Zdobądź wiedzę i umiejętności, które pomogą Ci tworzyć efektywne aplikacje dla systemu Android!

Java

Przygotowanie do programowania na platformę Android

Jeff Friesen

Odkryj Javę — od podstaw po zaawansowane mechanizmy tego języka

Dowiedz się, jak wykorzystać potencjał API platformy

Poznaj wszystkie aspekty programowania niezbędne do tworzenia aplikacji na urządzenia przenośne

Apress®

Helion

Tytuł oryginału: Learn Java for Android Development

Tłumaczenie: Daniel Kaczmarek (wstęp, rozdz. 1 – 7, dod. A)
Aleksander Lamża (rozdz. 8 – 10, dod. A)

ISBN: 978-83-246-3372-2

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jappan.zip

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jappan
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 6

Podstawowe interfejsy API — część I

Poważni programiści aplikacji dla systemu Android muszą gruntownie znać najważniejsze interfejsy API języka Java. Z kilkoma API mieliśmy już do czynienia, dość wspomnieć klasy `Object` i `String` oraz hierarchię klas `Throwable`. W tym rozdziale przedstawimy kolejne podstawowe interfejsy API przeznaczone do wykonywania obliczeń matematycznych, operowania na pakietach i typach podstawowych, a także mechanizm odśmiecania.

■ **Uwaga** • W rozdziale 6. zostaną opisane podstawowe klasy i interfejsy API zlokalizowane w pakietach `java.lang`, `java.lang.ref` i `java.math`.

Interfejsy API do wykonywania obliczeń matematycznych

W rozdziale 2. zaprezentowano operatory +, -, *, / i % języka Java przeznaczone do wykonywania najważniejszych operacji matematycznych na wartościach typów podstawowych. Java udostępnia także klasy przeznaczone do wykonywania operacji trygonometrycznych i innych zaawansowanych działań, precyzyjnej prezentacji wartości pieniężnych oraz obsługi bardzo długich liczb całkowitych wykorzystywanych do szyfrowania algorytmem RSA (*http://pl.wikipedia.org/wiki/RSA_(kryptografia)*) i w innych kontekstach.

Klasy Math i StrictMath

Klasa `java.lang.Math` deklaruje stałe typu `double` o nazwach `E` i `PI`, które reprezentują odpowiednio podstawę logarytmu naturalnego (`2,71828...`) oraz stosunek obwodu okręgu do jego średnicy (`3,141519...`). Stała `E` jest inicjalizowana wartością `2,718281828459045`, natomiast stała `PI` ma wartość `3,141592653589793`. W klasie `Math` zadeklarowane są także wybrane metody klasy, przeznaczone do wykonywania różnego rodzaju działań matematycznych. W tabeli 6.1 opisano większość dostępnych metod.

Tabela 6.1. *Metody klasy Math*

Metoda	Opis
`double abs(double d)`	Zwraca wartość bezwzględną liczby `d`. Istnieją cztery przypadki szczególne: `abs(-0.0) = +0.0`, `abs(+nieskończoność) = +nieskończoność`, `abs(-nieskończoność) = +nieskończoność` oraz `abs(NaN) = NaN`.
`float abs(float f)`	Zwraca wartość bezwzględną liczby `f`. Istnieją cztery przypadki szczególne: `abs(-0.0) = +0.0`, `abs(+nieskończoność) = +nieskończoność`, `abs(-nieskończoność) = +nieskończoność` oraz `abs(NaN) = NaN`.
`int abs(int i)`	Zwraca wartość bezwzględną liczby `i`. Istnieje jeden przypadek szczególny: wartością bezwzględną `Integer.MIN_VALUE` jest `Integer.MIN_VALUE`.
`long abs(long l)`	Zwraca wartość bezwzględną liczby `l`. Istnieje jeden przypadek szczególny: wartością bezwzględną `Long.MIN_VALUE` jest `Long.MIN_VALUE`.
`double acos(double d)`	Zwraca arcus cosinus kąta `d` z przedziału od 0 do `PI`. Istnieją trzy przypadki szczególne: `acos(wartość > 1) = NaN`, `acos(wartość < -1) = NaN` oraz `acos(NaN) = NaN`.
`double asin(double d)`	Zwraca arcus sinus kąta `d` z przedziału od `–PI/2` do `PI/2`. Istnieją trzy przypadki szczególne: `asin(wartość > 1) = NaN`, `asin(wartość < -1) = NaN` oraz `asin(NaN) = NaN`.
`double atan(double d)`	Zwraca arcus tangens kąta `d` z przedziału `–PI/2` do `PI/2`. Istnieje pięć przypadków szczególnych: `atan(+0.0) = +0.0`, `atan(-0.0) = -0.0`, `atan(+nieskończoność) = +PI/2`, `atan(-nieskończoność) = -PI/2` oraz `atan(NaN) = NaN`.
`double ceil(double d)`	Zwraca najmniejszą wartość (najbliższą minus nieskończoności), która nie jest mniejsza od `d` i jest liczbą całkowitą. Istnieje sześć przypadków szczególnych: `ceil(+0.0) = +0.0`, `ceil(-0.0) = -0.0`, `ceil(wartość > -1.0 oraz < 0.0) = -0.0`, `ceil(+nieskończoność) = +nieskończoność`, `ceil(-nieskończoność) = -nieskończoność` oraz `ceil(NaN) = NaN`.
`double cos(double d)`	Zwraca cosinus kąta `d` (wyrażonego w radianach). Istnieją trzy przypadki szczególne: `cos(+nieskończoność) = NaN`, `cos(-nieskończoność) = NaN` oraz `cos(NaN) = NaN`.
`double exp(double d)`	Zwraca liczbę Eulera e podniesioną do potęgi `d`. Istnieją trzy przypadki szczególne: `exp(+nieskończoność) = + nieskończoność`, `exp(-nieskończoność) = +0.0` oraz `exp(NaN) = NaN`.
`double floor(double d)`	Zwraca największą wartość (najbliższą plus nieskończoności), która nie jest większa od `d` i jest liczbą całkowitą. Istnieje pięć przypadków szczególnych: `floor(+0.0) = +0.0`, `floor(-0.0) = -0.0`, `floor(+nieskończoność) = +nieskończoność`, `floor(-nieskończoność) = -nieskończoność` oraz `floor(NaN) = NaN`.
`double log(double d)`	Zwraca logarytm naturalny (przy podstawie e) z liczby `d`. Istnieje sześć przypadków szczególnych: `log(+0.0) = - nieskończoność`, `log(-0.0) = - nieskończoność`, `log(wartość < 0) = NaN`, `log(+nieskończoność) = +nieskończoność`, `log(-nieskończoność) = NaN` oraz `log(NaN) = NaN`.

Tabela 6.1. *Metody klasy Math — ciąg dalszy*

Metoda	Opis
`double log10(double d)`	Zwraca logarytm przy podstawie 10 z liczby `d`. Istnieje sześć przypadków szczególnych: `log10(+0.0) = -nieskończoność`, `log10(-0.0) = -nieskończoność`, `log10(wartość < 0) = NaN`, `log10(+nieskończoność) = +nieskończoność`, `log10(-nieskończoność) = NaN` oraz `log10(NaN) = NaN`.
`double max(double d1, double d2)`	Zwraca największą (najbliższą plus nieskończoności) spośród liczb `d1` i `d2`. Istnieją cztery przypadki szczególne: `max(NaN, wartość) = NaN`, `max(wartość, NaN) = NaN`, `max(+0.0, -0.0) = +0.0` oraz `max(-0.0, +0.0) = +0.0`.
`float max(double f1, double f2)`	Zwraca największą (najbliższą plus nieskończoności) spośród liczb `f1` i `f2`. Istnieją cztery przypadki szczególne: `max(NaN, wartość) = NaN`, `max(wartość, NaN) = NaN`, `max(+0.0, -0.0) = +0.0` oraz `max(-0.0, +0.0) = +0.0`.
`int max(int i1, int i2)`	Zwraca największą (najbliższą plus nieskończoności) spośród liczb `i1` i `i2`.
`long max(long l1, long l2)`	Zwraca największą (najbliższą plus nieskończoności) spośród liczb `l1` i `l2`.
`double min(double d1, double d2)`	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb `d1` i `d2`. Istnieją cztery przypadki szczególne: `min(NaN, wartość) = NaN`, `min(wartość, NaN) = NaN`, `min(+0.0, -0.0) = -0.0` oraz `min(-0.0, +0.0) = -0.0`.
`float min(float f1, float f2)`	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb `f1` i `f2`. Istnieją cztery przypadki szczególne: `min(NaN, wartość) = NaN`, `min(wartość, NaN) = NaN`, `min(+0.0, -0.0) = -0.0` oraz `min(-0.0, +0.0) = -0.0`.
`int min(int i1, int i2)`	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb `i1` i `i2`.
`long min(long l1, long l2)`	Zwraca najmniejszą (najbliższą minus nieskończoności) spośród liczb `l1` i `l2`.
`double random()`	Zwraca liczbę pseudolosową z przedziału prawostronnie otwartego od 0,0 (włącznie) do 1,0.
`long round(double d)`	Zwraca wynik zaokrąglenia liczby `d` do długiej liczby całkowitej. Wynik jest równoważny wynikowi wyrażenia `(long) Math.floor(d+0.5)`. Istnieje siedem przypadków szczególnych: `round(+0.0) = +0.0`, `round(-0.0) = +0.0`, `round(wartość > Long.MAX_VALUE) = Long.MAX_VALUE`, `round(wartość < Long.MIN_VALUE) = Long.MIN_VALUE`, `round(+nieskończoność) = Long.MAX_VALUE`, `round(-nieskończoność) = Long.MIN_VALUE` oraz `round(NaN) = +0.0`.
`int round(float f)`	Zwraca wynik zaokrąglenia liczby `f` do liczby całkowitej. Wynik jest równoważny wynikowi wyrażenia `(long) Math.floor(f+0.5)`. Istnieje siedem przypadków szczególnych: `round(+0.0) = +0.0`, `round(-0.0) = +0.0`, `round(wartość > Integer.MAX_VALUE) = Integer.MAX_VALUE`, `round(wartość < Integer.MIN_VALUE) = Integer.MIN_VALUE`, `round(+nieskończoność) = Integer.MAX_VALUE`, `round(-nieskończoność) = Integer.MIN_VALUE` oraz `round(NaN) = +0.0`.
`double signum(double d)`	Zwraca znak liczby `d` jako liczbę –1,0 (jeżeli `d` jest mniejsze od 0,0), liczbę 0,0 (jeżeli `d` jest równe 0,0) lub 1,0 (jeżeli `d` jest większe niż 0,0). Istnieje pięć przypadków szczególnych: `signum(+0.0) = +0.0`, `signum(-0.0) = -0.0`, `signum(+nieskończoność) = +1.0`, `signum(-nieskończoność) = -1.0` oraz `signum(NaN) = NaN`.

Tabela 6.1. *Metody klasy Math — ciąg dalszy*

Metoda	Opis
`float signum(float f)`	Zwraca znak liczby `f` jako liczbę –1,0 (jeżeli `f` jest mniejsze od 0,0), liczbę 0,0 (jeżeli `f` jest równe 0,0) lub 1,0 (jeżeli `f` jest większe niż 0,0). Istnieje pięć przypadków szczególnych: `signum(+0.0) = +0.0`, `signum(-0.0) = -0.0`, `signum(+nieskończoność) = +1.0`, `signum(-nieskończoność) = -1.0` oraz `signum(NaN) = NaN`.
`double sin(double d)`	Zwraca sinus kąta `d` (wyrażonego w radianach). Istnieje pięć przypadków szczególnych: `sin(+0.0) = +0.0`, `sin(-0.0) = -0.0`, `sin(+nieskończoność) = NaN`, `sin(-nieskończoność) = NaN` oraz `sin(NaN) = NaN`.
`double sqrt(double d)`	Zwraca pierwiastek kwadratowy liczby `d`. Istnieje pięć przypadków szczególnych: `sqrt(+0.0) = +0.0`, `sqrt(-0.0) = -0.0`, `sqrt(wartość < 0) = NaN`, `sqrt(+nieskończoność) = +nieskończoność` oraz `sqrt(NaN) = NaN`.
`double tan(double d)`	Zwraca tangens kąta `d` (wyrażonego w radianach). Istnieje pięć przypadków szczególnych: `tan(+0.0) = +0.0`, `tan(-0.0) = -0.0`, `tan(+nieskończoność) = NaN`, `tan(-nieskończoność) = NaN` oraz `tan(NaN) = NaN`.
`double toDegrees (double angrad)`	Przekształca miarę kąta `angrad` z radianów na stopnie za pomocą wyrażenia `angrad*180/PI`. Istnieje pięć przypadków szczególnych: `toDegrees(+0.0) = +0.0`, `toDegrees(-0.0) = -0.0`, `toDegrees(+nieskończoność) = +nieskończoność`, `toDegrees(-nieskończoność) = -nieskończoność` oraz `toDegrees(NaN) = NaN`.
`double toRadians (angdeg)`	Przekształca miarę kąta `angdeg` ze stopni na radiany za pomocą wyrażenia `angdeg/180*PI`. Istnieje pięć przypadków szczególnych: `toRadians(+0.0) = +0.0`, `toRadians(-0.0) = -0.0`, `toRadians(+nieskończoność) = +nieskończoność`, `toRadians(-nieskończoność) = -nieskończoność` oraz `toRadians(NaN) = NaN`.

W tabeli 6.1 przedstawiono obszerny zbiór metod przydatnych do wykonywania działań matematycznych. Na przykład każda metoda `abs` zwraca **wartość bezwzględną** (czyli liczbę bez względu na znak) przekazanego do niej argumentu.

Metody `abs(double)` oraz `abs(float)` przydają się do bezpiecznego porównywania liczb zmiennopozycyjnych o podwójnej precyzji oraz liczb zmiennopozycyjnych. Na przykład wyrażenie `0.3 == 0.1+0.1+0.1` ma wartość `false`, ponieważ liczba 0,1 nie ma dokładnej reprezentacji. Wyrażenia te można jednak ze sobą porównać przy użyciu metody `abs()` i wartości tolerancji, która wskazuje akceptowalny poziom błędu. Na przykład wyrażenie `Math.abs ↪(0.3-(0.1+0.1+0.1)) < 0.1` będzie już mieć wartość `true`, ponieważ bezwzględna różnica między `0.3` i `0.1+0.1+0.1` jest mniejsza niż wartość tolerancji 0,1.

We wcześniejszych rozdziałach przedstawiono inne metody klasy `Math`. Na przykład w rozdziale 2. zostały wykorzystane metody `sin()`, `toRadians()`, `cos()`, `round(double)` i `random()` tej klasy.

Jak widać na przykładzie aplikacji `Lotto649` z rozdziału 5., funkcja `random()` (która zwraca liczbę wyglądającą na losową, choć w rzeczywistości jest wyznaczana przez określoną funkcję matematyczną i dlatego jest tak naprawdę **liczbą pseudolosową**) przydaje się do symulacji, gier i w innych zastosowaniach, w których potrzeba losowości. Najpierw jednak

liczbę zwracaną przez `random()`, która należy do przedziału od 0,0 do prawie 1,0, trzeba jakoś przekształcić do wartości bardziej przydatnej, na przykład należącej do przedziału od 0 do 49 albo od –100 do 100. Na listingu 6.1 znajduje się metoda `rnd()`, przydatna do wykonywania tego typu przekształceń.

Listing 6.1. *Przekształcanie wartości zwracanej przez random() do bardziej przydatnej wartości*

```
public static int rnd(int limit)
{
   return (int) (Math.random()*limit);
}
```

Metoda `rnd()` przekształca zwracaną przez `random()` liczbę zmiennopozycyjną o podwójnej precyzji z przedziału od 0,0 do 1,0 do liczby całkowitej z przedziału od 0 do `limit – 1`. Na przykład `rnd(50)` zwróci liczbę całkowitą z przedziału od 0 do 49. Z kolei instrukcja `-100+rnd(201)` przekształci przedział od 0,0 do prawie 1,0 do przedziału od –100 do 100 przez dodanie odpowiedniej wartości przesunięcia i wykorzystanie odpowiedniej wartości `limit`.

■ **Ostrzeżenie** • Nie należy wykonywać instrukcji `(int) Math.random()*limit`, ponieważ wyrażenie to zawsze będzie mieć wartość 0. W wyrażeniu ułamkowa liczba zmiennopozycyjna o podwójnej precyzji z przedziału od 0,0 do 0,99999… najpierw jest rzutowana na liczbę całkowitą 0 przez ucięcie części ułamkowej, a następnie 0 jest mnożone przez `limit`, co w efekcie daje również 0.

W tabeli 6.1 opisano także przypadki szczególne, dotyczące najczęściej wartości +nieskończoność, –nieskończoność, +0.0, –0.0 i NaN (ang. *Not a Number* — wartość, która nie jest liczbą).

Wynikiem obliczeń zmiennopozycyjnych wykonywanych w języku Java mogą być wartości +nieskończoność, –nieskończoność, +0.0, –0.0 i NaN, ponieważ Java w dużej mierze jest zgodna ze standardem IEEE 754 (*http://pl.wikipedia.org/wiki/IEEE_754*), który opisuje sposób wykonywania obliczeń zmiennopozycyjnych. Poniżej przedstawiono okoliczności, w których pojawiają się wspomniane wartości specjalne:

- +nieskończoność jest zwracana jako wynik dzielenia liczby dodatniej przez 0,0. Na przykład instrukcja `System.out.println(1.0/0.0);` zwróci wartość `Infinity`.
- –nieskończoność jest zwracana jako wynik dzielenia liczby ujemnej przez 0,0. Na przykład instrukcja `System.out.println(-1.0/0.0);` zwróci wartość `–Infinity`.
- `NaN` jest zwracana jako wynik dzielenia 0,0 przez 0,0, wartość pierwiastka kwadratowego liczby ujemnej oraz wynik innych dziwnych operacji. Na przykład instrukcje `System.out.println(0.0/0.0);` oraz `System.out.println(Math.sqrt(-1.0));` zwracają wynik `NaN`.
- `+0.0` jest zwracana jako wynik dzielenia liczby całkowitej przez +nieskończoność. Na przykład instrukcja `System.out.println(1.0/(1.0/0.0));` zwróci wartość `+0.0`.
- `-0.0` jest zwracana jako wynik dzielenia liczby ujemnej przez +nieskończoność. Na przykład instrukcja `System.out.println(-1.0/(1.0/0.0));` zwróci wartość `-0.0`.

Jeżeli wynikiem jakiegoś działania jest wartość specjalna +nieskończoność, –nieskończoność lub NaN, wówczas całe wyrażenie ma zwykle taki sam wynik równy wartości specjalnej. Na przykład wynikiem instrukcji `System.out.println(1.0/0.0*20.0);` jest `Infinity`. Jednak wyrażenie, w którym pojawia się wartość +nieskończoność lub –nieskończoność, może też zwrócić wartość NaN. Na przykład w wyrażeniu `1.0/0.0*0.0` najpierw pojawia się +nieskończoność (jako wynik wyrażenia `1.0/0.0`), a następnie NaN (jako wynik wyrażenia +nieskończoność*0, 0).

Kolejna ciekawostka wiąże się z wartościami `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Long.MAX_VALUE` oraz `Long.MIN_VALUE`. Każda z tych wartości jest klasą opakowującą typu podstawowego, która identyfikuje wartość maksymalną lub minimalną, jaka może być reprezentowana przez typ podstawowy skojarzony z klasą.

Można się także zastanawiać, dlaczego nie istnieją przeciążone wersje metod `abs()`, `max()` i `min()`, które obsługują argumenty typu `byte` i `short`, czyli `byte abs(byte b)` oraz `short abs(short s)`. Wersje takie nie są jednak potrzebne, ponieważ ograniczone przedziały liczb całkowitych typu `byte` i `short` sprawiają, że są one mało przydatne do wykonywania obliczeń. Jeżeli jednak metody takie są potrzebne, można skorzystać z implementacji przedstawionej na listingu 6.2.

Listing 6.2. *Przeciążone metody byte abs(byte b) i short abs(short s)*

```
public static byte abs(byte b)
{
   return (b < 0) ? (byte) -b : b;
}
public static short abs(short s)
{
   return (s < 0) ? (short) -s : s;
}
public static void main(String[] args)
{
   byte b = -2;
   System.out.println(abs(b)); // Wynik: 2
   short s = -3;
   System.out.println(abs(s)); // Wynik: 3
}
```

Rzutowania `(byte)` i `(short)` trzeba wykonać, ponieważ wyrażenie `–b` przekształca wartość zmiennej `b` z typu `byte` na typ `int`, a wyrażenie `–s` przekształca wartość zmiennej `s` z typu `short` na typ `int`. Z kolei rzutowania nie są potrzebne w wyrażeniach `(b < 0)` i `(s < 0)`, ponieważ w tych przypadkach wartości `b` i `s` są automatycznie rzutowane do typu `int` przed porównaniem ich z wartością 0 typu `int`.

Wskazówka • Brak wspomnianych wersji metod w klasie `Math` mógłby sugerować, że typy `byte` i `short` nie są zbyt przydatne w deklaracjach metod. Jednak typy te przydają się wówczas, gdy deklaruje się tablice, których elementy przechowują małe wartości (na przykład wartości bajtowe w pliku binarnym). Gdyby tablica do przechowywania takich wartości została zadeklarowana jako typu `int` lub `long`, zmarnowałoby się w ten sposób znaczną ilość miejsca na stercie (a w skrajnym przypadku mogłoby to doprowadzić do wyczerpania się pamięci).

Gdy w dokumentacji języka Java analizuje się informacje na temat pakietu `java.lang`, można natknąć się na klasę o nazwie `StrictMath`. Oprócz dłuższej nazwy, klasa wydaje się identyczna z klasą `Math`. Różnice między dwiema klasami można podsumować następująco:

- Metody klasy `StrictMath` zwracają identyczne wyniki na wszystkich platformach. Natomiast metody klasy `Math` mogą zwracać nieco odmienne wartości, zależnie od platformy.
- Ponieważ `StrictMath` nie może używać elementów charakterystycznych dla konkretnych platform, takich jak choćby koprocesor obliczeń matematycznych o zwiększonej precyzji, implementacja klasy `StrictMath` może być mniej wydajna od implementacji klasy `Math`.

W większości przypadków metody klasy `Math` mogą wywoływać swoje odpowiedniczki z klasy `StrictMath`. Dwoma wyjątkami od tej reguły są metody `toDegrees()` i `toRadians()`. Wprawdzie w obydwóch klasach obie metody mają taką samą implementację, lecz w nagłówkach tych metod w klasie `StrictMath` występuje zastrzeżone słowo `strictfp`:

```
public static strictfp double toDegrees(double angrad)
public static strictfp double toRadians(double angdeg)
```

Według Wikipedii (*http://en.wikipedia.org/wiki/Strictfp*) słowo zastrzeżone `strictfp` ogranicza obliczenia zmiennopozycyjne w taki sposób, aby zapewnić przenośność. Słowo to umożliwia przenośność dzięki zapewnieniu jednolitej pośredniej reprezentacji liczb zmiennopozycyjnych oraz w zakresie nadmiarów i niedomiarów (czyli generowania wartości zbyt dużej lub zbyt małej w stosunku do ograniczeń reprezentacji).

■ **Uwaga** • Zgodnie ze wspomnianym przed chwilą artykułem z Wikipedii na temat słowa zastrzeżonego `strictfp` klasa `Math` zawiera metodę `public static strictfp double abs(double);` oraz inne metody `strictfp`. Jednak gdy w Javie 6 update 16 przeanalizuje się kod źródłowy tej klasy, okaże się, że nie ma w nim żadnego wystąpienia słowa `strictfp`. Jednak wiele metod klasy `Math` (na przykład metoda `sin()`) wywołuje swoje odpowiedniczki z klasy `StrictMath`, które są już zaimplementowane w bibliotece dla odpowiedniej platformy, a implementacje metod w tej bibliotece bazują już na `strictfp`.

Jeżeli słowo zastrzeżone `strictfp` nie jest obecne, wówczas obliczenia pośrednie nie są ograniczone do 32-bitowych i 64-bitowych reprezentacji zmiennopozycyjnych obsługiwanych przez Javę. W zamian w obliczeniach można korzystać z szerszych reprezentacji (w szczególności 128-bitowych) na tych platformach, które takie reprezentacje obsługują.

W przypadku reprezentacji wartości na 32 lub 64 bitach w trakcie obliczeń pośrednich może dojść do nadmiaru lub niedomiaru. Natomiast jeśli reprezentacja bazuje na większej liczbie bitów, prawdopodobieństwo wystąpienia nadmiaru lub niedomiaru się zmniejsza.

Ze względu na te różnice zapewnienie pełnej przenośności nie jest możliwe. Dlatego słowo zastrzeżone `strictfp` do pewnego stopnia wyrównuje te niespójności przez nałożenie wymagania, by na wszystkich platformach obliczenia pośrednie były wykonywane na reprezentacji 32-bitowej lub 64-bitowej.

Gdy w deklaracji metody zostanie zawarte słowo zastrzeżone `strictfp`, będzie ono gwarantować, że wszystkie wykonywane w metodzie obliczenia zmiennopozycyjne zostaną wykonane zgodnie z regułami zapewnienia precyzji. Słowo zastrzeżone `strictfp` można umieścić

również w deklaracji klasy (na przykład `public strictfp class FourierTransform`), aby zapewnić, że wszystkie obliczenia zmiennopozycyjne wykonywane w ramach tej klasy będą przeprowadzane z zapewnieniem właściwej precyzji.

■ **Uwaga** • Klasy `Math` i `StrictMath` są zadeklarowane jako `final` i dlatego nie można ich rozszerzać. Ponadto w klasach tych znajdują się deklaracje prywatnych, pustych i bezargumentowych konstruktorów, co powoduje, że nie można tworzyć ich instancji.

Klasy `Math` i `StrictMath` to przykłady tak zwanych **klas narzędziowych** (ang. *utility classes*), ponieważ wyznaczają one obszar zarezerwowany dla stałych narzędziowych oraz metod narzędziowych (`static`).

Klasa BigDecimal

W rozdziale 2. zdefiniowano klasę `CheckingAccount` z polem o nazwie `balance`. Zgodnie z deklaracją pole `balance` jest typu `int`, a dodatkowo w kodzie źródłowym został umieszczony komentarz, według którego pole `balance` reprezentuje liczbę złotych, jaką można wypłacić z konta. Alternatywnie można było wskazać, że pole `balance` zawiera liczbę groszy dostępnych do wypłaty.

Można zadać pytanie, dlaczego pole `balance` nie zostało zadeklarowane jako pole typu `double` lub `float`. Dzięki temu w polu `balance` można by przechowywać na przykład wartość `18,26` (18 złotych w części całkowitoliczbowej i 26 groszy w części ułamkowej). Pole `balance` nie zostało zadeklarowane jako typu `float` lub `double` z następujących względów:

- Nie wszystkie wartości zmiennopozycyjne, które mogą reprezentować kwoty pieniężne (w złotych i groszach), mogą być przechowywane w pamięci z odpowiednią precyzją. Na przykład liczba `0.1` (która może oznaczać 10 groszy) nie posiada precyzyjnej reprezentacji w pamięci. Gdyby wykonano wyrażenie `double total = 0.1; for (int i = 0; i < 50; i++) total += 0.1; System.out.println(total);`, jego wynikiem byłaby wartość 5.099999999999998, a nie prawidłowa wartość 5.1.
- Wynik każdego obliczenia zmiennopozycyjnego musi zostać zaokrąglony do jednego grosza. Jeżeli tak się nie stanie, w obliczeniach pojawią się drobne niedokładności, które w efekcie mogą spowodować, że ostateczny wynik będzie się znacznie różnił od prawidłowego. Wprawdzie klasa `Math` udostępnia dwie metody `round()`, za pomocą których można próbować zaokrąglać wyniki obliczeń do jednego grosza, lecz metody te zaokrąglają wartości do najbliższych liczb całkowitych (czyli złotych).

Aplikacja `InvoiceCalc` z listingu 6.3 ilustruje obydwa wspomniane problemy. Jednak pierwszy z tych problemów nie jest aż tak poważny, ponieważ nie wpływa znacząco na dokładność ostatecznego wyniku. Zdecydowanie bardziej istotny jest drugi problem, przez który wyniki obliczeń nie są zaokrąglane do jednego grosza.

Listing 6.3. *Obliczanie zmiennopozycyjnej wartości faktury, które prowadzi do powstania błędów*

```
import java.text.NumberFormat;

class InvoiceCalc
{
```

```
   final static double DISCOUNT_PERCENT = 0.1; // 10%
   final static double TAX_PERCENT = 0.05; // 5%
   public static void main(String[] args)
   {
      double invoiceSubtotal = 285.36;
      double discount = invoiceSubtotal*DISCOUNT_PERCENT;
      double subtotalBeforeTax = invoiceSubtotal-discount;
      double salesTax = subtotalBeforeTax*TAX_PERCENT;
      double invoiceTotal = subtotalBeforeTax+salesTax;
      NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();
      System.out.println("Suma: " + currencyFormat.format(invoiceSubtotal));
      System.out.println("Rabat: " + currencyFormat.format(discount));
      System.out.println("Suma po uwzględnieniu rabatu: " +
                         currencyFormat.format(subtotalBeforeTax));
      System.out.println("Podatek: " + currencyFormat.format(salesTax));
      System.out.println("Łącznie: " + currencyFormat.format(invoiceTotal));
   }
}
```

Na listingu 6.3 wykorzystano klasę `NumberFormat` (z pakietu `java.text`) i jej metodę `format()`, aby sformatować wartość zmiennopozycyjną o podwójnej precyzji na postać wartości walutowej. Więcej na temat klasy `NumberFormat` powiemy w rozdziale 9. Gdy uruchomi się aplikację `InvoiceCalc`, zwróci ona następujący wynik:

```
Suma: 285,36 zł
Rabat: 28,54 zł
Suma po uwzględnieniu rabatu: 256,82 zł
Podatek: 12,84 zł
Łącznie: 269,67 zł
```

W wynikach działania aplikacji prawidłowo obliczone zostały kwoty sumy, rabatu, sumy po uwzględnieniu rabatu i podatku. Nieprawidłowo natomiast wskazana jest kwota łączna, której wartość wynosi 269,67 zamiast 269,66. Klient nie będzie skłonny zapłacić dodatkowego grosza, mimo że zgodnie z regułami obliczeń zmiennopozycyjnych to właśnie kwota 269,67 jest wartością prawidłową:

```
Suma: 285,36
Rabat: 28,536
Suma po uwzględnieniu rabatu: 256,824
Podatek: 12,8412
Łącznie: 269,6652
```

Źródłem problemu jest to, że po wykonaniu każdej operacji, a przed wykonaniem operacji następnej uzyskany wynik nie jest zaokrąglany do najbliższego grosza. W efekcie 0,024 w liczbie 256,824 oraz 0,012 w liczbie 12,84 są uwzględniane także w kwocie końcowej, przez co metoda `format()` klasy `NumberFormat` zaokrągla kwotę końcową do wartości 269,67.

Java udostępnia rozwiązanie obydwóch problemów w postaci klasy `java.math.BigDecimal`. Jest to klasa niezmienna (to znaczy instancja klasy `BigDecimal` nie może być zmieniona), która reprezentuje liczbę dziesiętną o określonym znaku (na przykład 23,653) ze wskazana **precyzją** (liczbą cyfr) i odpowiednią **skalą** (czyli z uwzględnieniem liczby całkowitej wyznaczającej liczbę cyfr po przecinku).

W klasie `BigDecimal` zadeklarowano trzy wygodne stałe: `ONE`, `TEN` i `ZERO`. Każda z tych stałych jest odpowiednikiem wartości 1, 10 i 0 ze skalą zerową.

■ **Ostrzeżenie** • W klasie `BigDecimal` zadeklarowano kilka stałych, których nazwy zaczynają się od słowa `ROUND_`. Stałe te są w większości przestarzałe i powinno się ich unikać. To samo dotyczy metod `public BigDecimal divide(BigDecimal divisor, int scale, int roundingMode)` oraz `public BigDecimal setScale(int newScale, int roundingMode)`, które są nadal obecne w klasie, aby zagwarantować prawidłową kompilację starszego kodu.

Klasa `BigDecimal` deklaruje ponadto kilka przydatnych konstruktorów i metod. Wybrane konstruktory i metody zostały opisane w tabeli 6.2.

Tabela 6.2. *Konstruktory i metody klasy BigDecimal*

Metoda	Opis
`BigDecimal(int val)`	Inicjalizuje instancję klasy `BigDecimal` liczbą cyfr wskazywaną przez `val` i skalą 0.
`BigDecimal(String val)`	Inicjalizuje instancję klasy `BigDecimal` dziesiętnym odpowiednikiem `val`. Jako skalę ustawia liczbę cyfr po przecinku lub 0, jeżeli przecinek nie występuje. Jeśli `val` będzie `null`, konstruktor rzuci wyjątek `java.lang.NullPointerException`. Jeżeli reprezentacja `val` w postaci ciągu znaków będzie nieprawidłowa (na przykład będzie zawierała litery), konstruktor rzuci wyjątek `java.lang.NumberFormatException`.
`BigDecimal abs()`	Zwraca nową instancję klasy `BigDecimal`, która zawiera bezwzględną wartość wartości instancji bieżącej. Skala nowej instancji jest taka sama jak skala instancji bieżącej.
`BigDecimal add(BigDecimal augend)`	Zwraca nową instancję klasy `BigDecimal`, która zawiera sumę wartości bieżącej oraz argumentu przekazanego do konstruktora. Skala w nowej instancji jest wyznaczana przez wartość większą spośród skali instancji bieżącej i skali instancji przekazanej jako argument. Jeżeli `augend` będzie `null`, metoda rzuci wyjątek `NullPointerException`.
`BigDecimal divide(BigDecimal divisor)`	Zwraca nową instancję klasy `BigDecimal`, która zawiera iloraz wartości bieżącej podzielonej i wartości argumentu. Skala nowej instancji to różnica między skalą instancji bieżącej i skalą instancji przekazanej jako argument. Skala ta może zostać odpowiednio dostosowana, jeżeli do przedstawienia wyniku dzielenia potrzebna będzie większa liczba cyfr. Jeżeli `divisor` będzie `null`, metoda rzuci wyjątek `NullPointerException`, a jeżeli `divisor` będzie reprezentować wartość 0 lub wyniku dzielenia nie będzie można zaprezentować precyzyjnie, metoda rzuci wyjątek `java.lang.ArithmeticException`.
`BigDecimal max(BigDecimal val)`	Zwraca `this` lub `val`, zależnie od tego, która z tych instancji posiada większą wartość. Jeżeli `val` będzie `null`, metoda rzuci wyjątek `NullPointerException`.
`BigDecimal min(BigDecimal val)`	Zwraca `this` lub `val`, zależnie od tego, która z tych instancji posiada mniejszą wartość. Jeżeli `val` będzie `null`, metoda rzuci wyjątek `NullPointerException`.

Tabela 6.2. *Konstruktory i metody klasy BigDecimal — ciąg dalszy*

Metoda	Opis
`BigDecimal multiply(BigDecimal multiplicand)`	Zwraca nową instancję klasy `BigDecimal`, która zawiera iloczyn wartości bieżącej i wartości instancji podanej jako argument. Skala nowej instancji jest sumą skali instancji bieżącej i instancji przekazanej jako argument. Jeżeli `multiplicand` jest `null`, metoda rzuci wyjątek `NullPointerException`.
`BigDecimal negate()`	Zwraca nową instancję klasy `BigDecimal`, która zawiera wartość przeciwną do wartości bieżącej. Skala nowej instancji jest taka sama jak skala instancji bieżącej.
`int precision()`	Zwraca precyzję bieżącej instancji klasy `BigDecimal`.
`BigDecimal remainder(BigDecimal divisor)`	Zwraca nową instancję klasy `BigDecimal`, która zawiera resztę z dzielenia wartości instancji bieżącej przez wartość instancji przekazanej jako argument. Skala nowej instancji to różnica między skalą bieżącą i skalą argumentu. Skala ta może zostać odpowiednio dostosowana, jeżeli do wyświetlenia wyniku potrzeba będzie większej liczby cyfr. Jeżeli `divisor` będzie `null`, metoda rzuci wyjątek `NullPointerException`, a jeśli `divisor` będzie mieć wartość 0, metoda rzuci wyjątek `ArithmeticException`.
`int scale()`	Zwraca skalę bieżącej instancji klasy `BigDecimal`.
`BigDecimal setScale(int newScale, RoundingMode roundingMode)`	Zwraca nową instancję klasy `BigDecimal` o wskazanej skali i trybie zaokrąglania. Jeżeli nowa skala jest większa niż skala dotychczasowa, wówczas do wartości, które były niewyskalowane, dodawane są zera. W takiej sytuacji nie potrzeba wykonywać zaokrąglania. Jeżeli nowa skala jest mniejsza niż skala dotychczasowa, wówczas usuwane są ostatnie cyfry. Jeżeli usuwane cyfry są inne niż zero, uzyskana w ten sposób liczba niewyskalowana musi zostać zaokrąglona. Zaokrąglenie wykonuje się w trybie zaokrąglania wskazanym jako argument. Jeżeli `roundingMode` jest `null`, metoda rzuci wyjątek `NullPointerException`. Jeżeli natomiast `roundingMode` będzie mieć wartość `RoundingMode.ROUND_UNNECESSARY`, a ze względu na bieżącą skalę zaokrąglenie będzie potrzebne, metoda rzuci wyjątek `ArithmeticException`.
`BigDecimal subtract(BigDecimal subtrahend)`	Zwraca nową instancję klasy `BigDecimal`, która zawiera wartość bieżącą pomniejszoną o wartość argumentu. Skala nowej instancji jest większą spośród skali bieżącej i skali argumentu. Jeżeli `subtrahend` będzie `null`, metoda rzuci wyjątek `NullPointerException`.
`String toString()`	Zwraca ciąg znaków, który stanowi reprezentację bieżącej instancji `BigDecimal`. Jeżeli to konieczne, zostanie użyta notacja naukowa.

W tabeli 6.2 wspomniano o typie `RoundingMode` wyznaczającym tryb zaokrąglania. `Rounding ↪Mode` to typ wyliczeniowy `enum`, który zawiera stałe odpowiadające różnorodnym trybom zaokrąglania. Stałe te opisano w tabeli 6.3.

Najlepszym sposobem oswojenia się z klasą `BigDecimal` jest zastosowanie jej w praktyce. Na listingu 6.4 wykorzystano tę klasę do wykonania prawidłowych obliczeń wartości na fakturze, którą przedstawiono wcześniej na listingu 6.3.

Tabela 6.3. *Stałe typu RoundingMode*

Stała	Opis
`CEILING`	Zaokrągla w kierunku plus nieskończoności.
`DOWN`	Zaokrągla w kierunku zera.
`FLOOR`	Zaokrągla w kierunku minus nieskończoności.
`HALF_DOWN`	Zaokrągla w kierunku „bliższej liczby sąsiedniej", chyba że odległość do liczb sąsiednich jest taka sama — wówczas zaokrągla w dół.
`HALF_EVEN`	Zaokrągla w kierunku „bliższej liczby sąsiedniej", chyba że odległość do liczb sąsiednich jest taka sama — wówczas zaokrągla w stronę tej liczby sąsiedniej, która jest parzysta.
`HALF_UP`	Zaokrągla w kierunku „bliższej liczby sąsiedniej", chyba że odległość do liczb sąsiednich jest taka sama — wówczas zaokrągla w górę (o tym trybie zaokrąglania najczęściej uczy się w szkole).
`UNNECESSARY`	Zaokrąglanie nie jest potrzebne, ponieważ wynik działania jest zawsze dokładny.
`UP`	Liczby dodatnie są zaokrąglane w kierunku plus nieskończoności, a liczby ujemne są zaokrąglane w kierunku minus nieskończoności.

Listing 6.4. *Obliczenia wartości na fakturze przy użyciu klasy BigDecimal, dzięki której unika się błędów zaokrągleń*

```
class InvoiceCalc
{
   public static void main(String[] args)
   {
      BigDecimal invoiceSubtotal = new BigDecimal("285.36");
      BigDecimal discountPercent = new BigDecimal("0.10");
      BigDecimal discount = invoiceSubtotal.multiply(discountPercent);
      discount = discount.setScale(2, RoundingMode.HALF_UP);
      BigDecimal subtotalBeforeTax = invoiceSubtotal.subtract(discount);
      subtotalBeforeTax = subtotalBeforeTax.setScale(2, RoundingMode.HALF_UP);
      BigDecimal salesTaxPercent = new BigDecimal("0.05");
      BigDecimal salesTax = subtotalBeforeTax.multiply(salesTaxPercent);
      salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
      BigDecimal invoiceTotal = subtotalBeforeTax.add(salesTax);
      invoiceTotal = invoiceTotal.setScale(2, RoundingMode.HALF_UP);
      System.out.println("Suma: " + invoiceSubtotal);
      System.out.println("Rabat: " + discount);
      System.out.println("Suma po uwzględnieniu rabatu: " + subtotalBeforeTax);
      System.out.println("Podatek: " + salesTax);
      System.out.println("Łącznie: " + invoiceTotal);
   }
}
```

W metodzie `main()` z listingu 6.4 najpierw tworzone są obiekty `BigDecimal` o nazwach `invoiceSubtotal` i `discountPercent`, inicjalizowane wartościami odpowiednio `285.36` i `0.10`. Wartość `invoiceSubtotal` jest mnożona przez `discountPercent`, a wynikowa instancja klasy `BigDecimal` tego mnożenia jest przypisywana zmiennej `discount`.

Na tym etapie zmienna `discount` ma wartość `28.5360`. Jeśli nie liczyć ostatniego zera, wartość ta jest taka sama jak wartość wynikowa wyrażenia `invoiceSubtotal*DISCOUNT_PERCENT`

z listingu 6.3. Wartością, która powinna zostać przypisana zmiennej `discount`, jest `28.54`. Aby rozwiązać ten problem jeszcze przed wykonaniem kolejnego obliczenia, w metodzie `main()` jest wywoływana metoda `setScale()` z następującymi argumentami:

- `2` — dwie cyfry po przecinku.
- `RoundingMode.HALF_UP` — standardowy sposób zaokrąglania.

Gdy określona jest już odpowiednia skala i tryb zaokrąglania, metoda `main()` odejmuje `discount` od `invoiceSubtotal`, a instancję `BigDecimal` stanowiącą reprezentację uzyskanej różnicy przypisuje zmiennej `subtotalBeforeTax`. Następnie na zmiennej `subtotalBeforeTax` jest wywoływana metoda `setScale()`, aby odpowiednio zaokrąglić jej wartość przed wykonaniem kolejnego obliczenia.

W kolejnym kroku `main()` tworzy obiekt klasy `BigDecimal` o nazwie `salesTaxPercent`, który zostaje zainicjalizowany wartością `0.05`. Dalej `subtotalBeforeTax` zostaje przemnożona przez `salesTaxPercent`, a iloczyn jest przypisywany zmiennej `salesTax`. Na tej zmiennej, której typem jest `BigDecimal`, jest wywoływana metoda `setScale()`, aby odpowiednio zaokrąglić wartość reprezentowaną przez ten obiekt.

Po wykonaniu tej czynności `main()` dodaje `salesTax` do `subtotalBeforeTax`. Suma zostaje przypisana zmiennej `invoiceTotal`, na której metoda `setScale()` dokonuje odpowiedniego zaokrąglenia. Na koniec wartości wszystkich obiektów klasy `BigDecimal` zostają przekazane do standardowego wyjścia za pomocą metody `System.out.println()`, która wywołuje na każdym obiekcie ich metody `toString()`, aby uzyskać ciąg znaków reprezentujący wartości poszczególnych instancji `BigDecimal`.

Gdy tak zmodyfikowana wersja aplikacji `InvoiceCalc` zostanie uruchomiona, uzyskamy następujący wynik:

```
Suma: 285,36
Rabat: 28,54
Suma po uwzględnieniu rabatu: 256,82
Podatek: 12,84
Łącznie: 269,66
```

■ **Ostrzeżenie** • Klasa `BigDecimal` deklaruje konstruktor `BigDecimal(double val)`, którego w miarę możliwości powinno się unikać. Konstruktor ten inicjalizuje instancję klasy `BigDecimal` wartością przekazaną jako `val`, przez co w sytuacji gdy wartość typu `double` nie może zostać przedstawiona jako wartość dokładna, tak utworzona instancja może odzwierciedlać nieprawidłową reprezentację wartości. Na przykład wynikiem wywołania `BigDecimal(0.1)` jest instancja z wartością `0.1000000000000000055511151231257827021181583404541015625`. Z kolei wywołanie `BigDecimal("0.1")` daje w wyniku instancję z prawidłową wartością `0.1`.

Klasa BigInteger

Klasa `BigDecimal` przechowuje liczbę dziesiętną ze znakiem w postaci niewyskalowanej wartości ze skalą określoną przez 32-bitową liczbę całkowitą. Wartość niewyskalowana jest przechowywana w instancji klasy `java.math.BigInteger`.

Klasa `BigInteger` jest klasą niezmienną, która reprezentuje liczbę całkowitą ze znakiem o określonej precyzji. Wartość jest w tej klasie przechowywana w **kodzie uzupełnień do dwóch** (inaczej mówiąc, wszystkie bity są zamienione — jedynki na zera, a zera na jedynki — a do wyniku dodawana jest wartość 1, aby wynik był zgodny z kodem uzupełnień do dwóch używanym w typach języka Java: całkowitoliczbowym bajtowym, krótkim całkowitoliczbowym, całkowitoliczbowym i długim całkowitoliczbowym).

■ **Uwaga** • Więcej informacji na temat zapisu w kodzie uzupełnień do dwóch można znaleźć w Wikipedii, na stronie pod adresem *http://pl.wikipedia.org/wiki/Kod_uzupe%C5%82nie%C5%84_do_dw%C3%B3ch*.

W klasie `BigInteger` są zadeklarowane trzy przydatne stałe o nazwach `ONE`, `TEN` i `ZERO`. Każda z tych stałych odpowiada instancji klasy `BigInteger` reprezentującej wartości odpowiednio 1, 10 i 0.

W klasie `BigInteger` jest zadeklarowana całkiem pokaźna liczba przydatnych konstruktorów i metod. Kilka wybranych konstruktorów i metod przedstawiono w tabeli 6.4.

Najlepszym sposobem, aby oswoić się z klasą `BigInteger`, jest wypróbowanie jej w praktyce. Na listingu 6.5 wykorzystano klasę `BigInteger` do zaimplementowania metod `factorial`, aby porównać sposób ich działania.

Listing 6.5. *Porównanie metod factorial()*

```
class FactComp
{
   public static void main(String[] args)
   {
      System.out.println(factorial(12));
      System.out.println();
      System.out.println(factorial(20L));
      System.out.println();
      System.out.println(factorial(170.0));
      System.out.println();
      System.out.println(factorial(new BigInteger("170")));
      System.out.println();
      System.out.println(factorial(25.0));
      System.out.println();
      System.out.println(factorial(new BigInteger("25")));
   }
   public static int factorial(int n)
   {
      if (n == 0)
         return 1;
      else
         return n*factorial(n-1);
   }
   public static long factorial(long n)
   {
      if (n == 0)
         return 1;
      else
         return n*factorial(n-1);
   }
   public static double factorial(double n)
```

Tabela 6.4. *Konstruktory i metody klasy BigInteger*

Metoda	Opis
`BigInteger(byte[] val)`	Inicjalizuje instancję klasy `BigInteger` liczbą całkowitą przechowywaną w tablicy `val`. Element `val[0]` tej tablicy zawiera osiem najbardziej znaczących (lewych) bitów. Jeżeli `val` będzie `null`, konstruktor rzuci wyjątek `NullPointerException`, natomiast jeśli `val.length` będzie mieć wartość 0, konstruktor rzuci wyjątek `NumberFormatException`.
`BigInteger(String val)`	Inicjalizuje instancję klasy `BigInteger` liczbą całkowitą odpowiadającą `val`. Jeżeli `val` będzie `null`, konstruktor rzuci wyjątek `NullPointerException`, a jeśli ciąg znaków reprezentujący `val` będzie nieprawidłowy (na przykład będzie zawierać litery), wówczas konstruktor rzuci wyjątek `NumberFormatException`.
`BigInteger abs()`	Zwraca nową instancję klasy `BigInteger`, która zawiera wartość bezwzględną wartości instancji bieżącej.
`BigInteger add(BigInteger augend)`	Zwraca nową instancję klasy `BigInteger`, która zawiera sumę wartości bieżącej oraz wartości przekazanej jako argument. Jeżeli `augend` będzie `null`, metoda rzuci wyjątek `NullPointerException`.
`BigInteger divide(BigInteger divisor)`	Zwraca nową instancję klasy `BigInteger`, która zawiera iloraz wartości bieżącej podzielonej przez wartość przekazaną jako argument. Jeżeli `divisor` jest `null`, metoda rzuci `NullPointerException`, natomiast jeśli `divisor` reprezentuje wartość `0` lub wynik dzielenia nie jest wartością dokładną, metoda rzuci wyjątek `ArithmeticException`.
`BigInteger max(BigInteger val)`	Zwraca `this` lub `val` zależnie od tego, która instancja klasy `BigInteger` zawiera większą wartość. Jeżeli `val` jest `null`, metoda rzuci wyjątek `NullPointerException`.
`BigInteger min(BigInteger val)`	Zwraca `this` lub `val` zależnie od tego, która instancja klasy `BigInteger` zawiera mniejszą wartość. Jeżeli `val` jest `null`, metoda rzuci wyjątek `NullPointerException`.
`BigInteger multiply(BigInteger multiplicand)`	Zwraca nową instancję klasy `BigInteger`, która zawiera iloczyn wartości bieżącej oraz wartości argumentu. Jeżeli `multiplicand` będzie `null`, metoda rzuci `NullPointerException`.
`BigInteger negate()`	Zwraca nową instancję klasy `BigInteger`, która zawiera wartość przeciwną do wartości bieżącej.
`BigInteger remainder(BigInteger divisor)`	Zwraca nową instancję klasy `BigInteger`, która zawiera resztę z podzielenia wartości bieżącej przez wartość argumentu. Jeżeli `divisor` jest `null`, metoda rzuci wyjątek `NullPointerException`, natomiast jeżeli `divisor` reprezentuje wartość `0`, metoda rzuci wyjątek `ArithmeticException`.
`BigInteger subtract(BigInteger subtrahend)`	Zwraca nową instancję klasy `BigInteger`, która zawiera wartość bieżącą pomniejszoną o wartość argumentu. Jeżeli `subtrahend` będzie `null`, metoda rzuci wyjątek `NullPointerException`.
`String toString()`	Zwraca ciąg znaków stanowiący reprezentację instancji `BigInteger`.

```
   {
      if (n == 1.0)
         return 1.0;
      else
         return n*factorial(n-1);
   }
   public static BigInteger factorial(BigInteger n)
   {
      if (n.equals(BigInteger.ZERO))
         return BigInteger.ONE;
      else
         return n.multiply(factorial(n.subtract(BigInteger.ONE)));
   }
}
```

Na listingu 6.5 porównano cztery wersje rekurencyjnej metody `factorial()`, której zadaniem jest wyliczanie silni. Porównanie to pozwala zidentyfikować największą wartość argumentu, jaką można przekazać do pierwszych trzech metod, zanim metody te zaczną zwracać nieprawidłowe wartości spowodowane przekroczeniem przedziału wartości poprawnie reprezentowanych przez dany typ liczbowy.

W pierwszej wersji metody wykorzystywany jest argument typu `int`, a przedział wartości przekazywanego argumentu wynosi od 0 do 12. Jeżeli do metody zostanie przekazany dowolny argument większy od 12, wartości silni tego argumentu nie będzie już można zaprezentować prawidłowo jako liczby typu `int`.

Przedział, do jakiego może należeć argument metody `factorial()`, można poszerzyć przez zmianę typu parametru na `long`. Przedział ten nie zwiększy się jednak znacząco. Po wprowadzeniu zmiany typu argumentu do metody będzie można przekazać wartość nie większą niż 20.

Aby jeszcze bardziej zwiększyć zakres obsługiwanych wartości, można zaimplementować metodę `factorial()`, której parametr i zwracana wartość będą typu `double`. Jest to możliwe, ponieważ za pomocą typu `double` można precyzyjnie reprezentować liczby całkowite. Jednak w takim przypadku największym argumentem, jaki można przekazać do metody, jest 170.0. Każda wartość większa niż 170.0 spowoduje, że metoda `factorial()` zwróci plus nieskończoność.

Może się zdarzyć sytuacja, że konieczne będzie wyliczenie silni wartości większej niż 170 — na przykład gdy będą wykonywane pewne obliczenia statystyczne uwzględniające kombinacje lub permutacje. Jedyny sposób, który pozwoli na obliczanie takiej silni, polega na użyciu wersji metody `factorial()` wykorzystującej typ `BigInteger`.

Gdy aplikacja z listingu 6.5 zostanie uruchomiona, zwróci następujące wyniki:

```
479001600

2432902008176640000

7.257415615307994E306

725741561530799896739672821112926311471699168129645137654357779890056184340170615785235074924261745951149099123783852077666602256544275302532890077320751090240043028005829560396661259965825710439855829425756896631343961226257109494680671120556888045719334021266145280000000000000000000000000000000000

1.5511210043330986E25

15511210043330985984000000
```

Pierwsze trzy wartości to najwyższe wartości silni zwracane przez metody `factorial()`, w których wykorzystano odpowiednio typy `int`, `long` i `double`. Czwarta wartość reprezentuje wartość typu `BigInteger`, która odpowiada najwyższej wartości silni typu `double`.

Warto zauważyć, że metoda z typem `double` nie jest w stanie dokładnie zaprezentować wartości 170! (! to matematyczny symbol silni). Precyzja tej metody jest po prostu zbyt mała. Wprawdzie metoda próbuje zaokrąglić najmniejszą cyfrę, lecz takie zaokrąglenie nie zawsze się udaje — liczba kończy się cyframi 7994 zamiast 7998. Zaokrąglenie sprawdza się jedynie w przypadkach argumentów nie większych niż 25.0, o czym świadczą dwa ostatnie wiersze danych wynikowych.

■ **Uwaga** • Algorytm szyfrowania RSA, klasa `BigDecimal` i obliczanie silni to praktyczne przykłady zastosowania klasy `BigInteger`. Klasy tej można używać jednak także w zastosowaniach niestandardowych. Na przykład w artykule opublikowanym w magazynie „JavaWorld" z lutego 2006 pod tytułem *Travel Through Time with Java* (*http://www.javaworld.com/javaworld/jw-02-2006/jw-0213-funandgames.html*), który stanowił kolejną część serii *Java Fun and Games*, została wykorzystana klasa `BigInteger` do przechowywania obrazka w postaci bardzo dużej liczby całkowitej. Pomysł polegał na tym, by poeksperymentować z klasą `BigInteger` i sprawdzić, jak za jej pomocą można wyszukiwać zdjęcia ludzi i miejsc, które istniały w przeszłości, będą istnieć w przyszłości lub nigdy nie istniały. Artykuł można polecić zwłaszcza programistom, którzy lubią trochę poszaleć.

Informacje na temat pakietów

Klasa `java.lang.Package` udostępnia informacje na temat pakietu (wstępne informacje na temat pakietów zostały przedstawione w rozdziale 4.). Informacje te dotyczą wersji implementacji i specyfikacji pakietu języka Java, nazwy pakietu, a także wskazują, czy pakiet został **upakowany** (ang. *sealed*), czyli czy wszystkie klasy wchodzące w skład pakietu zostały zarchiwizowane w tym samym pliku JAR.

■ **Uwaga** • Pliki JAR zostały opisane w rozdziale 1.

W tabeli 6.5 opisano wybrane metody klasy `Package`.

Na listingu 6.6 znajduje się kod źródłowy aplikacji `PackageInfo`, którą zaimplementowano, aby zaprezentować działanie większości metod klasy `Package` z tabeli 6.5.

Listing 6.6. *Uzyskiwanie informacji na temat pakietu*

```
public class PackageInfo
{
   public static void main(String[] args)
   {
      if (args.length == 0)
      {
         System.err.println("użycie: java PackageInfo nazwaPakietu [wersja]");
         return;
      }
      Package pkg = Package.getPackage(args[0]);
      if (pkg == null)
```

Tabela 6.5. *Metody klasy Package*

Metoda	Opis
`String getImplementationTitle()`	Zwraca tytuł implementacji pakietu, który może być `null`. Format tytułu jest nieokreślony.
`String getImplementationVendor()`	Zwraca nazwę sprzedawcy lub organizacji, która dostarczyła implementację pakietu. Nazwa ta może być `null`. Format nazwy jest nieokreślony.
`String getImplementationVersion()`	Zwraca numer wersji implementacji pakietu, który może być `null`. Ciąg reprezentujący numer wersji musi być sekwencją dodatnich dziesiętnych liczb całkowitych, oddzielonych od siebie znakami kropki. Liczby te mogą mieć poprzedzające zera.
`String getName()`	Zwraca nazwę pakietu w standardowym zapisie z kropkami, na przykład `java.lang`.
`static Package getPackage(String packageName)`	Zwraca obiekt klasy `Package`, który jest skojarzony z pakietem wskazywanym przez `packageName`. Jeżeli pakietu wskazywanego przez `packageName` nie można znaleźć, zwraca `null`. Jeżeli `packageName` jest `null`, metoda zwraca `NullPointerException`.
`static Package[] getPackages()`	Zwraca tablicę wszystkich obiektów klasy `Package`, które są dostępne dla kodu wywołującego.
`String getSpecificationTitle()`	Zwraca tytuł specyfikacji pakietu, który może być `null`. Format tytułu jest nieokreślony.
`String getSpecificationVendor()`	Zwraca nazwę sprzedawcy lub organizacji, która dostarczyła specyfikację zaimplementowaną w ramach pakietu. Nazwa ta może być `null`. Format nazwy jest nieokreślony.
`String getSpecificationVersion()`	Zwraca numer wersji specyfikacji implementacji pakietu, który może być `null`. Ciąg wskazujący numer wersji musi być sekwencją dodatnich dziesiętnych liczb całkowitych oddzielonych od siebie znakami kropek. Liczby te mogą mieć poprzedzające zera.
`boolean isCompatibleWith(String desired)`	Sprawdza, czy pakiet jest zgodny z podanym ciągiem wersji. W tym celu wersja specyfikacji pakietu jest porównywana z wersją `desired`. Jeżeli numer wersji specyfikacji pakietu jest wyższy od pożądanego numeru wersji `desired` lub mu równy, metoda zwraca wartość `true` (co oznacza, że pakiet jest zgodny); w przeciwnym razie metoda zwraca wartość `false`. Jeżeli `desired` jest `null`, metoda rzuca wyjątek `NullPointerException`, a jeżeli numer wersji pakietu lub numer wersji `desired` nie ma formatu z kropkami, metoda rzuca wyjątek `NumberFormatException`.
`boolean isSealed()`	Zwraca wartość `true`, jeżeli pakiet jest upakowany. W przeciwnym razie zwracana jest wartość `false`.

```
      {
         System.err.println("Nie znaleziono pakietu " + args[0]);
         return;
      }
      System.out.println("Nazwa: " + pkg.getName());
      System.out.println("Tytuł implementacji: " +
                         pkg.getImplementationTitle());
      System.out.println("Dostawca implementacji: " +
                         pkg.getImplementationVendor());
      System.out.println("Wersja implementacji: " +
                         pkg.getImplementationVersion());
      System.out.println("Tytuł specyfikacji: " +
                         pkg.getSpecificationTitle());
      System.out.println("Dostawca specyfikacji: " +
                         pkg.getSpecificationVendor());
      System.out.println("Wersja specyfikacji: " +
                         pkg.getSpecificationVersion());
      System.out.println("Upakowany: " + pkg.isSealed());
      if (args.length > 1)
         System.out.println("Zgodny z wersją " + args[1] + ": " +
                            pkg.isCompatibleWith(args[1]));
   }
}
```

Aby skorzystać z aplikacji, w wierszu poleceń należy podać przynajmniej nazwę pakietu. Na przykład polecenie `java PackageInfo java.lang` zwraca w wersji 6 języka Java następujące dane wynikowe:

```
Nazwa: java.lang
Tytuł implementacji: Java Runtime Environment
Dostawca implementacji: Sun Microsystems, Inc.
Wersja implementacji: 1.6.0_16
Tytuł specyfikacji: Java Platform API Specification
Dostawca specyfikacji: Sun Microsystems, Inc.
Wersja specyfikacji: 1.6
Upakowany: false
```

Aplikacja `PackageInfo` pozwala ustalić, czy specyfikacja pakietu jest zgodna z określonym numerem wersji. Pakiet jest zgodny ze swoimi poprzednikami.

Na przykład jeżeli aplikację wywoła się poleceniem `java PackageInfo java.lang 1.6`, zwróci ona wynik `Zgodny z wersją 1.6: true`, podczas gdy aplikacja wywołana poleceniem `java PackageInfo java.lang 1.8` zwróci wynik `Zgodny z wersją 1.6: false`.

Za pomocą aplikacji `PackageInfo` można także analizować pakiety utworzone samodzielnie. O tym, jak tworzy się własne pakiety, była już mowa w rozdziale 4. W przykładzie 4. zaprezentowano przykładowy pakiet `logging`.

Plik *PackageInfo.class* należy skopiować do katalogu, w którym znajduje się katalog pakietu `logging` (w którym z kolei znajdują się skompilowane pliki klas). Następnie należy wykonać polecenie `java PackageInfo logging`.

Aplikacja `PackageInfo` zwróci wówczas następujący wynik:

```
Nie znaleziono pakietu logging
```

Komunikat błędu jest zwracany dlatego, że metoda `getPackage()` wymaga załadowania z pakietu co najmniej jednego pliku klasy, zanim będzie mogła zwrócić obiekt klasy `Package` z opisem pakietu.

Jedynym sposobem, w jaki można wyeliminować powyższy komunikat błędu, jest załadowanie klasy z pakietu. W tym celu kod z listingu 6.7 należy uwzględnić w kodzie z listingu 6.6.

Listing 6.7. *Dynamiczne ładowanie klasy z pliku klasy*

```
if (args.length == 3)
try
{
   Class.forName(args[2]);
}
catch (ClassNotFoundException cnfe)
{
   System.err.println("nie można załadować " + args[2]);
   return;
}
```

Powyższy fragment kodu należy wstawić przed instrukcją `Package pkg = Package.get ↪Package(args[0]);`. Kod ten ładuje plik klasy o nazwie wskazanej przez trzeci argument wiersza poleceń aplikacji `PackageInfo`.

Gdy nową wersję aplikacji `PackageInfo` uruchomi się poleceniem `java PackageInfo logging 1.5 logging.File`, zwróci ona dane wynikowe przedstawione poniżej. W poleceniu wywołującym jako klasę, którą aplikacja ma załadować, wskazano klasę `File` pakietu `logging`.

```
Nazwa: logging
Tytuł implementacji: null
Dostawca implementacji: null
Wersja implementacji: null
Tytuł specyfikacji: null
Dostawca specyfikacji: null
Wersja specyfikacji: null
Upakowany: false
Exception in thread "main" java.lang.NumberFormatException: Empty version string
       at java.lang.Package.isCompatibleWith(Unknown Source)
       at PackageInfo.main(PackageInfo.java:43)
```

Tak wiele wartości `null` w danych wynikowych nie powinno dziwić, ponieważ do pakietu `logging` nie dodano żadnych informacji. Dodatkowo metoda `isCompatibleWith()` rzuca wyjątek `NumberFormatException`, ponieważ pakiet `logging` nie zawiera numeru wersji specyfikacji w zapisie z kropką (numer ten jest `null`).

Zapewne najprostszym sposobem umieszczenia informacji na temat pakietu w pakiecie `logging` jest utworzenie pliku *logging.jar* w sposób podobny do opisanego w rozdziale 4. Najpierw jednak trzeba utworzyć niewielki plik tekstowy, który będzie zawierać te informacje o pakiecie. Plik można nazwać dowolnie. Na listingu 6.8 została przedstawiona wersja pliku o nazwie *manifest.mf*.

Listing 6.8. *Zawartość pliku manifest.mf z informacjami o pakiecie*

```
Implementation-Title: Implementacja mechanizmu rejestracji
Implementation-Vendor: Jeff Friesen
Implementation-Version: 1.0a
Specification-Title: Specyfikacja mechanizmu rejestracji
Specification-Vendor: Jeff "JavaJeff" Friesen
Specification-Version: 1.0
Sealed: true
```

■ **Uwaga** • Należy pamiętać o tym, by ostatni wiersz pliku (`Sealed: true`) zakończyć naciśnięciem klawisza *Return* lub *Enter*. W przeciwnym razie aplikacja zwróci prawdopodobnie wartość `Upakowany: false`, ponieważ narzędzie *jar* pakietu JDK może nie zapisać w pakiecie `logging` odpowiadającego temu parametrowi wiersza z pliku manifestu. Potwierdza to powszechną opinię, że narzędzie *jar* jest dość kapryśne.

Aby utworzyć plik JAR, który będzie zawierał pakiet `logging` i składające się na niego pliki oraz którego **manifest**, czyli specjalny plik *MANIFEST.MF*, będzie zawierał informacje na temat zawartości pliku JAR przedstawione na listingu 6.8, należy wykonać następujące polecenie:

```
jar cfm logging.jar manifest.mf logging
```

Powyższe polecenie utworzy plik JAR o nazwie *logging.jar* (o czym decyduje opcja `c` odpowiedzialna za utworzenie i opcja `f`, która mówi o pliku). Polecenie włączy także zawartość pliku *manifest.mf* (dzięki opcji `m` jak manifest) do pliku *MANIFEST.MF*, który znajduje się w katalogu *META-INF* pakietu.

■ **Uwaga** • Aby dowiedzieć się więcej na temat manifestu pliku JAR, warto zapoznać się z punktem „Jar Manifest" na stronie „JAR File Specification" dokumentacji JDK (*http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#JAR%20Manifest*).

Jeżeli narzędzie *jar* nie zwróci żadnych komunikatów o błędach, należy wykonać przedstawione poniżej polecenie wiersza poleceń systemu Windows (lub wiersza systemu operacyjnego, który akurat jest używany). Polecenie to wykona aplikację `PackageInfo` i odczyta informacje na temat pakietu `logging`:

```
java -cp logging.jar;. PackageInfo logging 1.0 logging.File
```

Tym razem wynik działania aplikacji będzie następujący:

```
Nazwa: logging
Tytuł implementacji: Implementacja mechanizmu rejestracji
Dostawca implementacji: Jeff Friesen
Wersja implementacji: 1.0a
Tytuł specyfikacji: Specyfikacja mechanizmu rejestracji
Dostawca specyfikacji: Jeff "JavaJeff" Friesen
Wersja specyfikacji: 1.0
Upakowany: false
Zgodny z wersją 1.0: true
```

Podstawowe klasy opakowujące

Pakiet `java.lang` zawiera klasy `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` i `Short`. Są to tak zwane **podstawowe klasy opakowujące** (ang. *primitive wrapper classes*), ponieważ opakowują one wartości typów podstawowych.

■ **Uwaga** • Podstawowe klasy opakowujące są również znane jako **klasy wartości** (ang. *value classes*).

Java udostępnia osiem wspominanych podstawowych klas opakowujących z dwóch powodów:

- Platforma kolekcji (o której więcej powiemy w rozdziale 8.) udostępnia listy, zbiory i mapy, które mogą przechowywać jedynie obiekty, natomiast nie można w nich umieszczać wartości typów podstawowych. Wartość typu podstawowego umieszcza się w instancji podstawowej klasy opakowującej, a następnie tę instancję można umieścić w kolekcji.
- Podstawowe klasy opakowujące są kontenerami, w których można wygodnie skojarzyć stałe (na przykład `MAX_VALUE` i `MIN_VALUE`) i metody klas (takie jak metody `parseInt()` klasy `Integer` i metody `isDigit()`, `isLetter()` i `toUpperCase()` klasy `Character`) z typami podstawowymi.

W tym punkcie opiszemy wszystkie podstawowe klasy opakowujące, a także klasę o nazwie `Number`.

Klasa Boolean

Klasa `Boolean` to najmniejsza spośród podstawowych klas opakowujących. Klasa ta deklaruje trzy stałe, z których dwie to stałe `TRUE` i `FALSE`, oznaczające predefiniowane obiekty klasy `Boolean`.

Klasa `Boolean` deklaruje także dwa konstruktory, które inicjalizują obiekty klasy:

- `Boolean(boolean value)` inicjalizuje obiekt klasy `Boolean` wartością `value`.
- `Boolean(String s)` przekształca tekst przypisany parametrowi `s` na wartość `true` lub `false`, a następnie umieszcza tę wartość w obiekcie klasy `Boolean`.

Drugi konstruktor porównuje wartość parametru `s` z wartością `true`. Ponieważ w porównaniu tym nie bierze się pod uwagę wielkości liter, dowolna kombinacja czterech liter składających się na wartość `true` (czyli `true`, `TRUE`, `tRue` i tak dalej) zostanie umieszczona w obiekcie. W przeciwnym razie w obiekcie zostanie umieszczona wartość `false`.

Uzupełnieniem dla konstruktorów klasy `Boolean` jest metoda `boolean booleanValue()`, która zwraca opakowaną wartość klasy `Boolean`.

Klasa `Boolean` deklaruje także lub pokrywa następujące metody:

- `int compareTo(Boolean b)`, która porównuje bieżący obiekt klasy `Boolean` z parametrem `b`, aby ustalić ich kolejność. Metoda zwraca wartość 0, jeżeli bieżący obiekt zawiera tę samą wartość logiczną co parametr `b`, wartość dodatnią, jeżeli bieżący obiekt zawiera wartość `true`, a parametr `b` ma wartość `false`, oraz wartość ujemną, jeżeli bieżący obiekt zawiera wartość `false`, a parametr `b` wartość `true`.
- `boolean equals(Object o)`, która porównuje bieżący obiekt klasy `Boolean` z parametrem `o` i zwraca `true`, jeżeli `o` nie jest `null`, `o` jest typu `Boolean` i obydwa obiekty zawierają tę samą wartość logiczną.
- `static boolean getBoolean(String name)`, która zwraca `true`, jeżeli właściwość systemowa (o których powiemy w rozdziale 7.) identyfikowana przez `name` istnieje i jest równa `true`.

- `int hashCode()`,zwracająca odpowiedni kod skrótu, dzięki któremu możliwe jest wykorzystanie obiektów klasy `Boolean` w kolekcjach opartych na skrótach (powiemy o nich w rozdziale 8.).
- `static boolean parseBoolean(String s)`, która parsuje wartość parametru `s` i zwraca `true`, jeżeli `s` równa się `"true"`, `"TRUE"`, `"True"` lub dowolnej innej kombinacji tych liter. W przeciwnym razie metoda zwraca `false`. (**Parsowanie** to czynność, w trakcie której sekwencja znaków jest rozkładana na komponenty o określonym znaczeniu, tak zwane **tokeny**).
- `String toString()`, która zwraca `"true"`, jeżeli bieżąca instancja klasy `Boolean` zawiera `true`. W przeciwnym razie metoda zwraca `"false"`.
- `static String toString(boolean b)`, która zwraca `"true"`, jeżeli `b` zawiera `true`. W przeciwnym razie metoda zwraca wartość `"false"`.
- `static Boolean valueOf(boolean b)`, która zwraca `TRUE`, jeżeli `b` zawiera `true` lub `FALSE`, jeżeli `b` zawiera `false`.
- `static Boolean valueOf(String s)`, która zwraca `TRUE`, jeżeli `s` równa się `"true"`, `"TRUE"`, `"True"` lub dowolnej innej kombinacji tych liter. W przeciwnym razie metoda zwraca `FALSE`.

■ **Ostrzeżenie** • Programiści, którzy dopiero poznają klasę `Boolean`, często spodziewają się, że metoda `getBoolean()` zwróci wartość `true` lub `false` obiektu klasy `Boolean`. Jednak metoda `getBoolean()` zwraca wartość właściwości systemowej opartej na klasie `Boolean` (więcej na temat właściwości systemowych powiemy w rozdziale 7.). Aby uzyskać wartość `true` lub `false`, należy skorzystać z metody `booleanValue()`.

W większości przypadków lepiej jest użyć stałych `TRUE` lub `FALSE`, zamiast tworzyć obiekty klasy `Boolean`. Załóżmy na przykład, że trzeba zaimplementować metodę, która zwróci obiekt klasy `Boolean` z wartością `true`, jeżeli argument typu `double` tej metody będzie miał wartość ujemną, lub `false`, jeżeli argument ten będzie mieć wartość dodatnią lub będzie równy zero. Potrzebną metodę można zadeklarować w sposób analogiczny jak w przypadku metody `isNegative()` z listingu 6.9.

Listing 6.9. *Metoda isNegative(), w której tworzy się obiekt klasy Boolean, choć nie jest to konieczne*

```
public Boolean isNegative(double d)
{
   return new Boolean(d < 0);
}
```

Kod metody jest zwarty, lecz niepotrzebnie tworzy się w niej obiekt klasy `Boolean`. Jeśli metoda będzie często wywoływana, zostanie utworzonych wiele obiektów `Boolean`, które zajmą przestrzeń na stercie. Gdy na stercie zaczyna brakować miejsca, włączany jest proces odśmiecania, który spowalnia działanie aplikacji i obniża w ten sposób jej wydajność.

Na listingu 6.10 przedstawiono ulepszoną wersję metody `isNegative()`.

Listing 6.10. *Zrefaktoryzowana metoda isNegative(), w której nie tworzy się obiektów klasy Boolean*

```
public Boolean isNegative(double d)
{
   return (d < 0) ? Boolean.TRUE : Boolean.FALSE;
}
```

W nowej wersji metody nie trzeba już tworzyć obiektów klasy `Boolean`, ponieważ metoda zwraca predefiniowany obiekt `TRUE` lub `FALSE`.

■ **Wskazówka** • Należy dążyć do tego, by liczba tworzonych obiektów klasy `Boolean` była jak najmniejsza. Dzięki temu mniejsza będzie ilość pamięci zużywanej przez aplikację, a wydajność samej aplikacji będzie wyższa, ponieważ proces odśmiecania nie będzie musiał być uruchamiany zbyt często.

Klasa Character

Klasa `Character` jest największą spośród podstawowych klas opakowujących. Zawiera znaczną liczbę stałych, konstruktor, wiele metod oraz dwie klasy zagnieżdżone (`Subset` i `UnicodeBlock`).

■ **Uwaga** • Wysoki stopień złożoności klasy `Character` ma swoje źródło w tym, że Java obsługuje Unicode (*http://pl.wikipedia.org/wiki/Unicode*). Dla uproszczenia pominięto większość mechanizmów klasy związanych z obsługą Unicode, ponieważ wykraczają one poza zakres tego rozdziału.

Klasa `Character` deklaruje jeden konstruktor `Character(char value)`, za pomocą którego obiekt klasy `Character` inicjalizuje się wartością `value`. Uzupełnieniem konstruktora jest metoda `charValue()`, która zwraca wartość opakowanego znaku.

Programiści, którzy implementują swoje pierwsze aplikacje, często używają wyrażeń w postaci `ch >= '0' && ch <= '9'` (aby sprawdzić, czy `ch` zawiera cyfry) oraz `ch >= 'A' && ch <= 'Z'` (aby sprawdzić, czy `ch` zawiera wielką literę). Takiej praktyki powinno się jednak unikać z następujących względów:

- Bardzo łatwo jest popełnić błąd w wyrażeniu. Na przykład wyrażenie `ch > '0' && ch <= '9'` zawiera drobny błąd, który powoduje, że wartość `'0'` nie spełnia warunku wyrażenia.
- Na podstawie wyrażeń stosunkowo trudno jest ustalić, co tak naprawdę jest w nich sprawdzane.
- W wyrażeniach można używać jedynie cyfr łacińskich (*0 – 9*) oraz łacińskich liter (*A – Z* i *a – z*). W wyrażeniach nie są natomiast uwzględniane cyfry i litery, które są poprawne w innych językach. Na przykład `'\u0beb'` to literał znakowy, który reprezentuje jedną z cyfr w języku tamilskim.

Klasa `Character` deklaruje kilka metod narzędziowych, które służą do porównywania oraz przekształcania i w ten sposób rozwiązują przytoczone przed chwilą problemy. Metodami tymi są:

- `static boolean isDigit(char ch)`, która zwraca `true`, jeśli `ch` zawiera cyfrę (standardowo z przedziału od 0 do 9, ale również cyfry z innych języków).
- `static boolean isLetter(char ch)`, która zwraca `true`, jeśli `ch` zawiera literę (standardowo *A – Z* lub *a – z*, ale również litery z innych języków).
- `static boolean isLetterOrDigit(char ch)`, która zwraca `true`, jeśli `ch` zawiera literę lub cyfrę (standardowo *A – Z*, *a – z* lub *0 – 9*, ale również litery lub cyfry z innych języków).
- `static boolean isLowerCase(char ch)`, która zwraca `true`, jeśli `ch` zawiera małą literę.
- `static boolean isUpperCase(char ch)`, która zwraca `true`, jeśli `ch` zawiera wielką literę.
- `static boolean isWhitespace(char ch)`, która zwraca `true`, jeśli `ch` zawiera znak niewidoczny (zazwyczaj znak spacji, tabulacji poziomej, powrotu karetki lub nowego wiersza).
- `static char toLowerCase(char ch)`, która zwraca małą literę odpowiadającą wielkiej literze zawartej w `ch`. Jeżeli `ch` nie zawiera wielkiej litery, metoda zwraca wartość `ch`.
- `static char toUpperCase(char ch)`, która zwraca wielką literę odpowiadającą małej literze zawartej w `ch`. Jeżeli `ch` nie zawiera małej litery, metoda zwraca wartość `ch`.

Na przykład zamiast `ch >= '0' && ch <= '9'` zdecydowanie lepiej jest wykonać metodę `isDigit(ch)`, ponieważ w ten sposób unika się ryzyka popełnienia błędów, sama metoda jest zdecydowanie bardziej czytelna, a poza tym zwraca ona wartość `true` nie tylko dla cyfr łacińskich, ale także cyfr z innych języków (na przykład `'\u0beb'`).

Klasy Float i Double

Klasy `Float` i `Double` przechowują w obiektach typu `Float` i `Double` odpowiednio wartości zmiennopozycyjne i wartości zmiennopozycyjne o podwójnej precyzji. Klasy te deklarują następujące stałe:

- `MAX_VALUE` identyfikuje maksymalną wartość, którą można reprezentować jako wartość typu `float` lub `double`.
- `MIN_VALUE` identyfikuje minimalną wartość, którą można reprezentować jako wartość typu `float` lub `double`.
- `NaN` reprezentuje `0.0F/0.0F` jako typu `float` oraz `0.0/0.0` jako typu `double`.
- `NEGATIVE_INFINITY` reprezentuje minus nieskończoność jako typu `float` lub `double`.
- `POSITIVE_INFINITY` reprezentuje plus nieskończoność jako typu `float` lub `double`.

Klasy `Float` i `Double` deklarują także następujące konstruktory, które inicjalizują obiekty tych klas:

- `Float(float value)` inicjalizuje obiekt klasy `Float` wartością `value`.
- `Float(double value)` inicjalizuje obiekt klasy `Float` odpowiednikiem wartości `value` typu `float`.

- `Float(String s)` przekształca tekst ze zmiennej `s` w wartość zmiennopozycyjną i inicjalizuje tą wartością obiekt klasy `Float`.
- `Double(double value)` inicjalizuje obiekt klasy `Double` wartością `value`.
- `Double(String s)` przekształca tekst ze zmiennej `s` w wartość zmiennopozycyjną o podwójnej precyzji i inicjalizuje tą wartością obiekt klasy `Double`.

Uzupełnieniem dla konstruktorów klasy `Float` jest metoda `float floatValue()`, która zwraca opakowaną wartość zmiennopozycyjną. Analogicznie uzupełnieniem dla konstruktorów klasy `Double` jest metoda `double doubleValue()`, która zwraca opakowaną wartość zmiennopozycyjną o podwójnej precyzji.

Oprócz metody `floatValue()` klasa `Float` deklaruje kilka metod narzędziowych. Są to następujące metody:

- `static int floatToIntBits(float value)` przekształca `value` w 32-bitową liczbę całkowitą.
- `static boolean isInfinite(float f)` zwraca `true`, jeśli wartością `f` jest plus nieskończoność lub minus nieskończoność. Podobna do niej metoda `public boolean isInfinite()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Float` jest plus nieskończoność lub minus nieskończoność.
- `static boolean isNaN(float f)` zwraca `true`, jeśli wartością `f` jest `NaN`. Podobna do niej metoda `public boolean isNaN()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Float` jest `NaN`.
- `static float parseFloat(String s)` parsuje `s` i zwraca wartość zmiennopozycyjną, która odpowiada zawartej w `s` tekstowej reprezentacji wartości zmiennopozycyjnej lub rzuca wyjątek `NumberFormatException`, jeżeli reprezentacja ta jest nieprawidłowa (na przykład zawiera litery).

Oprócz metody `doubleValue()` klasa `Double` deklaruje kilka metod narzędziowych. Są to następujące metody:

- `static long doubleToLongBits(double value)` przekształca `value` na długą liczbę całkowitą.
- `static boolean isInfinite(double d)` zwraca `true`, jeśli wartością `d` jest plus nieskończoność lub minus nieskończoność. Podobna do niej metoda `boolean isInfinite()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Double` jest plus nieskończoność lub minus nieskończoność.
- `static boolean isNaN(double d)` zwraca `true`, jeśli wartością `d` jest `NaN`. Podobna do niej metoda `public boolean isNaN()` zwraca `true`, jeśli wartością bieżącego obiektu klasy `Double` jest `NaN`.
- `static double parseDouble(String s)` parsuje `s` i zwraca wartość zmiennopozycyjną o podwójnej precyzji, która odpowiada zawartej w `s` tekstowej reprezentacji wartości zmiennopozycyjnej o podwójnej precyzji lub rzuca wyjątek `NumberFormatException`, jeżeli reprezentacja ta jest nieprawidłowa.

Za pomocą metod `floatToIntBits()` i `doubleToIntBits()` implementuje się metody `equals()` i `hashCode()`, które mają dotyczyć pól typu `float` i `double`. Dzięki metodom `floatToIntBits()` i `doubleToIntBits()` metody `equals()` i `hashCode()` działają prawidłowo w opisanych poniżej sytuacjach.

- `equals()` musi zwrócić `true`, jeśli `f1` i `f2` zawierają `Float.NaN` (lub jeśli `d1` i `d2` zawierają `Double.NaN`). Gdyby metoda `equals()` była zaimplementowana w sposób analogiczny do `f1.floatValue() == f2.floatValue()` (albo `d1.doubleValue() == d2.doubleValue()`), metoda zwróciłaby `false`, ponieważ `NaN` nie jest równa żadnej innej wartości oprócz samej siebie.
- `equals()` musi zwrócić `false`, jeśli `f1` zawiera `+0.0` i `f2` zawiera `-0.0` (albo odwrotnie) lub jeśli `d1` zawiera `+0.0` i `d2` zawiera `-0.0` (albo odwrotnie). Gdyby metoda `equals()` była zaimplementowana w sposób analogiczny do `f1.floatValue() == f2.floatValue()` (lub `d1.doubleValue() == d2.doubleValue()`), metoda zwróciłaby `true`, ponieważ wyrażenie `+0.0 == -0.0` zwraca `true`.

Spełnienie tych wymagań jest niezbędne, aby kolekcje bazujące na skrótach (o których więcej powiemy w rozdziale 8.) funkcjonowały prawidłowo. Na listingu 6.11 widać, w jaki sposób opisane wymagania wpływają na działanie metod `equals()` klas `Float` i `Double`.

Listing 6.11. *Ilustracja działania metody equals() klasy Float w kontekście wartości NaN oraz metody equals() klasy Double w kontekście wartości +0.0 i –0.0*

```
public static void main(String[] args)
{
   Float f1 = new Float(Float.NaN);
   System.out.println(f1.floatValue());
   Float f2 = new Float(Float.NaN);
   System.out.println(f2.floatValue());
   System.out.println(f1.equals(f2));
   System.out.println(Float.NaN == Float.NaN);
   System.out.println();
   Double d1 = new Double(+0.0);
   System.out.println(d1.doubleValue());
   Double d2 = new Double(-0.0);
   System.out.println(d2.doubleValue());
   System.out.println(d1.equals(d2));
   System.out.println(+0.0 == -0.0);
}
```

Gdy aplikacja zostanie uruchomiona, zwróci przedstawione poniżej dane wynikowe. Dowodzą one, że metoda `equals()` klasy `Float` odpowiednio obsługuje wartość `NaN`, a metoda `equals()` klasy `Double` odpowiednio uwzględnia wartości `+0.0` i `-0.0`:

```
NaN
NaN
true
false

0.0
-0.0
false
true
```

■ **Wskazówka** • Aby sprawdzić, czy wartości typu `float` lub `double` są równe plus nieskończoności lub minus nieskończoności (ale nie obydwóm tym wartościom), nie należy do tego celu używać metody `isInfinite()`. Zamiast niej wartość należy porównać za pomocą operatora `==` ze stałą `NEGATIVE_INFINITY` lub `POSITIVE_INFINITY`, na przykład `f == Float.NEGATIVE_INFINITY`.

Metody `parseFloat()` i `parseDouble()` okażą się przydatne w wielu sytuacjach. Na przykład na listingu 6.12 wykorzystano metodę `parseDouble()`, aby wykonać parsowanie argumentów wiersza poleceń do wartości typu `double`.

Listing 6.12. *Parsowanie argumentów wiersza poleceń do wartości zmiennopozycyjnych o podwójnej precyzji*

```
public static void main(String[] args)
{
   if (args.length != 3)
   {
      System.err.println("użycie: java Calc wartość1 operator wartość2");
      System.err.println("operatorem może być +, -, * lub /");
      return;
   }
   try
   {
      double value1 = Double.parseDouble(args[0]);
      double value2 = Double.parseDouble(args[2]);
      if (args[1].equals("+"))
         System.out.println(value1+value2);
      else
      if (args[1].equals("-"))
         System.out.println(value1-value2);
      else
      if (args[1].equals("*"))
         System.out.println(value1*value2);
      else
      if (args[1].equals("/"))
         System.out.println(value1/value2);
      else
         System.err.println("nieprawidłowy operator: " + args[1]);
   }
   catch (NumberFormatException nfe)
   {
      System.err.println("Nieprawidłowy format liczby: " + nfe.getMessage());
   }
}
```

Aby wypróbować powyższą aplikację, można wywołać ją poleceniem `java Calc 1OE+3 + 66.0`. Wynikiem zwróconym przez aplikację będzie `10066.0`. Gdyby natomiast wywołać ją poleceniem `java Calc 1OE+3 + A`, aplikacja zwróciłaby następujący komunikat błędu: `Nieprawidłowy format liczby: "A"`. Komunikat ten jest spowodowany rzuceniem wyjątku `NumberFormatException`, co zachodzi w wyniku drugiego wywołania metody `parseDouble()`.

Choć klasa `NumberFormatException` opisuje wyjątki niekontrolowane, a wyjątków niekontrolowanych zwykle się nie obsługuje, ponieważ reprezentują one błędy w kodzie źródłowym, to jednak w tym przykładzie `NumberFormatException` nie pasuje do tego schematu. Wyjątek jest rzucany nie w wyniku błędu w kodzie źródłowym, lecz z powodu przekazania do aplikacji nieprawidłowego argumentu liczbowego, na co jakość kodu źródłowego nie ma żadnego wpływu.

Klasy Integer, Long, Short i Byte

Klasy `Integer`, `Long`, `Short` i `Byte` przechowują wartości całkowitoliczbowe odpowiednio 32-, 64-, 16- i 8-bitowe w obiektach typów odpowiednio `Integer`, `Long`, `Short` i `Byte`.

Każda z tych klas deklaruje stałe `MAX_VALUE` i `MIN_VALUE` identyfikujące wartości maksymalną i minimalną, które może reprezentować typ podstawowy powiązany z daną klasą.

Klasy deklarują także konstruktory, które inicjalizują ich obiekty.

- `Integer(int value)` inicjalizuje obiekt klasy `Integer` wartością `value`.
- `Integer(String s)` przekształca tekst w argumencie `s` na 32-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy `Integer`.
- `Long(long value)` inicjalizuje obiekt klasy `Long` wartością `value`.
- `Long(String s)` przekształca tekst w argumencie `s` na 64-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy `Long`.
- `Short(short value)` inicjalizuje obiekt klasy `Short` wartością `value`.
- `Short(String s)` przekształca tekst w argumencie `s` na 16-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy `Short`.
- `Byte(byte value)` inicjalizuje obiekt klasy `Byte` wartością `value`.
- `Byte(String s)` przekształca tekst w argumencie `s` na 8-bitową wartość całkowitoliczbową i inicjalizuje tą wartością obiekt klasy `Byte`.

Uzupełnieniem dla konstruktorów klasy `Integer` jest metoda `int intValue()`, dla konstruktorów klasy `Long` takim uzupełnieniem jest metoda `long longValue()`, dla konstruktorów klasy `Short` — metoda `short shortValue()`, zaś dla konstruktorów klasy `Byte` — metoda `byte byteValue()`. Wszystkie wspomniane metody zwracają opakowane liczby całkowite.

Wszystkie cztery klasy deklarują różnorodne przydatne metody do przetwarzania liczb całkowitych. Na przykład klasa `Integer` deklaruje wymienione poniżej metody klas, które przeprowadzają konwersję 32-bitowej liczby całkowitej na ciąg znaków `String` zgodnie ze wskazaną reprezentacją (binarną, szesnastkową, ósemkową i dziesiątkową):

- `static String toBinaryString(int i)` zwraca obiekt klasy `String`, który zawiera binarną reprezentację argumentu `i`. Na przykład `Integer.toBinaryString(255)` zwróci obiekt klasy `String`, który będzie zawierać ciąg `11111111`.
- `static String toHexString(int i)` zwraca obiekt klasy `String`, który zawiera szesnastkową reprezentację argumentu `i`. Na przykład `Integer.toHexString(255)` zwróci obiekt klasy `String`, który będzie zawierać ciąg `ff`.
- `static String toOctalString(int i)` zwraca obiekt klasy `String`, który zawiera ósemkową reprezentację argumentu `i`. Na przykład `toOctalString(64)` zwróci obiekt klasy `String`, który będzie zawierać ciąg `377`.
- `static String toString(int i)` zwraca obiekt klasy `String`, który zawiera dziesiątkową reprezentację argumentu `i`. Na przykład `toString(255)` zwróci obiekt klasy `String`, który będzie zawierać ciąg `255`.

Często wygodnie jest poprzedzać ciąg binarny zerami, tak aby ciągi te móc umieszczać w kolumnach o takiej samej szerokości. Na przykład można zaimplementować aplikację, która będzie wyświetlać następujący, obustronnie wyrównany zbiór danych wynikowych:

```
11110001
+
00000111
--------
11111000
```

Niestety, metoda `toBinaryString()` nie ułatwia wykonania tego zadania. Na przykład metoda `Integer.toBinaryString(7)` zwróci obiekt klasy `String`, który będzie zawierał ciąg 111, a nie 00000111. Na listingu 6.13 przedstawiono metodę `toAlignedBinaryString()`, która wyświetla ciąg binarny w pożądanym, obustronnie wyrównanym formacie.

Listing 6.13. *Wyrównywanie binarnego ciągu znaków*

```
public static void main(String[] args)
{
   System.out.println(toAlignedBinaryString(7, 8));
   System.out.println(toAlignedBinaryString(255, 16));
   System.out.println(toAlignedBinaryString(255, 7));
}
static String toAlignedBinaryString(int i, int numBits)
{
   String result = Integer.toBinaryString(i);
   if (result.length() > numBits)
      return null; // nie można zmieścić wyniku w numBits kolumnach
   int numLeadingZeros = numBits-result.length();
   String zerosPrefix = "";
   for (int j = 0; j < numLeadingZeros; j++)
      zerosPrefix += "0";
   return zerosPrefix + result;
}
```

Metoda `toAlignedBinaryString()` przyjmuje dwa argumenty. Pierwszy zawiera 32-bitową liczbę całkowitą, którą należy przekształcić na binarny ciąg znaków, zaś drugi argument wskazuje liczbę kolumn bitów, w których należy wyświetlić ten ciąg znaków.

Najpierw jest wywoływana metoda `toBinaryString()`, która zwraca binarny ciąg znaków stanowiący odpowiednik wartości `i`, lecz bez poprzedzających zer. Następująca po nim metoda `toAlignedBinaryString()` sprawdza, czy wszystkie cyfry binarnego ciągu znaków zmieszczą się w kolumnach bitów, których liczbę wyznacza `numBits`. Jeżeli cyfr będzie więcej niż kolumn, metoda zwróci `null`. (Więcej informacji na temat metody `length()` oraz innych metod klasy `String` przedstawimy w rozdziale 7.).

W dalszej kolejności metoda `toAlignedBinaryString()` oblicza, iloma zerami należy poprzedzić wartość `result`, po czym w pętli tworzy ciąg znaków złożony z takiej liczby zer. Na koniec metoda zwraca ciąg zer, które mają poprzedzić wynikowy ciąg znaków.

Złożona operacja połączenia ciągów znaków za pomocą operatora przypisania (`+=`) na pierwszy rzut oka nie budzi wątpliwości, jednak tak naprawdę jest to rozwiązanie mało wydajne, ponieważ pośrednie obiekty tworzone w trakcie tej operacji są najpierw tworzone, a następnie niszczone. Zastosowaliśmy jednak tak niewydajny kod, aby w rozdziale 7. przeciwstawić mu rozwiązanie zdecydowanie lepsze pod względem wydajności.

Gdy aplikacja zostanie uruchomiona, zwróci następujące dane wynikowe:

```
00000111
0000000011111111
null
```

Klasa Number

Wszystkie spośród klas `Float`, `Double`, `Integer`, `Long`, `Short` i `Byte` udostępniają nie tylko metodę *x*`Value()` odnoszącą się do samej klasy, ale również metody *x*`Value()` odnoszące się do wszystkich pozostałych klas. Na przykład w klasie `Float` dostępne są, oprócz `floatValue()`, również metody `doubleValue()`, `intValue()`, `longValue()`, `shortValue()` oraz `byteValue()`.

Wszystkie sześć metod to składowe klasy `java.lang.Number`, która jest abstrakcyjną klasą bazową dla klas `Float`, `Double`, `Integer`, `Long`, `Short` i `Byte`. Metody `floatValue()`, `doubleValue()`, `intValue()` i `longValue()` klasy `Number` są więc metodami abstrakcyjnymi. Klasa `Number` jest również klasą bazową dla klas `BigDecimal` i `BigInteger` (a także dla niektórych klas przeznaczonych do pracy współbieżnej, o czym więcej powiemy w rozdziale 9.).

Klasa `Number` istnieje po to, by uprościć iterowanie przez kolekcję obiektów klas potomnych po `Number`. Na przykład można zadeklarować zmienną typu `List<Number>` i zainicjalizować ją instancją klasy `ArrayList<Number>`. W tak utworzonej kolekcji można przechowywać różne klasy potomne po `Number` i iterować przez kolekcję, wykorzystując do tego wielopostaciowość wywołań metod klas podrzędnych.

API References

W rozdziale 2. opisano mechanizm odśmiecania, który usuwa obiekt ze sterty, jeśli nie istnieją już żadne odwołania do tego obiektu.

W rozdziale 3. z kolei przedstawiono metodę `finalize()` klasy `Object`. Jest to metoda, którą wywołuje proces odśmiecania, zanim usunie obiekt ze sterty. Metoda `finalize()` pozwala obiektowi na wykonanie czynności sprzątających.

W tym punkcie będzie kontynuowany wątek zapoczątkowany w rozdziałach 2. oraz 3. Zostanie też zaprezentowane API References. Interfejs ten umożliwia aplikacji komunikowanie się, w ograniczonym zakresie, z procesem odśmiecania.

Na początek przedstawimy podstawowe pojęcia związane z API References. Następnie opiszemy klasy `Reference` i `ReferenceQueue` interfejsu, po czym zakończymy prezentacją klas `SoftReference`, `WeakReference` i `PhantomReference`.

Podstawowe pojęcia

Gdy uruchamia się aplikację, tworzony jest dla niej **główny zbiór odwołań** (ang. *root set of references*). Zbiór ten jest kolekcją zmiennych lokalnych, parametrów, pól klas i pól instancji, które aktualnie istnieją i zawierają — potencjalnie puste — odwołania do obiektów. Zbiór główny zmienia się z upływem czasu i w miarę wykonywania przez aplikację kolejnych zadań. Na przykład po powrocie z metody ze zbioru znikają jej parametry.

Wiele procesów odśmiecania odczytuje główny zbiór odwołań w momencie, gdy zostają uruchomione. Na podstawie głównego zbioru odwołań proces odśmiecania sprawdza, czy obiekt jest **osiągalny** (albo czy obiekt **żyje**, to znaczy czy istnieją jakieś odwołania do niego), czy też jest **nieosiągalny** (czyli nie istnieją żadne odwołania do niego). Proces odśmiecania nie może niszczyć obiektów, które są osiągalne — proces ten może usuwać ze sterty jedynie te obiekty, które, począwszy od głównego zbioru odwołań, nie są osiągalne.

■ **Uwaga** • Obiektami osiągalnymi są także te obiekty, które są osiągalne pośrednio za pośrednictwem zmiennych ze zbioru głównego. Inaczej mówiąc, chodzi o obiekty, które są osiągalne za pośrednictwem żywych obiektów osiągalnych bezpośrednio za pomocą tych zmiennych. Obiekt, który pozostaje nieosiągalny na żadnej ścieżce, bez względu na to, którą zmienną z głównego zbioru odwołań się wybierze, podlega przetworzeniu przez mechanizm odśmiecania.

Począwszy od wersji 1.2 języka Java, obiekty osiągalne dzieli się na obiekty silnie osiągalne, miękko osiągalne, słabo osiągalne i złudnie osiągalne. W odróżnieniu od obiektów silnie osiągalnych, obiekty osiągalne miękko, słabo i złudnie mogą podlegać przetwarzaniu przez mechanizm odśmiecania.

W poniższych punktach scharakteryzowano cztery wspomniane typy osiągalności zależnie od mocy odwołania — od najsilniejszego do najsłabszego.

- Obiekt **jest silnie osiągalny** (ang. *strongly reachable*), jeżeli jest osiągalny dla wątku i wątek ten nie musi przechodzić przez obiekty API Reference — wątek korzysta z **silnego odwołania** (ang. *strong reference*) przypisanego zmiennej z głównego zbioru odwołań. Nowo utworzony obiekt (na przykład obiekt, do którego odwołuje się zmienna `d` z instrukcji `Double d = new Double(1.0);`) jest silnie osiągalny dla wątku, który ten obiekt stworzył. (Więcej na temat wątków powiemy w rozdziale 7.).
- Obiekt jest **miękko osiągalny** (ang. *softly reachable*), jeżeli nie jest silnie osiągalny, lecz staje się osiągalny za pośrednictwem **odwołania miękkiego** (ang. *soft reference*), czyli odwołania do obiektu, które jest przechowywane w obiekcie klasy `SoftReference`. Najsilniejszym odwołaniem do takiego obiektu jest odwołanie miękkie. Gdy ilość miejsca na stercie jest już niewielka, proces odśmiecania zwykle usuwa miękkie odwołania do najstarszych miękko osiągalnych obiektów, po czym finalizuje te obiekty (przez wywołanie metody `finalize()`) i je usuwa.
- Obiekt jest **słabo osiągalny** (ang. *weakly reachable*), jeżeli nie jest silnie ani miękko osiągalny, natomiast staje się osiągalny za pośrednictwem **odwołania słabego** (ang. *weak reference*), czyli odwołania do obiektu, które jest przechowywane w obiekcie klasy `WeakReference`. Najsilniejszym odwołaniem do takiego obiektu jest odwołanie słabe. Proces odśmiecania usuwa słabe odwołania do słabo osiągalnych obiektów i niszczy je (a wcześniej finalizuje) przy następnym uruchomieniu procesu, nawet jeśli w pamięci jest dużo wolnego miejsca.
- Obiekt jest **złudnie osiągalny** (ang. *phantom reachable*), jeżeli nie jest silnie, miękko ani słabo osiągalny, został sfinalizowany i proces odśmiecania jest gotowy na odzyskanie zajmowanej przez niego pamięci. Dodatkowo do obiektu złudnie osiągalnego

wiedzie **złudne odwołanie** (ang. *phantom reference*), czyli odwołanie do obiektu przechowywane w obiekcie klasy `PhantomReference`. Najsilniejszym odwołaniem do obiektu złudnie osiągalnego jest odwołanie złudne.

■ **Uwaga** • Jedyna różnica między odwołaniem miękkim a słabym sprowadza się do tego, że prawdopodobieństwo usunięcia obiektu miękko osiągalnego przez proces odśmiecania jest niższe niż w przypadku obiektu słabo osiągalnego. Poza tym odwołanie słabe nie ma wystarczającej mocy, by utrzymać obiekt w pamięci.

Obiekty, do których odwołanie jest przechowywane w obiektach klas `SoftReference`, `WeakReference` lub `PhantomReference`, to tak zwane **referenty**.

Klasy Reference i ReferenceQueue

API Reference zawiera pięć klas, które wchodzą w skład pakietu `java.lang.ref`. Jądrem tego pakietu są klasy `Reference` i `ReferenceQueue`.

Klasa `Reference` jest abstrakcyjną klasą główną dla wchodzących w skład pakietu klas potomnych `SoftReference`, `WeakReference` i `PhantomReference`.

Klasa `ReferenceQueue` to klasa, której instancje opisują strukturę kolejek danych. Jeśli instancja klasy `ReferenceQueue` zostanie skojarzona z obiektem klasy potomnej `Reference` (czyli, krócej mówiąc, z obiektem `Reference`), wówczas w momencie gdy referent, do którego wiedzie odwołanie enkapsulowane w obiekcie `Reference`, zostanie zniszczony przez proces odśmiecania, obiekt `Reference` zostanie dodany do kolejki.

■ **Uwaga** • Obiekt klasy `ReferenceQueue` kojarzy się z obiektem `Reference` w ten sposób, że obiekt klasy `ReferenceQueue` przekazuje się do konstruktora odpowiedniej klasy potomnej po klasie `Reference`.

Klasa `Reference` jest zadeklarowana jako ogólny typ `Reference<T>`, w którym `T` oznacza typ referenta. Klasa udostępnia następujące metody:

- `void clear()` przypisuje odwołaniu `null`. Obiekt klasy `Reference`, na którym jest wywoływana ta metoda, nie jest kolejkowany (wstawiany) do skojarzonej z nim kolejki odwołań (jeżeli oczywiście taka kolejka odwołań jest skojarzona z obiektem). (Proces odśmiecania czyści odwołania w sposób bezpośredni, a nie przez wywołanie metody `clear()`. Metoda `clear()` jest wywoływana przez aplikacje).
- `boolean enqueue()` dodaje obiekt klasy `Reference`, na którym metoda została wywołana, do skojarzonej z nią kolejki odwołań. Metoda zwraca `true`, gdy obiekt klasy `Reference` został zakolejkowany, lub `false` w przeciwnym wypadku — wówczas jest to znak, że obiekt został zakolejkowany już wcześniej albo w momencie jego tworzenia nie został skojarzony z żadną kolejką. (Proces odśmiecania kolejkuje obiekty klasy `Reference` w sposób bezpośredni, a nie przez wywołanie metody `enqueue()`. Metoda `enqueue()` jest wywoływana przez aplikacje).

- `T get()` zwraca odwołanie przechowywane w obiekcie klasy `Reference`. Jeżeli przechowywane odwołanie zostało wcześniej wyczyszczone czy to przez aplikację, czy to przez proces odśmiecania, zwracaną wartością jest `null`.
- `boolean isEnqueued()` zwraca `true`, jeśli obiekt klasy `Reference` został zakolejkowany przez aplikację lub przez proces odśmiecania. W przeciwnym razie metoda zwraca wartość `false`, co oznacza, że obiekt klasy `Reference` nie został w momencie jego utworzenia skojarzony z żadną kolejką.

Uwaga • Klasa `Reference` deklaruje także konstruktory. Ponieważ konstruktory te są prywatne w ramach pakietu, klasami potomnymi po `Reference` mogą być wyłącznie klasy z pakietu `java.lang.ref`. Nałożenie takiego ograniczenia było konieczne dlatego, że instancje klas potomnych po `Reference` muszą ściśle współpracować z procesem odśmiecania.

Klasa `ReferenceQueue` jest zadeklarowana jako ogólny typ `ReferenceQueue<T>`, w którym `T` wskazuje typ referenta. Klasa `ReferenceQueue` deklaruje konstruktor i metody opisane poniżej.

- `ReferenceQueue()` inicjalizuje nową instrukcję klasy `ReferenceQueue`.
- `Reference<? extends T> poll()` sprawdza, czy w kolejce jest dostępny obiekt klasy `Reference`. Jeżeli taki obiekt jest dostępny, zostaje on usunięty z kolejki i zwrócony przez metodę. W przeciwnym razie następuje natychmiastowy powrót z metody z wartością wynikową `null`.
- `Reference<? extends T> remove()` usuwa z kolejki następny obiekt klasy `Reference` i go zwraca. Przez nieokreślony czas metoda czeka na to, by obiekt klasy `Reference` stał się dostępny, a gdy to oczekiwanie zostanie przerwane, rzuca wyjątek klasy `java.lang.InterruptedException`.
- `Reference<? extends T> remove(long timeout)` usuwa z kolejki następny obiekt klasy `Reference` i go zwraca. Metoda czeka, aż obiekt klasy `Reference` stanie się dostępny lub upłynie tyle milisekund, ile wskazuje parametr `timeout`. Wywołanie metody z argumentem `0` spowoduje, że metoda będzie oczekiwać bez końca. Jeżeli upłynie czas wskazywany przez `timeout`, metoda zwróci `null`. Jeżeli wartość `timeout` będzie ujemna, metoda rzuci wyjątek `java.lang.IllegalArgumentException`, a jeżeli jej oczekiwanie zostanie przerwane, zostanie rzucony wyjątek klasy `InterruptedException`.

Klasa SoftReference

Klasa `SoftReference` opisuje obiekt klasy `Reference`, którego referent jest miękko osiągalny. Jest to klasa ogólna, która oprócz tego, że dziedziczy metody klasy `Reference` i pokrywa jej metodę `get()`, udostępnia także opisane poniżej konstruktory przeznaczone do inicjalizowania obiektów klasy `SoftReference`.

- `SoftReference(T r)` umieszcza w obiekcie odwołanie przekazane jako argument `r`. Obiekt klasy `SoftReference` zachowuje się jak miękkie odwołanie do `r`. Żaden obiekt klasy `ReferenceQueue` nie jest skojarzony z obiektem klasy `SoftReference`.

- `SoftReference(T r, ReferenceQueue<? super T> q)` umieszcza w obiekcie odwołanie przekazane jako argument `r`. Obiekt klasy `SoftReference` zachowuje się jak miękkie odwołanie do `r`. Z obiektem klasy `SoftReference` zostaje skojarzona kolejka `SoftReference` wskazywana przez `q`. Jeżeli przekazany do metody argument `q` będzie `null`, będzie to oznaczać, że z miękkim odwołaniem nie jest skojarzona żadna kolejka.

Klasa `SoftReference` przydaje się do implementowania buforów pamięci podręcznej, na przykład do przechowywania obrazków. Bufor obrazków przechowuje w pamięci obrazki (ponieważ ich załadowanie z dysku zabiera czas), a także zapewnia, że do pamięci nie trafiają duplikaty obrazków, które potencjalnie mogą mieć duże rozmiary.

Bufor obrazków zawiera odwołania do obiektów obrazków, które znajdują się już w pamięci. Gdyby były to silne odwołania, obrazki pozostawałyby w pamięci. Trzeba by wówczas ustalać, które z tych obrazków nie są już więcej potrzebne, i usuwać je z pamięci, aby mogły zostać przetworzone przez proces odśmiecania.

Gdyby konieczne było ręczne usuwanie obrazków, oznaczałoby to tak naprawdę powielanie zadań wykonywanych przez proces odśmiecania. Jeżeli jednak odwołania do obiektów obrazków opakuje się w obiektach klasy `SoftReference`, proces odśmiecania sam ustali, kiedy obiekty te należy usunąć (zazwyczaj będzie to wówczas, gdy na stercie będzie już mało wolnego miejsca), a także sam je usunie.

Na listingu 6.14 pokazano, w jaki sposób za pomocą klasy `SoftReference` można utrzymywać podręczny bufor obrazków.

Listing 6.14. *Utrzymywanie podręcznego bufora obrazków*

```
class Image
{
   private byte[] image;
   private Image(String name)
   {
      image = new byte[1024*100];
   }
   static Image getImage(String name)
   {
      return new Image(name);
   }
}
public class ImageCache
{
   final static int NUM_IMAGES = 200;
   @SuppressWarnings("unchecked")
   public static void main(String[] args)
   {
      String[] imageNames = new String[NUM_IMAGES];
      for (int i = 0; i < imageNames.length; i++)
         imageNames[i] = new String("obrazek" + i + ".gif");

      SoftReference<Image>[] cache = new SoftReference[imageNames.length];
      for (int i = 0; i < cache.length; i++)
         cache[i] = new SoftReference<Image>(Image.getImage(imageNames[i]));

      for (int i = 0; i < cache.length; i++)
      {
         Image im = cache[i].get();
```

```
            if (im == null)
            {
               System.out.println(imageNames[i] + " nie znajduje się w buforze");
               im = Image.getImage(imageNames[i]);
               cache[i] = new SoftReference<Image>(im);
            }
            System.out.println("Rysowanie obrazka");
            im = null; // Usunięcie silnego odwołania do obrazka.
         }
      }
}
```

Na listingu zadeklarowano klasę `Image`, która symuluje ładowanie obrazka. Każda instancja obrazka jest tworzona przez wywołanie metody klasy `getImage()`. Prywatna tablica `image` instancji zajmuje 100 KB pamięci.

Metoda `main()` najpierw tworzy tablicę obiektów klasy `String`, które zawierają nazwy plików obrazków. Sposób tworzenia tej tablicy jest daleki od wydajnego, a rozwiązanie alternatywne o odpowiedniej wydajności zostanie przedstawione w rozdziale 7.

Metoda `main()` tworzy następnie tablicę obiektów klasy `SoftReference`, która odgrywa rolę podręcznego bufora dla obiektów klasy `Image`. Tablica jest inicjalizowana obiektami klasy `SoftReference`, z których każdy jest inicjalizowany odwołaniem do obiektu klasy `Image`.

Na tym etapie metoda `main()` rozpoczyna wykonanie głównej pętli aplikacji. Pętla iteruje przez bufor i odczytuje kolejne obiekty klasy `Image` lub `null`, jeśli proces odśmiecania wyczyścił miękkie odwołanie do obiektu obrazka (aby zwolnić miejsce zajmowane przez niego na stercie).

Jeżeli odwołaniem przypisanym `im` nie jest `null`, oznacza to, że obiekt nie stał się nieosiągalny — następujący zaraz potem kod może więc wyrysować obrazek na ekranie. Instrukcja przypisania `im = null;` usuwa silne odwołanie do obiektu klasy `Image` ze zmiennej `im` głównego zbioru odwołań.

■ **Uwaga** • Wykonanie instrukcji przypisania `im = null;` nie jest w tej aplikacji konieczne, ponieważ zaraz w następnej iteracji `im` jest nadpisywane przez wartość zwróconą przez metodę `get()` albo pętla i aplikacja zostają zakończone. W przykładowym kodzie instrukcja ta została umieszczona po to, aby pokazać, w jaki sposób można się pozbyć wartości zmiennej `im`. Działanie takie okaże się przydatne, gdy wartość zmiennej pozostanie w pamięci na dłużej, jeśli przykładowa aplikacja zostanie rozbudowana do bardziej złożonej postaci, a proces odśmiecania nie będzie mógł usunąć ze sterty obiektu klasy `Image` wskazywanego przez `im` ze względu na fakt, że obiekt ten będzie silnie osiągalny.

Jeśli odwołanie przypisane zmiennej `im` będzie `null`, będzie to oznaczać, że obiekt klasy `Image` stał się nieosiągalny i prawdopodobnie został usunięty ze sterty. W takiej sytuacji obiekt obrazka trzeba ponownie utworzyć i umieścić w nowym obiekcie klasy `SoftReference`, który powinien następnie trafić do bufora podręcznego.

Poniżej znajduje się krótki wycinek danych wynikowych zwróconych przez aplikację — jej kod być może trzeba będzie odpowiednio dostosować, aby uzyskać takie same wyniki:

```
obrazek162.gif nie znajduje się w buforze
Rysowanie obrazka
obrazek163.gif nie znajduje się w buforze
Rysowanie obrazka
Rysowanie obrazka
```

Komunikat `Rysowanie obrazka` w ostatnim wierszu przytoczonych danych wynikowych oznacza, że obrazek *obrazek164.gif* nadal znajduje się w buforze. Inaczej mówiąc, obiekt klasy `Image` obrazka nadal jest osiągalny.

■ **Uwaga** • Jeżeli aplikacja będzie zwracać nieskończoną liczbę komunikatów „Rysowanie obrazka", może to oznaczać, że przestrzeń na stercie używana przez maszynę wirtualną Javy jest większa niż przestrzeń na stercie wykorzystywana przez aktualnie używaną maszynę wirtualną, gdy aplikacja została uruchomiona w systemie Windows XP. Jeżeli przestrzeń na stercie dostępna dla maszyny wirtualnej będzie odpowiednio duża, miękkie odwołania nie będą czyszczone i aplikacja będzie mogła zwracać wyniki w nieskończoność. Aby rozwiązać ten problem, można zwiększyć rozmiar tablicy obrazków `image` (na przykład z `1024*100` do `1024*500`) i ewentualnie przypisać stałej `NUM_IMAGES` większą wartość (na przykład `500`).

Klasa WeakReference

Klasa `WeakReference` opisuje obiekt klasy `Reference`, którego referent jest słabo osiągalny. `WeakReference` to klasa ogólna, która dziedziczy metody po klasie `Reference`, a także udostępnia konstruktory opisane poniżej, które inicjalizują obiekty klasy `WeakReference`:

- `WeakReference(T r)` umieszcza w obiekcie odwołanie `r`. Obiekt klasy `WeakReference` zachowuje się jak słabe odwołanie do `r`. Z obiektem tym nie jest skojarzona kolejka `ReferenceQueue`.
- `WeakReference(T r, ReferenceQueue<? super T> q)` umieszcza w obiekcie odwołanie `r`. Obiekt klasy `WeakReference` zachowuje się jak słabe odwołanie do `r`. Z obiektem tym zostanie skojarzony obiekt klasy `ReferenceQueue` wskazywany przez argument `q`. Jeżeli jako `q` zostanie przekazane `null`, będzie to oznaczać, że ze słabym odwołaniem nie jest skojarzona żadna kolejka.

Klasa `WeakReference` przydaje się do zapobiegania wyciekom pamięci, które mogą wystąpić w przypadku korzystania z map skrótów. Wyciek pamięci następuje wtedy, gdy do mapy skrótów są dodawane kolejne obiekty, które nie są nigdy usuwane. Obiekty te pozostają wówczas w pamięci, ponieważ mapa skrótów przechowuje silne odwołania do nich.

Idealnym rozwiązaniem byłoby, gdyby obiekty pozostawały w pamięci jedynie wówczas, gdy w aplikacji występują silne odwołania do nich. Z chwilą zniknięcia ostatniego silnego odwołania do obiektu (nie licząc silnego odwołania w mapie skrótów) obiekt powinien zostać usunięty przez proces odśmiecania.

Okolicznościom, w których dochodzi do wycieków pamięci, można zapobiec przez tworzenie słabych odwołań do elementów mapy skrótów. Dzięki temu elementy mapy skrótów zostaną usunięte, jeśli przestaną istnieć jakiekolwiek silne odwołania do ich kluczy. Do tego celu służy klasa `WeakHashmap` opisywana w rozdziale 8.

Klasa PhantomReference

Klasa `PhantomReference` opisuje obiekt klasy `Reference`, którego referent jest złudnie osiągalny. `PhantomReference` jest klasą ogólną, która dziedziczy metody po klasie `Reference` i pokrywa metodę `get()`, a także udostępnia opisany poniżej konstruktor przeznaczony do inicjalizowania obiektów klasy `PhantomReference`.

- `PhantomReference(T r, ReferenceQueue<? super T> q)` umieszcza w obiekcie odwołanie `r`. Obiekt klasy `PhantomReference` zachowuje się jak złudne odwołanie do `r`. Z obiektem klasy `PhantomReference` jest skojarzony obiekt klasy `ReferenceQueue` wskazywany przez argument `q`. Przekazanie `null` do argumentu `q` nie ma sensu, ponieważ pokrywająca metoda `get()` zwraca `null` i obiekt klasy `PhantomReference` nigdy nie zostanie zakolejkowany.

W odróżnieniu od obiektów klas `WeakReference` i `SoftReference`, które są kolejkowane w ich własnych kolejkach odwołań wówczas, gdy ich referenty stają się słabo osiągalne (przed finalizacją), a czasami także po tym, gdy ich referenty stają się miękko osiągalne (przed finalizacją), obiekty klasy `PhantomReference` są kolejkowane po odzyskaniu pamięci zajmowanej przez ich referenty.

Pomimo że referent obiektu klasy `PhantomReference` jest niedostępny (metoda `get()` zwraca `null`), klasa jest jak najbardziej przydatna, ponieważ zakolejkowanie obiektu klasy `PhantomReference` wskazuje dokładnie moment usunięcia referenta. Można więc na przykład opóźnić utworzenie dużego obiektu do momentu, aż inny duży obiekt zostanie usunięty z pamięci (i uniknąć w ten sposób rzucenia obiektu klasy `java.lang.OutOfMemoryError`).

Klasa `PhantomReference` jest przydatna również dlatego, że może stanowić substytut **wskrzeszenia** (ang. *resurrection*), czyli zjawiska, w którym obiekt nieosiągalny ponownie staje się osiągalny. Jako że referent jest niedostępny (metoda `get()` zwraca `null`) ze względu na fakt, że nie ma go już w pamięci w momencie, gdy obiekt klasy `PhantomReference` jest kolejkowany, obiekt może zostać wyczyszczony w pierwszym cyklu pracy procesu odśmiecania, w którym obiekt ten zostanie zidentyfikowany jako złudnie osiągalny. Po uzyskaniu odpowiedniego powiadomienia za pośrednictwem kolejki odwołań obiektu klasy `Phantom` ↪`Reference` można wyczyścić wszystkie zasoby powiązane z obiektem.

Uwaga • Wskrzeszenie zachodzi w metodzie `finalize()`, gdy zmiennej z głównego zbioru odwołań przypisuje się `this`. W metodzie `finalize()` można na przykład wykonać instrukcję `r = this;`, aby obiekt nieosiągalny wskazywany jako `this` przypisać polu klasy o nazwie `r`.

Z kolei proces odśmiecania potrzebuje co najmniej dwóch cykli, aby stwierdzić, czy obiekt pokrywający metodę `finalize()` może zostać przez ten proces przetworzony. Jeśli w pierwszym cyklu odśmiecania okaże się, że obiekt może zostać usunięty, zostanie wywołana metoda `finalize()`. Jednak metoda `finalize()` może doprowadzić do wskrzeszenia obiektu, dlatego potrzebny jest jeszcze drugi cykl, w którym proces odśmiecania sprawdzi, czy do wskrzeszenia rzeczywiście doszło.

■ **Ostrzeżenie** • Zjawisko wskrzeszania wykorzystano w implementacji pul obiektów, które odtwarzają te same obiekty, jeśli ich utworzenie jest kosztowne (czasochłonne). Przykładem takiego obiektu może być choćby połączenie z bazą danych. Jednak wskrzeszanie też jest czynnością czasochłonną, a klasa `PhantomReference` likwiduje wymóg wykonywania tej czynności, a zatem zdecydowanie należy unikać wskrzeszania we własnych aplikacjach.

Na listingu 6.15 pokazano, w jaki sposób za pomocą klasy `PhantomReference` można zidentyfikować fakt usunięcia dużego obiektu.

Listing 6.15. *Rozpoznawanie, czy usunięty został duży obiekt*

```
class LargeObject
{
   private byte[] memory = new byte[1024*1024*50]; // 50 megabajtów
}
public class LargeObjectDemo
{
   public static void main(String[] args)
   {
      ReferenceQueue<LargeObject> rq;
      rq = new ReferenceQueue<LargeObject>();
      PhantomReference<LargeObject> pr;
      pr = new PhantomReference<LargeObject>(new LargeObject(), rq);
      int counter = 0;
      int[] x;
      while (rq.poll() == null)
      {
         System.out.println("oczekiwanie na usunięcie dużego obiektu");
         if (counter++ == 10)
            x = new int[1024*1024];
      }
      System.out.println("duży obiekt został usunięty");
   }
}
```

W kodzie na listingu 6.15 zadeklarowano klasę `LargeObject`, której prywatna tablica `memory` zajmuje 50 MB. Jeżeli w momencie uruchomienia aplikacji używana implementacja Javy rzuci błąd `OutOfMemoryError`, konieczne będzie zmniejszenie rozmiaru tej tablicy.

Metoda `main()` tworzy najpierw obiekt klasy `ReferenceQueue` opisujący kolejkę, do której zostanie dodany utworzony w kolejnym kroku obiekt klasy `PhantomReference` zawierający odwołanie do `LargeObject`.

Następnie metoda `main()` tworzy obiekt klasy `PhantomReference`. W tym celu do konstruktora zostaje przekazane odwołanie do nowo utworzonego obiektu klasy `LargeObject` oraz odwołanie do wcześniej utworzonego obiektu klasy `ReferenceQueue`.

Po zainicjalizowaniu zmiennej `counter` (która wskazuje liczbę iteracji, jakie wykonano przed utworzeniem kolejnego dużego obiektu) oraz po zadeklarowaniu zmiennej lokalnej o nazwie `x`, przeznaczonej do przechowywania silnego odwołania do kolejnego dużego obiektu, metoda `main()` uruchamia pętlę odpytywania.

Pętla odpytywania rozpoczyna się od wywołania metody `poll()`, która sprawdza, czy obiekt klasy `LargeObject` został usunięty z pamięci. Dopóki metoda będzie zwracać `null`, co

oznacza, że obiekt klasy `LargeObject` nadal znajduje się w pamięci, pętla będzie zwracać komunikat i zwiększać wartość licznika `counter`.

Gdy licznik iteracji `counter` osiągnie wartość 10, zmiennej `x` zostaje przypisana tablica elementów typu `int` z milionem elementów w postaci liczb całkowitych. Tablica nie zostanie wyczyszczona przez proces odśmiecania do momentu zakończenia działania aplikacji, ponieważ odwołanie przypisane zmiennej `x` jest odwołaniem silnym.

Na niektórych platformach przypisanie odwołania do tablicy zmiennej `x` wystarczy do tego, by proces odśmiecania zniszczył obiekt klasy `LargeObject`. Obiekt klasy `PhantomReference` dużego obiektu jest kolejkowany w kolejce `ReferenceQueue`, na którą wskazuje odwołanie `rq`, a metoda `poll()` zwraca obiekt klasy `PhantomReference`.

W zależności od implementacji używanej maszyny wirtualnej aplikacja może zwrócić (choć nie musi) komunikat `duży obiekt został usunięty`. Jeżeli komunikat ten nie zostanie zwrócony przez aplikację, być może trzeba będzie zwiększyć rozmiar tablicy `x`, aby w ten sposób zapewnić, że nie zostanie rzucony błąd `OutOfMemoryError`.

Po uruchomieniu aplikacji można uzyskać dane wynikowe widoczne poniżej. Aby uzyskać podobny wynik, konieczne może być odpowiednie dostosowanie kodu źródłowego aplikacji.

```
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
oczekiwanie na usunięcie dużego obiektu
duży obiekt został usunięty
```

■ **Uwaga** • Więcej ciekawych przykładów zastosowania klasy `PhantomReference` oraz więcej bardziej szczegółowych informacji na temat procesu odśmiecania można znaleźć we wpisie na blogu Keitha D. Gregory'ego pod tytułem „Java Reference Objects" (*http://www.kdgregory.com/index.php?page=java.refobj*).

Ćwiczenia

Celem poniższych ćwiczeń jest sprawdzenie wiedzy wyniesionej z tego rozdziału, dotyczącej podstawowych API języka Java.

1. Jakie stałe deklaruje `Math`?
2. Dlaczego `Math.abs(Integer.MIN_VALUE)` jest równe `Integer.MIN_VALUE`?
3. Do czego służy metoda `random()` klasy `Math`?
4. Należy wskazać pięć wartości specjalnych, które mogą wystąpić w trakcie wykonywania obliczeń zmiennopozycyjnych.
5. Czym różnią się klasy `Math` i `StrictMath`?
6. Do czego służy słowo `strictfp`?
7. Co to jest `BigDecimal` i do czego się używa tej klasy?

8. Która stała `RoundingMode` opisuje tryb zaokrąglania, o którym mówi się w szkole na lekcjach?
9. Co to jest `BigInteger`?
10. Do czego służy metoda `isSealed()` klasy `Package`?
11. Metoda `getPackage()` wymaga, by z pakietu załadowano co najmniej jeden plik klas, zanim zostanie zwrócony obiekt klasy `Package` opisujący ten pakiet. Prawda czy fałsz?
12. Należy wskazać dwa najważniejsze zastosowania podstawowych klas opakowujących.
13. Dlaczego powinno się unikać implementowania wyrażeń w stylu `ch >= '0' && ch <= '9'` (do sprawdzenia, czy `ch` zawiera cyfry) albo `ch >= 'A' && ch <= 'Z'` (aby sprawdzić, czy `ch` zawiera wielką literę)?
14. Należy wskazać cztery rodzaje osiągalności.
15. Co to jest referent?
16. Która z klas API Reference jest odpowiednikiem metody `finalize()` klasy `Object`?
17. Zanim rozpowszechniły się ekrany graficzne, czasami programiści wyświetlali kształty na ekranach tekstowych. Na przykład okrąg mógł być prezentowany w następujący sposób:

```
         *
      *********
     **       **
   **           **
   *             *
  *               *
 **               **
 *                 *
 *                 *
 *                 *
**                 **
 *                 *
 *                 *
 *                 *
 **               **
  *               *
   *             *
   **           **
     **       **
      *********
         *
```

■ **Uwaga** • Powyższa figura ma kształt eliptyczny, a nie okrągły, ponieważ wyświetlana wysokość każdej gwiazdki jest większa niż jej wyświetlana szerokość. Gdyby wysokość i szerokość były takie same, figura miałaby kształt okręgu.

Należy zaimplementować aplikację `Circle`, która będzie generować i wyświetlać przedstawiony powyżej okrąg. Najpierw aplikacja powinna tworzyć dwuwymiarową tablicę `screen` złożoną z 22 wierszy i takiej samej liczby kolumn. Każdy element tablicy trzeba zainicjalizować znakiem spacji (który wskazuje czysty ekran). Dla każdego konta o mierze całkowitoliczbowej z przedziału od 0 do 360 należy obliczyć współrzędne `x` i `y` w ten sposób, że promień o wartości 10 będzie mnożony przez sinus i kosinus kąta. Do wartości `x` i `y` należy dodać 11, aby kształt okręgu umieścić w środku tablicy `screen`. Punktowi o wynikowych współrzędnych `(x,y)` należy przypisać znak gwiazdki. Gdy wykonywanie pętli dobiegnie końca, należy zwrócić zawartość tablicy do standardowego wyjścia.

18. **Liczba pierwsza** to dodatnia liczba całkowita, która jest podzielna tylko przez jeden i przez siebie samą. Należy utworzyć aplikację o nazwie `PrimeNumberTest`, która będzie sprawdzać, czy przekazany do niej pojedynczy argument całkowitoliczbowy wskazuje liczbę pierwszą, czy inną liczbę, i wyświetli stosowny komunikat. Na przykład aplikacja wywołana poleceniem `java PrimeNumber ↪Test 289` powinna zwrócić komunikat „289 nie jest liczbą pierwszą". Aby w prosty sposób sprawdzić, czy liczba jest liczbą pierwszą, należy przejść w pętli od liczby 2 do pierwiastka kwadratowego argumentu całkowitoliczbowego i wykorzystać w tej pętli operator reszty z dzielenia. Za jego pomocą można sprawdzić, czy argument jest podzielny przez indeks pętli. Na przykład ze względu na to, że wyrażenie `6%2` ma wartość 0 (6 jest podzielne przez 2), liczba całkowita 6 nie jest liczbą pierwszą.

Podsumowanie

Klasa `java.lang.Math` uzupełnia standardowe operacje matematyczne (realizowane przez operatory +, -, *, / oraz %) o możliwość wykonywania zaawansowanych operacji (na przykład obliczeń trygonometrycznych). Towarzysząca jej klasa `java.lang.StrictMath` zapewnia, że wszystkie operacje zaawansowane dadzą ten sam wynik na wszystkich platformach.

Wartości pieniężne nigdy nie powinny być reprezentowane przez wartości zmiennoprzecinkowe ani zmiennoprzecinkowe z podwójną precyzją, ponieważ nie wszystkie wartości pieniężne mają precyzyjną reprezentację. Natomiast klasa `java.math.BigDecimal` zapewnia dokładną prezentację takich wartości oraz wykonywanie na nich precyzyjnych obliczeń.

Klasa `BigDecimal` wykorzystuje klasę `java.math.BigInteger` do reprezentowania swojej wartości niewyskalowanej. Instancja klasy `BigInteger` opisuje wartość całkowitoliczbową, która może być wartością o dowolnej długości (ograniczonej jedynie limitami pamięci dostępnej dla maszyny wirtualnej).

Klasa `java.lang.Package` daje dostęp do informacji na temat pakietów. Informacje te zawierają wersję implementacji i specyfikacji pakietu języka Java, nazwę pakietu, a także wskazanie, czy pakiet jest upakowany, czy nie.

Instancje podstawowych klas opakowujących `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` i `Short` z pakietu `java.lang` stanowią opakowania dla wartości typów podstawowych. Klasy te przydają się do przechowywania wartości typów podstawowych w kolekcjach.

API References pozwala aplikacji na interakcję — w ograniczonym zakresie — z procesem odśmiecania. Pakiet `java.lang.ref` tego API zawiera klasy `Reference`, `ReferenceQueue`, `SoftReference`, `WeakReference` oraz `PhantomReference`.

Klasa `SoftReference` przydaje się do implementowania bufora obrazków, za pomocą klasy `WeakReference` można zapobiegać wyciekom pamięci potencjalnie zachodzącym w trakcie pracy z mapami skrótów, zaś dzięki `PhantomReference` można sprawdzać, czy obiekt został już zniszczony i czy w związku z tym można wyczyścić powiązane z nim zasoby.

Zakończyliśmy dopiero pierwszy etap poznawania podstawowych interfejsów API języka Java. W rozdziale 7. zapoznamy się z kolejnym podstawowym API — Reflection API, interfejsem do zarządzania ciągami znaków, klasą `System`, a także niskopoziomowym API Threading.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

Język Java, obecny na rynku od dobrych piętnastu lat, ugruntował już swoją pozycję w środowisku programistów. Wykorzystywany jest niemal w każdej dziedzinie informatycznego świata, od aplikacji internetowych, poprzez tradycyjne oprogramowanie biurowe, aż po rozwiązania dla urządzeń przenośnych. Został on również wybrany jako główny język platformy Android — aplikacje dla tego systemu pisane są w języku Java i korzystają z różnorodnych standardowych API tego języka. Dzięki temu tysiące programistów niemal z marszu rozpoczęło tworzenie aplikacji przeznaczonych dla systemu Android, właściwie nie ponosząc żadnych dodatkowych kosztów.

Jedną z najlepszych książek wprowadzających do języka Java... trzymasz właśnie w rękach. Dzięki niej błyskawicznie opanujesz język Java oraz zasady programowania obiektowego, skupiając się przede wszystkim na tych aspektach Javy, które pozwolą Ci zrozumieć istotę tworzenia aplikacji. Z tak solidnymi fundamentami zaczniesz odkrywać możliwości i ograniczenia Javy. Na kolejnych stronach znajdziesz szczegółowo omówione API platformy, jej potencjał w zakresie korzystania z kolekcji oraz tworzenia aplikacji wielojęzycznych. Ponadto nauczysz się wykonywać operacje wejścia-wyjścia. Książka ta jest idealną pozycją dla wszystkich osób chcących poznać niuanse języka Java, a następnie wykorzystać je podczas tworzenia aplikacji. Dzięki zawartym w niej ćwiczeniom błyskawicznie zweryfikujesz zdobytą wiedzę.

- **Java — język programowania i platforma**
- **Instalacja i korzystanie ze środowiska programistycznego Eclipse i NetBeans**
- **Elementarz języka JAVA — klasy, interfejsy, dziedziczenie**
- **Zaawansowane elementy języka — wyjątki, adnotacje, typy ogólne i wyliczeniowe**
- **Zastosowanie wątków**
- **Zarządzanie kolekcjami elementów — Collections Framework**
- **Operacje wejścia-wyjścia — dostęp do plików i strumieni**

Wykorzystaj potencjał Javy i przygotuj się do tworzenia aplikacji na urządzenia przenośne!

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-3372-2

Cena: 89,00 zł

Informatyka w najlepszym wydaniu