

Rozpocznij przygodę
z programowaniem w Javie



Ćwiczenia praktyczne

Wydanie II

Java

- Poznaj podstawowe elementy języka Java
 - ▼ Opanuj zasady programowania obiektowego
 - ▼ Napisz własne aplikacje
 - ▼ Wykorzystaj komponenty do tworzenia interfejsów użytkownika



Helion



Marcin Lis

Spis treści

Programowanie w Javie	5
Rozdział 1. Krótkie wprowadzenie	9
Instalacja JDK	9
Pierwszy program	12
B-kod, komplikacja i maszyna wirtualna	13
Java a C++	14
Obiektowy język programowania	15
Struktura programu	16
Rozdział 2. Zmienne, operatory i instrukcje	17
Zmienne	17
Operatory	26
Instrukcje	37
Rozdział 3. Obiekty i klasy	51
Metody	53
Konstruktory	59
Specyfikatory dostępu	62
Dziedziczenie	66
Rozdział 4. Wyjątki	71
Błędy w programach	71
Instrukcja try...catch	75
Zgłaszanie wyjątków	77
Hierarchia wyjątków	79

Rozdział 5. Rysowanie	81
Aplikacja a applet	81
Pierwszy applet	82
Jak to działa?	84
Cykl życia appletu	86
Czcionki	86
Rysowanie grafiki	89
Kolory	95
Wyświetlanie obrazów	98
Rozdział 6. Dźwięki	103
Rozdział 7. Animacje	107
Pływający napis	107
Pływający napis z buforowaniem	112
Zegar cyfrowy	114
Animacja poklatkowa	116
Zegar analogowy	118
Rozdział 8. Interakcja z użytkownikiem	123
Obsługa myszy	123
Rysowanie figur (I)	126
Rysowanie figur (II)	130
Rysowanie figur (III)	131
Rozdział 9. Okna i menu	137
Tworzenie okna aplikacji	137
Budowanie menu	139
Wielopoziomowe menu	146
Rozdział 10. Grafika i komponenty	151
Rysowanie elementów graficznych	151
Obsługa komponentów	152
Rozdział 11. Operacje wejścia-wyjścia	169
Wczytywanie danych z klawiatury	169
Operacje na plikach	176

Programowanie w Javie

Wstęp

 Chyba każdy, kto interesuje się informatyką, słyszał o Javie. Ten stosunkowo młody język programowania, w porównaniu do C++ czy Pascala, wyjątkowo szybko zdobył sobie bardzo dużą popularność i akceptację ze strony programistów na całym świecie. Jeszcze do niedawna wiele osób kojarzyło Javę tylko z appletami zawartymi na stronach WWW. To jednak tylko część zastosowań. Tak naprawdę to doskonaly, obiektowy język programowania, mający różnorodne zastosowania — od krótkich appletów do poważnych aplikacji. Początki były jednak zupełnie inne.

Być może trudno w to obecnie uwierzyć, ale język ten, pierwotnie znany jako *Oak* (z angielskiego „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 obecnie mniej istotne niż kiedyś. W dobie powszechnej komputeryzacji i podłączania rozmaitych urządzeń do sieci, w tym także wspomnianych lodówek i pralek, to 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 będzie więc uruchomić, przynajmniej teoretycznie, na komputerze PC i Macintosh, pod Windowsem i pod Unixem.

Historia *Oak* rozpoczęła się w 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 opracowywany 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 Internetu 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 przypadą 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 nie jest przedstawienie wszystkich aspektów programowania w Javie, ale jedynie pewien wycinek tego zagadnienia. W pierwszej części przedstawione są zasady programowania, ilustrowane wieloma przykładami. W części drugiej zamieszczono przykłady tworzenia różnorodnych appletów, a w części trzeciej podstawy tworzenia aplikacji z graficznym interfejsem oraz operacje wejścia-wyjścia. Niestety, ze względu na ograniczoną ilość miejsca nie możemy zaprezentować wielu ciekawych i bardziej zaawansowanych zagadnień, dlatego też będzie to raczej wycieczka po programowaniu w Javie niż metodyczny kurs opisujący całość zagadnienia.

Jak jednak sam tytuł wskazuje, 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 szybko chciałby zapoznać się ze strukturą języka, jak i uzupełnieniem bardziej metodycznego kursu, jakim jest np. publikacja *Praktyczny kurs Java*¹.

Narzędzia

Aby rozpocząć programowanie w Javie, potrzebujemy odpowiednich narzędzi. Konkretnie — kompilatora oraz maszyny wirtualnej, która interpretuje skompilowane programy. Będziemy opierać się tu na pakiecie Java Development Kit firmy Sun Microsystems. Można go pobrać bezpłatnie z witryny internetowej <http://java.sun.com>. Najlepiej korzystać z możliwie nowej wersji JDK, tzn. 1.5 (5.0) lub wyższej, choć podstawowe przykłady będą działać nawet na wiekowej już wersji 1.1.

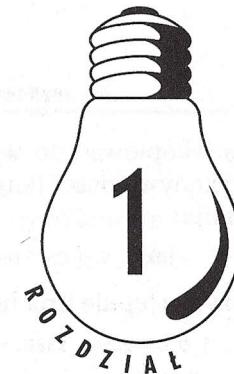
W ćwiczeniach z drugiej części książki, omawiającej tworzenie appletów, można skorzystać z dowolnej przeglądarki internetowej obsługującej język Java lub też dostępnej w JDK aplikacji *appletviewer*. Większość obecnie dostępnych na rynku przeglądarek udostępnia Javę poprzez mechanizm wtyczek, umożliwiając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

Oprócz JDK będzie jedynie potrzebny dowolny edytor tekstu pozwalający na wpisywanie tekstu programów i zapisywanie ich w plikach na dysku. Co prawda istnieje możliwość używania zintegrowanych środowisk programistycznych, jednak osobom początkującym polecałbym zestaw JDK i najprostszy edytor tekstu, tak aby poznąć dobrze sam język, a dopiero potem bardziej zaawansowane narzędzia programistyczne.

¹ Marcin Lis, *Praktyczny kurs Java*, Wydawnictwo Helion, Gliwice 2004.
Patrz: <http://helion.pl/ksiazki/pkjava.htm>.

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 (z 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, jak się wydaje ze względów czysto marketingowych, została przemianowana na 5.0. I ta wersja jest obecnie obowiązującą. Ta też wersja (*Java 2 Standard Edition 5.0*) była stosowana podczas przygotowywania materiałów do książki. Warto jednak zauważyć, że wewnętrzna numeracja narzędzi (widoczna np. w niektórych opcjach kompilatora javac) wciąż bazuje na wcześniejszej, logicznej numeracji (czyli Java 2.5 jest tożsama z Java 2.1.5).



Krótkie wprowadzenie

Instalacja JDK

 Instalacja pakietu JDK, który umożliwi nam tworzenie aplikacji, nikomu nie powinna przysporzyć problemów, niezależnie od tego, czy wybrana zostanie wersja dla systemu Linux, czy Windows¹. JDK instaluje się bowiem tak jak każdą inną aplikację. Proces ten jest także opisany na stronach <http://java.sun.com>. Podane niżej przykłady instalacji odnoszą się do wersji 1.5.0 dla systemów 32-bitowych.

Instalacja w systemie Linux

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

`jdk-1_5_0_wersja-linux-i586.bin`

gdzie *wersja* jest numerem wersji dystrybucji, np.:

`jdk-1_5_0_05-linux-i586.bin`

¹ JDK jest również dostępne dla systemu Solaris.

Należy go skopiować do wybranego katalogu na dysku oraz nadać prawo wykonywalności (ang. *executable*), co można wykonać, wydając polecenie:

```
chmod +x ./jdk-1_5_0_05-linux-i586.bin
```

Plik należy następnie uruchomić, wpisując komendę:

```
./jdk-1_5_0_05-linux-i586.bin
```

Na ekranie pojawi się treść licencji, którą należy zaakceptować, wpisując słowo *yes*, po czym w katalogu zostanie utworzony podkatalog o nazwie *jdk1.5.0_05*, do którego zostaną rozpakowane pliki pakietu.

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

```
jdk-1_5_0_wersja-linux-i586-rpm.bin
```

np.:

```
jdk-1_5_0_05-linux-i586-rpm.bin
```

Może on być instalowany przez użytkownika posiadającego prawa administratora systemu. Plikowi należy nadać prawa wykonywalności, podobnie jak w poprzednim przypadku:

```
chmod +x ./jdk-1_5_0_05-linux-i586-rpm.bin
```

a następnie uruchomić go za pomocą polecenia:

```
./jdk-1_5_0_05-linux-i586-rpm.bin
```

Również i w tym przypadku należy zaakceptować treść licencji oraz poczekać na zainstalowanie pakietu (pliki zostaną zapisane w katalogu */usr/java/jdk1.5.0_05*).

Po instalacji, aby usprawnić pracę z JDK, warto dodać do zmiennej środowiskowej PATH ścieżkę do podkatalogu *bin* tego pakietu (np. */usr/java/jdk1.5.0_05/bin/*) — nie będziemy wtedy musieli za każdym razem podawać pełnej ścieżki dostępu, aby uruchomić kompilator czy też inne narzędzie. Aby wykonać tę operację w przypadku powłoki systemowej *bash*, wykonujemy polecenie:

```
PATH=$PATH:/ścieżka_dostępu
```

np.:

```
PATH=$PATH:/usr/java/jdk1.5.0_05/bin/
```

Instalacja w systemie Windows

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

```
jdk-1_5_0_wersja-windows-i586.exe
```

np.:

```
jdk-1_5_0_05-windows-i586.exe
```

Uruchomienie tego pliku spowoduje rozpoczęcie typowego procesu instalacji aplikacji.

Po instalacji dobrze jest dopisać do zmiennej środowiskowej PATH ścieżkę dostępu do katalogu *bin* środowiska JDK. Aby zmienić ten parametr dla bieżącej sesji konsoli systemowej (po uruchomieniu aplikacji *command* w systemach 98 i Me oraz *cmd* w systemach 2000 i XP), należy wydać polecenie:

```
path=%path%;ścieżka_dostępu
```

np.:

```
path=%path%;c:\Program Files\java\jdk1.5.0_05\bin
```

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, zobaczymy jak uruchomić go w systemach Linux i w Windows.

Jeśli pracujemy w Linuksie na konsoli tekstowej, nie trzeba, oczywiście, wykonywać żadnych dodatkowych czynności. Jeśli jednak korzystamy z interfejsu graficznego, należy uruchomić program terminala. Na przykład dla systemu Fedora Core 4 i menedżera okien KDE należy z menu *Aplikacje* wybrać kolejno *Narzędzia systemowe* i *Terminal*.

Jeśli pracujemy w systemie Windows 98 lub Me, należy uruchomić aplikację *command.exe* (*Start/Uruchom/command.exe*), natomiast w przypadku systemów 2000 i XP — aplikację *cmd.exe* (*Start/Uruchom/cmd.exe*).

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

Pierwszy program

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

ĆWICZENIE

1.1. Aplikacja wyświetlająca tekst

Napisz program wyświetlający na ekranie dowolny napis.

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

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

`cd sciezka_dostepu/nazwa_katalogu`

np.:

`cd /usr/tmp/java/`

w systemie Linux lub:

`cd dysk:\sciezka_dostepu\nazwa_katalogu`

np.:

`cd c:\java\projekty`

w systemie Windows.



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

- Nie będziemy w tej chwili analizować sposobu działania tej aplikacji, postaramy się natomiast zobaczyć wyniki jej działania na ekranie. Przystępujemy do kompilacji, pisząc (zakładam, że pakiet JDK jest już poprawnie zainstalowany w systemie,

a zamienna systemowa PATH wskazuje ścieżkę dostępu do katalogu *bin* pakietu JDK):

`javac Main.java`

- Jeżeli kompilacja się powiedzie, na dysku powstanie nowy plik o nazwie *Main.class*. Aby uruchomić program, piszemy:

`java Main`

Efekt działania widzimy na rysunku 1.1. Zwracam uwagę, że w przypadku kompilacji podawaliśmy nazwę pliku z rozszerzeniem (*Main.java*), natomiast przy uruchamianiu rozszerzenie pominęliśmy (*Main*).

Rysunek 1.1.

Efekt działania
pierwszego
programu w Javie

```
E:\WINNT\system32\cmd.exe
E:\!>javac Main.java
E:\!>java Main
Pierwszy program w Javie
E:\!>
```

B-kod, kompilacja i maszyna wirtualna

Każdy napisany program przed wykonaniem musi być przetłumaczony na język zrozumiały dla komputera, czyli poddany procesowi kompilacji. W przypadku języków takich 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, jako że 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*). Zawarty jest on w plikach z rozszerzeniem *class*. W naszym przypadku tekst programu 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 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 operacyjnego). Wynika z tego, że wystarczy, aby dla każdego systemu powstała dedykowana mu maszyna wirtualna Javy, a będzie można uruchomić na nim każdy program w Javie bez wykonywania dodatkowych modyfikacji (to oczywiście pewne uproszczenie, nie uruchomimy np. programu graficznego na konsoli tekstowej). Jest to, niestety, również jedna z przyczyn powolności Javy. Kod interpretowany jest bowiem wolniejszy od wykonywanego bezpośrednio na danym procesorze. Obecnie ma to, co prawda, coraz mniejsze znaczenie — powstają bowiem coraz doskonalsze maszyny wirtualne, zawierające np. kompilatory JIT (z ang. *Just In Time*), które w locie komplują część (lub nawet całość) b-kodu do kodu natywnego procesora — niemniej Java wciąż jest jednak wolniejsza niż np. C czy C++.

Java a C++

Osobom, które znają C++, Java zapewne od razu się spodoba. Choć by dlatego, że jej składnia jest do C++ bardzo podobna. Jest to jednak miejscami nieco zładne, gdyż ten sam zapis w C++ wcale nie musi oznaczać tego samego w Javie. Mówiąc naukowo, 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 przestaje to 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;
}
```

Można by tu pominąć nieużywane parametry funkcji *main*, pisząc:

```
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.

Obiektowy język programowania

Java jest w pełni obiektowym 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, czy 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 jest to natomiast opis obiektu. Mówiąc 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.

Dla osób, które nie zetknęły się z programowaniem obiektowym, jest to zapewne całkowicie zagmatwane i mało zrozumiałe. Nie należy się tym jednak przejmować. Do wszystkiego dojdziemy po kolei.

Struktura programu

Każdy program w Javie składa się ze zbioru klas. Przyjmiemy zasadę, że każda klasa musi być zapisana w oddzielnym pliku o takiej samej nazwie, jak nazwa tej klasy oraz rozszerzeniu `.java`. Tę zasadę będziemy konsekwentnie stosować w pierwszej części książki. Wykonanie programu rozpoczyna się od metody `main`, która musi być zadeklarowana jako publiczna i statyczna. Struktura programu w przypadku naszego pierwszego przykładu była następująca:

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

Jak widać, zadeklarowaliśmy klasę o nazwie `Main`. W ciele tej klasy, tzn. pomiędzy nawiasami klamrowymi: `{ } 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). Zatem wykonanie rozpocznie się od tej metody. W ciele funkcji main znajduje się z kolei instrukcja powodująca wypisanie na ekranie tekstu Pierwszy program w Javie. Na razie nasze programy ilustrujące cechy języka będą miały pisać właśnie w taki sposób, by zawierały instrukcje w ciele funkcji main, przyjmując na słowo, że tak właśnie ma być.`



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, który informuje 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 (z ang. *integral types*),
- ❑ typy zmiennopozycyjne (rzeczywiste, z 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 C++ dokładnie określono sposób reprezentacji tych danych. Niezależnie więc od tego, na jakim systemie pracujemy (16-, 32- czy 64-bitowym), dokładnie wiadomo, na ilu bitach zapisała jest zmienna danego typu. Wiadomo też dokładnie, z jakiego zakresu wartości może ona przyjmować, nie ma więc dowolności, która w przypadku języka C mogła prowadzić do sporych trudności przy przenoszeniu programów pomiędzy różnymi platformami. W tabeli 2.1 zaprezentowano zakresy poszczególnych typów danych oraz liczbę bitów niezbędną do zapisania zmiennych danego typu.

Tabela 2.1. Zakresy 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 prezentuje tabela 2.2.

Tabela 2.2. Zakresy dla typów zmiennoprzecinkowych w Javie

Typ	Liczba bitów	Liczba bajtów	Zakres
float	32	4	od -3,4e38 do 3,4e38
double	64	8	od -1,8e308 do 1,8e308

Format danych float i double jest zgodny ze specyfikacją standardu ANSI/IEEE 754. Zapis 3,4e48 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, ogólnie wszelkich znaków alfanumerycznych), przy czym w Javie jest on 16-bitowy i zawiera znaki Unicode. Ponieważ znaki reprezentowane są tak naprawdę jako 16-bitowe kody liczbowe, typ ten zalicza się czasem do typów arytmetycznych.

Deklarowanie zmiennych typów podstawowych

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

```
typ_zmiennej nazwa_zmiennej;
```

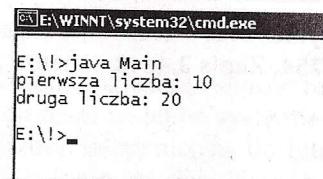
Po takiej deklaracji zmienna jest już gotowa do użycia, tzn. możemy jej przypisywać różne wartości bądź też wykonywać na niej różne operacje, np. dodawanie. Przypisanie wartości zmiennej odbywa się przy użyciu znaku (operatora) =.

ĆWICZENIE**2.1. Deklarowanie zmiennych**

Zadeklaruj dwie zmienne całkowite i przypisz im dowolne wartości. Wyniki wyświetl na ekranie (rysunek 2.1).

Rysunek 2.1.
Wynik działania programu z ćwiczenia 2.1

```
public
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);
    }
}
```



Instrukcja `System.out.println` pozwala wyprowadzić ciąg znaków na ekran. Wartość zmiennej można również przypisać już w trakcie deklaracji, pisząc:

```
typ_zmiennej nazwa_zmiennej = wartość;
```

Można również zadeklarować wiele zmiennych danego typu, oddzielając 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 wewnątrz funkcji czy metody.

ĆWICZENIE**2.2. Jednoczesna deklaracja i inicjacja zmiennych**

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

```
public
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);
    }
}
```

ĆWICZENIE**2.3. Deklarowanie zmiennych w jednym wierszu**

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

```
public
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);
    }
}
```

Przy nazywaniu zmiennych obowiązują pewne zasady. Otóż nazwa może się składać z wielkich i małych liter oraz cyfr, ale nie może się zaczynać od cyfry. Choć nie jest to zabronione, raczej unika się stosowania polskich znaków diakrytycznych. 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`. Mimo że tak długa nazwa może wydawać się dziwna, jednak bardzo poprawia czytelność programu oraz ułatwia jego analizę. Naprawdę warto ten sposób stosować. Przyjmuje się też, co również jest bardziej wygodne, że nazwę zmiennej rozpoczynamy małą literą, a poszczególne człony

tej nazwy (wyrazy, które się na nią składają) rozpoczynamy wielką literą — dokładnie tak jak w powyższych przykładach.

Typy odnośnikowe

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

- typy klasowe (ang. *class types*)¹,
- typy tablicowe (ang. *array types*).

Zaczniemy 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 poprzez 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 numer wiersza i kolumny itd. Jeśli chcemy zatem przechować w programie 10 liczb całkowitych, najwygodniej będzie użyć w tym celu 10-elementowej tablicy typu `int`.

Typy klasowe pozwalają na tworzenie klas i deklarowanie zmiennych obiektowych. Zajmiemy się nimi w rozdziale 3.

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 domyślnie

¹ Typy klasowe moglibyśmy podzielić z kolei na obiektowe i interfejsowe; są to jednak rozważania, którymi nie będziemy się w niniejszej publikacji zajmować.

przypisana jest 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;
```

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.

WICZENIE

2.4. Deklarowanie tablicy

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

```
public
class Main
{
    public static void main (String args[])
    {
        int tablica[];
        tablica[0] = 11;
        System.out.println ("Zeroowy 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 (jest to ważne, gdyż program może składać się z bardzo wielu klas, a każda z nich jest zazwyczaj definiowana w oddzielnym pliku), numer wiersza w tym pliku oraz konkretne

miejsce wystąpienia błędu. Na samym końcu kompilator podaje też całkowitą liczbę błędów.

```
E:\!>javac Main.java
Main.java:7: variable tablica might not have been initialized
    tablica[0] = 11;
          ^
1 error
E:\!>
```

Rysunek 2.2. Błąd komplikacji. Nie zainicjalizowaliśmy zmiennej tablica

Skoro jednak wystąpił błąd, należy go natychmiast naprawić.

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.

```
public
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 tablica[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]
```

Warto przy tym zauważyć, że elementy tablicy numerowane są 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?

ĆWICZENIE

2.6. Odwołanie do nieistniejącego indeksu

Zadeklaruj i zainicjalizuj tablicę dziesięcioelementową. Spróbuj przydzielić elementowi o indeksie 10 dowolną liczbę całkowitą.

```
public
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 została 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 więc 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.

```
E:\!>javac Main.java
E:\!>java Main
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Main.main(Main.java:7)
E:\!>
```

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. Operatory te możemy podzielić na następujące grupy²:

- 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.

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

² Można wydzielić również inne grupy, co wykracza jednak poza ramy tematyczne niniejszej publikacji.

```
System.out.println("b = " + b);
System.out.println("b - a = " + c);
c = a * b;
System.out.println("a * b = " + c);
}
```

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 2 razy, pozostawiając resztę 2 ($5 * 2 = 10$, $10 + 2 = 12$).

ĆWICZENIE

2.8. Dzielenie modulo

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

```
public
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);
    }
}
```

Kolejne operatory typu arytmetycznego to operator inkrementacji i dekrementacji. Operator inkrementacji (czyli zwiększenia), którego symbolem jest ++, powoduje przyrost wartości zmiennej o jeden. Może występować w formie przyrostkowej bądź przedrostkowej. 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.

ĆWICZENIE

2.9. Operator inkrementacji

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

```
public 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. Wynikiem działania tego programu będzie ciąg liczb: 2, 2, 3, 3, 6. Dlaczego? Na początku zmienne *x* przyjmuje wartość 1. W 2. wierszu 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 4. po prostu wyświetlamy wartość *x* (*x* = 3). W wierszu 5. najpierw zmiennej *y* jest przypisywana dotychczasowa wartość *x* (*x* = 3, *y* = 3), a następnie wartość *x* jest zwiększana o jeden (*x* = 4). W wierszu 6. wyświetlamy wartość *y* (*y* = 3). W wierszu 7. najpierw zwiększamy wartość *x* o jeden (*x* = 5), a następnie przypisujemy tę wartość zmiennej *y*. 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.

```
public
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);
    }
}
```

Działania operatorów arytmetycznych na liczbach całkowitych nie trzeba chyba wyjaśniać, z dwoma może wyjątkami. Otóż co się stanie, jeżeli wynik dzielenia dwóch liczb całkowitych nie będzie liczbą całkowitą? Odpowiedź na szczerze jest prosta, wynik zostanie zaokrąglony w dół. 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).

ĆWICZENIE

2.11. Dzielenie liczb całkowitych

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

```
public
class Main
{
    public static void main(String args[])
    {
        int a, b, c;
        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);
    }
}
```

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. Szybko przekonamy się, że kompilator do tego nie dopuści (rysunek 2.4).

Rysunek 2.4.

Próba przekroczenia dopuszczalnej wartości zmiennej

```
E:\>javac Main.java
Main.java:7: possible loss of precision
 found   : int
 required: byte
        zmienna = 128;
                  ^
1 error
E:\>
```

ĆWICZENIE

2.12. Przekroczenie zakresu w trakcie komplikacji

Zadeklaruj zmienną typu byte. Przypisz jej wartość 128. Spróbuj dokonać komplikacji otrzymanego kodu.

```
public
class Main
{
    public static void main (String args[])
    {
        byte zmienna;
        zmienna = 128;
        System.out.println(zmienna);
    }
}
```

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

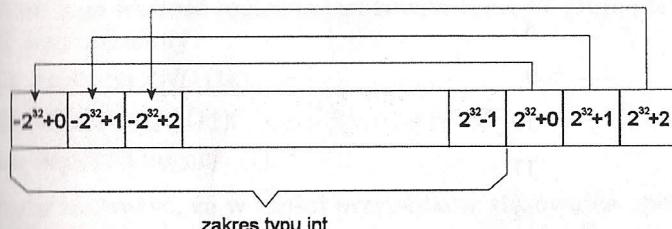
ĆWICZENIE

2.13. Przekroczenie zakresu w trakcie działania kodu

Zadeklaruj zmienne typu long. Wykonaj operacje arytmetyczne przekraczające dopuszczalną wartość takiej zmiennej. Wynik wyświetl na ekranie.

```
public
class Main
{
    public static void main (String args[])
    {
        long a, b = (long) Math.pow(2, 63) + 1;
        a = b + b;
        System.out.println ("a = " + a);
    }
}
```

Operacja `(long) Math.pow(2, 63)` oznacza podniesienie liczby 2 do potęgi 63., a następnie skonwertowanie wyniku (który jest liczbą typu double) do typu long. Zmiennej a jest przypisywany wynik działania `b + b` i okazuje się, że jest to 0. Dlaczego? Otóż jeżeli jakaś wartość przekracza dopuszczalny zakres swojego typu, jest „zawijana” do początku tego zakresu. Obrazowo ilustruje to rysunek 2.5.



Rysunek 2.5. Przekroczenie dopuszczalnego zakresu dla typu int

Operatory bitowe

Operacje te, jak sama nazwa wskazuje, dokonywane są 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 dwójkowym będą zatem wykorzystywane jedynie dwie cyfry — 0 i 1. Kolejne liczby budowane są z tych dwóch cyfr, dokładnie tak samo jak w systemie dziesiętnym; przedstawia to tabela 2.3. Widać wyraźnie, że np. 4 dziesiętnie to 100 dwójkowo, a 10 dziesiętnie to 1010 dwójkowo.

Tabela 2.3. Reprezentacja liczb w systemie dwójkowym i dziesiętnym

System dwójkowy	System dziesiętny
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Na tak zdefiniowanych liczbach możemy dokonywać znanych ze szkoły operacji bitowych *AND* (iloczyn bitowy), *OR* (suma bitowa) oraz *XOR* (bitowa alternatywa wykluczająca). Symbolem operatora *AND* jest znak & (ampersand), operatora *OR* znak | (pionowa kreska), natomiast operatora *XOR* znak ^ (strzałka w góre). Oprócz tego można również wykonywać operacje przesunięć bitów. Zestawienie występujących w Javie operatorów bitowych zostało przedstawione w tabeli 2.4.

Operatory logiczne

Argumentami operacji takiego typu 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),

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	>>>

jeżeli jego wartość logiczna jest równa true. W grupie tej wyróżniają się trzy operatory:

- logiczne *AND* (&&),
- logiczne *OR* (||),
- logiczna negacja (!).

Warto zauważyć, że w części przypadków stosowania operacji logicznych, aby otrzymać wynik, wystarczy obliczyć tylko pierwszy argument. Wynika to, oczywiście, z właściwości operatorów. Jeśli bowiem wynikiem obliczenia pierwszego argumentu jest 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 i wartością całego wyrażenia będzie false.

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.

Operator warunkowy

Operator warunkowy ma następującą składnię:

warunek ? wartość1 : wartość2;

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

Tabela 2.6. Operatory porównania w Javie

Operator	Opis
==	jeśli argumenty są sobie równe, wynikiem jest true
!=	jeśli argumenty są różne, wynikiem jest true
<	jeśli argument prawostronny jest mniejszy od lewostronnego, wynikiem jest true
>	jeśli argument prawostronny jest większy od lewostronnego, wynikiem jest true
<=	jeśli argument prawostronny jest mniejszy lub równy lewostronnemu, wynikiem jest true
>=	jeśli argument prawostronny jest większy lub równy lewostronnemu, wynikiem jest true

WZIĘCIE

2.14.

Wykorzystanie operatora warunkowego

Wykorzystaj operator warunkowy do zmodyfikowania wartości do wolnej zmiennej typu całkowitego (int).

```
public
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: jeżeli x jest równe 1, przypisz zmiennej y wartość 10, w przeciwnym przypadku przypisz zmiennej y wartość 20. Ponieważ zmienią x zainicjalizowaliśmy wartością 1, na ekranie zostanie wyświetlony ciąg znaków y = 10.

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. Przedstawia to tabela 2.7³. 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	Symbol
inkrementacja przyrostkowa	<code>++, --</code>
inkrementacja przedrostkowa, negacje	<code>++, --, ~, !</code>
mnożenie, dzielenie	<code>*, /, %</code>
przesunięcia bitowe	<code><<, >>, >>></code>
porównania	<code><, >, <=, >=</code>
porównania	<code>==, !=</code>
bitowe AND	<code>&</code>
bitowe XOR	<code>^</code>
bitowe OR	<code> </code>
logiczne AND	<code>&&</code>
logiczne OR	<code> </code>
warunkowe	<code>?</code>
przypisania	<code>=, +=, -=, *=, /=, %=, >>=, <<=,</code> <code>>>>=, &=, ^=, =</code>

³ Tabela nie przedstawia wszystkich operatorów występujących w Javie, a jedynie omawiane w książce.

Instrukcje

Instrukcja warunkowa if...else

Hardzo często w programie zachodzi potrzeba sprawdzenia 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
}
```

Wyrażenie warunkowe, inaczej niż w C i C++, musi dać w wyniku wartość typu boolean, tzn. true lub false.

WŁAŚCIWOŚĆ

Użycie instrukcji warunkowej if...else

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

```
public
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");
        }
    }
}
```

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

Jeżeli delta jest równa zero, istnieje tylko jedno rozwiązanie — mianowicie $x = -B / (2 * A)$. 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, to 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, ale podamy je bezpośrednio w kodzie.

ĆWICZENIE

2.16. Pierwiastki równania kwadratowego

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

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

        System.out.println ("Parametry równania:\n");
        System.out.println ("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;
            if (delta < 0){
                System.out.println ("Delta < 0.");
                System.out.println ("To równanie nie ma 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);
                }
            }
        }
    }
}
```

Jak łatwo zauważycy, instrukcję warunkową można zagnieździć, tzn. że 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. Możemy więc posłużyć się konstrukcją `if...else if`. Zamiast dwóch mniej wygodną konstrukcję, taką jak przedstawiona poniżej:

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

dalej możemy zapisać dużo prościej i czytelniej w postaci:

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

ĆWICZENIE

2.17. Zastosowanie instrukcji `if...else if`

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

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

```

{
    int parametrA = 1, parametrB = -1, parametrC = -6;
    System.out.println ("Parametry równania:\n");
    System.out.println ("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 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 ciąg warunków i wykonywać różny kod w zależności od tego, czy są one prawdziwe, czy fałszywe. Można nią zastąpić ciąg instrukcji if...else if. Jeżeli mamy w kodzie przykładową konstrukcję w postaci:

```

if (a == 1){
    instrukcje1;
}
else if (a == 50){
    instrukcje2;
}
else if (a == 23){
    instrukcje3;
}
else{
    instrukcje4;
}

```

Możemy zastąpić ją następująco:

```

switch (a){
    case 1:
        instrukcje1;
        break;
    case 50:
        instrukcje2;
        break;
    case 23:
        instrukcje3;
        break;
    default:
        instrukcje4;
}

```

Po kolej i jest tu sprawdzane, czy a jest równe 1, potem 50 i w końcu 23. Jeżeli równość zostanie w jednym z przypadków stwierdzona, wykonywane są instrukcje po odpowiedniej klauzuli case. Jeżeli a nie jest równe żadnej z wymienionych liczb, wykonywane są instrukcje po słowie default. Instrukcja break powoduje wyjście z bloku switch.

WIELENIE

2.10. Użycie instrukcji wyboru switch

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

```

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

```

Uwaga! Jeżeli zapomnimy o słowie break, wykonywanie instrukcji switch będzie kontynuowane, co może prowadzić do otrzymania niepoziewanych efektów. W szczególności zostanie wtedy wykonany

blok instrukcji występujący po default. Może to być, oczywiście, efektem zamierzonym, 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.

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

Widać wyraźnie (rysunek 2.5), że teraz według naszego programu zmienna a jest jednocześnie równa 10, jak i różna od dziesięciu.

Rysunek 2.5.
Ilustracja błędu
z ćwiczenia 2.19

The screenshot shows a Windows command prompt window. The user has typed 'javac Main.java' and 'java Main'. The output shows the variable 'a' is assigned the value 10, and a message stating 'a nie jest równe ani 1, ani 10' (a is not equal to 1 or 10). This indicates a logical error where the variable 'a' retains its previous value (10) after the switch statement.

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, to możemy zrobić to, pisząc

10 razy System.out.println("Java");. Jeżeli jednak chcielibyśmy mniej już 150 takich napisów, to, pomijając oczywiście sensowność tej czynności, byłby to już problem. Na szczęście z pomocą przychodzi nam właśnie pętle. 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 wykonania pętli. wyrażenie warunkowe określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli, wyrażenie modyfikujące jest zwykle używane do modyfikacji zmiennej będącej licznikiem.

ĆWICZENIE

2.20. Budowa pętli for

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

```
public
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 pętli jest ona zwiększana o jeden (i++) oraz wykonywana jest instrukcja System.out.println ("Java");. Wszystko trwa tak długo, aż i osiągnie wartość 10 (i <= 10).

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

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

ĆWICZENIE

2.21. Wyrażenie modyfikujące w bloku instrukcji

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

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

Zwróciły 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.

ĆWICZENIE

2.22. Łączenie wyrażenia warunkowego i modyfikującego

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

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

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

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

ĆWICZENIE

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.

```
public
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!

ĆWICZENIE

2.24. Pętla bez wyrażeń

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

```
public
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 oba średniki w nawiasach okrągłych występujących po `for` są niezbędne do prawidłowego funkcjonowania kodu. Warto też zwrócić uwagę na zmianę kierunku nierówności. We wcześniejszych przykładach sprawdzaliśmy bowiem, czy `i` jest mniejsze bądź równe 10, a teraz, czy jest większe bądź równe. Dzieje się tak, dlatego że poprzednio sprawdzaliśmy,

czy pętla ma być dalej wykonywana, natomiast obecnie, czy ma zostać zakończona. Przy okazji wykorzystaliśmy też kolejną instrukcję, mianowicie `break`. Służy ona do natychmiastowego przerwania wykonywania pętli.

Kolejna przydatna instrukcja, `continue`, powoduje rozpoczęcie kolejnej iteracji, 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`.

```
public 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. dostarcza on resztę z dzielenia. Nic jednak nie stoi na przeszkodzie, aby działającą w taki sam sposób aplikację 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`.

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

Pętla while

Jeśli pętla typu `for` służy raczej do wykonywania znanej z góry liczby operacji, to w przypadku pętli `while` liczba ta nie jest zwykle znana. 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 tak długo, dopóki wyrażenie warunkowe prawdziwe. Oznacza to, że gdzieś w pętli musi wystąpić modyfikacja warunku bądź też instrukcja `break`. Inaczej będzie się ona wykonywać w nieskończoność!

ĆWICZENIE

2.27. Budowa pętli while

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

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

ĆWICZENIE

2.28. Połączone wyrażenia

Zmodyfikuj kod z ćwiczenia 2.27 tak, aby wyrażenie warunkowe zmieniało jednocześnie wartość zmiennej `i`.

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

```

while (i++ <= 10){
    System.out.println ("Java");
}
}

```

ĆWICZENIE

2.29. Liczby nieparzyste i pętla while

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

```

public
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);

```

ĆWICZENIE

2.30. Budowa pętli do...while

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

```

public
class Main
{
    public static void main (String args[])
    {
}

```

```

int i = 1;
do{
    System.out.println ("Java");
}
while (i++ <= 9);

```

Wydawać by się mogło, że to przecież to samo, co zwykła pętla while. Jest jednak pewna różnica. Otóż w przypadku pętli do...while instrukcje wykonane są co najmniej jeden raz, nawet jeśli warunek jest na pewno fałszywy.

Pętla do...while z fałszywym warunkiem

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

```

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

```

Pętla foreach

Począwszy od wersji 1.5 (5.0), Java udostępnia nowy rodzaj pętli. Jest ona nazywana pętlą foreach lub rozszerzoną pętlą for (z ang. *enhanced for*) i pozwala na automatyczną iterację po kolekcji obiektów lub też po tablicy. Jej działanie pokażemy właśnie w tym drugim przypadku. Jeśli bowiem mamy tablicę tab zawierającą wartości pewnego typu, to do przejrzenia wszystkich jej elementów możemy użyć konstrukcji w postaci:

```

for(typ val: tablica){
    /instrukcje
}

```

W takim przypadku w kolejnych przebiegach pętli `for` pod `val` będzie podstawiana wartość kolejnej komórki.

ĆWICZENIE

2.32. 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.

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



Obiekty i klasy



Każdy program w Javie składa się z klas. W naszych dotychczasowych przykładach była to tylko jedna klasa o nazwie `Main`. W klasach zawarty jest kod wykonywalny, który realizuje przypisane mu zadania, są one także opisami 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.

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 jest to pewnego rodzaju plan, na podstawie którego w programie tworzone są obiekty, które z kolei mogą przechowywać dane i wykonywać różne zadania.

By nie poprzestawać na suchych definicjach, które mogą wydawać się niejasne, stwórzmy jakiś przykład. Założymy, że chcemy w programie zapisać dane dotyczące punktów na ekranie. W tym celu

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

powinniśmy stworzyć klasę `Punkt`. Każdy punkt powinien charakteryzować się dwoma współrzędnymi: `x` i `y`; klasa ta powinna więc zawierać dwa pola typu `int`.

ĆWICZENIE

3.1. Tworzenie prostej klasy

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

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

Składowymi klas są pola i metody. Pola to zmienne, w których możemy przechowywać dane dotyczące danej klasy. Metody to kod, który może wykonywać różne operacje. W naszym przypadku klasa zawiera tylko dwa pola — `x` i `y`, które opisują położenie punktu na ekranie.

Zadeklarujmy teraz zmienną typu `Punkt`. Jest to bardzo proste:

```
Punkt nowyPunkt;
```

Jak widać, podajemy najpierw nazwę klasy, potem nazwę zmiennej, podobnie jak dla zmiennych poznanych już wcześniej typów podstawowych. Wiemy też już, że w ten sposób zadeklarowaliśmy jedynie referencję (odniesienie) do obiektu typu `Punkt`. Aby móc skorzystać z obiektu takiego typu, musimy go najpierw utworzyć. Używamy do tego operatora `new` w postaci:

```
new NazwaKlasy();
```

W naszym przypadku będzie to wyglądało następująco:

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

Można też najpierw zadeklarować referencję, a dopiero potem utworzyć i przypisać jej obiekt danej klasy:

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

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

```
Punkt nowyPunkt;
```

powodowałoby 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.

Metody

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

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

W nawiasach okrągłych po nazwie metody podajemy jej parametry. W ten sposób możemy przekazać jej jakieś dane. W przypadku naszej klasy `Punkt` z ćwiczenia 3.1 moglibyśmy stworzyć dwie metody, które zwracałyby współrzędną `x` i współrzędną `y` punktu.

Pierwsze metody klasy `Punkt`

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

```
public
class Punkt
{
    int x, y;
    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
}
```

W tej chwili, aby uzyskać informację o wartości współrzędnej `x`, możemy napisać:

```
punkt.x;
```

lub:

```
punkt.getX();
```

zakładając oczywiście, że został wcześniej utworzony obiekt o nazwie punkt. Za zwrócenie wartości pól x i y odpowiada, oczywiście instrukcja return.

ĆWICZENIE

3.3. Ustawianie i pobieranie współrzędnych

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

```
public
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;
    }
}
```

Klasa ta zawiera dwa pola typu int o nazwach x i y, które będą służyły do przechowywania współrzędnych danego punktu. Mamy również dwie metody. Jedna zajmuje się ustawieniem pól danego obiektu, a druga pobiera te wartości. Metoda ustawWspolrzedne jest typu void², co oznacza, że nie zwraca ona żadnej wartości. Ma ona natomiast dwa parametry, wspX i wspY, oba typu int. W ciele tej metody do pola x przypisywana jest wartość parametru wspX, a do pola y — wartość parametru wspY. Zatem po wykonaniu instrukcji:

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

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

Dziuga metoda — pobierzPunkt — jest typu Punkt. Oznacza to, że zwraca ona referencję do typu Punkt. Można zatem napisać taką instrukcję:

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

W ciele tej metody najpierw tworzony jest nowy obiekt typu Punkt, a następnie odpowiednim polom tego obiektu przypisywane są pola x i y obiektu bieżącego. Obiekt punkt (dokładniej referencja do tego obiektu) jest zwracany instrukcją return. Instrukcja return powoduje przerwanie wykonywania danej metody i, ewentualnie, zwrócenie inną wartości wymienionej po return.

Przykaz czas wykorzystać tak stworzony kod w konkretnym przykładzie. W tym celu utworzymy dwa pliki. Jeden, tak jak do tej pory, o nazwie Main.java oraz drugi o nazwie Punkt.java (zgodnie z tym, co zostało napisane wcześniej, każda klasa powinna znajdować się w oddzielnym pliku o nazwie zgodnej z nazwą klasy). Zawartością pliku Punkt.java będzie podany wyżej kod klasy Punkt.

Wykorzystanie obiektu klasy Punkt

Napisz prosty program wykorzystujący obiekty klasy Punkt.

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

Kompilujemy teraz oba pliki, przy czym muszą się one znajdować w tym samym katalogu. Mamy dwie możliwości. Możemy najpierw

² Zostało tu zastosowane często spotykane uproszczenie, w którym typ metody jest utożsamiany z typem zwracanym przez metodę. W rzeczywistości pojęcia te nie są tożsame.

skompilować plik *Punkt.java* i następnie *Main.java* albo też skompilować tylko *Main.java*. W tym drugim przypadku, ponieważ kod klasy *Main* korzysta z klasy *Punkt*, komplikacja pliku *Punkt.java* odbiega się automatycznie.

Jak zatem działa nasz najnowszy 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();
```

Zauważmy jednak, że oznacza to również, iż tak naprawdę zmienna *pomocniczyPunkt* wcale nie jest nam potrzebna. Równie dobrze moglibyśmy dokonać odwołania bezpośredniego.

ĆWICZENIE

3.5. Bezpośrednie odwołanie do pól obiektu

Zmodyfikuj kod z ćwiczenia 3.4, tak aby nie było konieczności użycia zmiennej *pomocniczyPunkt*.

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

Czemu jednak dokonujemy tej, na pozór karkołomnej, konstrukcji, tworząc najpierw w metodzie *pobierzWspolrzedne* nowy obiekt typu *Punkt*, przypisując mu odpowiednie wartości *x* i *y* i zwracając dopiero ten nowy byt? Odpowiedź jest prosta. Nie możemy naraz zwrócić współrzędnych *x* i *y*. Metoda może bowiem zwracać tylko jedną wartość — typu podstawowego lub obiektowego (klasowego).

Moglibyśmy co najwyżej napisać dwie dodatkowe metody, które będą osobno zwracały wartość *x*, a osobno wartość *y*, tak jak zrobiliśmy to w ćwiczeniu 3.2. 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. Myślmy, że chcemy zmodyfikować tylko *x*, ustawiając jego wartość 5. Stosując dotychczasowy kod, należałoby użyć w tym celu instrukcji:

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

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

Nowe metody klasy *Punkt*

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

```
public 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;
    }
}
```

Oprócz przedstawionych w ostatnim ćwiczeniu, istnieje jeszcze jedna możliwość uzyskania naraz wartości *x* i *y*. Metodzie *pobierzPunkt* można bowiem przekazać parametr. Wyglądałoby to następująco:

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

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.

ĆWICZENIE

3.7. Przeciążanie metod

Dołącz do klasy Punkt przeciążoną metodę pobierzWspolrzedne.

```
public
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;
}
```

Wykorzystanie przeciążonych metod

Utwórz klasę Main korzystającą z przeciążonych metod klasy Punkt.

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

Uwaga! Uważamy przy tym, że mimo iż na początku nie ustawiliśmy żadnych współrzędnych punktu, mają one wartość zero. Otóż pola obiektów w Javie są standardowo zerowane (tzn. zawierają 0, \u0000, false lub null — w zależności od typu zmiennej).

Konstruktory

Przykazując 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 do tego celu dodatkową, zwykłą metodę i używać jej na przykład następująco:

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

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

```
void inicjujZmienne()
{
    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 za tem 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 bez rezultatu (nie zwracają żadnych wartości) oraz mają nazwę identyczną z nazwą klasy, której dotyczą. Mogą mieć natomiast różne parametry, może też istnieć kilka konstruktów dla danej klasy. Napiszmy więc dwa konstruktory dla klasy Punkt. Jeden będzie bezparametrowy i będzie ustawał wartości x i y na 320 i 200, zaś drugi ustawi współrzędne na wartości podane przez użytkownika w postaci parametrów.

ĆWICZENIE

3.9. Tworzenie konstruktorów

Napisz konstruktory dla klasy Punkt.

```
public
class Punkt
{
    int x, y;
    public Punkt()
    {
        ustawWspolrzedne(320, 200);
    }
    public Punkt(int x, int y)
    {
        ustawWspolrzedne(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;
```

Wykorzystanie konstruktorów

Wykorzystaj klasę Main do przetestowania konstruktorów klasy Punkt utworzonych w ćwiczeniu 3.9.

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

Specyfikatory dostępu

W naszych dotychczasowych programach przed słowem `class` pojawiało się zwykle słowo `public`. Oznacza ono, że dana klasa jest publiczna, czyli że mogą z niej korzystać wszystkie inne klasy. Gdyby deklaracja taka się nie pojawiła, oznaczałoby to, że dana klasa jest klasą pakietową, tzn. mają do niej dostęp tylko klasy z danego pakietu. Pakiet to grupa klas, zajmująca się zwykle jednym zagadnieniem np. pakiet `java.awt.net` zawiera klasy zajmujące się obsługą sieci. Klasy w Javie są zatem albo publiczne, albo pakietowe.

Specyfikatory dostępu (nazywane również modyfikatorami dostępu, ang. *access modifiers*) pojawiają się jednak nie tylko przy nazwach klas, ale także przy nazwach pól i metod. Pola oraz metody mogą być:

- publiczne,
- prywatne,
- chronione,
- pakietowe.

Publiczne składowe klasy oznacza się słowem `public` i oznacza to, że wszyscy mają do nich dostęp oraz że są dziedziczone przez klasy pochodne. 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 (o dziedziczeniu klas za chwilę) 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.

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

3.11. Prywatne pola klasy Punkt

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

```
public
class Punkt
{
```

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

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

Próba odwołania do prywatnych pól

W klasie `Main` odwołaj się bezpośrednio do pól `x` i `y` klasy `Punkt` z ćwiczenia 3.11. Spróbuj skompilować otrzymany kod.

```
public
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);
    }
}
```

Jak łatwo zauważyc, do pól typu `private` nie możemy się odwoływać ani przy odczytzie, ani przy zapisie (próba komplikacji powoduje wyświetlenie trzech błędów). Oczywiście, kiedy zmienimy dostęp do prywatnego na publiczny, czyli kiedy definicja pól będzie miała postać:

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

Kod z ćwiczenia 3.12 zadziała bez problemów. W tym miejscu może pojawić się pytanie: dlaczego więc nie stosować tylko tego ostatniego sposobu? Dlaczego nie deklarować wszystkich pól i metod jako publiczne? Otóż po to, aby ukryć wewnętrzną organizację obiektu, a na "zewnętrz" udostępnić jedynie interfejs pozwalający na wykonywanie określonych przez programistę operacji. Przydaje się to zwykle w przypadku bardziej skomplikowanych klas, jednak nawet na prostej klasie `Punkt` możemy pokazać istotę tego zagadnienia.

Otoż 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. Zatem w klasie Punkt nie będzie już 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 byłoby dokonanie przeliczeń współrzędnych. Spowodowałoby to ogromne komplikacje i z pewnością doprowadziłoby do powstania wielu błędów. Jeżeli jednak pola klasy będą prywatne, trzeba będzie tylko przedefiniować metody klasy Punkt. Cała reszta programu nawet nie zauważycie, że coś się zmieniło!

ĆWICZENIE

3.13. Zmiana sposobu reprezentacji współrzędnych

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

```
public
class Punkt
{
    private double modul, sinalfa;
    public Punkt()
    {
        ustawWspolrzedne(320, 200);
    }
    public Punkt (int x, int y)
    {
        ustawWspolrzedne(x, y);
    }
    void ustawWspolrzedne(int wspX, int wspY)
    {
        modul = Math.sqrt (Math.pow(wspX, 2) + Math.pow(wspY, 2));
        sinalfa = (double) wspY / modul;
    }
    Punkt pobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.sinalfa = sinalfa;
        punkt.modul = modul;
        return punkt;
    }
    void pobierzWspolrzedne(Punkt punkt)
    {
        punkt.sinalfa = sinalfa;
        punkt.modul = modul;
    }
}
```

```
void ustawX(int wspX)
{
    ustawWspolrzedne(wspX, pobierzY());
}
void ustawY(int wspY)
{
    ustawWspolrzedne(pobierzX(), wspY);
}
int pobierzX ()
{
    double x;
    x = modul * Math.sqrt(1 - Math.pow(sinalfa, 2));
    return (int) Math.round(x);
}
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 użyliśmy jej w powyższym przykładzie. Zatem sinus kąta alfa³, reprezentowany przez pole o nazwie $sinalfa$, jest równy $y/modul$. Natomiast moduł to $\sqrt{x^2 + y^2}$.

Przeliczenie współrzędnych biegunowych na kartezjańskie jest nieco bardziej skomplikowane. Co prawda y to po prostu $modul * sin(alfa)$, za to x to $modul * \sqrt{1 - sin^2(alfa)}$. Metoda Math.pow(arg1, arg2) zwraca wartość arg1 podniesioną do potęgi arg2. Metoda Math.round(arg) zamknięcia argument do liczby całkowitej⁴.

Wpis (nazwaTypu) zmienna, np. (double) x, oznacza konwersję zmiennej do podanego typu; w naszym przykładzie konwersję zmiennej x

³ Kąt pomiędzy prostą przechodzącą przez reprezentowany punkt i środek układu współrzędnych a osią OX.

⁴ Przedstawiony kod będzie działał poprawnie tylko dla dodatnich współrzędnych x , ujemne wartości x będą traciły znak (czyli zostaną przekształcone na wartości dodatnie). Ćwiczenie zostało pozostawione w takiej postaci, aby nie wprowadzać dodatkowych instrukcji, które zwiększyłyby tylko sedno zagadnienia. Ponieważ jednak poprawki nie są bardzo skomplikowane, proponuję potraktować je jako ćwiczenie do samodzielnego wykonania.

typu int do typu double. Konwersje te muszą być dokonane, gdyż jak pamiętamy, inne są wyniki działań arytmetycznych (w szczególności dzielenia) na liczbach całkowitych, a inne na liczbach zmiennoprzecinkowych.

Jeśli teraz tak zmienioną klasę Punkt użyjemy wraz z klasą Main z ćwiczenia 3.10, szybko przekonamy się, że całość działa bez zarzutu, mimo że reprezentacja danych jest całkowicie odmienna niż we wcześniejszych przykładach.

Dziedziczenie

Znamy już dosyć dobrze klasę Punkt, założymy jednak, że potrzebujemy obiektów opisujących nie tylko współrzędne punktu, ale również jego kolor. Chcielibyśmy jednak 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;
}
```

Musielibyśmy teraz dopisać wszystkie zdefiniowane wcześniej metody klasy Punkt oraz zapewne dodatkowo ustawKolor i pobierzKolor. W ten sposób jednak dwukrotnie piszemy 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”).

ĆWICZENIE

3.14. Dziedziczenie z klasy Punkt

Napisz klasę KolorowyPunkt rozszerzającą klasę Punkt o możliwość przechowywania informacji o kolorze.

```
public
```

```
class KolorowyPunkt extends Punkt
{
    protected 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;
    }
}
```

Testowanie klasy KolorowyPunkt

Simplex 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());
    }
}
```

Wyjaśnienia zapewne wymagają konstruktory klasy KolorowyPunkt, a właściwa występująca w nich metoda super. Otóż powoduje ona prostu wywołanie odpowiedniego konstruktora klasy bazowej, czyli klasy Punkt.

Obawywiście z klasy KolorowyPunkt możemy wyprowadzić kolejną, np. DwukolorowyPunkt, dla punktów, które przyjmują dwa różne kolory, np. w zależności od znaku wartości współrzędnej x. Z klasy DwukolorowyPunkt może dziedziczyć kolejna klasa itd. Wszystkie natomiast klasy bezpośrednio lub pośrednio dziedziczą z klasy bazowej Object. Jeżeli zatem piszemy:

```
public
class Punkt
{
}
```

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

```
public
class Punkt extends Object
{
}
```

Docieśliwi 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.

ĆWICZENIE

3.16. Przesłanianie metod przy dziedziczeniu

Napisz przykładowe klasy o nazwach KlasaA i KlasaB. KlasaB ma dziedziczyć z klasy A, natomiast w obu klasach ma zostać zdefiniowana metoda o nazwie f. Wykonaj komplikację obu klas.

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

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

Kiedy dokonamy komplikacji obu klas⁵ okaże się, że kompilator nie generuje żadnego błędu, a zatem kod jest formalnie poprawny. Jak go jednak interpretować? Wiemy, że klasa dziedzicząca przejmuje metody z klasy bazowej. Skoro w obu klasach występuje metoda o nazwie f,

⁵ Pamiętajmy, że trzeba je zapisać w osobnych plikach o nazwach zgodnych z nazwami klas.

to wydawałoby się, że obiekty klasy KlasaB powinny zawierać dwie metody o takich samych nazwach i parametrach. I tak jest w istocie! 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, natomiast odwołanie do metody z klasy KlasaA będzie możliwe jedynie za pomocą specjalnej konstrukcji ze słowem super. Schematycznie takie wywołanie ma postać:

```
super.nazwa_metody(argumenty_metody);
```

Wywołanie przesłoniętej metody

Modyfikuj kod z ćwiczenia 3.17 w taki sposób, aby w klasie KlasaB w metodzie f została wywołana metoda f z klasy KlasaA.

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

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

3.17. Testowanie przesłoniętych metod

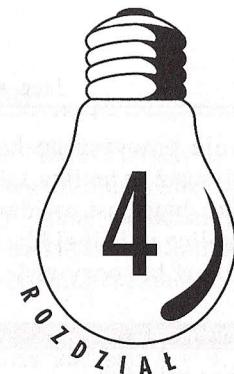
Napisz kod klasy Main, w którym zostaną utworzone obiekty klas KlasaA i KlasaB oraz wywołane metody f tych obiektów. Sprawdź działanie kodu z klasami z ćwiczeń 3.16 i 3.17.

```
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();
}
}

```



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 wykonywał 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 pięcioelementową, należałoby sprawdzić, czy nie następuje odwołanie do nieistniejącego elementu.

LEKCJA

4.1. Odwołanie do nieistniejącego elementu tablicy

W klasie Main zadeklaruj tablicę pięcioelementową. Spróbuj odczytać wartość nieistniejącego tej tablicy o indeksie 20.

```

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

```

Wykonanie powyższego kodu spowoduje oczywiście błąd (rysunek 4.1), ponieważ w tablicy table 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.

```
E:\!>javac Main.java
E:\!>java Main
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 20
at Main.main(Main.java:7)
E:\!>
```

Rysunek 4.1. Przekroczenie dopuszczalnego zakresu tablicy

ĆWICZENIE

4.2. Nieprawidłowa współpraca dwóch klas

Napisz takie dwie klasy, aby w jednej została zadeklarowana tablica 5-elementowa, a w drugiej następował 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];
    }
}
```

Pamiętajmy o konieczności zapisania klas Tablica i Main w oddzielnego plikach! W tej chwili w klasie Main wywołujemy metodę obiektu typu Tablica, nie wiedząc bezpośrednio, jakiej wielkości jest sama tablica. Bardzo łatwo przekroczyć więc maksymalny indeks i spowodować błąd. Tak też dzieje się w powyższym przykładzie.

Możemy tego uniknąć, sprawdzając, czy podawana jako argument metody getElement wartość nie przekracza zakresu 0 – 4. Takiego sprawdzenia można dokonać, stosując znaną już instrukcję if.

Sprawdzanie poprawności zakresu

Zmodyfikuj kod z ćwiczenia 4.2 tak, aby po przekroczeniu dopuszczalnego indeksu tablicy nie występował błąd w programie. Wprowadź do metody getElement instrukcję sprawdzającą, czy nie 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;
    }
}
```

Moglibyśmy, oczywiście, zakończyć wykonywanie programu już w funkcji `getElement`, zwykle jednak istnieje potrzeba poinformowania funkcji wywołującej o wystąpieniu błędu. W powyższym rozwiązaniu o błędzie informuje nas zwrócenie wartości `-1`. To rozwiązywanie ma jednak bardzo poważną wadę: żaden z elementów tablicy nie może być równy `-1`. Moglibyśmy poradzić sobie z tym problemem, dodając do klasy `Tablica` pole typu `boolean`, np. o nazwie `isError`.

ĆWICZENIE

4.4. Dodatkowe pole obsługi błędów

Zmodyfikuj kod z ćwiczenia 4.3 tak, aby o wystąpieniu błędu informowała dodatkowa zmienna umieszczona 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;
        }
    }
}
```

W ten sposób dodajemy jednak coraz więcej nowych zmiennych i funkcji. Warto by skorzystać z jakiejś innej metody obsługi błędów. Pomocą przychodzą nam tutaj tzw. wyjątki. Wyjątek jest to błąd programistyczny, który powstaje w chwili wystąpienia błędu. Mamy go jednak przechwycić i wykonać nasz własny kod obsługi błędu. Jeżeli tego nie zrobimy, zostanie on obsłużony przez system, a na konsoli systemowej zobaczymy wtedy odpowiedni komunikat. W ćwiczeniu 4.1 występował np. wyjątek o nazwie `ArrayIndexOutOfBoundsException` (widać to wyraźnie na rysunku 4.1). Jego wystąpienie oznacza, że został przekroczony dopuszczalny zakres indeksu tablicy.

Instrukcja try...catch

Po obsłudze wyjątków służy blok `try...catch`, którego schemat wykonywania wygląda następująco:

```
try{
    //blok instrukcji mogący spowodować wyjątek
}
catch(TypWyjątku1 identyfikatorWyjątku1){
    //obsługa wyjątku 1
}
catch(TypWyjątku2 identyfikatorWyjątku2){
    //obsługa wyjątku 2
}
catch(TypWyjątkuN identyfikatorWyjątkuN){
    //obsługa wyjątku n
}
finally{
    //instrukcje
}
```

Po `try` następuje blok instrukcji mogących spowodować wyjątek. Jeżeli podczas ich wykonywania zostanie on wygenerowany, wykonanie 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 w nim występujący 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.

ĆWICZENIE

4.5. Użycie bloku try...catch

Zastosuj instrukcję `try...catch` do przechwytcia wyjątku generowanego przez system w ćwiczeniu 4.2.

```
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;
    }
}
```

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` zyskaliśmy bardzo dużo. 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 programu¹. Teraz ta wiadomość nie jest nam już potrzebna.

¹ Tak naprawdę w Javie tablice są obiektami, które posiadają pole `size`, tak więc rozmiar jest zawsze znany. To jednak tylko ilustracja wykorzystania wyjątków, przyjmijmy zatem, że nie znamy rozmiarów tablicy.

Zgłaszanie wyjątków

Widzieliśmy już, że możemy zakończyć pracę w funkcji wywołującej metodę `getElement`, 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 wyjątek, możemy objąć ją klauzulą `try...catch`, przenosząc tym samym obsługę błędów poza klasę `Tablica`.

4.6. Przechwytywanie wyjątku

Modyfikuj kod z poprzedniego ćwiczenia, 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);
        } 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 generowanego wyjątku odbywa się w klasie Main przy wywoływaniu 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, która mogłaby go przechwycić, jest on przekazywany do funkcji, która tę metodę wywołała. W tym przypadku jest to funkcja (metoda) main z klasy Main.

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

```
throw new KlasaWyjątku();
```

ĆWICZENIE

4.7. Generowanie własnego wyjątku

Zmodyfikuj kod klasy Tablica 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, została wykorzystana zwykła instrukcja warunkowa if. Jeśli przekroczenie nastąpiło, tworzony jest nowy obiekt wyjątku klasy 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 wyjątek. Nie jest to jednak obligatoryjne, konstruktor może być bezargumentowy.

Hierarchia wyjątków

Wyjątki w Javie są obiektami. Oznacza to, że kiedy wyjątek ma być wygenerowany, tworzony jest obiekt danego typu. W naszym przykładzie ten typ to ArrayIndexOutOfBoundsException. Dostęp do tego obiektu mamy poprzez zadeklarowaną przez nas zmienną oznaczającą 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.

4.8. Kolejność przechwytywania wyjątków

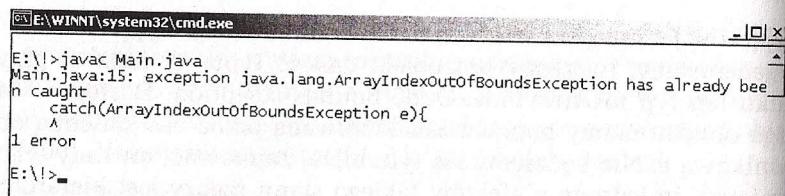
Zmodyfikuj klasę Main w taki sposób, aby najpierw przechwytywany był wyjątek ogólny Exception, a następnie wyjątek ArrayIndexOutOfBoundsException. Spróbuj dokonać komplikacji otrzymanego kodu.

```
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 4.2, komplikacja się nie udała. Ponieważ klasa wyjątku `ArrayIndexOutOfBoundsException` dziedziczy pośrednio z klasy `Exception`, instrukcja:

```
catch(ArrayIndexOutOfBoundsException e)
```

nie została nigdy osiągnięta, bowiem w przypadku wygenerowania wyjątku `ArrayIndexOutOfBoundsException` wcześniej została ona przechwycona przez instrukcję `catch (Exception e)`.



```
E:\>javac Main.java
Main.java:15: exception java.lang.ArrayIndexOutOfBoundsException has already been caught
    catch(ArrayIndexOutOfBoundsException e){
           ^
1 error
E:\>_
```

Rysunek 4.2. Błąd polegający na nieuwzględnieniu hierarchii wyjątków



Rysowanie

Aplikacja a aplet

 Programy w Javie mogą być apletymi (ang. *applet*) bądź 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).

W podziały na aplety i aplikacje czasem wynikają nieporozumienia. W pierwszych latach po wprowadzeniu Javy na rynek, w artykułach 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 dysk? Otóż oczywiście w języku programowania Java istnieje możliwość zapisu danych na dysk, jak i tworzenia wielu innych konstrukcji programistycznych. Pozwalają na to bezpośrednio programy pisane jako aplikacje, jak również po spełnieniu dodatkowych warunków — aplety.

Pierwszy aplet

W pierwszej części książki zaczynaliśmy od programu wypisującego napis na ekranie. Tym razem zrobimy tak samo. Stwórzmy aplet wyświetlający na ekranie napis Pierwszy aplet w Javie.

ĆWICZENIE

5.1. Aplet wyświetlający zdefiniowany tekst

Napisz aplet wyświetlający na ekranie napis Pierwszy aplet w Javie.

```
import java.applet.*;
import java.awt.*;

public class Hello extends Applet
{
    public void paint (Graphics gDC)
    {
        gDC.drawString ("Pierwszy aplet w Javie", 100, 100);
    }
}
```

Co należy z tym programem zrobić? Oczywiście najpierw skompilować. Następnie musimy napisać fragment kodu w języku HTML, który będzie zrozumiały dla przeglądarki i w którym będzie osadzony nasz aplet. W najprostszym przypadku kod ten będzie wyglądał w następujący sposób:

```
<applet
    code = "nazwa klasy"
    width = "szerokość apletu"
    height = "wysokość apletu">
</applet>
```

Najlepiej zapisać go w pliku o rozszerzeniu *html*, np. *index.html*.

ĆWICZENIE

5.2. Umieszczanie apletu na stronie WWW

Napisz kod HTML pozwalający na osadzenie kodu apletu Hello na stronie WWW.

```
<html>
<head>
<title>Pierwszy aplet</title>
</head>
<body>
```

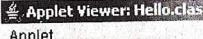
```
<applet
    code = "Hello.class"
    width = "320"
    height = "200">
</applet>
</body>
</html>
```

Teraz możemy już zobaczyć wyniki naszej pracy na ekranie. Można do tego celu użyć narzędzia o nazwie *appletviewer* z pakietu JDK, pisanego w wierszu poleceń:

```
appletviewer index.html
```

Na ekranie zobaczymy wtedy widok zaprezentowany na rysunku 5.1. Można też otworzyć plik *index.html* w dowolnej przeglądarce obsługującej Javę, w której została zainstalowana wtyczka (z ang. *plugin*) JRE.

Rysunek 5.1.
Aplet wyświetlający
napis na ekranie

 Applet Viewer: Hello.class
Applet

Pierwszy aplet w Javie

Applet started.

Podobny efekt możemy, oczywiście, osiągnąć, pisząc nie aplet, ale "prawdziwą" aplikację w Javie, chociaż wymaga to nieco większego nakładu pracy.

ĆWICZENIE

5.3. Graficzna aplikacja wyświetlająca napis

Napisz aplikację w Javie, która utworzy na ekranie okno i wyświetli w nim napis.

```
import java.awt.*;
```

```
public
class HelloApp extends Frame
{
    public HelloApp ()
    {
```

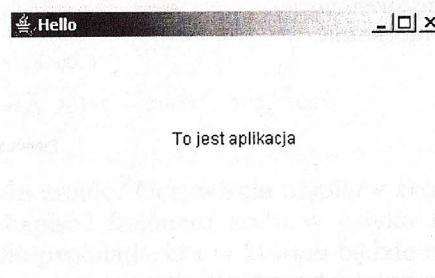
```

super("Hello");
setSize(320, 200);
setVisible(true);
}
public void paint(Graphics gDC)
{
    gDC.drawString ("To jest aplikacja", 120, 100);
}
public static void main(String args[])
{
    new HelloApp();
}
}

```

Warto zwrócić uwagę, że powstały w ten sposób okno (rysunek 5.2) nie da się zamknąć w tradycyjny sposób. Musimy przełączyć się na konsolę (wiersz poleceń), z której uruchamialiśmy tę aplikację, i wcisnąć kombinację klawiszy *Ctrl+C*.

Rysunek 5.2.
Aplikacja w Javie
wyświetlająca
na ekranie okno
z napisem



Jak to działa?

Aplet z ćwiczenia 5.1 dziedziczy z predefiniowanej klasy Applet, która opisuje zachowania apliów. Jest ona zawarta w pakiecie *java.awt.applet*, dlatego też na początku kodu znajduje się instrukcja:

```
import java.applet.*;
```

Jest w nim zadeklarowana tylko jedna metoda, mianowicie *paint*, która jest zawsze wywoływana przez system, kiedy zachodzi potrzeba odrysowania (odświeżenia) okna apletu, np. kiedy 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*) i jest to tzw. *graficzny kontekst urządzenia* (ang. *Graphics Device Context*), przez niektórych zwany również *wykreślaczem*¹. 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 naszym przypadku jest to napis podany jako pierwszy parametr metody *drawString*. Dwa kolejne parametry to współrzędne, od których zacznie się rysowanie tego napisu.

Nam skompilowany kod apletu nie wystarczy. Musimy jeszcze umieścić go na stronie WWW, tak aby przeglądarka wiedziała, gdzie go szukać i jak interpretować. Nie będziemy tutaj omawiać szczegółów tworzenia stron WWW w języku HTML. Opiszymy tylko jedyny interagujący nas w tej chwili znacznik, czyli *<applet>*. Jego definicja jest następująca:

```

<applet
    code = "nazwaKlasy.class"
    codebase = "URL"
    width = "szerokość"
    height = "wysokość"
    name = "nazwa apletu"
    align = "położenie"
    vspace = "odstęp w pionie"
    hspace = "odstęp w pionie"

    param name="nazwa 1" value="wartość 1"
    ...
    param name="nazwa n" value="wartość n">
</applet>

```

Parametr *code* określa nazwę głównej klasy apletu, *codebase* podaje natomiast lokalizację plików class zawierających b-kod. Domyślnym parametrem *codebase* jest lokalizacja, w której znajduje się plik z kodem HTML. Parametry *width* i *height* określają szerokość i wysokość apletu, natomiast *align*, *vspace* i *hspace* — jego położenie względem innych elementów na stronie WWW. Znaczniki *param* określają nazwy i wartości dodatkowych parametrów możliwych do przekazania apletowi. Niezbędne są jednak jedynie parametry *code*, *width* oraz *height* i na razie tylko te będziemy stosować.

Innym sposobem przekazywania parametrów jest pominięcie znaczników *param* i umieszczenie ich (parametrów) bezpośrednio w znaczniku

¹ Używany jest też termin *kontekst rysowniczy*.

<applet>. Z równym powodzeniem możemy więc zastosować konstrukcję w postaci:

```
<applet
  code = "nazwaKlasy.class"
  <!-- Pozostałe parametry znacznika-->
  nazwal = wartość
  <!-- ... -->
  nazwan = wartość
>
```

Cykl życia apletu

Kiedy przeglądarka obsługująca Javę odnajdzie w kodzie HTML znacznik <applet>, wykonuje określona sekwencję czynności. Zaczyna od sprawdzenia, czy w podanej lokalizacji znajduje się plik zawierający kod klasy wymienionej jako wartość argumentu code tego znacznika. Jeśli plik jest obecny we wskazanej lokalizacji, zawarty w nim kod jest ładowany 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 inicjalizacyjne.

Kiedy 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 kiedy aplet powinien zakończyć swoje działanie, wywoływana jest z kolei jego metoda stop.

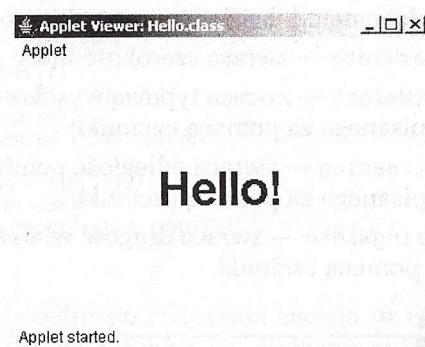
Czcionki

Napis, który wyświetliśmy w pierwszym aplecie, wyglądał nieco misernie. Warto by przynajmniej nieco go powiększyć i pogrubić. W tym celu musimy zmienić czcionkę. Aby tego dokonać, trzeba stworzyć nowy obiekt typu Font, a następnie użyć metody setFont.

Zastosowanie wybranego kroju czcionki

Napisz aplet wyświetlający na ekranie napis pogrubioną czcionką SansSerif o wielkości 36 punktów (rysunek 5.3).

Rysunek 5.3.
Aplet wyświetlający napis wybraną czcionką



```
import java.applet.*;
import java.awt.*;

public class Hello extends Applet
{
    public void paint (Graphics gDC)
    {
        Font font = new Font ("SansSerif", Font.BOLD, 36);
        gDC.setFont (font);
        gDC.drawString ("Hello!", 110, 110);
    }
}
```

W konstruktorze obiektu typu Font podajemy nazwę czcionki, a następnie jej typ oraz wielkość. Do dyspozycji mamy następujące typy:

- Font.PLAIN — czcionka zwykła,
- Font.BOLD — czcionka pogrubiona,
- Font.ITALIC — czcionka pochylona.

W środowisku Java 2 dostępnych jest jedynie pięć czcionek logicznych:

- Serif,
- SansSerif,
- Monospaced,
- Dialog,
- DialogInput.

Nastąpiła tu spora zmiana w stosunku do pierwotnych wersji Javy (JDK 1.1), gdzie dostępnych było sześć rodzajów czcionek (wspólnie są jedynie Dialog i DialogInput).

Domyślnie, jeżeli nie ustawimy własnej czcionki, używana jest czcionka Dialog. Własności poszczególnych czcionek można poznać dzięki obiekowi `FontMetrics` oraz metodzie `getFontMetrics`. Obiekt ten ma kilkanaście metod, bardziej interesujące to:

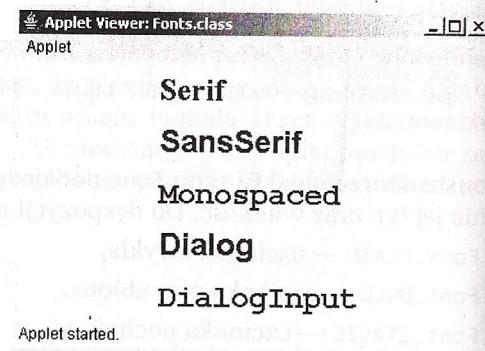
- ❑ `charWidth` — zwraca szerokość litery podanej jako parametr;
- ❑ `getHeight` — zwraca typową wysokość wiersza tekstu zapisanego za pomocą czcionki;
- ❑ `getLeading` — zwraca odległość pomiędzy wierszami tekstu zapisanego za pomocą czcionki;
- ❑ `stringWidth` — zwraca długość wiersza tekstu zapisanego za pomocą czcionki.

ĆWICZENIE

5.5. Dostępne kroje czcionek

Napisz applet prezentujący na ekranie dostępne standardowo kroje czcionek (rysunek 5.4).

Rysunek 5.4.
Aplet prezentujący
dostępne kroje
czcionek



```
import java.applet.*;
import java.awt.*;

public class Fonts extends Applet
{
    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);
}
```

W przeciwieństwie do poprzedniego ćwiczenia została tu wykorzystana metoda `init`, od której rozpoczyna się wykonywanie appletu. Tworzenie czcionek moglibyśmy, co prawda, wykonywać również w metodzie `paint`, ale w tym przypadku byłoby to gorsze rozwiązanie. Nie ma bowiem potrzeby tworzenia nowych obiektów klasy `Font` za każdym razem, gdy konieczne jest odświeżenie obszaru appletu, czyli w każdym wywołaniu metody `paint`. Powodowałoby to tylko niepotrzebne marnowanie zasobów systemowych. 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,
- ❑ luków,
- ❑ 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 prostą zaczynającą się w punkcie o współrzędnych x1, y1, a kończącą się w punkcie o współrzędnych x2, y2.

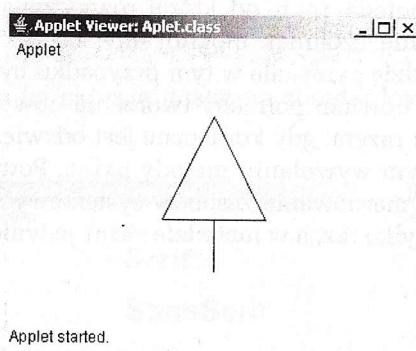
ĆWICZENIE

5.6. Rysowanie w obszarze apletu

Napisz aplet wyświetlający na ekranie strzałkę (rysunek 5.5).

Rysunek 5.5.

Efekt działania apletu z ćwiczenia 5.6



```
import java.applet.*;
import java.awt.*;

public class Aplet extends Applet
{
    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);
    }
}
```

ĆWICZENIE

5.7. Rysowanie linii

Korzystając z metody drawLine, napisz aplet wyświetlający na ekranie prostokąt (rysunek 5.6).

Rysunek 5.6.

Wynik działania apletu rysującego prostokąt

.Applet Viewer: Aplet.class
Applet

- [] x



```
import java.applet.*;
import java.awt.*;

public class Aplet extends Applet
{
    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);
    }
}
```

Koła i elipsy rysujemy metodami:

- drawOval (int x, int y, int width, int height),
- fillOval (int x, int y, int width, int height).

Podane parametry określają prostokąt okalający dane koło lub elipsę, x i y to współrzędne lewego górnego rogu, a width i height to odpowiednio szerokość i wysokość.

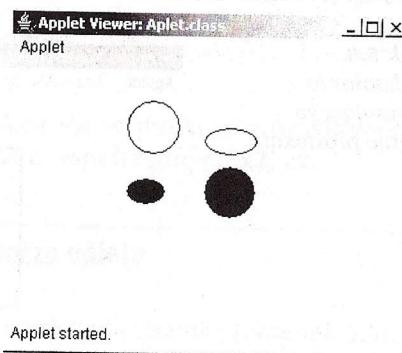
ĆWICZENIE

5.8. Rysowanie kół i elips

Napisz aplet wyświetlający na ekranie koła i elipsy (rysunek 5.7).

Rysunek 5.7.

Aplet wyświetlający na ekranie koła i elipsy



Applet started.

```
import java.applet.*;
import java.awt.*;

public class Aplet extends Applet
{
    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);
    }
}
```

Prostokąty można wyświetlać za pomocą metod:

- drawRect(int x, int y, int width, int height),
- fillRect(int x, int y, int width, int height).

Parametry x i y to współrzędne lewego górnego rogu, natomiast width i height to odpowiednio szerokość i wysokość danego prostokąta.

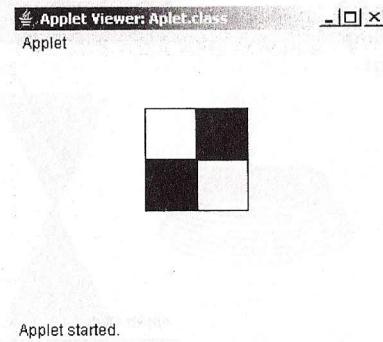
ĆWICZENIE

5.9. Rysowanie prostokątów

Napisz aplet rysujący na ekranie kilka prostokątów, takich jak na rysunku 5.8.

Rysunek 5.8.

Aplet rysujący prostokąty



Applet started.

```
import java.applet.*;
import java.awt.*;

public
class Aplet extends Applet
{
    public void paint (Graphics gDC)
    {
        gDC.drawRect (100, 40, 39, 39);
        gDC.fillRect (140, 40, 40, 40);
        gDC.fillRect (100, 80, 40, 40);
        gDC.drawRect (140, 80, 39, 39);
    }
}
```

Do rysowania wielokątów służą metody:

- drawPolygon(Polygon p),
- fillPolygon(Polygon p).

Parametr p jest typu Polygon, co oznacza, że musimy stworzyć obiekt takiego typu. Dokonujemy tego 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.

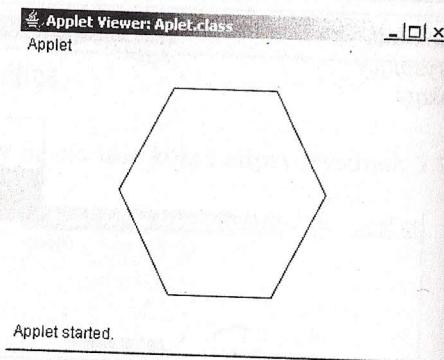
ĆWICZENIE

5.10. Rysowanie wielokątów

Napisz aplet wyświetlający na ekranie sześciokąt (rysunek 5.9).

```
import java.applet.*;
import java.awt.*;
```

Rysunek 5.9.
Aplet rysujący sześciokąt



```
public
class Aplet extends Applet
{
    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);
    }
}
```

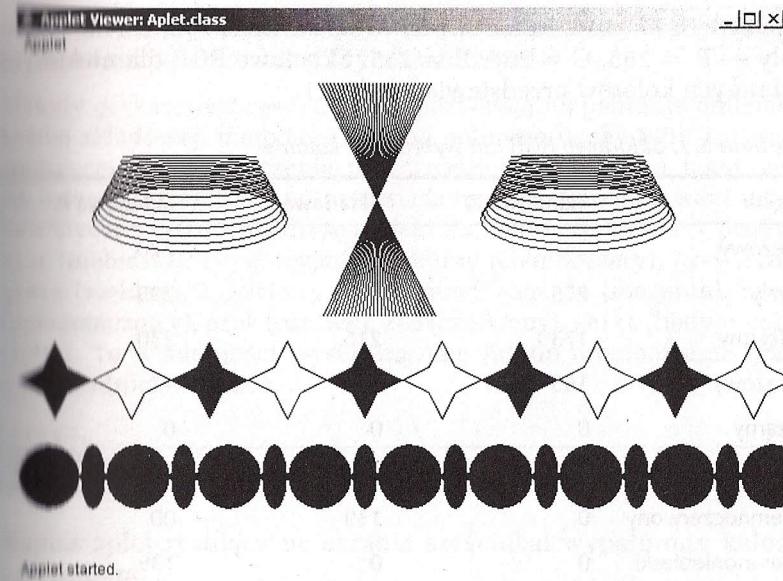
ĆWICZENIE

5.11. Rysowanie powtarzających się figur

Napisz aplet generujący na ekranie wzory widoczne na rysunku 5.10.

```
import java.applet.*;
import java.awt.*;

public
class Aplet extends Applet
{
    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 < 10; 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 * 60;
            }
        }
    }
}
```



Rysunek 5.10. Wzory generowane przez aplet z ćwiczenia 5.11

```
Polygon p = new Polygon(tempTabX, tabY, 9);
if (flag){
    gDC.drawPolygon(p);
}
else{
    gDC.fillPolygon(p);
}
flag = !flag;
```

Kolory

Do ustalania koloru używamy metody klasy Graphics o nazwie `setColor`, która przyjmuje jako parametr 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. Każda z tych składowych może opisywać przez 24 bity, maksymalna ich liczba wynosi zatem

2^{24} , czyli 16 777 216. Kolor czarny to składowe R = 0, G = 0, B = 0, biały — R = 255, G = 255, B = 255. Składowe RGB dla niektórych wybranych kolorów przedstawia tabela 5.1.

Tabela 5.1. Składowe RGB dla wybranych kolorów

Kolor	Składowa R	Składowa G	Składowa B
beżowy	245	245	220
biały	255	255	255
błękitny	173	216	230
brązowy	165	42	42
czarny	0	0	0
czerwony	255	0	0
ciemnoczerwony	0	139	00
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	128	128	0
purpurowy	128	0	128
srebrny	192	192	129
stalowoniebieski	70	130	180
szary	128	128	128
zielony	0	255	0
żółtozielony	154	205	50
żółty	255	255	0

Obiekt typu Color możemy stworzyć, 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.

Metody getRed, getGreen, getBlue pozwalają na pobranie oddzielnie każdej składowej, metoda getRGB na pobranie liczby typu int reprezentującej kolor. Znaczenie poszczególnych bitów jest takie samo, jak opisane wyżej. Klasa Color posiada również predefiniowane zmienne 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.

ćwiczenie

5.12. Kolorowy wielokąt

Napisz applet rysujący na ekranie sześciokąt wypełniony kolorem czerwonym.

```
import java.applet.*;
import java.awt.*;

public class Aplet extends Applet
{
    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);
    }
}
```

ćwiczenie

5.13. Kolorowanie figur

„Pokoloruj” wzory generowane przez applet z ćwiczenia 5.11.

```
import java.applet.*;
import java.awt.*;

public class Aplet extends Applet
{
    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 < 10; 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 * 60;
    }

    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;
}
}

```

Wyświetlanie obrazów

Umiemy już rysować proste figury geometryczne, nadszedł czas, by wyświetlić na ekranie grafikę zawartą w pliku. Klasa Applet na stronie zawiera odpowiednie metody służące temu celowi, 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. Inne formaty wymagają napisania własnych procedur wczytujących. Zakładając, że w katalogu, w którym jest umieszczony dokument HTML zawierający applet, znajduje się np. plik `obrazek.jpg`, możemy wczytać go następująco:

```
Image image = getImage (getDocumentBase(), "obrazek.jpg");
```

Jako rezultat otrzymujemy referencję do nowego obiektu klasy `Image`. Jest to właśnie wczytany, gotowy do wykorzystania obraz. Kiedy mamy już obraz, można go wyświetlić na ekranie. Wystarczy posłużyć się tutaj metodą `drawImage` klasy `Graphics`.

DEFINIE

5.14. Wyświetlanie zawartości pliku graficznego

Napisz applet wyświetlający na ekranie plik graficzny, np. obraz typu `jpg`.

```

import java.applet.*;
import java.awt.*;

public
class Aplet extends Applet
{
    Image img;
    public void init()
    {
        img = getImage(getDocumentBase(), "obrazek.jpg");
    }
    public void paint (Graphics gDC)
    {
        gDC.drawImage (img, 0, 0, this);
    }
}

```

Zastosowana metoda `getImage` przyjmuje jako parametr obiekt klasy URL (ang. *Universal Resource Locator*). W naszym przypadku jest to lokalizacja, w której znajdują się pliki z kodem HTML zawierającym applet. Adres ten uzyskujemy przez wywołanie metody `getDocumentBase`. Alternatywnie można użyć metody `getCodeBase`, która zwraca obiekt URL zawierający adres, pod którym znajduje się kod klasy appletu. Drugi parametr to nazwa pliku graficznego, który chcemy wczytać.

Metoda `drawImage` przyjmuje cztery parametry. 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 Aplet, dziedziczącej z Applet). W rzeczywistości parametr ten musi być odnośnikiem do obiektu klasy implementującej interfejs ImageObserver. W trakcie napływu danych z obrazem wywoływana jest metoda imageUpdate tego obiektu, co wykorzystamy w kolejnym ćwiczeniu.

ĆWICZENIE

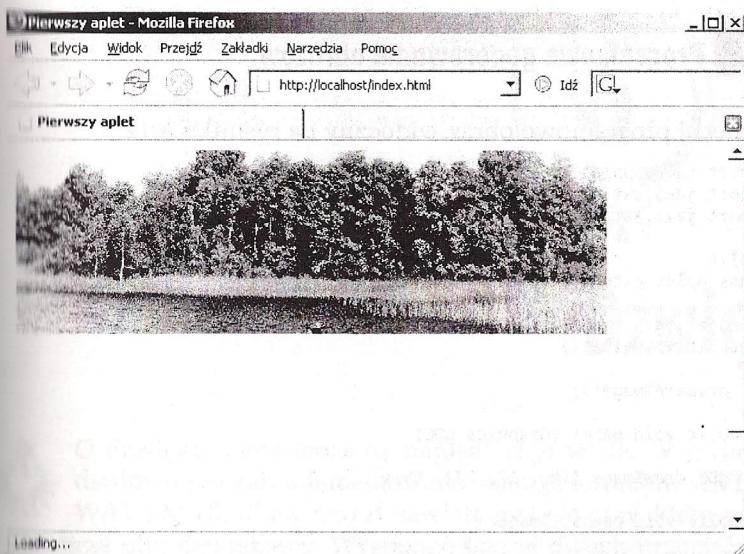
5.15. Śledzenie postępów ładowania obrazu

Napisz aplet wyświetlający na ekranie plik graficzny. Podczas wczytywania obrazka na pasku stanu przeglądarki powinien być wyświetlony napis "Loading..." (rysunek 5.11).

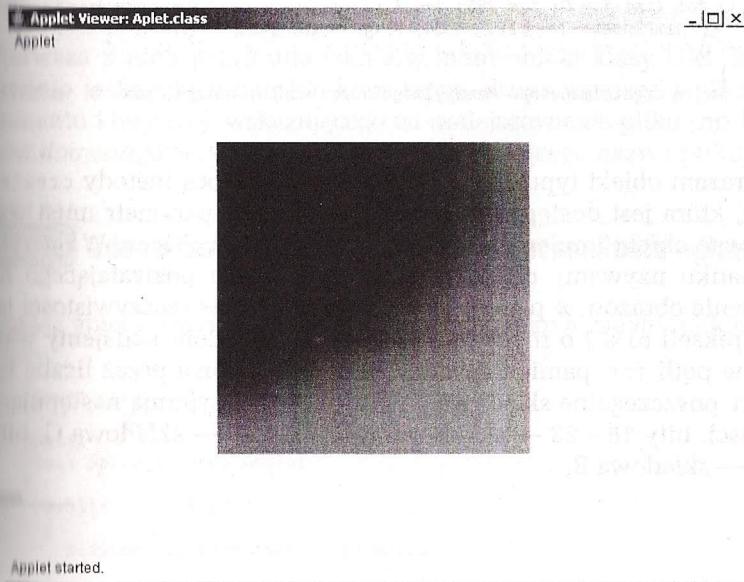
```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public
class Aplet extends Applet
{
    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 ("Loading...");
            getGraphics().drawImage(img, 0, 0, null);
            return true;
        }
        else{
            showStatus ("Loaded");
            getGraphics().drawImage(img, 0, 0, null);
            return false;
        }
    }
}
```

Spójrzmy teraz na rysunek 5.12. Jest to również obraz rysowany za pomocą metody drawImage. Nie jest jednak stworzony w programie graficznym, ale generowany w pamięci komputera.



Rysunek 5.11. Wyświetlanie informacji podczas ładowania pliku graficznego



Rysunek 5.12. Programowo wygenerowana paleta kolorów

ĆWICZENIE

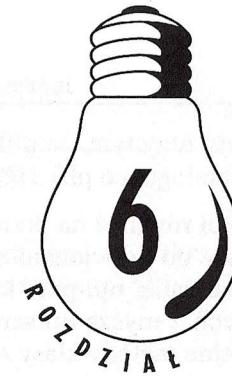
5.16. Programowe generowanie obrazów

Wygeneruj programowo obraz widoczny na rysunku 5.12.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public
class Aplet extends Applet
{
    Image img;
    public void init()
    {
        prepareImage();
    }
    public void paint (Graphics gDC)
    {
        gDC.drawImage (img, 172, 72, 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` tworzymy za pomocą metody `createImage`, która jest dostępna w klasie `Applet`. Jako parametr musi występować obiekt implementujący interfejs `ImageProducer`. W naszym przypadku używamy obiektu `MemoryImageSource` pozwalającego na tworzenie obrazów w pamięci. Obrazem tym jest w rzeczywistości tablica pikseli `pix[]` o rozmiarach 255×255 . Pikselom nadajemy wartości w pętli `for`, pamiętając, że w przypadku opisu przez liczbę typu `int` poszczególne składowe koloru piksela przyjmują następujące wartości: bity 16 – 23 — składowa R, bity 8 – 15 — składowa G, bity 0 – 7 — składowa B.



Dźwięki



O dźwiękach nie możemy napisać zbyt wiele. W Javie standardowo jest zaimplementowana obsługa formatów AIFF, AU, WAV i MIDI. Klasa `Applet` zawiera metodę `play`, która pobiera i odtwarza plik dźwiękowy. Występuje ona w dwóch przeciążonych wersjach:

- `play(URL url),`
- `play(URL url, String name).`

Pierwsza z nich przyjmuje jako argument obiekt klasy `URL` bezpośrednio wskazujący na plik dźwiękowy, druga wymaga podania argumentu klasy `URL` wskazującego na umiejscowienie pliku (np. `http://host.domena/java/sound/`) i drugiego, określającego nazwę pliku.

ĆWICZENIE

6.1. Odtwarzanie dźwięków podczas uruchamiania apletu

Napisz aplet odtwarzający w trakcie uruchamiania zadany plik dźwiękowy.

```
import java.applet.*;
public
class Sound extends Applet
{
    public void start()
    {
        play(getDocumentBase(), "yahoo1.au");
    }
}
```

Zakładamy przy tym, że plik o nazwie *yahoo1.au* znajduje się w tym samym katalogu, co plik HTML, w którym jest zawarty applet.

Nic nie stoi również na przeszkodzie, aby najpierw przygotować plik dźwiękowy do odtwarzania, a odtwarzać go dopiero przy zajściu jakiegoś zdarzenia, np. przy kliknięciu myszą. (Sposób obsługi zdarzeń związanych z myszą opisany jest w rozdziale 8.). Pomocne będą tu dwie kolejne metody klasy Applet:

- `getAudioClip (URL url),`
- `getAudioClip (URL baseURL, String nazwaPliku).`

ĆWICZENIE

6.2. Dźwiękowa reakcja na kliknięcia myszy

Napisz applet, który będzie wydawał dźwięk po naciśnięciu klawisza myszy przez użytkownika.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Sound extends Applet implements MouseListener
{
    AudioClip audioClip;
    public void init()
    {
        addMouseListener (this);
        audioClip = getAudioClip (getDocumentBase(), "yahoo1.au");
    }
    public void mousePressed (MouseEvent evt)
    {
        audioClip.play();
    }
    public void mouseClicked (MouseEvent evt){}
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
}
```

Znacznie ciekawsza byłaby jednak reakcja różnymi dźwiękami na kliknięcia w różnych obszarach ekranu. To również nie jest skomplikowane.

ĆWICZENIE

6.3.

Reagowanie różnymi dźwiękami

Napisz applet, który będzie reagował dźwiękami na kliknięcia w różnych obszarach ekranu. Obszary aktywne zaznacz różnymi kolorami.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Sound extends Applet implements MouseListener
{
    AudioClip audioClip1, audioClip2, audioClip3, audioClip4;
    FilledRectangle rect1, rect2, rect3, rect4;
    Color color1, color2, color3, color4;
    public void init()
    {
        addMouseListener (this);
        audioClip1 = getAudioClip (getDocumentBase(), "ding.au");
        audioClip2 = getAudioClip (getDocumentBase(), "beep.au");
        audioClip3 = getAudioClip (getDocumentBase(), "yahoo1.au");
        audioClip4 = getAudioClip (getDocumentBase(), "yahoo2.au");
        rect1 = new FilledRectangle (50, 50, 200, 100);
        rect2 = new FilledRectangle (350, 50, 200, 100);
        rect3 = new FilledRectangle (50, 250, 200, 100);
        rect4 = new FilledRectangle (350, 250, 200, 100);
        color1 = Color.red;
        color2 = Color.orange;
        color3 = Color.yellow;
        color4 = Color.blue;
    }
    public void paint (Graphics gDC)
    {
        gDC.setColor(color1);
        rect1.draw(gDC);
        gDC.setColor(color2);
        rect2.draw(gDC);
        gDC.setColor(color3);
        rect3.draw(gDC);
        gDC.setColor(color4);
        rect4.draw(gDC);
    }
    public void mousePressed (MouseEvent evt)
    {
        int x = evt.getX();
        int y = evt.getY();
        if (rect1.contains(x, y)){
            audioClip1.play();
        }
        else if (rect2.contains(x, y)){
            audioClip2.play();
        }
        else if (rect3.contains(x, y)){
            audioClip3.play();
        }
    }
}
```

```

        else if (rect4.contains(x, y)){
            audioClip4.play();
        }
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
}

import java.awt.*;
public
class FilledRectangle extends Rectangle
{
    public FilledRectangle(int x, int y, int width, int height)
    {
        super (x, y, width, height);
    }
    public void draw(Graphics gDC)
    {
        gDC.fillRect (x, y, width, height);
    }
}

```

Wykorzystaliśmy tutaj dodatkowo klasę `FilledRectangle`, która dziedziczy z klasy `Rectangle`. W `Rectangle` jest z kolei zdefiniowana metoda `contains`. Pozwala ona stwierdzić, czy podane jako parametry współrzędne należą do danego prostokąta. Jeżeli tak, zwracana jest wartość `true`, jeżeli nie — `false`.



Animacje

Pływający napis



Wiemy już, jak rysować i wyświetlać grafikę w Javie, nadszedł czas, by zająć się animacjami. Zaczniemy od wyświetlenia przesuwającego się w poziomie napisu.

WIEDZIEŃ

7.1. Próba animacji napisu

Napisz applet pozwalający na animację napisu.

```

import java.awt.*;
import java.applet.*;

public
class Aplet extends Applet
{
    int x, y, j, width;
    String napis;
    Font fontTimesRoman;
    public void init()
    {
        x = 0;
        y = 180;
        j = 1;
        width = getWidth();
        napis = new String ("HOP HOP");
        fontTimesRoman = new Font ("SansSerif", Font.BOLD, 36);
    }
    public void paint(Graphics gDC)

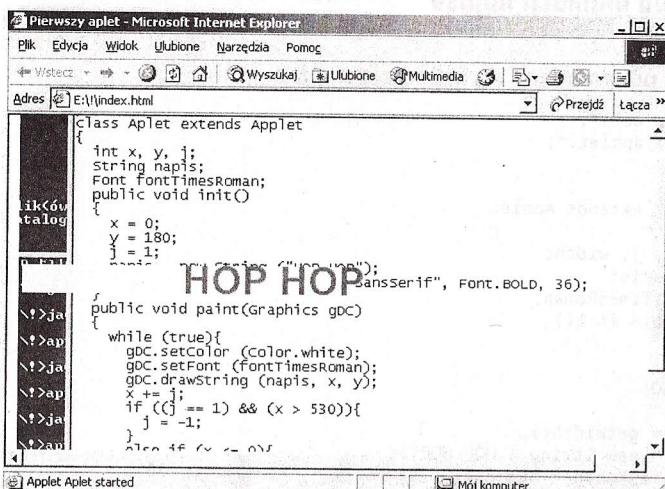
```

```

    {
        while (true){
            gDC.setColor (Color.white);
            gDC.setFont (fontTimesRoman);
            gDC.drawString (napis, x, y);
            x += j;
            if ((j == 1) && (x > width)){
                j = -1;
            }
            else if (x <= 0){
                j = 1;
            }
            gDC.setColor (Color.blue);
            gDC.setFont (fontTimesRoman);
            gDC.drawString (napis, x, y);
            for (int i = 0; i < 100000; i++)
                for (int j = 0; j < 30; j++);
        }
    }
}

```

Jest to realizacja najprostsza i, niestety, jednocześnie niezbyt dobra, mimo iż zaraz po uruchomieniu ujrzymy w miarę prawidłową (tylko migającą) animację napisu. Spróbujmy jednak zminimalizować i zmaksymalizować ten aplet. W zależności od tego, co akurat było na pulpicie, zobaczymy obraz jak na rysunku 7.1. Jest to oczywiście wynik braku prawidłowego odświeżania ekranu. Dlaczego ekran nie jest odświeżany? Dlatego że w metodzie paint, która się tym powinna zajmować, napisaliśmy niekontrolowaną nieskończoną pętlę — to poważny błąd, którego koniecznie trzeba unikać.



Rysunek 7.1. Wynik działania apletu. Błąd polegający na nieodświeżaniu ekranu podczas animacji

Co więcej, nie zamknijemy w zwykły sposób tego apletu, ponieważ jeśli go wyłącznie rysowaniem, a przeglądarka może się zacząć zachowywać w nieprzewidywalny sposób. Jeśli zostanie uruchomiony pod *appletviewerem*, nie da się zakończyć działania tej aplikacji w standardowy sposób. Zamiast tego na konsoli, w której aplikacja zastała uruchomiona, trzeba będzie przerwać jej działanie, weiskając kombinację klawiszy *Ctrl+C*. Zatem powyższa implementacja jest nie do przyjęcia. Aby prawidłowo zrealizować animację, musimy taki aplet napisać wielowątkowo. Nie mamy, niestety, miejsca na dokładne omówienie realizacji aplikacji wielowątkowych w Javie, przyjmijmy zatem, że taki aplet musi implementować interfejs *Runnable* oraz zawierać metodę *run*, od której zacznie się wykonywanie wątku. W metodzie *run* należy utworzyć nowy obiekt klasy *Thread*, a następnie wywołać jego metodę *start*. Nowy wątek rozpocznie się od metody *run* klasy apletu. Schematycznie taka konstrukcja będzie miała postać:

```

public
class NazwaKlasy extends Applet implements Runnable
{
    /*
     * wewnętrzne klasy NazwaKlasy
     */
    public void start()
    {
        Thread thread = new Thread (this);
        thread.start();
    }
    public void run()
    {
        //kod wątku uruchomionego instrukcją thread.start();
    }
}

```

LITERATURA

7.2. Animacja realizowana w oddzielnym wątku

Napisz wielowątkową wersję apletu animującego napis.

```

import java.awt.*;
import java.applet.*;

public
class Aplet extends Applet implements Runnable
{
    int x, y, j, width;
    String napis;
    Font fontTimesRoman;
    boolean stopped;

```

```

public void init()
{
    x = 0;
    y = 180;
    j = 1;
    width = getWidth();
    napis = new String ("HOP HOP");
    fontTimesRoman = new Font ("SansSerif", Font.BOLD, 36);
}
public void start()
{
    stopped = false;
    Thread thread = new Thread (this);
    thread.start();
}
public void run()
{
    while (!stopped){
        x += j;
        if ((j == 1) && (x > width)){
            j = -1;
        }
        else if (x <= 0){
            j = 1;
        }
        for (int i = 0; i < 100000; i++)
            for (int j = 0; j < 30; j++);
        repaint();
    }
}
public void stop()
{
    stopped = true;
}
public void paint(Graphics gDC)
{
    gDC.setColor (Color.blue);
    gDC.setFont (fontTimesRoman);
    gDC.drawString (napis, x, y);
}
}

```

Teraz całość działa już poprawie. Zwróćmy jednak uwagę na to, że prędkość animacji jest wciąż kontrolowana przez dwie pętle opóźniające. To też nie jest najlepsze rozwiązańe, gdyż uzależnia animację od prędkości komputera, na którym applet został uruchomiony. Lepszym rozwiązańiem jest wykorzystanie metody sleep, która usypia wątek na zadaną liczbę milisekund. Przy stosowaniu tej metody należy tylko pamiętać o przejęciu obsługi wyjątku InterruptedException, który jest generowany, jeżeli bieżący wątek zostanie przerwany przez inny proces. Oczywiście, obsługa tego wyjątku może być (i w naszym przypadku będzie) pusta.

ćwiczenie

7.3. Kontrolowanie prędkości animacji

Zastąp pętlę opóźniającą animację z ćwiczenia 7.2 metodą sleep. Pamiętaj o obsłudze wyjątku InterruptedException.

```

import java.awt.*;
import java.applet.*;

public class Aplet extends Applet implements Runnable
{
    int x, y, j, width;
    String napis;
    Font fontTimesRoman;
    boolean stopped;
    public void init()
    {
        x = 0;
        y = 180;
        j = 1;
        width = getWidth();
        napis = new String ("HOP HOP");
        fontTimesRoman = new Font ("SansSerif", Font.BOLD, 36);
    }
    public void start()
    {
        stopped = false;
        Thread thread = new Thread (this);
        thread.start();
    }
    public void run()
    {
        while (!stopped){
            x += j;
            if ((j == 1) && (x > width)){
                j = -1;
            }
            else if (x <= 0){
                j = 1;
            }
            try{
                Thread.sleep(10);
            }
            catch (InterruptedException e){}
            repaint();
        }
    }
    public void stop()
    {
        stopped = true;
    }
    public void paint(Graphics gDC)
    {
        gDC.setColor (Color.blue);
    }
}

```

```

gDC.setFont (fontTimesRoman);
gDC.drawString (napis, x, y);
}
}

```

Pływający napis z buforowaniem

Ostatni przykład działał już bardzo dobrze, wciąż miał jednak jedną nieprzyjemną wadę — napis, niestety, migał. To miganie występuje wtedy, gdy w krótkim odstępie czasu rysujemy coś i ścieramy w jednym miejscu ekranu. Rozwiązaniem tego problemu jest zastosowanie techniki tzw. podwójnego buforowania, w której wykorzystuje się naprzemiennie dwa obrazy. Kiedy jeden jest wyświetlany na ekranie, na drugim rysowana jest kolejna klatka animacji. Następnie dochodzi do zamiany. Obraz, na którym było wykonywane rysowanie, jest wyświetlany, a na obrazie, który był wyświetlany, następuje rysowanie. Takie wymiany przeprowadzane są cyklicznie, dzięki czemu animacja trwa, a mitotanie nie występuje.

ĆWICZENIE

7.4. Animacja z buforowaniem obrazu

Napisz aplet animujący na ekranie napis. Użyj buforowania obrazu w celu usunięcia efektu migotania.

```

import java.awt.*;
import java.applet.*;

public
class Aplet extends Applet implements Runnable
{
    int x, y, j;
    String napis;
    Font fontTimesRoman;
    Image img;
    Graphics gDC, mDC;
    boolean stopped;
    int width, height;
    public void init()
    {
        x = 0;
        y = 180;
        j = 1;
    }
}

```

```

width = getWidth();
height = getHeight();
napis = new String ("HOP HOP");
fontTimesRoman = new Font ("SansSerif", Font.BOLD, 36);
img = createImage (width, height);
mDC = img.getGraphics();
gDC = getGraphics();
}
public void start()
{
    stopped = false;
    Thread thread = new Thread (this);
    thread.start();
}
public void run()
{
    while (!stopped){
        x += j;
        if ((j == 1) && (x > width)){
            j = -1;
        }
        else if (x <= 0){
            j = 1;
        }
        try{
            Thread.sleep(1);
        }
        catch (InterruptedException e){}
        mDC.clearRect(0, 0, width, height);
        mDC.setColor (Color.blue);
        mDC.setFont (fontTimesRoman);
        mDC.drawString (napis, x, y);
        gDC.drawImage (img, 0, 0, this);
    }
}
public void stop()
{
    stopped = true;
}
public void paint(Graphics gDC)
{
    gDC.setColor (Color.blue);
    gDC.setFont (fontTimesRoman);
    gDC.drawString (napis, x, y);
}
}

```

W ćwiczeniu wykorzystujemy obiekt `img` klasy `Image`, który jest tworzony poprzez wywołanie metody `createImage`. Metoda ta przyjmuje jako parametry szerokość oraz wysokość obrazu podaną w pikselach. W tym przypadku rozmiar ten musi być zgodny z rozmiarami apletu. Szerokość i wysokość apletu jest pobierana za pomocą metod `getWidth` (szerokość) oraz `getHeight` (wysokość) i zapisywana w zmiennych `width` i `height`. Następnie za pomocą metody `getGraphics` otrzymujemy

graficzny kontekst urządzenia dla utworzonego właśnie obrazu i przypisujemy go zmiennej referencyjnej `gDC`. Dzięki temu możemy na naszym obrazie rysować, tak jakbyśmy robili to na ekranie.

W metodzie `run` najpierw czyścimy obraz metodą `clearRect`, a następnie rysujemy na nim napis w nowym położeniu. To wszystko odbywa się w pamięci. Dopiero wywołanie metody `drawImage` na rzecz obiektu `gDC`, czyli kontekstu urządzenia związanego z ekranem apletu, powoduje wyświetlenie obrazu na ekranie. Zrezygnowaliśmy przy tym całkowicie z metody `paint`, przejmując jej funkcje. W końcu ekran i tak musimy odświeżać samodzielnie — bez przerwy. Metodę `paint` można, rzecz jasna, dopisać. Wtedy w metodzie `run` zamiast wywołania:

```
gDC.drawImage (img, 0, 0, this);
```

znalazłaby się instrukcja:

```
paint(getGraphics());
```

Natomiast metoda `paint` wyglądałaby następująco:

```
public void paint(Graphics gDC)
{
    gDC.drawImage (img, 0, 0, this);
}
```

Zegar cyfrowy

Do stworzenia zegara wyświetlającego na ekranie czas (rysunek 7.2) potrzebny nam będzie obiekt typu `GregorianCalendar` udostępniający dane związane z datą i czasem. Znajduje się on w pakiecie `java.util`, musimy więc ten pakiet zaimportować. Obiekt ten udostępnia metodę `get`, w której jako parametr podajemy liczbę typu `int` wskazującą, jakie dane chcemy otrzymać. Najwygodniej użyć tutaj predefiniowanych stałych zawartych w tej klasie. Jeżeli zatem chcemy poznać aktualną godzinę, wywołujemy metodę `get` w następujący sposób:

```
get (GregorianCalendar.HOUR);
```

Czas będziemy aktualizować co sekundę (1000 milisekund), zapisując go w zmiennej `czas`. Co sekundę będzie też wywoływana metoda `repaint`, która odświeży ekran.

Rysunek 7.2.

Przykładowy wygląd zegara cyfrowego

01:35:58 PM

Applet started.

WZĘZENIE

7.5.

Tworzenie zegara cyfrowego

Napisz aplet wyświetlający na ekranie zegar cyfrowy, taki jak przedstawiony na rysunku 7.2.

```
import java.awt.*;
import java.applet.*;
import java.util.*;

public
class ZegarCyfrowy extends Applet implements Runnable
{
    Font fontTimesRoman;
    String czas;
    int i = 0;
    boolean stopped;
    public void init()
    {
        czas = new String();
        fontTimesRoman = new Font ("SansSerif", Font.BOLD, 36);
        stopped = false;
    }
    public void start()
    {
        Thread thread = new Thread (this);
        thread.start();
    }
    public void stop()
    {
        stopped = true;
    }
    public void run()
    {
        while (!stopped){
            try{
                Thread.sleep(1000);
            }
            catch(InterruptedException e){}
        }
    }
}
```

```

        updateTime();
        repaint();
    }

    public void updateTime()
    {
        int g, m, s, AM_PM;
        GregorianCalendar calendar = new GregorianCalendar();
        s = calendar.get(Calendar.SECOND);
        m = calendar.get(Calendar.MINUTE);
        g = calendar.get(Calendar.HOUR);
        AM_PM = calendar.get(Calendar.AM_PM);
        czas = "";
        if (g <= 9){
            czas += "0";
        }
        czas += g + ":";
        if (m <= 9){
            czas += "0";
        }
        czas += m + ":";
        if (s <= 9){
            czas += "0";
        }
        czas += s + " ";
        if (AM_PM == 0){
            czas += "AM";
        }
        czas += "PM";
    }

    public void paint (Graphics gDC)
    {
        gDC.setFont(fontTimesRoman);
        gDC.drawString (czas, 105, 100);
    }
}

```

Animacja poklatkowa

Sposób realizacji animacji poklatkowej jest bardzo prosty. Najpierw pobieramy obrazy z wybranej lokalizacji, a następnie wyświetlamy je po kolej na ekranie. Najłatwiej będzie zgromadzić wszystkie zmienne referencyjne wskazujące na poszczególne obrazy w obiekcie typu Vector. Co prawda, przy niewielkiej ilości klatek można również zdefiniować kilka zmiennych klasy Image, lepszy jest jednak pierwszy sposób.

WIZUALNE

7.6.

Wyświetlanie animacji poklatkowej

Napisz aplet realizujący animację poklatkową z przygotowanych wcześniej pojedynczych obrazów.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;

public class Animacja extends Applet implements Runnable
{
    Graphics gDC, mDC;
    Vector vector;
    Image img1;
    boolean stopped;
    int width, height;
    public void init()
    {
        width = getWidth();
        height = getHeight();
        vector = new Vector();
        gDC = getGraphics();
        img1 = createImage(width, height);
        mDC = img1.getGraphics();
        prepareImages();
    }
    public void start()
    {
        stopped = false;
        Thread thread = new Thread (this);
        thread.start();
    }
    public void stop()
    {
        stopped = true;
    }
    public void run()
    {
        int i = 0;
        int j = 1;
        Image img;
        while (!stopped){
            mDC.clearRect(0, 0, width, height);
            img = (Image) vector.elementAt(i);
            mDC.drawImage (img, 50, 100, this);
            gDC.drawImage (img1, 0, 0, this);
            try{
                Thread.sleep(100);
            }
            catch(InterruptedException e){}
            i += j;
            if (i >= 9){

```

```

        j = -1;
    } else if (i <= 0){
        j = 1;
    }
}
public void prepareImages()
{
    Image img;
    for (int i = 1; i <= 10; i++){
        img = getImage(getDocumentBase(), "T" + i + ".gif");
        vector.addElement (img);
    }
}

```

Do przykładowej animacji można wykorzystać obrazy dostarczone razem z pakietem JDK. Należy je zapisać w katalogu, w którym znajduje się kod HTML zawierający osadzony applet. Przy tego typu realizacji pierwsza faza animacji może być nieco przerywana. Wynika to z faktu, że metoda `getImage` nie powoduje załadowania obrazka z podanej lokalizacji, a jedynie przygotowanie do tej czynności. Ładowanie odbywa się dopiero przy wydaniu pierwszego polecenia rysowania. Rozwiążanie tego problemu pozostawiam jednak Czytelnikom.

Jeśli podany kod będzie komplikowany za pomocą JDK 1.5, podczas komplikacji zostanie wyświetlona informacja: `Aplet.java uses unchecked or unsafe operations`. Szczegóły ostrzeżenia można uzyskać, wykonując komplikację za pomocą polecenia:

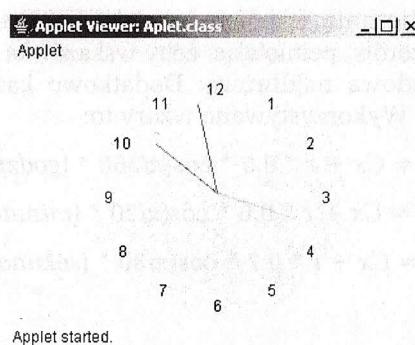
```
javac -Xlint:unchecked Aplet.java
```

Jest ono generowane ze względu na to, że do obiektu `vector` dodaje się elementy bez sprawdzania poprawności typu. Jest to jednak tylko sugestia kompilatora, która nie ma wpływu na działanie kodu.

Zegar analogowy

Przejdzmy teraz do bardziej zaawansowanego przykładu. Stwórzmy zegar analogowy. Analogowy, tzn. z tarczą ze wskazówkami (rysunek 7.3). To już nie będzie tak proste jak w przypadku poprzednich przykładów, ale z pewnością nie będzie również wymagało znajomości zaawansowanej algorytmiki. Podstawowe elementy appletu są takie same jak w przypadku przedstawionego wcześniej zegara cyfrowego.

Rysunek 7.3.
Efekt działania
appletu
wyświetlającego
zegar analogowy



Pozostaje nam więc tylko narysować tarczę zegara wraz z odpowiednio rozmieszczonymi godzinami oraz przygotować funkcję rysującą wskazówki.

W jaki sposób rozmieścić dwanaście liczb określających godziny w równych odstępach na kole zegara? Musimy sięgnąć do wiadomości z matematyki. Skoro pełne koło ma 2π radianów, to kolejne liczby powinny być oddalone od siebie o $2\pi/12$, czyli $\pi/6$ radiana. Do tego należy jeszcze całość „przekrącić” o $\pi/2$ w lewo, tak aby godzina 12, a wraz z nią i pozostałe, znajdowały się we właściwym miejscu. Teraz, korzystając z funkcji trygonometrycznych, możemy obliczyć współrzędne x i y każdej godziny. Przy czym będziemy starali się dopasowywać je automatycznie do wielkości tarczy. Konkretnie wzory wyglądają następująco:

$$\begin{aligned}x &= Cx + (r - digitHeight) * \cos(\pi * godzina / 6 - \pi/2) \\y &= Cy + (r - digitHeight) * \sin(\pi * godzina / 6 - \pi/2)\end{aligned}$$

po uproszczeniu da:

$$\begin{aligned}x &= Cx + (r - digitHeight) * \cos(\pi/6 * (godzina - 3)) \\y &= Cy + (r - digitHeight) * \sin(\pi/6 * (godzina - 3))\end{aligned}$$

gdzie:

- Cx — określa współrzędną x środka koła,
- Cy — określa współrzędną y środka koła,
- r — określa długość promienia,
- $digitHeight$ — określa wysokość cyfr,
- $godzina$ — określa kolejną godzinę.

Przy obliczaniu współrzędnych wskazówek korzystamy z analogicznych wzorów, pamiętając, żeby wskazówka godzinowa była najkrótsza, a sekundowa najdłuższa. Dodatkowo każdą narysujemy w innym kolorze. Wykorzystywane wzory to:

$$xH = Cx + r * 0.5 * \cos(\pi/360 * (\text{godzina} - 180))$$

$$xM = Cx + r * 0.6 * \cos(\pi/30 * (\text{minuta} - 15))$$

$$xS = Cx + r * 0.7 * \cos(\pi/30 * (\text{sekunda} - 15))$$

gdzie:

xH — określa współrzędną x końca wskazówki godzinowej,

xM — określa współrzędną x końca wskazówki minutowej,

xS — określa współrzędną x końca wskazówki sekundowej.

Współrzędne y uzyskujemy, zamieniając funkcję cosinus na sinus. W przypadku wyliczeń związanych ze współrzędną godzinową należy przedstawić aktualną godzinę jako liczbę minut, które upłyły od godziny zero, czyli dokonać przeliczenia: godzina = godzina * 60 + minuta.

ĆWICZENIE

7.7. Realizacja zegara analogowego

Napisz applet wyświetlający na ekranie zegar analogowy.

```
import java.applet.*;
import java.awt.*;
import java.util.*;

public class ZegarAnalogowy extends Applet implements Runnable
{
    int sizeX, sizeY, Cx, Cy, r;
    Color clockColor;
    boolean stopped;
    double hour, minute, second;
    Graphics mDC, gDC;
    Image img;

    public void init()
    {
        sizeX = getSize().width;
        sizeY = getSize().height;

        img = createImage(sizeX, sizeY);
        mDC = img.getGraphics();
```

```
gDC = getGraphics();

Cx = (sizeX / 2);
Cy = (sizeY / 2);

if (sizeX < sizeY){
    sizeY = sizeX;
}
else if (sizeX > sizeY){
    sizeX = sizeY;
}

r = (sizeX / 2);
clockColor = Color.yellow;

}

public void start()
{
    stopped = false;
    Thread thread = new Thread(this);
    thread.start();
}

public void run()
{
    while(!stopped){
        draw();
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e){}
    }
}

public void stop()
{
    stopped = true;
}

public void paint(Graphics gDC)
{
    draw();
}

public void draw()
{
    getTime();

    mDC.setColor(Color.white);
    mDC.fillRect(0, 0, getSize().width, getSize().height);
    mDC.setColor(clockColor);
    mDC.fillOval(Cx - sizeX / 2, Cy - sizeY / 2, sizeX, sizeY);

    drawDigits(mDC);
    drawHands(mDC);

    gDC.drawImage (img, 0, 0, null);
}

public void drawDigits(Graphics gDC)
{
    gDC.setColor(Color.black);

    for (int i = 1; i <= 12; i++){
```

```

int digitWidth = gDC.getFontMetrics().stringWidth(Integer.
➥toString(i));
int digitHeight = gDC.getFontMetrics().getHeight();
int x = (int)(Cx + (r - digitHeight) * Math.cos(Math.PI / 6 *
➥(i - 3)));
int y = (int)(Cy + (r - digitHeight) * Math.sin(Math.PI / 6 *
➥(i - 3)));
x -= digitWidth / 2;
y += digitHeight / 2;
gDC.drawString(Integer.toString(i), x, y);
}

public void drawHands(Graphics gDC)
{
    double hVal = hour * 60 + minute;

    int xH = (int)(Cx + (r * 0.6) * Math.cos(Math.PI/360 *
➥(hVal - 180)));
    int yH = (int)(Cy + (r * 0.6) * Math.sin(Math.PI/360 *
➥(hVal - 180)));

    int xM = (int)(Cx + (r * 0.65) * Math.cos(Math.PI / 30 *
➥(minute - 15)));
    int yM = (int)(Cy + (r * 0.65) * Math.sin(Math.PI / 30 *
➥(minute - 15)));

    int xS = (int)(Cx + (r * 0.7) * Math.cos(Math.PI / 30 *
➥(second - 15)));
    int yS = (int)(Cy + (r * 0.7) * Math.sin(Math.PI / 30 *
➥(second - 15)));

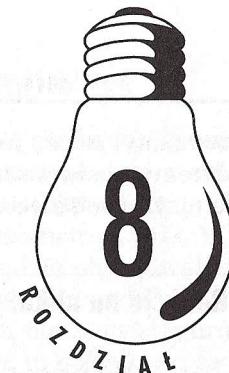
    gDC.setColor(Color.red);
    gDC.drawLine(Cx, Cy, xH, yH);
    gDC.drawLine(Cx - 1, Cy - 1, xH, yH);
    gDC.drawLine(Cx + 1, Cy + 1, xH, yH);

    gDC.setColor(Color.green);
    gDC.drawLine(Cx, Cy, xM, yM);
    gDC.drawLine(Cx - 1, Cy - 1, xM, yM);
    gDC.drawLine(Cx + 1, Cy + 1, xM, yM);

    gDC.setColor(Color.blue);
    gDC.drawLine(Cx, Cy, xS, yS);
}

public void getTime()
{
    GregorianCalendar calendar = new GregorianCalendar();
    hour = calendar.get(Calendar.HOUR);
    minute = calendar.get(Calendar.MINUTE);
    second = calendar.get(Calendar.SECOND);
}
}

```



Interakcja z użytkownikiem

Obsługa myszy

W celu wprowadzenie interakcji z użytkownikiem musimy nauczyć się, jak reagować na kliknięcie klawisza myszy czy też przesunięcie jej kurSORA. Nasz aplet będzie musiał w tym celu implementować 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, musimy 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 parametr otrzymuje referencję do obiektu `MouseEvent`, który dostarcza m.in. informacji o aktualnych współrzędnych, w jakich znajduje się kurSOR myszy. Oznacza to, że jeżeli naciśniemy klawisz, wystąpi zdarzenie `MousePressed` i zostanie wykonana powiązana z tym zdarzeniem metoda `mousePressed`. Jaki kod

¹ Drugą metodą jest wykorzystanie klas wewnętrznych, jednak omówienie tego tematu wykracza poza ramy książki.

w niej zawrzemy, zależy już tylko od nas. Zobaczmy to na przykładzie. Będziemy obsługiwać zdarzenie `mouseClicked` polegające na kliknięciu (czyli naciśnięciu i zwolnieniu) klawisza myszy.

ĆWICZENIE

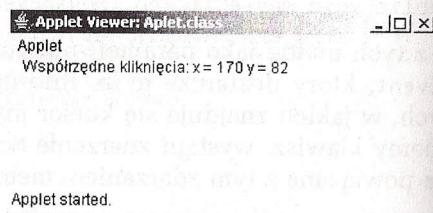
8.1. Reakcja na kliknięcie klawiszem myszy

Napisz aplet wyświetlający na ekranie współrzędne ostatniego kliknięcia myszy (rysunek 8.1).

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public
class Aplet extends Applet implements MouseListener
{
    private int x, y;
    String str;
    public void init()
    {
        addMouseListener (this);
    }
    public void paint (Graphics gDC)
    {
        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 myszy



Mimo że wykorzystujemy tylko metodę `mouseClicked`, musimy również zadeklarować wszystkie pozostałe, wymienione wyżej metody.

Nie będą one zawierały żadnego kodu, ale deklaracje muszą wystąpić. Metoda `repaint` powoduje wyczyszczenie ekranu apletu oraz wymusza wywołanie metody `paint`. 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 została uruchomiona. Warto na to zwrócić uwagę, gdyż łatwo tu o zwykłe przeoczenie.

Interfejs `MouseListener` obsługuje zdarzenia związane z naciśnięciem klawisza myszy. Co zrobić jednak, 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 na raz. `MouseMotionListener` wymaga zdefiniowania dwóch metod:

- public void mouseDragged (MouseEvent evt),
- public void mouseMoved (MouseEvent evt).

Pierwsza z nich jest wywoływana, jeżeli mysz jest przeciągana z wcisniętym klawiszem, druga — kiedy mysz jest tylko przesuwana.

ĆWICZENIE

8.2. Reakcja na zmianę położenia kurSORA myszy

Napisz aplet wyświetlający na ekranie współrzędne bieżącego położenia kurSORA myszy.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public
class Aplet extends Applet implements MouseMotionListener
{
    private int x, y;
    String str = new String();
    Font font;
    public void init()
    {
        font = new Font ("SansSerif", Font.BOLD, 18);
        addMouseMotionListener (this);
    }
    public void paint (Graphics gDC)
    {
        gDC.setFont (font);
        gDC.drawString (str, 80, 80);
    }
}
```

```

public void mouseDragged (MouseEvent evt)
{
    x = evt.getX();
    y = evt.getY();
    str = "Dragged: x = " + x + " y = " + y;
    repaint();
}
public void mouseMoved (MouseEvent evt)
{
    x = evt.getX();
    y = evt.getY();
    str = "Pozycja: x = " + x + " y = " + y;
    repaint();
}
}

```

Rysowanie figur (I)

Skoro wiemy już, jak posługiwać się myszą oraz jak rysować figury geometryczne, możemy stworzyć applet, który po kliknięciu myszą będzie rysował w danym punkcie jakąś figurę geometryczną, np. koło.

ĆWICZENIE

8.3. Rysowanie kół w miejscu kliknięcia myszą

Napisz applet rysujący koła w miejscu kliknięcia myszą.

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

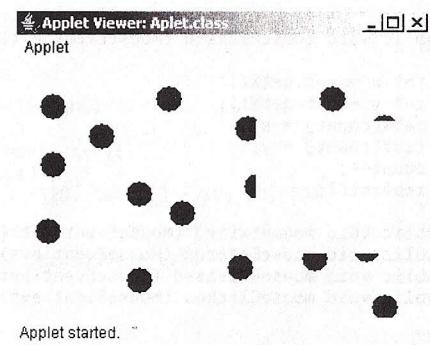
public
class Figury extends Applet implements MouseListener
{
    public void init()
    {
        addMouseListener (this);
    }
    public void mouseClicked (MouseEvent evt)
    {
        int x = evt.getX();
        int y = evt.getY();
        getGraphics().fillOval (x - 10, y - 10, 20, 20);
    }
    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, tym razem wykorzystujemy metodę getGraphics. Koła są natomiast rysowane za pomocą poznanej już w rozdziale 5. metody fillOval. Aplet działa poprawnie, jednak ma pewien mankament. Co się bowiem stanie, jeśli np. zamknijmy go częściowo innym oknem? Efekt może być taki jak na rysunku 8.2.

Rysunek 8.2.

Z powodu nieodświeżenia ekranu część kółek została starta



Okno appletu nie zostało odmalowane, tak więc część, która była zasłonięta, została bezpowrotnie zniszczona. Odmalowanie, oczywiście, nie mogło nastąpić, skoro nie użyliśmy metody paint. Sama metoda paint też nam nie pomoże, bo skąd miałaby wiedzieć, gdzie ma rysować nasze kółka? Niezbędne będzie zatem zapamiętywanie ich położień. W najprostszym przypadku możemy użyć do tego zmiennych tablicowych.

ĆWICZENIE

8.4. Użycie tablic do zapamiętywania współrzędnych

Napisz applet rysujący koła w miejscu kliknięcia myszą. Pamiętaj o odświeżaniu ekranu w metodzie paint.

```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public
class Figury extends Applet 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();
    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){}
}

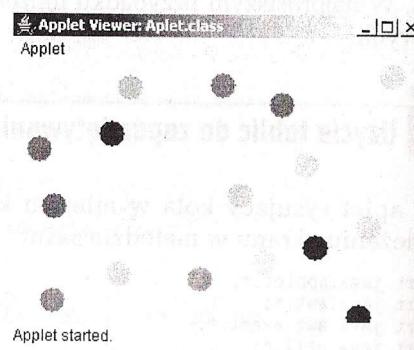
```

ĆWICZENIE**8.5. Wykorzystanie generatora liczb pseudolosowych**

Zmodyfikuj applet z ćwiczenia 8.4 tak, aby każde koło miało losowo wybrany kolor (rysunek 8.3).

Rysunek 8.3.

Aplet rysujący różnokolorowe koła



```

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

```

```

public class Figury extends Applet 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();
        tabX[count] = x;
        tabY[count] = y;
        colors[count++] = new Color (r.nextInt());
        repaint();
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
}

```

Teraz efekt jest dużo ciekawszy. Wykorzystaliśmy obiekt typu Random oraz jego metodę getNextInt, która zwraca losową liczbę całkowitą typu int. Kolory przechowujemy w tablicy colors, współrzędne x w tablicy wspX, a współrzędne y w tablicy wspY. W metodzie paint w pętli for rysujemy kółka o zapamiętanych wcześniej współrzędnych. Metoda fillOval rysuje wypełnione kolorem kółko wpisane w prostokąt o wymiarach podanych w parametrach metody. Dwa pierwsze parametry to współrzędne lewego górnego wierzchołka, dwa kolejne to szerokość i wysokość prostokąta.

Rysowanie figur (II)

Poprzedni przykład, jakkolwiek lepszy od wcześniejszego, ma również dosyć poważną wadę. Założyliśmy bowiem, że naszych kółek będzie nie więcej niż 100 i takie też ustaliliśmy rozmiary tablic. Co się jednak stanie, jeżeli figur będzie więcej? Otóż przekroczymy dopuszczalne wartości indeksów, w wyniku czego zostaną wygenerowane wyjątki i całość, niestety, nie będzie działać poprawnie. Moglibyśmy temu zaradzić, sprawdzając, czy liczba obiektów nie przekracza dopuszczalnych rozmiarów tablic. Jeśli tak, tworzylibyśmy nowe tablice o większych rozmiarach i przepisywali do nich wartości z tablicy dotychczasowych. Nie zaszkodzi spróbować.

ĆWICZENIE

8.6. Dynamiczne zwiększenie rozmiaru tablic

Zmień kod z ćwiczenia 8.5 w taki sposób, aby przy przekroczeniu dopuszczalnej liczby kół na ekranie nie występował błąd.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Figury extends Applet 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 tempTabX[], tempTabY[];
```

```
Color tempColors[];
int x = evt.getX();
int y = evt.getY();
tabX[count] = x;
tabY[count] = y;
colors[count++] = new Color (r.nextInt());
if (count >= tabX.length){
    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 zmieszcza się dane jeszcze jednego koła. Jeżeli nie, tworzymy nowe tablice o rozmiarach dwukrotnie większych od poprzednich oraz przepisujemy do nich wartość starych. Dalej referencjom `tabX`, `tabY` i `colors` przypisujemy nowe obiekty wskazywane przez `tempTabX`, `tempTabY` i `tempColors`. Warte uwagi jest to, że nie musimy zwalniać pamięci (nie mamy nawet takiej możliwości) przydzielonej wcześniej na obiekty `tabX`, `tabY` i `colors`. Tak zwany *garbage collector* maszyny wirtualnej zajmie się tym za nas.

Przedstawiony kod działa poprawnie, ale jest, można powiedzieć, mało obiektowy. Pozostanmy zatem jeszcze przez chwilę przy rysowaniu figur i napiszmy jeszcze jeden applet.

Rysowanie figur (III)

Zapomnijmy, przynajmniej częściowo, o wcześniejszych przykładach i spróbujmy zrobić wszystko jeszcze raz. Najpierw zastanówmy się, co tak naprawdę należy wykonać. Mamy rysować na ekranie kółka,

potrzebne więc nam będą obiekty klasy reprezentującej koła — nazwijmy ją `Circle`. Aby móc odświeżać ekran, musimy gdzieś takie kółka przechowywać. Potrzebna więc będzie kolejna klasa — niech nazywa się `CircleDatabase`. 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 koło. Klasa powinna mieć 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);
    }
}
```

Obiekty klasy `CircleDatabase` będą miały za zadanie przechowywać obiekty typu `Circle`. Moglibyśmy oczywiście stworzyć, podobnie jak wcześniej, tablice i zmieniać ich wielkość, zrobimy to jednak inaczej, za to dużo prościej. Skorzystamy z klasy `Vector`, która służy do przechowywania obiektów innych klas. Metodą `addElement` możemy dodać jakiś obiekt, metodą `elementAt` — pobrać.

8.8. Klasa przechowująca obiekty typu Circle

Napisz klasę `CircleDatabase` umożliwiającą przechowywanie obiektów typu `Circle`².

```
import java.awt.*;
import java.util.*;
public class CircleDatabase extends Vector
{
    private Vector vector;
    public CircleDatabase()
    {
        vector = new Vector();
    }
    public void add(Circle circle)
    {
        vector.addElement (circle);
    }
    public Circle getCircle(int index)
    {
        return (Circle) vector.elementAt (index);
    }
    public void drawAll(Graphics gDC)
    {
        for (int i = 0; i < vector.size(); i++)
            getCircle(i).draw(gDC);
    }
}
```

Wyjaśnienia wymaga zapewne metoda `get`, a dokładniej wiersz:

```
return (Circle) vector.elementAt (index);
```

Jak pamiętamy, dokonywaliśmy już konwersji typów, z tym że były to typy arytmetyczne. Wtedy była to konwersja typu `double` do typu `int`. Tutaj podobnie — musimy dokonać konwersji, 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 przypisanie wartości zmiennej `circle`, musimy dokonać konwersji z typu `Object` do typu `Circle`.

² Z powodu zmian wprowadzonych w JDK w wersji 1.5 podczas komplikacji przedstawionego kodu zostanie wygenerowane ostrzeżenie. Nie ma ono jednak wpływu na działanie klasy.

W rzeczywistości również w metodzie add została wykonana konwersja, tym razem odwrotna — z typu Circle do typu Object. Jest to jednak konwersja niejawnna, czyli dokonywana automatycznie przez kompilator. Jest to możliwe, dlatego że to Circle dziedziczy z Object, a nie odwrotnie.

ĆWICZENIE

8.9. Wykorzystanie klas Circle i CircleDatabase

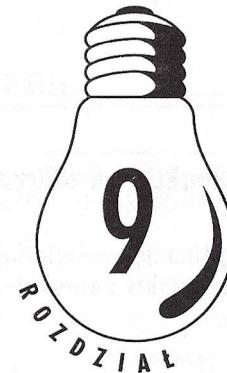
Napisz applet umożliwiający rysowanie kolorowych kół na ekranie. Skorzystaj z przygotowanych w poprzednich ćwiczeniach klas Circle i CircleDatabase.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Figury extends Applet 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.add (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 Aplet bardzo się uprościła w stosunku do wcześniejszych przykładów. 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 obiekowi 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 — wyświetlanie 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.



Okna i menu

Tworzenie okna aplikacji



W rozdziale piątym poznaliśmy różnicę między aplikacją a apletem. Wiemy zatem, że aplikacja jest samodzielny programem, który do uruchomienia potrzebuje tylko maszyny wirtualnej Javy. Nie jest natomiast potrzebna do tego celu żadna przeglądarka. Graficzną aplikację udało się nam już stworzyć w ćwiczeniu 5.3. Wyświetliliśmy na ekranie okno, a w nim zdefiniowany napis. Aplikacja ta miała jednak jedną wadę. Nie dało się jej zamknąć w standardowy sposób. Działało się tak, dlatego że nie obsłużyliśmy zdarzeń związanych z oknem, konkretnie zdarzenia generowanego przy próbie jego zamknięcia.

Aby zatem całość działała poprawnie, musimy dodać do naszej klasy `Interfejs WindowListener` oraz zdefiniowane w nim metody. Są to:

- `windowClosing` — wykonywana, kiedy okno jest zamknięte,
- `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, kiedy okno staje się nieaktywne.

ĆWICZENIE

9.1. Reakcja na zdarzenia związane z oknem

Napisz aplikację wyświetlającą na ekranie okno z napisem. Po kliknięciu przycisku zamkającego okno aplikacja powinna zakończyć działanie.

```
import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener
{
    public HelloApp ()
    {
        super("Moja pierwsza aplikacja");
        addWindowListener(this);
        setSize(320, 200);
        setVisible(true);
    }
    public void paint(Graphics gDC)
    {
        gDC.drawString ("To jest aplikacja.", 120, 100);
    }
    public static void main(String args[])
    {
        new HelloApp();
    }
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
    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 `System.exit(0)` w metodzie `windowClosing`. Identyczny efekt można również osiągnąć, wywołując metodę `dispose`. Ważne jest, aby w konstruktorze obiektu znalazła się linia `addWindowListener(this)` powodująca, że nasze okno zaczyna „słuchać” zdarzeń z nim związanych. Metoda `setSize` odpowiada za ustawienie wielkości okna, natomiast `setVisible` powoduje jego wyświetlenie na ekranie. Jest również wywoływany konstruktor klasy bazowej (`super`), któryremu w postaci argumentu przekazywany jest ciąg znaków, który pojawia 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 wykorzystywany przez nas przy pisaniu appletów.

Budowanie menu

Większość aplikacji posiada menu, niestety naszej brakuje na razie tej cechy. Dowiedzmy się zatem, w jaki sposób owo menu dodać, w tym celu należy bowiem wykonać kilka czynności. Po pierwsze, trzeba utworzyć obiekt klasy `MenuBar` i dodać go do okna za pomocą metody `setMenuBar`. Dopiero do tego obiektu będziemy mogli dodawać kolejne menu. Po drugie, dla każdego menu należy utworzyć obiekt klasy `Menu` oraz dla każdej pozycji w menu — obiekt klasy `MenuItem`. Najlepiej zobaczyć, jak to wygląda na praktycznym przykładzie.

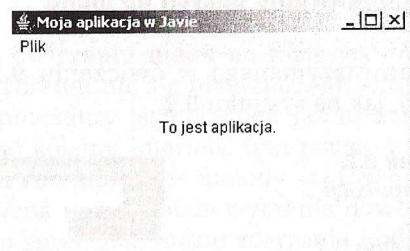
ĆWICZENIE

9.2. Dodanie menu do okna aplikacji

Do aplikacji z ćwiczenia 9.1 dodaj menu `Plik`, tak jak jest to widoczne na rysunku 9.1.

Rysunek 9.1.

Aplikacja
z dodanym
menu Plik



```
import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener
{
    public HelloApp ()
    {
        super();
    }
}
```

```

addWindowListener(this);
setSize(320, 200);
setTitle("Moja aplikacja w Javie");
MenuBar menuBar = new MenuBar();
setMenuBar(menuBar);
Menu menu = new Menu("Plik");
menuBar.add(menu);
setVisible(true);
}
public void paint(Graphics gDC)
{
    gDC.drawString ("To jest aplikacja.", 120, 100);
}
public static void main(String args[])
{
    new HelloApp();
}
public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
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){}
}

```

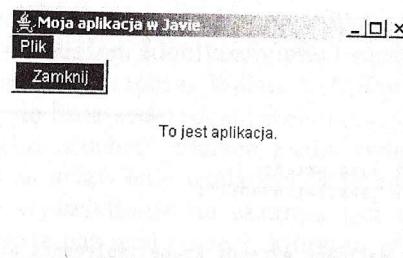
Do tak otrzymanego menu należałoby jednak dodać jakąś pozycję. Niech to będzie *Zamknij*. Będziemy mogli wykorzystać ją do końca pracy z aplikacją.

ĆWICZENIE

9.3. Dodawanie pozycji do menu

Do menu otrzymanego w ćwiczeniu 9.2 dodaj pozycję o nazwie *Zamknij*, jak na rysunku 9.2.

Rysunek 9.2.
Okno aplikacji
z menu
z ćwiczenia 9.3



```

import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener
{
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);
        Menu menu = new Menu("Plik");
        menu.add(new MenuItem("Zamknij"));
        menuBar.add(menu);
        setVisible(true);
    }
    public void paint(Graphics gDC)
    {
        gDC.drawString ("To jest aplikacja.", 120, 100);
    }
    public static void main(String args[])
    {
        new HelloApp();
    }
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
    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){}
}

```

Utworzone w poprzednim ćwiczeniu menu na razie jest, niestety, nieaktywne, tzn. po jego wybraniu nic się nie dzieje. Musimy dopiero stworzyć odpowiednie procedury jego obsługi. Jak to zrobić? Do klasy HelloApp trzeba dodać kolejny interfejs, tym razem ActionListener. Wymusza to z kolei zdefiniowanie metody actionPerformed. Metoda ta będzie wywoływana w momencie wybrania dowolnej pozycji z menu. Należy w niej sprawdzić rodzaj zdarzenia i odpowiednio na nie zareagować.

ĆWICZENIE

9.4. Obsługa menu

Uzupełnij kod z ćwiczenia 9.3 w taki sposób, aby po wybraniu z menu *Plik* pozycji *Zamknij* nastąpiło zakończenie pracy aplikacji.

```

import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);
        Menu menu = new Menu("Plik");
        menu.add(new MenuItem("Zamknij"));
        menu.addActionListener(this);
        menuBar.add(menu);
        setVisible(true);
    }
    public void paint(Graphics gDC)
    {
        gDC.drawString ("To jest aplikacja.", 120, 100);
    }
    public static void main(String args[])
    {
        new HelloApp();
    }
    public void actionPerformed(ActionEvent e)
    {
        String cmd = e.getActionCommand();
        if ("Zamknij".equals(cmd)){
            dispose();
        }
    }
    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){}
}

```

W konstruktorze klasy HelloApp znajduje się linia:

```
menu.addActionListener(this);
```

Kod ten oznacza, że chcemy, aby obiekt klasy HelloApp reagował na zdarzenia związane z obiektem menu. W momencie wybrania dowolnej pozycji z tego menu zostanie wywołana metoda actionPerformed. Jej parametrem jest obiekt ActionEvent, który pozwoli nam na określenie typu zdarzenia. Korzystamy tu z metody getActionCommand, która zwraca łańcuch znakowy (obiekt typu String). W przypadku przed-

stawionej aplikacji wybranie pozycji *Zamknij* spowoduje, że łańcuch ten będzie zawierał ciąg *Zamknij*¹. Dalej wystarczy już tylko skorzystać ze znanej instrukcji if, aby sprawdzić warunek. Do porównania ciągów wykorzystywana jest metoda equals. Zwraca ona wartość true, jeśli ciągi są sobie równe, lub wartość false w przeciwnym przypadku. Jeśli warunek jest prawdziwy, zamkamy okno aplikacji za pomocą metody dispose.

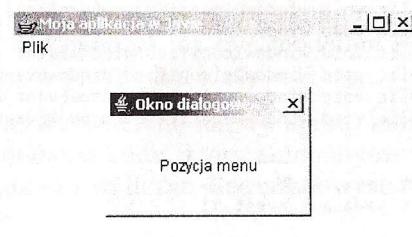
WICZENIE

9.5. Menu i okna dialogowe

Napisz aplikację zawierającą menu. Po wybraniu dowolnej pozycji z menu wyświetli jej nazwę w nowym oknie dialogowym (rysunek 9.3). Pozycja napisu w tym oknie powinna być dobierana automatycznie do jego rozmiaru.

Rysunek 9.3.

Po wybraniu pozycji z menu zostanie wyświetlane nowe okno dialogowe z jej nazwą



```

import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);

```

¹ W rzeczywistości ciąg znaków zwracany przez getActionCommand może być ustalany niezależnie od nazwy menu, za pomocą metody setActionCommand. Jeśli setActionCommand nie zostanie użyte, getActionCommand zwraca nazwę menu.

```

Menu menu = new Menu("Plik");
menu.add(new MenuItem("Pozycja menu"));
menu.addActionListener(this);
menuBar.add(menu);
setVisible(true);
}
public void paint(Graphics gDC)
{
    gDC.drawString ("To jest aplikacja.", 120, 100);
}
public static void main(String args[])
{
    new HelloApp();
}

public void actionPerformed(ActionEvent e)
{
    if(e.getSource() instanceof MenuItem)
        new MyDialog(this, "Okno dialogowe", e.getActionCommand());
}
public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
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){}
}

import java.awt.*;
import java.awt.event.*;

public
class MyDialog extends Dialog implements WindowListener
{
    String text;
    FontMetrics fontMetrics;
    int width = 160, height = 100;
    public MyDialog (Frame parent , String title , String text)
    {
        super(parent, title, true);
        addWindowListener(this);
        setSize(width, height);
        this.text = text;
        setVisible(true);
    }
    public void paint(Graphics gDC)
    {
        int strWidth, strHeight, xPos, yPos;
        fontMetrics = gDC.getFontMetrics();
        strWidth = fontMetrics.stringWidth(text);
        strHeight = fontMetrics.getHeight();
        xPos = (int) (width - strWidth) / 2;
        yPos = (int) (height - strHeight) / 2 + getInsets().top;
        gDC.drawString (text, xPos, yPos);
    }
    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){}
}
```

Jak widać, utworzona została nowa klasa o nazwie `MyDialog`, która została wyprowadzona ze standardowej klasy `Dialog`. Jedynym jej zadaniem jest wyświetlanie napisu w oknie dialogowym. Konstruktor zawiera trzy argumenty:

```
public MyDialog (Frame parent , String title , String text)
```

Parametr `parent` określa okno-rodzica, parametr `title` — tytuł okna, natomiast `text` — tekst, który ma się ukazać na ekranie. W konstruktorze klasy `MyDialog` wywołujemy konstruktor klasy bazowej, czyli klasy `Dialog`. Jego parametry to okno-rodzica, tytuł okna oraz zmienna typu `boolean`, określająca, czy okno to ma być modalne. Pozostałe konstrukcje są takie same jak w przypadku klasy `HelloApp`.

Ponieważ w oknie mają być wyświetlane nazwy menu, które mogą być różne, niezbędne jest dodanie kodu, który automatycznie pozyjonuje tekst. Pozycję początkową wyliczyć nietrudno, wystarczy skorzystać z ogólnych wzorów:

$$x = (\text{szerokość okna} - \text{długość tekstu}) / 2$$

$$y = (\text{wysokość okna} - \text{wysokość tekstu}) / 2 + \text{wysokość paska tytułu}$$

W uzyskaniu parametrów tekstu pomoże nam obiekt klasy `FontMetrics`. Występujące w powyższych wzorach wartości zobrazowane w kodzie następująco:

szerokość okna — wartość pola `width`,

wysokość okna — wartość pola `height`,

długość tekstu — wartość zwrócona przez wywołanie `fontMetrics.stringWidth(text)`,

wysokość tekstu — wartość zwrócona przez wywołanie `fontMetrics.getHeight()`,

wysokość paska tytułu — wartość zwrócona przez wywołanie `getInsets().top`.

Zwróćmy też uwagę na sposób obsługi zdarzenia (w klasie HelloApp) występującego po wybraniu dowolnej pozycji z menu. Otóż okno dialogowe jest wywoływanie tylko wtedy, gdy prawdziwy jest warunek:

```
e.getSource() instanceof MenuItem
```

Metoda getSource zwraca tu obiekt, w którym zostało zapoczątkowane zdarzenie, natomiast operator instanceof pozwala na stwierdzenie, czy jest to obiekt klasy MenuItem. A zatem okno dialogowe zostanie wywołane tylko wtedy, gdy źródłem zdarzenia będzie pozycja menu.

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, tak jak jest to widoczne na rysunku 9.4. Tego typu konstrukcję tworzy się analogicznie jak menu jednopoziomowe. Różnica jest taka, że menu, które ma być na drugim lub kolejnym poziomie, nie dodaje się do paska menu (obiektu klasyMenuBar), a do menu z wcześniejszego poziomu (obiektu klasy Menu). Najlepiej zobaczyć, jak to działa na konkretnym przykładzie.

ĆWICZENIE

9.6. Budowa wielopoziomowego menu

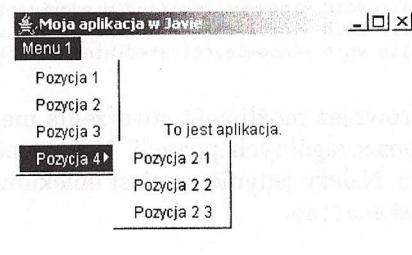
Napisz aplikację posiadającą wielopoziomowe menu, takie jak na rysunku 9.4. Po wybraniu dowolnej pozycji wypisz jej nazwę na konsole (w wierszu poleceń, z którego aplikacja została uruchomiona).

```
import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        MenuBar menuBar = new MenuBar();
```

Rysunek 9.4.

Aplikacja
z wielopoziomowym
menu



```
setMenuBar(menuBar);
Menu menu1 = new Menu("Menu 1");
MenuItem menuItem1 = new MenuItem("Pozycja 1");
MenuItem menuItem2 = new MenuItem("Pozycja 2");
MenuItem menuItem3 = new MenuItem("Pozycja 3");

Menu menu2 = new Menu("Pozycja 4");
MenuItem menuItem1_1 = new MenuItem("Pozycja 2 1");
MenuItem menuItem1_2 = new MenuItem("Pozycja 2 2");
MenuItem menuItem1_3 = new MenuItem("Pozycja 2 3");

menu1.add(menuItem1);
menu1.add(menuItem2);
menu1.add(menuItem3);

menu2.add(menuItem1_1);
menu2.add(menuItem1_2);
menu2.add(menuItem1_3);

menu1.add(menu2);
menu1.addActionListener(this);
menu2.addActionListener(this);

menuBar.add(menu1);
setVisible(true);
}

public void paint(Graphics gDC)
{
    gDC.drawString ("To jest aplikacja.", 120, 100);
}
public void actionPerformed(ActionEvent e)
{
    System.out.println("Wybrana pozycja to:" + e.getActionCommand());
}
public static void main(String args[])
{
    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){}
}
```

Istnieje również możliwość stworzenia menu umożliwiającego zaznaczanie poszczególnych pozycji. Nie wymaga to dużych modyfikacji w kodzie. Należy jedynie zamiast obiektów klasy MenuItem użyć klasy CheckboxMenuItem.

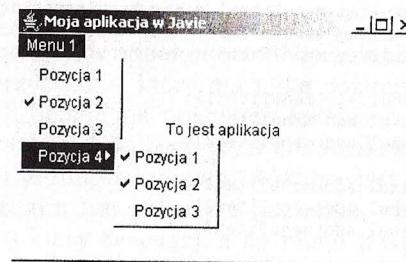
ĆWICZENIE

9.7. Menu umożliwiające zaznaczanie pozycji

Napisz aplikację zawierającą menu umożliwiające zaznaczanie poszczególnych pozycji, jak na rysunku 9.5.

Rysunek 9.5.

Dwupoziomowe
menu umożliwiające
zaznaczanie
poszczególnych
pozycji



```
import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener
{
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);
        Menu menu1 = new Menu("Menu 1");
        CheckboxMenuItem menu1item1 = new CheckboxMenuItem("Pozycja 1");
        CheckboxMenuItem menu1item2 = new CheckboxMenuItem("Pozycja 2");
        CheckboxMenuItem menu1item3 = new CheckboxMenuItem("Pozycja 3");

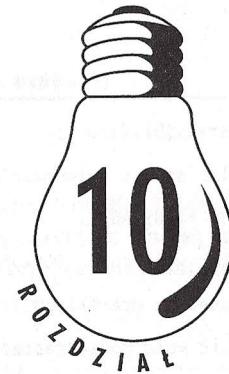
        Menu menu2 = new Menu("Pozycja 4");
        CheckboxMenuItem menu2item1 = new CheckboxMenuItem("Pozycja 1");
        CheckboxMenuItem menu2item2 = new CheckboxMenuItem("Pozycja 2");
        CheckboxMenuItem menu2item3 = new CheckboxMenuItem("Pozycja 3");
    }
}
```

```
menu1.add(menu1item1);
menu1.add(menu1item2);
menu1.add(menu1item3);

menu2.add(menu2item1);
menu2.add(menu2item2);
menu2.add(menu2item3);

menu1.add(menu2);
menuBar.add(menu1);
setVisible(true);

public void paint(Graphics gDC)
{
    gDC.drawString ("To jest aplikacja", 120, 100);
}
public static void main(String args[])
{
    new HelloApp();
}
public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
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){}
}
```



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 aparat rysujący koła z ćwiczenia 8.9 na aplikację.

WICZENIE

10.1. Aplikacja rysująca figury

Zmodyfikuj kod z ćwiczenia 8.9 w taki sposób, aby był on samodzielna aplikacją, niewymagającą przeglądarki do uruchamiania.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public
class FiguryApp extends Frame implements WindowListener, MouseListener
{
    private CircleDatabase database;
    private Random r;
    public FiguryApp()
    {
        super("Figury");
        addMouseListener (this);
        addWindowListener(this);
        database = new CircleDatabase();
        r = new Random();
        setSize(320, 200);
```

```

        setVisible(true);
    }
    public static void main(String args[])
    {
        new FiguryApp();
    }
    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.add (new Circle(x, y, color));
        repaint();
    }
    public void windowClosing(WindowEvent e)
    {
        dispose();
    }
    public void mouseExited (MouseEvent evt){}
    public void mouseEntered (MouseEvent evt){}
    public void mouseReleased (MouseEvent evt){}
    public void mouseClicked (MouseEvent evt){}
    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){}
}

```

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 obecna w aplecie). Niezbędne było też dodanie interfejsu WindowListener do obsługi zamknięcia okna oraz funkcji main, od której zaczyna się wykonywanie kodu. Zauważmy, że okno naszej aplikacji możemy dowolnie skalać, a ekran jest cały czas prawidłowo odświeżany.

Obsługa komponentów

Pakiet JDK oferuje kilka gotowych klas, które umożliwiają użycie typowych komponentów graficznych, takich jak przyciski, listy czy okna tekstowe. Bardzo ułatwia to tworzenie aplikacji graficznych. Na najbliższych stronach omówimy następujące klasy: Button, TextField, TextArea, Label, Checkbox oraz List.

Klasa Button

Klasa Button umożliwia utworzenie w oknie standardowego przycisku z dowolnym napisem. Aby z niej skorzystać, należy utworzyć nowy obiekt tej klasy, ustalić jego rozmiary oraz napisać procedurę obsługi zdarzeń.

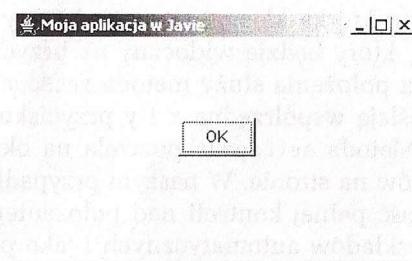
WICZENIE

10.2. Wykorzystanie przycisków

Umieść w oknie aplikacji przycisk z napisem OK (rysunek 10.1). Po kliknięciu przycisku program powinien zakończyć działanie.

Rysunek 10.1.

Przycisk umieszczony w oknie aplikacji



```

import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    Button button;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);
        button = new Button("OK");
        this.add(button);
        button.addActionListener(this);
        button.setActionCommand("cmdOK");
        button.setBounds(130, 85, 60, 30);
        setVisible(true);
        button.setVisible(true);
    }
    public static void main(String args[])
    {
        new HelloApp();
    }
}

```

```

}
public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand().equals("cmdOK")){
        dispose();
    }
}
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){}
}

```

W konstruktorze klasy `Button` podajemy — jako parametr — ciąg znaków, który będzie widoczny na przycisku. Do ustalenia rozmiarów oraz położenia służy metoda `setBounds`. Dwa pierwsze parametry określają współrzędne x i y przycisku, dwa kolejne — jego rozmiary. Metoda `setLayout` pozwala na określenie sposobu rozkładu elementów na stronie. W naszym przypadku, ponieważ chcemy mieć możliwość pełnej kontroli nad położeniem przycisku, nie korzystamy z rozkładów automatycznych i jako parametr podajemy wartość `null`. Za pomocą metody `setActionCommand` przypisujemy również przyciskowi ciąg znaków, który będzie zwracany po wywołaniu metody `getActionCommand`. Dzięki temu komenda zwracana przez `getActionCommand` może być różna od ciągu znaków znajdującego się na przycisku (por. opis menu z rozdziału 9.).

Klasa `TextField`

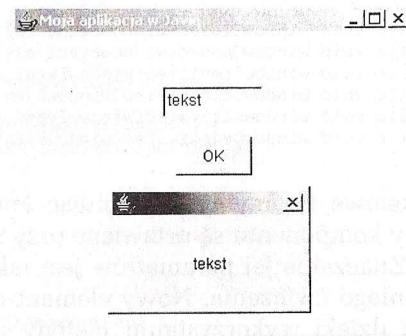
Klasa `TextField` służy do utworzenia pola tekstowego umożliwiającego wprowadzenie przez użytkownika ciągu znaków. Dane z tego pola mogą być następnie odczytane i wykorzystane w aplikacji.

ĆWICZENIE

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).

Rysunek 10.2.
Wyświetlenie napisu
wprowadzonego
w polu tekstowym



```

import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    Button button;
    TextField textField;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        button = new Button("OK");
        this.add(button);
        button.addActionListener(this);
        button.setActionCommand("cmdOK");
        button.setBounds(130, 105, 60, 30);

        textField = new TextField();
        textField.setBounds(120, 65, 80, 25);
        this.add(textField);

        setVisible(true);
        button.setVisible(true);
        textField.setVisible(true);
    }
    public static void main(String args[])
    {
        new HelloApp();
    }
    public void actionPerformed(ActionEvent e)
    {
        if (e.getActionCommand().equals("cmdOK")){
            new MyDialog(this, "", textField.getText());
        }
    }
    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){}
}

```

Pole tekstowe tworzymy, wywołując konstruktor klasy `TextField`. Rozmiary komponentu są ustawiane przy wykorzystaniu metody `setBounds`. Znaczenie jej parametrów jest takie samo, jak w przypadku poprzedniego ćwiczenia. Nowy element dodajemy do obszaru okna aplikacji dzięki wykorzystaniu metody `add(this.add(textField))`. W ćwiczeniu wykorzystaliśmy również klasę `MyDialog`, która powstała w przykładzie 9.5 z rozdziału 9.

ĆWICZENIE

10.4. Wykorzystanie wartości odczytanych z pola tekstopowego

Zmodyfikuj kod z ćwiczenia 10.3 w taki sposób, by po wprowadzeniu ciągu znaków `Zamknij` następowało zamknięcie aplikacji.

```

import java.awt.*;
import java.awt.event.*;

public class HelloApp extends Frame implements WindowListener, ActionListener
{
    Button button;
    TextField textField;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        button = new Button("OK");
        this.add(button);
        button.addActionListener(this);
        button.setActionCommand("cmdOK");
        button.setBounds(130, 105, 60, 30);

        textField = new TextField();
        textField.setBounds(120, 65, 80, 25);
        this.add(textField);

        setVisible(true);
        button.setVisible(true);
        textField.setVisible(true);
    }
    public static void main(String args[])
}

```

```

    }
    new HelloApp();
}
public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand().equals("cmdOK")){
        if (textField.getText().equals("Zamknij")){
            dispose();
        }
        else{
            new MyDialog(this, "", textField.getText());
        }
    }
}
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){}
}

```

W tym ćwiczeniu należy zwrócić szczególną uwagę na fakt, że nie używamy, oczywiście wydawałoby się, konstrukcji:

```
if (textField.getText() == "Zamknij")
```

Zamiast niej występuje linia:

```
if (textField.getText().equals("Zamknij"))
```

Otoż znana nam instrukcja warunkowa `if` wraz z operatorem `==` w tym przypadku nie zadziała, a właściwie nie zadziała zgodnie z naszymi oczekiwaniemi (można się o tym przekonać, wprowadzając ją do powyższego ćwiczenia). Dlaczego? Dzieje się tak, bowiem metoda `getText` zwraca referencję do obiektu typu `String` zawierającego ciąg znaków `Zamknij`. Podkreślam — referencję do obiektu, a nie sam ciąg znaków! Co więcej, znajdujący się po prawej stronie ciąg `Zamknij` jest również interpretowany jako referencja do obiektu typu `String`. Pisząc zatem:

```
if (textField.getText() == "Zamknij")
```

wykonujemy porównanie dwóch referencji, a nie dwóch napisów. Dlatego też wynik takiego porównania może nie być prawdziwy, mimo iż oba ciągi znaków zawierają to samo. W takich sytuacjach radzimy sobie jak w powyższym przypadku, stosując do porównywania metodę `equals` klasy `String`.

Klasa TextArea

Klasa `TextArea` służy do utworzenia komponentu umożliwiającego wprowadzenie przez użytkownika dłuższego tekstu. Posługujemy się nią podobnie jak klasą `TextField`. Obie wykorzystamy w kolejnym ćwiczeniu.

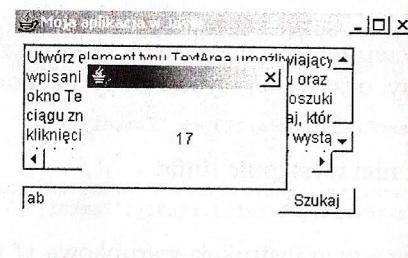
ĆWICZENIE

10.5. Wykorzystanie rozszerzonego pola tekstowego

Utwórz element typu `TextArea` umożliwiający wpisanie dowolnego dłuższego tekstu oraz okno `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).

Rysunek 10.3.

Aplikacja podająca ilość wystąpień szukanego ciągu znaków we wprowadzonym tekście



```
import java.awt.*;
import java.awt.event.*;

public class HelloApp extends Frame implements WindowListener, ActionListener
{
    Button button;
    TextField textField;
    TextArea textArea;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        button = new Button("Szukaj");
        this.add(button);
        button.addActionListener(this);
    }
}
```

```
button.setActionCommand("cmdSzukaj");
button.setBounds(210, 140, 60, 20);

textField = new TextField();
textField.setBounds(10, 140, 160, 20);
this.add(textField);

textArea = new TextArea();
textArea.setBounds(10, 30, 260, 100);
this.add(textArea);

setVisible(true);
button.setVisible(true);
textField.setVisible(true);
textArea.setVisible(true);

}

public static void main(String args[])
{
    new HelloApp();
}

public void szukaj()
{
    String tekst = textArea.getText();
    String ciag = textField.getText();
    int indeks = 0;
    int indeks_wystapienia = 0;
    int liczba_wystapien = 0;
    if(tekst.equals("") || ciag.equals ""){
        indeks_wystapienia = -1;
    }
    while (indeks_wystapienia != -1){
        indeks_wystapienia = tekst.indexOf (ciag, indeks);
        if (indeks_wystapienia != -1){
            indeks = indeks_wystapienia + 1;
            liczba_wystapien++;
        }
    }
    new MyDialog(this, "", String.valueOf(liczba_wystapien));
}

public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand().equals("cmdSzukaj")){
        szukaj();
    }
}

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){}
}
```

Jedyną nowością w powyższym ćwiczeniu jest wykorzystanie komponentu `TextArea` oraz metody `indexOf`. Jest ona zdefiniowana w klasie `String` i zwraca indeks szukanego ciągu znaków podanego jako pierwszy parametr. Drugi parametr wskazuje, od którego miejsca w ciągu ma się rozpocząć przeszukiwanie. Przeszukiwanie jest wykonywane w pętli `while`, a aktualna liczba wystąpień poszukiwanego ciągu jest zapisywana w zmiennej `liczba_wystapien`.

Klasa Label

Klasa `Label` służy do tworzenia etykiet, czyli obiektów wyświetlających tekst, który może być zmieniany przez aplikację, użytkownik nie ma natomiast możliwości jego bezpośredniej edycji. Aby utworzyć etykietę, należy wywołać konstruktor klasy oraz dodać nowy obiekt do okna aplikacji za pomocą metody `add`, analogicznie jak miało to miejsce w przypadku wcześniej omawianych komponentów. Zmianę wyświetlanego przez etykietę ciągu znaków osiągniemy, wywołując metodę `setText` i podając jako parametr nowy tekst.

ĆWICZENIE

10.6. Etykietą tekstową w oknie aplikacji

Dodaj do okna aplikacji komponent `Label` z dowolnym tekstem.

```
import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener
{
    Label label;
    public HelloApp()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        label = new Label("To jest etykieta.");
        label.setBounds(115, 90, 90, 20);
        this.add(label);

        setVisible(true);
        label.setVisible(true);
    }
}
```

```
public static void main(String args[])
{
    new HelloApp();
}
public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
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){}
```

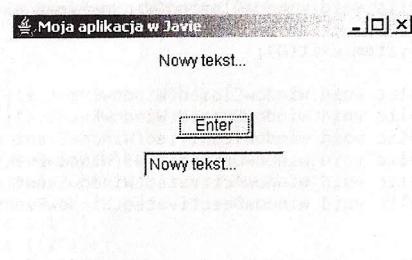
WICZENIE

10.7. Etykietą 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.

Rysunek 10.4.

Wygląd aplikacji
z ćwiczenia 10.7



```
import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    Label label;
    Button button;
    TextField textField;
    public HelloApp()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);

        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        button = new Button("Enter");
        button.addActionListener(this);
        button.setBounds(115, 115, 90, 20);
        this.add(button);

        textField = new TextField("Nowy tekst...");
        textField.setBounds(115, 90, 90, 20);
        this.add(textField);

        label = new Label("To jest etykieta.");
        label.setBounds(115, 90, 90, 20);
        this.add(label);

        setVisible(true);
    }
}
```

```

button.setBounds(130, 80, 60, 20);
this.add(button);
button.addActionListener(this);

textField = new TextField();
textField.setBounds(105, 110, 110, 20);
this.add(textField);

label = new Label("To jest etykieta");
label.setBounds(115, 30, 90, 20);
this.add(label);

setVisible(true);
label.setVisible(true);
button.setVisible(true);
textField.setVisible(true);
}

public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand().equals("Enter"))
        label.setText(textField.getText());
}

public static void main(String args[])
{
    new HelloApp();
}

public void windowClosing(WindowEvent e)
{
    System.exit(0);
}

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){}
}

```

Klasa Checkbox

Klasa Checkbox pozwala na umieszczenie w obszarze apletu pól wyboru, np. takich, jakie zostały zaprezentowane na rysunku 10.5. W celu utworzenia pojedynczego komponentu tego rodzaju należy utworzyć nowy obiekt typu Chcekbox oraz dodać go do apletu za pomocą metody add. W konstruktorze można podać ciąg znaków, który będzie wyświetlany obok pola, można również wykorzystać konstruktor bezargumentowy.

WICZENIE

10.8. Pola wyboru i rozkład automatyczny

Dodaj do okna aplikacji elementy typu Checkbox, rozmieszczone tak jak na rysunku 10.5.

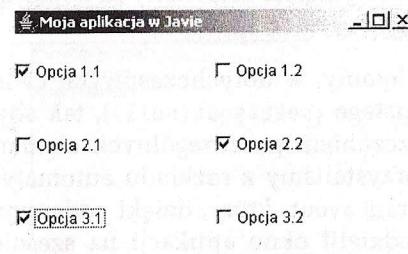
Rysunek 10.5.

Elementy typu

Checkbox

w rozkładzie

GridLayout



```

import java.awt.*;
import java.awt.event.*;

public class HelloApp extends Frame implements WindowListener
{
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(new GridLayout(3, 2));

        add(new Checkbox("Opcja 1.1"));
        add(new Checkbox("Opcja 1.2"));
        add(new Checkbox("Opcja 2.1"));
        add(new Checkbox("Opcja 2.2"));
        add(new Checkbox("Opcja 3.1"));
        add(new Checkbox("Opcja 3.2"));

        setVisible(true);
    }

    public static void main(String args[])
    {
        new HelloApp();
    }

    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }

    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){}
}
```

Jak łatwo zauważyc, 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));
```

Jak pamiętamy, 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 Checkbox „wpisywały się” następnie w kolejne komórki tej tabeli, dzięki czemu bez bezpośredniego określania położenia elementów uzyskaliśmy ich równomierny rozkład w oknie aplikacji.

Oprócz GridLayout można również zastosować wiele innych rozkładów, jak np.: BoxLayout, CardLayout, FlowLayout, GridBagLayout. Ich opisy można znaleźć w dokumentacji JDK. Są to klasy dziedziczące bezpośrednio z Object i implementujące interfejs LayoutManager.

Klasa List

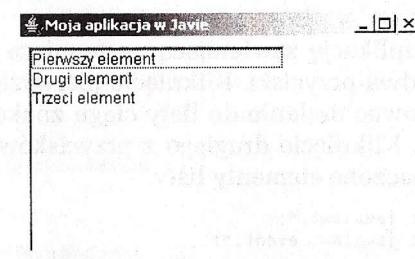
Klasa List pozwala na dodanie do obszaru apletu listy, czyli elementu graficznego zaprezentowanego na rysunku 10.6. Obiekt może zostać utworzony za pomocą bezargumentowego konstruktora (dostępne są także inne konstruktory, których opis można znaleźć w dokumentacji JDK) — należy go dodać do apletu w sposób analogiczny jak w przypadku wcześniej omawianych komponentów. Poszczególne elementy dodajemy do listy, wywołując jej metodę add.

WICZENIE

10.9. Aplikacja z komponentem typu List

Dodaj do okna aplikacji element typu List z kilkoma przykładowymi pozycjami (rysunek 10.6).

Rysunek 10.6.
Okno aplikacji z komponentem typu List



```
import java.awt.*;
import java.awt.event.*;

public class HelloApp extends Frame implements WindowListener
{
    List list;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        list = new List();
        list.setBounds(10, 30, 200, 160);
        list.add("Pierwszy element");
        list.add("Drugi element");
        list.add("Trzeci element");

        add(list);
        setVisible(true);
        list.setVisible(true);
    }
    public static void main(String args[])
    {
        new HelloApp();
    }
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
    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){}
}
```

ĆWICZENIE

10.10. Dodawanie i usuwanie elementów listy

Napisz aplikację zawierającą następujące elementy: listę, pole tekstowe i dwa przyciski. Kliknięcie pierwszego z przycisków powinno spowodować dodanie do listy ciągu znaków, podanego w polu tekstowym. Kliknięcie drugiego z przycisków powinno usuwać aktualnie zaznaczone elementy listy.

```
import java.awt.*;
import java.awt.event.*;

public class HelloApp extends Frame implements WindowListener, ActionListener
{
    List list;
    Button buttonAdd, buttonRemove;
    TextField textField;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(340, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        list = new List(20, true);
        list.setBounds(10, 30, 200, 150);

        textField = new TextField();
        textField.setBounds (220, 30, 110, 20);

        buttonAdd = new Button("Dodaj");
        buttonAdd.setBounds(220, 60, 50, 20);
        buttonAdd.addActionListener(this);

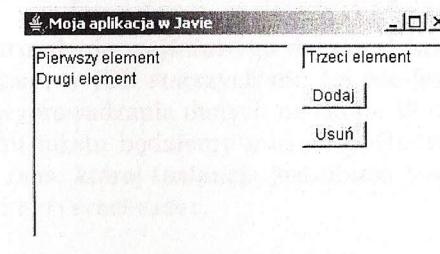
        buttonRemove = new Button("Usuń");
        buttonRemove.setBounds(220, 90, 50, 20);
        buttonRemove.addActionListener(this);

        add(list);
        add(buttonAdd);
        add(buttonRemove);
        add(textField);
        setVisible(true);
        list.setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        String command = e.getActionCommand();
```

```
if (command.equals("Dodaj")){
    String item = textField.getText();
    if (!item.equals(""))
        list.add(item);
}
else if (command.equals("Usuń")){
    String tab[] = list.getSelectedItems();
    for(int i = 0; i < tab.length; i++){
        list.remove(tab[i]);
    }
}
public static void main(String args[])
{
    new HelloApp();
}
public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
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){}
}
```

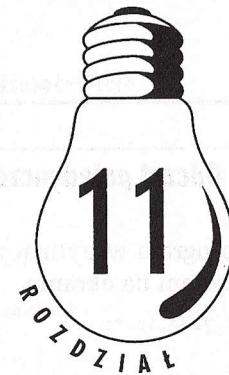
Wygląd aplikacji obrazuje rysunek 10.7. W celu dodania kolejnych elementów używamy metody `add` klasy `List`. Konstruktor klasy jest w tym przykładzie dwuparametry. Pierwszy parametr, typu `int`, podaje rozmiar listy, drugi specyfikuje, czy ma być to lista jedno-, czy wielokrotnego wyboru. Podczas usuwania elementów korzystamy z tablicy zawierającej wszystkie zaznaczone elementy, a dokładniej nazwy tych elementów. Jest ona zwracana przez metodę `getSelectedItems`. Samo usuwanie odbywa się za pomocą metody `remove`, której — jako parametr — podajemy nazwę elementu do usunięcia.

Rysunek 10.7.
Wygląd aplikacji
z ćwiczenia 10.10



Warto zauważyć, że ta implementacja zawiera pewien błąd koncepcyjny. Objawi się on w sytuacji, gdy dodamy kilka elementów listy

o takich samych nazwach, a następnie będziemy próbować je usuwać. Co się stanie na przykład po dodaniu pięciu elementów o nazwie „Java”, zaznaczeniu dwóch ostatnich i kliknięciu przycisku *Usuń*? Otóż zostaną usunięte dwa *pierwsze* elementy o nazwie „Java”. Podkreślam — zaznaczyliśmy *dwa ostatnie*, a zostaną usunięte *dwa pierwsze*! Dzieje się tak, dlatego że elementy są identyfikowane po ich nazwach, a nie indeksach. Rozwiązańiem jest albo uniemożliwienie dodawania elementów o takiej samej nazwie, albo zmiana procedury usuwania. To jednak pozostawiam jako „pracę domową” do samodzielnego wykonania.



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 (strumienie wejściowe) lub je do nich zapisywać (strumienie wyjściowe). Standardowo zdefiniowane mamy 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 w początkowych rozdziałach. Używaliśmy wtedy jednej z jego metod — `println` — do wyświetlenia linii znaków na ekranie.

Wczytywanie danych z klawiatury

Spróbujmy teraz użyć strumienia wejściowego do wczytywania danych z klawiatury. Niestety, w JDK starszych niż 1.5 nie jest to tak proste jak w przypadku wyprowadzania danych na ekran. W celu prawidłowego wczytania linii tekstu będziemy musieli posłużyć się aż trzema klasami: `InputStream`, której instancją jest obiekt `System.in`, oraz `InputStreamReader` i `BufferedReader`.

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

ĆWICZENIE

11.1. Odczyt pojedynczego wiersza tekstu

Napisz program wczytujący linię tekstu z klawiatury i wyświetlający ją 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);
        try{
            String line = inbr.readLine();
            System.out.println(line);
        }
        catch(IOException e){
            System.out.println("Błąd odczytu.");
        }
    }
}
```

Tworzymy tu 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 już korzystać z metody `readLine` obiektu `inbr` (klasy `BufferedReader`), która zwraca nam odczytaną ze strumienia wejściowego linię tekstu.



Uwaga: w JDK istnieje również klasa `DataInputStream` zawierająca metodę `readLine`, która w założeniach wykonuje takie samo zadanie. Nie należy jednak tej metody stosować, gdyż może ona niepoprawnie wykonywać konwersję bajtów ze strumienia na znaki.

ĆWICZENIE

11.2. Wczytywanie danych w pętli

Napisz program, który w pętli odczytuje kolejne linie tekstu i wprowadza 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));
        try{
            while (!line.equals("quit")){
                line = inbr.readLine();
                System.out.println(line);
            }
        }
        catch(IOException e){
            System.out.println("Błąd odczytu.");
        }
    }
}
```

Taki program będzie działał poprawnie, jednak jeśli strumień wejściowy zostanie niespodziewanie przerwany, np. przez wciśnięcie kombinacji klawiszy `Ctrl+C`, wygeneruje wyjątek `NullPointerException`. Dzieje się tak, dlatego że przy przerwaniu strumienia metoda `readLine` zwraca zamiast ciągu znaków wartość `null`. Jako zadanie do samodzielnego wykonania proponuję zastanowić się, jak zapobiegać tego typu sytuacji. Jedno z możliwych rozwiązań zostanie przedstawione w ćwiczeniu 11.7.

Umiemy już wczytywać ciągi znaków, pozostało nauczyć się, w jaki sposób wczytywać liczby. Można zrobić to, wczytując linię tekstu jak w poprzednich przykładach, a następnie dokonać konwersji tekstu na liczbę. Zamiast jednak wykonywać taką konwersję ręcznie, lepiej skorzystać z klasy `Scanner`. Dzieli ona strumień wejściowy na tokeny, czyli jednostki leksykalne. Dla nas jednak najważniejsza w tej chwili jest umiejętności rozpoznawania liczb.

ĆWICZENIE

11.3. Wczytywanie wartości liczbowych

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

```
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.nextToken();

        b = inp.nval;

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

```

Metoda `nextToken` pobiera kolejną jednostkę leksykalną. Jeśli jest to liczba, pole `nval` (zdefiniowane w klasie `StreamTokenizer`) obiektu `inp` zawiera jej wartość. Wystarczy więc wyświetlić ją na ekranie. Niestety, występują tu 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 konwersja typów. Drugi problem jest poważniejszy. Co się bowiem stanie, jeśli nie podamy liczby, ale dowolny inny ciąg znaków? Oczywiście, program nie będzie działał poprawnie. Co prawda, nie wystąpi żaden krytyczny błąd, ale pole `nval` będzie zawierało wartość `0.0`. Może to spowodować trudną do wykrycia usterkę w dalszej części aplikacji. Ustrzeże nas przed tym sprawdzanie stanu statycznego pola `TT_NUMBER` klasy `StreamTokenizer`, które wskazuje, czy ostatnio pobrany token jest liczbą.

ĆWICZENIE

11.4. Wykrywanie rodzaju wprowadzonej wartości

Napisz program wczytujący z klawiatury dwie liczby całkowite, a następnie wyświetlający je na ekranie. W przypadku gdyby użytkownik nie podał liczby, ponawiaj 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("Nie podałeś poprawnej liczby.\nPodaj liczbę a:");
            }
            a = (int) inp.nval;

            System.out.println("Podaj liczbę b:");
            while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
                System.out.println("Nie podałeś poprawnej liczby.\nPodaj liczbę b:");
            }

            b = (int) inp.nval;
            System.out.println("Podajes 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 wywoływaną w pętli typu `while` i dzieje się to dopóty, dopóki pole `TT_NUMBER` klasy `StreamTokenizer` jest różne od `true`. Oznacza to, że pętla działa tak długo, aż użytkownik wprowadzi prawidłową liczbę. W sytuacji, kiedy nie została wprowadzona prawidłowa wartość, na ekranie wyświetlany jest odpowiedni komunikat. Jeśli wprowadzony ciąg jest liczbą, następuje zakończenie pętli, a wartość jest konwertowana do typu `int` i przypisywana właściwej zmiennej.

ĆWICZENIE

11.5. 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.println("Podaj parametr A:");
        while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
            System.out.println("Nie podałeś poprawnej liczby.\nPodaj");
            System.out.print("parametr B:");
        }
        parametrA = inp.nval;

        System.out.println("Podaj parametr B:");
        while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
            System.out.println("Nie podałeś poprawnej liczby.\nPodaj");
            System.out.print("parametr B:");
        }
        parametrB = inp.nval;

        System.out.println("Podaj parametr C:");
        while(inp.nextToken() != StreamTokenizer.TT_NUMBER){
            System.out.println("Nie podałeś poprawnej liczby.\nPodaj");
            System.out.print("parametr C:");
        }
        parametrC = inp.nval;
    }
    catch (IOException e){
        System.out.println("Błąd odczytu!");
        System.exit(-1);
    }

    System.out.println ("Parametry równania:\n");
    System.out.println ("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 rozwiązań w zbiorze");
            System.out.println ("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 w sposób analogiczny do przypadku z ćwiczenia 2.17 w rozdziale 2. Nowością jest wprowadzanie parametrów równania z klawiatury, podczas gdy poprzednio były one na stałe zapisane w kodzie. Odczyt danych z klawiatury odbywa się jednak na tej samej zasadzie jak w przypadku ćwiczenia 11.4. Cała aplikacja jest więc swoistym połączeniem koncepcji z przykładów 2.17 i 11.4.

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

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

W tak prostej postaci spowoduje ona jednak, że wprowadzenie ciągu, który nie reprezentuje liczby całkowitej, spowoduje wygenerowanie wyjątku. 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, i `false` w przeciwnym przypadku, natomiast druga powoduje pominięcie kolejnego tokena.

C W I C Z E N I E

11.6. Wykorzystanie klasy Scanner

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

```
import java.util.*;  
  
public  
class Main  
{  
    public static void main(String args[])  
    {
```

```

Scanner sc = new Scanner(System.in);
System.out.println("Podaj liczbę całkowitą:");
while(!sc.hasNextInt()){
    System.out.print("Nie podałeś prawidłowej liczby całkowitej. ");
    System.out.println("Podaj liczbę całkowitą:");
    sc.next();
}
int i = sc.nextInt();
System.out.println("Wprowadzona liczba to: " + i);
}

```

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 klasę `File` będącą opisem pliku oraz klasy `InputStream` i `OutputStream` służące do strumieniowych operacji na plikach. Przydatne będą również: `BufferedReader`, `BufferedWriter`, `DataInputStream` i `DataOutputStream`.

ĆWICZENIE

11.7. 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;
        File file = new File("test.txt");
        try{
            fout = new FileOutputStream(file);
        }
        catch(FileNotFoundException e){
            System.out.println("Błąd operacji na pliku.");
            System.exit(-1);
        }
        DataOutputStream out = new DataOutputStream(fout);

```

```

        BufferedReader inbr = new BufferedReader(new InputStreamReader(System.in));
        try{
            while (!line.equals("quit")){
                if((line = inbr.readLine()) == null)
                    break;
                out.writeBytes(line + '\n');
            }
        }
        catch(IOException e){
            System.out.println("Read/Write error.");
        }
    }
}

```

Dane z klawiatury odczytujemy, wykorzystując obiekt `inbr` klasy `BufferedReader`, tak jak robiliśmy to we wcześniejszych przykładach. Do zapisu wykorzystujemy natomiast metodę `writeBytes` zdefiniowaną w klasie `DataOutputStream`. Aby utworzyć obiekt klasy `DataOutputStream`, w konstruktorze podajemy obiekt klasy `FileOutputStream` powiązany z plikiem `test.txt` przez obiekt `file` klasy `File`.

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, kiedy strumień wejściowy zostanie przerwany, np. przez wcisnięcie klawiszy `Ctrl+C`. Jeśli tego warunku nie zastosujemy, przerwanie strumienia będzie skutkowało powstaniem wyjątku `NullPointerException`, gdyż pod `line` zostanie podstawione `null` i nastąpi próba wywołania metody `equals` w stosunku do `null`.

Alternatywne rozwiążanie to zastosowanie instrukcji `try...catch` przechwytyjącej wyjątek `NullPointerException` lub też przebudowanie pętli `while`, która mogłaby mieć postać:

```

while (!"quit".equals(line)){
    if((line = inbr.readLine()) != null)
        out.writeBytes(line + '\n');
}

```

ĆWICZENIE

11.8. Wczytywanie 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;
    File file = new File("test.txt");
    try{
        fin = new FileInputStream(file);
    }
    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("Read/Write error.");
    }
}
}

```

W powyższym ćwiczeniu stosujemy podobne konstrukcje jak w poprzednich przykładach. Zwróćmy jedynie uwagę na postać wyrażenia znajdującego się w pętli while:

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

Oznacza ona: pobierz kolejne linie tekstu i wykonuj pętlę dopóty, dopóki line jest różne od null. Ten warunek jest potrzebny do określenia, kiedy zostanie osiągnięty koniec pliku. W takim bowiem przypadku metoda readLine zwraca właśnie wartość null.

Nasza aplikacja w tej postaci 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 tu z argumentów funkcji main. Dokładniej argument jest jeden:

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

Jest to tablica zawierająca referencje do obiektów typu String. Okazuje się, że zawiera ona wszystkie parametry podane w wierszu poleceń podczas uruchamiania programu.

ĆWICZENIE

11.9. Parametry wywołania aplikacji

Napisz aplikację wyświetlającą jej wszystkie parametry wywołania.

```

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

```

Warto zauważyć, że — odmiennie niż w C/C++ — pierwszym elementem tablicy nie jest nazwa samego programu!

Skoro wiemy już, jak odczytać argumenty aplikacji, możemy napisać program wyświetlający zawartość pliku, którego nazwę będziemy podawać w linii wywołania, pisząc:

```
java Main nazwa_pliku
```

ĆWICZENIE

11.10. Wykorzystanie argumentów wywołania aplikacji

Napisz program wyświetlający zawartość pliku tekstowego podanego 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;
        File file = new File(args[0]);
        try{
            fin = new FileInputStream(file);
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku " + args[0]);
            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("Read/Write error.");
    }
}

```

ĆWICZENIE

11.11. Kopiowanie plików

Napisz program kopiący 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 nazwa_
        pliku");
            System.exit(0);
        }
        FileInputStream fin = null;
        FileOutputStream fout = null;
        File fileIn = new File(args[0]);
        File fileOut = new File(args[1]);
        try{
            fin = new FileInputStream(fileIn);
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku: " + args[0]);
            System.exit(-1);
        }
        try{
            fout = new FileOutputStream(fileOut);
        }
        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.");
    }
}

```

Dane są odczytywane z pliku źródłowego za pomocą metody `read` obiektu `fin` (klasy `FileInputStream`), natomiast zapisywane za pomocą metody `write` obiektu `fout` (klasy `FileOutputStream`). Operacje te są wykonywane w pętli `while`, która działa tak długo, aż metoda `read` zwróci wartość `-1` oznaczającą osiągnięcie końca pliku. Należy jednak zwrócić uwagę na to, że zastosowana w ćwiczeniu metoda kopiowania bajt po bajcie jest mało efektywna. O wiele lepsze rezultaty dałoby kopiowanie większych bloków danych. To jednak pozostawiam jako zadanie do samodzielnego wykonania. Proszę zatrzymać się na dokumentacji JDK i sprawdzić, jaką metodą pozwala na kopiowanie więcej niż jednego bajtu. Pozwoli to na napisanie dużo szybciej działającego programu.

Wiemy już, jak przekazać do aplikacji konsolowej nazwę pliku. Co jednak zrobić w przypadku aplikacji graficznej, gdzie wyboru nazwy pliku chcielibyśmy dokonywać w dowolnym momencie działania programu? W takiej sytuacji powinniśmy wyświetlić użytkownikowi okno dialogowe służące do wyboru pliku. Takie zadanie pozwoli nam wykonać klasę `FileDialog`.

ĆWICZENIE

11.12. Wyświetlenie okna wyboru plików

Napisz graficzną aplikację, która po kliknięciu przycisku wyświetli okno służące do wyboru pliku.

```

import java.awt.*;
import java.awt.event.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    Button button;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 200);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);
    }
}

```

```

button = new Button("OK");
button.setBounds(135, 90, 50, 20);
button.setActionCommand("cmdOK");
button.addActionListener(this);

add(button);
setVisible(true);
button.setVisible(true);
}
public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    if (command.equals("cmdOK")){
        FileDialog fd = new FileDialog(this);
        fd.setVisible(true);
    }
}
public static void main(String args[])
{
    new HelloApp();
}
public void windowClosing(WindowEvent e)
{
    System.exit(0);
}
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){}
}

```

Sposób obsługi zdarzeń i okna aplikacji jest taki sam, jak w przypadku przykładów z rozdziału 9. W celu wyświetlenia na ekranie okna dialogowego służącego do wyboru plików konstruujemy obiekt klasy `FileDialog`, a następnie wywołujemy jego metodę `setVisible`, przekazując jej w postaci parametru wartość `true`. Analogicznie, jeśli tej metodzie przekażemy wartość `false`, okno (o ile było widoczne na ekranie) zostanie schowane. W celu wyświetlenia bądź schowania okna nie należy natomiast wykorzystywać dostępnych w klasie metod `show` i `hide`, gdyż od JDK w wersji 1.5 są one uznawane za przestarzałe.

Skoro wiemy już, jak wyświetlić okno służące do wyboru plików, dobrze byłoby wykorzystać je w bardziej praktyczny sposób. Przypomnijmy sobie ćwiczenia z rozdziału 10. Umieściliśmy wtedy w oknie aplikacji m.in. element `TextArea`, który umożliwiał wprowadzenie przez użytkownika dłuższego tekstu. A gdyby tak umożliwić również zapis i odczyt tego tekstu? Byłaby to już „prawdziwa” aplikacja. Taki mały edytor tekstu!

ĆWICZENIE

11.13. Zapis i odczyt tekstu

Napisz graficzną aplikację zawierającą element typu `TextArea`. Dodaj menu oraz procedury umożliwiające zapis i odczyt wprowadzonego przez użytkownika tekstu.

```

import java.awt.*;
import java.awt.event.*;
import java.io.*;

public
class HelloApp extends Frame implements WindowListener, ActionListener
{
    TextArea textArea;
    public HelloApp ()
    {
        super();
        addWindowListener(this);
        setSize(320, 240);
        setTitle("Moja aplikacja w Javie");
        setLayout(null);

        textArea = new TextArea();
        textArea.setBounds(10, 50, 300, 160);
        this.add(textArea);

        MenuBar menuBar = new MenuBar();
        setMenuBar(menuBar);
        Menu menu = new Menu("Plik");
        menu.add(new MenuItem("Otwórz"));
        menu.add(new MenuItem("Zapisz"));
        menu.add(new MenuItem("-"));
        menu.add(new MenuItem("Zamknij"));
        menuBar.add(menu);

        menu.addActionListener(this);
        setVisible(true);
        textArea.setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        String command = e.getActionCommand();
        if (command.equals("Zamknij")){
            System.exit(0);
        }
        else if(command.equals("Otwórz")){
            String fileName = showDialog(FileDialog.LOAD);
            if (fileName != null){
                load(fileName);
            }
        }
        else if (command.equals("Zapisz")){
            String fileName = showDialog(FileDialog.SAVE);
            if (fileName != null){
                save(fileName);
            }
        }
    }
}

```

```

        }
    }

    public String showDialog(int mode)
    {
        FileDialog fd = new FileDialog(this);
        fd.setMode(mode);
        fd.setVisible(true);
        String fileDir = fd.getDirectory();
        String fileName = fd.getFile();
        if(fileDir != null & fileName != null){
            return fileDir + fileName;
        }
        else{
            return null;
        }
    }

    public void load(String fileName)
    {
        String line = "";
        FileInputStream fin = null;
        File file = new File(fileName);
        try{
            fin = new FileInputStream(file);
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku " + fileName);
            return;
        }
        DataInputStream out = new DataInputStream(fin);
        BufferedReader inbr = new BufferedReader(new InputStreamReader(fin));
        textArea.setText("");
        try{
            while ((line = inbr.readLine()) != null){
                textArea.append(line + '\n');
            }
        }
        catch(IOException e){
            System.out.println("Read/Write error.");
        }
    }

    public void save(String fileName)
    {
        FileOutputStream fout = null;
        File file = new File(fileName);
        try{
            fout = new FileOutputStream(file);
        }
        catch(FileNotFoundException e){
            System.out.println("Brak pliku " + fileName);
            return;
        }
        DataOutputStream out = new DataOutputStream(fout);
        try{
            String line = textArea.getText();
            out.writeBytes(line + '\n');
        }
        catch(IOException e){
            System.out.println("Read/Write error.");
        }
    }
}

```

```

        }
    }

    public static void main(String args[])
    {
        new HelloApp();
    }

    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }

    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){}
}

```

Aplikacja łączy w sobie wiele elementów, z których korzystaliśmy już wcześniej, jak menu, obsługa zdarzeń i operacje na plikach. Główne zadania są wykonywane przez trzy funkcje (wywoływanie z metody actionPerformed, w zależności od wybranego menu):

- showDialog — wyświetla okno dialogowe służące do wyboru pliku,
- save — zapisuje dane w wybranym pliku,
- load — wczytuje dane z wybranego pliku.

Funkcja showDialog wyświetla okno dialogowe, które jest tworzone podobnie jak w poprzednim przykładzie. Różni się tylko wykorzystaniem metody setMode klasy FileDialog, która pozwala na ustalenie trybu (wpływa on na wygląd i zachowanie okna). Możliwe wartości to:

- FileDialog.SAVE — dla okna służącego do wyboru pliku do zapisu,
- FileDialog.LOAD — dla okna służącego do wyboru pliku do odczytu.

Wartością zwracaną przez funkcję jest ciąg znaków zawierający nazwę wybranego przez użytkownika pliku wraz z pełną ścieżką dostępu. Dane te zostały pobrane przez wywołanie metod: getFile i getDirectory.

