Programowanie obiektowe – Ćwiczenia 2

Podstawowe elementy języka Java

Poniższe przykłady są z książki

Cay Horstmann, Gary Cornell, Java. Podstawy, Wydawnictwo Helion, Wyd. 9, Gliwice 2013.

Napisanie w Javie programu z graficznym interfejsem użytkownika nie jest łatwe —wymaga dużej wiedzy na temat sposobów tworzenia okien, dodawania do nich pól tekstowych, przycisków, które reagują na zawartość tych pól itd. Jako że opis technik pisania programów GUI w Javie znacznie wykracza poza nasz cel przedstawienia podstaw programowania w tym języku, przykładowe programy w tym rozdziale są bardzo proste. Komunikują się za pośrednictwem okna konsoli, a ich przeznaczeniem jest tylko ilustracja omawianych pojęć.

Prosty program w Javie

Przyjrzyjmy się uważnie najprostszemu programowi w Javie, jaki można napisać — takiemu, który tylko wyświetla komunikat w oknie konsoli:

```
public class FirstSample
{
public static void main(String[] args)
{
System.out.println("Nie powiemy „Witaj, świecie!”");
}
}
```

Warto poświęcić trochę czasu i nauczyć się tego fragmentu na pamięć, ponieważ wszystkie aplikacje są oparte na tym schemacie. Przede wszystkim w Javie **wielkość liter ma znaczenie**. Jeśli w programie będzie literówka (jak np. słowo Main zamiast main), to program nie zadziała. Przestudiujemy powyższy kod wiersz po wierszu. Słowo kluczowe public nosi nazwę **modyfikatora dostępu** (ang. *access modifier*). Określa ono, jaki rodzaj dostępu do tego kodu mają inne części programu. Słowo kluczowe class przypomina, że wszystko w Javie należy do jakiejś klasy.

Ponieważ klasami bardziej szczegółowo zajmujemy się w kolejnym rozdziale, na razie będziemy je traktować jako zbiory mechanizmów programu, które są odpowiedzialne za jego działanie. Klasy to bloki, z których składają się wszystkie aplikacje i aplety Javy. **Wszystko** w programie w Javie musi się znajdować wewnątrz jakiejś klasy.

Po słowie kluczowym class znajduje się nazwa klasy. Reguły dotyczące tworzenia nazw klas w Javie są dosyć liberalne. Nazwa klasy musi się zaczynać od litery, po której może

znajdować się kombinacja dowolnych znaków i cyfr. Nie ma w zasadzie ograniczeń, jeśli chodzi o długość. Nie można stosować słów zarezerwowanych Javy (np. public lub class) — lista wszystkich słów zarezerwowanych znajduje się w dodatku.
Zgodnie ze standardową konwencją nazewniczą (której przykładem jest nazwa klasy FirstSample) nazwy klas powinny się składać z rzeczowników pisanych wielką literą. Jeśli nazwa klasy składa się z kilku słów, każde z nich powinno być napisane wielką literą; notacja polegająca na stosowaniu wielkich liter wewnątrz nazw jest czasami nazywana notacją wielbłądzią
(ang. *camel case* lub *CamelCase*).
Plik zawierający kod źródłowy musi mieć taką samą nazwę jak klasa publiczna oraz rozszerzenie *.java*. W związku z tym nasz przykładowy kod powinien zostać zapisany w pliku o nazwie *FirstSample.java* (przypominam, że wielkość liter ma znaczenie także tutaj — nie można napisać *firstsample.java)*

Jeśli plik ma prawidłową nazwę i nie ma żadnych literówek w kodzie źródłowym, w wyniku jego kompilacji powstanie plik zawierający kod bajtowy tej klasy. Kompilator automatycznie nada skompilowanemu plikowi nazwę *FirstSample.class* i zapisze go w tym samym katalogu, w którym znajduje się plik źródłowy. Program uruchamiamy za pomocą następującego polecenia (nie zapomnij o pominięciu rozszerzenia *.class*):

```
java FirstSample
```

Po uruchomieniu program ten wyświetla w konsoli łańcuch *Nie powiemy „Witaj, świecie!”*.
Polecenie:

```
java NazwaKlasy
```

zastosowane do skompilowanego programu powoduje, że wirtualna maszyna Javy zaczyna wykonywanie od kodu zawartego w metodzie main wskazanej klasy (terminem „metoda” określa się to, co w innych językach jest funkcją). W związku z tym metoda main **musi** się znajdować w pliku źródłowym klasy, którą chcemy uruchomić. Można oczywiście dodać własne metody do klasy i wywoływać je w metodzie main. Pisanie metod omawiamy w następnym rozdziale.

NB. Zgodnie ze specyfikacją języka Java (oficjalnym dokumentem opisującym ten język, który można pobrać lub przeglądać na stronie *http://docs.oracle.com/javase/specs*) metoda main musi być publiczna (public).
Jednak niektóre wersje maszyny wirtualnej Javy uruchamiały programy w Javie, których metoda main nie była publiczna. Pewien programista zgłosił ten błąd. Aby się o tym przekonać, wejdź na stronę *http://bugs.sun.com/bugdatabase/index.jsp* i wpisz numer identyfikacyjny błędu 4252539. Błąd ten został oznaczony jako zamknięty i nie do naprawy (ang. *closed, will not be fixed*). Jeden z inżynierów pracujących w firmie Sun wyjaśnił (*http://docs.oracle.com/javase/specs/jvms/se7/html*), że specyfikacja maszyny wirtualnej Javy nie wymaga, aby metoda main była publiczna, w związku z czym „naprawienie tego błędu może spowodować problemy”. Na szczęście sięgnięto po rozum do głowy i od wersji 1.4 Java SE metoda main jest publiczna.

Ta historia pozwala zwrócić uwagę na kilka rzeczy. Z jednej strony rozczarowuje nas sytuacja, że

osoby odpowiadające za jakość są przepracowane i nie zawsze dysponują wystarczającą wiedzą specjalistyczną z zakresu najbardziej zaawansowanych zagadnień związanych z Javą.
Przez to nie zawsze podejmują trafne decyzje. Z drugiej strony trzeba zauważyć, że firma Sun zamieszcza raporty o błędach na stronie internetowej, aby każdy mógł je zweryfikować. Taki spis błędów jest bardzo wartościowym źródłem wiedzy dla programistów. Można nawet głosować na swój ulubiony błąd. Błędy o największej liczbie głosów mają największą szansę poprawienia w kolejnej wersji pakietu JDK.
Zauważ, że w kodzie źródłowym użyto nawiasów klamrowych. W Javie, podobnie jak w C i C++, klamry oddzielają poszczególne części (zazwyczaj nazywane **blokami**) kodu programu. Kod każdej metody w Javie musi się zaczynać od otwierającej klamry {, a kończyć zamykającą klamrą }.

Styl stosowania nawiasów klamrowych wywołał niepotrzebną dyskusję. My stosujemy styl polegający na umieszczaniu dopełniających się klamer w tej samej kolumnie. Jako że kompilator ignoruje białe znaki, można stosować dowolny styl nawiasów klamrowych. Więcej do powiedzenia na temat stosowania klamer będziemy mieli przy okazji omawiania pętli.
Na razie nie będziemy się zajmować znaczeniem słów static void — traktuj je jako coś, czego potrzebujesz do kompilacji programu w Javie. Po rozdziale czwartym przestanie to być tajemnicą.
Teraz trzeba tylko zapamiętać, że każdy program napisany w Javie musi zawierać metodę main zadeklarowaną w następujący sposób:

```
public class NazwaKlasy
{
public static void main(String[] args)
{
instrukcje programu
}
}
```

NB. Programiści języka C++ doskonale znają pojęcie „klasa”. Klasy w Javie są pod wieloma względami podobne do tych w C++, ale jest też kilka różnic, o których nie można zapominać. Na przykład w Javie **wszystkie** funkcje są metodami jakiejś klasy (w standardowej terminologii są one nazywane metodami, a nie funkcjami składowymi). W związku z tym w Javie konieczna jest obecność klasy zawierającej metodę main. Programiści C++ pewnie znają też **statyczne funkcje składowe**. Są to funkcje zdefiniowane wewnątrz klasy, które nie wykonują żadnych działań na obiektach. Metoda main w Javie jest zawsze statyczna. W końcu słowo kluczowe void, podobnie jak w C i C++, oznacza, że metoda nie zwraca wartości. W przeciwieństwie do języka C i C++ metoda main w Javie nie zwraca żadnego kodu wyjścia (ang. *exit code*) do systemu operacyjnego. Jeśli metoda main zakończy działanie w normalny sposób, program ma kod wyjścia 0, który oznacza pomyślne zakończenie. Aby zakończyć działanie programu innym kodem wyjścia, należy użyć metody System.exit.

Teraz kierujemy naszą uwagę na poniższy fragment:

```
{
System.out.println("Nie powiemy „Witaj, świecie!”");
}
```

Klamry oznaczają początek i koniec **ciała** metody. Ta metoda zawiera tylko jedną instrukcję. Podobnie jak w większości języków programowania, instrukcje Javy można traktować jako zdania tego języka. Każda instrukcja musi być zakończona średnikiem. Przede wszystkim

należy pamiętać, że znak powrotu karetki nie oznacza końca instrukcji, dzięki czemu mogą one obejmować nawet kilka wierszy.
W treści metody main znajduje się instrukcja wysyłająca jeden wiersz tekstu do konsoli. W tym przypadku użyliśmy obiektu System.out i wywołaliśmy na jego rzecz metodę println. Zwróć uwagę na kropki zastosowane w wywołaniu metody. Ogólna składnia stosowana w Javie do wywołania jej odpowiedników funkcji jest następująca:

```
obiekt.metoda(parametry)
```

W tym przypadku wywołaliśmy metodę println i przekazaliśmy jej argument w postaci łańcucha. Metoda ta wyświetla zawartość parametru w konsoli. Następnie kończy wiersz wyjściowy, dzięki czemu każde wywołanie metody println wyświetla dane w oddzielnym wierszu.
Zwróć uwagę, że w Javie, podobnie jak w C i C++, łańcuchy należy ujmować w cudzysłowy — więcej informacji na temat łańcuchów znajduje się w dalszej części tego rozdziału.
Metody w Javie, podobnie jak funkcje w innych językach programowania, przyjmują zero, jeden lub więcej **parametrów** (często nazywanych **argumentami**). Nawet jeśli metoda nie przyjmuje żadnych parametrów, nie można pominąć stojących po jej nazwie nawiasów. Na przykład metoda println bez żadnych argumentów drukuje pusty wiersz. Wywołuje się ją następująco:

```
System.out.println();
```

NB. Na rzecz obiektu System.out można także wywoływać metodę print, która nie dodaje do danych wyjściowych znaku nowego wiersza. Na przykład wywołanie System.out.print("Witaj") drukuje napis *Witaj* bez znaku nowego wiersza. Kolejne dane zostaną umieszczone bezpośrednio po słowie *Witaj*.

Komentarze

Komentarze w Javie, podobnie jak w większości języków programowania, nie są uwzględniane w programie wykonywalnym. Można zatem stosować je w dowolnej ilości bez obawy, że nadmiernie zwiększą rozmiary kodu. W Javie są trzy rodzaje komentarzy. Najczęściej stosowana metoda polega na użyciu znaków //. Ten rodzaj komentarza obejmuje obszar od znaków // do końca wiersza, w którym się znajdują.

```
System.out.println("Nie powiemy „Witaj, świecie!”"); // Czy to nie słodkie?
```

Dłuższe komentarze można tworzyć poprzez zastosowanie znaków // w wielu wierszach lub użycie komentarza w stylu /* */. W ten sposób w komentarzu można ująć cały blok treści programu.
Wreszcie, trzeci rodzaj komentarza służy do automatycznego generowania dokumentacji. Ten rodzaj komentarza zaczyna się znakami /** i kończy */. Jego zastosowanie przedstawia listing:

FirstSample.java

```
/**
* Jest to pierwszy przykładowy program w rozdziale 3.
* @version 1.01 1997-03-22
* @author Gary Cornell
*/
public class FirstSample
```

```
{
public static void main(String[] args)
{
System.out.println("Nie powiemy „Witaj, świecie!”");
}
}
```

NB.Komentarzy /* */ nie można zagnieżdżać. Oznacza to, że nie można dezaktywować fragmentu kodu programu, otaczając go po prostu znakami /* i */, ponieważ kod ten może zawierać znaki */.

Typy danych

Java jest językiem o **ścisłej kontroli typów**. Oznacza to, że każda zmienna musi mieć określony typ. W Javie istnieje osiem **podstawowych typów**. Cztery z nich reprezentują liczby całkowite, dwa — liczby rzeczywiste, jeden o nazwie char zarezerwowano dla znaków reprezentowanych przez kody liczbowe należące do systemu Unicode, zaś ostatni jest logiczny (boolean) — przyjmuje on tylko dwie wartości: true albo false.

NB. W Javie dostępny jest pakiet do obliczeń arytmetycznych na liczbach o dużej precyzji. Jednak tak zwane „duże liczby” (ang. *big numbers*) są **obiektami**, a nie nowym typem w Javie. Sposób posługiwania się nimi został opisany w dalszej części tego rozdziału.

Typy całkowite

Typy całkowite to liczby pozbawione części ułamkowej. Zaliczają się do nich także wartości ujemne. Wszystkie cztery dostępne w Javie typy całkowite przedstawia tabela:

Typ	Liczba bajtów	Zakres (z uwzględnieniem wartości brzegowych)
int	4	od –2 147 483 648 do 2 147 483 647 (nieco ponad 2 miliardy)
short	2	od –32 768 do 32 767
long	8	od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
byte	1	od –128 do 127

Do większości zastosowań najlepiej nadaje się typ int. Aby zapisać liczbę mieszkańców naszej planety, trzeba użyć typu long. Typy byte i short są używane do specjalnych zadań, jak niskopoziomowa praca nad plikami lub duże tablice, kiedy pamięć jest na wagę złota. Zakres wartości typów całkowitych nie zależy od urządzenia, na którym uruchamiany jest kod Javy. Eliminuje to główny problem programisty, który chce przenieść swój program z jednej platformy na inną lub nawet z jednego systemu operacyjnego do innego na tej samej platformie. W odróżnieniu od Javy, języki C i C++ używają najbardziej efektywnego typu całkowitego dla każdego procesora. W wyniku tego program prawidłowo działający na procesorze 32-bitowym

może powodować błąd przekroczenia zakresu liczby całkowitej na procesorze 16-bitowym. Jako że programy w Javie muszą działać prawidłowo na wszystkich urządzeniach, zakresy wartości różnych typów są stałe.

Duże liczby całkowite (typu `long`) są opatrzone modyfikatorem `L` lub `l` (na przykład `4000000000L`). Liczby w formacie szesnastkowym mają przedrostek `0x` (na przykład `0xCAFE`). Liczby w formacie ósemkowym poprzedza przedrostek `0`. Na przykład liczba `010` w zapisie ósemkowym to `8` w zapisie dziesiętnym. Oczywiście zapis ten może wprowadzać w błąd, w związku z czym odradzamy jego stosowanie.
W Java 7 wprowadzono dodatkowo możliwość zapisu liczb w formacie binarnym, do czego służy przedrostek `0b`. Przykładowo `0b1001` to inaczej `9`. Ponadto również od tej wersji języka można w literałach liczbowych stosować znaki podkreślenia, np. `1_000_000` (albo `0b1111_0100_0010_0100_0000`) — milion. Znaki te mają za zadanie ułatwić czytanie kodu ludziom. Kompilator Javy je usuwa.

NB. W językach C i C++ typ int to liczba całkowita, której rozmiar zależy od urządzenia docelowego. W procesorach 16-bitowych, jak 8086, typ int zajmuje 2 bajty pamięci. W procesorach 32-bitowych, jak Sun SPARC, są to wartości czterobajtowe.
W przypadku procesorów Intel Pentium rozmiar typu int zależy od systemu operacyjnego: w DOS-ie i Windows 3.1 typ int zajmuje 2 bajty pamięci. W programach dla systemu Windows działających w trybie 32-bitowym typ int zajmuje 4 bajty. W Javie wszystkie typy numeryczne są niezależne od platformy.
Zauważ, że w Javie nie ma typu unsigned.

Typy zmiennoprzecinkowe

Typy zmiennoprzecinkowe służą do przechowywania liczb z częścią ułamkową. Dwa dostępne w Javie typy zmiennoprzecinkowe przedstawia tabela:

Typ	Liczba bajtów	Zakres
`float`	4	około ±3,40282347E+38F (6 – 7 znaczących cyfr dziesiętnych)
`double`	8	około ±1,79769313486231570E+308 (15 znaczących cyfr dziesiętnych)

Nazwa `double` (podwójny) wynika z tego, że typ ten ma dwa razy większą precyzję niż typ `float` (czasami liczby te nazywa się liczbami o **podwójnej precyzji**). W większości przypadków do reprezentacji liczb zmiennoprzecinkowych wybierany jest typ `double`. Ograniczona precyzja typu `float` często okazuje się niewystarczająca. Siedem znaczących (dziesiętnych) cyfr może wystarczyć do precyzyjnego przedstawienia naszej pensji w złotówkach i groszach, ale może być już to za mało precyzyjne do przechowywania liczby określającej zarobki naszego szefa. W związku z tym powodów do stosowania typu `float` jest niewiele; może to być sytuacja, w

której zależy nam na nieznacznym zwiększeniu szybkości poprzez zastosowanie liczb o pojedynczej precyzji lub kiedy chcemy przechowywać bardzo dużą ich ilość.

Liczby typu `float` mają przyrostek `F` lub `f` (na przykład `3.14F`). Liczby zmiennoprzecinkowe pozbawione tego przyrostka (na przykład `3.14`) są zawsze traktowane jako typ `double`. Można też podać przyrostek `D` lub `d` (na przykład `3.14D`).

NB. Liczby zmiennoprzecinkowe można podawać w zapisie szesnastkowym. Na przykład 0,125, czyli 2_{-3}, można zapisać jako 0x1.0p-3. Wykładnik potęgi w zapisie szesnastkowym to p, a nie e (e jest cyfrą szesnastkową). Zauważ, że mantysa jest w notacji szesnastkowej, a wykładnik w dziesiętnej. Podstawą wykładnika jest 2, nie 10.

Wszelkie obliczenia arytmetyczne wykonywane na liczbach zmiennoprzecinkowych są zgodne ze standardem IEEE 754. Istnieją trzy szczególne wartości pozwalające określić liczby, których wartości wykraczają poza dozwolony zakres błędu:
-dodatnia nieskończoność,
- ujemna nieskończoność,
- NaN — nie liczby (ang. *Not a Number*).
Na przykład wynikiem dzielenia dodatniej liczby przez zero jest dodatnia nieskończoność. Działanie dzielenia zero przez zero lub wyciągania pierwiastka kwadratowego z liczby ujemnej daje w wyniku NaN.

NB. Stałe Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY i Double.NaN (oraz ich odpowiedniki typu float) reprezentują wymienione specjalne wartości, ale są rzadko używane. Nie można na przykład wykonać takiego sprawdzenia:
`if (x == Double.NaN)` *// Nigdy nie jest true.*
aby dowiedzieć się, czy dany wynik jest równy stałej Double.NaN. Wszystkie tego typu wartości są różne. Można za to używać metody Double.isNaN:
`if (Double.isNaN(x))` *// Sprawdzenie, czy x jest „nie liczbą”.*

NB. Liczby zmiennoprzecinkowe **nie** nadają się do obliczeń finansowych, w których niedopuszczalny jest błąd zaokrąglania (ang. *roundoff error*). Na przykład instrukcja System.out.println(2.0 - 1.1) da wynik 0.8999999999999999 zamiast spodziewanego 0.9. Tego typu błędy spowodowane są tym, że liczby zmiennoprzecinkowe są reprezentowane w systemie binarnym. W systemie tym nie ma dokładnej reprezentacji ułamka 1/10, podobnie jak w systemie dziesiętnym nie istnieje dokładna reprezentacja ułamka 1/3. Aby wykonywać precyzyjne obliczenia numeryczne bez błędu zaokrąglania, należy użyć klasy BigDecimal, która jest opisana w dalszej części tego rozdziału.

Typ char

Typ `char` służy do reprezentacji pojedynczych znaków. Najczęściej są to stałe znakowe. Na przykład `'A'` jest stałą znakową o wartości `65`. Nie jest tym samym co `"A"` — łańcuchem zawierającym jeden znak. Kody Unicode mogą być wyrażane w notacji szesnastkowej, a ich wartości mieszczą się w zakresie od `\u0000` do `\uFFFF`. Na przykład kod `\u2122` reprezentuje symbol ™, a `\u03C0` to grecka litera Π.
Poza symbolem zastępczym `\u` oznaczającym zapis znaku w kodzie Unicode jest jeszcze kilka innych symboli zastępczych umożliwiających zapisywanie różnych znaków specjalnych. Zestawienie tych znaków przedstawia tabela poniżej. Można je stosować zarówno w stałych

znakowych, jak i w łańcuchach, np. '\u2122' albo "Witaj\n". Symbol zastępczy \u jest jedynym symbolem zastępczym, którego można używać także **poza** cudzysłowami otaczającymi
znaki i łańcuchy. Na przykład zapis:

```
public static void main(String\u005B\u005D args)
```

jest w pełni poprawny — kody \u005B i \u005D oznaczają znaki [i].

Symbol zastępczy	Nazwa	Wartość Unicode
\b	Backspace	\u0008
\t	Tabulacja	\u0009
\n	Przejście do nowego wiersza	\u000a
\r	Powrót karetki	\u 000d
\"	Cudzysłów	\u 0022
\'	Apostrof	\u0027
\\	Lewy ukośnik	\u005c

Aby w pełni zrozumieć typ char, trzeba poznać system kodowania znaków Unicode. Unicode opracowano w celu pozbycia się ograniczeń tradycyjnych systemów kodowania. Przed powstaniem systemu Unicode istniało wiele różnych standardów: ASCII w USA, ISO 8859-1 dla języków krajów Europy Zachodniej, ISO-8859-2 dla języków środkowo- i wschodnioeuropejskich
(w tym polskiego), KOI-8 dla języka rosyjskiego, GB18030 i BIG-5 dla języka chińskiego itd. Powoduje to dwa problemy: jeden kod może oznaczać różne znaki w różnych systemach kodowania, a poza tym kody znaków w językach o dużej liczbie znaków mają różne rozmiary — niektóre często używane znaki zajmują jeden bajt, a inne potrzebują dwóch bajtów.
Unicode ma za zadanie rozwiązać te problemy. Kiedy w latach osiemdziesiątych XX wieku podjęto próby unifikacji, wydawało się, że dwubajtowy stały kod był więcej niż wystarczający do zakodowania znaków używanych we wszystkich językach świata. W 1991 roku światło dzienne ujrzał Unicode 1.0. Wykorzystywana w nim była prawie połowa wszystkich dostępnych 65 536 kodów. Java od samego początku używała znaków 16-bitowego systemu Unicode, co dawało jej dużą przewagę nad innymi językami programowania, które stosowały znaki ośmiobitowe.

Niestety z czasem nastąpiło to, co było nieuchronne. Unicode przekroczył liczbę 65 536 znaków, głównie z powodu dodania bardzo dużych zbiorów ideogramów używanych w językach chińskim, japońskim i koreańskim. Obecnie 16-bitowy typ char nie wystarcza do opisu wszystkich znaków Unicode.
Aby wyjaśnić, jak ten problem został rozwiązany w Javie, zaczynając od Java SE 5.0, musimy wprowadzić nieco nowej terminologii. **Współrzędna kodowa znaku** (ang. *code point*) to wartość związana ze znakiem w systemie kodowania. W standardzie Unicode współrzędne

kodowe znaków są zapisywane w notacji szesnastkowej i są poprzedzane łańcuchem U+, np. współrzędna kodowa litery A to U+0041. Współrzędne kodowe znaków systemu Unicode są pogrupowane w 17 **przestrzeniach numeracyjnych** (ang. *code planes*). Pierwsza z nich, nazywana podstawową przestrzenią wielojęzyczną (ang. *Basic Multilingual Plane — BMP*), zawiera **klasyczne** znaki Unicode o współrzędnych kodowych z przedziału od U+0000 do U+FFFF. Pozostałe szesnaście przestrzeni o współrzędnych kodowych znaków z przedziału od U+10000 do U+10FFFF zawiera **znaki dodatkowe** (ang. *supplementary characters*).
Kodowanie UTF-16 to sposób reprezentacji wszystkich współrzędnych kodowych znaków za pomocą kodów o różnej długości. Znaki w podstawowej przestrzeni są 16-bitowymi wartościami o nazwie **jednostek kodowych** (ang. *code units*). Znaki dodatkowe są kodowane jako kolejne pary jednostek kodowych. Każda z wartości należących do takiej pary należy do zakresu 2048 nieużywanych wartości BMP, zwanych **obszarem surogatów** (ang. *surrogates area*) — zakres pierwszej jednostki kodowej to U+D800 – U+DBFF, a drugiej U+DC00 – U+DFFF. Jest to bardzo sprytne rozwiązanie, ponieważ od razu wiadomo, czy jednostka kodowa reprezentuje jeden znak, czy jest pierwszą lub drugą częścią znaku dodatkowego.
Na przykład matematyczny symbol oznaczający zbiór liczb całkowitych ma współrzędną kodową U+1D56B i jest kodowany przez dwie jednostki kodowe U+D835 oraz U+DD6B (opis algorytmu kodowania UTF-16 można znaleźć na stronie *http://en.wikipedia.org/wiki/UTF-16*).
W Javie typ char opisuje **jednostkę kodową** UTF-16.
Zdecydowanie odradzamy posługiwania się w programach typem char, jeśli nie ma konieczności wykonywania działań na jednostkach kodowych UTF-16. Prawie zawsze lepszym rozwiązaniem jest traktowanie łańcuchów (które opisujemy w podrozdziale 3.6, „Łańcuchy”) jako abstrakcyjnych typów danych.

Typ boolean

Typ boolean (logiczny) może przechowywać dwie wartości: true i false. Służy do sprawdzania warunków logicznych. Wartości logicznych nie można konwertować na wartości całkowitoliczbowe.

Zmienne

W Javie każda zmienna musi mieć określony **typ**. Deklaracja zmiennej polega na napisaniu nazwy typu, a po nim nazwy zmiennej. Oto kilka przykładów deklaracji zmiennych:

NB. W języku C++ zamiast wartości logicznych można stosować liczby, a nawet wskaźniki. Wartość 0 jest odpowiednikiem wartości logicznej false, a wartość różna od zera odpowiada wartości true. W Javie tak **nie** jest. Dzięki temu programiści Javy mają ochronę przed popełnieniem błędu:
if (x = 0) *// ups… miałem na myśli x == 0*

W C++ test ten przejdzie kompilację i będzie można go uruchomić, a jego wartością zawsze będzie false. W Javie testu tego nie będzie można skompilować, ponieważ wyrażenia całkowitoliczbowego x = 0 nie można przekonwertować na wartość logiczną.

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

Należy zauważyć, że na końcu każdej deklaracji znajduje się średnik. Jest on wymagany z tego względu, że deklaracja zmiennej jest instrukcją w Javie.

Nazwa zmiennej musi się zaczynać literą oraz składać się z liter i cyfr. Zwróćmy uwagę, że pojęcia „litera" i „cyfra" w Javie mają znacznie szersze znaczenie niż w większości innych języków. Zgodnie z definicją litera to jeden ze znaków 'A' – 'Z', 'a' – 'z', '_' lub **każdy** znak Unicode będący literą jakiegoś języka. Na przykład polscy programiści mogą w nazwach zmiennych używać liter z ogonkami, takich jak ą. Grek może użyć litery Π. Podobnie cyfry należą do zbioru '0' – '9' oraz są nimi **wszystkie** znaki Unicode, które oznaczają cyfrę w jakimś języku. W nazwach zmiennych nie można stosować symboli typu '+' czy © ani spacji. **Wszystkie** znaki użyte w nazwie zmiennej oraz ich **wielkość mają znaczenie**. Długość nazwy zmiennej jest w zasadzie nieograniczona.

NB. Aby sprawdzić, które znaki Unicode są w Javie literami, można użyć metod isJavaIdentifierStart i isJavaIdentifierPart, które są dostępne w klasie Character.

NB. Mimo że znak $ jest w Javie traktowany jak zwykła litera, nie należy go używać w swoim kodzie. Jest stosowany w nazwach generowanych przez kompilator i inne narzędzia Javy.

Dodatkowo nazwa zmiennej w Javie nie może być taka sama jak słowo zarezerwowane (listę słów zarezerwowanych zawiera dodatek).

Kilka deklaracji można umieścić w jednym wierszu:

```
int i, j; // Obie zmienne są typu int.
```

Nie polecamy jednak takiego stylu pisania kodu. Dzięki deklarowaniu każdej zmiennej oddzielnie programy są łatwiejsze do czytania.

NB. Jak wiemy, w nazwach są rozróżniane małe i wielkie litery. Na przykład nazwy hireday i hireDay to dwie różne nazwy. W zasadzie nie powinno się stosować nazw zmiennych różniących się tylko wielkością liter, chociaż czasami trudno jest wymyślić dobrą nazwę. Wielu programistów w takich przypadkach nadaje zmiennej taką samą nazwę jak nazwa typu:

```
Box box; // Box to nazwa typu, a box to nazwa zmiennej.
```

Inni wolą stosować przedrostek a:

```
Box aBox;
```

Inicjacja zmiennych

Po zadeklarowaniu zmiennej trzeba ją zainicjować za pomocą instrukcji przypisania — nie można użyć wartości niezainicjowanej zmiennej. Na przykład poniższe instrukcje w Javie są błędne:

```
int vacationDays;
System.out.println(vacationDays); // Błąd — zmienna nie została zainicjowana.
```

Przypisanie wartości do zadeklarowanej zmiennej polega na napisaniu nazwy zmiennej po lewej stronie znaku równości (=) i wyrażenia o odpowiedniej wartości po jego prawej stronie.

```
int vacationDays;
vacationDays = 12;
```

Zmienną można zadeklarować i zainicjować w jednym wierszu. Na przykład:

```
int vacationDays = 12;
```

Wreszcie, deklaracje w Javie można umieszczać w dowolnym miejscu w kodzie. Na przykład poniższy kod jest poprawny:

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // Zmienna może być zadeklarowana w tym miejscu.
```

Do dobrego stylu programowania w Javie zalicza się deklarowanie zmiennych jak najbliżej miejsca ich pierwszego użycia.

NB. W językach C i C++ rozróżnia się **deklarację** i **definicję** zmiennej. Na przykład:
`int i = 10;`
jest definicją zmiennej, podczas gdy:
`extern int i;`
to deklaracja. W Javie deklaracje nie są oddzielane od definicji.

Stałe

Stałe oznaczamy słowem kluczowym `final`. Na przykład:

```
public class Constants
{
public static void main(String[] args)
{
final double CM_PER_INCH = 2.54;
double paperWidth = 8.5;
double paperHeight = 11;
System.out.println("Rozmiar papieru w centymetrach: "
+ paperWidth * CM_PER_INCH + " na " + paperHeight * CM_PER_INCH);
}
}
```

Słowo kluczowe `final` oznacza, że można tylko jeden raz przypisać wartość i nie będzie można już jej zmienić w programie. Nazwy stałych piszemy zwyczajowo samymi wielkimi literami.

W Javie chyba najczęściej używa się stałych, które są dostępne dla wielu metod jednej klasy. Są to tak zwane **stałe klasowe**. Tego typu stałe definiujemy za pomocą słowa kluczowego `static final`. Oto przykład użycia takiej stałej:

```
public class Constants2
{
public static final double CM_PER_INCH = 2.54;
public static void main(String[] args)
{
double paperWidth = 8.5;
double paperHeight = 11;
System.out.println("Rozmiar papieru w centymetrach: "
```

```
+ paperWidth * CM_PER_INCH + " na " + paperHeight * CM_PER_INCH);
}
}
```

Zauważmy, że definicja stałej klasowej znajduje się **na zewnątrz** metody main. W związku z tym stała ta może być używana także przez inne metody tej klasy. Ponadto, jeśli (jak w naszym przykładzie) stała jest zadeklarowana jako publiczna (public), dostęp do niej mają także metody innych klas — jak w naszym przypadku Constants2.CM_PER_INCH.

NB. Słowo const jest słowem zarezerwowanym w Javie, ale obecnie nie jest do niczego używane. Do deklaracji stałych trzeba używać słowa kluczowego final.

Operatory

Znane wszystkim operatory arytmetyczne +, –, * i / służą w Javie odpowiednio do wykonywania operacji dodawania, odejmowania, mnożenia i dzielenia. Operator / oznacza dzielenie całkowitoliczbowe, jeśli obie liczby są typu całkowitoliczbowego, oraz dzielenie zmiennoprzecinkowe w przeciwnym przypadku. Operatorem reszty z dzielenia (**dzielenia modulo**) jest symbol %. Na przykład wynikiem działania 15/2 jest 7, a 15%2 jest 1, podczas gdy 15.0/2 = 7.5.

Pamiętajmy, że dzielenie całkowitoliczbowe przez zero powoduje wyjątek, podczas gdy wynikiem dzielenia zmiennoprzecinkowego przez zero jest nieskończoność lub wartość NaN.
Binarne operatory arytmetyczne w przypisaniach można wygodnie skracać. Na przykład zapis:

```
x+= 4;
```

jest równoważny z zapisem:

```
x = x + 4
```

Ogólna zasada jest taka, że operator powinien się znajdować po lewej stronie znaku równości, np. *= czy %=.

NB. Jednym z głównych celów, które postawili sobie projektanci Javy, jest przenośność. Wyniki obliczeń powinny być takie same bez względu na to, której maszyny wirtualnej użyto. Uzyskanie takiej przenośności jest zaskakująco trudne w przypadku działań na liczbach zmiennoprzecinkowych. Typ double przechowuje dane liczbowe w 64 bitach pamięci, ale niektóre procesory mają 80-bitowe rejestry liczb zmiennoprzecinkowych. Rejestry te w swoich obliczeniach pośrednich stosują zwiększoną precyzję. Przyjrzyjmy się na przykład poniższemu działaniu:

```
double w = x * y/z;
```

Wiele procesorów Intel wartość wyrażenia x * y zapisuje w 80-bitowym rejestrze. Następnie wykonywane jest dzielenie przez z, a wynik z powrotem obcinany do 64 bitów. Tym sposobem otrzymujemy dokładniejsze wyniki i unikamy przekroczenia zakresu wykładnika. Ale wynik może być **inny**, niż gdyby obliczenia były cały czas wykonywane w 64 bitach. Z tego powodu w pierwszych specyfikacjach wirtualnej maszyny Javy był zapisany wymóg, aby wszystkie obliczenia pośrednie używały zmniejszonej precyzji. Nie przepadała za tym cała społeczność programistyczna. Obliczenia o zmniejszonej precyzji mogą nie tylko powodować przekroczenie zakresu, ale są też **wolniejsze** niż obliczenia o zwiększonej precyzji, ponieważ obcinanie bitów zajmowało czas. W związku z tym opracowano aktualizację

języka Java mającą na celu rozwiązać problem sprzecznych wymagań dotyczących optymalizacji wydajności i powtarzalności wyników. Projektanci maszyny wirtualnej mogą obecnie stosować zwiększoną precyzję w obliczeniach pośrednich. Jednak metody oznaczone słowem kluczowym strictfp muszą korzystać ze ścisłych działań zmiennoprzecinkowych, które dają powtarzalne wyniki.

Na przykład metodę main można oznaczyć następująco:

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

W takim przypadku wszystkie instrukcje znajdujące się w metodzie main używają ograniczonych obliczeń zmiennoprzecinkowych. Jeśli oznaczymy w ten sposób klasę, wszystkie jej metody będą stosować obliczenia zmiennoprzecinkowe o zmniejszonej precyzji.

Sedno problemu leży w działaniu procesorów Intel. W trybie domyślnym obliczenia pośrednie mogą używać rozszerzonego wykładnika, ale nie rozszerzonej mantysy (chipy Intela umożliwiają obcinanie mantysy niepowodujące strat wydajności). W związku z tym główna różnica pomiędzy trybem domyślnym a ścisłym jest taka, że obliczenia ścisłe mogą przekroczyć zakres, a domyślne nie.

Muszę jednak uspokoić tych, u których na ciele wystąpiła gęsia skórka w trakcie lektury tej uwagi. Przekroczenie zakresu liczby zmiennoprzecinkowej nie zdarza się na co dzień w zwykłych programach. W tej książce nie używamy słowa kluczowego strictfp.

Operatory inkrementacji i dekrementacji

Programiści doskonale wiedzą, że jednym z najczęściej wykonywanych działań na zmiennych liczbowych jest dodawanie lub odejmowanie jedynki. Java, podobnie jak C i C++, ma zarówno operator inkrementacji, jak i dekrementacji. Zapis `n++` powoduje zwiększenie wartości zmiennej `n` o jeden, a `n--` zmniejszenie jej o jeden. Na przykład kod:

```
int n = 12
n++;
```

zwiększa wartość przechowywaną w zmiennej `n` na `13`. Jako że operatory te zmieniają wartość zmiennej, nie można ich stosować do samych liczb. Na przykład nie można napisać `4++`.

Operatory te występują w dwóch postaciach. Powyżej widzieliśmy postaci przyrostkowe, które — jak wskazuje nazwa — umieszcza się po operandzie. Druga postać to postać przedrostkowa — `++n`. Obie zwiększają wartość zmiennej o jeden. Różnica pomiędzy nimi ujawnia się, kiedy zostaną użyte w wyrażeniu. W przypadku zastosowania formy przedrostkowej wartość zmiennej jest zwiększana przed obliczeniem wartości wyrażenia, a w przypadku formy przyrostkowej wartość zmiennej zwiększa się po obliczeniu wartości wyrażenia.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // a ma wartość 16, a m — 8
int b = 2 * n++; // b ma wartość 14, a n — 8
```

Nie zalecamy stosowania operatora `++` w innych wyrażeniach, ponieważ zaciemnia to kod i często powoduje irytujące błędy.

(Jak powszechnie wiadomo, nazwa języka C++ pochodzi od operatora inkrementacji, który jest też „winowajcą” powstania pierwszego dowcipu o tym języku. Przeciwnicy C++ zauważają, że nawet nazwa tego języka jest błędna: „Powinna brzmieć ++C, ponieważ języka tego chcielibyśmy używać tylko po wprowadzeniu do niego poprawek”.)

Operatory relacyjne i logiczne

Java ma pełny zestaw operatorów relacyjnych. Aby sprawdzić, czy dwa argumenty są równe, używamy dwóch znaków równości (==). Na przykład wyrażenie:

```
3 == 7
```

zwróci wartość false.

Operator nierówności ma postać !=. Na przykład wyrażenie:

```
3 != 7
```

zwróci wartość true.

Dodatkowo dostępne są operatory większości (>), mniejszości (<), mniejszy lub równy (<=) oraz większy lub równy (>=).

Operatorem koniunkcji logicznej w Javie, podobnie jak w C++, jest &&, a alternatywy logicznej ||. Jak nietrudno się domyślić, znając operator !=, znak wykrzyknika (!) jest operatorem negacji. Wartości wyrażeń z użyciem operatorów && i || są obliczane metodą na skróty. Wartość drugiego argumentu nie jest obliczana, jeśli ostateczny rezultat wynika już z pierwszego. Jeżeli między dwoma wyrażeniami postawimy operator &&:

wyrażenie$_1$ && *wyrażenie*$_2$

i wartość logiczna pierwszego z nich okaże się false, to wartość całego wyrażenia nie może być inna niż false. W związku z tym wartość drugiego wyrażenia **nie** jest obliczana. Można to wykorzystać do unikania błędów. Jeśli na przykład wartość zmiennej x w wyrażeniu:

```
x != 0 && 1/x > x + y // Unikamy dzielenia przez zero.
```

jest równa zero, druga jego część nie będzie obliczana. Zatem działanie 1/x nie zostanie wykonane, jeśli x = 0, dzięki czemu nie wystąpi błąd dzielenia przez zero.

Podobnie wartość wyrażenia *wyrażenie*$_1$ || *wyrażenie*$_2$ ma automatycznie wartość true, jeśli pierwsze wyrażenie ma wartość true. Wartość drugiego nie jest obliczana.

W Javie dostępny jest też czasami przydatny operator trójargumentowy w postaci ?:. Wartością wyrażenia:

warunek ? *wyrażenie*$_1$: *wyrażenie*$_2$

jest *wyrażenie*$_1$, jeśli *warunek* ma wartość true, lub *wyrażenie*$_2$, jeśli *warunek* ma wartość false.

Na przykład wynikiem wyrażenia:

```
x < y ? x : y
```

jest x lub y — w zależności od tego, która wartość jest mniejsza.

Operatory bitowe

Do pracy na typach całkowitoliczbowych można używać operatorów dających dostęp bezpośrednio do bitów, z których się one składają. Oznacza to, że za pomocą techniki maskowania

można dobrać się do poszczególnych bitów w liczbie. Operatory bitowe to:
& (bitowa koniunkcja) | (bitowa alternatywa) ^ (lub wykluczające) ~(bitowa negacja)
Operatory te działają na bitach. Jeśli na przykład zmienna n jest typu int, to wyrażenie:
```
int fourthBitFromRight = (n & 8) / 8;
```
da wynik 1, jeśli czwarty bit od prawej w binarnej reprezentacji wartości zmiennej n jest jedynką, lub 0 w przeciwnym razie. Dzięki użyciu odpowiedniej potęgi liczby 2 można zamaskować wszystkie bity poza jednym.

NB. Operatory & i | zastosowane do wartości logicznych zwracają wartości logiczne. Są one podobne do operatorów && i ||, tyle że do obliczania wartości wyrażeń z ich użyciem nie jest stosowana metoda na skróty. A zatem wartości obu argumentów są zawsze obliczane przed zwróceniem wyniku.

Można też używać tak zwanych operatorów przesunięcia, w postaci >> i <<, które przesuwają liczbę o jeden bit w prawo lub w lewo. Często przydatne są przy tworzeniu ciągów bitów używanych przy maskowaniu:
```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```
Ostatni z operatorów bitowych >>> odpowiada za przesunięcie bitowe w prawo z wypełnieniem zerami, podczas gdy operator >> przesuwa bity w prawo i do ich wypełnienia używa znaku liczby. Nie ma operatora <<<.

Funkcje i stałe matematyczne

Klasa Math zawiera zestaw funkcji matematycznych, które mogą być bardzo przydatne przy pisaniu niektórych rodzajów programów.
Do wyciągania pierwiastka stopnia drugiego z liczby służy metoda sqrt:
```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // wynik 2.0
```

NB. Między metodami println i sqrt jest pewna różnica. Pierwsza działa na obiekcie System.out, który jest zdefiniowany w klasie System. Druga natomiast nie działa na żadnym obiekcie. Tego typu metody noszą nazwę **metod statycznych**.

W Javie nie ma operatora podnoszącego liczbę do potęgi. Do tego celu trzeba użyć metody pow dostępnej w klasie Math. Wyrażenie:
```
double y = Math.pow(x, a);
```
ustawia wartość zmiennej y na liczbę x podniesioną do potęgi a (x_a). Metoda pow przyjmuje parametry typu double i zwraca wynik tego samego typu.

Klasa Math udostępnia także metody obliczające funkcje trygonometryczne:

```
Math.sin
Math.cos
Math.tan
Math.atan
Math.atan2
```

a także funkcję wykładniczą i jej odwrotność, czyli logarytm naturalny, oraz logarytm dziesiętny:

```
Math.exp
Math.log
Math.log10
```

Dostępne są też dwie stałe określające w maksymalnym przybliżeniu stałe matematyczne Π i e:

```
Math.PI
Math.E
```

NB. Można uniknąć stosowania przedrostka Math przed metodami i stałymi matematycznymi, umieszczając poniższy wiersz kodu na początku pliku źródłowego:

```
import static java.lang.Math.*;
```

Na przykład:

```
System.out.println("Pierwiastek kwadratowy z \u03C0 wynosi " + sqrt(PI));
```

NB. Funkcje klasy Math używają procedur z jednostki liczb zmiennoprzecinkowych komputera w celu osiągnięcia jak najlepszej wydajności. Jeśli od prędkości ważniejsze są dokładne wyniki, należy posłużyć się klasą StrictMath. Implementuje ona algorytmy z biblioteki, którą można nieodpłatnie rozpowszechniać, o nazwie fdlibm, a która gwarantuje identyczne wyniki na wszystkich platformach. Kod źródłowy tych algorytmów można znaleźć na stronie *http://www.netlib.org/fdlibm* (dla każdej funkcji fdlibm, która ma więcej niż jedną definicję, klasa StrictMath używa wersji zgodnej ze standardem IEEE 754, której nazwa zaczyna się od litery *e*).

Konwersja typów numerycznych

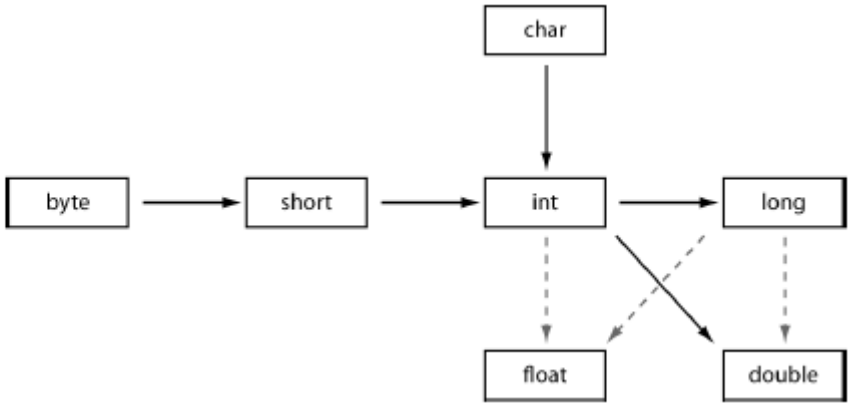
Często konieczna jest konwersja z jednego typu liczbowego na inny. Rysunek 3.1 przedstawia dozwolone rodzaje konwersji.

Sześć typów konwersji (rysunek 3.1) niepowodujących strat danych oznaczono strzałkami ciągłymi. Konwersje, które mogą spowodować utratę części danych, oznaczono strzałkami przerywanymi. Na przykład duża liczba całkowita, jak 123 456 789, składa się z większej liczby cyfr, niż może się zmieścić w typie float. Po konwersji tej liczby całkowitej na liczbę typu float stracimy nieco na precyzji:

```
int n = 123456789;
float f = n; // f ma wartość 1.23456792E8
```

Jeśli operatorem dwuargumentowym połączymy dwie wartości (np. n + f, gdzie n to liczba całkowita, a f liczba zmiennoprzecinkowa), zostaną one przekonwertowane na wspólny typ przed wykonaniem działania.

Rysunek 3.1. Dozwolone konwersje pomiędzy typami liczbowymi



Jeśli któryś z operandów jest typu double, drugi również zostanie przekonwertowany na typ double.
- W przeciwnym razie, jeśli któryś z operandów jest typu float, drugi zostanie przekonwertowany na typ float.
- W przeciwnym razie, jeśli któryś z operandów jest typu long, drugi zostanie przekonwertowany na typ long.
- W przeciwnym razie oba operandy zostaną przekonwertowane na typ int.

Rzutowanie

W poprzednim podrozdziale dowiedzieliśmy się, że wartości typu int są w razie potrzeby automatycznie konwertowane na typ double. Są jednak sytuacje, w których chcemy przekonwertować typ double na typ int. W Javie możliwe są takie konwersje, ale oczywiście mogą one pociągać za sobą utratę informacji. Konwersje, w których istnieje ryzyko utraty informacji, nazywają się **rzutowaniem** (ang. *casting*). Aby wykonać rzutowanie, należy przed nazwą rzutowanej zmiennej postawić nazwę typu docelowego w okrągłych nawiasach. Na przykład:

```
double x = 9.997;
int nx = (int) x;
```

W wyniku tego działania zmienna nx będzie miała wartość 9, ponieważ rzutowanie liczby zmiennoprzecinkowej na całkowitą powoduje usunięcie części ułamkowej.
Aby **zaokrąglić** liczbę zmiennoprzecinkową do **najbliższej** liczby całkowitej (co w większości przypadków bardziej się przydaje), należy użyć metody Math.round:

```
double x = 9.997;
int nx = (int) Math.round(x);
```

Teraz zmienna nx ma wartość 10. Przy zaokrąglaniu za pomocą metody round nadal konieczne jest zastosowanie rzutowania, tutaj (int). Jest to spowodowane tym, że metoda round zwraca wartość typu long, a tego typu wartość można przypisać zmiennej typu int wyłącznie na

drodze jawnego rzutowania, ponieważ istnieje ryzyko utraty danych.

NB. Wynikiem rzutowania na określony typ liczby, która nie mieści się w jego zakresie, jest obcięcie tej liczby i powstanie całkiem nowej wartości. Na przykład rzutowanie (byte) 300 da w wyniku liczbę 44.

NB. Nie można wykonać rzutowania pomiędzy wartościami liczbowymi i logicznymi. Zapobiega to powstawaniu wielu błędów. W nielicznych przypadkach, kiedy wymagana jest konwersja wartości logicznej na wartość liczbową, można użyć wyrażenia warunkowego, np. b ? 1 : 0.

Nawiasy i priorytety operatorów

Tabela 3.4 przedstawia zestawienie operatorów z uwzględnieniem ich priorytetów. Jeśli nie ma nawiasów, kolejność wykonywania działań jest taka jak kolejność operatorów w tabeli. Operatory o takim samym priorytecie są wykonywane od lewej do prawej, z wyjątkiem tych, które mają wiązanie prawostronne, podane w tabeli. Ponieważ operator && ma wyższy priorytet od operatora ||, wyrażenie:

```
a && b || c
```

jest równoznaczne z wyrażeniem:

```
(a && b) || c
```

Tabela 3.4. Priorytety operatorów

Operator	Wiązanie
[] . () (wywołanie metody)	lewe
! ~ ++ -- + (jednoargumentowy) () (rzutowanie) new	prawe
* / %	lewe
+ -	lewe
<< >> >>>	lewe
< <= > >= instanceof	lewe
== !=	lewe
&	lewe
^	lewe
|	lewe
&&	lewe
||	lewe
?:	prawe
= += -= *= /= %= &= /= ^= <<= >>= >>>=	prawe

Ze względu na fakt, że operator += ma wiązanie lewostronne, wyrażenie:

```
a += b += c
```

jest równoważne z wyrażeniem:

```
a += (b += c)
```

To znaczy, że wartość wyrażenia b += c (która wynosi tyle co b po dodawaniu) zostanie dodana do a.

NB. W przeciwieństwie do języków C i C++ Java nie ma operatora przecinka. Jednak w pierwszym i trzecim argumencie instrukcji for można używać **list wyrażeń oddzielonych przecinkami**.

Typ wyliczeniowy

Czasami zmienna może przechowywać tylko ograniczoną liczbę wartości. Na przykład kiedy sprzedajemy pizzę albo ubrania, możemy mieć rozmiary mały, średni, duży i ekstra duży. Oczywiście można te rozmiary zakodować w postaci cyfr 1, 2, 3 i 4 albo liter M, S, D i X.
To podejście jest jednak podatne na błędy. Zbyt łatwo można zapisać w zmiennej nieprawidłową wartość (jak 0 albo m).
Można też definiować własne **typy wyliczeniowe** (ang. *enumerated type*). Typ wyliczeniowy zawiera skończoną liczbę nazwanych wartości. Na przykład:

```
enum Rozmiar { MAŁY, ŚREDNI, DUŻY, EKSTRA_DUŻY };
```

Teraz możemy deklarować zmienne takiego typu:

```
Rozmiar s = Rozmiar.ŚREDNI;
```

Zmienna typu Rozmiar może przechowywać tylko jedną z wartości wymienionych w deklaracji typu lub specjalną wartość null, która oznacza, że zmienna nie ma w ogóle żadnej wartości.

Łańcuchy

W zasadzie łańcuchy w Javie składają się z szeregu znaków Unicode. Na przykład łańcuch "Java\u2122" składa się z pięciu znaków Unicode: J, a, v, a i ™. W Javie nie ma wbudowanego typu String. Zamiast tego standardowa biblioteka Javy zawiera predefiniowaną klasę o takiej właśnie nazwie. Każdy łańcuch w cudzysłowach jest obiektem klasy String:

```
String e = ""; // pusty łańcuch
String greeting = "Cześć!";
```

Podłańcuchy

Aby wydobyć z łańcucha podłańcuch, należy użyć metody substring klasy String. Na

przykład:

```
String greeting = "Cześć!";
String s = greeting.substring(0, 3);
```

Powyższy kod zwróci łańcuch "Cze".

Drugi parametr metody substring określa położenie pierwszego znaku, którego **nie** chcemy skopiować. W powyższym przykładzie chcieliśmy skopiować znaki na pozycjach 0, 1 i 2 (od pozycji 0 do 2 włącznie). Z punktu widzenia metody substring nasz zapis oznacza: od pozycji zero włącznie do pozycji 3 **z wyłączeniem**.

Sposób działania metody substring ma jedną zaletę: łatwo można obliczyć długość podłańcucha. Łańcuch s.substring(a, b) ma długość b - a. Na przykład łańcuch "Cze" ma długość 3 - 0 = 3.

Konkatenacja

W Javie, podobnie jak w większości innych języków programowania, można łączyć (konkatenować) łańcuchy za pomocą znaku +.

```
String expletive = "brzydkie słowo";
String PG13 = "usunięto";
String message = expletive + PG13;
```

Powyższy kod ustawia wartość zmiennej message na łańcuch "brzydkiesłowousunięto" (zauważ brak spacji pomiędzy słowami). Znak + łączy dwa łańcuchy w takiej kolejności, w jakiej zostały podane, **nic** w nich nie zmieniając.

Jeśli z łańcuchem zostanie połączona wartość niebędąca łańcuchem, zostanie ona przekonwertowana na łańcuch (w rozdziale 5. przekonamy się, że każdy obiekt w Javie można przekonwertować na łańcuch). Na przykład kod:

```
int age = 13;
String rating = "PG" + age;
```

ustawia wartość zmiennej rating na łańcuch "PG13".

Funkcjonalność ta jest często wykorzystywana w instrukcjach wyjściowych. Na przykład kod:

```
System.out.println("Odpowiedź brzmi " + answer);
```

jest w pełni poprawny i wydrukowałby to, co potrzeba (przy zachowaniu odpowiednich odstępów, gdyż po słowie brzmi znajduje się spacja).

Łańcuchów nie można modyfikować

W klasie String brakuje metody, która umożliwiałaby **zmianę** znaków w łańcuchach. Aby zmienić komunikat w zmiennej greeting na Czekaj, nie możemy bezpośrednio zamienić trzech ostatnich znaków na „kaj”. Programiści języka C są w takiej sytuacji zupełnie bezradni. Jak zmodyfikować łańcuch? W Javie okazuje się to bardzo proste. Należy połączyć

podłańcuch, który chcemy zachować, ze znakami, które chcemy wstawić w miejsce tych wyrzuconych.

```
greeting = greeting.substring(0, 3) + "kaj";
```

Ta deklaracja zmienia wartość przechowywaną w zmiennej greeting na "Czekaj".

Jako że w łańcuchach nie można zmieniać znaków, obiekty klasy String w dokumentacji języka Java są określane jako **niezmienialne** (ang. *immutable*). Podobnie jak liczba 3 jest zawsze liczbą 3, łańcuch "Cześć!" zawsze będzie szeregiem jednostek kodowych odpowiadających znakom C, z, e, ś, ć i !. Nie można zmienić tych wartości. Można jednak, o czym się przekonaliśmy, zmienić zawartość **zmiennej** greeting, sprawiając, aby odwoływała się do innego łańcucha. Podobnie możemy zadecydować, że zmienna liczbowa przechowująca wartość 3 zmieni odwołanie na wartość 4.

Czy to nie odbija się na wydajności? Wydaje się, że zmiana jednostek kodowych byłaby prostsza niż tworzenie nowego łańcucha od początku. Odpowiedź brzmi: tak i nie. Rzeczywiście generowanie nowego łańcucha zawierającego połączone łańcuchy "Cze" i "kaj" jest nieefektywne, ale niezmienialność łańcuchów ma jedną zaletę: kompilator może traktować łańcuchy jako **współdzielone**.

Aby zrozumieć tę koncepcję, wyobraźmy sobie, że różne łańcuchy są umieszczone w jednym wspólnym zbiorniku. Zmienne łańcuchowe wskazują na określone lokalizacje w tym zbiorniku. Jeśli skopiujemy taką zmienną, zarówno oryginalny łańcuch, jak i jego kopia współdzielą te same znaki.

Projektanci języka Java doszli do wniosku, że korzyści płynące ze współdzielenia są większe niż straty spowodowane edycją łańcuchów poprzez ekstrakcję podłańcuchów i konkatenację. Przyjrzyj się swoim własnym programom — zapewne w większości przypadków nie ma w nich modyfikacji łańcuchów, a głównie różne rodzaje porównań (jest tylko jeden dobrze znany wyjątek — składanie łańcuchów z pojedynczych znaków lub krótszych łańcuchów przychodzących z klawiatury bądź pliku; dla tego typu sytuacji w Javie przewidziano specjalną klasę, którą opisujemy w podrozdziale „Składanie łańcuchów").

Porównywanie łańcuchów

Do sprawdzania, czy dwa łańcuchy są identyczne, służy metoda equals. Wyrażenie:

```
s.equals(t)
```

zwróci wartość true, jeśli łańcuchy s i t są identyczne, lub false w przeciwnym przypadku. Zauważmy, że s i t mogą być zmiennymi łańcuchowymi lub stałymi łańcuchowymi. Na przykład wyrażenie:

```
"Cześć!".equals(greeting")
```

jest poprawne. Aby sprawdzić, czy dwa łańcuchy są identyczne, z pominięciem wielkości liter, należy użyć metody equalsIgnoreCase.

```
"Cześć!".equalsIgnoreCase("cześć!")
```

NB. Programiści języka C, którzy po raz pierwszy stykają się z łańcuchami w Javie, nie mogą ukryć zdumienia, ponieważ dla nich łańcuchy są tablicami znaków:
```
char greeting[] = "Cześć!";
```
Jest to nieprawidłowa analogia. Łańcuch w Javie można porównać ze wskaźnikiem char*:
```
char* greeting = "Cześć!";
```
Kiedy zastąpimy komunikat greeting jakimś innym łańcuchem, Java wykona z grubsza takie działania:
```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "kaj", 3);
greeting = temp;
```
Teraz zmienna greeting wskazuje na łańcuch "Czekaj". Nawet najbardziej zatwardziały wielbiciel języka C musi przyznać, że składnia Javy jest bardziej elegancka niż szereg wywołań funkcji strncpy. Co się stanie, jeśli wykonamy jeszcze jedno przypisanie do zmiennej greeting?
```
greeting = "Cześć!";
```
Czy to nie spowoduje wycieku pamięci? Przecież oryginalny łańcuch został umieszczony na stercie. Na szczęście Java automatycznie usuwa nieużywane obiekty. Jeśli dany blok pamięci nie jest już potrzebny, zostanie wyczyszczony.
Typ String Javy dużo łatwiej opanować programistom języka C++, którzy używają klasy string zdefiniowanej w standardzie ISO/ANSI tego języka. Obiekty klasy string w C++ także automatycznie przydzielają i czyszczą pamięć. Zarządzanie pamięcią odbywa się w sposób jawny za pośrednictwem konstruktorów, operatorów przypisania i destruktorów. Ponieważ w C++ łańcuchy są zmienialne (ang. *mutable*), można zmieniać w nich poszczególne znaki.

Do porównywania łańcuchów **nie** należy używać operatora ==! Za jego pomocą można tylko stwierdzić, czy dwa łańcuchy są przechowywane w tej samej lokalizacji. Oczywiście skoro łańcuchy są przechowywane w tym samym miejscu, to muszą być równe. Możliwe jest jednak też przechowywanie wielu kopii jednego łańcucha w wielu różnych miejscach.
```
String greeting = "Cześć!"; // Inicjacja zmiennej greeting łańcuchem.
if (greeting == "Cześć!") . . .
// prawdopodobnie true
if (greeting.substring(0, 3) == "Cze") . . .
// prawdopodobnie false
```
Gdyby maszyna wirtualna zawsze traktowała równe łańcuchy jako współdzielone, można by było je porównywać za pomocą operatora ==. Współdzielone są jednak tylko **stałe** łańcuchowe. Łańcuchy będące na przykład wynikiem operacji wykonywanych za pomocą operatora + albo metody substring nie są współdzielone. W związku z tym **nigdy** nie używaj operatora == do porównywania łańcuchów, chyba że chcesz stworzyć program zawierający najgorszy rodzaj błędu — pojawiający się od czasu do czasu i sprawiający wrażenie, że występuje losowo.

NB. Osoby przyzwyczajone do klasy string w C++ muszą zachować szczególną ostrożność przy porównywaniu łańcuchów. Klasa C++ string przesłania operator == do porównywania łańcuchów. W Javie dość niefortunnie nadano łańcuchom takie same własności jak wartościom liczbowym, aby następnie nadać im właściwości wskaźników, jeśli chodzi o porównywanie. Projektanci tego języka mogli zmienić definicję operatora == dla łańcuchów, podobnie jak zrobili z operatorem +. Cóż, każdy język ma swoje wady. Programiści języka C nigdy nie używają operatora == do porównywania łańcuchów. Do tego służy im funkcja strcmp. Metoda Javy compareTo jest dokładnym odpowiednikiem funkcji strcmp. Można napisać:
```
if (greeting.compareTo("Cześć!") == 0) . . .
```
ale użycie metody equals wydaje się bardziej przejrzystym rozwiązaniem.

Łańcuchy puste i łańcuchy null

Pusty łańcuch "" to łańcuch o zerowej długości. Aby sprawdzić, czy łańcuch jest pusty, można użyć instrukcji:

```
if (str.length() == 0)
```

lub

```
if (str.equals(""))
```

Pusty łańcuch jest w Javie obiektem zawierającym informację o swojej długości (0) i pustą treść. Ponadto zmienna typu String może też zawierać specjalną wartość o nazwie null, oznaczającą, że aktualnie ze zmienną nie jest powiązany żaden obiekt (więcej informacji na temat wartości null znajduje się w rozdziale 4.). Aby sprawdzić, czy wybrany łańcuch jest null, można użyć następującej instrukcji warunkowej:

```
if (str == null)
```

Czasami trzeba też sprawdzić, czy łańcuch nie jest ani pusty, ani null. Wówczas można się posłużyć poniższą instrukcją warunkową:

```
if (str != null && str.length() != 0)
```

Najpierw należy sprawdzić, czy łańcuch nie jest null, ponieważ wywołanie metody na wartości null jest błędem.

Współrzędne kodowe znaków i jednostki kodowe

Łańcuchy w Javie są ciągami wartości typu char. Jak wiemy z podrozdziału „Typ char", typ danych char jest jednostką kodową reprezentującą współrzędne kodowe znaków Unicode w systemie UTF-16. Najczęściej używane znaki Unicode mają reprezentacje składające się z jednej jednostki kodowej. Reprezentacje znaków dodatkowych składają się z par jednostek kodowych.

Metoda length zwraca liczbę jednostek kodowych, z których składa się podany łańcuch w systemie UTF-16. Na przykład:

```
String greeting = "Cześć!";
int n = greeting.length(); // wynik = 6
```

Aby sprawdzić rzeczywistą długość, to znaczy liczbę współrzędnych kodowych znaków, należy napisać:

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

Wywołanie s.charAt(n) zwraca jednostkę kodową znajdującą się na pozycji n, gdzie n ma wartość z zakresu pomiędzy 0 a s.length() - 1. Na przykład:

```
char first = greeting.charAt(0); // Pierwsza jest litera 'C'.
char last = greeting.charAt(4); // Piąty znak to 'ć'.
```

Aby dostać się do i-tej współrzędnej kodowej znaku, należy użyć następujących instrukcji:

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```

NB. W Javie, podobnie jak w C i C++, współrzędne i jednostki kodowe w łańcuchach są liczone od 0.

Dlaczego robimy tyle szumu wokół jednostek kodowych? Rozważmy poniższe zdanie:

Z oznacza zbiór liczb całkowitych

Znak **Z** wymaga dwóch jednostek kodowych w formacie UTF-16. Wywołanie:

```
char ch = sentence.charAt(1)
```

nie zwróci spacji, ale drugą jednostkę kodową znaku . Aby uniknąć tego problemu, nie należało używać typu char. Działa on na zbyt niskim poziomie.

Jeśli nasz kod przemierza łańcuch i chcemy zobaczyć każdą współrzędną kodową po kolei, należy użyć poniższych instrukcji:

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

Można też napisać kod działający w odwrotną stronę:

```
i--;
if (Character.isSurrogate(sentence.charAt(i))) i--;
int cp = sentence.codePointAt(i);
```

API String

Klasa String zawiera ponad 50 metod. Zaskakująco wiele z nich jest na tyle użytecznych, że możemy się spodziewać, iż będziemy ich często potrzebować. Poniższy wyciąg z API zawiera zestawienie metod, które w naszym odczuciu są najbardziej przydatne.

NB. Takie wyciągi z API znajdują się w wielu miejscach książki. Ich celem jest przybliżenie czytelnikowi API Javy. Każdy wyciąg z API zaczyna się od nazwy klasy, np. java.lang.String — znaczenie nazwy **pakietu** java.lang jest wyjaśnione w rozdziale 4. Po nazwie klasy znajdują się nazwy, objaśnienia i opis parametrów jednej lub większej liczby metod.
Z reguły nie wymieniamy wszystkich metod należących do klasy, ale wybieramy te, które są najczęściej używane, i zamieszczamy ich zwięzłe opisy. Pełną listę metod można znaleźć w dokumentacji dostępnej w internecie (zobacz podrozdział 3.6.8, „Dokumentacja API w internecie").
Dodatkowo podajemy numer wersji Javy, w której została wprowadzona dana klasa. Jeśli jakaś metoda została do niej dodana później, ma własny numer wersji.

java.lang.String **1.0**

- char charAt(int index)

Zwraca jednostkę kodową znajdującą się w określonej lokalizacji. Metoda ta jest przydatna tylko w pracy na niskim poziomie nad jednostkami kodowymi.

- int codePointAt(int index) 5.0

Zwraca współrzędną kodową znaku, która zaczyna się lub kończy w określonej lokalizacji.

- int offsetByCodePoints(int startIndex, int cpCount) 5.0

Zwraca indeks współrzędnej kodowej, która znajduje się w odległości cpCount

współrzędnych kodowych od współrzędnej kodowej startIndex.

- int compareTo(String other)

Zwraca wartość ujemną, jeśli łańcuch znajduje się przed innym (other) łańcuchem w kolejności słownikowej, wartość dodatnią, jeśli znajduje się za nim, lub 0, jeśli łańcuchy są identyczne.

- boolean endsWith(String suffix)

Zwraca wartość true, jeśli na końcu łańcucha znajduje się przyrostek suffix.

- boolean equals(Object other)

Zwraca wartość true, jeśli łańcuch jest identyczny z łańcuchem other.

- boolean equalsIgnoreCase(String other)

Zwraca wartość true, jeśli łańcuch jest identyczny z innym łańcuchem przy zignorowaniu wielkości liter.

- int indexOf(String str)
- int indexOf(String str, int fromIndex)
- int indexOf(int cp)
- int indexOf(int cp, int fromIndex)

Zwraca początek pierwszego podłańcucha podanego w argumencie str lub współrzędnej kodowej cp, szukanie zaczynając od indeksu 0, pozycji fromIndex czy też -1, jeśli napisu str nie ma w tym łańcuchu.

- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)
- int lastindexOf(int cp)
- int lastindexOf(int cp, int fromIndex)

Zwraca początek ostatniego podłańcucha podanego w argumencie str lub współrzędnej kodowej cp. Szukanie zaczyna od końca łańcucha albo pozycji fromIndex.

- int length()

Zwraca długość łańcucha.

- int codePointCount(int startIndex, int endIndex) 5.0

Zwraca liczbę współrzędnych kodowych znaków znajdujących się pomiędzy pozycjami StartIndex i endIndex - 1. Surogaty niemające pary są traktowane jako współrzędne kodowe.

- String replace(CharSequence oldString, CharSequence newString)

Zwraca nowy łańcuch, w którym wszystkie łańcuchy oldString zostały zastąpione łańcuchami newString. Można podać obiekt String lub StringBuilder dla parametru CharSequence.

- boolean startsWith(String prefix)

Zwraca wartość true, jeśli łańcuch zaczyna się od podłańcucha prefix.

- String substring(int beginIndex)
- String substring(int beginIndex, int endIndex)

Zwraca nowy łańcuch składający się ze wszystkich jednostek kodowych znajdujących się na pozycjach od beginIndex do końca łańcucha albo do endIndex - 1.

- String toLowerCase()

Zwraca nowy łańcuch zawierający wszystkie znaki z oryginalnego ciągu przekonwertowane na małe litery.

- String toUpperCase()

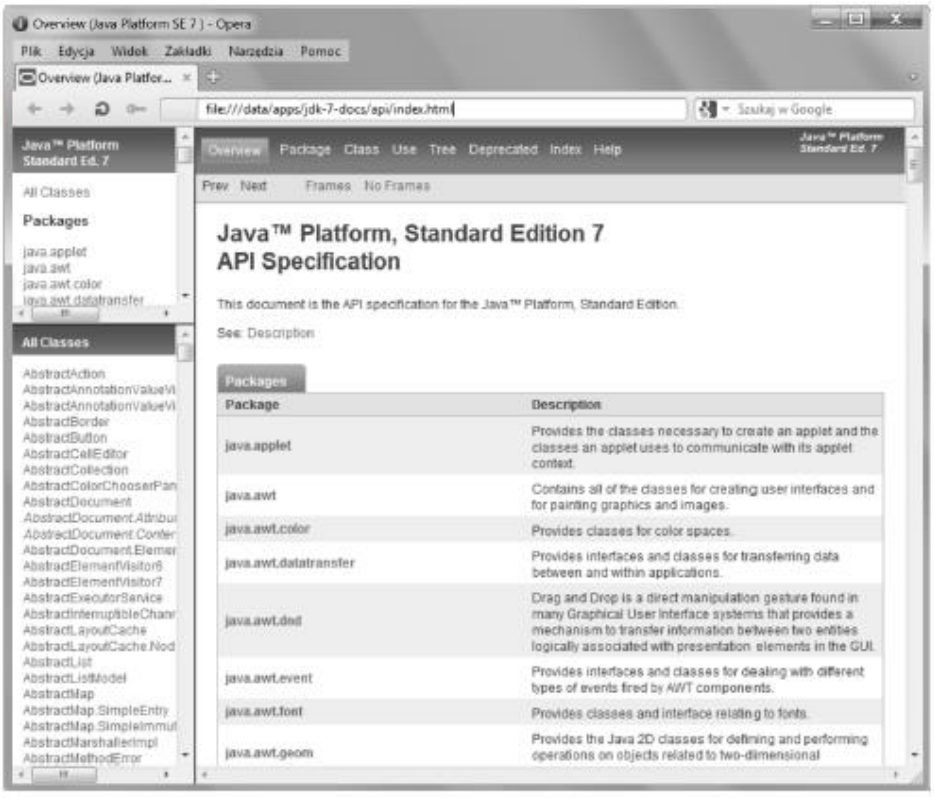
Zwraca nowy łańcuch zawierający wszystkie znaki z oryginalnego ciągu przekonwertowane na duże litery.

- String trim()
Usuwa wszystkie białe znaki z początku i końca łańcucha. Zwraca wynik jako nowy łańcuch.

Dokumentacja API w internecie

Jak się przed chwilą przekonaliśmy, klasa String zawiera mnóstwo metod. W bibliotekach standardowych jest kilka tysięcy klas, które zawierają dużo więcej metod. Zapamiętanie wszystkich przydatnych metod i klas jest niemożliwe. Z tego względu koniecznie trzeba się zapoznać z zamieszczoną w internecie dokumentacją API, w której można znaleźć informacje o każdej metodzie dostępnej w standardowej bibliotece. Dokumentacja API wchodzi też w skład pakietu JDK. Aby ją otworzyć, należy w przeglądarce wpisać adres pliku *docs/api/index.html* znajdującego się w katalogu instalacji JDK. Stronę tę przedstawia rysunek 3.2.

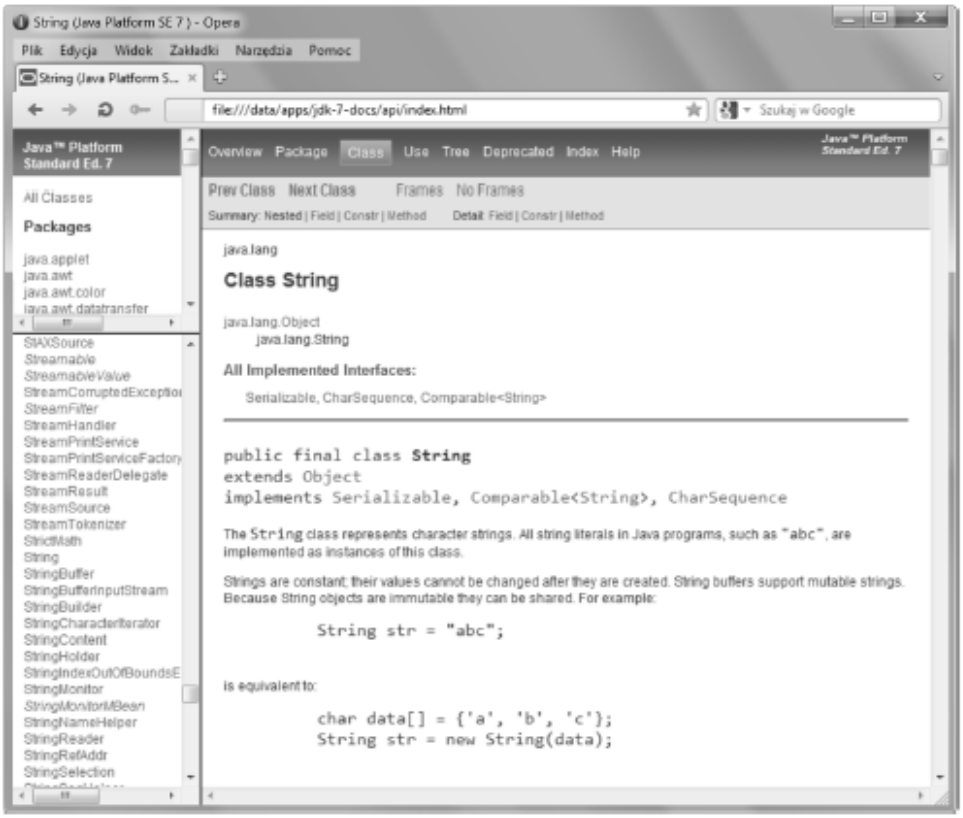
Rysunek 3.2. Trzyczęściowe okno dokumentacji API



Ekran jest podzielony na trzy części. W górnej ramce po lewej stronie okna znajduje się lista wszystkich dostępnych pakietów. Pod nią jest nieco większa ramka, która zawiera listy wszystkich klas. Kliknięcie nazwy jednej z klas powoduje wyświetlenie dokumentacji tej klasy

w dużym oknie po prawej stronie (zobacz rysunek 3.3). Aby na przykład uzyskać dodatkowe informacje na temat metod dostępnych w klasie String, należy w drugiej ramce znaleźć odnośnik *String* i go kliknąć.

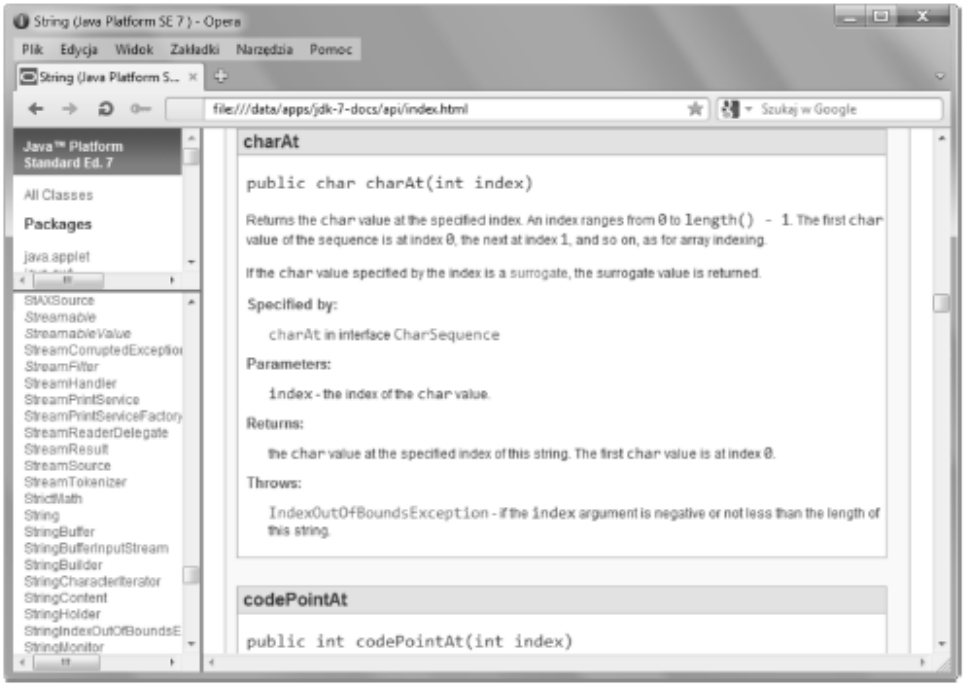
Rysunek 3.3.
Opis klasy String



Następnie za pomocą suwaka znajdujemy zestawienie wszystkich metod posortowanych w kolejności alfabetycznej (zobacz rysunek 3.4). Aby przeczytać dokładny opis wybranej metody, kliknij jej nazwę (rysunek 3.5). Jeśli na przykład klikniemy odnośnik *compareTo-IgnoreCase*, wyświetli się opis metody compareToIgnoreCase.

Rysunek 3.4. Zestawienie metod klasy String

Rysunek 3.5. Szczegółowy opis metody klasy String



Składanie łańcuchów

Czasami konieczne jest złożenie łańcucha z krótszych łańcuchów, takich jak znaki wprowadzane z klawiatury albo słowa zapisane w pliku. Zastosowanie konkatenacji do tego celu byłoby wyjściem bardzo nieefektywnym. Za każdym razem, gdy łączone są znaki, tworzony jest nowy obiekt klasy String. Zabiera to dużo czasu i pamięci. Klasa StringBuilder pozwala uniknąć tego problemu.
Aby złożyć łańcuch z wielu bardzo małych części, należy wykonać następujące czynności.
Najpierw tworzymy pusty obiekt builder klasy StringBuilder:

```
StringBuilder builder = new StringBuilder();
```

Kolejne części dodajemy za pomocą metody append.

```
builder.append(ch); // Dodaje jeden znak.
builder.append(str); // Dodaje łańcuch.
```

Po złożeniu łańcucha wywołujemy metodę toString. Zwróci ona obiekt klasy String zawierający sekwencję znaków znajdującą się w obiekcie builder.

```
String completedString = builder.toString();
```

NB. Klasa StringBuilder została wprowadzona w JDK 5.0. Jej poprzedniczka o nazwie StringBuffer jest nieznacznie mniej wydajna, ale pozwala na dodawanie lub usuwanie znaków przez wiele wątków. Jeśli edycja łańcucha odbywa się w całości w jednym wątku (tak jest zazwyczaj), należy używać metody StringBuilder. API obu tych klas są identyczne.

Poniższy wyciąg z API przedstawia najczęściej używane metody dostępne w klasie StringBuilder.

java.lang.StringBuilder **5.0**

- StringBuilder()

Tworzy pusty obiekt builder.

- int length()

Zwraca liczbę jednostek kodowych zawartych w obiekcie builder lub buffer.

- StringBuilder append(String str)

Dodaje łańcuch c.

- StringBuilder append(char c)

Dodaje jednostkę kodową c.

- StringBuilder appendCodePoint(int cp)

Dodaje współrzędną kodową, konwertując ją na jedną lub dwie jednostki kodowe.

- void setCharAt(int i, char c)

Ustawia *i*-tą jednostkę kodową na c.

- StringBuilder insert(int offset, String str)

Wstawia łańcuch, umieszczając jego początek na pozycji offset.

- StringBuilder insert(int offset, char c)

Wstawia jednostkę kodową na pozycji offset.

- StringBuilder delete(int startIndex, int endIndex)

Usuwa jednostki kodowe znajdujące się między pozycjami startIndex i endIndex - 1.

- String toString()
Zwraca łańcuch zawierający sekwencję znaków znajdującą się w obiekcie builder lub buffer.

Wejście i wyjście

Aby programy były bardziej interesujące, powinny przyjmować dane wejściowe i odpowiednio formatować dane wyjściowe. Oczywiście odbieranie danych od użytkownika w tworzonych obecnie programach odbywa się za pośrednictwem GUI. Jednak programowanie interfejsu wymaga znajomości wielu narzędzi i technik, które są nam jeszcze nieznane. Ponieważ naszym aktualnym priorytetem jest zapoznanie się z językiem programowania Java, poprzestaniemy na razie na skromnych programach konsolowych.

Odbieranie danych wejściowych

Wiadomo już, że drukowanie danych do standardowego strumienia wyjściowego (tzn. do okna konsoli) jest łatwe. Wystarczy wywołać metodę System.out.printIn. Pobieranie danych ze standardowego strumienia wejściowego System.in nie jest już takie proste. Czytanie danych odbywa się za pomocą skanera będącego obiektem klasy Scanner przywiązanego do strumienia System.in:

```
Scanner in = new Scanner(System.in);
```

Operator new i konstruktory zostały szczegółowo omówione w rozdziale 4.
Następnie dane wejściowe odczytuje się za pomocą różnych metod klasy Scanner. Na przykład metoda nextLine czyta jeden wiersz danych:

```
System.out.print("Jak się nazywasz? ");
String name = in.nextLine();
```

W tym przypadku zastosowanie metody nextLine zostało podyktowane tym, że dane na wejściu mogą zawierać spacje. Aby odczytać jedno słowo (ograniczone spacjami), należy wywołać poniższą metodę:

```
String firstName = in.next();
```

Do wczytywania liczb całkowitych służy metoda nextInt:

```
System.out.print("Ile masz lat? ");
int age = in.nextInt();
```

Podobne działanie ma metoda nextDouble, z tym że dotyczy liczb zmiennoprzecinkowych.
Program przedstawiony na listingu 3.2 prosi użytkownika o przedstawienie się i podanie wieku, a następnie drukuje informację typu:

```
Witaj użytkowniku Łukasz. W przyszłym roku będziesz mieć 32 lata.
```

Listing 3.2. InputTest.java

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

* Ten program demonstruje pobieranie danych z konsoli.
* @version 1.10 2004-02-10
* @author Cay Horstmann
```

```
*/
public class InputTest
{
public static void main(String[] args)
{
Scanner in = new Scanner(System.in);
// Pobranie pierwszej porcji danych.
System.out.print("Jak się nazywasz? ");
String name = in.nextLine();
// Pobranie drugiej porcji danych.
System.out.print("Ile masz lat? ");
int age = in.nextInt();
// Wydruk danych w konsoli.
System.out.println("Witaj użytkowniku" + name + ". W przyszłym roku będziesz
mieć " + (age + 1) + "lat.");
}
}
```

NB. Klasa Scanner nie nadaje się do odbioru haseł z konsoli, ponieważ wprowadzane dane są widoczne dla każdego. W Java SE 6 wprowadzono klasę Console, która służy właśnie do tego celu. Aby pobrać hasło, należy użyć poniższego kodu:

```
Console cons = System.console();
String username = cons.readLine("Nazwa użytkownika: ");
char[] passwd = cons.readPassword("Hasło: ");
```

Ze względów bezpieczeństwa hasło jest zwracane w tablicy znaków zamiast w postaci łańcucha. Po zakończeniu pracy z hasłem powinno się natychmiast nadpisać przechowującą je tablicę, zastępując obecne elementy jakimiś wartościami wypełniającymi (przetwarzanie tablic jest opisane w dalszej części tego rozdziału).

Przetwarzanie danych wejściowych za pomocą obiektu Console nie jest tak wygodne jak w przypadku klasy Scanner. Jednorazowo można wczytać tylko jeden wiersz danych. Nie ma metod umożliwiających odczyt pojedynczych słów lub liczb.

Należy także zwrócić uwagę na poniższy wiersz:

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

Znajduje się on na początku programu. Definicja klasy Scanner znajduje się w pakiecie java.util. Użycie jakiejkolwiek klasy spoza podstawowego pakietu java.lang wymaga wykorzystania dyrektywy import.

java.util.Scanner **5.0**

- Scanner(InputStream in)

Tworzy obiekt klasy Scanner przy użyciu danych z podanego strumienia wejściowego.

String nextLine()

Wczytuje kolejny wiersz danych.

- String text()

Wczytuje kolejne słowo (znakiem rozdzielającym jest spacja).

- int nextInt()
- double nextDouble()

Wczytuje i konwertuje kolejną liczbę całkowitą lub zmiennoprzecinkową.

- boolean hasNext()

Sprawdza, czy jest kolejne słowo.

- boolean hasNextInt()
- boolean hasNextDouble()

Sprawdza, czy dana sekwencja znaków jest liczbą całkowitą, czy liczbą

zmiennoprzecinkową.

java.lang.System **1.0**
- static Console console() **6**

Zwraca obiekt klasy Console umożliwiający interakcję z użytkownikiem za pośrednictwem okna konsoli, jeśli jest to możliwe, lub wartość null w przeciwnym przypadku. Obiekt Console jest dostępny dla wszystkich programów uruchomionych w oknie konsoli. W przeciwnym przypadku dostępność zależy od systemu.

java.io.Console **6**
- static char[] readPassword(String prompt, Object... args)
- static String readLine(String prompt, Object... args)

Wyświetla łańcuch prompt i wczytuje wiersz danych z konsoli. Za pomocą parametrów args można podać argumenty formatowania, o czym mowa w następnym podrozdziale.

Formatowanie danych wyjściowych

Wartość zmiennej x można wydrukować w konsoli za pomocą instrukcji System.out.print(x). Polecenie to wydrukuje wartość zmiennej x z największą liczbą cyfr niebędących zerami, które może pomieścić dany typ. Na przykład kod:

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

wydrukuje:

```
3333.3333333333335
```

Problemy zaczynają się wtedy, gdy chcemy na przykład wyświetlić liczbę dolarów i centów.

W pierwotnych wersjach Javy formatowanie liczb sprawiało sporo problemów. Na szczęście w wersji Java SE 5 wprowadzono zasłużoną już metodę printf z biblioteki C. Na przykład wywołanie:

```
System.out.printf("%8.2f", x);
```

drukuje wartość zmiennej x w **polu o szerokości** 8 znaków i z dwoma miejscami po przecinku. To znaczy, że poniższy wydruk zawiera wiodącą spację i siedem widocznych znaków:

```
3333,33
```

Metoda printf może przyjmować kilka parametrów. Na przykład:

```
System.out.printf("Witaj, %s. W przyszłym roku będziesz mieć lat %d", name, age);
```

Każdy **specyfikator formatu**, który zaczyna się od znaku %, jest zastępowany odpowiadającym mu argumentem. **Znak konwersji** znajdujący się na końcu specyfikatora formatu określa typ wartości do sformatowania: f oznacza liczbę zmiennoprzecinkową, s łańcuch, a d liczbę całkowitą dziesiętną. Tabela 3.5 zawiera wszystkie znaki konwersji.

Dodatkowo można kontrolować wygląd sformatowanych danych wyjściowych za pomocą kilku **znaczników**. Tabela 3.6 przedstawia wszystkie znaczniki. Na przykład przecinek dodaje separator grup. To znaczy:

```
System.out.printf("%, .2f", 10000.0 / 3.0);
```

wydrukuje:

3 333,33

Można stosować po kilka znaczników naraz, na przykład zapis "%,(.2f" oznacza użycie separatorów grup i ujęcie liczb ujemnych w nawiasy.

NB. Za pomocą znaku konwersji s można formatować dowolne obiekty. Jeśli obiekt taki implementuje interfejs Formattable, wywoływana jest jego metoda formatTo. W przeciwnym razie wywoływana jest metoda toString w celu przekonwertowania obiektu na łańcuch.

Tabela 3.5. Znaki konwersji polecenia printf

Znak konwersji	Typ	Przykład
d	Liczba całkowita dziesiętna	159
x	Liczba całkowita szesnastkowa	9f
o	Liczba całkowita ósemkowa	237
f	Liczba zmiennoprzecinkowa	15.9
e	Liczba zmiennoprzecinkowa w notacji wykładniczej	1.59e+01
g	Liczba zmiennoprzecinkowa (krótszy z formatów e i f)	–
a	Liczba zmiennoprzecinkowa szesnastkowa	0x1.fccdp3
s	Łańcuch	Witaj
c	Znak	H
b	Wartość logiczna	true
h	Kod mieszający	42628b2
tx	Data i godzina	Zobacz tabela 3.7
%	Symbol procenta	%
n	Separator wiersza właściwy dla platformy	–

Aby utworzyć sformatowany łańcuch, ale go nie drukować, należy użyć statycznej metody String.format:

```
String message = String.format("Witaj, %s. W przyszłym roku będziesz mieć lat %d",
 ·name, age);
```

Mimo że typ Date omawiamy szczegółowo dopiero w rozdziale 4., przedstawiamy krótki opis opcji metody printf do formatowania daty i godziny. Stosowany jest format dwuliterowy, w którym pierwsza litera to t, a druga jest jedną z liter znajdujących się w tabeli 3.7. Na przykład:

```
System.out.printf("%tc", new Date());
```

Wynikiem jest aktualna data i godzina w następującym formacie:

```
Pn lis 26 15:47:12 CET 2007
```

NB. Aby program zadziałał, na początku kodu źródłowego należy wstawić wiersz import java.util.Date;

Tabela 3.6. Znaczniki polecenia printf

Flaga	Przeznaczenie	Przykład
+	Oznacza, że przed liczbami zawsze ma się znajdować znak.	+3333,33
spacja	Oznacza, że liczby nieujemne są poprzedzone dodatkową spacją.	| 3333,33|
0	Oznacza dodanie początkowych zer.	003333,33
-	Oznacza, że pole ma być wyrównane do lewej.	|3333,33 |
(	Oznacza, że liczby ujemne mają być prezentowane w nawiasach.	(3333,33)
,	Oznacza, że poszczególne grupy mają być rozdzielane.	3 333,33
# (dla formatu f)	Oznacza, że zawsze ma być dodany przecinek dziesiętny.	3 333,
# (dla formatu x lub o)	Dodaje odpowiednio przedrostek 0x lub 0.	0xcafe
$	Określa indeks argumentu do sformatowania. Na przykład %1$d %1$x drukuje tę samą liczbę w notacji dziesiętnej i szesnastkowej.	159 9F
<	Formatuje podobnie jak poprzednia specyfikacja. Na przykład zapis %d %<x wydrukuje liczbę w formacie dziesiętnym i szesnastkowym.	159 9F

Tabela 3.7. Znaki konwersji Date i Time

Znak konwersji	Typ	Przykład
c	Pełna data i godzina	Pn lis 26 15:47:12 CET 2007
F	Data w formacie ISO 8601	2007-11-26
D	Data w formacie stosowanym w USA (miesiąc/dzień/rok)	11/26/07
T	Godzina w formacie 24-godzinnym	15:25:10
r	Godzina w formacie 12-godzinnym	03:52:55 PM
R	Godzina w formacie 24-godzinnym, bez sekund	15:25
Y	Rok w formacie czterocyfrowym	2007
y	Dwie ostatnie cyfry roku (z wiodącymi zerami)	07
C	Dwie pierwsze cyfry roku (z wiodącymi zerami)	20
B	Pełna nazwa miesiąca	listopad
b lub h	Skrót nazwy miesiąca	lis
m	Dwie cyfry oznaczające numer miesiąca (z wiodącym zerem)	02
d	Numer dnia miesiąca (z wiodącym zerem)	09
e	Numer dnia miesiąca (bez wiodącego zera)	9
A	Pełna nazwa dnia	poniedziałek
a	Skrót nazwy dnia	Pn
j	Dzień roku w formacie trzycyfrowym (z wiodącymi zerami), od 001 do 366	069
H	Godzina w formacie dwucyfrowym (z wiodącym zerem), od 00 do 23	18
k	Godzina w formacie dwucyfrowym (bez wiodącego zera), od 00 do 23	18
I	Godzina w formacie dwucyfrowym (z wiodącym zerem), od 01 do 12	06
l	Godzina w formacie dwucyfrowym (bez wiodącego zera), od 1 do 12	6
M	Minuty w formacie dwucyfrowym (z wiodącym zerem)	05
S	Sekundy w formacie dwucyfrowym (z wiodącym zerem)	19
L	Milisekundy w formacie trzycyfrowym (z wiodącymi zerami)	046
N	Nanosekundy w formacie dziewięciocyfrowym (z wiodącymi zerami)	047000000

Tabela 3.7. Znaki konwersji Date i Time — *ciąg dalszy*

Znak konwersji	Typ	Przykład
P	Symbol oznaczający godziny przedpołudniowe i popołudniowe (wielkie litery)	PM
p	Symbol oznaczający godziny przedpołudniowe i popołudniowe (małe litery)	pm
z	Przesunięcie względem czasu GMT w standardzie RFC 822	+0100
Z	Strefa czasowa	CET
s	Liczba sekund, które upłynęły od daty 1970-01-01, 00:00:00 GMT	1196089646
Q	Liczba milisekund, które upłynęły od daty 1970-01-01, 00:00:00	1196089667265

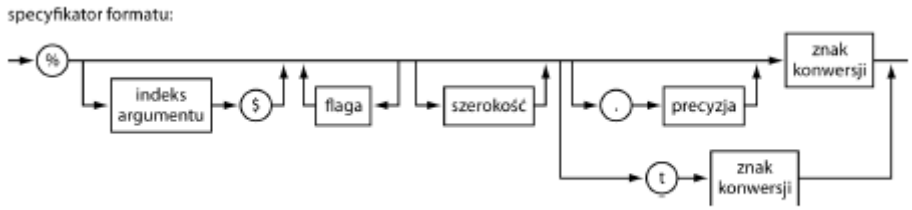
Jak widać w tabeli 3.7, niektóre formaty zwracają tylko określoną część daty, na przykład tylko dzień albo tylko miesiąc. Formatowanie każdej części daty oddzielnie byłoby nierozsądnym rozwiązaniem. Dlatego w łańcuchu formatującym można podać **indeks** argumentu, który ma być sformatowany. Indeks musi się znajdować bezpośrednio po symbolu % i kończyć się symbolem $. Na przykład:

```
System.out.printf("%1$s %2$te %2$tB %2$tY", "Data:", new Date());
```
Wynik wykonania tego wyrażenia będzie następujący:
```
Data: luty 9, 2004
```
Ewentualnie można użyć flagi <. Oznacza ona, że ten sam argument co w poprzedniej specyfikacji formatu powinien zostać użyty ponownie. Poniższa instrukcja:
```
System.out.printf("%s %te %<tB %<tY", "Data:", new Date());
```
da taki sam wynik jak poprzedni fragment kodu.

NB. Wartości indeksów argumentów zaczynają się od 1, nie od 0; zapis %1$... dotyczy pierwszego argumentu. W ten sposób zapobiegnięto myleniu ich z flagą 0.

Przedstawione zostały wszystkie własności metody printf. Rysunek 3.6 prezentuje schemat opisujący składnię specyfikatorów formatu.

NB. Niektóre z zasad formatowania są **związane z określoną lokalizacją**. Na przykład w Niemczech separatorem dziesiętnym jest przecinek, a zamiast „Poniedziałek" wyświetla się „Montag".



Rysunek 3.6. Składnia specyfikatora format

Zapis i odczyt plików

Aby odczytać dane z pliku, należy utworzyć obiekt Scanner:
```
Scanner in = new Scanner(Paths.get("mojplik.txt"));
```
Jeśli nazwa pliku zawiera lewe ukośniki, należy pamiętać o zastosowaniu dla nich symboli zastępczych: "c:\\mojkatalog\\mojplik.txt".
Po wykonaniu tych czynności można odczytać zawartość pliku za pomocą metod klasy Scanner, które były opisywane wcześniej.
Aby zapisać dane do pliku, należy posłużyć się obiektem PrintWriter. Należy podać konstruktorowi nazwę pliku:
```
PrintWriter out = new PrintWriter("mojplik.txt");
```
Jeśli plik nie istnieje, można użyć metod print, println lub printf, podobnie jak w przypadku drukowania do wyjścia System.out.

NB. Obiekt Scanner można utworzyć przy użyciu parametru łańcuchowego, ale parametr ten zostanie zinterpretowany jako dane, a nie nazwa pliku. Jeśli na przykład napiszemy:
Scanner in = new Scanner("mojplik.txt"); *// Błąd?*
obiekt klasy Scanner będzie widział dane składające się z jedenastu znaków: 'm', 'o', 'j' itd. Istnieje duże prawdopodobieństwo, że autorowi kodu chodziło o coś innego.

Jasne jest zatem, że dostęp do plików jest równie łatwy jak używanie wejścia System.in oraz wyjścia System.out. Jest tylko jedno „ale": jeśli obiekt klasy Scanner zostanie utworzony przy użyciu nazwy nieistniejącego pliku albo PrintWriter przy użyciu nazwy, której nie można utworzyć, wystąpi wyjątek. Dla kompilatora Javy wyjątki te mają większe znaczenie niż na przykład wyjątek dzielenia przez zero. Rozmaite techniki obsługi wyjątków zostały opisane w rozdziale 11. Na razie wystarczy, jeśli poinformujemy kompilator, że wiemy, iż istnieje możliwość wystąpienia wyjątku związanego z nieodnalezieniem pliku. Robimy to, dodając do metody main klauzulę throws:

```
public static void main(String[] args) throws FileNotFoundException
{
Scanner in = new Scanner(Paths.get("mojplik.txt"));
. . .
}
```

NB. Względne ścieżki do plików (np. *mojplik.txt*, *mojkatalog/mojplik.txt* lub *../mojplik.txt*) są lokalizowane względem katalogu, w którym uruchomiono maszynę wirtualną. Jeśli uruchomimy program z wiersza poleceń za pomocą polecenia:
java MyProg
katalogiem początkowym będzie aktualny katalog okna konsoli. W zintegrowanym środowisku programistycznym katalog początkowy jest określany przez IDE. Lokalizację tego katalogu można sprawdzić za pomocą poniższego wywołania:
String dir = System.getProperty("user.dir");
Jeśli nie możesz się połapać w lokalizacji plików, możesz zastosować ścieżki bezwzględne, takie jak "c:\\mojkatalog\\mojplik.txt" lub "/home/ja/mojkatalog/mojplik.txt".

Wiemy już, jak odczytywać i zapisywać pliki zawierające dane tekstowe. Bardziej zaawansowane zagadnienia, jak obsługa różnych standardów kodowania znaków, przetwarzanie danych binarnych, odczyt katalogów i zapis plików archiwum zip, zostały opisane w rozdziale

1. drugiego tomu.

NB. Przy uruchamianiu programu w wierszu poleceń można użyć właściwej danemu systemowi składni przekierowywania w celu dodania dowolnego pliku do wejścia System.in i wyjścia System.out:
java MyProg < mojplik.txt > output.txt
Dzięki temu nie trzeba się zajmować obsługą wyjątku FileNotFoundException.

java.util.Scanner **5.0**
- Scanner(Path p)

Tworzy obiekt klasy Scanner, który wczytuje dane z podanej ścieżki.
- Scanner(String data)
Tworzy obiekt klasy Scanner, który wczytuje dane z podanego łańcucha.

java.io.PrintWriter **1.1**
- PrintWriter(String fileName)
Tworzy obiekt PrintWriter, który zapisuje dane do pliku o podanej nazwie.

java.nio.file.Paths **7.0**
- static Path get(String pathname)
Tworzy obiekt Path ze ścieżki o podanej nazwie.

Przepływ sterowania

W Javie, podobnie jak w każdym języku programowania, do kontroli przepływu sterowania używa się instrukcji warunkowych i pętli. Zaczniemy od instrukcji warunkowych, aby później przejść do pętli. Na zakończenie omówimy nieco nieporęczną instrukcję switch, która może się przydać, gdy konieczne jest sprawdzenie wielu wartości jednego wyrażenia.

NB. Instrukcje sterujące Javy są niemal identyczne z instrukcjami sterującymi w C++. Różnica polega na tym, że w Javie nie ma instrukcji go to, ale jest wersja instrukcji break z etykietą, której można użyć do przerwania działania zagnieżdżonej pętli (w takich sytuacjach, w których w C prawdopodobnie użylibyśmy instrukcji go to). Nareszcie dodano wersję pętli for, która nie ma odpowiednika w językach C i C++. Jest podobna do pętli foreach w C#.

Zasięg blokowy

Zanim przejdziemy do instrukcji sterujących, musimy poznać pojęcie **blok**.
Blok, czyli instrukcja złożona, to dowolna liczba instrukcji Javy ujętych w nawiasy klamrowe. Blok określa zasięg zmiennych. Bloki można **zagnieżdżać** w innych blokach. Poniżej znajduje się blok zagnieżdżony w bloku metody main:

```
public static void main(String[] args)
{
int n;
. . .
{
int k;
. . .
} // Definicja zmiennej k jest dostępna tylko do tego miejsca.
}
```

Nie można zdefiniować dwóch zmiennych o takiej samej nazwie w dwóch zagnieżdżonych blokach. Na przykład poniższy kod jest błędny i nie można go skompilować:

```
public static void main(String[] args)
{
int n;
. . .
```

```
{
int k;
int n; // Błąd — nie można ponownie zdefiniować zmiennej n w bloku wewnętrznym.
. . .
}
}
```

NB. W C++ można wewnątrz bloku ponownie zdefiniować zmienną wcześniej zdefiniowaną na zewnątrz tego bloku. Ta definicja wewnętrzna przesłania wtedy definicję zewnętrzną. Może to być jednak źródłem błędów i z tego powodu operacja taka nie jest dozwolona w Javie.

Instrukcje warunkowe

W Javie instrukcja warunkowa ma następującą postać:

if (*warunek*) *instrukcja*

Warunek musi być umieszczony w nawiasach okrągłych.

Podobnie jak w wielu językach, w Javie często po spełnieniu jednego warunku konieczne jest wykonanie wielu instrukcji. W takim przypadku należy zastosować **blok instrukcji** w następującej postaci:

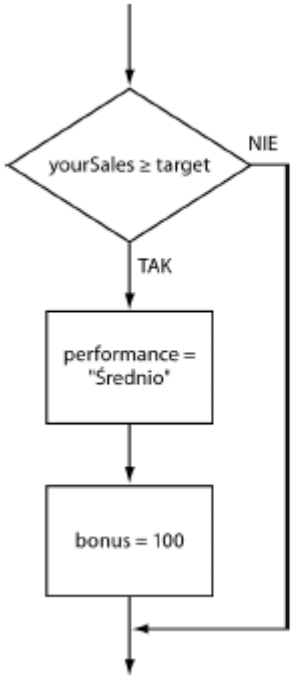
```
{
instrukcja1;
instrukcja2;
}
```

Na przykład:

```
if (yourSales >= target)
{
performance = "Średnio";
bonus = 100;
}
```

Wszystkie instrukcje znajdujące się pomiędzy klamrami zostaną wykonane, jeśli wartość zmiennej yourSales będzie większa lub równa wartości zmiennej target (zobacz rysunek 3.7).

Rysunek 3.7. Diagram przepływu sterowania instrukcji if



NB. Blok (czasami nazywany **instrukcją złożoną**) umożliwia wykonanie więcej niż jednej instrukcji we wszystkich miejscach, gdzie przewiduje się użycie instrukcji.

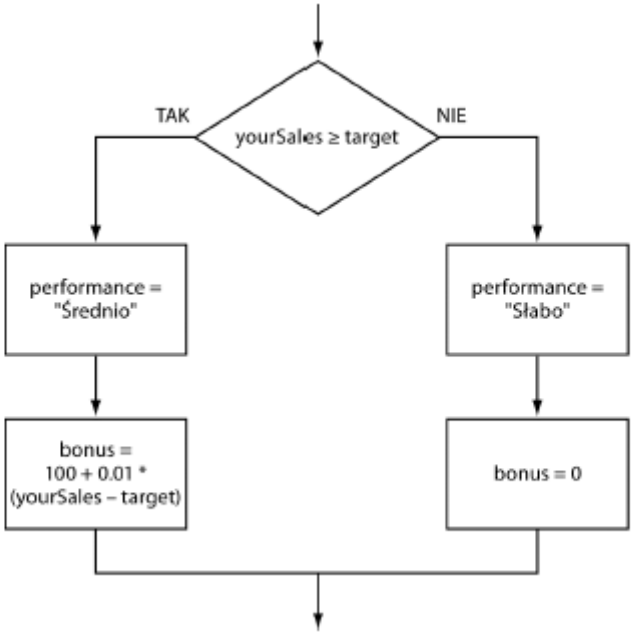
Bardziej ogólna postać instrukcji warunkowej w Javie jest następująca (zobacz rysunek 3.8):

if (*warunek*) *instrukcja*$_1$ else *instrukcja*$_2$

Na przykład:

```
if (yourSales >= target)
{
performance = "Średnio";
bonus = 100 + 0.01 * (yourSales - target);
}
else
{
performance = "Słabo";
bonus = 0;
}
```

Rysunek 3.8. Diagram przepływu sterowania instrukcji if-else



Stosowanie else jest opcjonalne. Dane else zawsze odpowiada najbliższemu poprzedzającemu je if. W związku z tym w instrukcji:

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

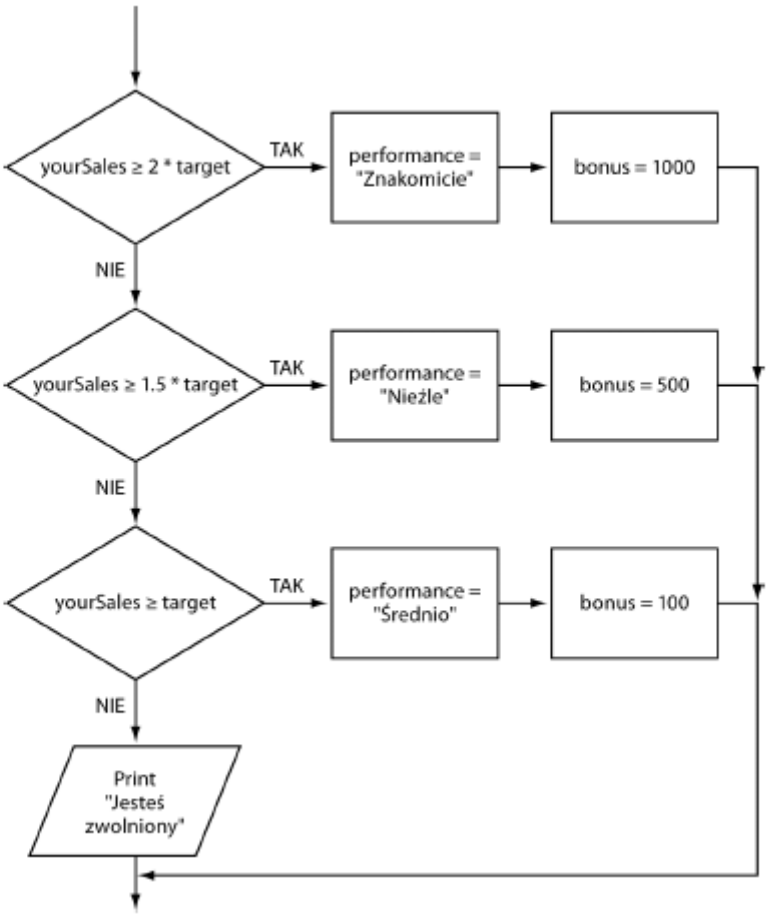
else odpowiada drugiemu if. Oczywiście dobrze by było zastosować klamry, aby kod był bardziej czytelny:

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

Często stosuje się kilka instrukcji else-if jedna po drugiej (zobacz rysunek 3.9). Na przykład:

```
if (yourSales >= 2 * target)
{
performance = "Znakomicie";
bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
performance = "Nieźle";
bonus = 500;
}
else if (yourSales >= target)
{
performance = "Średnio";
bonus = 100;
}
else
{
System.out.println("Jesteś zwolniony");
}
```

Rysunek 3.9. Diagram przepływu sterowania instrukcji if-else if (wiele odgałęzień)



Pętle

Pętla **while** wykonuje instrukcję (albo blok instrukcji) tak długo, jak długo warunek ma wartość **true**. Ogólna postać instrukcji **while** jest następująca:

while (*warunek*) *instrukcja*

Instrukcje pętli **while** nie zostaną nigdy wykonane, jeśli warunek ma wartość **false** na początku (zobacz rysunek 3.10).

Program z listingu 3.3 oblicza, ile czasu trzeba składać pieniądze, aby dostać określoną emeryturę, przy założeniu, że każdego roku wpłacana jest taka sama kwota, i przy określonej stopie oprocentowania wpłaconych pieniędzy.

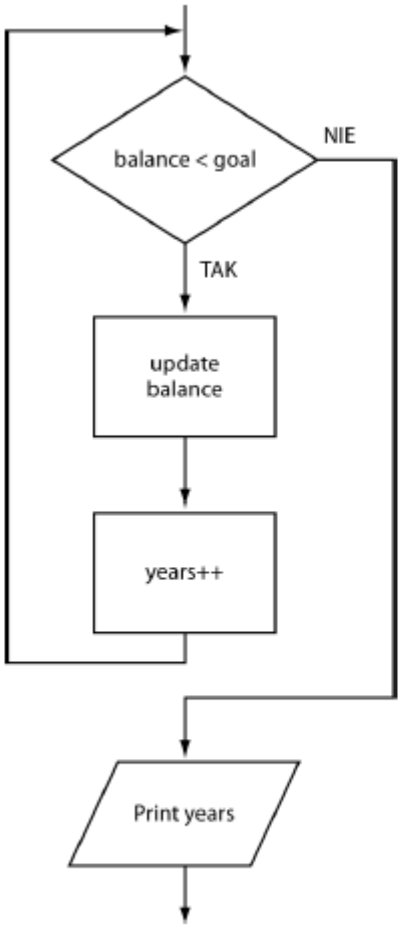
W ciele pętli zwiększamy licznik i aktualizujemy bieżącą kwotę uzbieranych pieniędzy, aż ich suma przekroczy wyznaczoną kwotę.

```
while (balance < goal)
{
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
years++;
}
System.out.println(years + "lat.");
```

(Nie należy ufać temu programowi przy planowaniu emerytury. Pominięto w nim kilka

szczegółów, takich jak inflacja i przewidywana długość życia).

Rysunek 3.10. Diagram przepływu sterowania instrukcji while




```
   // Zapytanie o gotowość do przejścia na emeryturę i pobranie danych.
   . . .
}
while (input.equals("N"));
```

Pętla while sprawdza warunek na samym początku działania. W związku z tym jej instrukcje mogą nie zostać wykonane ani razu. Aby mieć pewność, że instrukcje zostaną wykonane co najmniej raz, sprawdzanie warunku trzeba przenieść na sam koniec. Do tego służy pętla do-while. Jej składnia jest następująca:

do *instrukcja* while (*warunek*)

Ta instrukcja najpierw wykonuje instrukcję (która zazwyczaj jest blokiem instrukcji), a dopiero potem sprawdza warunek. Następnie znowu wykonuje instrukcję i sprawdza warunek itd.

Kod na listingu 3.4 oblicza nowe saldo na koncie emerytalnym, a następnie pyta, czy jesteśmy gotowi przejść na emeryturę:

```
do
{
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
year++;
// Drukowanie aktualnego stanu konta.
. . .
```

```
// Zapytanie o gotowość do przejścia na emeryturę i pobranie danych.
. . .
}
while (input.equals("N"));
```

Pętla jest powtarzana, dopóki użytkownik podaje odpowiedź N (zobacz rysunek 3.11). Ten program jest dobrym przykładem pętli, która musi być wykonana co najmniej jeden raz, ponieważ użytkownik musi zobaczyć stan konta, zanim podejmie decyzję o przejściu na emeryturę.

Listing 3.3. Retirement.java

```
import java.util.*;
/**
* Ten program demonstruje sposób użycia pętli <code>while</code>.
* @version 1.20 2004-02-10
* @author Cay Horstmann
*/
public class Retirement
{

public static void main(String[] args)
{
// Wczytanie danych.
Scanner in = new Scanner(System.in);
System.out.print("Ile pieniędzy potrzebujesz, aby przejść na emeryturę? ");
double goal = in.nextDouble();
System.out.print("Ile pieniędzy rocznie będziesz wpłacać? ");
double payment = in.nextDouble();
System.out.print("Stopa procentowa w %: ");
double interestRate = in.nextDouble();
double balance = 0;
int years = 0;
// Aktualizacja salda konta, jeśli cel nie został osiągnięty.
while (balance < goal)
{
// Dodanie tegorocznych płatności i odsetek.
balance += payment;
double interest = balance * interestRate / 100;

balance += interest;
years++;
}
System.out.println("Możesz przejść na emeryturę za " + years + " lat.");
}
}
```

Listing 3.4. Retirement2.java

```
import java.util.*;
/**
* Ten program demonstruje użycie pętli <code>do/while</code>.
* @version 1.20 2004-02-10
* @author Cay Horstmann
*/
public class Retirement2
{
public static void main(String[] args)
{
Scanner in = new Scanner(System.in);
System.out.print("Ile pieniędzy rocznie będziesz wpłacać? ");
double payment = in.nextDouble();
System.out.print("Stopa oprocentowania w %: ");
double interestRate = in.nextDouble();
double balance = 0;
int year = 0;
String input;
// Aktualizacja stanu konta, kiedy użytkownik nie jest gotowy do przejścia na emeryturę.
do
{
```

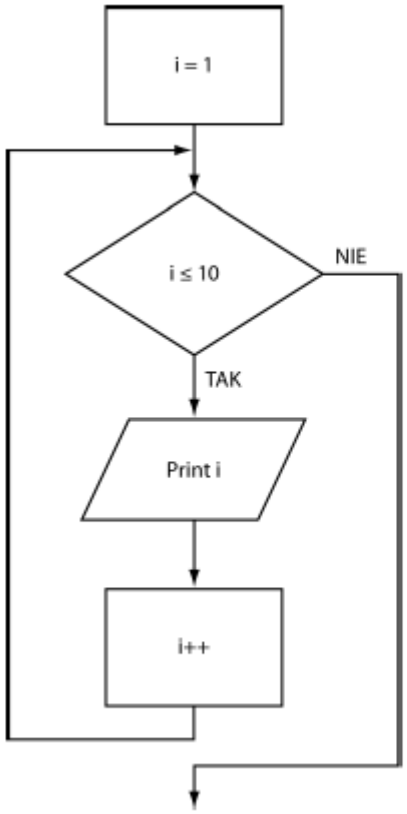
```
// Dodanie tegorocznych płatności i odsetek.
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
year++;
// Drukowanie aktualnego stanu konta.
System.out.printf("Po upływie %d lat stan twojego konta wyniesie %,.2f%n",
year, balance);
// Zapytanie o gotowość do przejścia na emeryturę i pobranie danych.
System.out.print("Chcesz przejść na emeryturę? (T/N) ");
input = in.next();
}
while (input.equals("N"));
}
}
```

Pętle o określonej liczbie powtórzeń

Liczba iteracji instrukcji for jest kontrolowana za pomocą licznika lub jakiejś innej zmiennej, której wartość zmienia się po każdym powtórzeniu. Z rysunku 3.12 wynika, że poniższa pętla drukuje na ekranie liczby od 1 do 10.

```
for (int i = 1; i <= 10; i++)
System.out.println(i);
```

Rysunek 3.12. Diagram przepływu sterowania pętli for



Na pierwszym miejscu z reguły znajduje się inicjacja licznika. Drugie miejsce zajmuje warunek, który jest sprawdzany przed każdym powtórzeniem instrukcji pętli. Na trzeciej pozycji umieszczamy informację na temat sposobu zmiany wartości licznika.

Mimo iż w Javie, podobnie jak w C++, w różnych miejscach pętli for można wstawić prawie każde wyrażenie, niepisana zasada głosi, że do dobrego stylu należy, aby w tych miejscach inicjować, sprawdzać i zmieniać wartość jednej zmiennej. Nie stosując się do tej reguły, można napisać bardzo zagmatwane pętle.
Jednak nawet w granicach dobrego stylu programowania można sobie pozwolić na wiele. Można na przykład utworzyć pętlę zmniejszającą licznik:

```
for (int i = 10; i > 0; i--)
System.out.println("Odliczanie . . . " + i);
System.out.println("Start!");
```

NB. Należy zachować szczególną ostrożność przy porównywaniu w pętli liczb zmiennoprzecinkowych.
Pętla for w takiej postaci:
for (double x = 0; x != 10; x += 0.1) . . .
może się nigdy nie skończyć. Wartość końcowa nie zostanie osiągnięta ze względu na błąd związany z zaokrąglaniem. Na przykład w powyższej pętli wartość x przeskoczy z wartości 9.99999999999998 na 10.09999999999998, ponieważ liczba 0,1 nie ma dokładnej reprezentacji binarnej.

Zmienna zadeklarowana na pierwszej pozycji w pętli for ma zasięg do końca ciała tej pętli.

```
for (int i = 1; i <= 10; i++)
{
. . .
}
// Tutaj zmienna i już nie jest dostępna.
```

Innymi słowy, wartość zmiennej zadeklarowanej w wyrażeniu pętli for nie jest dostępna poza tą pętlą. W związku z tym, aby móc użyć wartości licznika pętli poza tą pętlą, trzeba go zadeklarować poza jej nagłówkiem!

```
int i;
for (i = 1; i <= 10; i++)
{
. . .
}
// Zmienna i tutaj też jest dostępna.
```

Z drugiej jednak strony w kilku pętlach for można zdefiniować zmienną o takiej samej nazwie:

```
for (int i = 1; i <= 10; i++)
{
. . .
}
. . .
for (int i = 11; i <= 20; i++) // W tym miejscu dozwolona jest ponowna deklaracja zmiennej
i.
{
. . .
}
```

Pętla for jest krótszym sposobem zapisu pętli while. Na przykład:

```
for (int i = 10; i > 0; i--)
System.out.println("Odliczanie... " + i);
```

można zapisać następująco:

```
int i = 10;
while (i > 0)
{
System.out.println("Odliczanie... " + i);
i--;
}
```

Listing 3.5 przedstawia typowy przykład zastosowania pętli for.
Ten program oblicza szanse wygrania na loterii. Jeśli na przykład loteria polega na wybraniu sześciu liczb z przedziału 1 – 50, to istnieje (50*49*48*47*46*45)/(1*2*3*4*5*6) możliwych kombinacji, co oznacza, że nasze szanse są jak 1 do 15 890 700. Powodzenia!
W ogólnym przypadku losowania k liczb ze zbioru n istnieje:

$$\frac{n*(n-1)*(n-2)*...*(n-k+1)}{1*2*3*...*k}$$

możliwych wyników. Poniższa pętla for oblicza tę wartość:

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

Listing 3.5. LotteryOdds.java

```
import java.util.*;
/**
* Ten program demonstruje zastosowanie pętli <code>for</code>.
* @version 1.20 2004-02-10
* @author Cay Horstmann
*/
public class LotteryOdds
{
public static void main(String[] args)
{
Scanner in = new Scanner(System.in);
System.out.print("Ile liczb ma być wylosowanych? ");
int k = in.nextInt();
System.out.print("Jaka jest górna granica przedziału losowanych liczb? ");
int n = in.nextInt();
/*
* Obliczanie współczynnika dwumianowego n*(n–1)*(n–2)*...*(n–k+1)/(1*2*3*...*k)
*/
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
lotteryOdds = lotteryOdds * (n - i + 1) / i;
System.out.println("Twoje szanse to 1 do " + lotteryOdds + ". Powodzenia!");
}
}
```

Wybór wielokierunkowy — instrukcja switch

W sytuacjach gdy jest dużo opcji do wyboru, instrukcja warunkowa if-else może być mało efektywna. Dlatego w Javie udostępniono instrukcję switch, która niczym nie różni się od swojego pierwowzoru w językach C i C++.
Na przykład do utworzenia systemu menu zawierającego cztery opcje, jak ten na rysunku 3.13, można użyć kodu podobnego do tego poniżej:

```
Scanner in = new Scanner(System.in);
System.out.print("Wybierz opcję (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
case 1:
. . .
break;
case 2:
. . .
break;
case 3:
```

```
. . .
break;
case 4:
. . .
break;
default:
// Nieprawidłowe dane.
. . .
break;
}
```

Wykonywanie programu zaczyna się od etykiety case, która pasuje do wybranej opcji, i jest kontynuowane do napotkania instrukcji break lub końca instrukcji switch. Jeśli żadna z etykiet nie zostanie dopasowana, nastąpi wykonanie części oznaczonej przez etykietę default — jeśli taka istnieje.

Etykiety case mogą być:

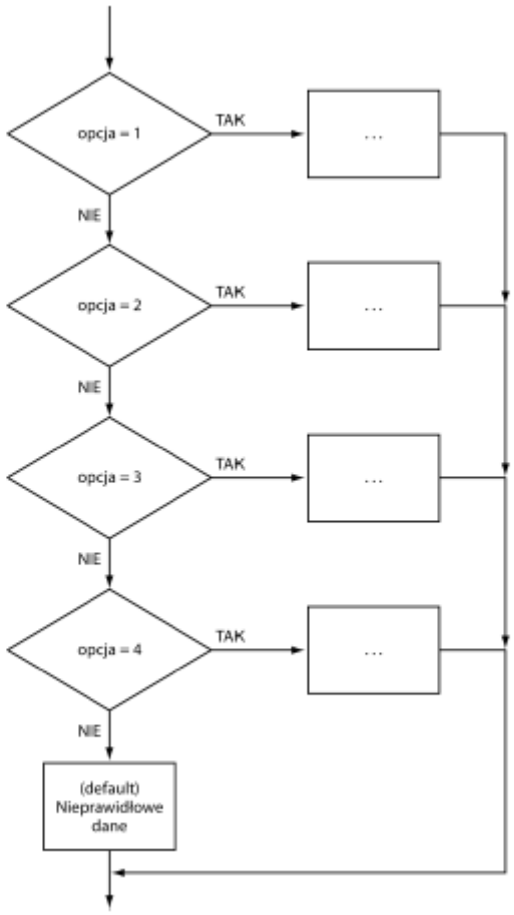
- wyrażeniami stałymi typu char, byte, short lub int (oraz odpowiednich klas opakowujących: Character, Byte, Short i Integer);
- stałymi wyliczeniowymi;
- łańcuchami od Java SE 7.

Na przykład:

```
String input = . . .;
switch (input.toLowerCase())
{
case "tak": // OK od Java SE 7

. . .
break;
. . .
}
```

Rysunek 3.13. Diagram przepływu sterowania instrukcji switch



Używając instrukcji switch ze stałymi wyliczeniowymi, nie ma konieczności podawania nazwy wyliczenia w każdej etykiecie — jest ona pobierana domyślnie z wartości switch. Na przykład:

```
Size sz = . . .;
switch (sz)
{
case SMALL: // Nie trzeba było pisać Size.SMALL.
. . .
break;
. . .
}
```

NB. Istnieje ryzyko, że zostanie uruchomionych kilka opcji. Jeśli przez przypadek na końcu jednej z opcji nie znajdzie się instrukcja break, sterowanie zostanie przekazane do kolejnej etykiety case! Taki sposób działania jest niebezpieczny i często prowadzi do błędów. Z tego powodu nigdy nie używamy instrukcji case w swoich programach. Jeśli jednak czujesz do instrukcji switch większą sympatię niż my, możesz kompilować swoje programy z użyciem opcji -Xlint:fallthrough:

```
javac -Xlint:fallthrough Test.java
```

Dzięki temu ustawieniu kompilator będzie zgłaszał wszystkie przypadki alternatyw niezawierających na końcu instrukcji break.
Gdy będziesz chciał wykonać bloki case po kolei, oznacz otaczającą je metodę adnotacją @SuppressWarnings("fallthrough"). Dzięki temu dla tej metody nie będą zgłaszane ostrzeżenia. (Adnotacje to technika przekazywania informacji do kompilatora lub innego narzędzia przetwarzającego kod źródłowy Java lub pliki klas.

Instrukcje przerywające przepływ sterowania

Mimo że projektanci języka Java zarezerwowali słowo goto, nie zdecydowali się wcielić go do języka. Instrukcje goto są uważane za wyznacznik słabego stylu programowania. Zdaniem niektórych programistów kampania skierowana przeciwko instrukcji goto jest przesadzona (zobacz słynny artykuł Donalda E. Knutha pod tytułem *Structured Programming with goto statements*). Ich zdaniem stosowanie instrukcji goto bez ograniczeń może prowadzić do wielu błędów, ale użycie jej od czasu do czasu w celu **wyjścia z pętli** może być korzystne. Projektanci Javy przychylili się do tego stanowiska i dodali nową instrukcję break z etykietą. Przyjrzyjmy się najpierw instrukcji break bez etykiety. Tej samej instrukcji break, za pomocą której wychodzi się z instrukcji switch, można użyć do przerwania działania pętli. Na przykład:

```
while (years <= 100)
{
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
if (balance >= goal) break;
years++;
}
```

Wyjście z pętli nastąpi, kiedy wartość znajdującej się na samej górze pętli zmiennej years przekroczy 100 albo znajdująca się w środku zmienna balance będzie miała wartość większą lub równą goal. Oczywiście tę samą wartość zmiennej years można by było obliczyć bez użycia instrukcji break:

```
while (years <= 100 && balance < goal)
{
balance += payment;
double interest = balance * interestRate / 100;
balance += interest;
if (balance < goal)
years++;
}
```

Należy jednak zauważyć, że wyrażenie sprawdzające balance < goal jest w tej wersji użyte dwukrotnie. Aby uniknąć tego powtórzenia, niektórzy programiści stosują instrukcję break.
W Javie dostępna jest też **instrukcja break z etykietą** (brak jej natomiast w języku C++), która umożliwia wyjście z kilku zagnieżdżonych pętli. Czasami w głęboko zagnieżdżonych pętlach dzieją się dziwne rzeczy. W takiej sytuacji najlepiej jest wyjść całkiem na zewnątrz. Zaprogramowanie takiego działania za pomocą dodatkowych warunków w różnych testach pętli jest rozwiązaniem mało wygodnym.
Poniżej znajduje się przykładowy kod prezentujący działanie instrukcji break. Należy zauważyć, że etykieta musi się znajdować przed najbardziej zewnętrzną pętlą, z której chcemy wyjść. Ponadto po etykiecie w tym miejscu musi się znajdować dwukropek.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .) // Ta pętla jest opatrzona etykietą.
{
. . .
for (. . .) // Ta zagnieżdżona pętla nie ma etykiety.
{
System.out.print("Podaj liczbę >= 0: ");
n = in.nextInt();
if (n < 0) // To nie powinno mieć miejsca — nie można kontynuować.
break read_data;
```

```
// Wyjście z pętli z etykietą read_data.
. . .
}
}
// Ta instrukcja jest wykonywana bezpośrednio po przerwaniu pętli.
if (n < 0) // Sprawdzenie, czy ma miejsce niepożądana sytuacja.
{
// Obsługa niechcianej sytuacji.
}
else
{
// Wykonywanie w normalnym toku.
}
```

Jeśli zostaną podane nieprawidłowe dane, instrukcja break z etykietą przeniesie sterowanie do miejsca bezpośrednio za blokiem opatrzonym tą etykietą. Następnie, tak jak w każdym przypadku użycia instrukcji break, trzeba sprawdzić, czy wyjście z pętli nastąpiło w toku normalnego działania, czy zostało spowodowane przez instrukcję break.

NB. Co ciekawe, etykietę można dodać do każdej instrukcji, nawet instrukcji warunkowej if i instrukcji blokowej:

```
etykieta:
{
. . .
if (warunek) break etykieta; // Wychodzi z bloku.
. . .
}
// Przechodzi do tego miejsca, jeśli zostanie wykonana instrukcja break.
```

W związku z tym, jeśli tęsknisz za instrukcją goto i możesz umieścić blok bezpośrednio przed miejscem, do którego ma nastąpić przejście, możesz użyć instrukcji break! Oczywiście nie polecamy tej metody programowania. Zauważ też, że przejście jest możliwe tylko w jedną stronę — nie można **wskoczyć do bloku**.

Na zakończenie została jeszcze instrukcja continue, która podobnie jak instrukcja break zmienia normalny przepływ sterowania. Instrukcja continue przenosi sterowanie do nagłówka najgłębiej zagnieżdżonej pętli. Na przykład:

```
Scanner in = new Scanner(System.in);
while (sum < goal)
{
System.out.print("Podaj liczbę: ");
n = in.nextInt();
if (n < 0) continue;
sum += n; // Wyrażenie nie zostanie wykonane, jeśli n < 0.
}
```

Jeśli wartość zmiennej n jest mniejsza od 0, instrukcja continue powoduje natychmiastowe przejście do nagłówka pętli, nie dopuszczając do wykonania reszty instrukcji w bieżącej iteracji.

Instrukcja continue użyta w pętli for powoduje przejście do części aktualizującej wartość zmiennej w nagłówku tej pętli. Przyjrzyjmy się następującemu przykładowi:

```
for (count = 1; count <= 100; count++)
{
System.out.print("Podaj liczbę (-1 kończy działanie programu): ");
n = in.nextInt();
if (n < 0) continue;
sum += n; // Wyrażenie nie zostanie wykonane, jeśli n < 0.
}
```

Jeśli n < 0, instrukcja continue powoduje przekazanie sterowania do instrukcji i++.
Istnieje także wersja instrukcji continue z etykietą, która powoduje przejście do nagłówka pętli z odpowiednią etykietą.

NB. Wielu programistów myli instrukcje break i continue. Ich stosowanie nie jest obowiązkowe i to, co można osiągnąć przy ich użyciu, da się zawsze uzyskać w inny sposób. W tej książce nigdy nie używamy instrukcji break i continue.