaśniona dopiero w dalszej części rozdziału.

```
        System.out.println ("Po ustawieniu współrzędnych:");  
        System.out.println ("Współrzędna x = " + pomocniczyPunkt.x);  
        System.out.println ("Współrzędna y = " + pomocniczyPunkt.y);  
    }  
}
```

Oba pliki (*Main.java* oraz *Punkt.java*) umieszczamy w jednym katalogu i kompilujemy. Mamy dwie możliwości. Możemy najpierw skompilować plik *Punkt.java*, a następnie *Main.java*, albo też skompilować tylko *Main.java*. W tym drugim przypadku, ponieważ kod klasy *Main* korzysta z klasy *Punkt*, kompilacja pliku *Punkt.java* odbędzie się automatycznie.

Jak zatem działa taki program? Na początku tworzymy referencję o nazwie *punkt* i przypisujemy jej nowy obiekt klasy *Punkt* oraz drugą referencję — *pomocniczyPunkt*. Ponieważ metoda *pobierzWspolrzedne* zwraca referencję do obiektu typu *Punkt*, możemy dokonać przypisania:

```
pomocniczyPunkt = punkt.pobierzWspolrzedne();
```

Wartości pól *x* i *y* obiektu *pomocniczyPunkt* są wyświetlane na ekranie. Będą to dwa zera. Wynika to z tego, że pola te nie zostały zainicjowane, a niezainicjowane pola typu *int* przyjmują wartość 0.

Następnie użyto metody *ustawWspolrzedne*:

```
punkt.ustawWspolrzedne (1, 10);
```

Oznacza to, że polu *x* została przypisana wartość 1, a polu *y* — wartość 10. Te wartości również zostały wyświetlone na ekranie.

Analizując kod z ćwiczenia 4.4, można zauważyć, że tak naprawdę zmienna *pomocniczyPunkt* wcale nie jest potrzebna, choć jej użycie zwiększa czytelność kodu programu. Równie dobrze można by jednak dokonać odwołania bezpośredniego.

Ć W I C Z E N I E

4.5 Bezpośrednie odwołanie do pól obiektu

Zmodyfikuj kod z ćwiczenia 4.4, tak aby nie było konieczności użycia zmiennej *pomocniczyPunkt*.

```
class Main {  
    public static void main (String args[]) {  
        Punkt punkt = new Punkt();
```

```
System.out.println("Przed ustawieniem współrzędnych:");
System.out.println ("Współrzędna x = " + punkt.pobierzWspolrzedne().x);
System.out.println ("Współrzędna y = " + punkt.pobierzWspolrzedne().y);

punkt ustawWspolrzedne (1, 10);

System.out.println("Po ustawieniu współrzędnych:");
System.out.println ("Współrzędna x = " + punkt.pobierzWspolrzedne().x);
System.out.println ("Współrzędna y = " + punkt.pobierzWspolrzedne().y);
}
}
```

Ponieważ w miejsce wywołania metody `pobierzWspolrzedne` jest poddawiany wynik jej działania, czyli zwrócony obiekt, można zastosować odwołanie typu:

```
punkt.pobierzWspolrzedne().x
```

Trzeba jednak pamiętać, że w tym przykładzie znajduje się dwukrotnie więcej wywołań metody `pobierzWspolrzedne` (niż w ćwiczeniu 4.4), co nie pozostaje bez wpływu na wydajność działania kodu. Oczywiście w tak prostym przypadku nie ma to znaczenia. Jednak przy bardzo dużej liczbie wywołań można się spodziewać spadku wydajności.

Dlaczego jednak stosujemy tę na pozór karkołomną konstrukcję, tworząc najpierw w metodzie `pobierzWspolrzedne` nowy obiekt typu `Punkt`, przypisując mu odpowiednie wartości `x` i `y` i zwracając dopiero referencję do tego nowego bytu? Odpowiedź jest prosta. Nie możemy jednocześnie zwrócić współrzędnych `x` i `y`. Metoda może bowiem zwracać tylko jedną wartość — typu podstawowego lub obiektowego (odnośnikowego).

Możemy co najwyżej napisać dwie dodatkowe metody, które będą osobno zwracały wartość `x`, a osobno wartość `y`, tak jak zrobiliśmy to w ćwiczeniu 4.2. Podobnie można stworzyć dwie metody ustawiające osobno wartości `x` i `y`. Może się to okazać bardzo przydatne, gdy gdzieś w programie będziemy chcieli zmodyfikować tylko jedną ze współrzędnych. Załóżmy, że chcemy zmodyfikować tylko `x`, ustawiając jego wartość na 5. Stosując dotychczasowy kod, należałoby użyć w tym celu instrukcji:

```
punkt.ustawWspolrzedne (5, punkt.pobierzWspolrzedne().y);
```

Nowe funkcje ułatwiłyby to zadanie. Dopiszmy więc brakujące metody, niech będą to: `ustawX`, `ustawY`, `pobierzX`, `pobierzY`.

Ć W I C Z E N I E

4.6 Nowe metody klasy Punkt

Zmodyfikuj klasę Punkt, dodając metody umożliwiające niezależne ustawianie i odczytywanie współrzędnych.

```
class Punkt
{
    int x, y;
    void ustawWspolrzedne(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    Punkt pobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
    void ustawX(int wspX)
    {
        x = wspX;
    }
    void ustawY(int wspY)
    {
        y = wspY;
    }
    int pobierzX()
    {
        return x;
    }
    int pobierzY()
    {
        return y;
    }
}
```

Metody ustawWspolrzedne i pobierzWspolrzedne mają taką samą postać jak w ćwiczeniu 4.3. Metoda ustawX przyjmuje jeden argument typu int (wspX) i przypisuje jego wartość polu x. Metoda ustawY działa analogicznie, z tym że przypisuje wartość otrzymanego argumentu polu y. Metody pobierzX i pobierzY po prostu zwracają wartości odpowiednich pól za pomocą instrukcji return.

Oprócz sposobów przedstawionych w ostatnim ćwiczeniu istnieje jeszcze jedna możliwość uzyskania jednocześnie wartości x i y . Metodzie `pobierzPunkt` można bowiem przekazać argument. Wyglądałoby to następująco:

```
void pobierzWspolrzedne(Punkt punkt)
{
    punkt.x = x;
    punkt.y = y;
}
```

Wtedy po wykonaniu metody pola obiektu wskazywanego przez argument `punkt` przyjmą wartości z pól x i y obiektu bieżącego.

Zatrzymajmy się na chwilę w tym miejscu. Przecież jeżeli dopiszemy do klasy `Punkt` taką metodę, to będzie ona zawierała dwie metody o takiej samej nazwie — `pobierzWspolrzedne`. Czy jest to dopuszczalne? Otóż tak, pod jednym wszakże warunkiem: muszą się one różnić od siebie parametrami wywołania. Proces ten nazywa się przeciążaniem metod lub funkcji. W naszym przypadku jedna metoda nie ma żadnych parametrów wywołania, druga jako parametr przyjmuje referencję do obiektu typu `Punkt`. Wszystko jest zatem zgodne z zasadami.

Ć W I C Z E N I E

4.7 Przeciążanie metod

Dołącz do klasy `Punkt` przeciążoną metodę `pobierzWspolrzedne`.

```
class Punkt
{
    int x, y;
    void ustawWspolrzedne(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    Punkt pobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
    void pobierzWspolrzedne(Punkt punkt)
    {
        punkt.x = x;
        punkt.y = y;
    }
}
```

```
}  
void ustawX(int wspX)  
{  
    x = wspX;  
}  
void ustawY(int wspY)  
{  
    y = wspY;  
}  
int pobierzX()  
{  
    return x;  
}  
int pobierzY()  
{  
    return y;  
}  
}
```

Aby się przekonać, że dwie metody o tej samej nazwie faktycznie mogą się znajdować w jednej klasie i nie spowoduje to kolizji w ich działaniu, napiszemy testową klasę Main, która będzie korzystała z najnowszej wersji klasy Punkt.

Ć W I C Z E N I E

4.8 Wykorzystanie przeciążonych metod

Napisz klasę Main korzystającą z przeciążonych metod klasy Punkt.

```
class Main {  
    public static void main (String args[]) {  
        Punkt punkt = new Punkt();  
        Punkt pomocniczyPunkt = new Punkt();  
  
        punkt.pobierzWspolrzedne(pomocniczyPunkt);  
  
        System.out.print ("Współrzędna x = " + pomocniczyPunkt.x);  
        System.out.println (" Współrzędna y = " + pomocniczyPunkt.y);  
  
        punkt.ustawWspolrzedne (1, 10);  
        punkt.pobierzWspolrzedne(pomocniczyPunkt);  
  
        System.out.print ("Współrzędna x = " + pomocniczyPunkt.x);  
        System.out.println (" Współrzędna y = " + pomocniczyPunkt.y);  
    }  
}
```

Na początku powstały dwa obiekty typu Punkt: punkt i pomocniczyPunkt. Instrukcja:

```
punkt.pobierzWspolrzedne(pomocniczyPunkt);
```

spowodowała wywołanie metody pobierzWspolrzedne obiektu punkt i przekazanie jej w postaci argumentu obiektu pomocniczyPunkt. To oznacza, że bieżące wartości pól x i y obiektu punkt zostaną przypisane polom x i y obiektu pomocniczyPunkt. Metoda pobierzWspolrzedne została też ponownie wywołana po zmianie współrzędnych obiektu punkt za pomocą metody ustawWspolrzedne. W obu przypadkach bieżące wartości współrzędnych zapisanych w pomocniczyPunkt są wyświetlane na ekranie, dzięki czemu można się przekonać, że wszystko działa zgodnie z opisem.

Jak więc widać, metoda pobierzWspolrzedne przyjmująca argument działa bez problemów mimo tego, że w klasie Punkt istnieje jej wersja nieprzyjmująca argumentów i działająca w inny sposób.

W tym miejscu ponownie zwróćmy też uwagę, że mimo iż na początku nie ustawialiśmy żadnych współrzędnych punktu, to mają one wartość zero, pola obiektów w Javie są bowiem standardowo zerowane (tzn. zawierają 0, \u0000, false lub null — w zależności od typu zmiennej).

Konstruktory

Zazwyczaj polom danego obiektu chcemy przypisać jakieś wartości początkowe. Jeżeli pola te zawierają zmienne referencyjne, zwykle trzeba stworzyć przypisane im obiekty. Czasami też przed użyciem danego obiektu chcemy wykonać bardziej złożony kod. Można oczywiście napisać w tym celu dodatkową, zwykłą metodę i używać jej na przykład tak:

```
Klasa obiekt = new Klasa();  
obiekt.inicjujPola();
```

Zakładając, że nowo tworzonemu obiektom znanej nam już klasy Punkt będziemy chcieli przypisać początkowe wartości x = 320, y = 200, metoda ta wyglądałaby następująco:

```
void inicjujPola()
{
    x = 320;
    y = 200;
}
```

Gdybyśmy chcieli nadać tym polom różne wartości w zależności od obiektu, zapewne skorzystalibyśmy z metody `ustawWspolrzedne` wywoływanej tuż po powołaniu obiektu do życia. Wyglądałoby to zatem następująco:

```
Punkt punkt = new Punkt();
punkt.ustawWspolrzedne(320, 200);
```

Takie czynności można jednak wykonywać automatycznie. Służą do tego specjalne metody zwane **konstruktorami**. Są one wykonywane zawsze przy tworzeniu nowego obiektu. Konstruktory są bezrezultatywne (nie zwracają żadnych wartości) oraz mają nazwę identyczną z nazwą klasy, której dotyczą. Mogą mieć natomiast różne parametry. Schemat budowy klasy z konstruktorem jest następujący:

```
class nazwa_klasy {
    nazwa_klasy() {
        // kod konstruktora
    }
}
```

Stwórzmy zatem najprostszy, bezargumentowy konstruktor dla klasy `Punkt`, który przypisze polom klasy wartości 320 i 200.

Ć W I C Z E N I E

4.9 Tworzenie konstruktora bezargumentowego

Dodaj do klasy `Punkt` bezargumentowy konstruktor, który ustawi wartość pola `x` na 320, a pola `y` na 200.

```
class Punkt
{
    int x, y;
    Punkt()
    {
        x = 320;
        y = 200;
    }
    // tutaj dalsze metody klasy Punkt
}
```

Konstruktor ma bardzo prostą budowę. Zgodnie z podanym wyżej schematem nie zwraca żadnej wartości (nie ma też przed nim słowa `void`!). W jego wnętrzu polu `x` przypisywana jest wartość 320, a polu `y` — wartość 200. Tak więc będą to początkowe wartości każdego obiektu typu `Punkt`.

Ć W I C Z E N I E

4.10 Testowanie konstruktora

Przetestuj działanie konstruktora klasy `Punkt` z ćwiczenia 4.9.

```
class Main {
    public static void main (String args[]) {
        Punkt punkt = new Punkt();
        System.out.print ("Współrzędna x = " + punkt.x);
        System.out.println (" Współrzędna y = " + punkt.y);
    }
}
```

Cały test zawiera się w trzech wierszach kodu. Najpierw tworzony jest nowy obiekt typu `Punkt`, a następnie wyświetlane są wartości jego pól `x` i `y`. Po uruchomieniu takiego programu okaże się, że mimo braku jawnego przypisania wartości polom `x` i `y` mają one wartości 320 i 200, a zatem konstruktor został prawidłowo wykonany.

Nic nie stoi na przeszkodzie, aby konstruktor przyjmował argumenty dokładnie tak jak zwyczajna metoda. Wtedy już w momencie tworzenia obiekt będzie mógł otrzymać pewne wartości. Mogą one posłużyć do zainicjowania pól.

Ć W I C Z E N I E

4.11 Konstruktor z argumentami

Napisz konstruktor klasy `Punkt` przyjmujący dwa argumenty określające wartości pól `x` i `y`.

```
class Punkt
{
    int x, y;
    Punkt(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    // tutaj dalsze metody klasy Punkt
}
```

Zaprezentowany konstruktor działa na takiej samej zasadzie jak metoda `ustawWspolrzedne` prezentowana we wcześniejszych ćwiczeniach. Otrzymuje argumenty `wspX` oraz `wspY` i przypisuje ich wartości polom `x` i `y`. Warto zauważyć, że skoro taki konstruktor w rzeczywistości dubluje kod metody `ustawWspolrzedne`, to dobrym pomysłem jest po prostu wywołanie tej metody w konstruktorze. Wtedy jego kod przyjąłby postać:

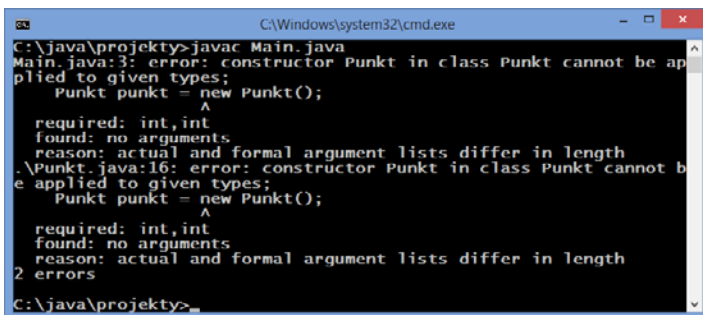
```
Punkt(int wspX, int wspY)
{
    ustawWspolrzedne(wspX, wspY);
}
```

Ć W I C Z E N I E

4.12 Próba wywołania konstruktora z argumentami

Spróbuj użyć klasy `Punkt` z ćwiczenia 4.11 i klasy `Main` z ćwiczenia 4.10. Co się stanie przy próbie kompilacji takiego kodu?

Przy próbie kompilacji klasy `Main` z ćwiczenia 4.10 wraz z klasą `Punkt` z ćwiczenia 4.11 zobaczymy jedynie komunikaty o błędach przedstawione na rysunku 4.1. Dlaczego kompilacja nie była możliwa? Otóż skoro w klasie `Punkt` jest tylko jeden konstruktor, który przyjmuje dwa argumenty, to musi być wywołany z tymi argumentami. Wywołanie bez argumentów nie jest możliwe. A zatem przy takiej konstrukcji kodu nie można tworzyć obiektów klasy `Punkt` bez podania wartości pól `x` i `y` — błąd występuje zarówno w klasie `Main`, jak i w klasie `Punkt` (w metodzie `pobierzWspolrzedne`, która nie przyjmuje argumentów).



Rysunek 4.1. Błędy kompilacji wynikające z nieprawidłowego użycia konstruktora

Skoro w klasie jest tylko konstruktor przyjmujący argumenty, to muszą być one podawane przy tworzeniu każdego obiektu tej klasy. Schemat wywołania jest wtedy następujący:

```
new Klasa(argumenty);
```

lub taki (gdy od razu następuje przypisanie referencji do nowego obiektu zmiennej):

```
Klasa zmienna = new Klasa(argumenty)
```

Warto zauważyć, że stosowany do tej pory zapis `new Klasa()` to nic innego jak wywołanie konstruktora bez przekazywania mu jakichkolwiek argumentów, czyli konstruktora bezargumentowego⁶.

Ć W I C Z E N I E

4.13 Dostosowanie kodu do konstruktora dwuargumentowego

Zmień kod klas z ćwiczeń 4.10 i 4.11, tak aby poprawnie ze sobą współpracowały.

W klasie `Main` z ćwiczenia 4.10 należy poprawić wiersz kodu:

```
Punkt punkt = new Punkt();
```

Skoro konstruktor musi przyjmować argumenty, to należy je podać przy tworzeniu obiektu. Wywołanie mogłoby zatem mieć postać:

```
Punkt punkt = new Punkt(0, 0);
```

Wtedy pola `x` i `y` obiektu `punkt` przyjmą wartość 0.

W klasie `Punkt` należy natomiast zmienić kod metody `pobierzWspolrzedne` zwracającej obiekt typu `Punkt` (kod tej metody nie był uwzględniony w ćwiczeniu 4.11, znajduje się natomiast w ćwiczeniu 4.7). Tam również powstawał nowy obiekt typu `Punkt`:

```
Punkt pobierzWspolrzedne()  
{  
    Punkt punkt = new Punkt();  
    punkt.x = x;
```

⁶ Jeśli w klasie nie zdefiniujemy żadnego konstruktora, to zostanie do niej dodany automatycznie pusty konstruktor bezargumentowy. Ten temat został omówiony dokładniej m.in. w publikacji *Praktyczny kurs Java* (<http://helion.pl/ksiazki/pkjav3.htm>).


```
punkt.y = y;  
return punkt;  
}
```

W tym przypadku poprawka, którą należy wprowadzić, spowoduje znaczne uproszczenie kodu, treść metody będzie można bowiem zapisać w jednym tylko wierszu:

```
Punkt pobierzWspolrzedne()  
{  
    return new Punkt(x, y);  
}
```

Nie ma tu potrzeby tworzenia żadnych zmiennych pomocniczych. Należy po prostu zwrócić za pomocą instrukcji `return` nowy obiekt typu `Punkt` (ściślej: referencję do obiektu typu `Punkt`), którego konstruktorowi zostały przekazane bieżące wartości `x` i `y`.

Konstruktory, tak jak i zwykle metody, można przeciążać, a więc może istnieć kilka konstruktorów w ramach jednej klasy. Dodajmy więc do klasy `Punkt` dwa konstruktory. Jeden będzie bezargumentowy i będzie ustawiał wartości `x` i `y` na 320 i 200, drugi zaś ustawi współrzędne na wartości podane przez użytkownika w postaci argumentów.

Ć W I C Z E N I E

4.14 Tworzenie przeciążonych konstruktorów

Umieść w klasie `Punkt` dwa różne konstruktory. Jeden bezargumentowy, a drugi przyjmujący dwa argumenty określające wartości pól `x` i `y`.

```
class Punkt  
{  
    int x, y;  
    Punkt()  
    {  
        x = 320;  
        y = 200;  
    }  
    Punkt(int wspX, int wspY)  
    {  
        x = wspX;  
        y = wspY;  
    }  
    // tutaj dalsze metody klasy Punkt  
}
```

Rozwiązanie ćwiczenia to nic innego jak umieszczenie w klasie Punkt konstruktorów przedstawionych w ćwiczeniach 4.9 i 4.11. W takim przypadku kod metody `pobierzWspolrzedne` będzie mógł mieć zarówno postać z ćwiczenia 4.13, jak i z ćwiczenia 4.7. Obie wersje będą prawidłowe.

Ć W I C Z E N I E

4.15 Użycie różnych konstruktorów

Przetestuj działanie konstruktorów klasy `Punkt` utworzonych w ćwiczeniu 4.14.

```
class Main {
    public static void main (String args[]) {
        Punkt punkt1 = new Punkt();
        Punkt punkt2 = new Punkt(100, 100);
        System.out.println("Punkt1: Współrzędna x = " + punkt1.pobierzX());
        System.out.println("Punkt1: Współrzędna y = " + punkt1.pobierzY());
        System.out.println("Punkt2: Współrzędna x = " + punkt2.pobierzX());
        System.out.println("Punkt2: Współrzędna y = " + punkt2.pobierzY());
    }
}
```

Powstały dwa obiekty: `punkt1` i `punkt2`. W pierwszym przypadku został użyty konstruktor bezargumentowy, a więc wartością pola `x` będzie 320, a pola `y` — 200. W drugim przypadku użyto konstruktora dwuargumentowego, a więc polom zostaną przypisane wartości przekazane w postaci argumentów (czyli `x` będzie równe 100 i `y` również będzie równe 100).

Jeżeli w każdym konstruktorze miałby się znaleźć taki sam fragment kodu, to (znając tylko dotychczas przedstawione techniki) kod ten można by dodawać do każdego konstruktora bądź zapisać w osobnej metodzie wywoływanej w każdym konstruktorze. W obu tych rozwiązaniach łatwo jednak o pomyłki — złym pomysłem jest wielokrotne powtarzanie identycznego fragmentu w wielu miejscach. Aby tego uniknąć, można zastosować tzw. **blok inicjalizacyjny**. Schemat postępowania wygląda następująco:

```
class nazwa_klasy
{
    // pola klasy
    {
        // treść bloku inicjalizacyjnego
    }
}
```

```
}  
// dalsze pola i metody klasy  
}
```

W takim przypadku wszystkie instrukcje z bloku inicjalizacyjnego będą zawsze wykonywane przy tworzeniu obiektów tej klasy (w praktyce — zostaną skopiowane do każdego z konstruktorów klasy).

Ć W I C Z E N I E

4.16 Klasa z blokiem inicjalizacyjnym

Przygotuj taką wersję klasy Punkt z ćwiczenia 4.2, aby przy tworzeniu obiektów pole *x* otrzymywało wartość 320, a pole *y* — 200. Nie dodawaj do klasy konstruktorów. Napisz program sprawdzający poprawność działania nowej klasy.

Skoro (zgodnie z treścią zadania) do klasy Punkt nie można dopisać jawnie konstruktorów⁷, trzeba zastosować blok inicjalizacyjny. Klasa ta będzie zatem miała następującą postać:

```
class Punkt  
{  
    int x, y;  
    {  
        x = 320;  
        y = 200;  
    }  
    int pobierzX()  
    {  
        return x;  
    }  
    int pobierzY()  
    {  
        return y;  
    }  
}
```

Aby przetestować jej działanie, można użyć osobnej klasy Main⁸, np. w poniższej postaci:

```
class Main {  
    public static void main (String args[]) {  
        Punkt punkt1 = new Punkt();  
        System.out.println("Współrzędna x = " + punkt1.pobierzX());  
    }  
}
```

⁷ Kompilator doda zatem niejawnie bezargumentowy konstruktor domyślny.

⁸ Można by było również dopisać metodę `main` do klasy `Punkt`.

```
        System.out.println("Współrzędna y = " + punkt1.pobierzY());  
    }  
}
```

Po skompilowaniu obu klas i uruchomieniu programu okaże się, że obiekt wskazywany przez zmienną `punkt1` ma pola o wartościach 320 i 200, mimo że wartości te nie były ustawiane w metodzie `main`. A zatem na pewno został wykonany blok inicjalizacyjny.

Specyfikatory dostępu

Specyfikatory dostępu (nazywane również modyfikatorami dostępu, ang. *access modifiers*) określają prawa dostępu do składowych klas (a także do samych klas, podrozdział „Pakiety”). W związku z tym pola oraz metody mogą być:

- ☐ publiczne (`public`),
- ☐ prywatne (`private`),
- ☐ chronione (`protected`),
- ☐ pakietowe.

Publiczne składowe klasy oznacza się słowem `public`, co sygnalizuje, że wszyscy mają do nich dostęp oraz że są dziedziczone przez klasy potomne (podrozdział „Dziedziczenie”). Do składowych prywatnych (`private`) można dostać się tylko z wnętrza danej klasy, natomiast do składowych chronionych (`protected`) można się dostać z wnętrza danej klasy, klas potomnych (pochodnych) oraz innych klas zawartych w danym pakiecie. Jeżeli nie występuje żadne z wyżej wymienionych słów, składowa klasy jest pakietowa, tzn. dostęp do niej jest ograniczony w obrębie pakietu (podrozdział „Pakiety”). Takie właśnie (pakietowe) były pola i metody dotychczas omawianych klas.

Jak to wygląda w praktyce? Wróćmy do przykładowej klasy `Punkt`. Pola oznaczające współrzędne `x` i `y` nie miały żadnego dodatkowego oznaczenia, a zatem były polami pakietowymi. Napiszmy teraz klasę `Punkt`, w której pola `x` i `y` będą prywatne.

ĆWICZENIE

4.17 Prywatne pola klasy Punkt

Napisz klasę Punkt zawierającą prywatne pola `x` i `y`.

```
class Punkt
{
    private int x, y;
    Punkt()
    {
        x = 320;
        y = 200;
    }
    Punkt (int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
}
```

Jest to uproszczona wersja klasy Punkt, zawierająca dwa konstruktory oraz prywatne pola `x` i `y` (prywatne, ponieważ przed ich definicją znajduje się słowo `private`). Gdyby składowe były definiowane w dwóch wierszach, to przed każdą definicją powinno się znaleźć słowo `private`:

```
private int x;
private int y;
```

Czy do tak zdefiniowanych pól uda się odwołać z innej klasy? Zgodnie z powyższymi informacjami — nie. Sprawdźmy to jednak w praktyce.

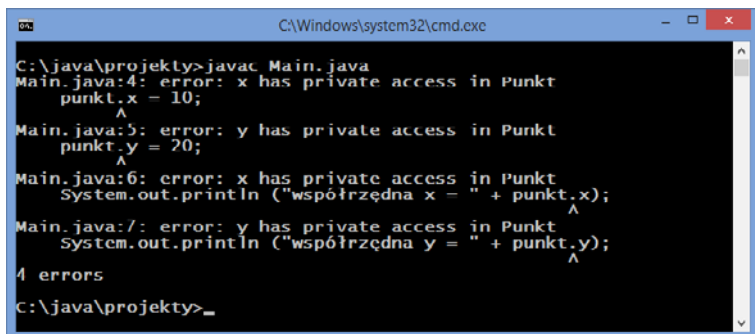
ĆWICZENIE

4.18 Próba odwołania do prywatnych pól

W klasie Main odwołaj się bezpośrednio do pól `x` i `y` klasy Punkt z ćwiczenia 4.17. Spróbuj skompilować otrzymany kod.

```
class Main {
    public static void main (String args[]) {
        Punkt punkt = new Punkt();
        punkt.x = 10;
        punkt.y = 20;
        System.out.println ("współrzędna x = " + punkt.x);
        System.out.println ("współrzędna y = " + punkt.y);
    }
}
```

Kod klasy wygląda standardowo. Utworzony został obiekt typu `Punkt`, a następnie polom `x` i `y` zostały przypisane wartości oraz nastąpiło ich wyświetlenie na ekranie. Próba kompilacji tego kodu (w połączeniu z klasą `Punkt` z ćwiczenia 4.17) spowoduje jednak wyłącznie wyświetlenie czterech komunikatów o błędach (rysunek 4.2), czyli po jednym na każde z odwołań do pól klasy. Skoro te pola są prywatne, to spoza klasy nie można się do nich odwoływać ani przy zapisie, ani przy odczycie.



Rysunek 4.2. Błędy kompilacji spowodowane niewłaściwymi odwołaniami do składowych prywatnych

Jak w takim razie odwoływać się do prywatnych pól? Wystarczy przygotować publiczne (lub pakietowe) metody operujące na tych polach. Z takimi metodami mieliśmy już do czynienia. To `ustawX`, `ustawY`, `pobierzX`, `pobierzY` z ćwiczenia 4.6. Wystarczy dodać je do klasy `Punkt` i uczynić publicznymi, aby za ich pośrednictwem każda inna klasa mogła operować na współrzędnych `x` i `y` dowolnego obiektu typu `Punkt`.

ĆWICZENIE

4.19 Metody operujące na prywatnych polach

Przygotuj wersję klasy `Punkt` z prywatnymi polami `x` i `y` oraz publicznymi metodami pozwalającymi na operowanie na tych polach.

```
class Punkt
{
    private int x, y;
    public Punkt()
    {
```

```
        x = 320;
        y = 200;
    }
    public Punkt (int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    public void ustawX(int wspX)
    {
        x = wspX;
    }
    public void ustawY(int wspY)
    {
        y = wspY;
    }
    public int pobierzX()
    {
        return x;
    }
    public int pobierzY()
    {
        return y;
    }
}
```

Ta wersja kodu łączy przykłady z ćwiczeń 4.6 oraz 4.17, z tym że pola klasy zostały zadeklarowane jako prywatne, natomiast wszystkie pozostałe metody, łącznie z konstruktorami — jako publiczne. A zatem i do metod, i do konstruktorów będzie swobodny dostęp bez żadnych ograniczeń.

Ć W I C Z E N I E

4.20 Korzystanie z obiektów klasy zawierających prywatne pola

Zmień kod klasy Main z ćwiczenia 4.18, tak aby współpracował z klasą Punkt z ćwiczenia 4.19.

```
class Main {
    public static void main (String args[]) {
        Punkt punkt = new Punkt();
        punkt.ustawX(10);
        punkt.ustawY(20);
        System.out.println ("współrzędna x = " + punkt.pobierzX());
        System.out.println ("współrzędna y = " + punkt.pobierzY());
    }
}
```

Ogólna struktura kodu pozostała tu taka sama jak w ćwiczeniu 4.18, jednak bezpośrednie odwołania do pól x i y zostały zamienione na wywołania metod: `ustawX` i `ustawY` (które ustawiają wartości pól) oraz `pobierzX` i `pobierzY` (które pobierają wartości pól). Tym samym uzyskaliśmy dostęp do prywatnych składowych obiektu.

Po co jednak tworzyć prywatne składowe klasy? Przecież można by zmienić dostęp do pól x i y z prywatnego na publiczny (lub pakietowy). Gdyby definicja pól miała postać:

```
public int x, y;
```

to kod z ćwiczenia 4.18 zadziałałby przecież bez problemów. Dlaczego więc nie stosować tylko tego ostatniego sposobu i nie deklarować wszystkich pól oraz metod jako publiczne? Otóż po to, aby ukryć wewnętrzną organizację obiektu, a na „zewnątrz” udostępnić jedynie interfejs pozwalający na wykonywanie ściśle określonych przez programistę operacji. Przydaje się to zwykle w przypadku bardziej skomplikowanych klas, jednak nawet na tak prostym przykładzie jak klasa `Punkt` można pokazać istotę tego zagadnienia.

Otóż założmy, że mamy już napisany program, ale z jakichś powodów musimy zmienić reprezentację współrzędnych i teraz punkt identyfikujemy za pomocą kąta alfa oraz odległości punktu od początku układu współrzędnych, czyli musimy przejść ze współrzędnych w układzie kartezjańskim na współrzędne w układzie biegunowym. W klasie `Punkt` nie będzie już zatem pól x i y , nie będą też miały sensu odwołania do nich. Nie dość, że we wszystkich innych klasach trzeba by w takim przypadku zmienić odwołania do obiektów typu `Punkt`, to także w każdym miejscu konieczne by było dokonanie przeliczeń współrzędnych. Spowodowałoby to ogromne komplikacje i z pewnością doprowadziłoby do powstania wielu błędów. Jeżeli jednak pola klasy będą prywatne, trzeba będzie tylko przededefiniować metody klasy `Punkt`. Cała reszta programu nawet nie zauważy, że coś się zmieniło!

Ć W I C Z E N I E

4.21 Zmiana sposobu reprezentacji współrzędnych

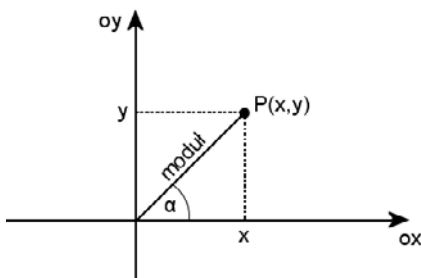
Zmień definicję klasy `Punkt` w taki sposób, aby położenie punktu było reprezentowane w układzie biegunowym.


```
class Punkt
{
    private double modul, sinalfa;
    public Punkt()
    {
        ustawWspolrzedne(320, 200);
    }
    public Punkt (int x, int y)
    {
        ustawWspolrzedne(x, y);
    }
    public void ustawWspolrzedne(int wspX, int wspY)
    {
        modul = Math.sqrt (wspX * wspX + wspY * wspY);
        sinalfa = wspY / modul;
    }
    public Punkt pobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.sinalfa = sinalfa;
        punkt.modul = modul;
        return punkt;
    }
    public void pobierzWspolrzedne(Punkt punkt)
    {
        punkt.sinalfa = sinalfa;
        punkt.modul = modul;
    }
    void ustawX(int wspX)
    {
        ustawWspolrzedne(wspX, pobierzY());
    }
    public void ustawY(int wspY)
    {
        ustawWspolrzedne(pobierzX(), wspY);
    }
    public int pobierzX()
    {
        double x;
        x = modul * Math.sqrt(1 - sinalfa * sinalfa);
        return (int) Math.round(x);
    }
    public int pobierzY()
    {
        double y;
        y = modul * sinalfa;
        return (int) Math.round(y);
    }
}
```

Przeliczenie współrzędnych kartezjańskich (tzn. w postaci x, y) na układ biegunowy (czyli kąt i moduł) nie jest skomplikowane. Do reprezentacji kąta najwygodniejsza jest funkcja *sinus*, dlatego też została użyta w powyższym przykładzie (patrz też rysunek 4.3). Zatem sinus kąta α ⁹, reprezentowany przez pole o nazwie *sinalfa*, jest równy $\frac{y}{\text{modul}}$. Natomiast sam moduł, reprezentowany przez pole o nazwie *modul*, to $\sqrt{x^2 + y^2}$.

Rysunek 4.3.

Punkt w układzie współrzędnych



Przeliczenie współrzędnych biegunowych na kartezjańskie jest nieco trudniejsze. Co prawda y to po prostu $\text{modul} * \sin(\alpha)$, za to x to $\text{modul} * \sqrt{1 - \sin^2(\alpha)}$. Obliczenia są wykonywane w metodach *pobierzWspolrzedne*, *ustawX*, *ustawY*, *pobierzX* i *pobierzY*¹⁰. Zamiast podnoszenia do drugiej potęgi wykonywane jest po prostu odpowiednie mnożenie. Do uzyskania wartości pierwiastka kwadratowego (drugiego stopnia) została użyta metoda *Math.sqrt()*. Sinus kąta α reprezentowany jest przez pole *sinalfa*. Metoda *Math.round(arg)* zaokrąglą argument do liczby całkowitej i zwraca wynik typu *long*. Niestosowany do tej pory zapis (nazwaTypu) *zmienna*, np. *(int) x*, oznacza konwersję (rzutowanie) zmiennej do podanego typu — w naszym przykładzie konwersję

⁹ Chodzi o kąt pomiędzy prostą przechodzącą przez reprezentowany punkt i środek układu współrzędnych a osią *OX*.

¹⁰ Przedstawiony kod będzie działał poprawnie tylko dla dodatnich współrzędnych x , natomiast ujemne wartości x będą traciły znak (czyli zostaną przekształcone na wartości dodatnie). Ćwiczenie zostało pozostawione w takiej postaci, aby nie wprowadzać dodatkowych instrukcji zaciemniających sedno zagadnienia. Ponieważ jednak poprawki nie są bardzo skomplikowane, warto potraktować je jako ćwiczenie do samodzielnego wykonania.

wyniku działania metody `round` do typu `int` (który ma być wynikiem działania funkcji). Warto samodzielnie sprawdzić, co się stanie po usunięciu jawnej konwersji).

Należy też zwrócić uwagę, że w metodach `ustawX` i `ustawY` (ponieważ ustawiają one wartość tylko jednego z argumentów) konieczne było uzyskanie bieżącej wartości drugiego argumentu. Dlatego też zostały w nich użyte metody `pobierzX` i `pobierzY`.

Jeśli teraz tak zmienionej klasy `Punkt` użyjemy wraz z jedną z dotychczasowych klas `Main`, np. tą z ćwiczenia 4.20, szybko się przekonamy, że całość działa bez zarzutu, mimo iż reprezentacja danych jest całkowicie odmienna niż we wcześniejszych przykładach.

Pakiety i typy klas

Pakiet to grupa klas zajmująca się zwykle jednym zagadnieniem, np. pakiet `java.awt.net` zawiera klasy zajmujące się obsługą sieci. Jeśli chcemy w programie użyć klasy z jakiegoś pakietu (czyli mieć dostęp do klas zawartych w tym pakiecie), na początku kodu umieszczamy słowo `import`, a po nim nazwę klasy poprzedzoną nazwą pakietu, schematycznie:

```
import nazwa_pakietu.nazwa_klasy;
```

Jeśli chcemy mieć dostęp do wszystkich klas z danego pakietu, zamiast nazwy klasy umieszczamy znak `*`:

```
import nazwa_pakietu.*;
```

Przykładowo jeśli ma być używana wyłącznie klasa `Vector` z pakietu `java.util`, użyjemy konstrukcji:

```
import java.util.Vector;
```

Jeżeli natomiast mają być dostępne wszystkie klasy z pakietu `java.util`, właściwa będzie konstrukcja:

```
import java.util.*;
```

Wraz z JDK dostarczanych jest kilka tysięcy klas, z których możemy dowolnie korzystać w naszych programach.

W tym miejscu trzeba też koniecznie zaznaczyć, że klasy w Javie mogą być publiczne lub pakietowe. Klasa jest publiczna, jeżeli została zadeklarowana ze słowem `public`, schematycznie:

```
public class nazwa_klasy{  
    // wewnątrz klasy  
}
```

Gdy definicja nie zawiera słowa `public`, klasa jest pakietowa. Z tego wynika, że wszystkie dotychczas tworzone klasy były pakietowe. Jaka jest różnica? Otóż do klasy publicznej dostęp jest swobodny. Czyli można tworzyć obiekty takich klas w dowolnej innej klasie. W przypadku klas pakietowych dostęp do nich jest możliwy tylko w obrębie pakietu, do którego należą. Definicje klas pakietowych muszą się znajdować w plikach w jednym katalogu, przy czym nazwa klasy pakietowej nie musi być zgodna z nazwą pliku.

Co ważne, w przypadku klas publicznych jest inaczej. Klasa publiczna musi się znajdować w pliku o nazwie zgodnej z nazwą klasy. Czyli jeśli mamy publiczną klasę `Punkt`, to musi być zdefiniowana w pliku o nazwie `Punkt.java`. Umieszczenie takiej klasy w pliku o innej nazwie spowoduje błąd kompilacji.

Ogólnie rzecz ujmując, klasy pakietowe możemy traktować jak klasy pomocnicze, usługowe dla klas publicznych. Spotyka się rozwiązania polegające na tym, że definicje takich klas znajdują się w tym samym pliku co definicja klasy publicznej, z którą są powiązane.

Od tego miejsca w większości dalszych przykładów będą tworzone klasy publiczne.

Dziedziczenie

Znamy już dosyć dobrze klasę `Punkt`, załóżmy jednak, że potrzebujemy obiektów opisujących nie tylko współrzędne punktu, ale również jego kolor. Przyjmijmy, że kolor będzie określany przez wartość typu `int` (powiedzmy, że będzie to indeks koloru). Chcielibyśmy przy tym mieć możliwość jednoczesnego korzystania z obu wersji. Można oczywiście stworzyć nową klasę typu `KolorowyPunkt`, która mogłaby wyglądać następująco:

```
public class Punkt {  
    private int x, y, kolor;  
}
```

Musieliśmy teraz dopisać wszystkie zdefiniowane wcześniej metody klasy Punkt oraz zapewne dodatkowo ustawKolor i pobierzKolor. W ten sposób jednak dwukrotnie napisalibyśmy ten sam kod. Przecież klasy Punkt i KolorowyPunkt robią w dużej części to samo, a dokładniej to klasa KolorowyPunkt jest swego rodzaju rozszerzeniem klasy Punkt. Zatem niech KolorowyPunkt przejmie własności klasy Punkt, a dodatkowo dodajmy do niej pole określające kolor. Jest to dziedziczenie, znane m.in. z C++, które w Javie definiuje się poprzez słowo extends (rozszerza). Klasa Punkt będzie zatem klasą nadrzędną (bazową), a klasa KolorowyPunkt — klasą podrzędną (potomną). Powiemy wtedy, że klasa KolorowyPunkt dziedziczy po klasie Punkt (rozpowszechniony jest również zwrot „dziedziczy z klasy”).

ĆWICZENIE

4.22 Dziedziczenie po klasie Punkt

Napisz klasę KolorowyPunkt rozszerzającą klasę Punkt (np. w wersji z ćwiczenia 4.19) o możliwość przechowywania informacji o kolorze.

```
public class KolorowyPunkt extends Punkt {
    private int kolor;
    public KolorowyPunkt()
    {
        super();
        kolor = 0;
    }
    public KolorowyPunkt(int wspX, int wspY, int nowyKolor)
    {
        super(wspX, wspY);
        kolor = nowyKolor;
    }
    public void ustawKolor(int nowyKolor)
    {
        kolor = nowyKolor;
    }
    public int pobierzKolor()
    {
        return kolor;
    }
}
```

Klasa KolorowyPunkt jest rozszerzeniem klasy Punkt. To znaczy, że zawiera pola i metody klasy Punkt oraz dodatkowo pola i metody zdefiniowane w KolorowyPunkt. Dopisane zostały metody operujące na nowej składowej o nazwie kolor. Ponieważ jest ona prywatna, tylko

dzięki nim możliwe będzie pobieranie tej wartości (`pobierzKolor`) oraz jej ustawianie (`ustawKolor`). Te metody działają analogicznie do przedstawionych w ćwiczeniu 4.19 `pobierzX` i `ustawX`.

W klasie dostępne są również dwa konstruktory: bezargumentowy i trójargumentowy. Pierwszy ustawia wartość pola `kolor` na 0, a drugi — na wartość argumentu `nowyKolor`. Oba ustawiają też wartości pól `x` i `y`, ale odbywa się to przez wywołanie konstruktora klasy bazowej, czyli klasy `Punkt`. Jest to wykonywane za pomocą metody `super`. A zatem zapis:

```
super();
```

powoduje wywołanie bezargumentowego konstruktora klasy `Punkt` (klasy nadrzędnej w stosunku do `KolorowyPunkt`), a wywołanie:

```
super(wspX, wspY);
```

powoduje wywołanie dwuargumentowego konstruktora klasy `Punkt` i przekazanie mu wartości `wspX` i `wspY`.

Ć W I C Z E N I E

4.23 Testowanie klasy `KolorowyPunkt`

Napisz klasę `Main` umożliwiającą przetestowanie działania klasy `KolorowyPunkt`.

```
public class Main {  
    public static void main (String args[]) {  
        KolorowyPunkt punkt = new KolorowyPunkt(100, 200, 10);  
        System.out.println ("współrzędna x = " + punkt.pobierzX());  
        System.out.println ("współrzędna y = " + punkt.pobierzY());  
        System.out.println ("kolor = " + punkt.pobierzKolor());  
    }  
}
```

Oczywiście z klasy `KolorowyPunkt` możemy wyprowadzić kolejną, np. `DwuKolorowyPunkt`, dla punktów, które przyjmują dwa różne kolory, np. w zależności od znaku wartości współrzędnej `x`. Po klasie `DwuKolorowyPunkt` może dziedziczyć kolejna klasa itd.

Wszystkie klasy bezpośrednio lub pośrednio dziedziczą po klasie bazowej `Object`. Jeżeli zatem piszemy:

```
public class Punkt{  
    // wewnątrz klasy  
}
```

w domyśle jest przyjmowane, że klasa Punkt dziedziczy po klasie Object, tzn. zapis ten jest tłumaczony jako:

```
public class Punkt extends Object {  
    // wewnątrz klasy  
}
```

Dociekliwi czytelnicy spytają zapewne: co się stanie, jeśli zarówno w klasie bazowej, jak i potomnej znajdzie się metoda o takiej samej nazwie? Czy kompilator zgłosi wtedy błąd? Przekonajmy się o tym. Napiszmy dwie przykładowe klasy spełniające taki warunek.

Ć W I C Z E N I E

4.24 Przesłanianie metod przy dziedziczeniu

Napisz przykładowe klasy o nazwach KlasaA i KlasaB. KlasaB ma dziedziczyć po klasie KlasaA, natomiast w obu klasach ma zostać zdefiniowana metoda o nazwie f. Wykonaj kompilację obu klas.

```
class KlasaA {  
    void f()  
    {  
        System.out.println("Metoda f z klasy KlasaA.");  
    }  
}  
  
class KlasaB extends KlasaA {  
    void f()  
    {  
        System.out.println("Metoda f z klasy KlasaB.");  
    }  
}
```

Obie klasy są pakietowe, zatem możemy je umieścić w pliku o dowolnej nazwie, najlepiej *Main.java* (to ułatwi przeprowadzenie kolejnych ćwiczeń). Klasa KlasaB dziedziczy po klasie KlasaA. Tymczasem w obu znajduje się bezargumentowa metoda o nazwie f. Gdy jednak dokonamy kompilacji obu klas, okaże się, że kompilator nie generuje żadnego błędu, a zatem kod jest formalnie poprawny.

Jak zatem interpretować kod z powyższego ćwiczenia? Wiadomo, że klasa dziedzicząca przejmuje metody z klasy bazowej. Skoro w obu klasach występuje metoda o nazwie *f*, to wydawałoby się, że obiekty klasy *KlasaB* powinny zawierać dwie metody o takich samych nazwach i parametrach. I tak jest w istocie! Jest to możliwe dzięki temu, że w takim przypadku metoda z klasy bazowej jest przesłaniana przez metodę z klasy pochodnej. W związku z tym standardowo będzie dostępna tylko metoda z klasy *KlasaB*.

Ć W I C Z E N I E

4.25 Wywoływanie przesłoniętych metod

Utwórz obiekty klas *KlasaA* i *KlasaB*. Wywołaj ich metody *f*. Sprawdź zachowanie programu.

// tutaj kod z ćwiczenia 4.24

```
public class Main {
    public static void main (String args[]) {
        KlasaA obiektA = new KlasaA();
        KlasaB obiektB = new KlasaB();

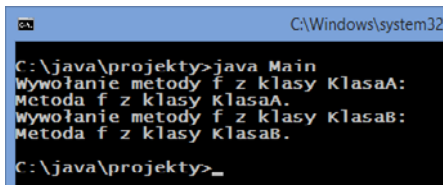
        System.out.println("Wywołanie metody f z klasy KlasaA:");
        obiektA.f();

        System.out.println("Wywołanie metody f z klasy KlasaB:");
        obiektB.f();
    }
}
```

Powstały dwa obiekty: *obiektA* i *obiektB*. Pierwszy jest typu *KlasaA*, drugi — typu *KlasaB*. Następnie zostały wywołane metody *f* każdego z obiektów. Zgodnie z wcześniejszym opisem w pierwszym przypadku zostanie użyta metoda *f* z klasy *KlasaA*, a w drugim — metoda *f* z klasy *KlasaB*. Zatem po uruchomieniu takiego programu zobaczymy widok przedstawiony na rysunku 4.4.

Rysunek 4.4.

Zostały wywołane metody właściwe dla każdej z klas



W tym miejscu pojawia się pytanie: czy można w takim razie w obiekcie typu `KlasaB` uzyskać dostęp do zawartej w nim metody `f` pochodzącej z klasy `KlasaA`? Jest to jak najbardziej możliwe. Niezbędne jest tylko użycie specjalnej konstrukcji ze słowem `super`. Schematycznie takie wywołanie ma postać:

```
super.nazwa_metody(argument_metody);
```

ĆWICZENIE

4.26 Dostęp do przesłoniętej metody

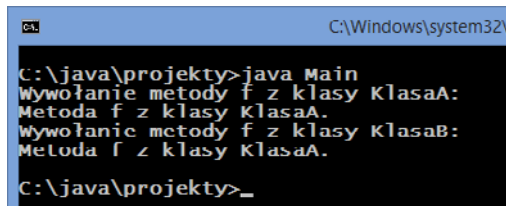
Zmodyfikuj kod z ćwiczenia 4.24 w taki sposób, aby w klasie `KlasaB` w metodzie `f` została wywołana metoda `f` z klasy `KlasaA`.

```
class KlasaA {  
    void f()  
    {  
        System.out.println("Metoda f z klasy KlasaA.");  
    }  
}  
  
class KlasaB extends KlasaA {  
    void f()  
    {  
        super.f();  
    }  
}
```

Metoda `f` z klasy `KlasaB` nie zawiera teraz żadnego kodu, oprócz wywołania metody `f` z klasy nadrzędnej (`KlasaA`). Tym samym wywołanie metody `f` obiektu typu `KlasaB` spowoduje wywołanie przesłoniętej metody pochodzącej z klasy `KlasaA`, a to właśnie było celem ćwiczenia. O tym, że tak jest w istocie, można się przekonać, testując powyższy kod wraz z klasą `Main` przedstawioną w ćwiczeniu 4.25. Wtedy zamiast efektu widocznego na rysunku 4.4 zobaczymy ten przedstawiony na rysunku 4.5.

Rysunek 4.5.

*Efekt wywołania
przesłoniętej
metody z klasy
bazowej*



```
C:\Windows\system32\cmd.exe  
C:\java\projekty>java Main  
Wywołanie metody f z klasy KlasaA:  
Metoda f z klasy KlasaA.  
Wywołanie metody f z klasy KlasaB:  
Metoda f z klasy KlasaA.  
C:\java\projekty>_
```


5

Obsługa błędów oraz wyjątki

Błędy w programach

W każdym większym programie występują jakieś błędy. Oczywiście staramy się przed nimi ustrzec, nigdy jednak nie uda nam się ich całkowicie wyeliminować. Co więcej, aby program wychwytywał przynajmniej część błędów, których przyczyną jest np. wprowadzenie złych wartości przez użytkownika, musimy napisać wiele wierszy dodatkowego kodu, który zaciemnia główny sens programu. Jeżeli na przykład zadeklarowaliśmy tablicę 5-elementową, należałoby sprawdzić, czy nie następuje odwołanie do nieistniejącego elementu.

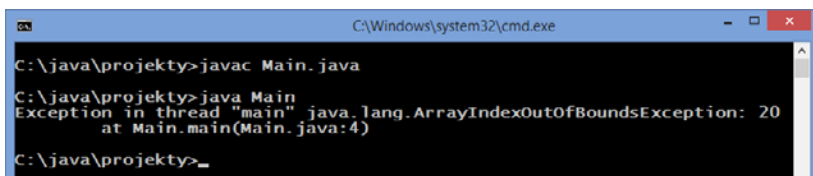
Ć W I C Z E N I E

5.1 Odwołanie do nieistniejącego elementu tablicy

W klasie Main zadeklaruj tablicę 5-elementową. Spróbuj odczytać wartość nieistniejącego elementu tej tablicy.

```
public class Main {  
    public static void main (String args[]) {  
        int tab[] = new int[5];  
        int value = tab[20];  
        System.out.println("Element nr 20 to: " + value);  
    }  
}
```

Wykonanie powyższego kodu spowoduje oczywiście błąd (rysunek 5.1), ponieważ w tablicy tab nie ma elementu o indeksie 20. W tym przypadku błąd jest ewidentnie widoczny. Gdybyśmy jednak deklarowali tablicę w jednej klasie, a odwoływali się do niej w drugiej, nie byłoby to już tak oczywiste.



Rysunek 5.1. Przekroczenie dopuszczalnego zakresu tablicy

Ć W I C Z E N I E

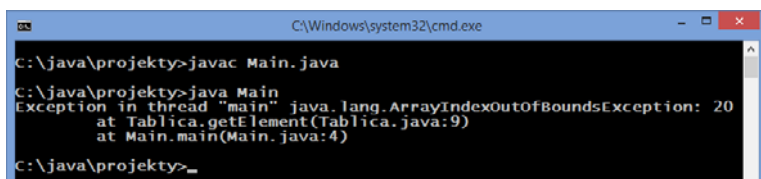
5.2 Nieprawidłowa współpraca dwóch klas

Napisz dwie takie klasy, aby w jednej została zadeklarowana tablica 5-elementowa, a w drugiej następowało odwołanie do tej tablicy.

```
public class Main {  
    public static void main (String args[]) {  
        Tablica tab = new Tablica();  
        int value = tab.getElement(20);  
        System.out.println("Element nr 20 to: " + value);  
    }  
}  
  
public class Tablica {  
    int tab[];  
    public Tablica()  
    {
```

```
        tab = new int[5];
    }
    int getElement(int index)
    {
        return tab[index];
    }
}
```

W tej chwili w klasie Main wywołujemy metodę obiektu typu *Tablica*, nie wiedząc bezpośrednio, jakiej wielkości jest sama *tablica*. Bardzo łatwo jest więc przekroczyć maksymalny indeks i spowodować błąd. Tak też dzieje się w powyższym przykładzie (pamiętajmy o konieczności zapisania klas *Tablica* i *Main* w oddzielnych plikach: *Tablica.java* i *Main.java*). Warto zwrócić uwagę, że tym razem w komunikacie o błędzie znajdzie się więcej informacji (rysunek 5.2). Będzie widać, że usterka z klasy *Main* i metody *main* (plik *Main.java*, wiersz 4.) ma swoje źródło w klasie *Tablica* i metodzie *getElement* (plik *Tablica.java*, wiersz 9.).



Rysunek 5.2. Komunikat zawierający informacje o źródle pochodzenia błędu

Błędów z ćwiczenia 5.2 można uniknąć, sprawdzając, czy wartość podawana jako argument metody *getElement* nie przekracza dopuszczalnego zakresu, czyli przedziału 0 – 4. Takiego sprawdzenia można dokonać, stosując znaną już instrukcję warunkową *if*.

ĆWICZENIE

5.3 Sprawdzanie poprawności zakresu

Kod z ćwiczenia 5.2 zmodyfikuj tak, aby po przekroczeniu dopuszczalnego indeksu tablicy nie występował błąd w programie. Wprowadź do metody *getElement* instrukcję sprawdzającą, czy następuje przekroczenie zakresu indeksów tablicy.

```
public class Main {
    public static void main (String args[]) {
        Tablica tab = new Tablica();
```

```
int value = tab.getElement(20);
if (value == -1){
    System.out.println("Nie ma elementu numer 20!");
}
else{
    System.out.println("Element nr 20 to: " + value);
}
}
}

public class Tablica {
    int tab[];
    public Tablica()
    {
        tab = new int[5];
    }
    int getElement(int index)
    {
        if ((index >= 0) && (index < 5))
            return tab[index];
        else
            return -1;
    }
}
```

W metodzie `getElement` klasy `Tablica` następuje sprawdzenie, czy indeks przekazany jako argument jest właściwy, czyli czy mieści się w zakresie 0 – 4. Do tego używana jest instrukcja warunkowa `if`. Gdy zakres jest prawidłowy, zwracana jest wartość zapisana w tablicy pod tym indeksem, natomiast gdy jest nieprawidłowy, zwracana jest wartość -1. Moglibyśmy wprowadzić zakończyć wykonywanie programu już w metodzie `getElement`, zwykle jednak istnieje potrzeba poinformowania funkcji wywołującej o wystąpieniu błędu.

W funkcji `main` badana jest wartość zwrócona przez wywołanie `tab.getElement(20)`. Jeśli jest to -1, oznacza to, że indeks był nieprawidłowy. Wtedy wyświetlany jest komunikat. Gdy otrzymana wartość jest różna od -1, jest to pobrana wartość elementu. I ta wartość jest wyświetlana.

Rozwiązanie przedstawione w ćwiczeniu 5.3 ma jednak bardzo poważną wadę: otóż żaden z elementów tablicy nie może być równy -1. Sprawdza się więc np. wtedy, gdy przechowujemy wyłącznie wartości nieujemne. Z problemem sygnalizacji błędu moglibyśmy sobie poradzić zatem nieco inaczej — dodając do klasy `Tablica` dodatkowe pole typu `boolean`, np. o nazwie `isError`.

ĆWICZENIE

5.4 Dodatkowe pole obsługi błędów

Kod z ćwiczenia 5.3 zmodyfikuj tak, aby o wystąpieniu błędu informowało dodatkowe pole umieszczone w klasie *Tablica*.

```
public class Main {
    public static void main (String args[]) {
        Tablica tab = new Tablica();
        int value = tab.getElement(20);
        if (tab.isError){
            System.out.println("Nie ma elementu numer 20!");
        }
        else{
            System.out.println("Element nr 20 to: " + value);
        }
    }
}

public class Tablica {
    int tab[];
    boolean isError;
    public Tablica()
    {
        tab = new int[5];
        isError = false;
    }
    int getElement(int index)
    {
        if ((index >=0) && (index < 5)){
            isError = false;
            return tab[index];
        }
        else{
            isError = true;
            return -1;
        }
    }
}
```

Tym razem informacja o tym, czy wystąpił błąd, znajduje się w polu *isError*, któremu w konstruktorze przypisywana jest początkowa wartość *false*. W metodzie *getElement* w standardowy sposób badany jest zakres wartości argumentu *index*. Jeżeli zawiera się on w prawidłowym przedziale, pole *isError* otrzymuje wartość *false* i zwracana jest wartość elementu pobranego z tablicy. W przeciwnym razie pole *isError* otrzymuje wartość *true* (co sygnalizuje wystąpienie błędu) i zwracana jest wartość -1. Jest ona wprawdzie w takim przypadku bezuży-

teczna, ale instrukcja `return` nie może się obyć bez podania wartości typu `int`, gdyż taką wartość musi zwrócić metoda `getElement`.

W klasie `Main`, która korzysta z obiektów typu `Tablica`, po wywołaniu metody `getElement` badany jest stan pola `isError` obiektu `tab`. Gdy jest równe `true` (`if(tab.isError())`), pojawia się komunikat o błędzie, a gdy jest równe `false`, wyświetlana jest wartość pobranego elementu.

Jak widać, z błędami można sobie radzić na różne sposoby, jednak powoduje to dodawanie coraz większej liczby zmiennych i warunków. Warto by więc skorzystać z jakiejś innej metody obsługi. Z pomocą przychodzą tutaj tzw. wyjątki. Wyjątek jest to byt programistyczny, który powstaje w chwili wystąpienia błędu (ogólniej: sytuacji wyjątkowej). Możemy go jednak przechwycić i wykonać nasz własny kod obsługi błędu. Jeżeli tego nie zrobimy, zostanie on obsłużony przez system, a na konsoli¹ zobaczymy wtedy odpowiedni komunikat. W ćwiczeniu 5.1 występował np. wyjątek o nazwie `ArrayIndexOutOfBoundsException`, który nie został przez nas obsłużony, a więc na ekranie pojawił się komunikat (widać to wyraźnie na rysunkach 5.1 i 5.2). Wystąpienie tego wyjątku oznacza, że został przekroczony dopuszczalny zakres indeksu tablicy.

Instrukcja `try...catch`

Do obsługi wyjątków służy blok `try...catch`, którego schemat wykorzystania wygląda następująco:

```
try{
    // blok instrukcji mogący spowodować wyjątek
}
catch(TypWyjtku1 identyfikatorWyjtku1){
    // obsługa wyjątku 1
}
catch(TypWyjtku2 identyfikatorWyjtku2){
    // obsługa wyjątku 2
}
catch(TypWyjtkuN identyfikatorWyjtkuN){
    // obsługa wyjątku N
}
```

¹ O ile program został uruchomiony z konsoli systemowej bądź też konsola Javy została udostępniona w inny sposób.


```
}  
finally{  
    //instrukcje  
}
```

W bloku try powinien się znaleźć ciąg instrukcji mogących spowodować wyjątek. Jeżeli podczas ich przetwarzania zostanie on wygenerowany, wykonywanie instrukcji zostanie przerwane, a sterowanie przekazane do bloku instrukcji catch. Tu z kolei jest sprawdzane, czy któryś z bloków catch odpowiada wygenerowanemu wyjątkowi. Jeśli tak, kod danego bloku zostanie wykonany. Instrukcje znajdujące się po słowie finally wykonywane są zawsze — niezależnie od tego, czy wyjątek wystąpił, czy nie. Blok finally nie jest jednak obligatoryjny i nie musimy go stosować.

Ć W I C Z E N I E

5.5 Użycie bloku try...catch

Zastosuj instrukcję try...catch do przechwycenia wyjątku generowanego przez system w ćwiczeniu 5.2. Wyświetl własny komunikat o błędzie.

```
public class Main {  
    public static void main (String args[]) {  
        Tablica tab = new Tablica();  
        int value = tab.getElement(20);  
        System.out.println("Element nr 20 to: " + value);  
    }  
}  
  
public class Tablica {  
    int tab[];  
    public Tablica()  
    {  
        tab = new int[5];  
    }  
    int getElement(int index)  
    {  
        int val = 0;  
        try{  
            val = tab[index];  
        }  
        catch(ArrayIndexOutOfBoundsException e){  
            System.out.println("Nie ma elementu o numerze 20!");  
            System.exit(-1);  
        }  
        return val;  
    }  
}
```

Klasa Main pozostała w formie z ćwiczenia 5.2. W klasie Tablica została natomiast zmieniona metoda getElement. Instrukcja pobierająca element tablicy o indeksie wskazywanym przez argument index została ujęta w blok try. Dzięki temu jeśli indeks przekroczy dopuszczalny zakres tablicy, wygenerowany wyjątek zostanie przechwycony przez blok catch. Ponieważ będzie to wyjątek typu `ArrayIndexOutOfBoundsException`, taki też typ został zastosowany w bloku catch.

W bloku catch następuje wyświetlenie komunikatu o błędzie oraz zakończenie działania programu. Za tę drugą czynność odpowiada instrukcja `System.exit(-1)` (powoduje zakończenie wykonywania aplikacji i powrót do systemu z kodem powrotu równym -1).

Być może nie widać tego od razu, ale dzięki obsłudze wyjątku `ArrayIndexOutOfBoundsException` zrealizowanej w ćwiczeniu 5.5 bardzo dużo zyskaliśmy. Otóż nie trzeba się teraz przejmować rozmiarem tablicy tab. W poprzednim przypadku, gdy używaliśmy instrukcji warunkowej `if`, musieliśmy znać rozmiar tablicy, a ten może być różny i zmieniać się podczas wykonywania programu². Teraz ta informacja nie jest nam już potrzebna.

Wyjątki w Javie są obiektami, a obiekt wyjątku jest dostępny w bloku catch. W przykładzie z ćwiczenia 5.5 jest on reprezentowany przez symbol `e`. Taki obiekt ma wbudowaną metodę `toString`, która zwraca ciąg znaków będący opisem błędu (o ile taki opis był dostępny przy tworzeniu wyjątku lub został dodany w trakcie jego przetwarzania). Gdybyśmy zatem chcieli oprócz naszego komunikatu o błędzie wyświetlić również ten systemowy, możemy użyć tej metody.

Ć W I C Z E N I E

5.6 Systemowy komunikat o błędzie

Kod klasy Tablica z ćwiczenia 5.5 zmodyfikuj tak, aby dodatkowo był wyświetlany również systemowy komunikat o błędzie.

```
try{
    val = tab[index];
}
```

² Tak naprawdę w Javie tablice są obiektami, które posiadają pole `length`, więc rozmiar jest zawsze znany. To jednak tylko ilustracja wykorzystania wyjątków, przyjmijmy zatem, że nie znamy rozmiarów tablicy.

```
catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Nie ma elementu o numerze 20!");
    System.out.println(e.toString());
    System.exit(-1);
}
```

W celu wykonania ćwiczenia wystarczyło dodać w bloku catch wiersz powodujący wyświetlanie wartości (ciągu znaków) zwróconej przez wywołanie `e.toString()`³.

Zgłaszanie wyjątków

W ćwiczeniu 5.5 pokazano, że można w wygodny sposób przechwycić wyjątek i wykonać własny kod obsługi. Jednym z wcześniejszych założeń było jednak to, że chcemy obsługiwać błąd w funkcji wywołującej metodę `getElement` (czyli w funkcji `main`), a nie od razu kończyć pracę w aplikacji. Czy grozi nam więc powrót do sposobu ze zmienną `isError`, ustawianą tym razem w bloku `catch`? Oczywiście nie. Wyjątek nie musi być obsługiwany bezpośrednio w miejscu jego wystąpienia. Przykładowo jeśli spodziewamy się, że metoda `getElement` może wygenerować błąd, możemy objąć ją klauzulą `try...catch`, przenosząc tym samym obsługę błędów poza klasę `Tablica`.

Ć W I C Z E N I E

5.7 Przechwytywanie wyjątku

Kod z ćwiczenia 5.5 zmodyfikuj tak, aby przechwycenie wyjątku odbywało się w klasie `Main`.

```
public class Main {
    public static void main (String args[]) {
        Tablica tab = new Tablica();
        int value = 0;
        try{
            value = tab.getElement(20);
        }
    }
}
```

³ Prawidłowym zapisem byłby też `System.out.println(e)`, gdyż w takim przypadku zostałaby wykonana niejawna konwersja do typu `String` polegająca właśnie na wykonaniu metody `toString`. To jednak temat wykraczający poza ramy tego rozdziału.

```
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Nie ma elementu o numerze 20!");
            System.exit(-1);
        }
        System.out.println("Element nr 20 to: " + value);
    }
}

public class Tablica {
    int tab[];
    public Tablica()
    {
        tab = new int[5];
    }
    int getElement(int index)
    {
        return tab[index];
    }
}
```

Jak widać, klasa `Tablica` nie zawiera teraz żadnych procedur obsługi błędów. Przechwycenie wyjątku odbywa się w klasie `Main` przy wywołaniu metody `getElement`. A zatem przy próbie odwołania do nieistniejącego elementu tablicy zostanie wygenerowany wyjątek. Ponieważ w metodzie `getElement` w klasie `Tablica` nie ma instrukcji `try...catch`, mogącej go przechwycić, jest on przekazywany do funkcji, która tę metodę wywołała. W tym przypadku jest to funkcja (metoda) `main` z klasy `Main`, w której znajduje się standardowy blok obsługi.

Technika wyjątków nie jest jednak ograniczona jedynie do tych, które są generowane przez system (maszynę wirtualną). Programista sam może spowodować zgłoszenie wyjątku — stosuje się w tym celu instrukcję `throw`. Tej instrukcji należy przekazać nowo utworzony obiekt wyjątku. Trzeba zatem zastosować konstrukcję w postaci:

```
throw new KlasaWyjtku();
```

Ć W I C Z E N I E

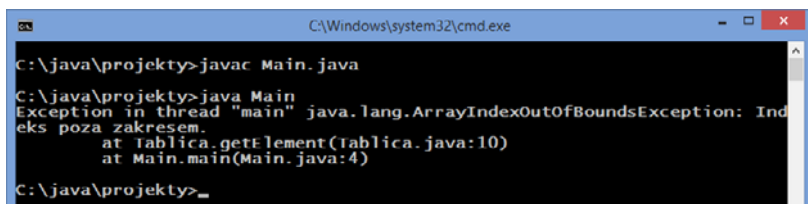
5.8 Generowanie własnego wyjątku

Kod klasy `Tablica` zmodyfikuj w taki sposób, aby w przypadku przekroczenia indeksu tablicy samodzielnie wygenerować wyjątek `ArrayIndexOutOfBoundsException`.

```
public class Tablica {  
    int tab[];  
    public Tablica()  
    {  
        tab = new int[5];  
    }  
    int getElement(int index)  
    {  
        if(index < 0 || index > 4){  
            throw new ArrayIndexOutOfBoundsException("Indeks poza zakresem.");  
        }  
        return tab[index];  
    }  
}
```

Do sprawdzenia, czy został przekroczony indeks tablicy, wykorzystano zwykłą instrukcję warunkową `if`. Jeśli przekroczenie nastąpiło, tworzony jest nowy obiekt typu `ArrayIndexOutOfBoundsException`, który jest zgłaszany maszynie wirtualnej (żargonowo stosuje się też termin *wyrzucanie wyjątku*) za pomocą instrukcji `throw`. Konstruktorowi klasy `ArrayIndexOutOfBoundsException` został przekazany ciąg znaków w postaci argumentu, zawierający dodatkowy opis wyjątku (nie jest to jednak obligatoryjne, konstruktor może być również bezargumentowy).

Jeśli teraz użyjemy takiej klasy `Tablica` wraz z klasą `Main` z ćwiczenia 5.5, zobaczymy, że na ekranie pojawi się komunikat: Indeks poza zakresem. Zaprezentowano to na rysunku 5.3.



Rysunek 5.3. Wyjątek z komunikatem w języku polskim

Hierarchia wyjątków

Wyjątki w Javie są obiektami. Oznacza to, że kiedy wyjątek ma być wygenerowany, tworzony jest obiekt danego typu. W prezentowanych przykładach ten typ to `ArrayIndexOutOfBoundsException`. Dostęp do takiego

obiektu mamy poprzez zadeklarowaną przez nas zmienną odnośnikową `e`. Nie będziemy się tym bliżej zajmować, musimy tylko pamiętać, że jednym z efektów takiego stanu rzeczy jest hierarchia wyjątków. Nie jest zatem obojętne, w jakiej kolejności będziemy je przechwytywać.

Ogólna zasada jest następująca: najpierw przechwytywane są wyjątki bardziej ogólne, a później bardziej szczegółowe. Wyjątki znajdujące się na tym samym poziomie hierarchii mogą być natomiast przechwytywane w dowolnej kolejności.

Ć W I C Z E N I E

5.9 Kolejność przechwytywania wyjątków

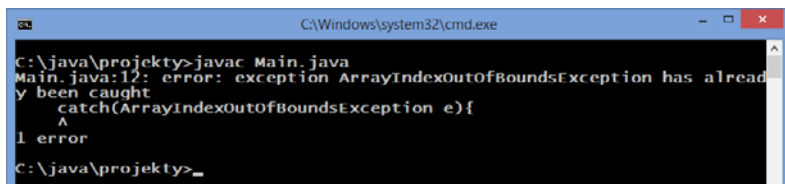
Zmodyfikuj klasę `Main` z ćwiczenia 5.7 w taki sposób, aby najpierw przechwytywany był wyjątek ogólny `Exception`, a następnie wyjątek szczegółowy `ArrayIndexOutOfBoundsException`. Spróbuj dokonać kompilacji otrzymanego kodu.

```
public class Main {
    public static void main (String args[]) {
        Tablica tab = new Tablica();
        int value = 0;
        try{
            value = tab.getElement(20);
        }
        catch(Exception e){
            System.out.println("Jakiś błąd!");
            System.exit(-1);
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Nie ma elementu o numerze 20!");
            System.exit(-1);
        }
        System.out.println("Element nr 20 to: " + value);
    }
}
```

Jak widać na rysunku 5.4, kompilacja się nie udała. Ponieważ klasa wyjątku `ArrayIndexOutOfBoundsException` dziedziczy pośrednio po klasie `Exception`, instrukcja:

```
catch(ArrayIndexOutOfBoundsException e)
```

nie zostałaby nigdy osiągnięta, w przypadku wygenerowania wyjątku `ArrayIndexOutOfBoundsException` zostałby on bowiem przechwycony wcześniej przez instrukcję `catch (Exception e)`.

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt is at "C:\java\projekty>". The user has entered "javac Main.java". The output shows a compilation error: "Main.java:12: error: exception ArrayIndexOutOfBoundsException has already been caught". The error points to a line with a "catch(ArrayIndexOutOfBoundsException e){". Below this, there is another "catch" block for "IOException". The prompt is now at "C:\java\projekty>".

```
C:\Windows\system32\cmd.exe
C:\java\projekty>javac Main.java
Main.java:12: error: exception ArrayIndexOutOfBoundsException has already
been caught
    catch(ArrayIndexOutOfBoundsException e){
    ^
1 error
C:\java\projekty>
```

Rysunek 5.4. Błąd polegający na nieuwzględnieniu hierarchii wyjątków

Aby zatem poprawić kod ćwiczenia tak, aby można było dokonać kompilacji, a program działał poprawnie, należy zamienić miejscami bloki catch. Wówczas najpierw będzie przechwytywany wyjątek `ArrayIndexOutOfBoundsException` (szczegółowy), a dopiero za nim wyjątek `Exception` (ogólny).

6

Operacje wejścia-wyjścia

„Prawdziwa” aplikacja nie obejdzie się bez operacji wejścia-wyjścia. Dzięki nim możemy np. wprowadzać dane z klawiatury czy dokonywać operacji na plikach. W Javie operacje tego typu opierają się na strumieniach, czyli obiektach, z których możemy odczytywać dane (strumień wejściowy) lub je do nich zapisywać (strumień wyjściowy). Standardowo zdefiniowane są trzy strumienie: `System.in`, `System.out` i `System.err`¹. `System.in` to strumień wejściowy, `System.out` — wyjściowy, natomiast `System.err` to strumień związany z obsługą błędów. Ze strumienia `System.out` już korzystaliśmy we wcześniejszych rozdziałach. Używaliśmy wtedy jednej z jego metod — `println` — do wyświetlenia linii znaków na ekranie.

¹ Ściślej rzecz ujmując, chodzi o statyczne obiekty: `in` (typu `InputStream`), `out` (typu `PrintStream`) i `err` (typu `PrintStream`) zdefiniowane w finalnej klasie `System`.

Wyświetlanie danych na ekranie

Sposób wyświetlania danych na ekranie konsoli został przedstawiony na samym początku książki. Teraz już wiadomo, że to, co było nazywane instrukcją `System.out.println`, to w rzeczywistości wywołanie metody `println` obiektu `in` zawartego w klasie `System`. Typem tego obiektu jest `PrintStream` (czyli jest to obiekt klasy `PrintStream` lub — jeszcze dokładniej — obiekt ten jest instancją klasy `PrintStream`), a zatem możliwe jest korzystanie z metod klasy `PrintStream`. Zamiast `println` można więc stosować również metodę `print`. Działa prawie tak samo, z tą różnicą że nie dodaje do wyświetlanego ciągu znaku końca wiersza. To oznacza, że za pomocą `print` można wyświetlić różne dane w jednym wierszu.

ĆWICZENIE

6.1 Wyświetlanie danych w jednym wierszu

Zadeklaruj i zainicjuj kilka zmiennych różnych typów. Wyświetl ich zawartość na ekranie w jednym wierszu. Wartość każdej zmiennej wyświetlaj za pomocą osobnej instrukcji programu.

```
public class Main {  
    public static void main(String args[]) {  
        int liczba1 = 152;  
        char znak = 'b';  
        double liczba2 = 2.54;  
        System.out.print(liczba1);  
        System.out.print(" ");  
        System.out.print(znak);  
        System.out.print(" ");  
        System.out.print(liczba2);  
    }  
}
```

Zadeklarowano trzy zmienne: `liczba1`, `znak` i `liczba2`. Pierwsza jest typu `int`, druga — `char`, a trzecia — `double`. Następnie każda z nich została wyświetlona za pomocą metody `print`, a pomiędzy zmienne zostały wprowadzone znaki spacji. Ponieważ metoda `print` nie dodaje do danych znaku końca wiersza, wszystkie wartości pojawiają się w jednej linii.

Trzeba zauważyć, że w powyższym ćwiczeniu metoda `print` otrzymywała wartości różnych typów. Wiadomości podane w rozdziale 4. pozwalają się domyślić, że w takim razie zapewne mamy do czynienia z przeciążonymi wersjami tej metody. Tak jest w istocie. W klasie `PrintStream` zdefiniowane są wersje metod `print` i `println` dla każdego z typów podstawowych, a także dla typu `Object`² oraz dla tablicy znaków. To z kolei oznacza, że argumentem może być zmienna dowolnego typu.

Jeśli pomiędzy wartości przekazywane metodom `print` i `println` będziemy wstawiać łańcuchy znakowe (ciągi znaków ujęte w cudzysłów), będziemy mogli je dowolnie łączyć, uzyskując tym samym jeden wiersz tekstu z różnymi wartościami.

Ć W I C Z E N I E

6.2 Konwersja danych na ciągi znaków

Kod z ćwiczenia 6.1 zmień tak, aby efekt działania był taki sam, ale by wyświetlanie wszystkich danych odbywało się tylko w jednym wywołaniu metody `print`.

```
public class Main {
    public static void main(String args[]) {
        int liczba1 = 152;
        char znak = 'b';
        double liczba2 = 2.54;
        System.out.print(liczba1 + " " + znak + " " + liczba2);
    }
}
```

Należy jednak podkreślić, że konstrukcja przedstawiona w ćwiczeniu 6.2 zadziała zgodnie z założeniami tylko wtedy, gdy między liczbami znajdują się ciągi znaków. Dzieje się tak dlatego, że gdy kompilator napotka operację dodawania liczby i łańcucha znakowego, nie może wykonać operacji dodawania arytmetycznego. Dlatego najpierw zamienia liczbę w ciąg znaków, a następnie łączy łańcuchy znakowe. To oznacza, że operacja typu `"abc" + 123` jest zamieniana na `"abc" + "123"` i dlatego ostatecznym wynikiem jest `"abc123"`. Nato-

² W przypadku użycia zmiennej referencyjnej jako argumentu metody `print` lub `println` w miejsce argumentu zostanie podstawiony ciąg znaków zwrócony za pomocą metody `toString` obiektu wskazywanego przez tę zmienną.

miast w przypadku dodawania dowolnych typów liczbowych wykonywane jest zwykle dodawanie arytmetyczne. Zostało to zilustrowane w ćwiczeniu 6.3.

ĆWICZENIE

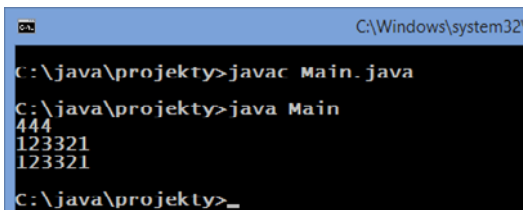
6.3 Liczby i łańcuchy znakowe

Wyświetl na ekranie wynik dodawania dwóch wartości w trzech wersjach: gdy obie są w postaci liczbowej, gdy obie są przedstawione jako łańcuchy znakowe i gdy jedna jest łańcuchem, a druga liczbą.

```
public class Main {  
    public static void main(String args[]) {  
        System.out.println(123 + 321);  
        System.out.println("123" + "321");  
        System.out.println("123" + 321);  
    }  
}
```

W efekcie powstaną trzy ciągi znaków: 444, 123321 i 123321 (rysunek 6.1). W pierwszym przypadku będziemy mieli do czynienia z dodawaniem arytmetycznym, w drugim — łączeniem (konkatenacją) ciągów znaków, a w trzecim — konwersją liczby (321) na łańcuch i łączeniem łańcuchów.

Rysunek 6.1.
*Wynik działania
operatora
dodawania
na różnych
typach danych*



```
C:\Windows\system32  
C:\java\projekty>javac Main.java  
C:\java\projekty>java Main  
444  
123321  
123321  
C:\java\projekty>_
```

Wczytywanie danych z klawiatury

Spróbujmy użyć strumienia wejściowego do wczytywania danych z klawiatury. W JDK starszych niż 1.5 nie jest to niestety tak proste jak w przypadku wyprowadzania danych na ekran. W celu prawidłowego wczytania wiersza tekstu będziemy musieli posłużyć się aż trzema

klasami: `InputStream`, której instancją jest obiekt `System.in`, oraz `InputStreamReader` i `BufferedReader`. `System.in` jest strumieniem bajtowym, czyli pozwala na odczytywanie bajtów. Klasa `InputStreamReader` umożliwia konwersję bajtów na znaki (czyli dzięki niej otrzymamy strumień znakowy), natomiast `BufferedReader` realizuje buforowanie danych.

Ć W I C Z E N I E

6.4 Odczyt pojedynczego wiersza tekstu

Napisz program wczytujący z klawiatury wiersz tekstu i wyświetlający go z powrotem na ekranie.

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        InputStreamReader inp = new InputStreamReader(System.in);
        BufferedReader inbr = new BufferedReader(inp);
        System.out.println("Wprowadź wiersz tekstu:");
        try{
            String line = inbr.readLine();
            System.out.println("Wprowadzony wiersz to:");
            System.out.println(line);
        }
        catch(IOException e){
            System.out.println("Błąd odczytu.");
        }
    }
}
```

Na początku importujemy klasy z pakietu `java.io` — to w nim znajdują się potrzebne nam definicje. Tworzymy następnie dwa obiekty: `inp` klasy `InputStreamReader` oraz `inbr` klasy `BufferedReader`. W celu utworzenia obiektu `inp` powiązanego ze standardowym strumieniem wejściowym konstruktorowi klasy `InputStreamReader` przekazaliśmy obiekt `System.in`. Obiekt `inp` został natomiast wykorzystany jako parametr konstruktora klasy `BufferedReader`. Po wykonaniu tych czynności można korzystać z metody `readLine` obiektu `inbr` (klasy `BufferedReader`), która zwraca odczytany ze strumienia wejściowego wiersz tekstu (czyli ciąg znaków zakończony znakiem końca linii). Ciąg jest zapisywany w zmiennej pomocniczej (`line`) typu `String` (czyli takiej, która może przechowywać ciągi znaków).

Instrukcja wywołująca metodę `readLine` jest ujęta w blok `try`, dzięki któremu możemy obsłużyć sytuację, gdy przy próbie odczytu danych wystąpi błąd. Byłby to błąd typu `IOException` (błąd wejścia-wyjścia), dlatego też przechwytywany jest wyjątek tego właśnie typu. Jeśli błąd wystąpi, zostanie wyświetlony stosowny komunikat.

Należy zauważyć, że użyta w ćwiczeniu 6.4 zmienna `inr` jest wyłącznie zmienną pomocniczą. Dzięki niej kod jest bardziej czytelny. W praktyce często obiekt typu `BufferedReader` jest tworzony w jednej instrukcji, która w tym przypadku wyglądałaby następująco:

```
BufferedReader inbr = new BufferedReader(  
    new InputStreamReader(System.in)  
);
```



W JDK istnieje również klasa `DataInputStream` zawierająca metodę `readLine`, która w założeniu ma wykonywać takie samo zadanie. Nie należy jednak stosować tej metody, gdyż może ona niepoprawnie konwertować bajty ze strumienia na znaki.

Ć W I C Z E N I E

6.5 Wczytywanie danych w pętli

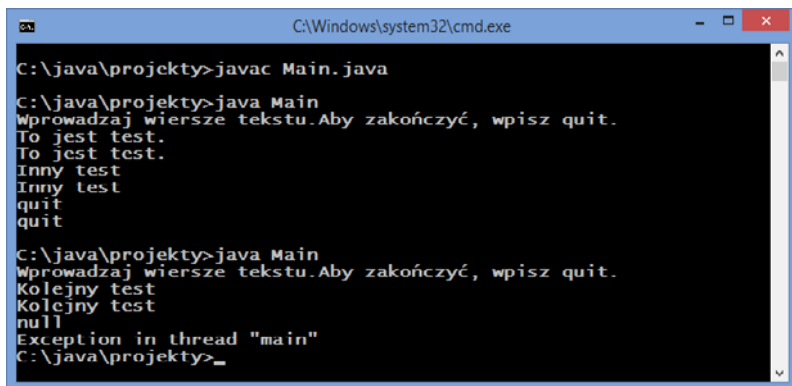
Napisz program, który w pętli wczytuje kolejne wiersze tekstu i wyprowadza je na ekran. Program ma zakończyć działanie w momencie wprowadzenia z klawiatury ciągu znaków "quit".

```
import java.io.*;  
  
public class Main {  
    public static void main(String args[]) {  
        String line = "";  
        BufferedReader inbr = new BufferedReader(  
            new InputStreamReader(System.in)  
        );  
        System.out.println("Wprowadzaj wiersze tekstu." +  
            "Aby zakończyć, wpisz quit.");  
        try{  
            while (!line.equals("quit")){  
                line = inbr.readLine();  
                System.out.println(line);  
            }  
        }  
        catch(IOException e){  
            System.out.println("Błąd odczytu.");  
        }  
    }  
}
```

```
    }  
  }  
}
```

Dane są wprowadzane za pomocą metody `readLine` obiektu `BufferedReader` powiązanego za pośrednictwem klasy `InputStreamReader` ze standardowym strumieniem wejściowym `System.in`. Odczyt odbywa się w pętli `while`, a uzyskany ciąg znaków (wiersz tekstu) jest zapisywany w zmiennej `line`. Pętla skończy się, gdy zmienna `line` będzie zawierała ciąg `quit`. Do porównywania bieżącej wartości `line` z `quit` jest używana metoda `equals`³ (nie należy porównywać ciągów znakowych za pomocą operatorów `==` i `!=`). Użyty warunek (korzystający z operatora negacji logicznej `!`) `!line.equals("quit")` jest prawdziwy, gdy `line` nie zawiera ciągu `quit`, a fałszywy, gdy `line` zawiera ciąg `quit`.

Taki program będzie działał poprawnie, jeśli jednak strumień wejściowy zostanie przerwany, np. przez wciśnięcie kombinacji klawiszy `Ctrl+C` (lub `Ctrl+Z+Enter`), wówczas wygeneruje wyjątek `NullPointerException`. Dzieje się tak dlatego, że przy przerwaniu strumienia metoda `readLine` zamiast ciągu znaków zwraca wartość `null` (widać to na rysunku 6.2). W ramach zadania do samodzielnego wykonania można się zastanowić, jak zapobiegać tego typu sytuacji. Jedno z rozwiązań zostanie przedstawione w ćwiczeniu 6.13.



```
C:\Windows\system32\cmd.exe  
  
C:\java\projekty>javac Main.java  
C:\java\projekty>java Main  
Wprowadzaj wiersze tekstu.Aby zakończyć, wpisz quit.  
To jest test.  
To jest test.  
Inny test  
Inny test  
quit  
quit  
  
C:\java\projekty>java Main  
Wprowadzaj wiersze tekstu.Aby zakończyć, wpisz quit.  
Kolejny test  
Kolejny test  
null  
Exception in thread "main"  
C:\java\projekty>
```

Rysunek 6.2. Nagłe przerwanie strumienia spowodowało błąd programu

³ Jest to możliwe, ponieważ zmienna `line` jest typu `String`, a klasa `String` (reprezentująca ciągi znaków) zawiera metodę `equals`.

Jak wspomniano w opisie ćwiczenia 6.5, do porównywania łańcuchów znakowych nie należy używać operatorów porównywania `==` i `!=` (choć mogłoby się to wydawać oczywistą czynnością). Dlaczego nie należy stosować konstrukcji typu `if (line != "quit")`?

Otóż znana nam instrukcja warunkowa `if` wraz z operatorem porównywania `==` lub `!=` po prostu nie zadziała zgodnie z intuicyjnymi oczekiwaniami (można się o tym przekonać, wprowadzając ją do kodu z ćwiczenia 6.5). Dzieje się tak, gdyż w takich przypadkach po obu stronach operatora znajdują się referencje do obiektów przechowujących ciągi znaków (ciąg znaków umieszczony w cudzysłowie jest traktowany jako obiekt typu `String`). Warto podkreślić — są to referencje do obiektów, a nie same ciągi znaków! Pisząc zatem np.:

```
if (line != "quit")
```

porównujemy dwie referencje, a nie dwa napisy. Dlatego w takich sytuacjach stosujemy do porównywania metodę `equals` klasy `String`.

Umiemy już wczytywać ciągi znaków, pozostaje nauczyć się, w jaki sposób wczytywać liczby. Można zrobić to, wczytując linię tekstu jak w poprzednich przykładach, a następnie konwertując tekst na liczbę. Zamiast jednak wykonywać taką konwersję ręcznie, lepiej skorzystać z wyspecjalizowanej klasy `StreamTokenizer`. Dzieli ona strumień wejściowy na tokeny, czyli jednostki leksykalne. Dla nas jednak najważniejsze w tej chwili jest to, że można w ten sposób rozpoznawać liczby.

Ć W I C Z E N I E

6.6 Wczytywanie wartości liczbowych

Napisz program wczytujący z klawiatury dwie liczby, a następnie wyświetlający je na ekranie. Skorzystaj z klasy `StreamTokenizer`.

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        double a, b;
        Reader r = new BufferedReader(
            new InputStreamReader(System.in)
        );
        StreamTokenizer inp = new StreamTokenizer(r);

        try{
            System.out.println("Podaj a:");
            inp.nextToken();
```



```
a = inp.nval;

System.out.println("Podaj b:");
inp.next token();
b = inp.nval;

System.out.println("Podane wartości to a = " + a +
                    ", b = " + b);
}
catch (IOException e){
    System.out.println("Błąd odczytu!");
}
}
```

Najpierw powstał obiekt klasy `BufferedReader` powiązany ze standardowym strumieniem wejściowym (utworzono go w sposób analogiczny do przedstawionego w ćwiczeniu 6.5), a następnie został użyty jako argument konstruktora klasy `StreamTokenizer`. Tak powstał obiekt `inp` typu `StreamTokenizer`, również powiązany ze standardowym strumieniem wejściowym.

Za pomocą metody `nextToken` zostały pobrane dwie kolejne jednostki leksykalne. Ich wartości — o ile były to liczby — zostały odczytane z pola `nval` (zdefiniowanego w klasie `StreamTokenizer`) i zapisane w zmiennych `a` i `b`. Wartości tych zmiennych zostały następnie w sposób standardowy wyświetlone na ekranie.

Testując program z ćwiczenia 6.6, można zauważyć, że przy takim rozwiązaniu występują dwa problemy. Po pierwsze pole `nval` jest typu `double`, zatem otrzymujemy w wyniku liczbę zmiennoprzecinkową podwójnej precyzji. Co zrobić, jeśli potrzebujemy wartości całkowitej? Oczywiście rozwiązaniem jest tu konwersja typów. Jeśli zatem zmienne `a` i `b` byłyby typu `int`, to należałoby dokonywać przypisań w następujący sposób:

```
a = (int) inp.nval;
b = (int) inp.nval;
```

Drugi problem jest poważniejszy. Co się stanie, jeśli przy wprowadzaniu danych nie podamy liczby, ale dowolny inny ciąg znaków? Oczywiście program nie będzie działał poprawnie. Nie wystąpi wprawdzie żaden błąd krytyczny, ale pole `nval` będzie zawierało wartość 0.0. Może to spowodować trudną do wykrycia usterkę w dalszej części

aplikacji. Ustrzeże nas przed tym sprawdzenie stanu statycznego⁴ pola `TT_NUMBER` klasy `StreamTokenizer`, które wskazuje, czy ostatnio pobrany token jest liczbą.

Ć W I C Z E N I E

6.7 Wykrywanie rodzaju wprowadzonej wartości

Napisz program wczytujący z klawiatury dwie liczby całkowite, a następnie wyświetlający je na ekranie. W przypadku gdy użytkownik nie poda liczby, aplikacja ma ponawiać prośbę o jej podanie.

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        int a, b;
        Reader r = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer inp = new StreamTokenizer(r);

        try{
            System.out.println("Podaj liczbę a:");
            while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
                System.out.println("Podaj poprawną liczbę a:");
            }
            a = (int) inp.nval;

            System.out.println("Podaj liczbę b:");
            while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
                System.out.println("Podaj poprawną liczbę b:");
            }

            b = (int) inp.nval;
            System.out.println("Podane wartości to: a = " + a +
                               ", b = " + b);
        }
        catch (IOException e){
            System.out.println("Błąd odczytu!");
        }
    }
}
```

Obiekty klas `BufferedReader` oraz `StreamTokenizer` są tworzone analogicznie do poprzedniego ćwiczenia. Główna różnica jest taka, że metoda `nextToken` jest wywoływana w pętli typu `while` i dzieje się tak

⁴ Statyczność pola oznacza, że jest ono wspólne dla wszystkich instancji (wystąpień obiektów) danej klasy oraz że istnieje zawsze, i to niezależnie od tego, czy został utworzony jakiś obiekt tej klasy. To zagadnienie wykracza jednak poza ramy tematyczne książki.

dopóty, dopóki zwrócona przez nią wartość jest różna od wartości zapisanej w `StreamTokenizer.TT_NUMBER`. A zatem taka pętla zakończy się dopiero wtedy, gdy odczytana zostanie poprawna wartość liczbową. W sytuacji gdy nie została wprowadzona prawidłowa wartość, na ekranie wyświetlany będzie odpowiedni komunikat. Jeśli wprowadzony ciąg jest liczbą, następuje zakończenie pętli, a wartość jest konwertowana do typu `int` i przypisywana właściwej zmiennej.

Ć W I C Z E N I E

6.8 Wykorzystanie wczytywanych danych do obliczeń

Napisz program obliczający pierwiastki równania kwadratowego. Wczytaj parametry równania z klawiatury.

```
import java.io.*;

public class Main {
    public static void main (String args[]) {
        double parametrA = 0, parametrB = 0, parametrC = 0;

        Reader r = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer inp = new StreamTokenizer(r);

        try{
            System.out.print("Podaj parametr A: ");
            while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
                System.out.print("Podaj poprawny parametr A: ");
            }
            parametrA = inp.nval;

            System.out.print("Podaj parametr B: ");
            while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
                System.out.print("Podaj poprawny parametr B: ");
            }
            parametrB = inp.nval;

            System.out.print("Podaj parametr C: ");
            while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
                System.out.print("Podaj poprawny parametr C: ");
            }
            parametrC = inp.nval;
        }
        catch (IOException e){
            System.out.println("Błąd odczytu!");
            System.exit(-1);
        }
        System.out.println("Parametry równania:");
```

```
System.out.print("A: " + parametrA + ", B: " + parametrB +
                ", C: " + parametrC + "\n");

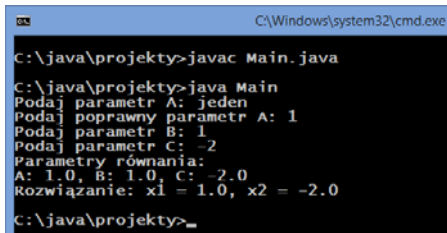
if (parametrA == 0){
    System.out.println ("To nie jest równanie kwadratowe: A = 0!");
}
else{
    double delta = parametrB * parametrB - 4 * parametrA * parametrC;
    double wynik;

    if (delta < 0){
        System.out.println ("Delta < 0.");
        System.out.println ("To równanie nie ma rozwiązania "+
                            "w zbiorze liczb rzeczywistych.");
    }
    else if (delta == 0){
        wynik = -parametrB / (2 * parametrA);
        System.out.println ("Rozwiązanie: x = " + wynik);
    }
    else{
        wynik = (-parametrB + Math.sqrt(delta)) / (2 * parametrA);
        System.out.print ("Rozwiązanie: x1 = " + wynik);
        wynik = (-parametrB - Math.sqrt(delta)) / (2 * parametrA);
        System.out.println (" , x2 = " + wynik);
    }
}
}
```

Postać kodu z tego ćwiczenia nie powinna być żadnym zaskoczeniem. Obliczenia są wykonywane analogicznie do przypadku z ćwiczenia 2.17 w rozdziale 2. Nowością jest wprowadzanie parametrów równania z klawiatury (rysunek 6.3), podczas gdy poprzednio były one na stałe zapisane w kodzie. Odczyt danych z klawiatury odbywa się jednak na tej samej zasadzie jak w ćwiczeniu 6.7. Cała aplikacja jest więc swoistym połączeniem koncepcji z przykładów 2.17 i 6.7.

Rysunek 6.3.

*Obliczanie
rozwiązań
równania
o parametrach
wprowadzanych
z klawiatury*



```
C:\Windows\system32\cmd.exe
c:\java\projekty>javac Main.java
c:\java\projekty>java Main
Podaj parametr A: jeden
Podaj poprawny parametr A: 1
Podaj parametr B: 1
Podaj parametr C: -2
Parametry równania:
A: 1.0, B: 1.0, C: -2.0
Rozwiązanie: x1 = 1.0, x2 = -2.0
c:\java\projekty>
```

Nowe sposoby wprowadzania danych

Jeśli dysponujemy JDK w wersji co najmniej 1.5, możemy również skorzystać z innego sposobu przetwarzania danych ze strumienia wejściowego. Pozwala na to klasa `Scanner` z pakietu `java.util`. Ma ona bardzo wiele możliwości, ale my skupimy się tylko na sposobie wczytania za jej pomocą liczb całkowitych z klawiatury. Umożliwia to konstrukcja w postaci:

```
Scanner sc = new Scanner(System.in);
int liczba = sc.nextInt();
```

W tak prostej formie spowoduje ona jednak, że wprowadzenie ciągu, który nie reprezentuje liczby całkowitej, wygeneruje wyjątek. Jeśli chcemy temu zapobiec, możemy zastosować pętlę `while` oraz metody `hasNextInt` i `next`. Pierwsza z nich zwraca wartość `true`, jeśli kolejny token reprezentuje wartość całkowitą, a `false` w przeciwnym wypadku, natomiast druga powoduje pobranie kolejnego tokena niezależnie od jego typu i zwrócenie go w postaci ciągu znaków (może być więc użyta do pominięcia tokena).

ĆWICZENIE

6.9 Użycie klasy `Scanner`

Korzystając z klasy `Scanner`, napisz aplikację, która wczyta z klawiatury liczbę całkowitą i wyświetli ją na ekranie. W przypadku wprowadzenia ciągu znaków, który nie reprezentuje takiej liczby, ponawiaj prośbę o jej podanie.

```
import java.util.*;

public class Main {
    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Podaj liczbę całkowitą: ");
        while(!sc.hasNextInt()){
            System.out.print("To nie jest liczba całkowita. ");
            System.out.print("Podaj liczbę całkowitą: ");
            sc.next();
        }
        int i = sc.nextInt();
        System.out.println("Wprowadzona liczba to: " + i);
    }
}
```

Konstruktorowi klasy `Scanner` został przekazany obiekt reprezentujący standardowy strumień wejściowy. Pętla `while` została użyta do pominięcia wszystkich tokenów, które nie są liczbą całkowitą. Warunkiem jej zakończenia jest bowiem `!sc.hasNextInt()`. Trwa więc tak długo, aż wartość zwrócona przez metodę `hasNextInt` będzie równa `true`, czyli gdy na wejściu pojawi się liczba całkowita. Dopóki liczby całkowitej nie ma, dopóty wyświetlany jest komunikat o braku wartości całkowitej i wykonywana jest metoda `next` obiektu `sc` typu `Scanner`, a tym samym bieżący token (np. ciąg znaków) jest pomijany (wywołanie `sc.next()` powoduje pobranie i zwrócenie kolejnego tokena, ale ponieważ nigdzie go nie używamy, jest to równoznaczne z jego pominięciem).

Oczywiście jeżeli od razu zostanie wprowadzona wartość całkowita, warunek `!sc.hasNextInt()` będzie fałszywy, a więc pętla się nie wykona. Zostaną natomiast wykonane dalsze instrukcje. Zmiennnej `i` zostanie przypisana wartość zwrócona przez metodę `nextInt`, która pobiera ze strumienia wartość typu `int`. Wartość ta zostanie też wyświetlona na ekranie.

Warto w tym miejscu zwrócić uwagę, że program z ćwiczenia 6.9 nie jest odporny na przerwanie strumienia wejściowego (np. przez kombinację *Ctrl+C* lub *Ctrl+Z+Enter*; podobna sytuacja miała miejsce w ćwiczeniu 6.5). Wygenerowany zostanie wtedy wyjątek `NoSuchElementException`. Odpowiednia poprawka pozostanie jednak ćwiczeniem do samodzielnego wykonania (można się wzorować na rozwiązaniu z ćwiczenia 6.11).

Ć W I C Z E N I E

6.10 Sumowanie wprowadzanych wartości

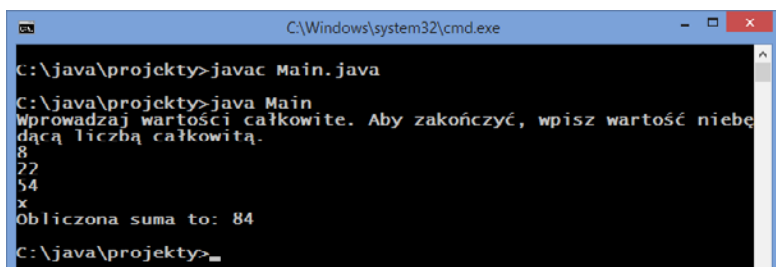
Korzystając z możliwości klasy `Scanner`, napisz program, który będzie sumował wprowadzane wartości całkowite. Program ma zakończyć działanie, gdy na wejściu zostanie podana wartość inna niż całkowita (rysunek 6.4).

```
import java.util.*;

public class Main {
    public static void main(String args[]) {
        int suma = 0;
```

```
Scanner sc = new Scanner(System.in);
System.out.print("Wprowadzaj wartości całkowite. Aby zakończyć,");
System.out.println(" wpisz wartość niebędącą liczbą całkowitą.");
while(sc.hasNextInt()){
    suma += sc.nextInt();
}
System.out.println("Obliczona suma to: " + suma);
}
```

Na początku kodu tworzony jest obiekt `sc` typu `Scanner` powiązany ze standardowym strumieniem wejściowym oraz zmienna `suma`, która będzie przechowywała sumę wprowadzonych wartości całkowitych. Warunek pętli `while` to wynik działania metody `hasNextInt` obiektu `sc`, a zatem instrukcje wewnątrz pętli będą wykonywane, dopóki na wejściu nie pojawi się token niereprezentujący wartości całkowitej. Umieszczona wewnątrz pętli instrukcja `suma += sc.nextInt()` powoduje z kolei dodanie do zmiennej `suma` wartości całkowitej odczytanej ze strumienia wejściowego.



```
C:\Windows\system32\cmd.exe

C:\java\projekty>javac Main.java
C:\java\projekty>java Main
Wprowadzaj wartości całkowite. Aby zakończyć, wpisz wartość niebę
dącą liczbą całkowitą.
8
22
54
x
Obliczona suma to: 84
C:\java\projekty>
```

Rysunek 6.4. Program sumujący wartości całkowite wprowadzane z klawiatury

Możliwości klasy `Scanner` nie ograniczają się oczywiście do przetwarzania wyłącznie liczb całkowitych. Są w niej zdefiniowane metody o konstrukcji `hasNextTYP` oraz `nextTYP`, przetwarzające liczby całkowite, zmiennoprzecinkowe, bajty itp. Przykładowo dla liczb zmiennoprzecinkowych o podwójnej precyzji dostępne są metody `hasNextDouble` i `nextDouble`. Prezentowaną wcześniej aplikację rozwiązującą równania kwadratowe moglibyśmy zatem napisać również za pomocą klasy `Scanner`.

Ć W I C Z E N I E

6.11 Klasa Scanner w realnej aplikacji

Wykonaj zadanie z ćwiczenia 6.8, używając klasy Scanner do wczytywania wartości z konsoli.

```
import java.util.*;
import java.io.*;

public class Main {
    public static void main (String args[]) {
        double parametrA = 0, parametrB = 0, parametrC = 0;

        Scanner sc = new Scanner(System.in);

        try{
            System.out.print("Podaj parametr A: ");
            while(!sc.hasNextDouble()){
                System.out.print("Podaj poprawny parametr A: ");
                sc.next();
            }
            parametrA = sc.nextDouble();

            System.out.print("Podaj parametr B: ");
            while(!sc.hasNextDouble()){
                System.out.print("Podaj poprawny parametr B: ");
                sc.next();
            }
            parametrB = sc.nextDouble();

            System.out.print("Podaj parametr C: ");
            while(!sc.hasNextDouble()){
                System.out.print("Podaj poprawny parametr C: ");
                sc.next();
            }
            parametrC = sc.nextDouble();
        }
        catch (Exception e){
            System.out.println("Błąd odczytu!");
            System.exit(-1);
        }
        // tutaj dalszy kod klasy z ćwiczenia 6.8
    }
}
```

Przedstawiony kod jest połączeniem koncepcji przedstawionych w ćwiczeniu 6.8 oraz przykładów ilustrujących użycie klasy Scanner. Funkcje wczytywania korzystające z klas pośredniczących `BufferedReader` i `InputStreamReader` oraz z klasy przetwarzającej dane (jednostki

leksykalne) `StreamTokenizer` zostały zrealizowane za pośrednictwem obiektu `sc` typu `Scanner`. Zastosowany został sposób odczytu liczb rzeczywistych podobny do przedstawionego w ćwiczeniu 6.9, z tą różnicą że użyto metod `hasNextDouble` i `nextDouble`. Część kodu związana z wykonaniem obliczeń pozostała taka sama jak w ćwiczeniu 6.8, dlatego też pominięto ją na listingu. Wszystkie operacje pobierające i przetwarzające dane zostały ujęte w blok `try...catch`, przechwytyjący wyjątki, które mogłyby się pojawić podczas tych operacji. W przypadku wystąpienia błędu (np. związanego z przerwaniem działania aplikacji odpowiednią kombinacją klawiszy) użytkownik zobaczy więc stosowny komunikat.

Obsługa konsoli

Przedstawione w dwóch pierwszych podrozdziałach sposoby wczytywania i wyświetlania danych za pomocą standardowych strumieni (wejściowego i wyjściowego) mają tę zaletę, że są standardowe i mogą być zastosowane do dowolnych strumieni, np. również plikowych (zostanie to pokazane w dalszej części rozdziału). W Java 6 (1.6) pojawiła się jednak dodatkowo klasa wyspecjalizowana w obsłudze konsoli. Jej nazwa to — jakżeby inaczej — `Console`. Dzięki temu możemy pobrać odwołanie do konsoli, na której operuje Java (o ile taka jest udostępniona; nie musi to być konsola systemowa), i pobierać oraz wyświetlać dane.

Klasa `Console` jest dostępna w pakiecie `java.io`. Aby uzyskać obiekt reprezentujący konsolę, używamy konstrukcji w postaci:

```
Console nazwa = System.console();
```

Jest to więc wywołanie metody `console` obiektu `System`.

Najczęściej używane są trzy metody z klasy `Console`:

- ❑ `readLine` — wczytuje wiersz tekstu (może wyświetlać tekst zachęty),
- ❑ `readPassword` — wczytuje wiersz tekstu z maskowaniem,
- ❑ `printf` — wyświetla dane z uwzględnieniem różnych sposobów formatowania.

Ć W I C Z E N I E

6.12 Dostęp do konsoli

Napisz program wczytujący z konsoli wiersz tekstu i wyświetlający go na ekranie. Do obsługi wejścia-wyjścia użyj klasy `Console`.

```
import java.io.Console;

public class Main {
    public static void main (String args[]) {
        Console con = System.console();
        if(con == null){
            System.out.println("Brak konsoli!");
            System.exit(-1);
        }
        String line = con.readLine("Wprowadź tekst: ");
        con.printf("Wprowadzony tekst: " + line);
    }
}
```

Odwołanie do konsoli (obiekt klasy `Console`) zostało pobrane w sposób opisany wyżej — za pomocą wywołania metody `console` obiektu `System`. Ponieważ w przypadku niedostępności konsoli metoda `console` zwraca wartość `null`, zostało to sprawdzone w instrukcji warunkowej `if`. Gdyby zatem nie można było korzystać z konsoli, program zakończy działanie z kodem powrotu równym `-1`⁵. Jeżeli jednak reprezentujący konsolę obiekt `con` jest różny od `null`, wywoływana jest metoda `readLine`, której w postaci argumentu został przekazany tekst zachęty. Metoda `readLine` zwróci tekst (ciąg znaków) wprowadzony przez użytkownika i zapisze go w zmiennej `line` (w przypadku osiągnięcia końca strumienia będzie to wartość `null`), a zawartość tej zmiennej zostanie wyprowadzona na ekran za pomocą metody `printf`.

Ć W I C Z E N I E

6.13 Wczytywanie danych w pętli

Korzystając z klasy `Console`, napisz program, który będzie wczytywał z konsoli wiersze tekstu i wyświetlał je z powrotem na ekranie. Program ma zakończyć działanie po wprowadzeniu ciągu `quit`.

```
import java.io.Console;
```

⁵ Zamiast strumienia wyjściowego `out` właściwsze byłoby w takim przypadku użycie strumienia błędów `err` (`System.err.println("Brak konsoli!");`).

```
public class Main {  
    public static void main (String args[]) {  
        Console con = System.console();  
        if(con == null){  
            System.out.println("Brak konsoli!");  
            System.exit(-1);  
        }  
        String line = "";  
        while (!"quit".equals(line)){  
            line = con.readLine("Wprowadź tekst: ");  
            con.printf("Wprowadzony tekst: " + line + "\n");  
        }  
    }  
}
```

Obiekt powiązany z konsolą został utworzony w sposób opisany w ćwiczeniu 6.12. Odczyt wierszy tekstu odbywa się w pętli `while`. Wprowadzane przez użytkownika ciągi znaków są odczytywane przez metodę `readLine` i zapisywane w zmiennej `line`. Następnie tekst zmiennej `line` jest łączony za pomocą operatora `+` z napisem `Wprowadzony tekst:` oraz znakiem końca linii `\n` (taka technika łączenia ciągów znaków była już stosowana wielokrotnie) i wyprowadzany na ekran za pomocą metody `printf`.

Wyrażenie warunkowe w pętli `while` bada, czy ciąg znaków zawarty w `line` odpowiada ciągowi `quit`. Została tu zastosowana przedstawiona w ćwiczeniu 6.5 metoda `equals`. To badanie jest jednak „odwrócone” w stosunku do przykładu 6.5, dzięki czemu unikamy wygenerowania wyjątku, w przypadku gdy zmienna `line` zawiera wartość `null`. Zapis `"quit".equals(line)` jest możliwy, ponieważ każdy ciąg znaków ujęty w cudzysłów może być traktowany jako obiekt typu `String`, można więc na jego rzecz wywołać dowolną metodę z klasy `String` (w tym także zastosowaną metodę `equals`)⁶.

⁶ W takim rozwiązaniu, w przypadku gdy `line` jest równe `null`, wartość ta zostanie wyświetlona na ekranie (aby to sprawdzić, wystarczy przerwać działanie aplikacji, wciskając kombinację klawiszy `Ctrl+C`). Poprawka likwidująca ten defekt jest jednak tak prosta, że można ją pozostawić jako ćwiczenie do samodzielnego wykonania.

W ćwiczeniach 6.12 i 6.13 metoda `printf` była używana tak samo jak `println` z klasy `PrintStream` (w wywołaniach `System.out.println`). Ma jednak dużo większe możliwości. Potrafi formatować łańcuchy znakowe. Jeżeli w tekście, który chcemy wyświetlić, umieścimy znaczniki formatujące, a wartości, które chcemy wyświetlić, podamy jako kolejne argumenty, będziemy mogli prezentować dane w rozmaity sposób.

Znacznik formatujący składa się ze znaku `%` oraz litery określającej typ wartości⁷, np. `%s` oznacza ciąg znaków, a `%d` wartość całkowitą. Dopuszczalne symbole zostały przedstawione w tabeli 6.1. Schematyczna postać wywołania metody `printf` będzie wtedy następująca (zakładając, że obiekt `con` reprezentuje konsolę):

```
con.printf("format", wartość1, wartość2, ..., wartośćN);
```

Tabela 6.1. Znaczniki formatujące dla metody `printf`

Ciąg	Znaczenie
b lub B	Argument będzie traktowany jako wartość boolowska.
h lub H	Ciąg będący wynikiem wykonania funkcji skrótu (hash code) argumentu.
s lub S	Argument będzie traktowany jako ciąg znaków.
c lub C	Argument będzie traktowany jako znak.
d	Argument będzie traktowany jako wartość całkowita dziesiętna.
o	Argument będzie traktowany jako wartość całkowita ósemkowa.
x lub X	Argument będzie traktowany jako wartość całkowita szesnastkowa.
e lub E	Argument będzie traktowany jako wartość rzeczywista w notacji naukowej (wykładniczej).
f	Argument będzie traktowany jako wartość rzeczywista dziesiętna.
g lub G	Argument będzie traktowany jako wartość rzeczywista dziesiętna w notacji zwykłej lub naukowej — w zależności od zastosowanej precyzji.

⁷ W rzeczywistości ciąg formatujący może mieć dużo bardziej złożoną postać i określać np. precyzję, z jaką ma być wyświetlona wartość. Dokładne dane można znaleźć w dokumentacji technicznej JDK.

Tabela 6.1. Znaczniki formatujące dla metody `printf` — ciąg dalszy

Ciąg	Znaczenie
a lub A	Argument będzie traktowany jako wartość rzeczywista szesnastkowa.
t lub T	Argument będzie traktowany jako data i (lub) czas. Wymaga określenia dodatkowego formatowania.
%	Znak %.
n	Znak nowego wiersza.

Jeżeli zatem w kodzie zdefiniujemy przykładowe zmienne:

```
boolean zmienna1 = true;  
int zmienna2 = 254;
```

to ich wartości możemy wpisać w wyświetlany na ekranie ciąg znaków w następujący sposób:

```
con.printf("zmienna1 = %b, zmienna2 = %d", zmienna1, zmienna2);
```

Wtedy w miejsce ciągu `%b` zostanie podstawiona wartość zmiennej `zmienna1`, a pod `%d` — wartość zmiennej `zmienna2` (oczywiście zmienna `con` musi zawierać odwołanie do konsoli).

ĆWICZENIE

6.14 Formatowanie ciągów wyjściowych

Napisz program wczytujący z konsoli liczbę całkowitą i wyświetlający ją w postaci dziesiętnej, ósemkowej i szesnastkowej. Użyj klasy `Console`. W przypadku gdy odczytana wartość nie może być interpretowana jako wartość całkowita, program ma ponawiać prośbę o podanie właściwych danych.

```
import java.io.Console;  
  
public class Main {  
    public static void main (String args[]) {  
        Console con = System.console();  
        if(con == null){  
            System.out.println("Brak konsoli!");  
            System.exit(-1);  
        }  
        String line = "";  
        int liczba;
```

```
con.printf("Podaj liczbę całkowitą: ");
while(true){
    line = con.readLine();
    try{
        liczba = Integer.parseInt(line);
    }
    catch(NumberFormatException e){
        con.printf("To nie jest liczba całkowita. ");
        con.printf("Podaj liczbę całkowitą: ");
        continue;
    }
    break;
}

con.printf("Dziesiętnie: %d\n", liczba);
con.printf("Ósemkowo: %o\n", liczba);
con.printf("Szesnastkowo: %x\n", liczba);
}
```

W tym przykładzie został zastosowany nowy sposób odczytu wartości typu `int`. Warunkiem pętli `while` jest `true`, czyli trwa ona dopóty, dopóki nie zostanie przerwana za pomocą instrukcji `break`, a ta instrukcja jest wykonywana dopiero wtedy, gdy wprowadzona przez użytkownika wartość może być przetworzona na liczbę całkowitą. Wprowadzane linie tekstu są odczytywane standardowo za pomocą metody `readLine` i zapisywane w zmiennej `line`. Później następuje próba przekonwertowania ciągu znaków zapisanego w `line` na wartość całkowitą. Jest do tego używana metoda `parseInt` z klasy `Integer`:

```
liczba = Integer.parseInt(line);
```

Rezultatem wywołania jest wartość całkowita, zapisywana w zmiennej `liczba`. W przypadku gdy ciąg zapisany w `line` nie może być przekształcony w liczbę, generowany jest wyjątek `NumberFormatException`, który jest przechwytywany w instrukcji `catch`. Jeśli zatem wyjątek się pojawi, oznacza to, że wprowadzone dane są nieprawidłowe. Wtedy wyświetlany jest stosowny komunikat, a pętla kontynuowana za pomocą instrukcji `continue`.

Ciągi formatujące w metodzie `printf` zostały skonstruowane zgodnie z przedstawionym wyżej opisem. Znacznik `%n` to znak nowego wiersza. Zamiast niego można też użyć ciągu specjalnego `\n`, co będzie miało takie samo znaczenie.

Operacje na plikach

Żadna poważna aplikacja nie obejdzie się bez operacji na plikach — trzeba przecież w jakiś sposób zapisywać i odczytywać dane. W tej sekcji omówimy więc kilka klas, które pozwalają na wykonywanie tego typu zadań. Wykorzystamy klasy `FileInputStream` i `FileOutputStream` służące do przeprowadzania operacji strumieniowych na plikach. Przydatne będą również klasy: `BufferedReader`, `DataInputStream` i `DataOutputStream`.

ĆWICZENIE

6.15 Zapis danych do pliku

Napisz program zapisujący do pliku kolejne linie tekstu wczytywane z klawiatury. Program powinien zakończyć działanie po wprowadzeniu ciągu znaków "quit".

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        String line = "";
        FileOutputStream fout = null;
        try{
            fout = new FileOutputStream("test.txt");
        }
        catch(FileNotFoundException e){
            System.out.println("Błąd przy otwieraniu pliku.");
            System.exit(-1);
        }
        DataOutputStream out = new DataOutputStream(fout);
        BufferedReader inbr = new BufferedReader(
            new InputStreamReader(System.in)
        );
        try{
            while (true){
                if((line = inbr.readLine()) == null ||
                    line.equals("quit")){
                    break;
                }
                out.writeBytes(line + '\n');
            }
        }
        catch(IOException e){
            System.out.println("Read/Write error.");
        }
    }
}
```

Na początku tworzony jest obiekt typu `FileOutputStream` powiązany z plikiem `test.txt`. Gdyby otwarcie takiego pliku nie było możliwe, zostanie wygenerowany wyjątek `FileNotFoundException` przechwytywany przez blok `try...catch` (mimo nazwy wyjątku plik nie musi istnieć na dysku — zostanie utworzony przez aplikację). Uwaga! Jeśli plik istnieje, zawarte w nim dane zostaną skasowane.

Dane z klawiatury są odczytywane za pomocą obiektu `inbr` klasy `BufferedReader`, podobnie jak miało to miejsce we wcześniejszych przykładach. Do zapisu do pliku jest natomiast używana metoda `writeBytes` z klasy `DataOutputStream`. Aby utworzyć obiekt klasy `DataOutputStream`, w jej konstruktorze został podany obiekt `fout` klasy `FileOutputStream` powiązany z plikiem `test.txt`.

Odczyt i zapis danych odbywa się w pętli `while`. Należy zwrócić uwagę na występującą w niej instrukcję warunkową `if`. Sprawdza ona, czy metoda `readLine` zwróciła wartość `null` (taka sytuacja będzie miała miejsce, gdy strumień wejściowy zostanie przerwany, np. przez wciśnięcie klawiszy `Ctrl+C`) oraz czy wprowadzony tekst to `quit`. Użyto złożonej instrukcji:

```
(line = inbr.readLine()) == null
```

w której wartość uzyskana przez wywołanie metody `readLine` obiektu wskazywanego przez `inbr` jest najpierw przypisywana zmiennej `line`, a następnie wartość tej zmiennej jest porównywana z `null`.

Jeżeli jeden z warunków instrukcji `if` jest prawdziwy, następuje przerwanie pętli za pomocą instrukcji `break`. W przeciwnym razie odczytane dane (zawarte w zmiennej `line`) są zapisywane do pliku. Na końcu każdego zapisywanego wiersza jest dodawany znak końca wiersza symbolizowany przez `\n`.

Ważna jest również kolejność wyrażeń w warunku występującym w `if`. Wykorzystana została cecha skróconego przetwarzania wyrażeń. Wiadomo przecież (m.in. z ćwiczenia 6.5), że sam zapis `line.equals("quit")` mógłby spowodować wystąpienie wyjątku `NullPointerException`. Ponieważ jednak wcześniejsza część wyrażenia (występująca przed operatorem `||`) wyklucza istnienie wartości `null` w zmiennej `line` w drugiej części wyrażenia, zapis formalnie jest prawidłowy. W praktyce lepiej byłoby jednak stosować przedstawiany już zapis `"quit".equals(line)`, który jest bezpieczniejszy i nie spowoduje błędów np. po modyfikacji kodu i usunięciu pierwszego warunku.

ĆWICZENIE

6.16 Odczyt danych z pliku

Napisz program wczytujący dane z pliku tekstowego i wyświetlający je na ekranie.

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        String line = "";
        FileInputStream fin = null;
        try{
            fin = new FileInputStream("test.txt");
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku.");
            System.exit(-1);
        }
        DataInputStream out = new DataInputStream(fin);
        BufferedReader inbr = new BufferedReader(new InputStreamReader(fin));
        try{
            while ((line = inbr.readLine()) != null){
                System.out.println(line);
            }
        }
        catch(IOException e){
            System.out.println("Błąd wejścia-wyjścia.");
        }
    }
}
```

W powyższym ćwiczeniu stosujemy podobne konstrukcje jak w poprzednich przykładach. Ponieważ tym razem dane mają być odczytywane, a nie zapisywane, zamiast klasy `FileOutputStream` stosowana jest `FileInputStream`, a zamiast `DataOutputStream` — `DataInputStream`. Obiekt wskazywany przez zmienną `fin` wiązany jest w wywołaniu konstruktora klasy `FileInputStream` z plikiem *test.txt*, a następnie używany do utworzenia obiektu `inbr` klasy `BufferedReader`. Odczyt danych odbywa się w pętli `while`. Zwróćmy uwagę na postać wyrażenia warunkowego:

```
while ((line = inbr.readLine()) != null)
```

Oznacza ona: pobieraj kolejne linie tekstu, zapisuj je w zmiennej `line` i wykonuj pętlę dopóty, dopóki `line` jest różne od `null`. Ten warunek jest potrzebny do określenia, kiedy zostanie osiągnięty koniec pliku. W takim wypadku metoda `readln` zwraca bowiem właśnie wartość `null`.

Aplikacja z ćwiczenia 6.16 nie jest jednak w pełni funkcjonalna, gdyż potrafi odczytać jedynie zawartość pliku o nazwie *test.txt*. W jaki sposób zmienić ją, tak aby można było odczytać zawartość dowolnego pliku tekstowego? Należy skorzystać z argumentu funkcji `main`:

```
public static void main(String args[])
```

Jest to tablica z referencjami do obiektów typu `String`, która zawiera wszystkie parametry podane w wierszu poleceń podczas uruchamiania programu. Można je zatem bez problemów odczytać.

ĆWICZENIE

6.17 Parametry wywołania aplikacji

Napisz aplikację wyświetlającą jej wszystkie parametry wywołania.

```
public class Main {  
    public static void main(String args[]) {  
        for (int i = 0; i < args.length; i++){  
            System.out.println(args[i]);  
        }  
    }  
}
```

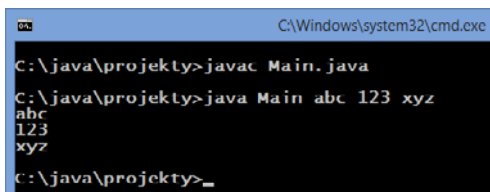
Jeśli teraz przy wywołaniu programu podamy jakieś parametry, np. użyjemy polecenia:

```
java Main abc 123 xyz
```

to wszystkie te parametry zostaną wyświetlone (rysunek 6.5). Warto zauważyć, że — odmiennie niż w C/C++ — pierwszym elementem tablicy nie jest nazwa samego programu.

Rysunek 6.5.

*Parametry
wywołania zostały
wyświetlone
na ekranie*



Skoro wiadomo już, jak odczytać argumenty aplikacji, bez problemów można napisać program wyświetlający zawartość pliku, którego nazwa będzie podawana podczas wywołania:

```
java Main nazwa_pliku
```

ĆWICZENIE

6.18 Wykorzystanie argumentów wywołania aplikacji

Napisz program wyświetlający zawartość pliku tekstowego o nazwie podanej jako argument wywołania.

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        if (args.length < 1){
            System.out.println("Wywołanie programu: Main nazwa_pliku");
            System.exit(0);
        }
        String line = "";
        FileInputStream fin = null;
        try{
            fin = new FileInputStream(args[0]);
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku " + args[0]);
            System.exit(-1);
        }
        BufferedReader inbr = new BufferedReader(
            new InputStreamReader(fin)
        );
        try{
            while ((line = inbr.readLine()) != null){
                System.out.println(line);
            }
        }
        catch(IOException e){
            System.out.println("Błąd wejścia-wyjścia.");
        }
    }
}
```

Program najpierw sprawdza, czy tablica args zawiera przynajmniej jeden element. Jeśli tak, traktuje go jako nazwę pliku, którego zawartość ma być wyświetlona, jeśli nie, wyświetla komunikat informujący o sposobie wywoływania aplikacji. Pozostałe czynności są wykonywane w taki sam sposób jak we wcześniej prezentowanych przykładach, z tym że konstruktor klasy FileInputStream otrzymuje w postaci argumentu wartość pierwszego elementu tablicy args (args[0]).

Skoro wiadomo, jak odczytywać dane z pliku i jak je zapisywać, a także jak odczytać argumenty wywołania programu, to korzystając z przedstawionych technik, można napisać aplikację, która umożliwi kopiowanie plików.

Ć W I C Z E N I E


6.19 Kopiowanie plików

Napisz program kopiujący pliki, których nazwy zostały podane jako parametry wywołania.

```
import java.io.*;

public class Main {
    public static void main(String args[]) {
        if (args.length < 2){
            System.out.println("Wywołanie programu: "+
                "Main nazwa_pliku_źródłowego nazwa_pliku_docelowego");
            System.exit(0);
        }
        FileInputStream fin = null;
        FileOutputStream fout = null;
        try{
            fin = new FileInputStream(args[0]);
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku: " + args[0]);
            System.exit(-1);
        }
        try{
            fout = new FileOutputStream(args[1]);
        }
        catch(FileNotFoundException e){
            System.out.println("Nie można utworzyć pliku: " + args[1]);
            System.exit(-1);
        }
        try{
            int b;
            while ((b = fin.read()) != -1){
                fout.write(b);
            }
            System.out.println("Kopiowanie zakończone!");
        }
        catch(IOException e){
            System.out.println("Błąd wejścia-wyjścia.");
        }
    }
}
```

Obiekty `fin` typu `FileInputStream` (powiązany z plikiem wejściowym) i `fout` typu `FileOutputStream` (powiązany z plikiem wyjściowym) są tworzone w standardowy sposób. Dane są odczytywane z pliku źródłowego za pomocą metody `read` obiektu `fin` (klasy `FileInputStream`; metoda zwraca jeden odczytany ze strumienia bajt w postaci wartości typu `int`), natomiast zapisywane za pomocą metody `write` obiektu `fout` (klasy `FileOutputStream`; metoda zapisuje do strumienia jeden bajt przekazany jej w postaci argumentu typu `int`). Operacje te są wykonywane w pętli `while`, która działa dopóty, dopóki metoda `read` nie zwróci wartości `-1` oznaczającej osiągnięcie końca pliku. Operacje odczytu i zapisu zostały ujęte w blok `try...catch`, pozwalający na obsługę sytuacji, w której pojawiłby się błąd wejścia-wyjścia uniemożliwiający wykonanie operacji kopiowania (zarówno metoda `read`, jak i `write` może wygenerować wyjątek typu `IOException`).



Należy zwrócić uwagę, że zastosowana w ćwiczeniu 6.19 metoda kopiowania bajt po bajcie jest mało efektywna. O wiele lepsze rezultaty dałoby kopiowanie większych bloków danych. To jednak pozostanie zadaniem do samodzielnego wykonania. Zglądając do dokumentacji JDK, można sprawdzić, która metoda pozwala na kopiowanie więcej niż jednego bajtu. Pozwoli to na napisanie dużo szybciej działającego programu.

7

Aplety

Aplikacja a aplet

Programy w Javie mogą być m.in. apletami (ang. *applet*) i aplikacjami (ang. *application*). Różnica jest taka, że aplikacja jest programem samodzielny, natomiast aplet jest kodem interpretowanym przez maszynę wirtualną wbudowaną w przeglądarkę lub inny program. Aplety to zwykle małe programy umieszczane na stronach WWW. Aplet ma również bardzo ograniczony dostęp do systemu, np. nie może nic zapisać na dysku, tak aby złośliwy programista umieszczający niebezpieczny kod na stronie WWW nie był w stanie nam zaszkodzić (chyba że został wyposażony w odpowiedni podpis cyfrowy, a użytkownik zezwolił na wykonywanie takich operacji).

Z podziału na aplety i aplikacje czasem wynikają nieporozumienia. W pierwszych latach po wprowadzeniu Javy na rynek w artykułach często można było przeczytać, że jest ona całkowicie bezpieczna, bo nie ma dostępu do dysku (choć to tylko jeden z mechanizmów bezpieczeństwa), a dwa akapity dalej, że ma być również platformą do budowania dużych, „prawdziwych” aplikacji. Potem pojawiały się pytania czytelników, jak napisać np. edytor tekstu bez możliwości

zapisu na dysku. Otóż oczywiście w języku programowania Java istnieje możliwość zarówno zapisu danych na dysku, jak i tworzenia wielu innych konstrukcji programistycznych. Pozwalają na to bezpośrednio programy pisane jako aplikacje, a po spełnieniu dodatkowych warunków — również aplety.

Obecnie aplety straciły na znaczeniu i nie są już tak często używane, wciąż są jednak powszechne np. na stronach prezentujących notowania giełdowe, oferujących narzędzia do analizy technicznej związanej z grą na giełdach papierów wartościowych, a także w aplikacjach typu czat czy też niektórych grach działających w przeglądarkach WWW.

Pierwszy aplet

W rozdziale 1. książki zaczynaliśmy od programu wypisującego napis na ekranie. Tym razem zrobimy tak samo. Stworzymy aplet wyświetlający na ekranie napis *Pierwszy aplet w Javie*. Aplet może być tworzony na bazie klasy `Applet` z pakietu `java.applet` lub też nowocześniejszej klasy pochodnej `JApplet` zawartej w pakiecie `javax.swing`. Będziemy korzystać z tej drugiej, choć w przykładach z tego rozdziału różnice między tymi klasami nie mają praktycznego znaczenia.

ĆWICZENIE

7.1 Aplet wyświetlający zdefiniowany tekst

Napisz aplet wyświetlający na ekranie napis *Pierwszy aplet w Javie*.

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class AppletHello extends JApplet {
    public void paint (Graphics gDC) {
        gDC.drawString ("Pierwszy aplet w Javie", 100, 100);
    }
}
```

Co należy z tym programem zrobić? Przede wszystkim umieścić w pliku *AppletHello.java* i skompilować. Uzyskamy wtedy plik z kodem pośrednim *AppletHello.class*, który może być zagnieżdżony w stronie WWW.

Aby aplet mógł być wyświetlony, trzeba go umieścić w kodzie strony WWW¹. Dawniej służył do tego znacznik HTML `<applet>`, który w uproszczeniu można przedstawić tak:

```
<applet
  code = "nazwa pliku z kodem apletu"
  width = "szerokość apletu"
  height = "wysokość apletu">
</applet>
```

Obecnie należy używać znacznika `<object>`:

```
<object
  type="application/x-java-applet"
  code = "nazwa pliku z kodem apletu"
  width = "szerokość apletu"
  height = "wysokość apletu">
</object>
```

Kod strony najlepiej zapisać w pliku o rozszerzeniu *html*, np. *index.html*.



Formalnie w HTML5 w znaczniku `<object>` zamiast atrybutu `code` powinien się znajdować atrybut `data`. Nie był on jednak rozpoznawany przez żadną z wersji aplikacji `appletviewer` z pakietu JDK dostępnych w trakcie przygotowywania materiałów do książki. Dlatego też na listingach użyty został atrybut `code` (niepoprawny, ale rozpoznawany przez wszystkie popularne przeglądarki).

Ć W I C Z E N I E

7.2 Umieszczanie apletu na stronie WWW

Napisz kod HTML pozwalający na osadzenie klasy apletu z ćwiczenia 7.1 na stronie WWW.

```
<!DOCTYPE html>
<html lang="pl">
  <head>
    <meta charset="utf-8">
    <title>Moja strona WWW</title>
  </head>
  <body>
    <div>
      <object
```

¹ Osobom nieznającym języka opisu strony (HTML) i technik tworzenia WWW można polecić książkę *Tworzenie stron WWW. Praktyczny kurs* (<http://helion.pl/ksiazki/twspk2.htm>).

```
type="application/x-java-applet"  
code="AppletHello.class"  
width="320"  
height="200">  
</object>  
</div>  
</body>  
</html>
```

Ć W I C Z E N I E

7.3 Uruchomienie apletu

Uruchom aplet przygotowany w ćwiczeniu 7.1.

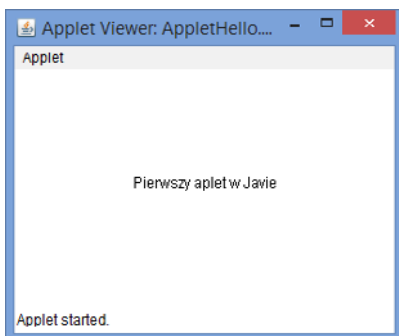
Aby uruchomić aplet, należy umieścić pliki *AppletHello.class* (skompilowany kod z ćwiczenia 7.1) i *index.html* (kod z ćwiczenia 7.2) w jednym katalogu, a następnie otworzyć plik *index.html* w dowolnej przeglądarce WWW obsługującej Javę (z reguły poprzez mechanizm wtyczek). Druga możliwość to użycie narzędzia o nazwie *appletviewer* z pakietu JDK. Wystarczy w wierszu poleceń wydać komendę:

```
appletviewer index.html
```

Na ekranie pojawi się wtedy widok zaprezentowany na rysunku 7.1.

Rysunek 7.1.

*Aplet wyświetlający
napis na ekranie*



Jak to działa?

Aplet z ćwiczenia 7.1 dziedziczy po predefiniowanej klasie `JApplet`, która opisuje zachowania apletów. Jest ona zawarta w pakiecie *javax.swing*, dlatego też na początku kodu znajduje się instrukcja:

```
import javax.swing.JApplet;
```

Jest w nim zadeklarowana tylko jedna metoda, mianowicie `paint`, która jest zawsze wywoływana przez system, kiedy zachodzi potrzeba odrysowania (odświeżenia) okna apletu, np. gdy zostanie ono schowane za innym oknem, a następnie ponownie odsłonięte. Jedynym parametrem tej metody jest referencja do obiektu typu `Graphics` (definicja tej klasy znajduje się w pakiecie *java.awt*, stąd druga instrukcja `import` na początku kodu) i jest to tzw. **graficzny kontekst urządzenia** (ang. *Graphics Device Context*), przez niektórych zwany również **wykreślaczem**². Pozostańmy jednak przy nazwie kontekst urządzenia. Co to jest? Otóż jest to obiekt, dzięki któremu możemy rysować na ekranie. Wydając mu odpowiednie komendy, czyli wywołując zdefiniowane w nim metody, powodujemy wyświetlenie na ekranie różnych obiektów. W przypadku apletu z ćwiczenia 7.1 jest to napis podany jako pierwszy parametr metody `drawString`. Dwa kolejne parametry to współrzędne, od których zacznie się rysowanie tego napisu.

Sam skompilowany kod apletu nie wystarczył. Trzeba go było jeszcze umieścić na stronie WWW, tak aby przeglądarka wiedziała, gdzie go szukać i jak interpretować. Dlatego też w kodzie HTML z ćwiczenia 7.2 pojawił się znacznik `<object>` z parametrem `code` wskazującym nazwę pliku ze skompilowanym kodem apletu (*AppletHello.class*) oraz parametrem `type` wskazującym, że kod ma być traktowany jako aplet (`application/x-java-applet`).

Apletowi można przekazać dodatkowe parametry, np. dane sterujące jego pracą. Należy wtedy użyć znaczników `<param>` wewnątrz znacznika `<object>`. Schematycznie wygląda to następująco:

```
<object
  type="application/x-java-applet"
  code = "nazwa klasy"
  width = "szerokość apletu"
  height = "wysokość apletu">
```

² Stosowany jest też termin *kontekst rysowniczy*.

```
<param name="nazwa1" value="wartość1" />  
<param name="nazwa2" value="wartość2" />  
<!-- dalsze znaczniki param -->  
</object>
```

Po takiej definicji aplet będzie miał dostęp do parametru nazwa1 o wartości wartość1, parametru nazwa2 o wartości wartość2 itd.

Cykl życia apletu

Gdy przeglądarka obsługująca Javę odnajdzie w kodzie HTML znacznik `<object>` z atrybutem `type` o wartości `application/x-java-applet`, wykonuje określoną sekwencję czynności. Zaczyna od sprawdzenia, czy w podanej lokalizacji znajduje się plik zawierający kod klasy wymienionej jako wartość argumentu `code` (lub `data`) znacznika. Jeśli plik znajduje się we wskazanej lokalizacji, zawarty w nim kod ładowany jest do pamięci i tworzona jest instancja (czyli obiekt) znalezionej klasy. Tym samym wywoływany jest również konstruktor. Po tych czynnościach wykonywana jest metoda `init` utworzonego obiektu, informująca go o tym, że został załadowany do systemu. W metodzie `init` można zatem umieścić (o ile zachodzi taka potrzeba) procedury inicjacyjne.

Gdy aplet jest gotowy do uruchomienia, przeglądarka wywołuje jego metodę `start`, informując, że powinien rozpocząć swoje działanie. Przy pierwszym ładowaniu apletu metoda `start` wykonywana jest zawsze po metodzie `init`. W momencie gdy aplet powinien zakończyć swoje działanie, zostaje wywołana z kolei jego metoda `stop`.

Kroje pisma (fonty)

Napis, który wyświetliliśmy w pierwszym aplecie, wyglądał nieco mizernie. Warto by przynajmniej odrobinę go powiększyć i pogrubić. W tym celu musimy zmienić font. Aby tego dokonać, trzeba stworzyć nowy obiekt typu `Font`, a następnie użyć metody `setFont`.

ĆWICZENIE

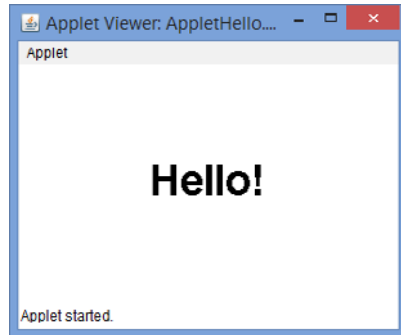
7.4 Zastosowanie wybranego kroju pisma

Napisz aplet wyświetlający na ekranie napis pogrubionym krojem *SansSerif* (bezseryfowym) o wielkości 36 punktów (rysunek 7.2).

```
import javax.swing.JApplet;  
import java.awt.*;  
  
public class AppletHello extends JApplet {  
    public void paint (Graphics gDC) {  
        Font font = new Font ("SansSerif", Font.BOLD, 36);  
        gDC.setFont (font);  
        gDC.drawString ("Hello!", 110, 110);  
    }  
}
```

Rysunek 7.2.

*Aplet wyświetlający
napis wybraną
czcionką*



W konstruktorze obiektu typu `Font` podajemy nazwę fontu, a następnie jej typ oraz wielkość. Do dyspozycji mamy następujące typy:

1. `Font.PLAIN` — krój zwykły,
2. `Font.BOLD` — krój pogrubiony,
3. `Font.ITALIC` — krój pochylony.

W środowisku Java 2 i wyższych dostępnych jest pięć fontów logicznych:

1. *Serif.*
2. *SansSerif.*
3. *Monospaced.*
4. *Dialog.*
5. *DialogInput.*

Nastąpiła tu spora zmiana w stosunku do pierwotnych wersji Javy (JDK 1.1), w których dostępnych było sześć rodzajów fontów (wspólne są jedynie *Dialog* i *DialogInput*). Domyślnie (jeżeli nie ustawimy własnego kroju) jest używany krój *Dialog*. Wymienione fonty są dostępne w każdej wersji JRE i JDK niezależnie od użytego środowiska uruchomieniowego.

Nie oznacza to jednak, że nie można stosować innych czcionek. Można użyć dowolnej czcionki znajdującej się w systemie, z tym że nie mamy wtedy pewności, że ta czcionka będzie dostępna w systemie użytkownika apletu. Jeśli nie będzie dostępna, środowisko uruchomieniowe (JRE) postara się dopasować najbardziej zbliżoną czcionkę.

Własności poszczególnych czcionek można poznać dzięki obiektowi `FontMetrics` oraz metodzie `getFontMetrics`. Obiekt ten ma kilkanaście metod, spośród których bardziej interesujące to:

- ❑ `charWidth` — zwraca szerokość litery podanej jako parametr,
- ❑ `getHeight` — zwraca typową wysokość wiersza tekstu zapisanego za pomocą czcionki,
- ❑ `getLeading` — zwraca odległość pomiędzy wierszami tekstu zapisanego za pomocą czcionki,
- ❑ `stringWidth` — zwraca długość fragmentu tekstu zapisanego za pomocą czcionki.

Ć W I C Z E N I E

7.5 Dostępne kroje pisma

Napisz aplet prezentujący na ekranie dostępne standardowo kroje (rysunek 7.3).

```
import javax.swing.JApplet;
import java.awt.*;

public class AppletFonts extends JApplet {
    Font fontSerif, fontSansSerif, fontMonospaced,
        fontDialog, fontDialogInput;

    public void init() {
        fontSerif = new Font("Serif", Font.BOLD, 36);
        fontSansSerif = new Font("SansSerif", Font.BOLD, 36);
        fontMonospaced = new Font("Monospaced", Font.BOLD, 36);
        fontDialog = new Font("Dialog", Font.BOLD, 36);
        fontDialogInput = new Font("DialogInput", Font.BOLD, 36);
    }
}
```

```
public void paint (Graphics gDC) {  
    gDC.setFont(fontSerif);  
    gDC.drawString("Serif", 110, 30);  
    gDC.setFont(fontSansSerif);  
    gDC.drawString("SansSerif", 110, 70);  
    gDC.setFont(fontMonospaced);  
    gDC.drawString("Monospaced", 110, 110);  
    gDC.setFont(fontDialog);  
    gDC.drawString("Dialog", 110, 150);  
    gDC.setFont(fontDialogInput);  
    gDC.drawString("DialogInput", 110, 190);  
}  
}
```

Rysunek 7.3.

*Aplet prezentujący
dostępne kroje*



W przeciwieństwie do poprzedniego ćwiczenia została tu wykorzystana metoda `init`, od której rozpoczyna się wykonywanie apletu. To w niej tworzone są niezbędne obiekty typu `Font`. Wprawdzie te czynności mogłyby być również wykonywane w metodzie `paint`, ale w tym przypadku byłoby to gorsze rozwiązanie. Nie ma bowiem potrzeby tworzenia nowych obiektów typu `Font` za każdym razem, gdy konieczne jest odświeżenie obszaru apletu, czyli przy każdym wywołaniu metody `paint`. Niepotrzebnie marnowałoby to tylko zasoby systemowe. Dlatego też obiekty tworzymy tylko raz, a w metodzie `paint` jedynie się do nich odwołujemy.

Rysowanie grafiki

Klasa `Graphics` umożliwia nam rysowanie prostych figur geometrycznych, można przy tym rysować zarówno same obrzeża figur, jak i figury wypełnione kolorem. Dostępne metody pozwalają na narysowanie:

- ☐ linii,
- ☐ łuków,
- ☐ kół i elips,
- ☐ prostokątów,
- ☐ wielokątów.

Nie ma natomiast dedykowanej metody, która pozwala na rysowanie pojedynczych punktów. Można zamiast tego rysować linie o długości jednego piksela.

Do rysowania linii służy metoda:

```
drawLine(int x1, int y1, int x2, int y2);
```

Rysuje ona odcinek zaczynający się w punkcie o współrzędnych $x1, y1$, a kończący się w punkcie o współrzędnych $x2, y2$.

Ć W I C Z E N I E

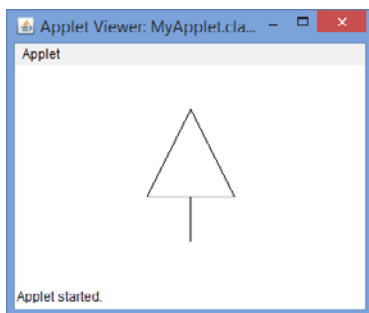
7.6 Rysowanie w obszarze apletu

Napisz aplet wyświetlający na ekranie strzałkę (rysunek 7.4).

```
import javax.swing.JApplet;  
import java.awt.Graphics;  
  
public class MyApplet extends JApplet {  
    public void paint (Graphics gDC) {  
        gDC.drawLine (120, 120, 160, 40);  
        gDC.drawLine (200, 120, 160, 40);  
        gDC.drawLine (120, 120, 200, 120);  
        gDC.drawLine (160, 160, 160, 120);  
    }  
}
```

Rysunek 7.4.

*Efekt działania apletu
z ćwiczenia 7.6*



Ć W I C Z E N I E

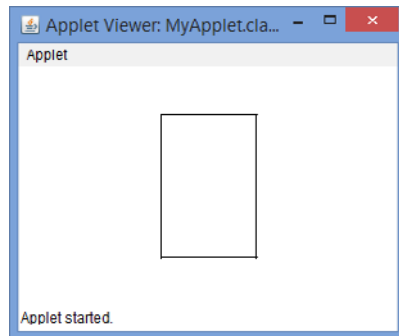
7.7 Rysowanie linii

Korzystając z metody `drawLine`, napisz aplet wyświetlający na ekranie prostokąt (rysunek 7.5).

```
import javax.swing.JApplet;  
import java.awt.Graphics;  
  
public class MyApplet extends JApplet {  
    public void paint (Graphics gDC) {  
        gDC.drawLine (120, 40, 200, 40);  
        gDC.drawLine (200, 40, 200, 160);  
        gDC.drawLine (200, 160, 120, 160);  
        gDC.drawLine (120, 160, 120, 40);  
    }  
}
```

Rysunek 7.5.

*Wynik działania
apletu rysującego
na ekranie
prostokąt*



Koła i elipsy można rysować za pomocą metod:

```
drawOval (int x, int y, int szerokość, int wysokość)  
fillOval (int x, int y, int szerokość, int wysokość)
```

Pierwsza rysuje figury bez wypełnienia, druga — z wypełnieniem. Podane parametry określają prostokąt okalający dane koło (okrąg) lub elipsę, x i y to współrzędne lewego górnego rogu, a *szerokość* i *wysokość* to odpowiednio szerokość i wysokość (podane w pikselach).

ĆWICZENIE

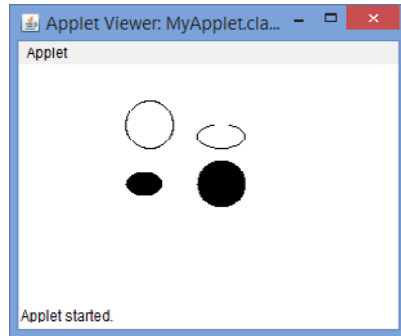
7.8 Rysowanie kół i elips

Napisz aplet wyświetlający na ekranie koła i elipsy (rysunek 7.6).

```
import javax.swing.JApplet;  
import java.awt.Graphics;  
  
public class MyApplet extends JApplet {  
    public void paint (Graphics gDC) {  
        gDC.drawOval(90, 30, 40, 40);  
        gDC.drawOval(150, 50, 40, 20);  
        gDC.fillOval(90, 90, 30, 20);  
        gDC.fillOval(150, 80, 40, 40);  
    }  
}
```

Rysunek 7.6.

*Aplet wyświetlający
na ekranie koła
i elipsy*



Prostokąty można wyświetlać za pomocą metod:

```
drawRect(int x, int y, int szerokość, int wysokość)  
fillRect(int x, int y, int szerokość, int wysokość)
```

Pierwsza rysuje figury bez wypełnienia, druga — z wypełnieniem (podobnie jak miało to miejsce w przypadku kół i elips). Podane parametry określają prostokąt, *x* i *y* to współrzędne lewego górnego rogu, a *szerokość* i *wysokość* to odpowiednio szerokość i wysokość (podane w pikselach).

Ć W I C Z E N I E

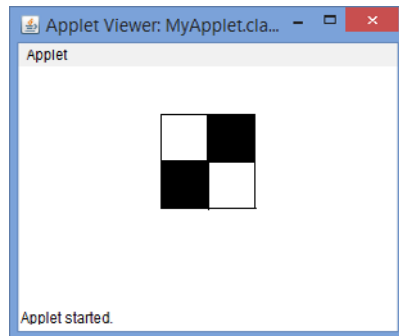
7.9 Rysowanie prostokątów

Napisz aplet rysujący na ekranie kilka prostokątów, np. ustawionych tak jak na rysunku 7.7.

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class MyApplet extends JApplet {
    public void paint (Graphics gDC) {
        gDC.drawRect (120, 40, 39, 39);
        gDC.fillRect (160, 40, 40, 40);
        gDC.fillRect (120, 80, 40, 40);
        gDC.drawRect (160, 80, 39, 39);
    }
}
```

Rysunek 7.7.
*Aplet rysujący
prostokąty*



Do rysowania wielokątów służą metody:

```
drawPolygon(Polygon p)
fillPolygon(Polygon p)
drawPolygon(int tabX[], int tabY[], int nPoints)
fillPolygon(int tabX[], int tabY[], int nPoints)
```

Parametr *p* jest typu *Polygon*, co oznacza, że należy utworzyć obiekt takiego typu. Można tego dokonać za pomocą następującego konstruktora:

```
Polygon (int xPoints[], int yPoints[], int nPoints)
```

Tablice *xPoints* i *yPoints* muszą zawierać współrzędne kolejnych punktów wielokąta, natomiast parametr *nPoints* oznacza liczbę tych punktów.

W kolejnych dwóch ćwiczeniach użyto zarówno wersji z argumentem typu Polygon, jak i wersji korzystających z tablic.

Ć W I C Z E N I E

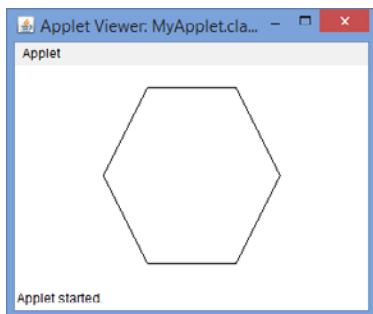
7.10 Rysowanie wielokątów

Napisz aplet wyświetlający na ekranie sześciokąt (rysunek 7.8).

```
import javax.swing.JApplet;  
import java.awt.Graphics;  
  
public class MyApplet extends JApplet {  
    int tabX[] = {80, 120, 200, 240, 200, 120};  
    int tabY[] = {100, 20, 20, 100, 180, 180};  
    public void paint (Graphics gDC){  
        gDC.drawPolygon (tabX, tabY, 6);  
    }  
}
```

Rysunek 7.8.

*Aplet rysujący
sześciokąt*



Ć W I C Z E N I E

7.11 Rysowanie powtarzających się figur

Napisz aplet generujący na ekranie wzory widoczne na rysunku 7.9.

```
import javax.swing.JApplet;  
import java.awt.*;  
  
public class MyApplet extends JApplet {  
    int tabX[] = {30, 40, 60, 40, 30, 20, 0, 20, 30};  
    int tabY[] = {220, 240, 250, 260, 280, 260, 250, 240, 220};  
    public void paint (Graphics gDC) {  
        for(int i = 0; i < 30; i++){  
            gDC.drawLine (235 + i * 3, 20, 325 - i * 3, 200);  
            gDC.drawOval (60 + i * 2, 100 - i * 2, 150 - i * 4, 50 - i * 4);  
            gDC.drawOval (360 + i * 2, 100 - i * 2, 150 - i * 4, 50 - i * 4);  
        }  
    }  
}
```

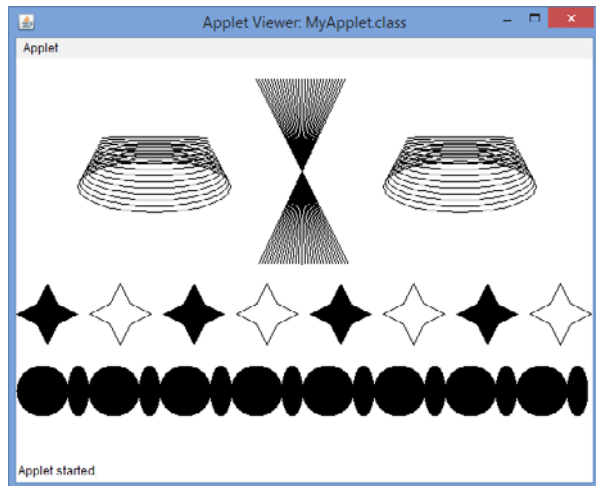
```

    }
    boolean flag = false;
    for(int i = 0; i < 8; i++){
        gDC.fillOval (i * 70, 300, 50, 50);
        gDC.fillOval (50 + i * 70, 300, 20, 50);
        int tempTabX[] = new int[9];
        for (int j = 0; j < 9; j++){
            tempTabX[j] = tabX[j] + i * 72;
        }
        Polygon p = new Polygon(tempTabX, tabY, 9);
        if (flag){
            gDC.drawPolygon(p);
        }
        else{
            gDC.fillPolygon(p);
        }
        flag = !flag;
    }
}
}

```

Rysunek 7.9.

Wzory
generowane
przez aplet
z ćwiczenia 7.11



W pierwszej pętli for rysowane są nakładające się na siebie elipsy i przecinające się odcinki. Użyto w tym celu standardowych metod `drawOval` i `drawLine`, z tym że ich parametry są wyliczane na podstawie wartości zmiennej iteracyjnej `i`. Dzięki temu otrzymujemy figury o nieco zmienionym kształcie i (lub) położeniu.

Druga pętla odpowiada za rysowanie dolnego wiersza kół i elips oraz wielokątów. Koła i elipsy są rysowane za pomocą metody `fillOval`, a przesunięcie figur jest uzyskiwane również dzięki uwzględnieniu wartości zmiennej `i` w wartościach parametrów metody. Nieco inaczej jest w przypadku wielokątów. Współrzędne figury bazowej zostały opisane w tablicach `tabX` i `tabY`. Współrzędne `y` pozostają takie same dla wszystkich kopii, natomiast współrzędne `x` muszą być przesuwane w prawo, czyli zwiększane. Każdy wielokąt jest odsuwany o 72 (`i * 72`) piksele od poprzednika, a jego współrzędne `x` są zapisywane w tablicy pomocniczej `tempTabX`. Tablic `tempTabX` i `tabY` użyto do skonstruowania obiektu typu `Polygon`, który jest następnie stosowany jako argument metody `drawPolygon` lub `fillPolygon`.

O tym, która z metod zostanie użyta, decyduje stan zmiennej `flag`. Gdy jest równa `true`, rysowany jest tylko obrys wielokąta, a gdy jest równa `false`, rysowany jest wielokąt z wypełnionym tłem. Stan zmiennej jest zmieniany na przeciwny w każdym przebiegu pętli — odpowiada za to instrukcja `flag = !flag`. Dzięki temu uzyskujemy naprzemiennie wypełnienie i obrys.

Kolory

Do ustalania koloru używamy metody `setColor` z klasy `Graphics`, która przyjmuje jako argument obiekt typu `Color`. Obiekty `Color` służą do reprezentacji kolorów w formacie RGB. Format ten zakłada, że każdy kolor jest opisany przez trzy składowe: czerwoną (ang. *Red*), zieloną (ang. *Green*) i niebieską (ang. *Blue*). Każda ze składowych może przyjmować wartości od 0 do 255. Wynika z tego, że pojedynczy kolor jest opisywany przez 24 bity, a maksymalna ich liczba wynosi 2^{24} , czyli 16 777 216. Kolor czarny to składowe $R = 0$, $G = 0$, $B = 0$, natomiast biały to $R = 255$, $G = 255$, $B = 255$. Składowe RGB dla niektórych wybranych kolorów przedstawiono w tabeli 7.1.

Obiekt typu `Color` możemy utworzyć, podając w konstruktorze liczbę całkowitą typu `int`. Składową R specyfikują bity 16 – 23 tej liczby, składową G — bity 8 – 15, składową B — bity 0 – 7. Można również użyć konstruktora przyjmującego jako parametry trzy liczby całkowite odpowiadające poszczególnym składowym.

Tabela 7.1. Składowe RGB dla wybranych kolorów

Kolor	Składowa R	Składowa G	Składowa B
beżowy	245	245	220
biały	255	255	255
błękitny	0	191	255
brązowy	139	69	19
czarny	0	0	0
czerwony	255	0	0
ciemnoczerwony	139	0	0
ciemnoniebieski	0	0	139
ciemnoszary	169	169	169
ciemnozielony	0	100	0
fioletowy	238	130	238
koralowy	255	127	80
niebieski	0	0	255
oliwkowy	107	142	35
purpurowy	160	32	240
srebrny	192	192	192
stalowoniebieski	70	130	180
szary	128	128	128
zielony	0	255	0
żółtozielony	195	205	50
żółty	255	255	0

Metody `getRed`, `getGreen`, `getBlue` pozwalają na pobranie oddzielnie każdej składowej, natomiast metoda `getRGB` na pobranie liczby typu `int` reprezentującej kolor. Znaczenie poszczególnych bitów jest takie samo, jak opisane wyżej. Klasa `Color` posiada również predefiniowane pola statyczne odzwierciedlające niektóre kolory. Są to `black`

(czarny), blue (niebieski), cyan (cyjan), darkGray (ciemnoszary), grey (szary), green (zielony), lightGray (jasnoszary), magenta (magenta), orange (pomarańczowy), pink (różowy), red (czerwony), white (biały) i yellow (żółty). Te wiadomości wystarczą nam już do urozmaicenia grafiki z poprzednich ćwiczeń.

Ć W I C Z E N I E

7.12 Kolorowy wielokąt

Napisz aplet rysujący na ekranie sześciokąt wypełniony kolorem czerwonym.

```
import javax.swing.JApplet;
import java.awt.*;

public class MyApplet extends JApplet {
    int tabX[] = {80, 120, 200, 240, 200, 120};
    int tabY[] = {100, 20, 20, 100, 180, 180};
    public void paint (Graphics gDC) {
        gDC.setColor(Color.red);
        gDC.fillPolygon (tabX, tabY, 6);
    }
}
```

Ć W I C Z E N I E

7.13 Kolorowanie figur

„Pokoloruj” wzory generowane przez aplet z ćwiczenia 7.11.

```
import javax.swing.JApplet;
import java.awt.*;

public class MyApplet extends JApplet {
    int tabX[] = {30, 40, 60, 40, 30, 20, 0, 20, 30};
    int tabY[] = {220, 240, 250, 260, 280, 260, 250, 240, 220};
    public void paint (Graphics gDC) {
        Color color1 = Color.red;
        for(int i = 0; i < 30; i++){
            gDC.setColor (color1);
            color1 = new Color (color1.getRed() - 8, color1.getGreen(),
                                color1.getBlue());
            gDC.drawLine (235 + i * 3, 20, 325 - i * 3, 200);
            gDC.setColor(Color.green);
            gDC.drawOval (60 + i * 2, 100 - i * 2, 150 - i * 4, 50 - i * 4);
            gDC.setColor(Color.blue);
            gDC.drawOval (360 + i * 2, 100 - i * 2, 150 - i * 4, 50 - i * 4);
        }
    }
}
```



```
boolean flag = false;
Color color2 = Color.green;
Color color3 = Color.blue;
for(int i = 0; i < 8; i++){
    gDC.setColor(color2);
    color2 = new Color (color2.getRed(), color2.getGreen() - 20,
                        color2.getBlue() + 20);

    gDC.fillOval (i * 70, 300, 50, 50);
    gDC.fillOval (50 + i * 70, 300, 20, 50);

    int tempTabX[] = new int[9];
    for (int j = 0; j < 9; j++){
        tempTabX[j] = tabX[j] + i * 72;
    }

    Polygon p = new Polygon(tempTabX, tabY, 9);
    gDC.setColor (color3);
    color3 = new Color (color3.getRed() + 25, color3.getGreen(),
                        color3.getBlue() - 25);

    if (flag){
        gDC.drawPolygon(p);
    }
    else{
        gDC.fillPolygon(p);
    }
    flag = !flag;
}
}
```

Sposób generowania figur jest taki sam jak w ćwiczeniu 7.11, zostały jednak dodane instrukcje zmieniające kolory. Wszystkie odcienie są tworzone na bazie czerwonego (Color.red), niebieskiego (Color.blue) i zielonego (Color.green). Górne elipsy są wypełniane jednolitymi odcieniami: zielonym i niebieskim. Kolory odcinków są już modyfikowane. Składowa *R* (czerwona) każdej kolejnej linii jest zmniejszana o 8, przy niezmiennych składowych *G* (zielona) i *B* (niebieska). Ponieważ początkowym kolorem był czerwony, w rezultacie otrzymujemy odcienie od czerwonego do czarnego.

Na takiej samej zasadzie tworzone są różne odcienie dla dolnego rzędu kół i elips oraz dla wielokątów. Modyfikowane są tylko inne składowe. Dla kół są to składowe *G* (zwiększana o 20) i *B* (zmniejszana o 20), a dla wielokątów — składowe *R* (zwiększana o 25) i *B* (zmniejszana o 25). Można samodzielnie poeksperymentować z innymi

modyfikacjami składowych czy też z ustawieniami kolorów początkowych, pamiętając jednak, że każda ze składowych może się zmieniać wyłącznie w przedziale od 0 do 255.

Wyświetlanie obrazów

Umiemy już rysować proste figury geometryczne, nadszedł więc czas, aby wyświetlić na ekranie grafikę zawartą w pliku. Klasa `JApplet` na szczęście zawiera odpowiednie metody, nie jest to więc bardzo skomplikowane zadanie. Do wczytywania obrazów służy metoda `getImage`, której — jako parametr — podajemy lokalizację danego pliku graficznego. Standardowo obsługiwane są tylko trzy formaty plików graficznych: GIF, JPG i PNG. Zakładając, że w katalogu, w którym jest umieszczony dokument HTML zawierający aplet, znajduje się np. plik *obrazek.jpg*, możemy wczytać go tak:

```
Image image = getImage (getDocumentBase(), "obrazek.jpg")
```

Jeśli chcemy samodzielnie podać pełną ścieżkę dostępu do pliku, możemy też użyć konstrukcji:

```
Image image = getImage ("http://adres.domeny/katalog/obrazek.jpg")
```

W pierwszym przypadku do uzyskania ścieżki dostępu (w postaci adresu URL), w której znajduje się dokument z apletem, została użyta metoda `getDocumentBase()` z klasy `JApplet`.

W obu przypadkach jako rezultat otrzymujemy referencję do nowego obiektu klasy `Image`. Reprezentuje on dane graficzne, o ile tylko we wskazanej lokalizacji był dostępny prawidłowy plik graficzny. Gdy mamy odwołanie do obiektu obrazu, można go wyświetlić na ekranie. Wystarczy posłużyć się metodą `drawImage` z klasy `Graphics`. Należy pamiętać, że obraz nie jest gotowy do wyświetlenia od razu po wywołaniu metody `getImage` — dane zostaną pobrane dopiero przy próbie wyświetlenia.

Ć W I C Z E N I E

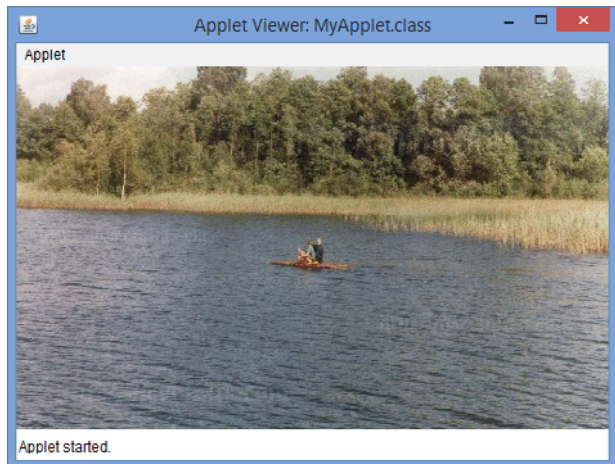
7.14 Wyświetlanie zawartości pliku graficznego

Napisz aplet wyświetlający na ekranie plik graficzny, np. obraz typu *jpg* (rysunek 7.10).

```
import javax.swing.JApplet;  
import java.awt.*;  
  
public class MyApplet extends JApplet {  
    Image img;  
    public void init() {  
        img = getImage(getDocumentBase(), "obrazek.jpg");  
    }  
    public void paint (Graphics gDC) {  
        gDC.drawImage (img, 0, 0, this);  
    }  
}
```

Rysunek 7.10.

*Aplet
wyświetlający
obraz pobrany
z pliku
graficznego*



Obraz jest reprezentowany przez pole *img* typu *Image*. Obiekt *img* tworzony jest z kolei w metodzie *init*, czyli w trakcie inicjacji apletu. Użyto dwuargumentowej metody *getImage*. Zgodnie z wcześniejszym opisem pierwszy argument wskazuje lokalizację, w której znajdują się pliki z kodem HTML zawierającym aplet (rezultat wywołania metody *getDocumentBase*)³, a drugi — nazwę pliku z grafiką.

³ Alternatywnie można użyć metody *getCodeBase* — zwraca ona obiekt URL zawierający adres, pod którym znajduje się kod klasy apletu.

Metoda `drawImage` przyjmuje cztery argumenty. Pierwszy to referencja do obiektu typu `Image`. Dwa następne to współrzędne, od których zacznie się wyświetlanie obrazka. Jako ostatni parametr występuje słowo `this`, czyli wskazanie na obiekt bieżący (w tym przypadku obiekt klasy `MyApplet`, która dziedziczy po `JApplet`).

Czwarty argument metody `drawImage` nie musi być obiektem apletu. W rzeczywistości powinna to być referencja do obiektu klasy implementującej interfejs `ImageObserver`. Niestety, w książce nie ma miejsca na omówienie interfejsów i klas interfejsowych. Ważne jest jednak, że jeżeli obiekt implementuje interfejs `ImageObserver` (a tak jest w przypadku apletów), to w trakcie napływu danych z obrazem wywoływana jest metoda `imageUpdate` tego obiektu. To pozwala śledzić postępy ładowania grafiki. Ta właściwość zostanie wykorzystana w kolejnym ćwiczeniu.

Ć W I C Z E N I E

7.15 Śledzenie postępów ładowania obrazu

Napisz aplet wyświetlający na ekranie plik graficzny. Podczas wczytywania obrazka na pasku stanu przeglądarki powinien być wyświetlany napis `Trwa ładowanie obrazu...`, a po zakończeniu tego procesu — napis `Obraz załadowany!`.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.image.*;

public class MyApplet extends JApplet {
    Image img;
    public void init(){
        img = getImage(getDocumentBase(), "obrazek.jpg");
    }
    public void paint (Graphics gDC){
        gDC.drawImage (img, 0, 0, this);
    }
    public boolean imageUpdate(Image img, int flags, int x,
        int y, int width, int height) {
        if ((flags & ImageObserver.ALLBITS) == 0){
            showStatus ("Trwa ładowanie obrazu...");
            return true;
        }
    }
}
```

```
    }  
    else{  
        showStatus ("Obraz załadowany!");  
        repaint();  
        return false;  
    }  
}  
}
```

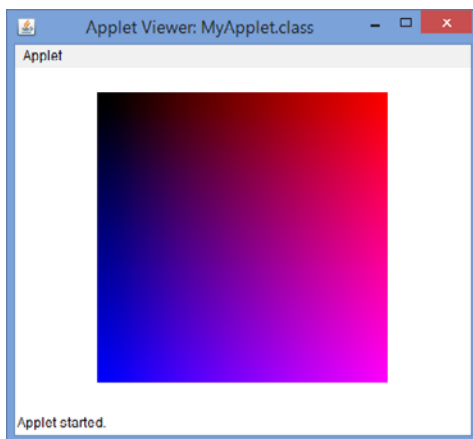
Metody `init` i `paint` pozostały tu w formie znanej z ćwiczenia 7.14. Za aktualizację statusu odpowiada z kolei metoda `imageUpdate`. Zgodnie z podanym wcześniej opisem będzie ona wywoływana w trakcie ładowania obrazu (w trakcie napływania danych obrazu z sieci). Przyjmuje ona sześć argumentów: `img` (referencja do rysowanego obrazu), `x` i `y` (współrzędne powierzchni apletu, w których rozpoczyna się obraz), `width` i `height` (wysokość i szerokość obrazu) oraz `flags` (wartość całkowita informująca o aktualnym stanie obrazu).

Obraz jest całkowicie załadowany, gdy wynik operacji bitowej `flags & ImageObserver.ALLBITS` jest różny od 0. Jest to badane w instrukcji warunkowej `if`. W zależności od wyniku tej operacji zmieniany jest napis na pasku stanu. W tym celu używa się metody `setStatus` pochodzącej z klasy `JApplet`. Po załadowaniu obrazu wywoływana jest metoda `repaint()`, dzięki której obszar apletu zostanie odświeżony. Należy zwrócić uwagę, że gdy obraz nie jest jeszcze gotowy (czyli konieczne są doładowania danych), metoda `imageUpdate` ma zwrócić wartość `true`, a jeśli jest gotowy — wartość `false`.

Dzięki tak działającemu apletowi w przypadku ładowania dużej grafiki (co może zająć sporo czasu) użytkownik będzie informowany o tym, że ten proces trwa i należy poczekać na jego zakończenie.

Na koniec spójrzmy na rysunek 7.11. Zawiera on prostokąt z płynnie zmieniającymi się kolorami (z oczywistych względów w książce widoczne są tylko odcienie szarości). Jest to również obraz rysowany za pomocą metody `drawImage`. Nie jest jednak stworzony w programie graficznym, ale jest generowany w pamięci komputera. Sposób jego wykonania został podany w treści ćwiczenia 7.16.

Rysunek 7.11.
*Programowo
wygenerowana
paleta kolorów*



Ć W I C Z E N I E


7.16 Programowe generowanie obrazów

Wygeneruj programowo obraz widoczny na rysunku 7.11.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.image.*;

public class MyApplet extends JApplet {
    Image img;
    public void init() {
        prepareImage();
    }
    public void paint (Graphics gDC) {
        gDC.drawImage (img, 72, 22, this);
    }
    public void prepareImage() {
        int width = 255, height = 255;
        int pix[] = new int[255 * 255];
        int index = 0;
        for (int i = 0; i < 255; i++){
            for (int j = 0; j < 255; j++){
                pix[index++] = (255 << 24) | (j << 16) | i;
            }
        }
        img = createImage(
            new MemoryImageSource(width, height, pix, 0, width)
        );
    }
}
```

Tym razem obiekt typu `Image` stworzymy za pomocą metody `createImage` dostępnej w klasie `JApplet` (wywołanie znajduje się wewnątrz metody `prepareImage`, która zostanie wywołana przez metodę `init` w trakcie inicjacji apletu). Jako argument musi występować obiekt implementujący interfejs `ImageProducer`. W tym przypadku używany jest obiekt typu `MemoryImageSource` pozwalający na tworzenie obrazów w pamięci. Obrazem tym jest w rzeczywistości tablica pikseli `pix[]` o rozmiarze 255×255 . Wartości są nadawane pikselom w podwójnej pętli `for`. Każdy piksel opisywany jest przez liczbę typu `int` określającą poszczególne składowe koloru w formacie RGB. Bity 16 – 23 odpowiadają za składową *R*, bity 8 – 15 — za składową *G*, a bity 0 – 7 — za składową *B* (wartość wynikowa została utworzona za pomocą operatora przesunięcia bitowego w prawo `<<` oraz operatora sumy bitowej `|`). Zastosowane rozwiązanie pozwala uzyskać atrakcyjne, wizualnie płynne przejścia tonalne.



8

Interakcja z użytkownikiem

Programy działające w środowisku tekstowym (w wierszu poleceń) mogą być obsługiwane za pomocą systemu wejścia-wyjścia opisanego w rozdziale 6. Natomiast programy uruchamiane w środowisku graficznym wymagają bardziej złożonych działań. Między innymi muszą reagować na zdarzenia związane z obsługą myszy. To główny temat rozdziału 8. Zostanie pokazane, jak reagować na kliknięcia i przemieszczanie kursora wewnątrz okna. Powstanie też prawdziwie obiektowy aplet umożliwiający rysowanie figur geometrycznych.

Choć jako przykłady będą używane aplety, to opisane techniki w równym stopniu dotyczą aplikacji z interfejsem graficznym, które zostaną przedstawione w kolejnym rozdziale. A zatem prezentowane w tym rozdziale kody obsługi myszy mogą być stosowane również w aplikacjach.

Obsługa myszy

Aby wprowadzić interakcję z użytkownikiem, musimy nauczyć się, jak reagować na kliknięcie klawisza myszy czy też przesunięcie jej kursora. Nasz aplet będzie w tym celu implementował tzw. interfejs, którym będzie klasa `MouseListener`¹. Nie mamy niestety miejsca na bliższe zajęcie się interfejsami. W tej chwili ważne jest dla nas to, że jeżeli nasz aplet ma implementować wymieniony wyżej interfejs, trzeba będzie w nim zdefiniować pięć metod:

- ❑ `public void mousePressed (MouseEvent evt),`
- ❑ `public void mouseExited (MouseEvent evt),`
- ❑ `public void mouseEntered (MouseEvent evt),`
- ❑ `public void mouseReleased (MouseEvent evt),`
- ❑ `public void mouseClicked (MouseEvent evt).`

Każda z tych metod jako argument otrzymuje referencję do obiektu `MouseEvent` dostarczającego m.in. informacji o aktualnych współrzędnych, w których znajduje się kursor myszy. Oznacza to, że jeżeli naciśniemy klawisz, wystąpi zdarzenie `MousePressed` i zostanie wykonana powiązana z tym zdarzeniem metoda `mousePressed`. Jaki kod w niej zawrzemy, zależy już tylko od nas. Zobaczmy to na przykładzie. Będziemy jednak obsługiwać zdarzenie `MouseClicked` polegające na kliknięciu (czyli naciśnięciu i zwolnieniu) klawisza myszy.

ĆWICZENIE

8.1 Reakcja na kliknięcie klawisza myszy

Napisz aplet wyświetlający na ekranie współrzędne ostatniego kliknięcia myszą (rysunek 8.1).

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.event.*;

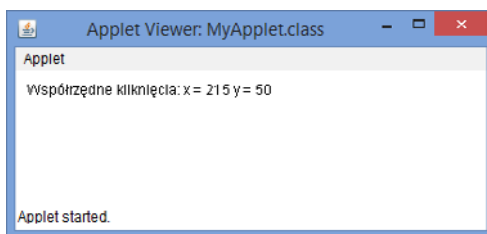
public class MyApplet extends JApplet implements MouseListener{
    private int x, y;
    public void init() {
        addMouseListener (this);
    }
}
```

¹ Nie jest to jedyna możliwość. Inną metodą jest m.in. wykorzystanie klas wewnętrznych, jednak omówienie tego tematu wykracza poza ramy książki.

```
}  
public void paint (Graphics gDC) {  
    gDC.clearRect(0, 0, getSize().width, getSize().height);  
    gDC.drawString ("Współrzędne kliknięcia: x = " + x +  
                    " y = " + y, 10, 10);  
}  
public void mouseClicked (MouseEvent evt) {  
    x = evt.getX();  
    y = evt.getY();  
    repaint();  
}  
public void mousePressed (MouseEvent evt){}  
public void mouseExited (MouseEvent evt){}  
public void mouseEntered (MouseEvent evt){}  
public void mouseReleased (MouseEvent evt){}  
}
```

Rysunek 8.1.

*Aplet podający
współrzędne
kliknięcia myszą*



Do przechowywania współrzędnych kliknięć używane są prywatne składowe *x* i *y*. W metodzie *init* pojawiło się wywołanie metody *addMouseListener* z parametrem *this*. W ten sposób system jest informowany, że aplet będzie obsługiwał zdarzenia związane z myszą. Bez tego wywołania, pomimo zdefiniowania pięciu metod do obsługi myszy, żadna z nich nie zostałaby uruchomiona. Warto na to zwrócić uwagę, gdyż łatwo tu o zwykłe przeoczenie. Trzeba też pamiętać, że mimo iż wykorzystujemy tylko metodę *mouseClicked*, musimy również zadeklarować wszystkie pozostałe metody pochodzące z interfejsu *MouseListener* (*mousePressed*, *mouseExited* itd.) — inaczej kompilacja się nie uda. Wprawdzie nie zawierają one żadnego kodu, ale deklaracje muszą wystąpić.

Współrzędne kliknięcia dostępne są w obiekcie typu *MouseEvent*. Referencja do tego obiektu jest przekazywana jako argument metody *mouseClicked*. Wartości współrzędnych odczytujemy przez wywołanie metod *getX* (współrzędna *x*) i *getY* (współrzędna *y*). Wywołanie metody *repaint* jest równoznaczne z informacją dla apletu, że jego obszar powinien zostać odświeżony. Jest to w tym przypadku jednoznaczne

z wywołaniem metody `paint`. Ponieważ generowany napis przy każdym kliknięciu będzie nieco inny (ze względu na wyświetlanie różnych współrzędnych), przed zastosowaniem metody `drawString` obszar apletu jest czyszczony przez wywołanie metody `clearRect`. Czyści ona obszar zaczynający się we współrzędnych przekazanych za pomocą dwóch pierwszych argumentów i o szerokości oraz wysokości podanych w postaci dwóch kolejnych argumentów (szerokość i wysokość obszaru apletu uzyskujemy dzięki odwołaniom `getSize().width`, `getSize().height`).

Interfejs `MouseListener` obsługuje zdarzenia związane z naciśnięciem klawisza myszy. Co jednak zrobić, gdy chcemy znać współrzędne kursora myszy podczas jego przesuwania? Musimy skorzystać z innego interfejsu — `MouseMotionListener`. Przy czym jedno nie wyklucza drugiego, tzn. możemy korzystać z obu interfejsów jednocześnie. `MouseMotionListener` wymaga zdefiniowania dwóch metod:

- ❑ `public void mouseDragged (MouseEvent evt),`
- ❑ `public void mouseMoved (MouseEvent evt).`

Pierwsza z nich zostaje wywołana, gdy mysz jest przeciągana z wciśniętym klawiszem, druga — kiedy mysz jest tylko przesuwana.

Ć W I C Z E N I E

8.2 Reakcja na zmianę położenia kursora myszy

Napisz aplet wyświetlający na ekranie współrzędne bieżącego położenia kursora myszy.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;

public class MyApplet extends JApplet implements MouseMotionListener {
    String str;
    Font font;
    public void init() {
        font = new Font ("SansSerif", Font.BOLD, 18);
        str = "";
        addMouseMotionListener (this);
    }
    public void paint (Graphics gDC) {
        gDC.setFont (font);
```

```
gDC.clearRect(0, 0, getSize().width, getSize().height);
gDC.drawString (str, 90, 100);
}
public void mouseDragged (MouseEvent evt) {
    int x = evt.getX();
    int y = evt.getY();
    str = "Dragged: x = " + x + " y = " + y;
    repaint();
}
public void mouseMoved (MouseEvent evt) {
    int x = evt.getX();
    int y = evt.getY();
    str = "Moved: x = " + x + " y = " + y;
    repaint();
}
}
```

Ogólna konstrukcja apletu jest podobna do tej z ćwiczenia 8.1, tyle że zdefiniowane zostały metody dla interfejsu `MouseListener`: `mouseDragged` i `mouseMoved`. W obu tych metodach współrzędne kursora myszy pobierane są za pomocą metod `getX` i `getY` pochodzących z obiektu typu `MouseEvent` (dostępnego przez referencję `evt` przekazaną w postaci argumentu). Nieco inaczej została rozwiązana kwestia tworzenia napisu z wartościami współrzędnych. Wyświetlana treść jest przechowywana w polu `str` i przypisywana w metodach `mouseDragged` i `mouseMoved`. To spowodowało, że pola `x` i `y` stały się zbędne. Używane są tylko zmienne pomocnicze `x` i `y`, choć ich zadaniem jest tylko zwiększenie czytelności kodu. Można z nich również zrezygnować, stosując konstrukcje typu:

```
str = "Dragged: x = " + evt.getX() + " y = " + evt.getY();
```

Rysowanie figur (I)

Skoro już wiemy, jak posługiwać się myszą oraz jak rysować figury geometryczne, możemy stworzyć aplet, który po kliknięciu myszą będzie rysował w danym punkcie jakąś figurę geometryczną, np. koło. Prześledźmy, jakie mogłyby być etapy powstawania takiego apletu.

ĆWICZENIE

8.3 Rysowanie kół w miejscu kliknięcia myszą

Napisz najprostszy aplet pozwalający na rysowanie kół w miejscach kliknięcia przyciskiem myszy.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;

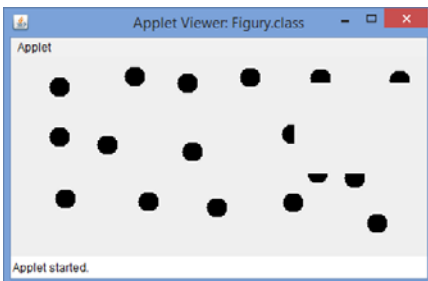
public class Figury extends JApplet implements MouseListener {
    public void init() {
        addMouseListener (this);
    }
    public void mouseClicked (MouseEvent evt) {
        int x = evt.getX();
        int y = evt.getY();
        getGraphics().fillOval (x - 10, y - 10, 20, 20);
    }
    public void mousePressed (MouseEvent evt){}
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
}
```

Jak można zauważyć, nie używamy tu metody `paint`. Aby uzyskać dostęp do graficznego kontekstu urządzenia (który pozwoli rysować w obszarze apletu), tym razem wykorzystujemy metodę `getGraphics` (jest dostępna w klasie `JApplet`). Koła są natomiast rysowane za pomocą poznanej już w rozdziale 7. metody `fillOval`.

Aplet z ćwiczenia 8.3 działa poprawnie, jednak ma pewien mankament. Co stanie się bowiem, jeśli np. zasłonimy go częściowo innym oknem? Efekt może być taki jak na rysunku 8.2.

Rysunek 8.2.

*Z powodu
nieodświeżenia
ekranu część
kółek została
strata*



Okno apletu nie zostało odmalowane, tak więc część, która była zasłonięta, została bezpowrotnie zniszczona. Odmalowanie nie mogło oczywiście nastąpić, skoro nie użyliśmy metody `paint`. Sama metoda `paint` też nam nie pomoże, bo skąd miałaby wiedzieć, gdzie ma rysować kółka? Niezbędne będzie zatem zapamiętywanie ich położeń. W najprostszym przypadku możemy użyć do tego zwykłych zmiennych tablicowych.

ĆWICZENIE

8.4 Użycie tablic do zapamiętywania współrzędnych

Napisz aplet rysujący koła w miejscu kliknięcia myszą. Pamiętaj o odświeżaniu ekranu w metodzie `paint`. Do przechowywania współrzędnych kół użyj zwykłych tablic.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;

public class Figury extends JApplet implements MouseListener {
    private int tabX[], tabY[];
    private int count;
    public void init() {
        count = 0;
        tabX = new int [100];
        tabY = new int [100];
        addMouseListener (this);
    }
    public void paint (Graphics gDC) {
        for (int i = 0; i < count; i++){
            gDC.fillOval(tabX[i] - 10, tabY[i] - 10, 20, 20);
        }
    }
    public void mousePressed (MouseEvent evt) {
        int x = evt.getX();
        int y = evt.getY();
        if(count < tabX.length){
            tabX[count] = x;
            tabY[count] = y;
            count++;
        }
        repaint();
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
}
```

Współrzędne x i y są przechowywane w tablicach `tabX` i `tabY`, natomiast bieżącą liczbę kół przechowuje pole `count`. Inicjacja wszystkich pól odbywa się w metodzie `init` (tworzone tablice mają po 100 elementów). Każde kliknięcie myszą powoduje wywołanie metody `mousePressed`, w której odczytywane są współrzędne kliknięcia. Odczytane dane trafiają do tablic `tabX` i `tabY` (bieżący indeks wskazuje zmienna `count`). Zapis odbywa się tylko wtedy, gdy nie został przekroczony maksymalny rozmiar tablic, co jest badane za pomocą instrukcji warunkowej `if (if(count < tabX.length))`.

W metodzie `paint` znajduje się pętla typu `for`, która odczytuje współrzędne x i y kół i wyświetla je na ekranie za pomocą standardowej metody `fillOval`. Ponieważ w `count` jest zawarta aktualna liczba kół, warunkiem zakończenia pętli jest `i < count`. Dzięki temu zawsze gdy tylko wystąpi konieczność odświeżenia ekranu, zostaną na nim narysowane wszystkie koła.

Aplet z ćwiczenia 8.4 może być urozmaicony w ten sposób, że każde koło będzie miało losowy kolor. Aby uzyskać losową liczbę całkowitą opisującą kolor, można użyć obiektu klasy `Random` i metody `nextInt` (klasa `Random` jest zawarta w pakiecie `java.util`). Zostanie to pokazane w kolejnym ćwiczeniu.

ĆWICZENIE

8.5 Wykorzystanie generatora liczb pseudolosowych

Zmodyfikuj aplet z ćwiczenia 8.4 tak, aby każde koło miało losowo wybrany kolor (rysunek 8.3; na rysunku widoczne są oczywiście jedynie odcienie szarości).

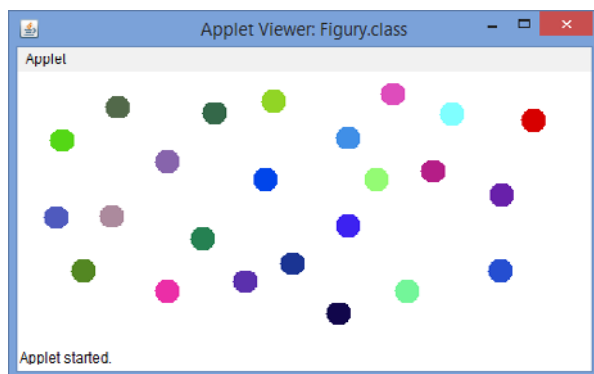
```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;

public class Figury extends JApplet implements MouseListener {
    private int tabX[], tabY[];
    private int count;
    private Color colors[];
    private Random r;
    public void init() {
        count = 0;
        tabX = new int [100];
```



```
        tabY = new int [100];
        colors = new Color[100];
        r = new Random();
        addMouseListener (this);
    }
    public void paint (Graphics gDC) {
        for (int i = 0; i < count; i++){
            gDC.setColor(colors[i]);
            gDC.fillOval(tabX[i] - 10, tabY[i] - 10, 20, 20);
        }
    }
    public void mousePressed (MouseEvent evt) {
        int x = evt.getX();
        int y = evt.getY();
        if(count < tabX.length){
            tabX[count] = x;
            tabY[count] = y;
            colors[count] = new Color (r.nextInt());
            count++;
        }
        repaint();
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
}
```

Rysunek 8.3.
*Aplet rysujący
różnokolorowe
koła*



Konstrukcja apletu pozostała taka sama jak w poprzedniej wersji, ale niezbędne było użycie dodatkowej tablicy przechowującej dane dotyczące kolorów punktów (tablica colors) oraz obiektu generującego liczby pseudolosowe (obiekt r typu Random). Tablica colors oraz obiekt r zostały zainicjowane w metodzie init. Przy każdym kliknięciu oprócz

współrzędnych zapisywane są dane dotyczące koloru. Nowy kolor jest tworzony jako obiekt klasy `Color`, któremu w konstruktorze została przekazana liczba całkowita określająca kolor. Ta liczba jest generowana przez obiekt `r` za pomocą metody `nextInt`. Aby koła „nabrały” kolorów, niezbędna była też zmiana w metodzie `paint`. Osobny kolor dla każdego koła ustawia się za pomocą wywołania `gDC.setColor(colors[i]);`.

Rysowanie figur (II)

Przykłady z ćwiczeń 8.4 i 8.5 miały pewną wadę. Nie można było narysować więcej niż 100 kół. Takie były rozmiary tablic, a przekroczenie wartości 100 było wychwytywane przez instrukcję warunkową w metodzie `mousePressed`. Gdyby tej instrukcji nie było, próba narysowania 101 kółka spowodowałaby wygenerowanie wyjątku `ArrayIndexOutOfBoundsException`. Jak rozwiązać ten problem? Można ustalić bardzo duże rozmiary tablic. Trudno się spodziewać, aby ktoś wprowadził np. 10 000 kół w aplecie. To nie byłoby jednak dobre rozwiązanie. Po pierwsze to marnotrawstwo pamięci, po drugie nie można z kolei przyjmować założenia, że coś, co jest mało prawdopodobne, na pewno się nie zdarzy. Można jednak postąpić inaczej. Wystarczy sprawdzić, czy liczba obiektów nie przekracza dopuszczalnych rozmiarów tablic i w takiej sytuacji tworzyć nowe tablice o większych rozmiarach, do których zostaną przepisane wartości z utworzonych dotychczas tablic. Nie zaszkodzi spróbować.

Ć W I C Z E N I E

8.6 Dynamiczne zwiększanie rozmiaru tablic

Zmień kod z ćwiczenia 8.5 w taki sposób, aby po przekroczeniu rozmiarów tablic tworzone były nowe, o większej pojemności.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;

public class Figury extends JApplet implements MouseListener {
    private int tabX[], tabY[];
```

```
private int count;
private Color colors[];
private Random r;
public void init() {
    count = 0;
    tabX = new int [2];
    tabY = new int [2];
    colors = new Color[2];
    r = new Random();
    addMouseListener (this);
}
public void paint (Graphics gDC) {
    for (int i = 0; i < count; i++){
        gDC.setColor(colors[i]);
        gDC.fillOval(tabX[i] - 10, tabY[i] - 10, 20, 20);
    }
}
public void mousePressed (MouseEvent evt) {
    int x = evt.getX();
    int y = evt.getY();
    tabX[count] = x;
    tabY[count] = y;
    colors[count++] = new Color (r.nextInt());

    if (count >= tabX.length){
        int tempTabX[], tempTabY[];
        Color tempColors[];

        tempTabX = new int [tabX.length * 2];
        tempTabY = new int [tabY.length * 2];
        tempColors = new Color [colors.length * 2];

        for (int i = 0; i < tabX.length; i++){
            tempTabX[i] = tabX[i];
            tempTabY[i] = tabY[i];
            tempColors[i] = colors[i];
        }

        tabX = tempTabX;
        tabY = tempTabY;
        colors = tempColors;
    }
    repaint();
}
public void mouseExited (MouseEvent evt){}
public void mouseEntered (MouseEvent evt){}
public void mouseReleased (MouseEvent evt){}
public void mouseClicked (MouseEvent evt){}
}
```

Po naciśnięciu klawisza myszy w metodzie `mousePressed` zapamiętujemy, tak jak w poprzednim przypadku, nowe współrzędne i nowy kolor. Następnie sprawdzamy, czy w tablicach zmieszczą się dane jeszcze jednego koła. Jeżeli nie, tworzymy nowe tablice o rozmiarach dwukrotnie większych od poprzednich oraz przepisujemy do nich zawartość starych. Dalej referencjom `tabX`, `tabY` i `colors` przypisujemy nowe obiekty wskazywane przez `tempTabX`, `tempTabY` i `tempColors`. Warto uwagi jest to, że nie musimy zwalniać pamięci (nie mamy nawet takiej możliwości) przydzielonej wcześniej na obiekty `tabX`, `tabY` i `colors`. Zajmie się tym za nas tzw. *garbage collector* maszyny wirtualnej.

Rysowanie figur (III)

Aplet z ćwiczenia 8.6 działa prawidłowo i jest to rozwiązanie poprawne, którego można używać w praktyce. Jednocześnie aplet ten jest jednak, można powiedzieć, mało obiektowy, a więc niezupełnie zgodny z filozofią programowania w Javie. Pozostańmy zatem jeszcze przez chwilę przy rysowaniu figur. Zapomnijmy jednakże, przynajmniej częściowo, o wcześniejszych przykładach i spróbujmy zrobić wszystko jeszcze raz od początku.

Najpierw zastanówmy się, co tak naprawdę należy wykonać. Mamy rysować na ekranie kółka, będą nam więc potrzebne obiekty reprezentujące koła. Do opisu kół użyjemy nowej klasy — nazwiemy ją `Circle`. Obiekt `Circle` musi mieć następujące pola:

- ☐ współrzędna *X* środka,
- ☐ współrzędna *Y* środka,
- ☐ kolor.

Do tego potrzebne będą jeszcze metody:

- ☐ konstruktor,
- ☐ metoda rysująca kółko.

ĆWICZENIE

8.7 Tworzenie klasy opisującej koła

Napisz kod klasy opisującej koła. Klasa powinna zawierać metodę umożliwiającą wyświetlenie obiektu na ekranie.

```
import java.awt.*;

public class Circle {
    public int x, y;
    public Color color;
    public Circle (int x, int y, Color color) {
        this.x = x;
        this.y = y;
        this.color = color;
    }
    public void draw (Graphics gDC) {
        gDC.setColor (color);
        gDC.fillOval (x - 10, y - 10, 20, 20);
    }
}
```

W klasie znalazły się pola *x* i *y* opisujące położenie figury na ekranie oraz pole *color* określające jej kolor. Wartości są nadawane polom w trójargumentowym konstruktorze. Klasa *Circle* zawiera również metodę *draw*, której zadaniem jest narysowanie koła za pomocą kontekstu urządzenia otrzymanego w postaci argumentu *gDC*. Rysowanie odbywa się dzięki metodzie *fillOval* po uprzednim ustawieniu koloru za pomocą metody *setColor*.

Obiekty opisywane klasą *Circle* będą musiały być gdzieś przechowywane. Potrzebna więc będzie kolejna klasa będąca pewnego rodzaju kontenerem dla kół — niech nazywa się *CircleDatabase*. Moglibyśmy oczywiście stworzyć (podobnie jak w ćwiczeniu 8.6) tablice i zmieniać ich wielkość, zrobimy to jednak inaczej, za to dużo prościej. Skorzystamy z klasy *Vector*, która służy do przechowywania obiektów innych klas (może być właśnie kontenerem dla innych obiektów). Metodą *addElement* możemy dodać jakiś obiekt, natomiast metodą *elementAt* — pobrać go.

Ć W I C Z E N I E

8.8 Klasa przechowująca obiekty typu Circle

Napisz klasę `CircleDatabase` umożliwiającą przechowywanie obiektów typu `Circle`.

```
import java.awt.Graphics;
import java.util.Vector;

public class CircleDatabase extends Vector {
    public void drawAll(Graphics gDC){
        for (int i = 0; i < this.size(); i++){
            ((Circle)this.elementAt(i)).draw(gDC);
        }
    }
}
```

Klasa `CircleDatabase` jest klasą potomną klasy `Vector`. To znaczy, że będzie można w niej przechowywać obiekty dowolnego typu, w tym przypadku obiekty typu `Circle`². Jediną metodą jest metoda `drawAll`, która będzie rysowała wszystkie zawarte w obiekcie koła na urządzeniu reprezentowanym przez kontekst `gDC` przekazany w postaci argumentu.

Wyjaśnienia wymaga zapewne konstrukcja:

```
((Circle)this.elementAt(i)).draw(gDC);
```

Jak pamiętamy, konwertowaliśmy już typy, z tym że były to typy arytmetyczne. Wtedy była to konwersja typu `double` do typu `int`. Tutaj jest podobnie — należy wykonać konwersję, gdyż metoda `elementAt` zwraca w wyniku referencję do typu `Object`. Musi tak być, ponieważ `Vector` może przechowywać elementy dowolnego typu, a nie tylko `Circle`. Aby więc możliwe było wywołanie metody `draw` z klasy `Circle`, trzeba wykonać jawną konwersję z typu `Object` do typu `Circle`³.

² Ze względu na wprowadzenie w Java 1.5 obsługi typów uogólnionych (ang. *generics*) deklaracja powinna uwzględniać to, jaki typ danych będzie przechowywany. Z formalnego punktu widzenia lepszym rozwiązaniem byłoby więc użycie konstrukcji w postaci `public class CircleDatabase extends Vector <Circle>`. To jednak wykracza poza ramy tematyczne książki. W praktyce jednak jedynym mankamentem tego przykładu będzie generowanie ostrzeżenia podczas kompilacji całego przykładu (włącznie z klasą `Figury`). Nie będzie to jednak miało żadnego wpływu na działanie apletu.

³ Nie byłyby to konieczne w przypadku korzystania z typów uogólnionych.

Czas przygotować najnowszą wersję apletu Figury, która będzie korzystała z tak przygotowanych klas `Circle` i `CircleDatabase`.

ĆWICZENIE

8.9 Wykorzystanie klas `Circle` i `CircleDatabase`


Napisz aplet umożliwiający rysowanie kolorowych kół na ekranie. Użyj klas `Circle` i `CircleDatabase` zaprezentowanych w ćwiczeniach 8.7 i 8.8.

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;
import java.util.Random;

public class Figury extends JApplet implements MouseListener {
    private CircleDatabase database;
    private Random r;
    public void init()
    {
        database = new CircleDatabase();
        r = new Random();
        addMouseListener (this);
    }
    public void paint (Graphics gDC)
    {
        database.drawAll(gDC);
    }
    public void mousePressed (MouseEvent evt)
    {
        int x = evt.getX();
        int y = evt.getY();
        Color color = new Color (r.nextInt());
        database.addElement (new Circle(x, y, color));
        repaint();
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
}
```

Jak można zauważyć, klasa `Figury` bardzo się uprościła w porównaniu z wcześniejszymi przykładami. Tworzymy w niej obiekt `database` typu `CircleDatabase` i po każdym kliknięciu myszą dodajemy do niego nowy obiekt typu `Circle`. W metodzie `paint`, czyli przy każdej konieczności odświeżenia ekranu, każemy obiektowi `database` narysować

wszystkie zawarte w nim elementy. Wywołujemy w tym celu metodę `drawAll`, której — w postaci argumentu — przekazywany jest obiekt `gDC` określający obszar apletu. Nic więcej nas nie interesuje — wyświetlaniem zajmuje się obiekt klasy `CircleDatabase`. Pobiera on po kolei wszystkie zawarte w nim kółka i wywołuje metodę `draw` każdego z nich. Rysowanie odbywa się więc w rzeczywistości dopiero w metodzie `draw` klasy `Circle`.



9

Aplikacje z interfejsem graficznym

Tworzenie okna aplikacji

W rozdziale 8. poznaliśmy różnicę między aplikacją a apletem. Wiemy zatem, że aplikacja jest samodzielnym programem, który do uruchomienia potrzebuje tylko maszyny wirtualnej Javy. Nie jest natomiast potrzebna do tego żadna przeglądarka czy inny program. Napiszmy więc kod aplikacji z interfejsem graficznym, której zadaniem będzie wyświetlenie w oknie napisu.

Podobnie jak w przypadku apletów, możemy w tym celu użyć klas pochodzących z pakietu AWT lub bardziej nowoczesnego — Swing. W tym pierwszym przypadku powinniśmy skorzystać z klasy `Frame`, a w drugim — `JFrame`. Przykłady z tego rozdziału będą uwzględniały użycie klasy `JFrame` z pakietu `javax.swing`.

ĆWICZENIE

9.1 Graficzna aplikacja wyświetlająca napis

Napisz w Javie aplikację, która utworzy na ekranie okno i wyświetli w nim napis.

```
import javax.swing.JFrame;
import java.awt.Graphics;

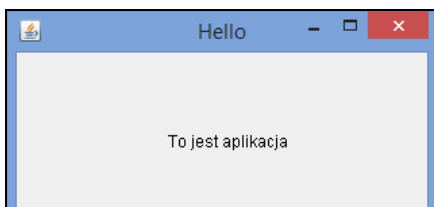
public class HelloApp extends JFrame {
    public HelloApp () {
        super("Hello");
        setSize(320, 200);
        setVisible(true);
    }
    public void paint(Graphics gDC) {
        gDC.clearRect(0, 0, getSize().width, getSize().height);
        gDC.drawString ("To jest aplikacja", 120, 100);
    }
    public static void main(String args[]) {
        new HelloApp();
    }
}
```

Sposób rysowania ciągu znaków jest dokładnie taki sam jak w przypadku apletów. W metodzie `paint` ekran jest czyszczony za pomocą wywołania `clearRect` (w Javie powyżej wersji 1.6 nie jest to konieczne), napis pojawia się dzięki metodzie `drawString`. W konstruktorze są natomiast wykonywane trzy czynności. Po pierwsze wywoływany jest konstruktor klasy bazowej i jest mu przekazywany tytuł okna aplikacji. Po drugie za pomocą metody `setSize` są ustalane rozmiary okna (320×200 pikseli). Po trzecie okno jest wyświetlane dzięki metodzie `setVisible`, która otrzymuje argument o wartości `true`. Aby okno mogło być wyświetlone, po uruchomieniu programu musi powstać obiekt typu `HelloApp`. Jest on zatem tworzony w metodzie `main`.

Jeśli teraz w konsoli uruchomimy aplikację `HelloApp`, pisząc `java HelloApp`, na ekranie pojawi się okno przedstawione na rysunku 9.1.

Rysunek 9.1.

*Aplikacja w Javie
wyświetlająca
na ekranie okno
z napisem*



Wraz z platformą Java 5 (1.5) pojawił się nowy model obsługi zdarzeń dla biblioteki *Swing*, w którym operacji związanych z komponentami nie należy obsługiwać bezpośrednio. Zamiast tego powinny one trafić do kolejki zdarzeń. Dotyczy to również samego uruchamiania aplikacji. Dlatego począwszy od Java 5, w aplikacjach korzystających ze *Swing* w metodzie *main* należy używać metody *invokeLater* klasy *SwingUtilities*. Dla klasy *HelloApp* kod metody powinien wyglądać następująco:

```
public static void main(String args[]) {  
    SwingUtilities.invokeLater(new Runnable() {  
        public void run() {  
            new HelloApp();  
        }  
    });  
}
```

W kolejnych ćwiczeniach należy stosować to rozwiązanie¹. Począwszy od ćwiczenia 9.4, treść metody *main* nie będzie jednak prezentowana, aby niepotrzebnie nie wydłużać listingów (jej zawartość dla ćwiczeń od 9.3 do 9.10 będzie identyczna).

Warto zwrócić uwagę, że tak powstałe okno da się co prawda zamknąć, ale program pozostanie aktywny (w przypadku użycia klasy *Frame* zamknięcie w tradycyjny sposób w ogóle nie byłoby możliwe). Zostanie zatrzymany dopiero po przełączeniu się na konsolę (wiersz poleceń), z której uruchamialiśmy tę aplikację, i wciśnięciu kombinacji klawiszy *Ctrl+C*.

Istnieją dwa rozwiązania tego problemu. Pierwsze to wywołanie metody *setDefaultCloseOperation* i przekazanie jej jako argumentu wartości *JFrame.EXIT_ON_CLOSE*. Wystarczy więc w konstruktorze umieścić instrukcję:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Ta metoda jest dostępna wtedy, gdy korzystamy z klasy *JFrame*. W przypadku klasy *Frame* nie ma takiej możliwości.

¹ W wyniku kompilacji będą wtedy powstawały dwa pliki: plik główny z kodem aplikacji oraz dodatkowy plik (o nazwie *HelloApp\$1.class* lub podobnej) z kodem klasy anonimowej dziedziczącej po interfejsie *Runnable*. Zagadnienia interfejsów i klas anonimowych zostały szerzej opisane we wspominatej już publikacji *Java. Praktyczny kurs*.

Metoda druga to obsługa zdarzeń — podobnie jak miało to miejsce w przypadku obsługi myszy w rozdziale 8. Można zatem dodać do klasy obsługę interfejsu `WindowListener` oraz zdefiniowane w nim metody. Są to:

- ❑ `windowClosing` — wykonywana, gdy okno jest zamykane,
- ❑ `windowClosed` — wykonywana po zamknięciu okna,
- ❑ `windowOpened` — wykonywana po otwarciu okna,
- ❑ `windowIconified` — wykonywana po minimalizacji okna,
- ❑ `windowDeiconified` — wykonywana po maksymalizacji okna,
- ❑ `windowActivated` — wykonywana po aktywacji okna,
- ❑ `windowDeactivated` — wykonywana, gdy okno staje się nieaktywne.

Ć W I C Z E N I E

9.2 Reakcja na zdarzenia związane z oknem

Napisz aplikację wyświetlającą na ekranie okno z napisem. Po kliknięciu przycisku zamykającego okno aplikacja powinna zakończyć działanie. Użyj metod interfejsu `WindowListener`.

```
import javax.swing.*;
import java.awt.Graphics;
import java.awt.event.*;

public class HelloApp extends JFrame implements WindowListener {
    public HelloApp () {
        super("Moja pierwsza aplikacja");
        addWindowListener(this);
        setSize(320, 200);
        setVisible(true);
    }
    public void paint(Graphics gDC) {
        gDC.clearRect(0, 0, getSize().width, getSize().height);
        gDC.drawString ("To jest aplikacja.", 120, 100);
    }
    public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new HelloApp();
            }
        });
    }
    public void windowClosing(WindowEvent e) {
        dispose();
    }
}
```

```

public void windowClosed(WindowEvent e){}
public void windowOpened(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public void windowDeiconified(WindowEvent e){}
public void windowActivated(WindowEvent e){}
public void windowDeactivated(WindowEvent e){}
}

```

Zamknięcie aplikacji jest wykonywane poprzez wywołanie metody `dispose` w metodzie `windowClosing` (podobny efekt można również osiągnąć, stosując wywołanie `System.exit(0)`). Ważne jest, aby w konstruktorze obiektu znalazła się linia `addWindowListener(this)` powodująca, że okno zaczyna „słuchać” zdarzeń z nim związanych. Jest również wywoływany konstruktor klasy bazowej (`super`), któremu w postaci argumentu przekazywany jest ciąg znaków pojawiający się na pasku tytułu okna. Alternatywnie można wywołać ten konstruktor bez argumentów (`super()`), a tytuł ustawić za pomocą metody `setTitle`:

```
setTitle("Tytuł okna");
```

lub też pozostawić okno z pustym paskiem tytułu. Samo rysowanie napisu odbywa się w dobrze znany sposób, wielokrotnie wykorzystany przez nas przy pisaniu apletów.



Oba przedstawione przykłady ilustrują możliwość bezpośredniego rysowania w obszarze okna aplikacji za pomocą metody `paint`. W „prawdziwych” aplikacjach korzystających z pakietu `Swing` z reguły nie rysujemy bezpośrednio w oknie, ale korzystamy z komponentów i kontrolerek udostępnianych przez `Swing` (część z nich została opisana w kolejnym rozdziale).

Budowanie menu

Większość aplikacji posiada menu, niestety naszej brakuje na razie tego elementu. Sprawdźmy zatem, w jaki sposób owo menu dodać. W tym celu należy bowiem wykonać kilka czynności. Po pierwsze trzeba utworzyć obiekt klasy `JMenuBar` i dodać go do okna za pomocą metody `setJMenuBar`. Dopiero do tego obiektu będziemy mogli dodawać kolejne menu. Po drugie dla każdego menu należy utworzyć obiekt klasy `JMenu` oraz dla każdej pozycji w menu — obiekt klasy `JMenuItem`. Najlepiej zobaczyć, jak to wygląda na konkretnym przykładzie.

ĆWICZENIE

9.3 Dodanie menu do okna aplikacji

Napisz kod aplikacji zawierającej menu *Plik*, tak jak widać na rysunku 9.2.

```
import javax.swing.*;

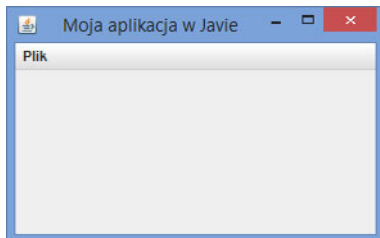
public class Aplikacja extends JFrame {
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Moja aplikacja w Javie");

        JMenuBar menuBar = new JMenuBar();
        JMenu menu1 = new JMenu("Plik");
        menuBar.add(menu1);
        setJMenuBar(menuBar);

        setSize(320, 200);
        setVisible(true);
    }
    public static void main(String args[]) {
        // kod metody main
    }
}
```

Rysunek 9.2.

*Aplikacja
z dodanym
menu Plik*



Obsługa zamykania okna została zapewniona przez wywołanie metody `setDefaultCloseOperation`, dzięki czemu nie było konieczności implementowania interfejsu `WindowListener`. Pasek menu został utworzony jako obiekt typu `JMenuBar`, a jedyne dostępne menu (*Plik*) jako obiekt typu `JMenu`. Parametrem konstruktora klasy `JMenu` była nazwa menu. Menu zostało dodane do paska menu za pomocą metody `add` (`menuBar.add(menu1)`), a pasek menu do okna aplikacji za pomocą metody `setJMenuBar` (`setJMenuBar(menuBar)`). Po uruchomieniu aplikacji zobaczymy więc okno wyposażone w pasek menu z menu *Plik* (rysunek 9.2).

Do menu utworzonego w ćwiczeniu 9.3 należałoby jednak dodać jakąś pozycję. Niech będzie to *Zamknij*. Będziemy mogli wykorzystać ją do końca pracy z aplikacją.

Ć W I C Z E N I E

9.4 Dodawanie pozycji do menu

Do menu otrzymanego w ćwiczeniu 9.3 dodaj pozycję o nazwie *Zamknij*, tak jak na rysunku 9.3.

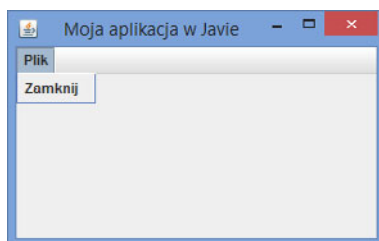
```
import javax.swing.*;

public class Aplikacja extends JFrame {
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Moja aplikacja w Javie");

        JMenuBar menuBar = new JMenuBar();
        JMenu menu1 = new JMenu("Plik");
        JMenuItem menuItem1 = new JMenuItem("Zamknij");
        menu1.add(menuItem1);
        menuBar.add(menu1);
        setJMenuBar(menuBar);

        setSize(320, 200);
        setVisible(true);
    }
    public static void main(String args[]) {
        // kod metody main
    }
}
```

Rysunek 9.3.
Okno aplikacji
z menu
z ćwiczenia 9.3



Pozycje menu są odzwierciedlane przez obiekty typu `JMenuItem`. Dlatego kod został uzupełniony o instrukcję tworzącą nowy obiekt tego typu. Jako argument konstruktora została użyta nazwa pozycji menu (`new JMenuItem("Zamknij")`). Pozycja została przypisana zmiennej po-

mocniczej menuItem1 i dodana do menu za pomocą wywołania metody add(menu1.add(menuItem1)).

Jeśli przechowywanie odwołania do pozycji menu nie jest konieczne (tak jest w omawianym przykładzie), można też zrezygnować z tworzenia zmiennej pomocniczej i umieścić wywołanie konstruktora bezpośrednio w metodzie add:

```
menu1.add(new JMenuItem("Zamknij"));
```

Menu utworzone w ćwiczeniu 9.3 jest niestety nieaktywne, tzn. po jego wybraniu nic się nie dzieje. Trzeba bowiem dopiero stworzyć odpowiednie procedury jego obsługi. Jak to zrobić? Jednym ze sposobów jest dodanie do klasy Aplikacja odpowiedniego interfejsu — ActionListener. To z kolei wymusi zdefiniowanie metody actionPerformed. Metoda ta będzie wywoływana (m.in.) w momencie wybrania dowolnej pozycji z menu. Należy w niej sprawdzić rodzaj zdarzenia i odpowiednio na niego zareagować.

Ć W I C Z E N I E

9.5 Obsługa menu

Uzupełnij kod z ćwiczenia 9.4 w taki sposób, aby po wybraniu z menu *Plik* pozycji *Zamknij* następowało zakończenie pracy aplikacji.

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    public Aplikacja() {
        super("Aplikacja z menu");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenuBar menuBar = new JMenuBar();
        JMenu menu1 = new JMenu("Plik");

        JMenuItem menuItem1 = new JMenuItem("Zamknij");
        menuItem1.setActionCommand("Zamknij");
        menuItem1.addActionListener(this);

        menu1.add(menuItem1);
        menuBar.add(menu1);
        setJMenuBar(menuBar);

        setSize(320, 200);
```



```

        setVisible(true);
    }
    public void actionPerformed(ActionEvent e){
        String cmd = e.getActionCommand();
        if ("Zamknij".equals(cmd)){
            dispose();
        }
    }
    public static void main(String args[]) {
        // kod metody main
    }
}

```

W konstruktorze klasy *Aplikacja* znalazły się dwa ważne wiersze:

```

menuItem1.setActionCommand("Zamknij");
menuItem1.addActionListener(this);

```

Pierwszy to ustalenie komendy powiązanej z daną pozycją menu. Czyli menu o nazwie *Zamknij* reprezentowanemu przez obiekt *menuItem1* została przypisana komenda o nazwie *Zamknij* (zatem i komenda, i nazwa menu są takie same). Drugi wiersz oznacza, że chcemy, aby obiekt klasy *Aplikacja* reagował na zdarzenia związane z obiektem menu *menuItem1*. Skoro tak, to w momencie wybrania dowolnej pozycji z tego menu zostanie wywołana metoda *actionPerformed* obiektu klasy *Aplikacja*.

Argumentem metody *actionPerformed* jest obiekt typu *ActionEvent*, który pozwala na określenie rodzaju zdarzenia. Korzystamy tu z metody *getActionCommand* zwracającej komendę powiązaną z obiektem wywołującym zdarzenie. A zatem jeśli wybrana została pozycja menu, której przypisano komendę *Zamknij*, to wywołanie *getActionCommand* zwróci ciąg znaków *Zamknij*². Wystarczy więc użyć instrukcji warunkowej *if* i metody *equals*, aby sprawdzić, czy zdarzenie zostało spowodowane przez wybranie pozycji *Zamknij*. Jeśli tak jest, zamykamy okno aplikacji za pomocą metody *dispose*.

² W rzeczywistości w tym przypadku użycie metody *setActionCommand* nie było konieczne (ale jej pomijanie nie jest zalecane), gdyż jeśli komenda nie została jawnie ustalona, to wywołanie *getActionCommand* zwraca ciąg znaków będący nazwą menu (przekazany w konstruktorze klasy *JMenuItem*). A zatem skoro nazwa menu brzmi *Zamknij* i ten ciąg jest używany w metodzie *actionPerformed*, to bez użycia *setActionCommand* program również by działał poprawnie.

Aplikacja okienkowa nie jest skazana wyłącznie na obsługę graficznego interfejsu użytkownika. Ponieważ prezentowane programy uruchamiamy z konsoli, to na tej konsoli możemy np. wyświetlić dane.

Ć W I C Z E N I E

9.6 Aplikacja okienkowa i obsługa konsoli

Utwórz okno aplikacji zawierającej kilka menu z kilkoma pozycjami. Wybranie danej pozycji powinno spowodować wyświetlenie jej nazwy w konsoli, z której została uruchomiona aplikacja (rysunek 9.4).

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    public Aplikacja() {
        super("Aplikacja z menu");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenuBar menuBar = new JMenuBar();
        JMenu menu1 = new JMenu("Plik");
        JMenu menu2 = new JMenu("Menu 2");
        JMenu menu3 = new JMenu("Menu 3");

        JMenuItem menuItem11 = new JMenuItem("Zamknij");
        menuItem11.addActionListener(this);
        menu1.add(menuItem11);

        JMenuItem menuItem21 = new JMenuItem("Pozycja 2 1");
        JMenuItem menuItem22 = new JMenuItem("Pozycja 2 2");
        menuItem21.addActionListener(this);
        menuItem22.addActionListener(this);
        menu2.add(menuItem21);
        menu2.add(menuItem22);

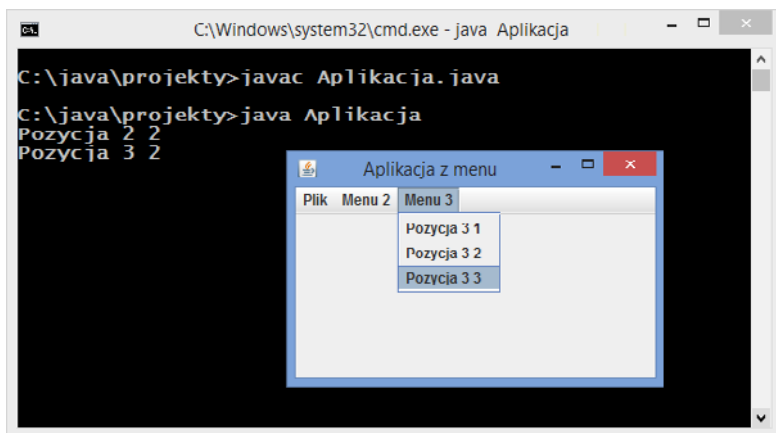
        JMenuItem menuItem31 = new JMenuItem("Pozycja 3 1");
        JMenuItem menuItem32 = new JMenuItem("Pozycja 3 2");
        JMenuItem menuItem33 = new JMenuItem("Pozycja 3 3");
        menuItem31.addActionListener(this);
        menuItem32.addActionListener(this);
        menuItem33.addActionListener(this);
        menu3.add(menuItem31);
        menu3.add(menuItem32);
        menu3.add(menuItem33);

        menuBar.add(menu1);
        menuBar.add(menu2);
        menuBar.add(menu3);
        setJMenuBar(menuBar);
    }
}
```

```

        setSize(320, 200);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        System.out.println(cmd);
        if ("Zamknij".equals(cmd)){
            dispose();
        }
    }
    public static void main(String args[]) {
        // kod metody main
    }
}

```



Rysunek 9.4. Aplikacja zawierająca kilka menu

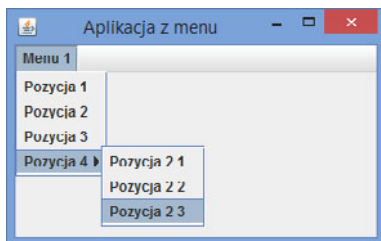
Zasada działania kodu pozostała tu taka sama jak we wcześniejszych ćwiczeniach. Rozbudowany został po prostu pasek menu oraz pojawiło się więcej pozycji. Wybranie dowolnej pozycji będzie powodowało wywołanie metody `actionPerformed` w klasie `Aplikacja` (dzieje się tak ze względu na serię wywołań w postaci `obiektpozycji.menu.add(ActionListener(this));`). Nazwa wybranej pozycji jest wyświetlana w konsoli za pomocą standardowej i doskonale znanej instrukcji `System.out.println(cmd);`.

Wielopoziomowe menu

Dotychczas tworzyliśmy menu jednopoziomowe. Nic jednak nie stoi na przeszkodzie, aby stworzyć takie, które ma kilka poziomów, tzn. po wybraniu jednej pozycji będzie się pojawiało kolejne menu. Zostało to zaprezentowane na rysunku 9.5. Tego typu konstrukcję tworzy się analogicznie do menu jednopoziomowego. Różnica jest taka, że menu, które ma być na drugim lub kolejnym poziomie, dodaje się nie do paska menu (obiektu klasy `JMenuBar`), a do menu z wcześniejszego poziomu (obiektu klasy `JMenu`). Najlepiej zobaczyć na konkretnym przykładzie, jak to działa.

Rysunek 9.5.

*Aplikacja
z wielopoziomowym
menu*



Ć W I C Z E N I E

9.7 Budowa wielopoziomowego menu

Napisz aplikację posiadającą wielopoziomowe menu, takie jak na rysunku 9.5. Po wybraniu dowolnej pozycji wypisz jej nazwę na konsoli (w wierszu poleceń, z którego aplikacja została uruchomiona).

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    public Aplikacja() {
        super("Aplikacja z menu");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenu menu1 = new JMenu("Menu 1");
        JMenuItem menuItem1 = new JMenuItem("Pozycja 1");
        JMenuItem menuItem2 = new JMenuItem("Pozycja 2");
        JMenuItem menuItem3 = new JMenuItem("Pozycja 3");
        menuItem1.addActionListener(this);
        menuItem2.addActionListener(this);
        menuItem3.addActionListener(this);
        menu1.add(menuItem1);
```

```

menu1.add(menuItem2);
menu1.add(menuItem3);

JMenu menu2 = new JMenu("Pozycja 4");
JMenuItem menuItem1 = new JMenuItem("Pozycja 2 1");
JMenuItem menuItem2 = new JMenuItem("Pozycja 2 2");
JMenuItem menuItem3 = new JMenuItem("Pozycja 2 3");
menuItem1.addActionListener(this);
menuItem2.addActionListener(this);
menuItem3.addActionListener(this);
menu2.add(menuItem1);
menu2.add(menuItem2);
menu2.add(menuItem3);

menu1.add(menu2);

JMenuBar menuBar = new JMenuBar();
menuBar.add(menu1);
setJMenuBar(menuBar);

setSize(320, 200);
setVisible(true);
}
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    System.out.println(cmd);
}
public static void main(String args[]) {
    // kod metody main
}
}

```

Istnieje również możliwość stworzenia menu umożliwiającego zaznaczanie poszczególnych pozycji. Nie wymaga to dużych modyfikacji w kodzie. Należy jedynie zamiast obiektów klasy `JMenuItem` użyć klasy `JCheckboxMenuItem`.

ĆWICZENIE

9.8 Menu umożliwiające zaznaczanie pozycji

Napisz aplikację zawierającą menu umożliwiające zaznaczanie poszczególnych pozycji, tak jak na rysunku 9.6.

```

import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame {

```

```

public Aplikacja() {
    super("Aplikacja z menu");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JMenu menu1 = new JMenu("Menu 1");
    JCheckBoxMenuItem menuItem1 = new JCheckBoxMenuItem("Pozycja 1");
    JCheckBoxMenuItem menuItem2 = new JCheckBoxMenuItem("Pozycja 2");
    JCheckBoxMenuItem menuItem3 = new JCheckBoxMenuItem("Pozycja 3");
    menu1.add(menuItem1);
    menu1.add(menuItem2);
    menu1.add(menuItem3);

    JMenu menu2 = new JMenu("Pozycja 4");
    JCheckBoxMenuItem menu2item1 = new JCheckBoxMenuItem("Pozycja 2 1");
    JCheckBoxMenuItem menu2item2 = new JCheckBoxMenuItem("Pozycja 2 2");
    JCheckBoxMenuItem menu2item3 = new JCheckBoxMenuItem("Pozycja 2 3");
    menu2.add(menu2item1);
    menu2.add(menu2item2);
    menu2.add(menu2item3);

    menu1.add(menu2);

    JMenuBar menuBar = new JMenuBar();
    menuBar.add(menu1);
    setJMenuBar(menuBar);

    setSize(320, 200);
    setVisible(true);
}

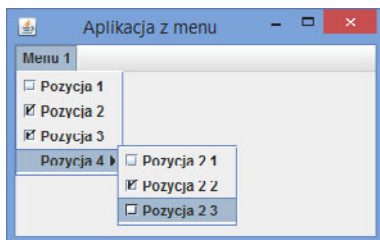
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    System.out.println(cmd);
}

public static void main(String args[]) {
    // kod metody main
}
}

```

Rysunek 9.6.

*Dwupoziomowe
menu umożliwiające
zaznaczanie
poszczególnych
pozycji*



Okna dialogowe

Nieodłącznym elementem graficznego interfejsu użytkownika są wszelkiego rodzaju okna dialogowe. W pakiecie Swing znajduje się klasa `JOptionPane`, dzięki której można wyświetlać różne typy takich okien. Wystarczy skorzystać z zawartych w niej metod. Podstawowe metody to:

- ❑ `showConfirmDialog` — wyświetla okno dialogowe pozwalające na potwierdzanie operacji,
- ❑ `showInputDialog` — wyświetla okno dialogowe pozwalające na wprowadzanie danych,
- ❑ `showMessageDialog` — wyświetla okno dialogowe z informacją tekstową.

Wszystkie te metody występują w kilku przeciążonych wersjach. Między innymi dostępna jest wersja przyjmująca następujące argumenty: okno nadrzędne, komunikat, tytuł okna, typ okna dialogowego. Okno nadrzędne to okno, z którego jest wyświetlane okno dialogowe (np. główne okno aplikacji). Typ okna może być określony przez poniższe wartości:

- ❑ `JOptionPane.ERROR_MESSAGE` — okno błędu,
- ❑ `JOptionPane.INFORMATION_MESSAGE` — okno informacyjne,
- ❑ `JOptionPane.WARNING_MESSAGE` — okno z ostrzeżeniem,
- ❑ `JOptionPane.QUESTION_MESSAGE` — okno z pytaniem,
- ❑ `JOptionPane.PLAIN_MESSAGE` — okno z wiadomością bez wyróżników.

Ć W I C Z E N I E

9.9 Menu i okna dialogowe

Napisz aplikację zawierającą menu. Po wybraniu dowolnej pozycji z menu wyświetl jej nazwę w nowym oknie dialogowym (rysunek 9.7).

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener{
    public Aplikacja() {
        super("Aplikacja z menu");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JMenu menu1 = new JMenu("Plik");
```

```
JMenuItem menuItem1 = new JMenuItem("Pozycja 1");
JMenuItem menuItem2 = new JMenuItem("Pozycja 2");
menuItem1.addActionListener(this);
menuItem2.addActionListener(this);
menu1.add(menuItem1);
menu1.add(menuItem2);

JMenuBar menuBar = new JMenuBar();
menuBar.add(menu1);
setJMenuBar(menuBar);

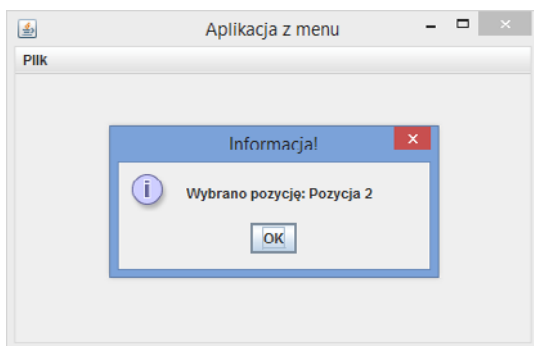
setSize(320, 200);
setVisible(true);
}

public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    JOptionPane.showMessageDialog(this, "Wybrano pozycję: " + cmd,
        "Informacja!", JOptionPane.INFORMATION_MESSAGE);
}

public static void main(String args[]) {
    // kod metody main
}
}
```

Rysunek 9.7.

Po wybraniu
pozycji z menu
zostanie
wyświetlone
nowe okno
dialogowe
z jej nazwą



Powstało tu menu Plik z dwiema pozycjami. Wybór jednej z nich spowoduje wywołanie metody `actionPerformed`. W tej metodzie jest pobierana komenda przypisana danej pozycji. W tym przypadku (ponieważ wcześniej nie została użyta metoda `setActionCommand`) komenda jest równoznaczna z nazwą pozycji. Informacja o tym, która pozycja została wybrana, jest wyświetlana w oknie dialogowym (rysunek 9.7) za pomocą metody `showMessageDialog` z klasy `JOptionPane`.

Istnieją okna dialogowe pozwalające na wykonywanie bardziej złożonych operacji. Przykładem jest okno wyboru pliku. Może być wyświetlone dzięki klasie `JFileChooser` z pakietu `Swing`. Należy utworzyć obiekt tej klasy, a następnie wywołać jego metodę `showOpenDialog` (wyświetla okno wyboru pliku do otwarcia) lub `showSaveDialog` (wyświetla okno wyboru pliku do zapisu), przekazując jej w postaci argumentu wskazanie do okna nadrzędnego (zwykle do okna aplikacji, może to być też wartość `null`). Wynikiem działania każdej z tych metod jest wartość określająca, jaka akcja została wykonana przez użytkownika:

- ❑ `JFileChooser.CANCEL_OPTION` — wybór został anulowany,
- ❑ `JFileChooser.APPROVE_OPTION` — wybór został zatwierdzony,
- ❑ `JFileChooser.ERROR_OPTION` — wystąpił błąd lub zamknięto okno bez dokonania wyboru.

Wskazany plik może być pobrany dzięki metodzie `getSelectedFile`. Zwraca ona obiekt typu `File`. Aby zatem odczytać nazwę, trzeba skorzystać z dodatkowego wywołania metody `getName`. Przykładowo jeśli zmienna `fc` zawiera obiekt typu `FileChooser`:

```
JFileChooser fc = new FileChooser();
```

została wywołana metoda `showOpenDialog`:

```
fc.showOpenDialog();
```

oraz sprawdzono, że wartością zwróconą przez `showOpenDialog` jest `JFileChooser.APPROVE_OPTION`, to nazwę pliku można odczytać za pomocą instrukcji:

```
fc.getSelectedFile().getName();
```

ĆWICZENIE

9.10 Wyświetlenie okna wyboru plików

Napisz aplikację graficzną, która po kliknięciu przycisku wyświetli okno służące do wyboru pliku (rysunek 9.8). Nazwa wybranego pliku powinna zostać wyświetlona za pomocą informacyjnego okna dialogowego.

```
import javax.swing.*;  
import java.awt.event.*;
```

```
public class Aplikacja extends JFrame implements ActionListener{  
    public Aplikacja() {
```

```

super("Aplikacja z menu");
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JMenu menu1 = new JMenu("Plik");
JMenuItem menuItem1 = new JMenuItem("Wybierz plik");
JMenuItem menuItem2 = new JMenuItem("Zamknij");
menuItem1.setActionCommand("chooseFile");
menuItem2.setActionCommand("close");
menuItem1.addActionListener(this);
menuItem2.addActionListener(this);
menu1.add(menuItem1);
menu1.add(menuItem2);

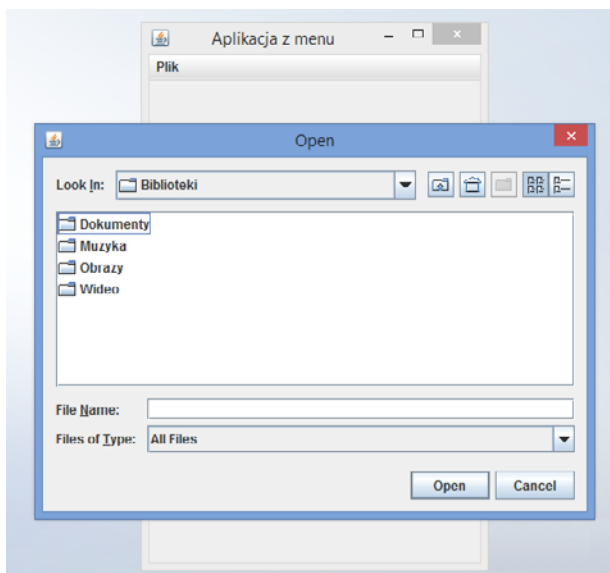
JMenuBar menuBar = new JMenuBar();
menuBar.add(menu1);
setJMenuBar(menuBar);

setSize(320, 200);
setVisible(true);
}
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if("choosefile".equals(cmd)){
        JFileChooser fc = new JFileChooser();
        if(fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
            String nazwaPliku = fc.getSelectedFile().getName();
            JOptionPane.showMessageDialog(this, "Wybrano plik: " + nazwaPliku,
                "Informacja!", JOptionPane.INFORMATION_MESSAGE);
        }
    }
    else if("close".equals(cmd)){
        dispose();
    }
}
public static void main(String args[]) {
    // kod metody main
}
}

```

Aplikacja zawiera jedno menu z dwiema pozycjami: *Wybierz plik* oraz *Zamknij*. Obu pozycjom przypisano odpowiednie komendy (`choosefile` i `close`) odpowiadające akcjom otwarcia okna dialogowego wyboru plików (`choosefile`) i zamknięcia aplikacji (`close`). Sprawdzenie, z którą akcją mamy do czynienia, odbywa się standardowo, w metodzie `actionPerformed`.

Rysunek 9.8.
*Aplikacja
 wyświetlająca
 okno wyboru
 pliku*



Otwarcie okna dialogowego za pomocą wywołania metody `showOpenDialog` połączone jest w instrukcji warunkowej `if` ze sprawdzeniem wartości zwróconej przez tę metodę:

```
if(fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION){
```

Gdy jest to `JFileChooser.APPROVE_OPTION`, za pomocą opisanego wyżej mechanizmu pobierana jest nazwa pliku wybranego przez użytkownika. Nazwa ta jest następnie wyświetlana w oknie dialogowym stworzonym w sposób analogiczny do przedstawionego w ćwiczeniu 9.9.

10

Grafika i komponenty

Rysowanie elementów graficznych

Rysowanie prostych elementów graficznych w aplikacjach odbywa się podobnie jak w przypadku apletów. Używamy tych samych metod klasy `Graphics`, musimy również pamiętać o odświeżaniu ekranu w metodzie `paint`. Przekonajmy się o tym, zamieniając aplet rysujący koła z ćwiczenia 8.9 na aplikację.

ĆWICZENIE

10.1 Aplikacja rysująca figury

Zmodyfikuj kod z ćwiczenia 8.9 z rozdziału 8. w taki sposób, aby był on samodzielną aplikacją, niewymagającą do uruchomienia przeglądarki.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class FiguryApp extends JFrame implements MouseListener {
    private CircleDatabase database;
    private Random r;
    public FiguryApp() {
        super("Figury");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        addMouseListener (this);
        database = new CircleDatabase();
        r = new Random();
        setSize(320, 200);
        setVisible(true);
    }
    public void paint (Graphics gDC) {
        gDC.clearRect(0, 0, getSize().width, getSize().height);
        database.drawAll(gDC);
    }
    public void mousePressed (MouseEvent evt) {
        int x = evt.getX();
        int y = evt.getY();
        Color color = new Color (r.nextInt());
        database.addElement (new Circle(x, y, color));
        repaint();
    }
    public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new FiguryApp();
            }
        });
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
}
```

Jak widać, zmiany nie są bardzo duże. Należało oczywiście zmodyfikować konstruktor, a konkretnie przenieść do niego instrukcje z metody `init`, której teraz po prostu nie ma (była w `aplecie`). Niezbędne było też

dodanie metody `main`, od której zaczyna się wykonywanie kodu. Ma ona taką samą postać jak w przykładach z rozdziału 9. Zauważmy, że okno takiej aplikacji można dowolnie skalować, a ekran jest cały czas prawidłowo odświeżany.

W wersjach Javy powyżej 5 (1.5) w trakcie kompilacji (ze względu na użycie klasy `Vector` bez wykorzystania składni typów uogólnionych) pojawi się informacja `FiguryApp.java uses unchecked or unsafe operations` (podobnie było w przypadku apletu). Nie ma to jednak żadnego wpływu na kod wygenerowany przez kompilator. W listingach dostępnych na FTP została zawarta wersja klasy pozwalająca na uniknięcie wygenerowania powyższego ostrzeżenia (plik *CircleDatabase.java*).

Obsługa komponentów

Pakiet `Swing` oferuje zestaw gotowych klas, które umożliwiają korzystanie z typowych komponentów graficznych, takich jak przyciski, listy czy okna tekstowe. Bardzo ułatwia to tworzenie aplikacji graficznych. W dalszej części rozdziału zostaną omówione następujące komponenty: przyciski typu `JButton`, pola tekstowe typu `TextField`, rozszerzone pola tekstowe typu `TextArea`, etykiety typu `JLabel`, pola wyboru typu `JCheckbox` oraz listy typu `JList`.

Położenie i rozmiar komponentów graficznych można ustalać za pomocą metody `setBounds`. Przyjmuje ona cztery argumenty. Dwa pierwsze określają współrzędne `x` i `y`, dwa kolejne — szerokość i wysokość:

```
setBounds(x, y, szerokość, wysokość);
```

Do ustalania wyłącznie rozmiarów służy metoda `setSize` przyjmująca w postaci argumentów określenie wysokości i szerokości:

```
setSize(szerokość, wysokość);
```

Metoda `setLayout` pozwala na określenie sposobu rozkładu elementów w oknie. Można skorzystać z rozkładów automatycznych. Wtedy aplikacja będzie decydowała o konkretnym położeniu każdej z kontrolek — zostanie to pokazane w ćwiczeniu 10.8. Jeśli jednak chcemy mieć możliwość samodzielnego określania położenia komponentów (tak będzie w większości ćwiczeń), należy wywołać tę metodę, podając jako argument wartość `null`:

```
setLayout(null);
```

Przyciski JButton

Klasa JButton umożliwia utworzenie w oknie standardowego przycisku z dowolnym napisem. Aby z niej skorzystać, należy utworzyć nowy obiekt tej klasy, ustalić rozmiary i (ewentualnie) położenie kontrolki oraz napisać procedurę obsługi zdarzeń.

Ć W I C Z E N I E

10.2 Wykorzystanie przycisków

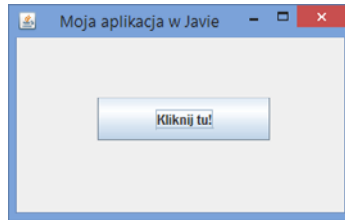
Umieść w oknie aplikacji przycisk z dowolnym napisem (rysunek 10.1). Po kliknięciu przycisku program powinien zakończyć działanie.

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    JButton button;
    public Aplikacja () {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);
        button = new JButton("Kliknij tu!");
        this.add(button);
        button.setActionCommand("cmdOK");
        button.addActionListener(this);
        button.setBounds(75, 55, 160, 40);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        if ("cmdOK".equals(e.getActionCommand())){
            dispose();
        }
    }
    public static void main(String args[]) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new Aplikacja();
            }
        });
    }
}
```


Rysunek 10.1.

*Przycisk
umieszczony
w oknie aplikacji*



W konstruktorze klasy `JButton` podajemy (jako argument) ciąg znaków, który będzie widoczny na przycisku. Jeśli nie podamy go w konstruktorze (może być bezargumentowy), tekst może zostać nadany później przy użyciu metody `setText` . Za pomocą metody `setActionCommand` przypisujemy przyciskowi ciąg znaków (komendę), który będzie zwracany po wywołaniu metody `getActionCommand` — jest to takie samo postępowanie, z którym mieliśmy do czynienia w przypadku menu. Dzięki temu komenda zwracana przez `getActionCommand` może być różna od ciągu znaków znajdującego się na przycisku (porównaj opis menu z rozdziału 9.). W metodzie `actionPerformed` zrezygnowano z zapisywania komendy do zmiennej pomocniczej — rozpoznanie komendy odbywa się w całości w wyrażeniu warunkowym instrukcji warunkowej `if` .

Pola tekstowe `JTextField`

Klasa `JTextField` służy do utworzenia jednowierszowego pola tekstowego umożliwiającego wprowadzenie przez użytkownika ciągu znaków. Dane z tego pola mogą być następnie odczytane i wykorzystane w aplikacji.

Ć W I C Z E N I E

10.3 Obsługa pól tekstowych

Umieść w oknie aplikacji pole tekstowe i przycisk. Po kliknięciu przycisku wyświetl wprowadzony tekst w oknie dialogowym (rysunek 10.2).

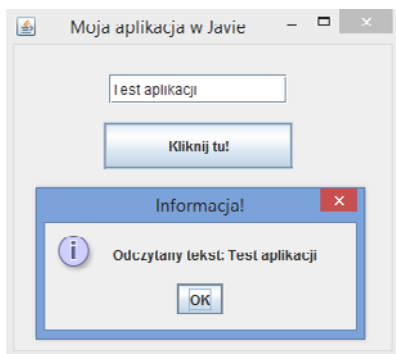
```
import javax.swing.*;
import java.awt.event.*;
```

```
public class Aplikacja extends JFrame implements ActionListener {
    JButton button;
```

```
TextField textField;  
public Aplikacja() {  
    super();  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    setSize(320, 200);  
    setTitle("Moja aplikacja w Javie");  
    setLayout(null);  
  
    button = new JButton("Kliknij tu!");  
    this.add(button);  
    button.addActionListener(this);  
    button.setActionCommand("cmdGetText");  
    button.setBounds(75, 65, 160, 40);  
  
    textField = new JTextField();  
    textField.setBounds(80, 25, 150, 25);  
    this.add(textField);  
  
    setVisible(true);  
}  
public void actionPerformed(ActionEvent e) {  
    if ("cmdGetText".equals(e.getActionCommand())){  
        String tekst = textField.getText();  
        JOptionPane.showMessageDialog(this, "Odczytany tekst: " + tekst,  
            "Informacja!", JOptionPane.INFORMATION_MESSAGE);  
    }  
}  
public static void main(String args[]) {  
    // treść metody main  
}
```

Rysunek 10.2.

Wyświetlenie napisu
wprowadzonego
w polu tekstowym



Pole tekstowe tworzymy, wywołując bezargumentowy konstruktor klasy `JTextField`. Rozmiary tego komponentu są ustawiane przy wykorzystaniu metody `setBounds` i jest on dodawany do obszaru okna

aplikacji dzięki metodzie `add (this.add(textField))`. Sposób obsługi kliknięcia przycisku jest taki sam jak w poprzednim ćwiczeniu (zmieniona została jedynie komenda przypisana przyciskowi z `cmdOK` na `cmdGetText`). Zawarty w polu tekst uzyskujemy dzięki wywołaniu metody `getText`:

```
String tekst = textField.getText();
```

Ć W I C Z E N I E

10.4 Wykorzystanie wartości odczytanych z pola tekstowego

Zmodyfikuj kod z ćwiczenia 10.3 w taki sposób, aby po wprowadzeniu ciągu znaków `Zamknij` następowało zamknięcie aplikacji.

```
public void actionPerformed(ActionEvent e) {
    if ("cmdGetText".equals(e.getActionCommand())){
        String tekst = textField.getText();
        if(tekst.equals("Zamknij")){
            dispose();
        }
        else{
            JOptionPane.showMessageDialog(this, "Odczytany tekst: " +
                tekst, "Informacja!", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

Wprowadzone zmiany ograniczyły się do modyfikacji metody `actionPerformed`. Należy po prostu porównać ciąg pobrany z pola tekstowego z ciągiem `Zamknij`. Jeśli występuje zgodność tekstów, okno aplikacji jest zamykane. Jeżeli natomiast zgodność nie występuje, pobrany tekst jest wyświetlany w oknie dialogowym.

Pola tekstowe `JTextArea`

Klasa `JTextArea` służy do utworzenia komponentu umożliwiającego wprowadzenie przez użytkownika dłuższego tekstu, tzw. rozszerzonego pola tekstowego. Posługujemy się nią jednak podobnie jak klasą `TextField`. Obie wykorzystamy w kolejnym ćwiczeniu.

Ć W I C Z E N I E

10.5 Wykorzystanie rozszerzonego pola tekstowego

Utwórz element typu `JTextArea` umożliwiający wpisanie dowolnego dłuższego tekstu oraz pole `TextField` służące do wpisania poszukiwanego ciągu znaków. Zdefiniuj przycisk *Szukaj*, którego kliknięcie spowoduje obliczenie liczby wystąpień szukanego ciągu znaków we wprowadzonym tekście (rysunek 10.3).

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    JButton button;
    JTextField textField;
    JTextArea textArea;
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

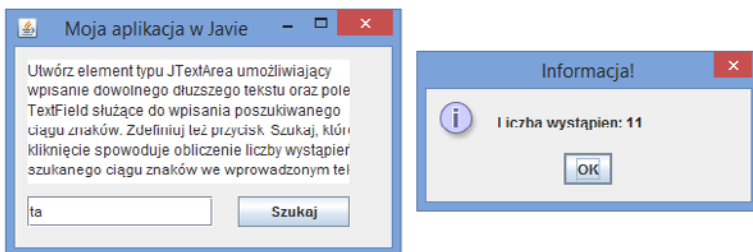
        button = new JButton("Szukaj");
        this.add(button);
        button.addActionListener(this);
        button.setActionCommand("cmdSzukaj");
        button.setBounds(180, 120, 90, 25);

        textField = new JTextField();
        textField.setBounds(10, 120, 150, 25);
        this.add(textField);

        textArea = new JTextArea();
        textArea.setBounds(10, 10, 260, 100);
        this.add(textArea);

        setVisible(true);
    }
    public void szukaj() {
        String tekst = textArea.getText();
        String ciag = textField.getText();
        int indeks = 0;
        int indeksWystapienia = 0;
        int liczbaWystapien = 0;
        if(tekst.equals("") || ciag.equals("")){
            indeksWystapienia = -1;
        }
        while (indeksWystapienia != -1){
```

```
indeksWystapienia = tekst.indexOf (ciag, indeks);
if (indeksWystapienia != -1){
    indeks = indeksWystapienia + 1;
    liczbaWystapien++;
}
}
OptionPane.showMessageDialog(this, "Liczba wystapien: " +
    liczbaWystapien, "Informacja!", JOptionPane.INFORMATION_MESSAGE);
}
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("cmdSzukaj")){
        szukaj();
    }
}
public static void main(String args[]) {
    // kod metody main
}
}
```



Rysunek 10.3. Aplikacja podająca liczbę wystąpień szukanego ciągu znaków we wprowadzonym tekście

Nowością w powyższym ćwiczeniu jest użycie komponentu `JTextArea` oraz wywoływanie metody `szukaj` w odpowiedzi na kliknięcie przycisku *Szukaj* (reprezentowanego przez zmienną referencyjną `button`). W metodzie `szukaj` następuje przeszukiwanie tekstu odczytanego z rozszerzonego pola tekstowego `textArea` pod kątem występowania w nim ciągu odczytanego z pola `textField`. Przeszukiwanie odbywa się w pętli `while`. Jest przy tym używana metoda `indexOf` pochodząca z klasy `String`, zwracająca indeks szukanego ciągu znaków podanego jako pierwszy argument. Drugi parametr wskazuje, od którego miejsca w ciągu ma się rozpocząć przeszukiwanie. A zatem wywołanie:

```
indeksWystapienia = tekst.indexOf (ciag, indeks);
```

oznacza: w zmiennej `indeksWystapienia` zapisz indeks, w którym tekst zapisany w `ciag` występuje w tekście zapisanym w `tekst`; przeszukiwanie rozpocznij od znaku wskazywanego przez `indeks`.

Jeżeli poszukiwanego ciągu nie ma w tekście, metoda `indexOf` zwraca wartość `-1`. Aktualna liczba wystąpień poszukiwanego ciągu jest zapisywana w zmiennej `liczbaWystapien`. Po zakończeniu pętli wartość tej zmiennej jest wyświetlana w oknie dialogowym (tworzonym na takiej samej zasadzie jak w ćwiczeniach 10.3 i 10.4).

Warto zauważyć, że zaprezentowany sposób przeszukiwania tekstu jest co prawda poprawny, ale nie jest najwydajniejszy. Można go za to łatwo przyspieszyć. Wystarczy jeśli po każdym odnalezieniu poszukiwanego ciągu przesuniemy się w ciągu przeszukiwanym nie o 1 (indeks = indeksWystapienia + 1), ale o długość ciągu poszukiwanego. Tę długość można uzyskać dzięki metodzie `length` z klasy `String` (pisząc np. `ciag.length()`). Niech pozostanie to jednak ćwiczeniem do samodzielnego wykonania.

Etykiety JLabel

Klasa `JLabel` służy do tworzenia etykiet, czyli obiektów wyświetlających tekst, który może być zmieniany przez aplikację, natomiast użytkownik nie ma możliwości jego bezpośredniej edycji. Aby utworzyć etykietę, należy wywołać konstruktor klasy oraz dodać nowy obiekt do okna aplikacji za pomocą metody `add`, analogicznie do wcześniej omawianych komponentów. Zmianę wyświetlanego przez etykietę ciągu znaków osiągniemy, wywołując metodę `setText` i podając jako parametr nowy tekst.

Ć W I C Z E N I E

10.6 Etykieta tekstowa w oknie aplikacji

Dodaj do okna aplikacji etykietę `JLabel` z dowolnym tekstem.

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame {
    JLabel label;
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
    }
}
```

```
setTitle("Moja aplikacja w Javie");
setLayout(null);

label = new JLabel("To jest etykieta.");
label.setBounds(115, 60, 90, 20);
this.add(label);
setVisible(true);
}
public static void main(String args[]) {
    // kod metody main
}
}
```

Ć W I C Z E N I E

10.7 Etykieta i inne komponenty

Umieść w oknie aplikacji pole tekstowe, etykietę tekstową i przycisk (rysunek 10.4). Po kliknięciu przycisku przypisz etykiecie ciąg znaków wprowadzony przez użytkownika w polu tekstowym.

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    JButton button;
    JTextField textField;
    JLabel label;
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        button = new JButton("Enter");
        this.add(button);
        button.addActionListener(this);
        button.setActionCommand("cmdEnter");
        button.setBounds(195, 90, 90, 25);

        textField = new JTextField();
        textField.setBounds(25, 90, 150, 25);
        this.add(textField);

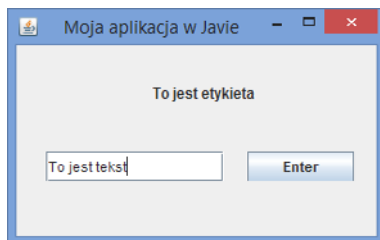
        label = new JLabel("To jest etykieta");
        label.setBounds(115, 30, 90, 20);
        this.add(label);

        setVisible(true);
    }
}
```

```
}  
public void actionPerformed(ActionEvent e) {  
    if ("cmdEnter".equals(e.getActionCommand())) {  
        label.setText(textField.getText());  
    }  
}  
public static void main(String args[]) {  
    // kod metody main  
}  
}
```

Rysunek 10.4.

*Wygląd aplikacji
z ćwiczenia 10.7*



Obsługa pola tekstowego oraz przycisku jest tu taka sama jak we wcześniej prezentowanych ćwiczeniach. W metodzie `actionPerformed` — po stwierdzeniu, że został kliknięty przycisk *Enter* (z przypisaną komendą `cmdEnter`) — tekst z pola jest odczytywany za pomocą metody `getText` (`textField.getText()`) i przypisywany etykietce `label` za pomocą metody `setText` (`label.setText()`). Odbywa się to bez użycia zmiennej pomocniczej do przechowywania tekstu — odwołania są bezpośrednie (`label.setText(textField.getText());`).

Pola wyboru JCheckBox

Klasa `JCheckBox` pozwala na umieszczenie w oknie aplikacji pól wyboru. Pojedyncze pole tworzymy, wywołując konstruktor, a następnie dodając powstały obiekt do okna aplikacji za pomocą metody `add`. W konstruktorze można podać ciąg znaków, który będzie wyświetlany obok pola, można również wykorzystać konstruktor bezargumentowy.

Ć W I C Z E N I E

10.8 Pola wyboru i rozkład automatyczny

Dodaj do okna aplikacji elementy typu `Checkbox`, rozmieszczone tak jak na rysunku 10.5.

```
import javax.swing.*;
import java.awt.*;

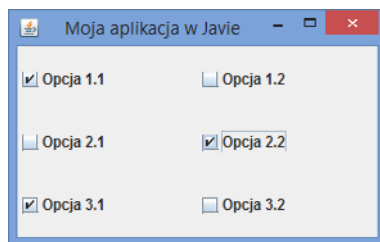
public class Aplikacja extends JFrame {
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(new GridLayout(3, 2));

        add(new JCheckBox("Opcja 1.1"));
        add(new JCheckBox("Opcja 1.2"));
        add(new JCheckBox("Opcja 2.1"));
        add(new JCheckBox("Opcja 2.2"));
        add(new JCheckBox("Opcja 3.1"));
        add(new JCheckBox("Opcja 3.2"));

        setVisible(true);
    }
    public static void main(String args[]) {
        // kod metody main
    }
}
```

Rysunek 10.5.

*Elementy typu
Checkbox
w rozkładzie
GridLayout*



Jak łatwo zauważyć, tym razem nie ustawialiśmy współrzędnych każdego z elementów osobno. W ogóle nie zajmowaliśmy się tą sprawą. Za równomierne rozłożenie odpowiada rozkład tabelaryczny (`GridLayout`), który został ustawiony za pomocą instrukcji:

```
setLayout(new GridLayout(3, 2));
```

W dotychczasowych ćwiczeniach używaliśmy rozkładu pustego (`setLayout(null)`), tak aby mieć pełną kontrolę nad rozmieszczeniem poszczególnych elementów na ekranie. Tym razem skorzystaliśmy z rozkładu automatycznego. Utworzony został obiekt `GridLayout`, który (dzięki podanym w konstruktorze parametrom) podzielił okno aplikacji na sześciokomórkową tabelę składającą się z trzech wierszy i dwóch kolumn. Dodawane metodą `add` elementy typu `JCheckbox` „wpisywały się” następnie w kolejne komórki tej tabeli, dzięki czemu bez bezpośredniego określania położenia elementów uzyskaliśmy ich równomierny rozkład w oknie aplikacji.

Oprócz zastosowanego w ćwiczeniu 10.8 rozkładu `GridLayout` można również zastosować wiele innych, jak np.: `BoxLayout`, `CardLayout`, `FlowLayout`, `GridBagLayout`. Ich opisy można znaleźć w dokumentacji JDK. Są to klasy dziedziczące bezpośrednio po `Object` i implementujące interfejs `LayoutManager`.

Listy rozwijane JComboBox

Klasa `JComboBox` umożliwia utworzenie komponentu pozwalającego na przechowywanie listy elementów tekstowych. Obiekt może zostać utworzony za pomocą bezargumentowego konstruktora (wtedy powstanie pusta lista) lub też można przekazać konstruktorowi tablicę ciągów znaków (a dokładniej tablicę obiektów) — wówczas elementy tablicy staną się elementami listy (dostępne są także inne konstruktory, których opis można znaleźć w dokumentacji JDK). Listę rozwijaną dodajemy do aplikacji tak samo jak każdy inny komponent. Poszczególne elementy umieszczamy na liście, wywołując jej metodę `addItem`.

Indeks aktualnie zaznaczonego (wybranego) elementu można uzyskać, wywołując metodę `getSelectedIndex`, natomiast odwołanie do tego elementu — za pomocą metody `getSelectedItem`. W tym drugim przypadku zwracana jest referencja do typu `Object`. Dostęp do elementu o wybranym indeksie uzyskuje się dzięki metodzie `getItemAt` (podając indeks jako argument wywołania). Jeśli aplikacja ma zareagować na zmianę aktywnego elementu (wybór elementu), można użyć interfejsu `ActionListener`.

Począwszy od Java 5.0 (1.5) przy deklaracji zmiennej typu JComboBox oraz tworzeniu obiektu należałoby skorzystać ze składni typów uogólnionych i podawać typ obiektów, które będą umieszczane na liście. Zostało to uwzględnione w komentarzach w ćwiczeniu 10.9 (dzięki temu kompilator może pilnować zgodności typów, nie ma to jednak wpływu na kod wynikowy; więcej informacji o typach uogólnionych można znaleźć we wspomnianej już książce *Java. Praktyczny kurs*).

Ć W I C Z E N I E

10.9 Aplikacja z komponentem typu JComboBox

Dodaj do okna aplikacji element typu JComboBox z kilkoma przykładowymi pozycjami (rysunek 10.6). Wybranie pozycji z listy powinno powodować wyświetlenie okna dialogowego z jej nazwą.

```
import javax.swing.*;
import java.awt.event.*;

public class Aplikacja extends JFrame implements ActionListener {
    JComboBox cmb = new JComboBox();
    // w Java 1.5 i wyższych lepiej tak:
    // JComboBox<String> cmb = new JComboBox<String>();
    // w Java 1.7 i wyższych można też tak:
    // JComboBox<String> cmb = new JComboBox<>();
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

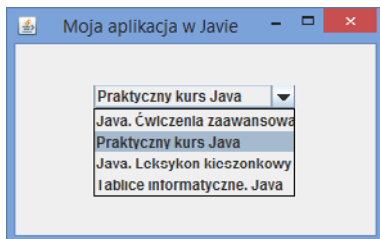
        cmb.setBounds(80, 40, 170, 20);
        cmb.addItem("Java. Ćwiczenia zaawansowane");
        cmb.addItem("Praktyczny kurs Java");
        cmb.addItem("Java. Leksykon kieszonkowy");
        cmb.addItem("Tablice informatyczne. Java");
        cmb.addActionListener(this);

        add(cmb);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        String tekst = cmb.getSelectedItem().toString();
        // lub tak:
        // String tekst = cmb.getItemAt(cmb.getSelectedIndex()).toString();
        // w przypadku skorzystania z typów uogólnionych również tak:
        // String tekst = cmb.getItemAt(cmb.getSelectedIndex());
        JOptionPane.showMessageDialog(this, "Wybrany element: " +
```

```
        tekst, "Informacja!", JOptionPane.INFORMATION_MESSAGE);  
    }  
    public static void main(String args[]) {  
        // kod metody main  
    }  
}
```

Rysunek 10.6.

*Okno aplikacji
z listą rozwijaną*



Lista rozwijana została utworzona za pomocą bezargumentowego konstruktora `JComboBox`, a następnie za pomocą metody `addItem` dodano do niej cztery pozycje — tytuły przykładowych książek. Instrukcja:

```
cmb.addActionListener(this);
```

powoduje, że zdarzenia związane z listą są przekazywane do obiektu aplikacji. Dzięki temu wybór pozycji z listy spowoduje wywołanie metody `actionPerformed` z klasy `Aplikacja`. W tej metodzie wybrany element listy (referencja do elementu) jest pobierany za pomocą wywołania `getSelectedItem()`, konwertowany do ciągu znaków przez wywołanie `toString()` oraz zapisywany w zmiennej pomocniczej `tekst`:

```
String tekst = cmb.getSelectedItem().toString();
```

Wartość tej zmiennej jest następnie wyświetlana w oknie dialogowym. W komentarzu została podana instrukcja alternatywna wykonująca takie samo zadanie za pomocą metod `getSelectedIndex` i `getItemAt`.

„Prawdziwa” aplikacja

Skoro wiemy, jak tworzyć okno aplikacji, jak dodać do niego menu i pola tekstowe, jak wyświetlić okno służące do wyboru plików, a także jak wykonywać operacje wejścia-wyjścia, możemy wykorzystać tę wiedzę w praktyczny sposób. Umieścimy więc w oknie aplikacji

element `JTextArea`, który pozwoli na wprowadzenie tekstu, i pozwólmy na odczyt oraz zapis tego tekstu z i do pliku. Byłaby to już „prawdziwa” aplikacja. Taki bardzo prosty edytor tekstu.

Ć W I C Z E N I E**10.10 Zapis i odczyt tekstu**

Napisz graficzną aplikację zawierającą rozszerzone pole tekstowe. Dodaj menu oraz procedury umożliwiające zapis i odczyt wprowadzonego przez użytkownika tekstu.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class Aplikacja extends JFrame implements ActionListener {
    private JTextArea textArea;
    public Aplikacja() {
        super();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(320, 200);
        setTitle("Edytor tekstu");
        setLayout(new GridLayout(0, 1));

        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("Plik");

        JMenuItem miOtworz = new JMenuItem("Otwórz");
        miOtworz.addActionListener(this);
        miOtworz.setActionCommand("Open");

        JMenuItem miZapisz = new JMenuItem("Zapisz jako");
        miZapisz.addActionListener(this);
        miZapisz.setActionCommand("Save");

        JMenuItem miZamknij = new JMenuItem("Zamknij");
        miZamknij.addActionListener(this);
        miZamknij.setActionCommand("Close");

        menu.add(miOtworz);
        menu.add(miZapisz);
        menu.add(new JSeparator());
        menu.add(miZamknij);

        textArea = new JTextArea();
        add(textArea);
```

```
        menuBar.add(menu);
        setJMenuBar(menuBar);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        String command = e.getActionCommand();
        if ("Close".equals(command)){
            dispose();
        }
        else if ("Open".equals(command)){
            JFileChooser fc = new JFileChooser();
            if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
                open(fc.getSelectedFile());
            }
        }
        else if ("Save".equals(command)){
            JFileChooser fc = new JFileChooser();
            if (fc.showSaveDialog(this) == JFileChooser.APPROVE_OPTION) {
                save(fc.getSelectedFile());
            }
        }
    }

    public void open(File file) {
        FileInputStream fin = null;
        try{
            fin = new FileInputStream(file);
        }
        catch(FileNotFoundException e){
            JOptionPane.showMessageDialog(this, "Błąd przy otwieraniu pliku.",
                "Błąd", JOptionPane.ERROR_MESSAGE);
            return;
        }
        BufferedReader inbr = new BufferedReader(new InputStreamReader(fin));
        textArea.setText("");
        String line = "";
        try{
            while ((line = inbr.readLine()) != null){
                textArea.append(line + '\n');
            }
        }
        catch(IOException e){
            JOptionPane.showMessageDialog(this, "Błąd wejścia-wyjścia.",
                "Błąd", JOptionPane.ERROR_MESSAGE);
        }
    }

    public void save(File file) {
        FileOutputStream fout = null;
        try{
            fout = new FileOutputStream(file);
        }
```

```

catch(FileNotFoundException e){
    JOptionPane.showMessageDialog(this, "Błąd przy zapisie pliku.",
        "Błąd", JOptionPane.ERROR_MESSAGE);
    return;
}
DataOutputStream out = new DataOutputStream(fout);
try{
    String line = textArea.getText();
    out.writeBytes(line + '\n');
}
catch(IOException e){
    JOptionPane.showMessageDialog(this, "Błąd wejścia-wyjścia.",
        "Błąd", JOptionPane.ERROR_MESSAGE);
}
}
public static void main(String args[]) {
    // kod metody main
}
}

```

Aplikacja łączy w sobie wiele elementów, z których korzystaliśmy już wcześniej, takich jak: menu, obsługa zdarzeń i operacje na plikach. Elementy interfejsu są tworzone w standardowy sposób. W oknie aplikacji pojawi się więc rozszerzone pole tekstowe typu `JTextArea` oraz menu. Użyty został rozkład tabelaryczny z zerową liczbą wierszy i jedną kolumną:

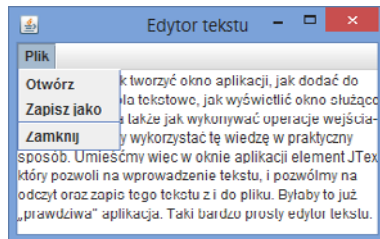
```
setLayout(new GridLayout(0, 1));
```

a zatem pole tekstowe zajmie całą dostępną powierzchnię okna aplikacji poza menu. Jedyną nowością w menu jest wprowadzenie pozycji będącej separatorem:

```
menu.add(new JSeparator());
```

To pozwala oddzielić pozycje *Otwórz* i *Zapisz jako* od *Zamknij* (rysunek 10.7).

Rysunek 10.7.
Wygląd edytora
tekstu z funkcją
zapisu i odczytu



Wybranie dowolnej pozycji menu powoduje wywołanie metody `actionPerformed`. Tam następuje sprawdzenie komendy przypisanej pozycji, która spowodowała to zdarzenie (patrz ćwiczenie 9.5). Jeśli jest to `Close` (wybrano pozycję *Zamknij*), okno aplikacji jest zamykane za pomocą wywołania metody `dispose`, co jest równoznaczne z zakończeniem pracy programu. Jeżeli jest to `Open` (wybrano pozycję *Otwórz*) lub `Save` (wybrano pozycję *Zapisz jako*), za pomocą klasy `JFileChooser` jest wyświetlane okno dialogowe pozwalające na wybór pliku (patrz ćwiczenie 9.10).

Tym razem nie odczytujemy jednak nazwy pliku wybranego przez użytkownika, ale pobieramy referencję do obiektu reprezentującego plik. Jest to obiekt typu `File`, a referencja do niego powstaje jako rezultat wywołania `fc.getSelectedFile()`. Referencja jest przekazywana metodzie `open` lub `save`, w zależności od tego, czy ma być dokonany odczyt, czy zapis.

Obie metody korzystają z operacji strumieniowych na plikach tekstowych (patrz ćwiczenia 6.15 i 6.16). Różnica w stosunku do przykładów z rozdziału 6. jest taka, że przy konstruowaniu strumieni nie jest używana nazwa pliku, ale korzysta się z opisującego go obiektu typu `File` przekazanego w postaci argumentu. Ponieważ odczyt odbywa się linia po linii, kolejne wiersze są dodawane do pola tekstowego `textArea` za pomocą metody `append`:

```
textArea.append(line + '\n');
```

