Helion

Java
Kompendium programisty

Wydanie XII

Herbert Schildt

McGraw Hill

Tytuł oryginału: Java: The Complete Reference, Twelfth Edition

Tłumaczenie: Piotr Rajca, Mikołaj Szczepaniak, Rafał Jońca

ISBN: 978-83-8322-156-4

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/javk12
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

CZĘŚĆ IV
Stosowanie Javy w praktyce

ROZDZIAŁ

7

Dokładniejsze omówienie metod i klas

Niniejszy rozdział kontynuuje omówienie metod i klas, zapoczątkowane w poprzednim rozdziale. Porusza kilka interesujących tematów związanych z metodami, między innymi ich przeciążanie, przekazywanie parametrów i rekurencję. W dalszej części wrócimy do klas, a dokładnie do zarządzania dostępem, zastosowania słowa kluczowego `static` i jednej z podstawowych klas Javy — `String`.

Przeciążanie metod

W języku Java można zdefiniować w jednej klasie dwie lub więcej metod o takiej samej nazwie, o ile posiadają one inne deklaracje parametrów. W takim przypadku mówi się o **przeciążaniu metody**. Przeciążanie metod to jeden ze sposobów obsługi polimorfizmu w języku Java. Jeśli nigdy wcześniej nie używało się języka programowania dopuszczającego przeciążanie, koncepcja może się wydawać dziwaczna. W praktyce jednak nie jest szczególnie trudna i co ważniejsze, znacząco ułatwia użytkownikom korzystanie z metod.

W momencie wywołania przeciążonej metody Java potrzebuje typu i (lub) liczby argumentów do określenia, którą tak naprawdę wersję przeciążonej metody należy wykonać. Choć przeciążone metody mogą zwracać wartości różnych typów, znaczenie ma jedynie liczba i typ argumentów. Z tego powodu przeciążane metody muszą różnić się parametrami, a nie tylko typem zwracanej wartości. Java w momencie napotkania wywołania przeciążonej metody szuka wersji najlepiej pasującej do przekazywanych argumentów i wykonuje jej kod.

Następujący przykład obrazuje przeciążanie metod.

```
// R07\Overload.java

// Przykład przeciążania metod
class OverloadDemo {
  void test() {
    System.out.println("Brak parametrów");
  }

  // Przeciążona metoda test z jednym parametrem typu int
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Przeciążona metoda test z dwoma parametrami
  void test(int a, int b) {
    System.out.println("a i b: " + a + " " + b);
  }
```

```
  // Przeciążona metoda test z jednym parametrem typu double
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

class Overload {
  public static void main(String[] args) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // Wywołanie wszystkich wersji metody test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Wynik wykonania ob.test(123.25): " + result);
  }
}
```

Program generuje następujące wyniki.

```
Brak parametrów
a: 10
a i b: 10 20
double a: 123.25
Wynik wykonania ob.test(123.25): 15190.5625
```

Metoda `test()` jest przeciążona cztery razy. Pierwsza wersja nie przyjmuje żadnych parametrów, druga przyjmuje jeden parametr typu `int`, trzecia przyjmuje dwa parametry, a czwarta przyjmuje jeden parametr typu `double`. Dodatkowo ostatnia wersja zwraca wartość typu `double`, ale nie wpływa to w żaden sposób na wybór odpowiedniej metody.

W momencie wywołania przeciążonej metody Java szuka dopasowania przekazanych argumentów z przyjmowanymi parametrami. Nie zawsze dopasowanie musi być dokładne. Pod uwagę brane są również typy zgodne. Rozważmy następujący program.

```
// R07\Overload2.java
// Przy wyborze przeciążanej metody Java stosuje również automatyczną konwersję typów
class OverloadDemo {
  void test() {
    System.out.println("Brak parametrów");
  }

  // Przeciążona wersja dla dwóch parametrów
  void test(int a, int b) {
    System.out.println("a i b: " + a + " " + b);
  }

  // Przeciążona wersja dla jednego parametru typu double
  void test(double a) {
    System.out.println("Wewnątrz test(double) a: " + a);
  }
}

class Overload2 {
  public static void main(String[] args) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i); // Spowoduje wywołanie test(double)
    ob.test(123.2); // Spowoduje wywołanie test(double)
  }
}
```

Program po uruchomieniu generuje następujące wyniki.

```
Brak parametrów
a i b: 10 20
Wewnątrz test(double) a: 88.0
Wewnątrz test(double) a: 123.2
```

Przedstawiona wersja klasy `OverloadDemo` nie definiuje metody `test(int)`. Z tego powodu wywołanie metody `test()` z argumentem typu `int` nie powoduje znalezienia idealnie pasującej wersji. Ponieważ jednak Java dokonuje automatycznej konwersji typu `int` na `double`, zostanie znaleziona metoda `test(double)`. Zostanie jej przekazana wersja argumentu po konwersji do typu double. Oczywiście gdyby istniała wersja `test(int)`, to ona zostałaby wykonana dla argumentu typu `int`. Java stosuje automatyczną konwersję tylko wtedy, gdy nie istnieje idealne dopasowanie.

Przeciążanie metod to element polimorfizmu, gdyż to jedyne rozwiązanie zapewniające spełnienie w Javie paradygmatu „jeden interfejs, wiele metod". Warto zauważyć następującą kwestię: w języku programowania nieobsługującym przeciążania metod trzeba by nadać każdej metodzie unikatową nazwę. Sytuacja, w której chce się zaimplementować tę samą metodę dla wielu różnych danych, pojawia się stosunkowo często. Rozważmy sytuację z metodą zwracającą wartość bezwzględną. W językach, które nie obsługują przeciążania, poszczególne wersje muszą mieć odrobinę inne nazwy (w języku C funkcja `abs()` dotyczy liczb całkowitych, `labs()` liczb `long`, natomiast `fabs()` liczb zmiennoprzecinkowych). Jak łatwo zauważyć, nie jest to rozwiązanie wygodne, a przy większych projektach staje się znaczącym problemem — choć wszystkie wersje funkcji wykonują to samo zadanie, trzeba pamiętać trzy nazwy. W Javie przedstawiony problem nie występuje — istnieje tylko jedna nazwa dla metody zwracającej wartość bezwzględną. Jest to metoda `abs()` z klasy `Math`. Klasa zawiera wiele przeciążonych wersji tej metody obsługujących wszystkie podstawowe typy numeryczne. Java identyfikuje właściwą wersję metody `abs()` na podstawie typu argumentu.

Głównym zadaniem przeciążania jest zebranie wszystkich powiązanych ze sobą metod pod jedną nazwą. Innymi słowy, `abs` oznacza ogólną **akcję wykonywaną** dla liczb. To na kompilator spada odpowiedzialność co do wyboru **odpowiedniej** wersji metody dla danych argumentów. My, programiści, musimy pamiętać jedynie ogólną operację, którą chcemy wykonać. Dzięki polimorfizmowi kilka nazw redukuje się do jednej. Choć przedstawiony przykład jest stosunkowo prosty, pozwala zauważyć, iż przeciążanie pomaga lepiej opanować zagadnienia o znacznej złożoności.

Każda z przeciążonych wersji metody może wykonywać dowolne zadania. Nie istnieje zasada mówiąca, iż przeciążone metody muszą działać podobnie. W praktyce jednak wzajemne powiązanie wersji występuje bardzo często, gdyż właśnie tego oczekuje użytkownik. Innymi słowy, nie ma wielkiego sensu, by metoda o nazwie `sqr()` zwracała liczbę całkowitą podniesioną do kwadratu i **pierwiastek kwadratowy** dla liczb zmiennoprzecinkowych, gdyż obie operacje są sobie przeciwne. Takie zastosowanie przeciążenia przeczyłoby jego podstawowej idei. Z przedstawionych powodów przeciążania metod używa się jedynie dla powiązanych ze sobą operacji.

Przeciążanie konstruktorów

Przeciążanie nie dotyczy tylko metod — może być stosowane również dla konstruktorów. W zasadzie w większości programów Javy przeciążone konstruktory to raczej norma niż wyjątek. Aby lepiej zrozumieć, dlaczego przeciążanie konstruktorów jest tak popularne, przyjrzyjmy się jeszcze raz przykładowi z pudełkami z poprzedniego programu. Oto wcześniej stosowana wersja klasy `Box`.

// R07\Box.java

```
class Box {
  double width;
  double height;
  double depth;

  // Konstruktor klasy Box
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

```
  // Oblicz i zwróć objętość
  double volume() {
    return width * height * depth;
  }
}
```

Konstruktor `Box()` wymaga przekazania trzech parametrów. Oznacza to, że przy tworzeniu każdego z obiektów klasy `Box` trzeba podać początkowe wymiary pudełka. Dla powyższej klasy następujący wiersz nie jest poprawny.

```
Box ob = new Box();
```

Ponieważ konstruktor wymaga podania trzech argumentów, wywołanie go bez argumentów jest błędem. Pojawia się następujący problem: w jaki sposób utworzyć pudełko, gdy nie interesują nas początkowe wymiary lub ich nie znamy? A co, jeśli chcemy utworzyć sześcian, więc wystarczyłoby podanie tylko jednej wartości dla wszystkich trzech wymiarów? W aktualnej wersji klasy `Box` oba te podejścia nie są dostępne.

Na szczęście łatwo rozwiązać przedstawione problemy: wystarczy przeciążyć konstruktor klasy `Box`, aby spełniał wymienione wcześniej warunki. Poniżej znajduje się ulepszona wersja klasy `Box` zawierająca trzy wersje konstruktora.

```
// R07\OverloadCons.java

/* Klasa Box zawiera trzy konstruktory, które
    na różne sposoby inicjalizują wymiary pudełka
*/
class Box {
  double width;
  double height;
  double depth;

  // Konstruktor używany, gdy podano wszystkie wymiary
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Konstruktor używany przy braku wymiarów
  Box() {
    width = -1;  // Wartość -1
    height = -1; // wskazuje
    depth = -1;  // niezainicjalizowane pudełko
  }

  // Konstruktor używany do tworzenia sześcianów
  Box(double len) {
    width = height = depth = len;
  }

  // Oblicz i zwróć objętość
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons {
  public static void main(String[] args) {
    // Utworzenie pudełek za pomocą różnych konstruktorów
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    double vol;

    // Pobranie objętości pierwszego pudełka
    vol = mybox1.volume();
```

```
    System.out.println("Objętość mybox1 wynosi " + vol);

    // Pobranie objętości drugiego pudełka
    vol = mybox2.volume();
    System.out.println("Objętość mybox2 wynosi " + vol);

    // Pobranie objętości sześcianu
    vol = mycube.volume();
    System.out.println("Objętość mycube wynosi " + vol);
  }
}
```

Wynik działania programu jest następujący.

```
Objętość mybox1 wynosi 3000.0
Objętość mybox2 wynosi -1.0
Objętość mycube wynosi 343.0
```

Jak łatwo zauważyć, na podstawie argumentów podanych w operatorze `new` został wybrany odpowiedni przeciążony konstruktor.

Obiekty jako parametry

Do tej pory jako parametrów używaliśmy jedynie typów prostych. Często jednak do metod przekazuje się obiekty. Rozważmy następujący program.

```
// R07\PassOb.java

// Do metod można przekazywać obiekty
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // Zwraca true, jeśli obiekt o jest równy aktualnemu obiektowi
  boolean equalTo(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String[] args) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
    System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
  }
}
```

Program generuje następujące wyniki.

```
ob1 == ob2: true
ob1 == ob3: false
```

Metoda `equalTo()` klasy `Test` porównuje dwa obiekty pod względem równości i zwraca wynik takiego porównania. Innymi słowy, porównuje aktualny obiekt z przekazanym jako parametr. Jeśli obiekty zawierają te same wartości, metoda zwraca wartość `true`. W przeciwnym razie zwraca wartość `false`. Zauważ, że jako typ parametru metody `equalTo()` pojawia się nazwa klasy `Test`. Choć klasa ta jest definiowana bezpośrednio w programie, Java traktuje ją dokładnie tak samo jak typy wbudowane.

Bardzo często parametry obiektowe pojawiają się w konstruktorach. Na przykład zależy nam na tym, by nowy obiekt został początkowo ustawiony na takie same wartości jak inny obiekt. Poniższa wersja klasy Box dopuszcza inicjalizację nowego obiektu Box innym obiektem tej samej klasy.

```
// R07\OverloadCons2.java

// Jeden obiekt Box może posłużyć do inicjalizacji innego obiektu Box

class Box {
  double width;
  double height;
  double depth;

  // Tworzenie klonu obiektu
  Box(Box ob) { // Przekazanie obiektu do konstruktora
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // Konstruktor używany, gdy podano wszystkie wymiary
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // Konstruktor używany przy braku wymiarów
  Box() {
    width = -1;  // Wartość -1
    height = -1; // wskazuje
    depth = -1;  // niezainicjalizowane pudełko
  }

  // Konstruktor używany do tworzenia sześcianów
  Box(double len) {
    width = height = depth = len;
  }

  // Oblicz i zwróć objętość
  double volume() {
    return width * height * depth;
  }
}

class OverloadCons2 {
  public static void main(String[] args) {
    // Tworzenie pudełek za pomocą różnych konstruktorów
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    Box myclone = new Box(mybox1); // Tworzenie kopii obiektu mybox1

    double vol;

    // Pobranie objętości pierwszego pudełka
    vol = mybox1.volume();
    System.out.println("Objętość mybox1 wynosi " + vol);

    // Pobranie objętości drugiego pudełka
    vol = mybox2.volume();
    System.out.println("Objętość mybox2 wynosi " + vol);

    // Pobranie objętości sześcianu
    vol = mycube.volume();
    System.out.println("Objętość mycube wynosi " + vol);
```

```
    // Pobranie objętości klonu
    vol = myclone.volume();
    System.out.println("Objętość clone wynosi " + vol);
  }
}
```

Każdy programista, który zaczyna tworzyć własne klasy, szybko odkrywa, że definiowanie wielu form konstruktorów nierzadko decyduje o efektywności obiektów i wygodzie ich używania.

Dokładniejsze omówienie przekazywania argumentów

Ogólnie w językach programowania istnieją dwa sposoby przekazywania wartości do podprogramów. Pierwszy nosi nazwę **przekazywania przez wartość**. Powoduje to skopiowanie **wartości** argumentu do zmiennej będącej parametrem podprogramu. Z tego względu modyfikacje parametru wykonane w podprogramie nie mają wpływu na argument. Drugi sposób to **przekazywanie przez referencję**. W tym podejściu do zmiennej parametru wpisywana jest referencja do argumentu (zamiast wartości argumentu). Podprogram używa referencji do odczytania wartości przekazanego argumentu. Z tego względu podprogram może zmienić zawartość przekazanego argumentu. Java stosuje oba sposoby w zależności od tego, co jest przekazywane.

W Javie typy proste są przekazywane do metod przez wartość, więc zmiana parametru w metodzie nie wpływa na zawartość argumentu poza nią. Rozważmy następujący program.

```
// R07\CallByValue.java

// Typy proste są przekazywane przez wartość
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}

class CallByValue {
  public static void main(String[] args) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a i b przed wywołaniem: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a i b po wywołaniu: " +
                       a + " " + b);
  }
}
```

Wyniki działania programu są następujące.

```
a i b przed wywołaniem: 15 20
a i b po wywołaniu: 15 20
```

Operacje wykonywane wewnątrz metody `meth()` w żaden sposób nie wpłynęły na wartości zmiennych `a` i `b` użytych w wywołaniu metody. Nie zostały im przypisane wartości 30 i 10.

Gdy do metody przekazuje się obiekt, sytuacja ulega dramatycznej zmianie, ponieważ przekazywanie obiektów odbywa się przez referencję. Pamiętaj, że zmienna typu klasowego przechowuje tak naprawdę tylko referencję do obiektu. Przekazanie tej referencji do metody powoduje, że parametr będzie wskazywał na ten sam obiekt, na który wskazywał argument w wywołaniu. Oznacza to, że zmiana obiektu wewnątrz metody *wpłynie* na obiekt przekazany jako argument. Rozważmy następujący program.

```
// R07\PassObjRef.java

// Obiekty są przekazywane przez referencję

class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // Przekazanie obiektu
  void meth(Test o) {
    o.a *=  2;
    o.b /= 2;
  }
}

class PassObjRef {
  public static void main(String[] args) {
    Test ob = new Test(15, 20);

    System.out.println("ob.a i ob.b przed wywołaniem: " +
                       ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a i ob.b po wywołaniu: " +
                       ob.a + " " + ob.b);
  }
}
```

Program po wykonaniu generuje następujące wyniki.

```
ob.a i ob.b przed wywołaniem: 15 20
ob.a i ob.b po wywołaniu: 30 10
```

Jak łatwo zauważyć, instrukcje zawarte wewnątrz metody `meth()` spowodowały modyfikację obiektu przekazanego jako argument.

Gdy dochodzi do przekazywania referencji do obiektu w wywołaniu metody, sama referencja jest przekazywana przez wartość. Ponieważ jednak wskazuje ona pewien obiekt, jej kopia zapisana do parametru także dotyczy tego obiektu.

Zwracanie obiektów

Metoda może zwrócić dowolny typ danych, w tym typ klasowy. Na przykład w poniższym przykładzie metoda `incrByTen()` zwraca obiekt, którego wartość `a` została zwiększona o 10 w stosunku do obiektu, dla którego metoda została wywołana.

```
// R07\RetOb.java

// Zwrócenie obiektu
class Test {
  int a;

  Test(int i) {
    a = i;
  }

  Test incrByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}
```

```
class RetOb {
  public static void main(String[] args) {
    Test ob1 = new Test(2);
    Test ob2;

    ob2 = ob1.incrByTen();
    System.out.println("ob1.a: " + ob1.a);
    System.out.println("ob2.a: " + ob2.a);

    ob2 = ob2.incrByTen();
    System.out.println("ob2.a po drugim zwiększeniu: "
                       + ob2.a);
  }
}
```

Program generuje następujące wyniki.

```
ob1.a: 2
ob2.a: 12
ob2.a po drugim zwiększeniu: 22
```

W każdym wywołaniu metody `incrByTen()` powstaje nowy obiekt i zwracana jest referencja do niego.

Po analizie powyższego programu warto zwrócić uwagę na istotny fakt: ponieważ wszystkie obiekty są alokowane dynamicznie za pomocą operatora `new`, nie trzeba przejmować się utratą zasięgu obiektu spowodowaną zakończeniem metody, w której został zdefiniowany. Obiekt będzie istniał tak długo, jak istnieje gdzieś chociaż jedna referencja do niego. Jeśli nie będzie żadnych referencji, mechanizm odzyskiwania pamięci zniszczy taki obiekt.

Rekurencja

Java obsługuje **rekurencję**. Rekurencja polega na odwołaniu się do siebie samego w trakcie wykonywania obliczeń. W kontekście programowania rekurencja oznacza, że metoda może wywołać samą siebie. Taką metodę nazywa się **metodą rekurencyjną**.

Klasycznym przykładem rekurencji jest obliczanie silni z liczby. Silnia z liczby *N* to wynik mnożenia wszystkich liczb całkowitych od 1 do *N*. Na przykład silnia z 3 (oznaczana jako 3!) to wynik działania 1×2×3 = 6. Poniżej znajduje się metoda wykorzystująca rekurencję do obliczenia silni.

```
// R07\Recursion.java

// Prosty przykład rekurencji
class Factorial {
  // To jest metoda rekurencyjna
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String[] args) {
    Factorial f = new Factorial();

    System.out.println("3! wynosi " + f.fact(3));
    System.out.println("4! wynosi " + f.fact(4));
    System.out.println("5! wynosi " + f.fact(5));
  }
}
```

Wyniki działania programu są następujące.

```
3! wynosi 6
4! wynosi 24
5! wynosi 120
```

Jeśli wcześniej nie korzystało się z funkcji rekurencyjnych, zapis metody `fact()` może wydawać się nieco dziwny. Oto sposób działania metody. Gdy argumentem metody `fact()` będzie wartość 1, metoda zwróci wartość 1. W przeciwnym razie zwróci wynik działania `fact(n-1)*n`. W celu obliczenia tego wyrażenia metoda jest wywoływana ponownie z argumentem `n-1`. Cały proces powtarza się aż do momentu, w którym `n` będzie równe 1 — wtedy nastąpi zwrócenie wszystkich wartości.

Aby lepiej zrozumieć sposób działania metody `fact()`, wykorzystajmy prosty przykład. W momencie obliczenia silni dla wartości 3 wywołanie `fact(3)` spowoduje przeprowadzenie drugiego wywołania z argumentem równym 2. To wywołanie spowoduje ponowne wywołanie metody `fact()`, ale z argumentem 1. Wynikiem `fact(1)` jest wartość 1, która następnie zostanie pomnożona przez wartość 2 (wartość `n` w drugim wywołaniu). Wynik (wartość 2) zostanie zwrócony do oryginalnego wywołania metody i pomnożony przez 3 (oryginalna wartość `n`). Spowoduje to zwrócenie wartości 6. Warto umieścić w tej metodzie wywołanie metody `println()`, aby prześledzić pośrednie odpowiedzi na poszczególnych poziomach rekurencji.

Gdy metoda wywołuje samą siebie, na stosie alokowane są nowe lokalne zmienne i parametry, a kod metody jest wykonywany od początku z nowymi wartościami. Przy powrocie stare zmienne lokalne i parametry są usuwane ze stosu oraz dochodzi do wznowienia działania metody, która dokonała wywołania rekurencyjnego. Metody rekurencyjne przypominają w działaniu rozkładaną lunetę (najpierw są rozwijane, a następnie zwijane).

Wersje rekurencyjne wielu algorytmów mogą działać nieco wolniej od ich odpowiedników iteracyjnych, ponieważ dodatkowe wywołania metod generują dodatkowe koszty. Bardzo zaawansowane metody rekurencyjne mogą doprowadzić do przepełnienia stosu. Wynika to z faktu, iż każde kolejne rekurencyjne wywołanie metody powoduje alokację miejsca na zmienne lokalne. Jeśli dojdzie do przepełnienia stosu, system wykonawczy Javy zgłosi wyjątek. Na ogół jednak nie trzeba zaprzątać sobie głowy tą kwestią, gdyż w zasadzie dotyczy ona tylko błędnie napisanych metod rekurencyjnych.

Główna zaleta metod rekurencyjnych polega na tym, iż dzięki nim można znacznie łatwiej zapisać niektóre algorytmy, których wersje iteracyjne są bardzo złożone. Na przykład algorytm QuickSort raczej trudno zaimplementować w sposób iteracyjny. Pewne problemy, szczególnie te związane ze sztuczną inteligencją, na ogół prowadzą do rozwiązań rekurencyjnych. Co więcej, niektórym osobom łatwiej zrozumieć algorytm rekurencyjny niż iteracyjny.

Gdy pisze się metody rekurencyjne, musi w nich wystąpić instrukcja warunkowa `if`, która spowoduje, iż po spełnieniu pewnych warunków nie dojdzie do wykonania kroku rekurencyjnego. Jeśli się nie doda takiego warunku, nigdy nie dojdzie do powrotu. Jest to typowy błąd występujący przy pisaniu algorytmów rekurencyjnych. Warto w trakcie testowania dodać do metody rekurencyjnej instrukcję `println()`, aby móc śledzić, co dzieje się w metodzie.

Oto kolejny przykład zastosowania rekurencji. Metoda `printArray()` wyświetla `i` pierwszych elementów tablicy `values`.

```
// R07\Recursion2.java
// Kolejny przykład obrazujący rekurencję

class RecTest {
  int[] values;

  RecTest(int i) {
    values = new int[i];
  }

  // Rekurencyjne wyświetlanie tablicy
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}
class Recursion2 {
  public static void main(String[] args) {
    RecTest ob = new RecTest(10);
    int i;
```

```
    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
  }
}
```

Wynik działania programu jest następujący.

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Wprowadzenie do kontroli dostępu

Hermetyzacja łączy dane z kodem, który je modyfikuje. Hermetyzacja obejmuje też jeszcze jeden ważny element — **kontrolę dostępu**. Dzięki hermetyzacji można określić, które części programu powinny mieć dostęp do składowych klasy. W ten sposób można łatwo zapobiec nadużyciom. Udostępniając dane tylko za pomocą dobrze zdefiniowanego interfejsu (metod), zapobiega się niewłaściwemu użyciu lub zmodyfikowaniu danych. Dobrze napisana klasa jest jak czarna skrzynka, która może być używana, ale jej wnętrze jest chronione przed niepowołanym dostępem. Prezentowane do tej pory klasy nie do końca korzystały z tych rozwiązań. Na przykład klasa `Stack` z końca poprzedniego rozdziału udostępnia metody `pop()` i `push()` stanowiące interfejs do sterowania stosem. Stosowanie tego interfejsu nie jest jednak w żaden sposób wymuszane. Kod innej klasy może obejść te metody i bezpośrednio zmienić zawartość tablicy. Niepowołane osoby mogą w ten sposób popsuć działanie stosu. W tym podrozdziale omówię mechanizm zapewniający kontrolowany dostęp do poszczególnych składowych klasy.

Sposób dostępu do składowej klasy określa **modyfikator dostępu** znajdujący się na samym początku deklaracji elementu. Java oferuje bogaty zestaw modyfikatorów. Pewne aspekty sterowania dostępem są związane z dziedziczeniem oraz pakietami. (**Pakiet** to, ogólnie rzecz biorąc, grupa klas). Modyfikatory z tej grupy zostaną omówione w dalszych rozdziałach. Tutaj zajmę się tylko dostępem z perspektywy pojedynczej klasy. Programista, który raz opanuje podstawy kontroli dostępu, bez trudu zrozumie pozostałe zagadnienia.


Uwaga

Wpływ na dostępność poszczególnych elementów kodu mogą mieć także moduły — możliwość wprowadzona do języka Java w JDK 9. Moduły zostaną przedstawione w rozdziale 16.

Modyfikatory dostępu w języku Java to `public`, `private` i `protected`. Dodatkowo istnieje jeszcze domyślny poziom dostępu. Modyfikator `protected` dotyczy dziedziczenia. Pozostałe modyfikatory zostały omówione poniżej.

Zacznę od zdefiniowania dostępu publicznego i prywatnego. Gdy składowa klasy stosuje modyfikator `public`, oznacza to, że jest dostępna dla dowolnego innego kodu. Gdy stosuje modyfikator `private`, oznacza to, że jest dostępna tylko dla kodu swojej klasy. W ten sposób łatwo wyjaśnić, dlaczego metoda `main()` zawsze musi być poprzedzona modyfikatorem `public` — ponieważ jest wywoływana przez zewnętrzny kod, w tym przypadku system wykonawczy Javy. Jeśli nie zastosuje się żadnego modyfikatora, składowa klasy jest dostępna publicznie dla wszystkich innych klas swojego pakietu, ale nie jest dostępna dla kodu z innych pakietów. (Pakiety zostaną omówione w rozdziale 9.).

W tworzonych do tej pory klasach wszystkie składowe używały domyślnego poziomu dostępu. Niestety, bardzo rzadko korzysta się z takiego rozwiązania. Najczęściej bardzo mocno ogranicza się dostęp do danych klasy — innymi słowy, wymusza się, by dane były dostępne jedynie za pomocą metod. Także niektóre metody pomocnicze są deklarowane jako prywatne składowe klasy.

Modyfikator dostępu zawsze znajduje się na samym początku specyfikacji składowej, czyli musi rozpoczynać instrukcję deklaracji. Oto przykład.

```
public int i;
private double j;

private int myMethod(int a, char b) { // ...
```

Aby lepiej zrozumieć efekt działania modyfikatorów, rozważmy następujący program.

```
// R07\AccessTest.java

/* Program ilustruje różnicę między dostępem
   prywatnym i publicznym
*/
class Test {
  int a; // Dostęp domyślny
  public int b; // Dostęp publiczny
  private int c; // Dostęp prywatny

  // Metoda korzystająca ze zmiennej c
  void setc(int i) { // Ustawienie wartości zmiennej c
    c = i;
  }
  int getc() { // Pobranie wartości zmiennej c
    return c;
  }
}

class AccessTest {
  public static void main(String[] args) {
    Test ob = new Test();

    // Poniższe instrukcje są poprawne, zmienne a i b są dostępne publicznie
    ob.a = 10;
    ob.b = 20;

    // Wykonanie poniższej instrukcji spowodowałoby zgłoszenie błędu
    // ob.c = 100; // Błąd!

    // Dostęp do zmiennej c tylko przez metodę
    ob.setc(100); // OK

    System.out.println("a, b i c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

Wewnątrz klasy `Test` zmienna składowa `a` używa dostępu domyślnego, który w tym przykładzie jest równoważny dostępowi publicznemu. Zmiennej `b` zostaje jawnie przyporządkowany dostęp publiczny (`public`). Zmiennej `c` zostaje jawnie przyporządkowany dostęp prywatny (`private`). Oznacza to, że klasa `AccessTest` nie może bezpośrednio odwołać się do zmiennej składowej `c`. Z tego powodu poniższy wiersz jest opatrzony komentarzem.

```
// ob.c = 100; // Błąd!
```

Gdyby usunąć komentarz, w trakcie kompilacji zostałby zgłoszony błąd dostępu.

Aby przekonać się, jak stosować mechanizm kontroli dostępu w bardziej praktycznym przykładzie, przeanalizujmy poniższą, ulepszoną wersję klasy `Stack` z końca rozdziału 6.

```
// R07\Stack.java

// Klasa definiuje stos liczb całkowitych mogący przechowywać do 10 wartości
class Stack {
  /* Teraz zmienne stck i tos są prywatne. Oznacza to, że
     nie mogą zostać zmienione celowo lub przypadkowo
     w sposób zagrażający działaniu stosu
  */
  private int[] stck = new int[10];
```

```
  private int tos;

  // Inicjalizacja szczytu stosu
  Stack() {
    tos = -1;
  }

  // Umieszczenie elementu na szczycie stosu
  void push(int item) {
    if(tos==9)
      System.out.println("Stos jest pełny.");
    else
      stck[++tos] = item;
  }

  // Zdjęcie elementu ze szczytu stosu
  int pop() {
    if(tos < 0) {
      System.out.println("Stos nie zawiera żadnych elementów.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

Teraz zarówno składowa `stck` (zawierająca stos), jak i zmienna `tos` (zawierająca indeks najwyższego elementu) zostały zdefiniowane jako składowe prywatne (`private`). Oznacza to, że nie można ich zmienić ani odczytać w sposób bezpośredni — trzeba skorzystać z metod `pop()` i `push()`. Takie rozwiązanie chroni na przykład zmienną `tos` przed ustawieniem wartości spoza zakresu tablicy przez inny fragment programu.

Poniższy program wykorzystuje ulepszą wersję klasy `Stack`. Usuń komentarze z ostatnich wierszy, aby przekonać się, że zmienne składowe `tos` i `stck` naprawdę nie są dostępne poza klasą `Stack`.

```
// R07\TestStack.java

class TestStack {
  public static void main(String[] args) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // Umieszczenie liczb na stosach
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // Zdjęcie liczb ze stosów
    System.out.println("Stos w mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stos w mystack2:");

    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());

    // Poniższe instrukcje nie są poprawne
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

Choć na ogół to właśnie metody zapewniają dostęp do danych klasy, nie zawsze musi tak być. Jeżeli z pewnych powodów zmienna składowa powinna być publiczna, warto tak zrobić. Na przykład w większości prostych klas zdefiniowanych w tej książce zrezygnowano z kontroli dostępu, aby maksymalnie uprościć ich kod. W rzeczywistych zastosowaniach najlepiej jednak udostępniać dane tylko poprzez metody. Wrócimy do tego zagadnienia także w następnym rozdziale, ponieważ kontrola dostępu należy do najważniejszych aspektów dziedziczenia.

Składowe statyczne

Czasem zachodzi potrzeba zdefiniowania składowej klasy, która nie zależałaby od żadnego konkretnego obiektu tej klasy. W tradycyjnym podejściu składowa dostępna jest tylko w powiązaniu z konkretną instancją klasy (obiektem). W Javie można jednak utworzyć składową udostępnianą bez podawania konkretnego obiektu. Aby utworzyć taki element, trzeba poprzedzić jego deklarację słowem kluczowym `static`. Składową statyczną można odczytywać i modyfikować, zanim zostaną utworzone jakiekolwiek obiekty klasy, i nie jest do tego potrzebna referencja do obiektu. Statyczne mogą być zarówno zmienne, jak i metody. Najlepszym przykładem składowej statycznej jest metoda `main()` — musi ona zostać zadeklarowana ze słowem `static`, ponieważ jest wywoływana przed utworzeniem jakichkolwiek obiektów.

Zmienne składowe zadeklarowane jako statyczne są tak naprawdę zmiennymi globalnymi. Gdy Java tworzy obiekt takiej klasy, jej zmienne statyczne nie są kopiowane. Wszystkie obiekty klasy współdzielą tę samą zmienną statyczną.

Metody zadeklarowane jako `static` mają kilka ograniczeń.

- Mogą wywoływać tylko inne metody statyczne tej samej klasy.
- Mają dostęp tylko do zmiennych statycznych tej samej klasy.
- Nie mogą w żaden sposób korzystać ze słów kluczowych `this` i `super`. (Słowo kluczowe super jest związane z dziedziczeniem i zostanie omówione w kolejnym rozdziale).

Jeżeli do inicjalizacji zmiennych `static` potrzebne są dodatkowe obliczenia, deklaruje się blok `static`, który zostanie wykonany dokładnie jeden raz w momencie pierwszego załadowania klasy. Poniższy przykład przedstawia klasę zawierającą metodę statyczną, kilka zmiennych statycznych i blok inicjujący `static`.

```
// R07\UseStatic.java

// Przykład statycznych zmiennych, metod i bloków
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }


  static {
    System.out.println("Inicjalizacja w bloku statycznym");
    b = a * 4;
  }

  public static void main(String[] args) {
    meth(42);
  }
}
```

W momencie załadowania klasy `UseStatic` zostaną wykonane wszystkie wyrażenia oznaczone jako część statyczna tej klasy. W pierwszym kroku zmiennej `a` zostanie przypisana wartość 3, następnie zostanie wykonany blok `static` (wyświetlenie komunikatu), a na końcu zmienna `b` przyjmie wartość `a * 4`, czyli 12. Wykonanie metody `main()` spowoduje wywołanie metody `meth()` i przekazanie `42` jako argumentu. Trzy wywołania metody `println()` otrzymują na wejściu odpowiednio dwie zmienne statyczne (`a` i `b`) oraz parametr `x`.

Wynik działania programu jest następujący.

```
Inicjalizacja w bloku statycznym
x = 42
a = 3
b = 12
```

Metody i zmienne statyczne poza klasą, w której zostały zdefiniowane, są używane niezależnie od obiektów tej klasy. Trzeba wtedy użyć nazwy samej klasy wraz z operatorem kropki. Na przykład wywołanie metody statycznej spoza jej klasy ma następującą ogólną postać.

```
nazwa-klasy.metoda()
```

Element *nazwa-klasy* to nazwa klasy, w której znajduje się wywoływana metoda statyczna. Format jest więc bardzo podobny do stosowanego dla metod niestatycznych, choć wtedy zamiast nazwy klasy występuje nazwa obiektu. Zmienne statyczne odczytuje się w identyczny sposób, używając nazwy klasy i operatora kropki przed nazwą zmiennej. W ten sposób Java implementuje łatwą do sterowania wersję globalnych metod i zmiennych.

Oto kolejny przykład. Metoda `main()` znajdująca się w osobnej klasie korzysta z metody statycznej `callme()` i zmiennej statycznej `b`.

```
// RO7\StaticDemo.java
class StaticDemo {
  static int a = 42;
  static int b = 99;

  static void callme() {
    System.out.println("a = " + a);
  }
}

class StaticByName {
  public static void main(String[] args) {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
  }
}
```

Program generuje następujące wyniki.

```
a = 42
b = 99
```

Słowo kluczowe final

Zmienne można zadeklarować z użyciem słowa kluczowego `final`. Słowo kluczowe `final` wyklucza możliwość modyfikacji wartości tak zadeklarowanej zmiennej, czyli w praktyce służy do deklarowania stałych. Oznacza to również, że inicjalizacja zmiennej tego typu musi nastąpić na etapie deklaracji. Taka deklaracja może mieć jedną z dwóch form — możemy przypisać wartość zmiennej albo w momencie deklaracji, albo w ciele konstruktora. To pierwsze rozwiązanie jest prawdopodobnie najczęściej stosowane. Oto przykład.

```
// RO7\listing_17.java
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Inne części programu mogą stosować zmienne `FILE_NEW`, `FILE_OPEN` itd. bez obawy o to, iż zostaną one w pewnym momencie zmodyfikowane. W zasadzie zmienne te można traktować jako stałe. Zgodnie z powszechną konwencją programistyczną nazwy zmiennych tego typu zapisuje się wielkimi literami.

Słowo kluczowe `final` można stosować nie tylko dla pól, ale też dla parametrów metod i zmiennych lokalnych. Zadeklarowanie parametru metody z tym słowem zapobiega zmianom tego parametru w ciele metody. Zadeklarowanie zmiennej ze słowem `final` wyklucza możliwość przypisania wartości tej zmiennej więcej niż raz.

Słowo kluczowe `final` stosuje się również dla metod, ale wtedy jego znaczenie jest całkowicie inne niż dla zmiennych. Zastosowanie tego słowa kluczowego dla metod zostanie dokładniej omówione w następnym rozdziale przy wprowadzaniu dziedziczenia.

Powtórka z tablic

Tablice zostały w niniejszej książce wprowadzone bardzo wcześnie, jeszcze przed omówieniem klas. Skoro już znamy klasy, warto wspomnieć o jednej z cech tablic — tablice są implementowane jak obiekty. Z tego względu istnieje specjalny atrybut tablicy, z którego korzysta się niezmiernie często. Zmienna składowa `length` tablicy zawiera informację o liczbie elementów, które może przyjąć tablica. Wszystkie tablice mają tę zmienną i zawsze przechowuje ona rozmiar tablicy. Oto program wykorzystujący tę właściwość.

// R07\Length.java

```
// Program korzystający ze zmiennej length tablic
class Length {
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = {3, 5, 7, 1, 8, 99, 44, -10};
    int[] a3 = {4, 3, 2, 1};


    System.out.println("rozmiar a1 wynosi " + a1.length);
    System.out.println("rozmiar a2 wynosi " + a2.length);
    System.out.println("rozmiar a3 wynosi " + a3.length);
  }
}
```

Program generuje następujące wyniki.

```
rozmiar a1 wynosi 10
rozmiar a2 wynosi 8
rozmiar a3 wynosi 4
```

Jak łatwo zauważyć, program wyświetla rozmiary poszczególnych tablic. Pamiętaj, że wartość zawarta w `length` nie ma nic wspólnego z liczbą elementów rzeczywiście znajdujących się w tablicy. Zmienna informuje jedynie o maksymalnej liczbie elementów, które mogą zostać umieszczone w tablicy.

Istnieje bardzo dużo zastosowań wartości ze zmiennej `length`. Poniżej znajduje się ulepszona wersja klasy `Stack`. Poprzednia wersja klasy zawsze tworzyła stos 10-elementowy. Nowa wersja potrafi utworzyć stos o dowolnym rozmiarze. Wartość `stck.length` stosujemy w celu uniemożliwienia przepełnienia stosu.

// R07\TestStack2.java

```
// Ulepszona klasa Stack wykorzystująca rozmiar tablicy
class Stack {
  private int[] stck;
  private int tos;

  // Alokacja i inicjalizacja stosu
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Umieszczenie elementu na szczycie stosu
  void push(int item) {
    if(tos==stck.length-1) // Użycie zmiennej składowej length
      System.out.println("Stos jest pełny.");
    else
      stck[++tos] = item;
  }
```

```
  // Zdjęcie elementu ze szczytu stosu
  int pop() {
    if(tos < 0) {
      System.out.println("Stos nie zawiera żadnych elementów.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class TestStack2 {
  public static void main(String[] args) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);

    // Umieszczenie liczb na stosach
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=10; i<8; i++) mystack2.push(i);

    // Zdjęcie liczb ze stosów
    System.out.println("Stos w mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stos w mystack2:");
    for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
  }
}
```

Zauważ, że program tworzy dwa stosy: jeden pięcioelementowy, drugi ośmioelementowy. Ponieważ informacja o rozmiarze znajduje się w samej tablicy, łatwo wykonać stos o dowolnym rozmiarze.

Klasy zagnieżdżone i klasy wewnętrzne

Można zdefiniować klasę wewnątrz innej klasy. Wtedy klasę znajdującą się wewnątrz nazywamy **klasą zagnieżdżoną**. Zasięg zagnieżdżonej klasy jest ściśle związany z zasięgiem klasy zewnętrznej. Jeśli klasa B została zdefiniowana wewnątrz klasy A, klasa B nie może istnieć niezależnie od klasy A (poza tą klasą). Zagnieżdżona klasa ma dostęp do wszystkich składowych, także prywatnych, należących do klasy zewnętrznej. Klasa zewnętrzna nie ma dostępu do składowych klasy zagnieżdżonej. Klasa zagnieżdżona, którą zadeklarowano bezpośrednio w zasięgu klasy zewnętrznej, jest składową klasy zewnętrznej. Istnieje też możliwość deklarowania klas zagnieżdżonych w ramach bloków kodu.

Istnieją dwa rodzaje klas zagnieżdżonych: **statyczne** i **niestatyczne**. Statyczna klasa zagnieżdżona posiada na początku modyfikator `static`. Ponieważ jest klasą statyczną, dostęp do składowych klasy zewnętrznej musi się odbywać przez konkretną nazwę obiektu (nie można odwoływać się do zmiennych klasy zewnętrznej w sposób bezpośredni).

Drugim rodzajem klas zagnieżdżonych są klasy **wewnętrzne**. Klasa wewnętrzna to niestatyczna klasa zagnieżdżona. Ma ona pełny dostęp do wszystkich zmiennych i metod klasy zewnętrznej i może się do nich odnosić w sposób bezpośredni, podobnie jak metody klasy zewnętrznej.

Poniższy program ilustruje definiowanie i zastosowanie klas wewnętrznych. Klasa o nazwie `Outer` zawiera jedną zmienną składową `outer_x`, jedną metodę składową `test()` oraz definicję klasy wewnętrznej `Inner`.

```
// R07\InnerClassDemo.java

// Przykład klasy wewnętrznej
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
```

```
    inner.display();
  }

  // To jest klasa wewnętrzna
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Wynik działania programu jest następujący.

```
display: outer_x = 100
```

W programie klasa wewnętrzna o nazwie `Inner` została zdefiniowana w zasięgu klasy `Outer`. Z tego powodu dowolny kod klasy `Inner` ma bezpośredni dostęp do zmiennej `outer_x`. Metoda `display()` znajduje się wewnątrz klasy `Inner` i powoduje przekazanie wartości zmiennej `outer_x` na standardowe wyjście. Metoda `main()` klasy `InnerClassDemo` tworzy instancję klasy `Outer` i wywołuje metodę `test()` dla nowego obiektu. Metoda `test()` tworzy instancję klasy `Inner` i wywołuje metodę `display()`.

Należy podkreślić istotny fakt: klasa `Inner` znana jest jedynie w kontekście klasy `Outer`. Próba utworzenia instancji klasy `Inner` w innym miejscu spowoduje, że kompilator Javy zgłosi błąd. Ogólnie rzecz biorąc, instancje klasy wewnętrznej są zazwyczaj tworzone przez kod umieszczony w zasięgu, w którym klasa ta została zdefiniowana, co pokazano na poniższym przykładzie.

Klasa wewnętrzna ma dostęp do wszystkich elementów klasy zewnętrznej, ale sytuacja odwrotna nie jest prawdziwa. Składowe klasy wewnętrznej znane są tylko w jej zasięgu i nie są dostępne dla klasy zewnętrznej. Oto przykład.

```
// R07\InnerClassDemo2.java

// Tego programu nie uda się skompilować
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // Klasa wewnętrzna
  class Inner {
    int y = 10; // y jest lokalne dla Inner

    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
  void showy() {
    System.out.println(y); // Błąd, y nie jest znane!
  }
}

class InnerClassDemo2 {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

W powyższym kodzie zmienna y stanowi składową klasy `Inner` i jako taka nie jest widoczna poza tą klasą, zatem nie może zostać wyświetlona przez metodę `showy()`.

Choć skupiliśmy się przede wszystkim na zagnieżdżonych klasach deklarowanych wewnątrz zasięgu klasy zewnętrznej, tak naprawdę klasy wewnętrzne można definiować w dowolnym bloku kodu. Nic nie stoi na przeszkodzie, aby zdefiniować klasę wewnątrz bloku metody lub nawet wewnątrz bloku pętli `for` (patrz poniższy przykład).

```
// R07\InnerClassDemo3.java

// Definicja klasy wewnętrznej wewnątrz bloku pętli for
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
      }
      Inner inner = new Inner();
      inner.display();
    }
  }
}

class InnerClassDemo3 {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Program generuje następujące wyniki.

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

Choć w tradycyjnym programowaniu klasy wewnętrzne stosuje się niezmiernie rzadko, są one bardzo pomocne przy obsłudze zdarzeń w apletach. Do tematu klas wewnętrznych powrócę w rozdziale 25., w którym pokażę, w jaki sposób tego rodzaju klasy upraszczają kod obsługi zdarzeń. Omówię tam również **anonimowe klasy wewnętrzne** — klasy wewnętrzne bez jawnie określonych nazw.

Ostatnia uwaga: klasy zagnieżdżone zostały dodane dopiero w Javie 1.1, więc nie można ich stosować w kodzie zgodnym z Javą 1.0.

Omówienie klasy String

Choć klasa `String` zostanie szczegółowo omówiona w części II książki, już tutaj potrzebne jest pewne wprowadzenie, gdyż będziemy dosyć intensywnie korzystać z tej klasy w przykładach pod koniec części I książki. Klasa `String` jest prawdopodobnie najczęściej stosowaną klasą języka Java. To o tyle naturalne, że łańcuchy stanowią bardzo ważny aspekt programowania.

Przede wszystkim trzeba zrozumieć, iż każdy tekst w kodzie programu zostanie zamieniony na obiekt typu `String`. Dotyczy to również stałych tekstowych. Oto przykład.

```
System.out.println("To także jest obiekt klasy String.");
```

Stała tekstowa `"To także jest obiekt klasy String."` jest obiektem klasy `String`.

Drugą bardzo istotną kwestią jest tak zwana niezmienność obiektów typu `String`. Gdy obiekt `String` zostanie utworzony, jego zawartość nie może zostać zmodyfikowana. Choć może się to wydawać dużym ograniczeniem, nie jest tak z dwóch powodów:

- Gdy modyfikuje się łańcuch znaków, zawsze można utworzyć nowy łańcuch zawierający wszystkie modyfikacje.
- Java definiuje klasy pomocnicze `StringBuffer` oraz `StringBuilder`, które umożliwiają modyfikację łańcuchów, więc wszystkie tradycyjne operacje na łańcuchach są dostępne również w Javie. (Klasy `StringBuffer` i `StringBuilder` są dokładnie omówione w części II książki).

Obiekty `String` tworzy się na wiele różnych sposobów. Najprościej użyć poniższego polecenia.

```
String myString = "to jest test";
```

Tak utworzony obiekt klasy `String` można wykorzystywać wszędzie tam, gdzie jest dopuszczalne stosowanie łańcuchów. Następujący wiersz spowoduje wyświetlenie zawartości zmiennej `myString`.

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

Java definiuje dla łańcuchów tylko jeden operator +, który służy do łączenia (konkatenacji) łańcuchów. Oto przykład.

```
String myString = "Lubię " + "Javę.";
```

Powyższa instrukcja spowoduje przypisanie łańcucha `"Lubię Javę."` zmiennej `myString`.

Następujący program obrazuje wykorzystanie konkatenacji łańcuchów znaków.

```
// R07\StringDemo.java

// Łączenie łańcuchów znaków
class StringDemo {
  public static void main(String[] args) {
    String strOb1 = "Pierwszy tekst";
    String strOb2 = "Drugi tekst";
    String strOb3 = strOb1 + " i " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

Program po uruchomieniu generuje następujące komunikaty.

```
Pierwszy tekst
Drugi tekst
Pierwszy tekst i Drugi tekst
```

Klasa `String` zawiera wiele przydatnych metod. Tutaj przedstawię tylko kilka z nich. Do porównywania dwóch łańcuchów służy metoda `equals()`. Metoda `length()` zwraca długość łańcucha. Aby pobrać pojedynczy znak łańcucha z konkretnej pozycji, należy użyć metody `charAt()`. Ogólne postacie wspomnianych metod są następujące.

```
boolean equals(String obiekt)
int length()
char charAt(int indeks)
```

Poniżej znajduje się program wykorzystujący omawiane metody.

```
// R07\StringDemo2.java

// Przykład użycia kilku metod klasy String
class StringDemo2 {
  public static void main(String[] args) {
    String strOb1 = "Pierwszy tekst";
    String strOb2 = "Drugi tekst";
    String strOb3 = strOb1;
```

```
    System.out.println("Długość strOb1: " +
                       strOb1.length());

    System.out.println("Znak o indeksie 3 z strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

Program generuje następujące wyniki.

```
Długość strOb1: 14
Znak o indeksie 3 z strOb1: r
strOb1 != strOb2
strOb1 == strOb3
```

Programista może oczywiście utworzyć tablicę łańcuchów, tak samo jak tworzy tablice elementów dowolnych innych typów. Oto przykład.

```
// RO7\StringDemo3.java

// Tablica obiektów typu String
class StringDemo3 {
  public static void main(String[] args) {
    String[] str = { "jeden", "dwa", "trzy" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
}
```

Wynik działania programu jest następujący.

```
str[0]: jeden
str[1]: dwa
str[2]: trzy
```

W dalszej części rozdziału przekonamy się, że tablice łańcuchów odgrywają ważną rolę w wielu programach Javy.

Wykorzystanie argumentów wiersza poleceń

W pewnych sytuacjach zachodzi potrzeba przekazania do programu dodatkowych informacji w momencie jego uruchamiania. W Javie zadanie to polega na przekazaniu **argumentów wiersza poleceń** do metody `main()`. Argument wiersza poleceń to informacja występująca w wierszu poleceń tuż po nazwie uruchamianego programu. Odczytanie argumentów w języku Java jest bardzo proste — wystarczy sprawdzić tablicę obiektów `String` przekazywaną do metody `main()`. Poniższy program wyświetla wszystkie argumenty przekazane do niego w wierszu poleceń.

```
// RO7\CommandLine.java

// Wyświetla wszystkie argumenty wiersza poleceń
class CommandLine {
  public static void main(String[] args) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```

Uruchom program, używając następującego polecenia.

```
java CommandLine to jest test 100 -1
```

Wynik działania programu po wykonaniu tego zadania będzie następujący.

```
args[0]: to
args[1]: jest
args[2]: test
args[3]: 100
args[4]: -1
```

Wszystkie argumenty wiersza poleceń są przekazywane w formie łańcuchów. Wartości liczbowe należy więc ręcznie konwertować już w kodzie programu (patrz rozdział 19.).

Zmienna liczba argumentów

Nowoczesne wersje Javy oferują rozwiązanie ułatwiające tworzenie metod, które muszą przyjmować zmienną liczbę argumentów. **Zmienną liczbę argumentów** często nazywa się **varargs**. Metodę, która wykorzystuje zmienną liczbę argumentów, nazywa się **metodą zmiennoargumentową** lub **metodą varargs**.

Sytuacje, w których potrzeba zmiennej liczby argumentów, nie są rzadkością. Na przykład metoda otwierająca połączenie internetowe może albo otrzymywać nazwę użytkownika, hasło, nazwę pliku, protokół itp., albo stosować wartości domyślne dla argumentów, które nie zostały przekazane. W tej sytuacji najlepiej byłoby przekazywać do metody tylko te argumenty, których wartości domyślne nie pasują do bieżącego scenariusza. Innym przykładem jest metoda `printf()` stanowiąca część biblioteki wejścia-wyjścia. W rozdziale 21. pokażę, iż przyjmuje ona zmienną liczbę argumentów (format danych oraz wartości).

W początkowym okresie stosowania języka Java zmienną liczbę argumentów można było obsługiwać na dwa sposoby, z których żaden nie gwarantował należytej wygody. Jeśli maksymalna liczba argumentów była niewielka i dobrze znana, można było utworzyć przeciążone wersje metody z różną liczbą przyjmowanych argumentów. Niestety takie podejście, choć w miarę wygodne, mogło być stosowane tylko w niektórych sytuacjach.

Gdy maksymalna liczba argumentów była duża lub po prostu nieznana, stosowało się drugie rozwiązanie, wciąż spotykane w niektórych starych kodach, polegające na umieszczaniu argumentów w tablicy i przekazywaniu tej tablicy do metody. Takie podejście ilustruje poniższy program.

```
// R07\PassArray.java

// Użycie tablicy do przekazywania zmiennej liczby argumentów
// do metody. Jest to rozwiązanie stosowane przed wprowadzeniem
// metod o zmiennej liczbie argumentów
class PassArray {
  static void vaTest(int[] v) {
    System.out.print("Liczba argumentów: " + v.length +
                     " Zawartość: ");

    for(int x : v)
      System.out.print(x + " ");
    System.out.println();
  }

  public static void main(String[] args)
  {
    // Zauważ sposób tworzenia tablicy wymagany do
    // przechowywania argumentów
    int[] n1 = { 10 };
    int[] n2 = { 1, 2, 3 };
    int[] n3 = { };

    vaTest(n1); // 1 argument
    vaTest(n2); // 3 argumenty
```

```
    vaTest(n3); // Brak argumentów
  }
}
```

Wynik działania programu jest następujący.

```
Liczba argumentów: 1 Zawartość: 10
Liczba argumentów: 3 Zawartość: 1 2 3
Liczba argumentów: 0 Zawartość:
```

W powyższym programie metoda `vaTest()` pobiera przekazane do niej argumenty za pośrednictwem tablicy `v`. Takie rozwiązanie stosowane w starszych programach Javy umożliwiało pobranie dowolnej liczby argumentów. Niestety wymagało ręcznego pakowania poszczególnych argumentów do tablicy przed wywołaniem metody `vaTest()`. Tworzenie tablicy przy każdym wywołaniu metody jest nie tylko żmudne, ale również łatwo popełnić błąd. Nowa cecha języka Java 5 dostarcza prostsze, lepsze rozwiązanie.

Zmienną liczbę argumentów określa się przez zastosowanie trzech kropek (`...`). Poniżej znajduje się początek definicji metody `vaTest()` w wersji `varargs`.

```
static void vaTest(int ... v) {
```

Powyższa składnia informuje kompilator, że metoda `vaTest()` może zostać wywołana z zerem lub większą liczbą argumentów. Parametr `v` jest niejawnie deklarowany jako tablica typu `int[]`. Z tego powodu wewnątrz metody `v` traktuje się go jak normalną tablicę. Poniżej znajduje się ten sam program co wcześniej, ale napisany przy użyciu nowego podejścia.

```
// R07\VarArgs.java

// Metoda o zmiennej liczbie argumentów
class VarArgs {

  // Metoda vaTest() jest typu varargs
  static void vaTest(int ... v) {
    System.out.print("Liczba argumentów: " + v.length +
                       " Zawartość: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }


  public static void main(String[] args)
  {

    // Zauważ, że teraz metoda vaTest() jest
    // po prostu wywoływana ze zmienną liczbą argumentów
    vaTest(10);       // 1 argument
    vaTest(1, 2, 3); // 3 argumenty
    vaTest();         // Brak argumentów
  }
}
```

Wynik działania powyższego programu jest dokładnie taki sam jak poprzedniego programu.

W programie warto zauważyć dwie istotne kwestie. Po pierwsze, we wnętrzu metody `vaTest()` zmienna `v` niczym nie różni się od tradycyjnej tablicy. To dość naturalne — w końcu zmienna `v` *jest* tablicą. Składnia `...` po prostu informuje kompilator, aby wszystkie argumenty przekazane do metody umieścił w kolejnych elementach tablicy `v`. Po drugie, wewnątrz metody `main()` metoda `vaTest()` jest wywoływana z różną liczbą argumentów, a nawet bez żadnych argumentów. Programista nie musi zajmować się umieszczaniem argumentów w tablicy. Brak argumentów oznacza tablice o zerowej długości.

Oprócz parametrów ze zmienną liczbą argumentów metoda może otrzymywać także tradycyjne parametry. Wymaga to jednak umieszczenia parametru o zmiennej liczbie argumentów na samym końcu parametrów metody. Następująca deklaracja metody jest w pełni poprawna.

```
int doIt(int a, int b, double c, int ... vals) {
```

Pierwsze trzy argumenty przekazane do metody `doIt()` znajdą się w trzech pierwszych parametrach. Wszystkie pozostałe argumenty zostają umieszczone w tablicy `vals`.

Ponieważ parametr o zmiennej liczbie argumentów musi pojawić się jako ostatni, poniższa deklaracja nie jest poprawna.

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Błąd!
```

Próba zadeklarowania tradycyjnego parametru po zmiennej liczbie argumentów zawsze spowoduje zgłoszenie błędu przez kompilator.

Co więcej, jedna metoda może posiadać tylko jeden parametr o zmiennej liczbie argumentów. Z tego powodu następująca deklaracja nie jest poprawna.

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Błąd!
```

Próba zadeklarowania drugiego parametru o zmiennej liczbie argumentów zawsze spowoduje zgłoszenie błędu przez kompilator.

Poniżej znajduje się zmodyfikowana wersja metody `vaTest()`, która przyjmuje zarówno tradycyjny parametr, jak i zmienną liczbę argumentów.

```
// R07\VarArgs2.java

// Metoda typu varargs z tradycyjnymi parametrami
class VarArgs2 {

  // Tutaj msg to tradycyjny parametr, a v to
  // parametr typu varargs
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                       " Zawartość: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String[] args)
  {
    vaTest("Jeden parametr typu vararg: ", 10);
    vaTest("Trzy parametry typu vararg: ", 1, 2, 3);
    vaTest("Brak parametrów typu vararg: ");
  }
}
```

Wynik działania programu jest następujący.

```
Jeden parametr typu vararg: 1 Zawartość: 10
Trzy parametry typu vararg: 3 Zawartość: 1 2 3
Brak parametrów typu vararg: 0 Zawartość:
```

Przeciążanie metod o zmiennej liczbie argumentów

Metody otrzymujące zmienną liczbę argumentów można przeciążać. Poniższa klasa posiada trzy przeciążone wersje metody `vaTest()`.

```
// R07\VarArgs3.java

// Metody typu vararg i przeciążanie
class VarArgs3 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Liczba argumentów: " + v.length +
                     " Zawartość: ");

    for(int x : v)
      System.out.print(x + " ");
```

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

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Liczba argumentów: " + v.length +
                     " Zawartość: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
                     msg + v.length +
                     " Zawartość: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String[] args)
  {
    vaTest(1, 2, 3);
    vaTest("Testowanie: ", 10, 20);
    vaTest(true, false, false);
  }
}
```

Wynik działania programu jest następujący.

```
vaTest(int ...): Liczba argumentów: 3 Zawartość: 1 2 3
vaTest(String, int ...): Testowanie: 2 Zawartość: 10 20
vaTest(boolean ...) Liczba argumentów: 3 Zawartość: true false false
```

Program ilustruje oba dopuszczalne sposoby przeciążania metod o zmiennej liczbie argumentów. Pierwszy sposób to zróżnicowanie typów zmiennego parametru. Dotyczy to przypadków `vaTest(int ...)` oraz `vaTest(boolean ...)`. Pamiętaj, że składnia `...` powoduje traktowanie parametru jak tablicy określonego typu. Z tego powodu podobnie jak można tworzyć przeciążone metody o różnych typach tablic, można również tworzyć parametry typu `vararg` różnych typów. Java używa tego typu do rozróżnienia, którą wersję metody należy wywołać.

Drugi sposób przeciążania polega na dodaniu tradycyjnego parametru (przypadek `vaTest(String, int...)`). W takiej sytuacji do rozróżnienia wersji metody Java używa zarówno tradycyjnego parametru, jak i parametru o zmiennej liczbie argumentów.

Metoda `varargs` może być przeciążona także przez metodę pozbawioną parametrów wieloargumentowych. Na przykład metoda `vaTest(int x)` jest prawidłowym przeciążeniem metody `vaTest()` (patrz program w kolejnym punkcie). Wspomniana wersja zostanie wywołana tylko w przypadku przekazania pojedynczego argumentu typu `int`. Przekazanie co najmniej dwóch argumentów typu `int` spowoduje wykonanie wersji `varargs`, czyli `vaTest (int...v)`.

Zmienna liczba argumentów i niejednoznaczności

Gdy korzysta się z przeciążania metod o zmiennej liczbie argumentów, czasem mogą pojawić się nieoczekiwane wyniki. Wynikają one z niejednoznaczności wywołań tak przeciążonej metody. Rozważmy następujący program.

```
// R07\VarArgs4.java

// Zmienna liczba argumentów, przeciążanie i niejednoznaczność
//
// Program zawiera błąd, więc nie zostanie skompilowany!
class VarArgs4 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(Integer ...): " +
                     "Liczba argumentów: " + v.length +
                     " Zawartość: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Liczba argumentów: " + v.length +
                     " Zawartość: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }


  public static void main(String[] args)
  {
    vaTest(1, 2, 3);  // OK
    vaTest(true, false, false); // OK

    vaTest(); // Błąd: niejednoznaczność!
  }
}
```

W programie przeciążone wersje metody `vaTest()` zostały napisane poprawnie. Niestety, programu nie uda się skompilować z powodu następującego wywołania.

```
vaTest(); // Błąd: niejednoznaczność!
```

Ponieważ parametr wieloargumentowy nie wymaga żadnych argumentów, powyższe wywołanie może dotyczyć zarówno metody `vaTest(int ...)`, jak i metody `vaTest(boolean ...)`. Obie wersje są równie poprawne, więc powstaje niejednoznaczność.

Oto inny przykład niejednoznaczności. Poniższe wersje metody `vaTest()` także mogą prowadzić do niejednoznaczności, choć jedna z wersji stosuje tradycyjny parametr.

```
static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...
```

Choć lista parametrów obu wersji jest różna, kompilator nie potrafi wybrać odpowiedniej wersji metody dla poniższego wywołania.

```
vaText(1);
```

Przedstawione wywołanie pasuje zarówno do `vaTest(int ...)`, jak i do `vaTest(int, int ...)`. Kompilator nie potrafi rozstrzygnąć, która wersja jest poprawna — istnieje niejednoznaczność.

Ze względu na tego rodzaju problemy warto czasem poświęcić przeciążanie i po prostu zastosować dwie różne nazwy metod. Co więcej, czasami niejednoznaczności odsłaniają błędy koncepcyjne w kodzie programu, które warto poprawić.

Stosowanie wnioskowania typów zmiennych lokalnych z typami referencyjnymi

Jak już wiesz z rozdziału 3., od JDK 10 język Java udostępnia mechanizm wnioskowania typów zmiennych lokalnych. Zapewne pamiętasz, że podczas korzystania z wnioskowania typów zmiennych lokalnych typ zmiennej jest określany jako `var`, a sama zmienna musi zostać zainicjalizowana. We wcześniejszych przykładach przedstawiłem stosowanie tego mechanizmu z typami prostymi i referencyjnymi. Właściwie mechanizm ten jest stosowany przede wszystkim w zmiennych typów referencyjnych. Poniższy przykład przedstawia deklarację zmiennej `myStr` typu `String`:

```
var myStr = "To jest łańcuch";
```

Ponieważ inicjalizatorem jest łańcuch zapisany pomiędzy znakami cudzysłowu, wywnioskowanym typem będzie `String`.

Jak wyjaśniłem w rozdziale 3., jedną z korzyści stosowania mechanizmu wnioskowania typów zmiennych lokalnych jest skracanie i upraszczanie kodu, przy czym najłatwiej to zauważyć właśnie w przypadku stosowania typów referencyjnych.

Wynika to z faktu, że wiele klas w Javie ma długie nazwy. Na przykład w rozdziale 13. poznasz klasę `FileInputStream`, służącą do otwierania pliku w celu wczytywania danych. W przeszłości, gdy konieczne było stosowanie tradycyjnych deklaracji, zmienną tego typu trzeba było deklarować i inicjalizować w następujący sposób:

```
FileInputStream fin = new FileInputStream("test.txt");
```

Obecnie, w przypadku stosowania `var`, deklarację tę możemy zapisać tak:

```
var fin = new FileInputStream("test.txt");
```

Na podstawie takiej deklaracji Java wywnioskuje, że zmienna `fin` jest typu `FileInputStream`, gdyż taki jest typ użytego inicjalizatora. Nie trzeba jawnie powtarzać nazwy typu. W efekcie ta druga deklaracja zmiennej `fin` jest znacząco krótsza niż w przypadku stosowania tradycyjnego zapisu. Jak zatem widać, użycie `var` upraszcza deklaracje. Ta zaleta mechanizmu wnioskowania typów zmiennych lokalnych staje się jeszcze wyraźniejsza w przypadku bardziej złożonych deklaracji, takich jak deklaracje korzystające z typów sparametryzowanych. Ogólnie rzecz ujmując, ta cecha mechanizmu wnioskowania typów zmiennych lokalnych, która umożliwia upraszczanie kodu, eliminuje w pewnym stopniu konieczność wpisywania w kodzie nazw typów.

Oczywiście mechanizm wnioskowania typów zmiennych lokalnych w celu upraszczania kodu musi być stosowany rozważnie, by nie doprowadził do pogorszenia czytelności kodu i ukrycia jego znaczenia. W ramach przykładu przyjrzyjmy się następującej deklaracji:

```
var x = o.getNext();
```

W tym przypadku osoba analizująca kod może nie być w stanie bezproblemowo określić, jakiego typu będzie zmienna `x`. A zatem, mówiąc ogólnie, wnioskowanie typów zmiennych lokalnych jest mechanizmem, którego należy używać mądrze.

Jak łatwo się domyślić, mechanizmu wnioskowania typów zmiennych lokalnych można także używać w przypadku korzystania z własnych klas, co pokazałem na kolejnym przykładzie. Poniższy program definiuje klasę `MyClass`, a następnie korzysta z mechanizmu wnioskowania typów zmiennych lokalnych w celu zadeklarowania i zainicjalizowania obiektu tej klasy:

```
// R07\RefVarDemo.java

// Użycie wnioskowania typów zmiennych lokalnych z klasą zdefiniowaną przez programistę
class MyClass {
  private int i;

  MyClass(int k) { i = k;}

  int geti() { return i; }
  void seti(int k) { if(k >= 0) i = k; }
}
```

```
class RefVarDemo {
  public static void main(String[] args) {
    var mc = new MyClass(10); // Zwróć tu uwagę na użycie var

    System.out.println("Wartość i w mc wynosi " + mc.geti());
    mc.seti(19);
    System.out.println("A teraz wartość i w mc wynosi " + mc.geti());
  }
}
```

Wykonanie tego programu zwróci następujące wyniki:

```
Wartość i w mc wynosi 10
A teraz wartość i w mc wynosi 19
```

W tym przykładzie zwróć szczególną uwagę na następujący wiersz:

```
var mc = new MyClass(10); // Zwróć tu uwagę na użycie var
```

Wywnioskowanym typem zmiennej `mc` będzie tu `MyClass`, gdyż inicjalizatorem jest nowy obiekt tej klasy.

Jak już zaznaczyłem wcześniej, z myślą o czytelnikach korzystających ze starszych wersji środowiska Javy, które nie obsługują mechanizmu wnioskowania typów zmiennych lokalnych, nie będę z niego korzystał w przykładach zamieszczonych w dalszej części książki. Dzięki temu większość czytelników będzie mogła bez problemu kompilować i wykonywać te przykłady.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ź

Ż

Java: wszystko, co musisz wiedzieć, w jednym kompendium!

Java od wielu lat utrzymuje swoją wyjątkową pozycję. To język programowania, który łączy w sobie technologiczną dojrzałość i nowoczesność. Jego twórcy od pierwszego wydania kładą nacisk na programowanie rozwiązań internetowych, a także na elastyczność i szybkość. W efekcie obecnie Java często jest pierwszym i najlepszym wyborem programistów, którzy chcą tworzyć szerokie spektrum oprogramowania. Wszędzie tam, gdzie wymaga się niezawodności, można znaleźć kod Javy.

Oto dwunaste wydanie wyczerpującego kompendium, w pełni zaktualizowane, uzupełnione o nowości wprowadzone w Java SE 17. Opisano w nim cały język Java: jego składnię, słowa kluczowe i najistotniejsze zasady programowania. Znalazły się tu także informacje dotyczące biblioteki Java API, takie jak operacje wejścia-wyjścia, biblioteka strumieni i techniki programowania współbieżnego. Nie zabrakło opisu biblioteki Swing, JavaBeans i serwletów, jak również licznych przykładów praktycznego zastosowania Javy. Wyczerpująco omówiono najnowsze możliwości języka, takie jak rekordy, klasy zapieczętowane czy też wyrażenia switch. Podręcznik został napisany w sposób przejrzysty, jasnym i zrozumiałym językiem, co znakomicie ułatwia naukę, a poszczególne zagadnienia zilustrowano licznymi przykładowymi fragmentami kodu źródłowego. To sprawia, że z kompendium skorzystają wszyscy programiści Javy, zarówno początkujący, jak i profesjonalni deweloperzy.

W książce między innymi:

- zasady programowania obiektowego
- obsługa zdarzeń, moduły i wyrażenia lambda
- klasy i interfejsy zapieczętowane
- rozszerzenia instrukcji switch, rekordy, bloki tekstu
- Collections, AWT, Swing, JavaBean i serwlety

Herbert Schildt jest niekwestionowanym autorytetem w dziedzinie programowania w Javie. Od ponad trzydziestu lat pisze książki poświęcone nauce Javy, C, C++ i C#; wiele z nich sprzedano w milionach egzemplarzy i przetłumaczono na różne języki. Pasjonuje się wszystkim, co jest związane z komputerowym przetwarzaniem, ale najbardziej językami programowania.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-156-4
9 788383 221564
Cena: 199,00 zł
Mc Graw Hill