JAVA

Podstawy

WYDANIE XII

ORACLE

Cay S. Horstmann

Tytuł oryginału: Core Java, Volume I: Fundamentals, 12th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-9479-7

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/javp12
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem: *https://ftp.helion.pl/przyklady/javp12.zip*

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

5

Dziedziczenie

W tym rozdziale:

Rozdział 4. wprowadził pojęcia klas i obiektów. Ten rozdział wprowadza kolejne zagadnienie o fundamentalnym znaczeniu dla programowania obiektowego — **dziedziczenie** (ang. *inheritance*). Z założenia technika ta umożliwia tworzenie nowych klas na bazie klas już istniejących. Klasa, która dziedziczy po innej klasie, przejmuje jej metody i pola oraz dodaje własne metody i pola, które służą przystosowaniu do nowych zadań. Technika ta ma kluczowe znaczenie dla programowania w Javie.

Dodatkowo rozdział ten opisuje **refleksję** (ang. *reflection*), czyli technikę inspekcji klas w trakcie działania programu. Mimo że refleksja daje ogromne możliwości, jest bez wątpienia techniką skomplikowaną. Ponieważ ma ona większe znaczenie dla twórców narzędzi niż programistów aplikacji, tę część rozdziału można przeczytać pobieżnie i wrócić do niej w razie potrzeby.

5.1. Klasy, nadklasy i podklasy

Wróćmy do omawianej w poprzednim rozdziale klasy `Employee`. Wyobraźmy sobie, że (niestety) pracujemy w firmie, w której kierownicy są traktowani inaczej niż pozostali pracownicy. Oczywiście istnieje między nimi też wiele podobieństw. Zarówno zwykli pracownicy, jak i kierownictwo dostają wypłatę. Jednak podczas gdy zwykli pracownicy, aby otrzymać pensję, muszą ukończyć powierzone im zadania, kierownicy, jeśli osiągną zamierzony cel, dostają *dodatek* do wypłaty. Jest to typowa sytuacja, w której należy wykorzystać dziedziczenie. Dlaczego? Oczywiście można utworzyć całkiem nową klasę o nazwie `Manager` o odpowiednich właściwościach. Można jednak wykorzystać część kodu, który został napisany wcześniej w klasie `Employee`. *Wszystkie* pola oryginalnej klasy zostałyby zachowane. Stosując bardziej abstrakcyjną terminologię, istnieje oczywisty związek „jest" pomiędzy klasami `Manager` i `Employee`. Każdy kierownik (ang. *manager*) *jest* pracownikiem (ang. *employee*). Relacja typu „jest" stanowi cechę charakterystyczną dziedziczenia.

W rozdziale tym omawiamy klasyczny przykład z pracownikami i dyrektorami, ale pamiętaj, że należy do niego podchodzić z przymrużeniem oka. W prawdziwym świecie pracownik może zostać dyrektorem, więc dyrektor powinien być rolą pracownika, a nie podklasą. W naszym przykładzie jednak zakładamy, że świat korporacji zamieszkują tylko dwa rodzaje ludzi — ci, którzy po wsze czasy będą szeregowymi pracownikami, i ci, którzy od zawsze byli dyrektorami.

5.1.1. Definiowanie podklas

Do wyrażania relacji dziedziczenia służy słowo kluczowe `extends`. W poniższym przykładowym kodzie klasa `Manager` dziedziczy po klasie `Employee`.

```
class Manager extends Employee
{
   Dodatkowe metody i pola.
}
```

Dziedziczenie w Javie i C++ jest podobne, jednak w Javie wyraża się je za pomocą słowa kluczowego `extends`, a w C++ za pomocą symbolu `:`. W Javie dziedziczenie może być wyłącznie publiczne. Nie ma w tym języku odpowiedników znanych z C++ dziedziczenia prywatnego i chronionego.

Słowo kluczowe `extends` oznacza, że tworzona jest nowa klasa na podstawie istniejącej klasy. Klasa istniejąca nazywana jest **nadklasą** (ang. *superclass*), **klasą bazową** (ang. *base class*) lub **klasą macierzystą** (ang. *parent class*). Nowo utworzona klasa nazywa się **podklasą** (ang. *subclass*), **klasą pochodną** (ang. *derived class*) lub **klasą potomną** (ang. *child class*). Większość programistów Javy używa terminów „nadklasa" i „podklasa", ale niektórym bardziej odpowiada analogia klasy macierzystej i potomnej, która bardzo dobrze pasuje do koncepcji dziedziczenia.

Klasa `Employee` jest nadklasą, ale nie ze względu na to, że jest wyższa rangą albo ma większą funkcjonalność. *W rzeczywistości jest odwrotnie*: podklasa ma *większą* funkcjonalność niż nadklasa. Będzie można się o tym przekonać, kiedy przejdziemy do analizy klasy `Manager`, która zawiera więcej danych i ma większą funkcjonalność niż jej nadklasa `Employee`.

Przedrostki *nad-* i *pod-* pochodzą od sposobu opisu zbiorów w informatyce teoretycznej i matematyce. Zbiór wszystkich pracowników zawiera zbiór wszystkich kierowników, czyli zbiór pracowników jest **nadzbiorem** zbioru kierowników. Albo w drugą stronę — zbiór kierowników jest **podzbiorem** zbioru pracowników.

Klasa `Manager` zawiera dodatkowe pole do przechowywania dodatku do pensji i nową metodę do ustawiania jego wysokości:

```
public class Manager extends Employee
{
   private double bonus;

   . . .
   public void setBonus(double bonus)
   {
      this.bonus = bonus;
   }
}
```

Powyższego pola i powyższej metody nie wyróżnia nic szczególnego. Po utworzeniu obiektu klasy `Manager` można na jego rzecz wywołać metodę `setBonus`:

```
Manager boss = . . .;
boss.setBonus(5000);
```

Oczywiście na rzecz obiektu klasy `Employee` nie można wywołać metody `setBonus`. Nie ma jej wśród metod tej klasy.

Natomiast na rzecz obiektów klasy `Manager` *można* wywoływać metody `getName` i `getHireDay`, mimo że nie zostały one zdefiniowane bezpośrednio w tej klasie. Są dostępne, ponieważ zostały odziedziczone po klasie `Employee`.

Każdy obiekt klasy `Manager` ma cztery pola: `name`, `salary`, `hireDay` i `bonus`. Pola `name`, `salary` i `hireDay` pochodzą z nadklasy.

W specyfikacji języka Java napisano: „Składowe klasy, które są zadeklarowane jako prywatne, nie są dziedziczone przez podklasy tej klasy". To zdanie od lat wprowadza w konsternację wielu moich czytelników. W specyfikacji słowo „dziedziczenie" jest używane w bardzo wąskim znaczeniu. Prywatne pola zostały opisane jako niedziedziczone, ponieważ klasa `Manager` nie ma do nich bezpośredniego dostępu. W związku z tym każdy obiekt klasy `Manager` ma trzy pola z nadklasy, ale klasa `Manager` ich nie „dziedziczy".

W definicji klasy rozszerzającej inną klasę konieczne jest podanie tylko *różnic* pomiędzy tymi klasami. Metody ogólnego przeznaczenia należy umieszczać w nadklasie, a metody bardziej

wyspecjalizowane — w podklasie. Wydzielanie wspólnego kodu i umieszczanie go w nadklasie jest często stosowaną techniką programowania obiektowego.

W rozdziale 4. zostały opisane **rekordy**, czyli klasy, których stan w całości określają parametry konstruktora. Rekordu nie można rozszerzyć i rekord nie może rozszerzać żadnej innej klasy.

5.1.2. Przesłanianie metod

Niektóre obecne w klasie nadrzędnej metody są niewłaściwe dla klasy `Manager`. Jest tak w przypadku metody `getSalary`, która powinna zwracać sumę podstawy wynagrodzenia i dodatku. Konieczne jest napisanie nowej metody **przesłaniającej** (ang. *override*) metodę z nadklasy:

```
public class Manager extends Employee
{
   . . .
   public double getSalary()
   {
      . . .
   }
   . . .
}
```

Jak powinna wyglądać implementacja tej metody? Na pierwszy rzut oka wydaje się to proste — wystarczy zwrócić sumę pól `salary` i `bonus`:

```
public double getSalary()
{
   return salary + bonus;    // nie działa
}
```

Tak się jednak nie da. Tylko metody klasy `Employee` mają bezpośredni dostęp do prywatnych pól klasy `Employee`. Oznacza to, że metoda `getSalary` klasy `Manager` nie ma bezpośredniego dostępu do pola `salary`, mimo że każdy obiekt tej klasy zawiera pole o nazwie `salary`. Jeśli metody klasy `Manager` chcą uzyskać do nich dostęp, muszą zrobić to co wszystkie inne metody — użyć interfejsu publicznego. W tym przypadku konieczne jest użycie metody `getSalary` klasy `Employee`.

Spróbujmy jeszcze raz. Zamiast bezpośrednio odwoływać się do pola `salary`, użyjemy metody `getSalary`:

```
public double getSalary()
{
   double baseSalary = getSalary();    // nadal nie działa
   return baseSalary + bonus;
}
```

Problem polega na tym, że metoda `getSalary` wywołuje *sama siebie*, ponieważ klasa `Manager` zawiera metodę o takiej nazwie (dokładniej mówiąc, jest to ta metoda, którą próbujemy zaimplementować). W ten sposób powstała nieskończona seria wywołań jednej metody, która prowadzi do załamania programu.

Musimy zaznaczyć, że odwołujemy się do metody `getSalary` klasy `Employee`, a nie klasy, w której się znajdujemy. Do tego celu służy słowo kluczowe `super`. Instrukcja:

```
super.getSalary()
```

wywoła metodę `getSalary` klasy `Employee`. Poniżej znajduje się poprawna wersja metody `get➥Salary` w klasie `Manager`:

```
public double getSalary()
{
   double baseSalary = super.getSalary();
   return baseSalary + bonus;
}
```

Niektórzy programiści uważają, że słowo kluczowe `super` jest odpowiednikiem referencji `this`. Nie jest to jednak precyzyjne porównanie, ponieważ słowo `super` nie jest referencją do obiektu. Nie można na przykład przypisać jego wartości do innej zmiennej obiektowej. `super` to słowo kluczowe, które nakazuje kompilatorowi wywołanie metody z nadklasy.

Wiemy już, że do podklasy można *dodawać* nowe pola oraz *dodawać* lub *przesłaniać* metody z nadklasy. Dziedziczenie nie umożliwia natomiast pozbycia się żadnych metod ani pól.

W Javie do wywoływania metod nadklasy służy słowo kluczowe `super`. W C++ w takiej sytuacji należy użyć nazwy nadklasy z operatorem `::`. Na przykład metoda `getSalary` klasy `Manager` wywoływałaby `Employee::getSalary` zamiast `super.getSalary`.

5.1.3. Konstruktory podklas

Na koniec dodamy konstruktor.

```
public Manager(String n, double s, int year, int month, int day)
{
   super(n, s, year, month, day);
   bonus = 0;
}
```

W tym miejscu słowo kluczowe `super` znaczy co innego. Instrukcja:

```
super(name, salary, year, month, day);
```

mówi: „wywołaj konstruktor nadklasy `Employee` z parametrami `n`, `s`, `year`, `month` i `day`”.

Ponieważ konstruktor `Manager` nie ma dostępu do pól prywatnych klasy `Employee`, musi je zainicjalizować poprzez inny konstruktor. Konstruktor jest wywoływany za pomocą specjalnej składni z użyciem słowa kluczowego `super`. Wywołanie z tym słowem kluczowym musi być pierwszą instrukcją konstruktora podklasy.

Jeśli obiekt podklasy jest tworzony bez jawnego wywołania konstruktora nadklasy, to nadklasa musi mieć konstruktor bezargumentowy, który jest wywoływany przed konstruktorem podklasy.

Przypomnijmy, że słowo `this` ma dwa zastosowania: określa referencję do parametru niejawnego i wywołuje inny konstruktor tej samej klasy. Podobnie dwa zastosowania ma słowo `super`: wywołuje metody nadklasy i konstruktory nadklasy. W przypadku wywoływania konstruktorów słowa te są blisko spokrewnione. Wywołanie konstruktora może następować tylko w pierwszej instrukcji innego konstruktora. Parametry konstrukcyjne są przekazywane albo do innego konstruktora tej samej klasy (`this`), albo do konstruktora nadklasy (`super`).

W języku C++ nie używa się w konstruktorze słowa kluczowego `super`, tylko składni listy inicjalizującej nadklasy. Konstruktor `Manager` w C++ wyglądałby następująco:

```
Manager::Manager(String n, double s, int year, int month, int day)   // C++
: Employee(n, s, year, month, day)
{
   bonus = 0;
}
```

Po ponownym zdefiniowaniu metody `getSalary` dla obiektów `Manager` dodatki do pensji kierowników będą dodawane *automatycznie*.

Oto praktyczny przykład obrazujący powyższe rozważania. Utworzymy nowy obiekt klasy `Manager` i ustawimy dodatek dla kierownika:

```
Manager boss = new Manager("Karol Parol", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

Tworzymy tablicę trzech pracowników:

```
Employee[] staff = new Employee[3];
```

Wstawiamy do niej obiekty zwykłych pracowników i kierowników:

```
staff[0] = boss;
staff[1] = new Employee("Henryk Kwiatek", 50000, 1989, 10, 1);
staff[2] = new Employee("Artur Kwiatkowski", 40000, 1990, 3, 15);
```

Wyświetlamy pensję każdego z nich:

```
for (Employee e : staff)
   System.out.println(e.getName() + " " + e.getSalary());
```

Powyższa pętla zwraca następujące dane:

```
Karol Parol 85000.0
Henryk Kwiatek 50000.0
Artur Kwiatkowski 40000.0
```

Obiekty `staff[1]` i `staff[2]` drukują podstawowe wynagrodzenie, ponieważ są to obiekty klasy `Employee`. Natomiast obiekt `Staff[0]` jest obiektem klasy `Manager`, a więc jego metoda `getSalary` dolicza dodatek do podstawowego wynagrodzenia.

Na uwagę zasługuje fakt, że wywołanie:

```
e.getSalary()
```

wybiera *odpowiednią* metodę `getSalary`. Warto zauważyć, że zmienna `e` jest *zadeklarowana* jako typ `Employee`, ale *rzeczywistym* typem, do którego odwołuje się ta zmienna, może być albo `Emp` ↪`loyee`, albo `Manager`.

Kiedy zmienna `e` odwołuje się do obiektu klasy `Employee`, wywołanie `e.getSalary()` wywołuje metodę `getSalary` klasy `Employee`. Jeśli jednak `e` odwołuje się do obiektu klasy `Manager`, metoda `getSalary` jest wywoływana z klasy `Manager`. Maszyna wirtualna rozpoznaje, do jakiego typu odwołuje się zmienna `e`, dzięki czemu może wywołać odpowiednią metodę.

Możliwość odwoływania się przez obiekty (jak zmienna `e`) do wielu różnych typów nosi nazwę **polimorfizmu** (ang. *polymorphism*). Automatyczny dobór odpowiednich metod w trakcie działania programu nazywa się **wiązaniem dynamicznym** (ang. *dynamic binding*). Oba te zagadnienia zostały szczegółowo opisane w tym rozdziale.

> **C++** W języku C++ wiązaniu dynamicznemu podlegają metody, które mają w deklaracji słowo kluczowe `virtual`. W Javie wiązanie dynamiczne jest domyślne. Aby metoda **nie była** wirtualna, należy użyć słowa kluczowego `final` (opis słowa kluczowego `final` znajduje się dalej w tym rozdziale).

Listing 5.1 przedstawia program demonstrujący różnicę naliczania pensji dla obiektów klasy `Employee` (listing 5.2) i `Manager` (listing 5.3).

Listing 5.1. inheritance/ManagerTest.java

```
package inheritance;

/**
 * Ten program demonstruje dziedziczenie.
 * @version 1.21 2004-02-21
 * @author Cay Horstmann
 */
public class ManagerTest
{
   public static void main(String[] args)
   {
      // Tworzenie obiektu klasy Manager.
      var boss = new Manager("Karol Parol", 80000, 1987, 12, 15);
      boss.setBonus(5000);

      var staff = new Employee[3];

      // Wstawienie obiektów klas Manager i Employee do tablicy staff.

      staff[0] = boss;
      staff[1] = new Employee("Henryk Kwiatek", 50000, 1989, 10, 1);
      staff[2] = new Employee("Artur Kwiatkowski", 40000, 1990, 3, 15);

      // Wyświetlanie informacji o wszystkich obiektach klasy Employee.
```

```
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
   }
}
```

Listing 5.2. inheritance/Employee.java

```
package inheritance;

import java.time.*;

public class Employee
{
   private String name;
   private double salary;
   private LocalDate hireDay;

   public Employee(String name, double salary, int year, int month, int day)
   {
      this.name = name;
      this.salary = salary;
      hireDay = LocalDate.of(year, month, day);
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public LocalDate getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }
}
```

Listing 5.3. inheritance/Manager.java

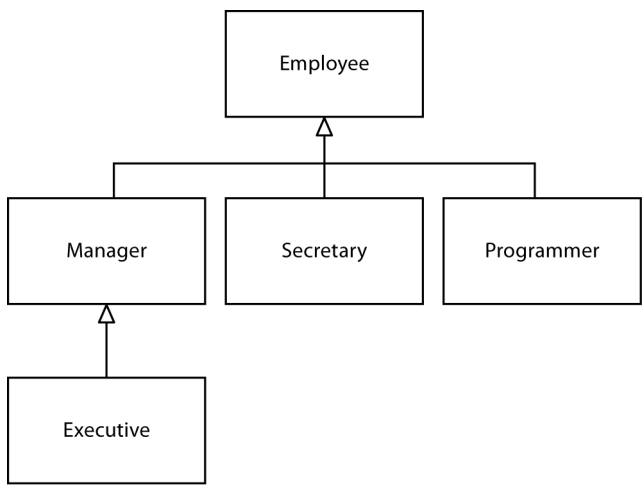
```
package inheritance;

public class Manager extends Employee
{
   private double bonus;
   /**
    * @param n imię i nazwisko pracownika
    * @param s pensja
    * @param year rok przyjęcia do pracy
```

```
     * @param month miesiąc przyjęcia do pracy
     * @param day dzień przyjęcia do pracy
     */
    public Manager(String n, double s, int year, int month, int day)
    {
       super(n, s, year, month, day);
       bonus = 0;
    }

    public double getSalary()
    {
       double baseSalary = super.getSalary();
       return baseSalary + bonus;
    }

    public void setBonus(double b)
    {
       bonus = b;
    }

}
```

5.1.4. Hierarchia dziedziczenia

Dziedziczenie nie dotyczy tylko jednego poziomu klas. Na przykład rozszerzeniem klasy `Manager` może być klasa `Executive` (dyrektor). Strukturę klas i ich drogi dziedziczenia od wspólnej klasy bazowej nazywa się **hierarchią dziedziczenia** (ang. *inheritance hierarchy*) — zobacz rysunek 5.1. Ścieżka od danej klasy do jej przodków w hierarchii dziedziczenia ma nazwę **łańcucha dziedziczenia** (ang. *inheritance chain*).

Rysunek 5.1. Hierarchia dziedziczenia klasy Employee



Łańcuchów dziedziczenia może być wiele. Na przykład klasę `Employee` mogą rozszerzać klasy `Programmer` i `Secretary`, które nie muszą mieć nic wspólnego z klasą `Manager` (albo ze sobą nawzajem). Struktura ta może mieć dowolną długość.


Java nie obsługuje dziedziczenia wielokrotnego. Sposoby naśladowania techniki dziedziczenia wielokrotnego są opisane w części o interfejsach w podrozdziale 6.1.

5.1.5. Polimorfizm

W podjęciu decyzji dotyczącej tego, czy w danym przypadku zastosować dziedziczenie, pomaga prosta zasada. Reguła „jest" mówi, że każdy obiekt podklasy jest obiektem nadklasy. Na przykład każdy kierownik jest pracownikiem. W związku z tym klasa `Manager` może być podklasą klasy `Employee`. Oczywiście twierdzenia tego nie można odwrócić — nie każdy pracownik jest kierownikiem.

Regułę relacji „jest" można także sformułować jako **zasadę zamienialności** (ang. *substitution principle*). Zasada ta głosi, że wszędzie tam, gdzie można użyć obiektu nadklasy, można użyć obiektu podklasy.

Można na przykład przypisać obiekt podklasy do zmiennej nadklasy.

```
Employee e;
e = new Employee(. . .);    // Powinien być obiekt klasy Employee.
e = new Manager(. . .);     // OK, obiekt klasy Manager też może być.
```

W Javie zmienne obiektowe są **polimorficzne**. Zmienna typu `Employee` może odwoływać się do dowolnego obiektu klasy `Employee`, jak również do każdego obiektu podklasy klasy `Employee` (jak `Manager`, `Executive`, `Secretary` itd.).

Skorzystaliśmy z tej zasady w programie z listingu 5.1:

```
Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];
staff[0] = boss;
```

W tym przypadku zmienne `staff[0]` i `boss` odwołują się do tego samego obiektu. Jednak dla kompilatora `staff[0]` jest obiektem wyłącznie klasy `Employee`.

Oznacza to, że można użyć poniższego wywołania:

```
boss.setBonus(5000);        // OK
```

Ale poniższe spowoduje błąd:

```
staff[0].setBonus(5000);    // błąd
```

Typ zadeklarowany zmiennej `staff[0]` to `Employee`, a metoda `setBonus` nie jest obecna w tej klasie.

Nie można przypisać referencji nadklasy do zmiennej podklasy. Na przykład poniższe przypisanie jest niedozwolone:

```
Manager m = staff[i];       // błąd
```

Powód jest jasny — nie wszyscy pracownicy są kierownikami. Gdyby powyższe przypisanie udało się i zmienna `m` byłaby referencją do obiektu klasy `Employee`, który nie jest kierownikiem, to możliwe byłoby też wywołanie `m.setBonus(...)`, a to z kolei spowodowałoby błąd czasu wykonania.

W Javie możliwa jest konwersja tablic referencji do obiektów podklasy na tablicę referencji do obiektów nadklasy bez rzutowania. Spójrzmy na poniższą tablicę obiektów klasy `Manager`:

```
Manager[] managers = new Manager[10];
```

Można ją przekonwertować na tablicę `Employee[]`:

```
Employee[] staff = managers;   // OK]
```

Można pomyśleć, czemu nie. Przecież każdy kierownik jest też pracownikiem. Jednak dzieje się tu coś zaskakującego. Pamiętajmy, że zmienne `managers` i `staff` są referencjami do tej samej tablicy. Spójrzmy teraz na poniższą instrukcję:

```
staff[0] = new Employee("Henryk Kwiatek", ...);
```

Kompilator nie zgłosi żadnego sprzeciwu przy kompilacji tego przypisania, ale zmienne `staff[0]` i `manager[0]` są tą samą referencją, a więc wygląda na to, że awansowaliśmy zwykłego pracownika na stanowisko kierownicze. Byłaby to bardzo zła sytuacja. Wywołanie `managers[0].setBonus(1000)` usiłowałoby uzyskać dostęp do nieistniejącego pola i spowodowałoby uszkodzenie okolicznej pamięci.

Aby uniknąć takiego zniszczenia, wszystkie tablice pamiętają, z jakim typem zostały utworzone, i pilnują, aby przechowywane w nich były tylko odpowiednie referencje. Na przykład tablica utworzona za pomocą instrukcji `new Manager[10]` pamięta, że jest tablicą kierowników. Próba zapisania w niej referencji do typu `Employee` spowoduje wyjątek `ArrayStoreException`.

5.1.6. Zasady wywoływania metod

Programista musi dokładnie wiedzieć, jak wywołanie metody jest stosowane do obiektu. Powiedzmy, że mamy w programie wywołanie `x.f(arg)` i że niejawny parametr `x` jest zadeklarowany jako obiekt klasy `C`. Oto co się dzieje:

1. Kompilator sprawdza zadeklarowany typ obiektu i nazwę metody. Pamiętajmy, że metod o nazwie `f` może być kilka, a różnica między nimi polega na tym, że mają różne listy parametrów. Na przykład mogą to być metody `f(int)` i `f(String)`. Kompilator tworzy listę wszystkich metod o nazwie `f` dostępnych w klasie `C` i wszystkich dostępnych metod o tej nazwie w nadklasie klasy `C` (prywatne metody nadklasy są niedostępne).

   W ten sposób powstaje lista wszystkich potencjalnych metod do wywołania.

2. Następnie kompilator sprawdza typy parametrów podanych w wywołaniu metody. Jeśli wśród zebranych metod o nazwie `f` znajduje się taka, której parametry pasują dokładnie do podanych parametrów, to zostaje ona wywołana. Proces ten nazywa się **rozstrzyganiem przeciążania**. Na przykład dla wywołania `x.f("Witaj")`

kompilator wybierze metodę `f(String)`, a nie `f(int)`. Sytuacja może się skomplikować ze względu na konwersję typów (`int` na `double`, `Manager` na `Employee` itd.). Jeśli kompilator nie może znaleźć metody pasującej do podanych parametrów lub znajdzie kilka pasujących metod, zgłasza błąd.

W ten sposób kompilator znajduje metodę, którą należy wywołać.

3. Jeśli metoda jest prywatna, statyczna lub finalna albo jest konstruktorem, kompilator wie, którą dokładnie metodę wywołać (modyfikator `final` jest opisany w kolejnym podrozdziale). Nazywa się to **wiązaniem statycznym** (ang. *static binding*). W przeciwnym przypadku to, która metoda zostanie wywołana, zależy od rzeczywistego typu parametru niejawnego; musi też być zastosowane wiązanie dynamiczne w trakcie działania programu. W naszym przykładzie kompilator wygenerowałby instrukcję wywołującą metodę `f(String)` za pomocą wiązania dynamicznego.

Przypomnijmy, że nazwa oraz lista typów parametrów metody są nazywane **sygnaturą** metody. Na przykład metody `f(int)` i `f(String)` mają takie same nazwy, ale różne sygnatury. Jeśli w podklasie zostanie zdefiniowana metoda o takiej samej sygnaturze jak metoda w klasie nadrzędnej, dana metoda nadklasy zostanie przesłonięta.

Typ zwrotny nie jest częścią sygnatury, ale musi się on w metodzie przesłaniającej zgadzać z tym w metodzie przesłoniętej. Podklasa może zmienić typ zwrotny na podtyp oryginalnego typu. Wyobraźmy sobie na przykład, że klasa `Employee` zawiera metodę

```
public Employee getKumpel() { ... }
```

Menedżer nie chciałby się kolegować ze zwykłym pracownikiem. Dlatego w podklasie metoda ta może zostać przesłonięta:

```
public Manager getKolega() { ... }    // Można zmienić typ zwrotny
```

Mówi się, że obie metody `getKolega` mają **kowariantne** typy zwrotne (ang. *covariant return types*).

4. Kiedy program działa i wywołuje metodę za pomocą wiązania dynamicznego, maszyna wirtualna musi wywołać tę wersję niniejszej metody, która odpowiada *rzeczywistemu* typowi obiektu, do którego odwołuje się zmienna `x`. Niech tym typem będzie `D` — podklasa klasy `C`. Jeśli klasa `D` zawiera definicję metody `f(String)`, wywołana zostaje właśnie ta metoda. W przeciwnym przypadku metoda ta jest szukana w nadklasie klasy `D` itd.

   Gdyby to wyszukiwanie było przeprowadzane przy każdym wywołaniu tej metody, tracono by dużo czasu. Dlatego maszyna wirtualna tworzy na początku **tabelę metod** zawierającą sygnatury i ciała wszystkich metod, które mogą być wywołane.

   Maszyna wirtualna może utworzyć tabelę metod po załadowaniu klasy, przez połączenie metod znalezionych w pliku klasy z tabelą metod nadklasy.

   Kiedy dana metoda jest wywoływana, maszyna wirtualna odszukuje ją w swojej tabeli. W naszym przykładzie maszyna wirtualna przeszukuje tabelę metod klasy `D` i odnajduje metodę `f(String)`. Może to być metoda `D.f(String)` albo `X.f(String)`,

gdzie X to jedna z nadklas klasy D. Scenariusz ten może się zmienić w jednej sytuacji. Jeśli wywołanie ma postać super.f(param), to maszyna wirtualna przeszukuje tabelę metod nadklasy.

Przeanalizujmy ten proces na przykładzie wywołania e.getSalary() z listingu 5.1. Obiekt e jest typu Employee. Klasa Employee zawiera tylko jedną metodę o nazwie getSalary, która nie ma parametrów. Dzięki temu w tym przypadku nie ma problemu z rozstrzyganiem przeciążania.

Ponieważ metoda getSalary nie jest prywatna, statyczna ani finalna, jest wiązana dynamicznie. Maszyna wirtualna tworzy tabele metod dla klas Employee i Manager. Tabela Employee wskazuje, że wszystkie jej metody są zdefiniowane w samej klasie Employee:

```
Employee:
   getName() -> Employee.getName()
   getSalary() -> Employee.getSalary()
   getHireDay() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
```

To jednak nie wszystko. Jak się niebawem dowiemy, klasa Employee ma nadklasę Object, po której dziedziczy kilka metod. Na razie ignorujemy te dodatkowe metody.

Tabela metod klasy Manager wygląda nieco inaczej. Trzy metody są odziedziczone, jedna przedefiniowana, a jedna została dodana.

```
Manager:
   getName() -> Employee.getName()
   getSalary() -> Manager.getSalary()
   getHireDay() -> Employee.getHireDay()
   raiseSalary(double) -> Employee.raiseSalary(double)
   setBonus(double) -> Manager.setBonus(double)
```

Oto co się dzieje z wywołaniem e.getSalary() w trakcie działania programu:

1. Maszyna wirtualna tworzy tabelę metod dla rzeczywistego typu e. Może to być tabela klasy Employee lub jednej z jej podklas, np. Manager.
2. Następnie maszyna wirtualna szuka klasy zawierającej definicję metody o sygnaturze getSalary(). W ten sposób znajduje metodę, którą ma wywołać.
3. Ostatecznie maszyna wirtualna wywołuje odpowiednią metodę.

Wiązanie dynamiczne ma jedną bardzo ważną cechę: umożliwia *rozszerzanie* programów bez modyfikacji istniejącego kodu. Przypuśćmy, że została utworzona klasa Executive i istnieje możliwość, że zmienna e odwołuje się do obiektu tej klasy. Kod zawierający wywołanie e.getSa
↪lary() nie musi być ponownie kompilowany. Metoda Executive.getSalary() zostanie wywołana automatycznie, jeśli zmienna e odwołuje się do obiektu typu Executive.

Kiedy metoda jest przesłaniana, jej odpowiednik w podklasie musi mieć *przynajmniej taką samą widoczność* jak oryginał. Jeśli metoda w nadklasie jest publiczna, w podklasie również musi być publiczna. Pominięcie specyfikatora public w metodzie podklasy jest częstym błędem. W takim przypadku kompilator informuje, że usiłowano zastosować niższy przywilej dostępu.

5.1.7. Wyłączanie dziedziczenia — klasy i metody finalne

Zdarzają się sytuacje, kiedy chcemy, aby nie tworzono podklas jednej z klas. Klasy, których nie można rozszerzać, nazywają się klasami **finalnymi** (ang. *final*), a do ich oznaczania służy modyfikator `final`. Załóżmy na przykład, że nie chcemy, aby klasa `Executive` była rozszerzana. W tym celu należy w jej definicji użyć modyfikatora `final`:

```
public final class Executive extends Manager
{
   . . .
}
```

Finalna może też być metoda w klasie. W takim przypadku nie można jej przesłonić w żadnej z podklas (wszystkie metody w klasie finalnej są finalne). Na przykład:

```
public class Employee
{
   . . .
   public final String getName()
   {
      return name;
   }
   . . .
}
```

Przypomnijmy, że pola również mogą być finalne. Wartość takiego pola nie może być zmieniana po utworzeniu obiektu. Jeśli klasa jest finalna, tylko jej metody są automatycznie finalne, nie dotyczy to pól.

Jest tylko jeden powód, dla którego warto zdefiniować klasę lub metodę jako finalną: aby zapewnić, że żadna podklasa nie zmieni semantyki. Na przykład metody `getTime` i `setTime` klasy `Calendar` są finalne. Oznacza to, że projektanci tej klasy wzięli na siebie ciężar odpowiedzialności za konwersję pomiędzy klasą `Date` a stanem kalendarza. Żadna podklasa nie powinna mieć możliwości mieszania się w to. Klasa `String` też jest finalna. Oznacza to, że nie można utworzyć jej podklasy. Innymi słowy, jeśli mamy referencję do obiektu `String`, wiemy, że jest to obiekt klasy `String` i nic innego.

Według niektórych programistów wszystkie metody powinny być finalne, chyba że istnieje dobry powód, dla którego potrzebny jest w danym przypadku polimorfizm. W C++ i C# metody nie są polimorficzne, dopóki jawnie się tego nie zażąda. Może to jest w pewnym sensie skrajne podejście, ale zgadzamy się, że przy projektowaniu hierarchii klas należy poważnie rozważyć możliwość zastosowania modyfikatora `final` dla klas i metod.

Na początku istnienia Javy niektórzy programiści używali słowa kluczowego `final` w nadziei, że unikną narzutu powodowanego przez wiązanie dynamiczne. Jeśli metoda nie jest przesłonięta i jest krótka, kompilator może zoptymalizować jej wywoływanie poprzez proces polegający na wstawieniu jej kodu w **miejsce wywołania** (ang. *inlining*). Na przykład wywołanie metody `e.getName()` zostałoby zastąpione dostępem do pola `e.name`. Jest to godna uwagi poprawa — procesory nie przepadają za rozgałęzianiem, ponieważ stoi ono w sprzeczności z ich strategią

pobierania zawczasu kolejnej funkcji podczas przetwarzania innej. Jeśli jednak metoda `getName` może być przesłonięta w innej klasie, kompilator nie może zastosować wstawiania kodu, ponieważ nie wie, jak działa ta przesłonięta wersja.

Na szczęście kompilator JIT w maszynie wirtualnej spisuje się lepiej niż zwykły kompilator. Wie dokładnie, które klasy są rozszerzeniem danej klasy, i ma możliwość sprawdzenia, czy dana metoda jest przesłonięta w którejś z tych klas. Jeśli metoda jest krótka, często wywoływana i nie jest przesłonięta, kompilator JIT może zastosować inlining. Co się stanie, jeśli maszyna wirtualna załaduje inną podklasę, która przesłania naszą metodę? Wtedy proces inliningu musi zostać cofnięty. Jest to operacja powolna, ale zdarza się bardzo rzadko.


Wyliczenia i rekordy zawsze są finalne, tzn. nie można ich rozszerzać.

5.1.8. Rzutowanie

Przypomnijmy z rozdziału 3., że proces wymuszania konwersji pomiędzy dwoma typami nazywa się rzutowaniem (ang. *casting*). W Javie dostępna jest specjalna notacja oznaczająca rzutowanie. Na przykład:

```
double x = 3.405;
int nx = (int) x;
```

W powyższym kodzie wartość zmiennej `x` została przekonwertowana na typ całkowitoliczbowy, co spowodowało utratę części ułamkowej.

Podobnie jak od czasu do czasu konieczna jest konwersja typu `double` na `int`, tak samo bywa, że trzeba przekonwertować referencję do obiektu jednej klasy na referencję do obiektu innej klasy. Po raz kolejny posłużę się przykładem tablicy zawierającej mieszankę obiektów klas `Employee` i `Manager`:

```
var staff = new Employee[3];
staff[0] = new Manager("Carl Cracker", 80000, 1987, 12, 15);
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

Do rzutowania referencji do obiektów używa się podobnej notacji jak do rzutowania typów liczbowych. Nazwę klasy docelowej należy umieścić w nawiasach i wstawić przed referencją, która ma być rzutowana. Na przykład:

```
Manager boss = (Manager) staff[0];
```

Tego typu rzutowanie może być potrzebne tylko w jednego rodzaju sytuacji — aby w pełni wykorzystać obiekt, którego typ został chwilowo zgubiony. Na przykład w klasie `TestManager` tablica `staff` musiała być typu `Employee`, ponieważ *niektóre* z przechowywanych w niej obiektów reprezentowały zwykłych pracowników. Aby uzyskać dostęp do nowych składowych obiektów klasy `Manager`, konieczne by było ich przekonwertowanie z powrotem na typ `Manager` (w zaprezentowanym wcześniej przykładowym kodzie specjalnie uniknęliśmy rzutowania, inicjalizując

zmienną boss obiektem klasy Manager przed zapisaniem jej w tablicy — aby ustawić dodatek do wypłaty, potrzebny jest odpowiedni typ obiektu).

Wiadomo, że każdy obiekt w Javie ma typ. Typ ten określa rodzaj obiektu, do którego odwołuje się zmienna, i wskazuje, co obiekt ten może robić. Na przykład zmienna staff[i] odwołuje się do obiektu klasy Employee (a więc może także odwoływać się do obiektu klasy Manager).

Kompilator sprawdza, czy programista nie wymaga zbyt wiele, zapisując wartość w zmiennej. Jeśli przypisze referencję do obiektu podklasy do zmiennej obiektowej nadklasy, zmniejsza swoje wymagania, więc kompilator bez problemu się na to zgodzi. Jeśli jednak referencja do obiektu nadklasy zostanie przypisana zmiennej obiektu podklasy, zwiększa swoje wymagania. W takim przypadku konieczne jest zastosowanie rzutowania, które umożliwi sprawdzenie tych wymagań w trakcie działania programu.

Co się stanie, jeśli programista spróbuje wykonać rzutowanie w dół łańcucha dziedziczenia i „oszuka" w kwestii zawartości obiektu?

```
Manager boss = (Manager) staff[1];   // Błąd
```

W trakcie działania programu systemy wykonawcze Javy wykryją niedorzeczne wymagania i wygenerują wyjątek ClassCastException. Jeśli wyjątek nie zostanie przechwycony, program zostanie zamknięty. W związku z tym do dobrych praktyk programistycznych należy sprawdzenie, czy rzutowanie się powiedzie, przed jego zastosowaniem. Do tego celu służy operator instan
↪ceof. Na przykład:

```
if (staff[1] instanceof Manager)
{
   boss = (Manager) staff[1];
   . . .
}
```

I wreszcie, kompilator nie pozwoli na rzutowanie, które nie ma szans powodzenia. Na przykład rzutowanie:

```
Date c = (Date) staff[1];
```

spowoduje błąd kompilacji, ponieważ Date nie jest podklasą klasy Employee.

Podsumowując:

- Rzutowanie jest możliwe wyłącznie w obrębie hierarchii dziedziczenia.
- Przy rzutowaniu typu nadklasy na typ podklasy należy sprawdzić wykonalność operacji za pomocą operatora instanceof.

Test:

```
x instanceof C
```

nie wygeneruje wyjątku, jeśli zmienna x ma wartość null, tylko zwróci wartość false. Sens tego zachowania polega na tym, że skoro null oznacza, iż referencja nie wskazuje na żaden obiekt, z pewnością nie odwołuje się do obiektu klasy C.

Faktem jest, że konwersja typu obiektu za pomocą rzutowania nie jest z reguły dobrym pomysłem. W naszym przykładzie rzadko potrzebne jest rzutowanie obiektu klasy `Employee` na obiekt klasy `Manager`. Metoda `getSalary` działa prawidłowo na obiektach obu tych klas. Wiązanie dynamiczne, na którym opiera się polimorfizm, automatycznie lokalizuje odpowiednią metodę.

Jedyna sytuacja, w której potrzebne jest takie rzutowanie, ma miejsce wtedy, gdy chcemy użyć metody dostępnej tylko dla obiektów klasy `Manager`, czyli `setBonus`. Jeśli wywołanie metody `set`
`↪Bonus` na rzecz obiektów klasy `Employee` okaże się konieczne, należy rozważyć możliwość, że nadklasa została źle zaprojektowana. Niewykluczone, że dobrym rozwiązaniem okaże się dodanie do nadklasy metody `setBonus`. Pamiętajmy, że do przerwania działania programu wystarczy jeden nieprzechwycony wyjątek. Zasadniczo najlepiej jest wystrzegać się rzutowania i operatora `instanceof`.

Składnia rzutowania w Javie pochodzi ze „starego i złego" języka C, natomiast działanie tego mechanizmu jest podobne do bezpiecznego rzutowania `dynamic_cast` w C++. Na przykład:

```
Manager boss = (Manager) staff[1];                    // Java
```

jest odpowiednikiem:

```
Manager* boss = dynamic_cast<Manager*>(staff[1]);    // C++
```

Jest tylko jedna różnica. Jeśli rzutowanie nie powiedzie się, nie powstaje obiekt `null`, ale wyjątek. W tym sensie przypomina to znane z C++ rzutowanie *referencji*. Jest to jedna z bolączek. W C++ można zadbać o test typu i konwersję w jednej operacji.

```
Manager* boss = dynamic_cast<Manager*>(staff[1]);    // C++
if (boss != NULL) . . .
```

W Javie konieczne jest użycie operatora `instanceof` i zastosowanie rzutowania:

```
if (staff[1] instanceof Manager)
{
   Manager boss = (Manager) staff[1];
   . . .
}
```

5.1.9. Operator instanceof i dopasowywanie wzorców

Kod

```
if (staff[i] instanceof Manager)
{
    Manager boss = (Manager) staff[i];
    boss.setBonus(5000);
}
```

nie grzeszy zwięzłością. Czy naprawdę musimy powtarzać nazwę podklasy `Manager` aż trzy razy?

Od Javy 16 można to uprościć. Zmienną podklasy można zadeklarować wprost w teście operatora `instanceof`:

```
if (staff[i] instanceof Manager boss)
{
    boss.setBonus(5000);
}
```

Jeśli `staff[i]` jest egzemplarzem klasy `Manager`, to zostaje on przypisany zmiennej `boss` i można jej używać jako obiektu klasy `Manager`. Pominęliśmy rzutowanie.

Jeśli `staff[i]` nie odnosi się do obiektu klasy `Manager`, zmienna `boss` nie otrzymuje przypisania i operator `instanceof` zwraca wartość `false`. Treść główna instrukcji `if` zostaje pominięta.

W większości sytuacji, w których jest używany operator `instanceof`, konieczne jest zastosowanie metody podklasy. W takich przypadkach, zamiast stosować rzutowanie, należy używać tej wersji operatora „z dopasowywaniem wzorców".

Bezsensowne użycie wzorca operatora `instanceof` jest błędem:

```
Manager boss = . . .;
if (boss instanceof Employee e) . . . // BŁĄD: to oczywiste, że to jest obiekt klasy Employee
```

Równie bezsensowna konstrukcja

```
if (boss instanceof Employee) . . .
```

jest dozwolona ze względu na zgodność z Javą 1.0.

Jeśli wzorzec operatora `instanceof` wprowadza zmienną, to można jej od razu użyć w tym samym wyrażeniu:

```
      Employee e;
if (e instanceof Manager m && m.getBonus() > 10000) . . .
```

To działa, ponieważ prawa strona wyrażenia `&&` jest obliczana tylko wtedy, gdy lewa strona jest prawdziwa. Jeśli prawa strona zostanie obliczona, to znaczy, że zmienna `m` została powiązana z egzemplarzem klasy `Manager`.

Jednak poniższy kod spowoduje błąd kompilacji:

```
if (e instanceof Manager m || m.getBonus() > 10000) . . . // BŁĄD
```

Prawa strona wyrażenia `||` jest wykonywana, gdy lewa strona jest fałszywa, a więc zmienna `m` nie jest z niczym powiązana.

Poniżej znajduje się kolejny przykład, w którym użyto operatora warunkowego:

```
double bonus = e instanceof Manager m ? m.getBonus() : 0;
```

Zmienna `m` jest zdefiniowana w części wyrażenia za znakiem `?`, ale nie ma jej w części za znakiem `:`.

Wersja operatora `instanceof` z deklaracją zmiennej to tzw. wersja „z dopasowywaniem wzorca", ponieważ jest podobna do wzorców typów w konstrukcji `switch`, która na próbę została dodana do Javy 17. Nie opisuję szczegółowo próbnych konstrukcji, ale poniżej przedstawiam przykład składni:

```
String description = switch (e)
{
    case Executive exec -> "Dyrektor o efektownym tytule " + exec.getTitle();
    case Manager m -> "Kierownik, któremu przysługuje premia " + m.getBonus();
    default -> "Zwykły pracownik zarabiający " + e.getSalary();
}
```

Podobnie jak w przypadku wzorca operatora `instanceof` każdy wzorzec typu deklaruje zmienną.

Zmienna lokalna zdefiniowana przez wzorzec operatora `instanceof` jak każda zmienna lokalna może zasłaniać pole. Na przykład:

```
class Value
{
    private double v;
    public boolean equals(Value other)
    {
        if (other instanceof LabeledValue v)
            // v jest taka sama jak other
        else
            // v oznacza pole
    }
    . . .
}
```

5.1.10. Ograniczanie dostępu

Wiemy już, że najlepiej jest, kiedy pola metody są prywatne, a metody publiczne. Wszystko, co jest oznaczone słowem kluczowym `private`, jest niewidoczne dla innych klas. Na początku tego rozdziału dowiedzieliśmy się też, że powyższa zasada dotyczy także podklas — podklasa nie ma dostępu do pól prywatnych swojej nadklasy.

W niektórych sytuacjach konieczne jest ograniczenie widoczności metody tylko do podklas lub, rzadziej, zezwolenie metodom podklas na dostęp do pól nadklasy. W takim przypadku należy użyć słowa kluczowego `protected`. Jeśli na przykład klasa `Employee` zawiera pole `hireDay` zadeklarowane jako `protected`, a nie `private`, metody klasy `Manager` mają do tego pola bezpośredni dostęp.

W Javie pole chronione jest dostępne dla każdej klasy znajdującej się w tym samym pakiecie. Teraz wyobraź sobie podklasę `Administrator` z innego pakietu. Jej metody mogą zaglądać tylko do pola `hireDay` klasy `Administrator`, ale nie klasy `Employee`. Ograniczenie to ma zapobiec nadużywaniu mechanizmu ochrony w celu tworzenia podklas tylko po to, aby uzyskać dostęp do chronionych pól.

W praktyce z pól chronionych należy korzystać ostrożnie. Załóżmy, że nasza klasa, która zawiera pola chronione, jest używana przez innych programistów. Ci programiści mogą tworzyć bez naszej wiedzy klasy dziedziczące po naszej klasie i w ten sposób uzyskać dostęp do chronionych pól naszej klasy. W takiej sytuacji zmiana implementacji owej nadklasy pociągałaby za sobą problemy u wspomnianych programistów. Takie działanie jest sprzeczne z ideą OOP, która opiera się na hermetyzacji danych.

Więcej sensu ma tworzenie chronionych metod. Metoda może być chroniona, jeśli jej użycie może sprawiać problemy. Oznacza to, że podklasy (które prawdopodobnie dobrze znają swoich przodków) z pewnością użyją danej metody prawidłowo, podczas gdy metody innych klas niekoniecznie.

Dobrym przykładem tego rodzaju metody jest metoda `clone` z klasy `Object` — więcej szczegółów znajdziesz w rozdziale 6.

> **C++** Składowe chronione klasy w Javie są widoczne dla wszystkich jej podklas i innych klas w pakiecie. Jest to nieco inne pojęcie ochrony niż w C++ i powoduje ono, że składowe chronione w Javie są jeszcze mniej bezpieczne niż w C++.

Oto zestawienie wszystkich czterech modyfikatorów dostępu Javy służących do kontroli widoczności:

1. `private` — widoczność w obrębie klasy;
2. `public` — widoczność wszędzie;
3. `protected` — widoczność w pakiecie i wszystkich podklasach;
4. widoczność w obrębie pakietu — (niefortunne) zachowanie domyślne, które nie wymaga żadnego modyfikatora.

5.2. Kosmiczna klasa wszystkich klas — Object

Klasa `Object` jest podstawą wszystkich pozostałych klas w Javie. Każda klasa w tym języku rozszerza klasę `Object`. Nigdy jednak nie trzeba pisać czegoś takiego:

```
public class Employee extends Object
```

Jeśli żadna nadklasa nie jest jawnie podana, to automatycznie jest nią klasa `Object`. Ponieważ *każda* klasa w Javie stanowi rozszerzenie klasy `Object`, trzeba się zapoznać z usługami świadczonymi przez tę klasę. W tym rozdziale opisujemy tylko podstawowe zagadnienia, a po szczegółowe informacje odsyłamy do kolejnych rozdziałów lub dokumentacji internetowej (niektóre metody klasy `Object` mogą być używane tylko podczas pracy z wątkami — więcej o wątkach dowiesz się w rozdziale 12.).

5.2.1. Zmienne typu Object

Za pomocą zmiennej typu `Object` można się odwoływać do wszystkich typów obiektów:

```
Object obj = new Employee("Henryk Kwiatek", 35000);
```

Oczywiście zmienna typu `Object` jest jedynie generycznym kontenerem dla dowolnych wartości. Aby zrobić z niej użytek, trzeba posiadać wiedzę na temat oryginalnego typu i wykonać rzutowanie:

```
Employee e = (Employee) obj;
```

W Javie tylko *typy podstawowe* (liczby, znaki i wartości logiczne) nie są obiektami.

Wszystkie typy tablicowe — bez względu na to, czy przechowują obiekty, czy typy podstawowe — są typami klasowymi rozszerzającymi klasę `Object`.

```
Employee[] staff = new Employee[10];
obj = staff;          // OK
obj = new int[10];    // OK
```

> W języku C++ nie ma uniwersalnej klasy bazowej, jednak każdy wskaźnik można przekonwertować na wskaźnik `void*`.

5.2.2. Metoda equals

Dostępna w klasie `Object` metoda `equals` porównuje dwa obiekty. Jej implementacja w klasie `Object` sprawdza, czy dwie referencje do obiektów są identyczne. Jest to bardzo rozsądne działanie domyślne — jeśli dwa obiekty są identyczne, powinny być sobie równe. W przypadku wielu klas nie potrzeba nic więcej. Na przykład porównywanie dwóch obiektów `PrintStream` pod kątem równości nie ma większego sensu, jednak często potrzebne jest porównywanie stanów, czyli sytuacji, w której dwa obiekty są równe, jeśli mają ten sam stan.

Na przykład dwóch pracowników uznamy za równych, jeśli mają identyczne imię i nazwisko, pensję i zostali zatrudnieni w tym samym czasie (w prawdziwej bazie danych pracowników lepiej byłoby porównać identyfikatory pracowników; ten przykład ma na celu zobrazowanie mechanizmów implementacyjnych metody `equals`).

```
public class Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      // Szybkie sprawdzenie, czy obiekty są identyczne.
      if (this == otherObject) return true;

      // Musi zwrócić false, jeśli parametr jawny ma wartość null.
      if (otherObject == null) return false;

      // Jeśli klasy nie pasują, nie mogą być równe.
```

```
        if (getClass() != otherObject.getClass())
           return false;

        // Wiadomo, że otherObject nie jest obiektem null klasy Employee.
        Employee other = (Employee) otherObject;

        // Sprawdzenie, czy pola mają identyczne wartości.
        return name.equals(other.name)
           && salary == other.salary
           && hireDay.equals(other.hireDay);
     }
}
```

Metoda `getClass` zwraca nazwę klasy obiektu — szczegółowy opis tej metody znajduje się w dalszej części tego rozdziału. W naszym teście dwa obiekty mogą być równe tylko wtedy, kiedy należą do tej samej klasy.

Aby zabezpieczyć się na wypadek, gdyby zmienne `name` lub `hireDay` były `null`, można użyć metody `Objects.equals`. Wywołanie `Objects.equals(a, b)` zwraca `true`, jeśli oba argumenty są `null`, `false` — jeśli jeden z argumentów jest `null`, a w pozostałych przypadkach wywołuje `a.equals(b)`. Przy użyciu tej metody ostatnią instrukcję w metodzie `Employee.equals` można zmienić następująco:

```
return Objects.equals(name, other.name)
   && salary == other.salary
   && Objects.equals(hireDay, other.hireDay);
```

Implementując metodę `equals` w podklasie, najpierw należy wywołać metodę `equals` należącą do nadklasy. Jeśli test zakończy się niepowodzeniem, obiekty nie mogą być równe. Jeśli pola nadklasy są równe, można porównywać składowe obiektów podklasy.

```
public class Manager extends Employee
{
   . . .
   public boolean equals(Object otherObject)
   {
      if (!super.equals(otherObject)) return false;
      // Metoda super.equals sprawdziła, czy this i otherObject należą do tej samej klasy.
      Manager other = (Manager) otherObject;
      return bonus == other.bonus;
   }
}
```

Przypomnę z rozdziału 4., że rekord jest specjalnym rodzajem niezmiennej klasy, której stan jest w całości zdefiniowany przez pola ustawione w konstruktorze „kanonicznym". Rekordy automatycznie definiują metodę `equals`, która porównuje pola. Dwa egzemplarze rekordu są równe, gdy odpowiednie wartości pól są równe.

5.2.3. Porównywanie a dziedziczenie

Jak powinna się zachować metoda `equals`, jeśli parametry jawny i niejawny nie należą do tej samej klasy? Jest to dość kontrowersyjna kwestia. W poprzednim przykładzie metoda `equals` zwracała wartość `false`, jeśli klasy nie były identyczne. Jednak wielu programistów zamiast tej metody używa operatora `instanceof`:

```
if (!(otherObject instanceof Employee)) return false;
```

Dzięki temu obiekt `otherObject` może należeć do podklasy klasy `Employee`. Metoda ta może jednak sprawiać problemy. Specyfikacja języka Java wymaga, aby metoda `equals` miała następujące własności:

1. **Zwrotność**: `x.equals(x)` powinno zwracać `true`, jeśli `x` nie ma wartości `null`.
2. **Symetria**: dla dowolnych referencji `x` i `y`, `x.equals(y)` powinno zwrócić wartość `true` wtedy i tylko wtedy, gdy `y.equals(x)` zwróci wartość `true`.
3. **Przemienność**: dla dowolnych referencji `x`, `y` i `z`, jeśli `x.equals(y)` zwraca wartość `true` i `y.equals(z)` zwraca `true`, to `x.equals(y)` zwraca tę samą wartość.
4. **Niezmienność**: jeśli obiekty, do których odwołują się zmienne `x` i `y`, nie zmieniły się, kolejne wywołania `x.equals(y)` zwracają tę samą wartość.
5. Dla każdego `x` różnego od `null`, wywołanie `x.equals(null)` powinno zwrócić wartość `false`.

Powyższe reguły są oczywiście podyktowane zdrowym rozsądkiem. Programista biblioteki nie powinien być zmuszany do tracenia czasu na podejmowanie decyzji, czy wywołać `x.equals(y)`, czy `y.equals(x)`, aby zlokalizować jakiś element w strukturze danych.

Jednak w przypadku gdy parametry należą do różnych klas, reguła symetrii może pociągnąć za sobą pewne konsekwencje. Przeanalizujmy poniższe wywołanie:

```
e.equals(m)
```

gdzie `e` jest obiektem klasy `Employee`, a `m` — klasy `Manager`. Tak się składa, że każdy z nich ma takie samo imię i nazwisko, pensję i datę zatrudnienia. Jeśli wywołanie `Employee.equals` użyje operatora `instanceof`, zostanie zwrócona wartość `true`. Oznacza to jednak, że wywołanie odwrotne:

```
m.equals(e)
```

także musi zwrócić wartość `true` — reguła symetrii nie zezwala na zwrócenie wartości `false` ani spowodowanie wyjątku.

To krępuje klasę `Manager`. Jej metoda `equals` zmuszą ja do porównywania się z klasą `Employee` bez brania pod uwagę informacji właściwych tylko kierownikom! Nagle operator `instanceof` przestał wydawać się tak atrakcyjny!

Niektórzy twierdzą, że test `getClass` jest błędny, ponieważ łamie regułę zamienialności. Często przytaczany jest przykład metody `equals` w klasie `AbstractSet`, która sprawdza, czy dwa zbiory

mają te same elementy. Klasa `AbstractSet` ma dwie konkretne podklasy: `TreeSet` i `HashSet`. Każda z nich lokalizuje elementy za pomocą innego algorytmu. Bez względu na implementację potrzebna jest możliwość porównywania zbiorów.

Jednak przykład zbioru dotyczy raczej wąskiej specjalizacji. Można by było zdefiniować metodę `AbstractSet.equals` jako finalną, ponieważ nikt nie powinien zmieniać semantyki równości zbiorów (obecnie metoda ta nie jest finalna, dzięki czemu możliwe jest zaimplementowanie w podklasie bardziej efektywnego algorytmu porównującego).

Z naszego punktu widzenia możliwe są dwa odrębne scenariusze:

- Jeśli podklasy mają własny mechanizm porównywania, reguła symetrii zmusza nas do użycia testu `getClass`.
- Jeśli mechanizm porównujący jest ustalony w nadklasie, można użyć operatora `instanceof`, pozwalając, aby obiekty różnych podklas były równe.

W przypadku klas `Employee` i `Manager` dwa obiekty uznajemy za równe, jeśli mają takie same pola. Jeśli dwa obiekty klasy `Manager` mają takie same imiona i nazwiska, pensje i daty zatrudnienia, ale różne dodatki do pensji, chcemy, aby były uznane za różne. Dlatego użyliśmy testu `getClass`.

Załóżmy jednak, że do porównywania użyliśmy identyfikatorów pracowników. Ten rodzaj porównania jest odpowiedni dla wszystkich podklas. W takim przypadku moglibyśmy zastosować operator `instanceof`, a metodę `Employee.equals` zadeklarować jako finalną.

Standardowa biblioteka Javy zawiera ponad 150 implementacji metody `equals`. Znajdują się wśród nich wersje z operatorem `instanceof`, wywołaniem `getClass`, przechwytywaniem wyjątku `ClassCastException` i nierobiące nic. W dokumentacji API klasy `java.sql.Timestamp` można znaleźć notatkę, w której implementatorzy sami ze wstydem przyznają, że zapędzili się w kozi róg. Klasa `java.sql.Timestamp` dziedziczy po klasie `java.util.Date`, której metoda `equals` wykorzystuje operator `instanceof`. Nie da się przesłonić metody `equals`, aby była dokładna i symetryczna.

Poniżej znajduje się opis procedury pisania idealnej metody `equals`:

1. Nadaj parametrowi jawnemu nazwę `otherObject` — później konieczne będzie rzutowanie go na inną zmienną, która powinna mieć nazwę `other`.
2. Sprawdź, czy `this` i `otherObject` są identyczne:

   ```
   if (this == otherObject) return true;
   ```

   Ta instrukcja służy tylko do optymalizacji. Jest to dość często spotykany przypadek. Łatwiej sprawdzić tożsamość obiektów, niż porównywać ich pola.
3. Sprawdź, czy obiekt `otherObject` jest równy `null`, i zwróć wartość `false`, jeśli jest.

   ```
   if (otherObject == null) return false;
   ```

4. Porównaj klasy obiektów `this` i `otherObject`. Jeśli semantyka metody `equals` może zmienić się w podklasach, użyj testu `getClass`:

```
if (getClass() != otherObject.getClass()) return false;
ClassName other = (ClassName) otherObject;
```

   Jeśli *wszystkie* podklasy korzystają z tej samej semantyki, można użyć operatora `instanceof`:

```
if (!(otherObject instanceof ClassName)) return false;
```

   Zwróć uwagę, że test operatora `instanceof` w przypadku powodzenia ustawia zmienną `other` na `otherObject`. Nie trzeba stosować rzutowania.

5. Rzutuj obiekt `otherObject` na zmienną typu swojej klasy:

```
ClassName other = (ClassName) otherObject
```

6. Porównaj pola zgodnie z własnymi wymaganiami. Dla pól typów podstawowych użyj operatora `==`, a metody `Objects.equals` dla obiektów. Zwróć wartość `true`, jeśli wszystkie pola się zgadzają, lub `false` w przeciwnym przypadku.

```
return field1 == other.field1
   && field2.equals(other.field2)
   && . . .;
```

   Jeśli przedefiniujesz metodę `equals` w podklasie, użyj odwołania `super.equals(other)`.

Do porównania elementów dwóch tablic można użyć statycznej metody `Arrays.equals`. W przypadku tablic wielowymiarowych należy używać metody `Arrays.deepEquals`.

Poniżej znajduje się często spotykany błąd w implementacji metody `equals`. Na czym on polega?

```
public class Employee
{
   public boolean equals(Employee other)
   {
      return other != null
        && getClass() == other.getClass()
        && Objects.equals(name, other.name)
        && salary == other.salary
        && Objects.equals(hireDay, other.hireDay);
   }
   ...
}
```

Ta metoda deklaruje parametr jawny jako typ `Employee`. W wyniku tego nie przesłania ona metody `equals` z klasy `Object`, ale tworzy zupełnie nową metodę.

Można chronić się przed tego typu błędem, oznaczając metody mające przesłaniać metody nadklasy znacznikiem `@Override`:

```
@Override public boolean equals(Object other)
```

> Jeśli zostanie popełniony błąd i zdefiniowana nowa metoda, kompilator zgłosi błąd. Przypuśćmy na przykład, że dodano poniższą deklarację do klasy `Employee`:
>
> ```
> @Override public boolean equals(Employee other)
> ```
>
> Zostanie zgłoszony błąd, ponieważ ta metoda nie przesłania żadnej metody w nadklasie `Object`.

`java.util.Arrays` **1.2**

- `static boolean equals(`*xxx*`[] a, `*xxx*`[] b)` **5**

  Zwraca wartość `true`, jeśli tablice są równe pod względem liczby elementów i elementy te są takie same na odpowiadających sobie pozycjach w obu tablicach. Tablice te mogą przechowywać elementy następujących typów: `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, `double`.

`java.util.Objects` **7**

- `static boolean equals(Object a, Object b)`

  Zwraca wartość `true`, jeśli `a` i `b` są `null`, `false` — jeśli `a` lub `b` jest `null`, oraz `a.equals(b)` w pozostałych przypadkach.

5.2.4. Metoda hashCode

Wartość skrótu (ang. *hash code*) obiektu to obliczona dla niego specjalna liczba całkowita. Skróty powinny mieć różne wartości. To znaczy, że jeśli `x` i `y` to dwa różne obiekty, powinno istnieć wysokie prawdopodobieństwo, że `x.hashCode()` i `y.hashCode()` to dwie różne liczby. Tabela 5.1 przedstawia trzy przykładowe wartości skrótu zwrócone przez metodę `hashCode` klasy `String`.

Tabela 5.1. Wartości skrótu zwrócone przez metodę hashCode

Łańcuch	Wartość skrótu
`Witaj`	`83588971`
`Henryk`	`-2137002381`
`Kwiatek`	`1350142454`

Klasa `String` oblicza wartości skrótu za pomocą następującego algorytmu:

```
int hash = 0;
for (int i = 0; i < length(); i++)
   hash = 31 * hash + charAt(i);
```

Metoda `hashCode` znajduje się w klasie `Object`. Dlatego każdy obiekt ma domyślny skrót, który jest obliczany na podstawie adresu obiektu w pamięci. Przeanalizujmy poniższy kod:

```
var s = "Ok";
var sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
```

```
var t = new String("Ok");
var tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Wynik przedstawia tabela 5.2.

Tabela 5.2. Skróty łańcuchów i obiektów klasy StringBuilder

Obiekt	Wartość skrótu	Obiekt	Wartość
s	2556	t	2556
sb	20526976	tb	20527144

Należy zauważyć, że łańcuchy `s` i `t` mają takie same wartości skrótu, ponieważ skróty łańcuchów są pochodnymi ich *zawartości*. Obiekty `sb` i `tb` mają różne skróty, ponieważ dla klasy `String ➥Builder` nie zdefiniowano metody `hashCode`. W związku z tym domyślna metoda `hashCode` klasy `Object` wygenerowała ich wartości skrótu na podstawie adresów w pamięci.

W przypadku przedefiniowania metody `equals` należy także przedefiniować metodę `hashCode` dla obiektów, które użytkownicy mogą wstawiać do tablicy skrótów (ang. *hash table*) — tablice skrótów zostały opisane w rozdziale 9.

Metoda `hashCode` powinna zwracać liczbę całkowitą (może być ujemna). Aby wartości skrótu różnych obiektów były różne, wystarczy użyć kombinacji skrótów pól tych obiektów.

Poniżej znajduje się przykładowa metoda `hashCode` klasy `Employee`:

```
public class Employee
{
   public int hashCode()
   {
      return 7 * name.hashCode()
      + 11 * Double.valueOf(salary).hashCode()
      + 13 * hireDay.hashCode();
   }
   . . .
}
```

Można tu jednak dokonać dwóch udoskonaleń. Po pierwsze, lepiej użyć zabezpieczonej przed `null` metody `Objects.hashCode`, która zwraca `0`, jeśli jej argument jest `null`, albo wynik wywołania `hashCode` na tym argumencie w pozostałych przypadkach. Dodatkowo można użyć statycznej metody `Double.hashCode`, aby nie musieć tworzyć obiektu typu `Double`:

```
public int hashCode()
{
   return 7 * Objects.hashCode(name)
   + 11 * Double.hashCode(salary)
   + 13 * Objects.hashCode(hireDay);
}
```

Po drugie, gdy trzeba połączyć kilka wartości skrótu (ang. *hash value*), można wywołać metodę `Objects.hash`, przekazując jej wszystkie te wartości. Spowoduje to wywołanie metody `Objects. ➥hashCode` dla każdego argumentu i połączenie wartości. Wówczas metoda `Employee.hashCode` może być o wiele prostsza:

```
public int hashCode()
{
   return Objects.hash(name, salary, hireDay);
}
```

Definicje metod `equals` i `hashCode` muszą się ze sobą zgadzać — jeśli `x.equals(y)` zwraca wartość `true`, to `x.hashCode()` musi mieć taką samą wartość jak `y.hashCode()`. Jeśli na przykład metoda `Employee.equals` porównuje identyfikatory pracowników, metoda `hashCode` musi obliczać skróty na podstawie identyfikatorów, a nie imion i nazwisk pracowników czy adresów w pamięci.

Jeśli pola są typu tablicowego, można użyć metody `Arrays.hashCode`, która oblicza skrót złożony ze skrótów elementów tablicy.

Typ `record` automatycznie dostarcza metodę `hashCode`, która generuje kod skrótu z kodów skrótu wartości pól.

Jeśli zmienne egzemplarza mają małe zakresy możliwych wartości, to należy osiągnąć możliwie jak najwięcej osobnych kodów skrótu. Weźmy na przykład daty z kalendarza. Operacja 7 * rok + 11 * miesiąc + 13 * dzień generuje wiele kolizji. Natomiast 31 * 12 * rok + 31 * miesiąc + dzień jest „idealną funkcją skrótu". Przy założeniu rozsądnego zakresu lat żadne dwie daty nie mają takiego samego kodu skrótu. (Metoda `hashCode` z klasy `LocalDate`, która obsługuje zakres ±999 999 999 lat, jest nieco bardziej skomplikowana).

`java.lang.Object` **1.0**

- `int hashCode()`

  Zwraca wartość skrótu dla obiektu. Skrót ten może być każdą dodatnią lub ujemną liczbą całkowitą. Identyczne obiekty powinny mieć takie same skróty.

`java.lang.Objects` **7**

- `int hash(Object... objects)`

  Zwraca wartość skrótu będącą kombinacją wartości skrótu wszystkich podanych obiektów.

- `static int hashCode(Object a)`

  Zwraca 0, jeśli `a` jest `null`, lub `a.hashCode()` w pozostałych przypadkach.

`java.lang.(Integer|Long|Short|Byte|Double|Float|Character|Boolean)` **1.0**

- `static int hashCode(`*xxx* `value)` **8**

  Zwraca skrót podanej wartości. W tym przypadku *xxx* oznacza typ podstawowy odpowiadający danemu typowi opakowania.

`java.util.Arrays` **1.2**

- `static int hashCode(xxx[] a)` **5**

  Oblicza skrót tablicy `a`, która może przechowywać elementy następujących typów: `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, `double`.

5.2.5. Metoda toString

Kolejną ważną metodą klasy `Object` jest metoda `toString`, która zwraca obiekt reprezentujący wartość obiektu. Z typowym przykładem jej zastosowania mamy do czynienia, gdy metoda `toString` klasy `Point` zwraca następujący łańcuch:

```
java.awt.Point[x=10,y=20]
```

Większość metod `toString` (ale nie wszystkie) ma następujący format: nazwa klasy plus wartości pól wymienione w nawiasach kwadratowych. Poniżej znajduje się implementacja metody `toString` w klasie `Employee`:

```
public String toString()
{
   return "Employee[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

Można to jednak zrobić nieco lepiej. Zamiast na sztywno wpisywać nazwę klasy w metodzie `toString`, nazwę tę można pobrać za pomocą wywołania `getClass().getName()`.

```
public String toString()
{
   return getClass().getName()
      + "[name=" + name
      + ",salary=" + salary
      + ",hireDay=" + hireDay
      + "]";
}
```

Dzięki temu metoda ta będzie działała także w podklasach.

Oczywiście twórca podklasy powinien zdefiniować własną metodę `toString`, uwzględniającą pola tej klasy. Jeśli w nadklasie użyto wywołania `getClass().getName()`, w podklasie można użyć wywołania `super.toString()`. Poniżej znajduje się przykładowa metoda `toString` klasy `Manager`:

```
public class Manager extends Employee
{
   . . .
   public String toString()
   {
      return super.toString()
         + "[bonus=" + bonus
         + "]";
   }
}
```

Wydruk zawartości obiektu Manager wyglądałby następująco:

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Metoda toString jest bardzo często używana z jednego ważnego powodu: za każdym razem, gdy obiekt jest łączony z łańcuchem za pomocą operatora +, kompilator automatycznie wywołuje metodę toString, aby utworzyć łańcuchową reprezentację obiektu. Na przykład:

```
var p = new Point(10, 20);
String message = "Aktualne położenie to " + p;
// Automatyczne wywołanie p.toString().
```

Zamiast x.toString() można napisać "" + x. Ta instrukcja łączy pusty łańcuch z łańcuchową reprezentacją x, czyli robi dokładnie to samo co x.toString(). W przeciwieństwie do metody toString, ta instrukcja działa nawet wtedy, gdy x jest typu podstawowego.

Jeśli x jest dowolnego typu, w wywołaniu:

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

metoda println wywołuje x.toString() i drukuje powstały w ten sposób łańcuch.

Klasa Object zawiera metodę toString, która drukuje nazwę klasy i skrót obiektu. Na przykład wywołanie:

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

zwróci wynik podobny do poniższego:

```
java.io.PrintStream@187aeca
```

Jest to spowodowane tym, że programista implementujący klasę PrintStream nie zadał sobie trudu, aby przesłonić metodę toString.

Niestety tablice dziedziczą metodę toString po klasie Object, przez co typ tablicy jest drukowany w przestarzałym formacie. Na przykład:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

Powyższy kod zwraca łańcuch typu [I@e48e1b (przedrostek [I oznacza tablicę liczb całkowitych). Można temu zaradzić poprzez wywołanie metody Arrays.toString. Poniższy kod:

```
String s = Arrays.toString(luckyNumbers);
```

zwróci łańcuch [2, 3, 5, 7, 11, 13].

Aby prawidłowo wydrukować zawartość tablicy wielowymiarowej (tzn. tablicy tablic), należy użyć metody Arrays.deepToString.

Metoda `toString` jest doskonałym narzędziem do rejestracji danych. Wiele klas biblioteki standardowej zawiera metodę `toString`, która umożliwia uzyskanie informacji na temat stanu obiektu. Jest to szczególnie przydatne w przypadku komunikatów rejestracyjnych typu:

```
System.out.println("Aktualne położenie = " + position);
```

Jak piszemy w rozdziale 7., jeszcze lepszym rozwiązaniem jest użycie obiektu klasy `Logger` i zastosowanie następującego wywołania:

```
Logger.global.info("Aktualne położenie = " + position);
```

Zdecydowanie zalecam dodawanie metody `toString` do każdej nowej klasy. Każdy, kto będzie tych klas używał, z pewnością doceni ułatwienia dotyczące rejestracji danych.

W typach `record` metoda `toString` jest od razu dostępna. Zwraca ona nazwę klasy oraz nazwy i zamienione na łańcuchy wartości pól.

Program z listingu 5.4 testuje metody `equals`, `hashCode` oraz `toString` w klasach `Employee` (listing 5.5) i `Manager` (listing 5.6).

Listing 5.4. equals/EqualsTest.java

```
package equals;

/**
 * Jest to program demonstrujący użycie metody equals.
 * @version 1.12 2012-01-26
 * @author Cay Horstmann
 */
public class EqualsTest
{
   public static void main(String[] args)
   {
      var alice1 = new Employee("Alicja Adamczuk", 75000, 1987, 12, 15);
      var alice2 = alice1;
      var alice3 = new Employee("Alicja Adamczuk", 75000, 1987, 12, 15);
      var bob = new Employee("Bartosz Borkowski", 50000, 1989, 10, 1);

      System.out.println("alice1 == alice2: " + (alice1 == alice2));

      System.out.println("alice1 == alice3: " + (alice1 == alice3));

      System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));

      System.out.println("alice1.equals(bob): " + alice1.equals(bob));

      System.out.println("bob.toString(): " + bob);

      var carl = new Manager("Karol Krakowski", 80000, 1987, 12, 15);
      var boss = new Manager("Karol Krakowski", 80000, 1987, 12, 15);
      boss.setBonus(5000);
      System.out.println("boss.toString(): " + boss);
      System.out.println("carl.equals(boss): " + carl.equals(boss));
```

```
      System.out.println("alice1.hashCode(): " + alice1.hashCode());
      System.out.println("alice3.hashCode(): " + alice3.hashCode());
      System.out.println("bob.hashCode(): " + bob.hashCode());
      System.out.println("carl.hashCode(): " + carl.hashCode());
   }
}
```

Listing 5.5. equals/Employee.java

```
package equals;

import java.time.*;
import java.util.Objects;

public class Employee
{
   private String name;
   private double salary;
   private LocalDate hireDay;

   public Employee(String n, double s, int year, int month, int day)
   {
      name = n;
      salary = s;
      hireDay = LocalDate.of(year, month, day);
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public LocalDate getHireDay()
   {
      return hireDay;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public boolean equals(Object otherObject)
   {
      // Sprawdzenie, czy obiekty są identyczne
      if (this == otherObject) return true;

      // Musi zwrócić false, jeśli jawny parametr jest null
      if (otherObject == null) return false;

      // Jeśli klasy nie zgadzają się, nie mogą być jednakowe
      if (getClass() != otherObject.getClass()) return false;
```

```
        // Teraz wiadomo, że otherObject jest typu Employee i nie jest null
        var other = (Employee) otherObject;

        // Sprawdzenie, czy pola mają identyczne wartości
        return Objects.equals(name, other.name) && salary == other.salary &&
        ↪Objects.equals(hireDay, other.hireDay);
    }

    public int hashCode()
    {
        return Objects.hash(name, salary, hireDay);
    }

    public String toString()
    {
        return getClass().getName() + "[name=" + name + ",salary=" + salary +
        ↪",hireDay=" + hireDay
              + "]";
    }
}
```

Listing 5.6. equals/Manager.java

```
package equals;

public class Manager extends Employee
{
    private double bonus;

    public Manager(String n, double s, int year, int month, int day)
    {
        super(n, s, year, month, day);
        bonus = 0;
    }

    public double getSalary()
    {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

    public void setBonus(double b)
    {
        this.bonus = bonus;
    }

    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        var other = (Manager) otherObject;
        // Metoda super.equals określiła, że obiekty należą do tej samej klasy
        return bonus == other.bonus;
    }

    public int hashCode()
    {
        return super.hashCode() + 17 * new Double(bonus).hashCode();
```

```
    }

    public String toString()
    {
       return java.util.Objects.hash(super.hashCode(), bonus);
    }
}
```

`java.lang.Object` **1.0**

- `Class getClass()`

  Zwraca obiekt `Class` zawierający informacje dotyczące klasy obiektu. W dalszej części tego rozdziału dowiemy się, że w Javie istnieje zamknięta w klasie `Class` reprezentacja czasu wykonywania klas.

- `boolean equals(Object otherObject)`

  Porównuje dwa obiekty. Zwraca wartość `true`, jeśli oba obiekty wskazują ten sam obszar pamięci, lub `false` w przeciwnym przypadku. We własnych klasach powinno się tę metodę przesłaniać.

- `String toString()`

  Zwraca łańcuch reprezentujący wartość obiektu. We własnych klasach powinno się tę metodę przesłaniać.

`java.lang.Class` **1.0**

- `String getName()`

  Zwraca nazwę klasy.

- `Class getSuperClass()`

  Zwraca nadklasę danej klasy w postaci obiektu klasy `Class`.

5.3. Generyczne listy tablicowe

W niektórych językach programowania, zwłaszcza w C, rozmiary wszystkich tablic muszą być ustalone w czasie kompilacji. Nie podoba się to programistom, ponieważ zmusza ich to do niewygodnych kompromisów. Ilu pracowników będzie zatrudniał dany dział? Z pewnością nie więcej niż 100. Co zrobić, jeśli jeden dział będzie zatrudniać aż 150 pracowników? Czy dla każdego działu z tylko 10 pracownikami konieczne jest marnowanie 90 pozostałych miejsc?

W Javie sytuacja ta przedstawia się nieco lepiej. Rozmiar tablicy można ustawić w trakcie działania programu.

```
int actualSize = . . .;
var staff = new Employee[actualSize];
```

Oczywiście powyższy fragment kodu nie rozwiązuje w pełni problemu dynamicznej zmiany rozmiaru tablic w trakcie działania programu. Kiedy rozmiar tablicy jest ustalony, nie można go łatwo zmienić. W Javie najprostszy sposób na poradzenie sobie z taką często spotykaną sytuacją jest użycie klasy o nazwie `ArrayList`. Obiekty tej klasy są podobne do tablic, z tą różnicą, że automatycznie dostosowują swoje rozmiary w wyniku dodawania i odejmowania elementów. Nie wymaga to żadnych modyfikacji kodu.

`ArrayList` jest **klasą generyczną** z **parametrem typu**. Aby określić typ obiektów przechowywanych przez listę tablicową, należy podać nazwę klasy w nawiasach ostrych, np. `ArrayList<Employee>`. Sposób definiowania klas generycznych został opisany w rozdziale 8., choć znajomość tych technik nie jest konieczna, aby móc używać typu `ArrayList`.

W kolejnych punktach dowiesz się, jak posługiwać się listami tablicowymi.

5.3.1. Deklarowanie list tablicowych

Poniższy kod deklaruje i tworzy listę tablicową przechowującą obiekty klasy `Employee`:

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

Od Javy 10 dobrym pomysłem jest używanie słowa `var`, aby uniknąć powielania nazwy klasy:

```
var staff = new ArrayList<Employee>();
```

Jeśli nie używasz słowa kluczowego `var`, parametr po prawej stronie możesz opuścić.

```
ArrayList<Employee> staff = new ArrayList<>();
```

Jest to tzw. składnia diamentowa (ang. *diamond syntax*), nazwana tak ze względu na to, że pusty nawias <> przypomina kształtem diament. Należy ją stosować w połączeniu z operatorem `new`. Kompilator sprawdza, co się dzieje z nową wartością. Jeżeli zostaje przypisana do zmiennej, przekazana do metody lub zwrócona przez metodę, to kompilator sprawdza generyczny typ tej zmiennej lub tego parametru albo tej metody. Następnie wstawia ten typ w nawias <>. W przedstawionym przykładzie znajduje się przypisanie `new ArrayList()` do zmiennej typu `ArrayList<Employee>`, więc typ generyczny to `Employee`.

Jeśli deklarujesz listę `ArrayList` przy użyciu słowa kluczowego `var`, nie stosuj składni diamentowej. Deklaracja:

```
var elements = new ArrayList<>();
```

da w wyniku `ArrayList<Object>`.

Przed wersją 5.0 w Javie nie było klas generycznych. Zamiast tego była sama klasa `ArrayList`, która mogła przechowywać elementy typu `Object`. Nadal można używać klasy `ArrayList` bez <...>. Jest to tak zwany typ surowy (ang. *raw*), w którym usunięto parametr typu.

W jeszcze starszych wersjach Javy tablice dynamiczne tworzono za pomocą klasy `Vector`. Klasa `ArrayList` jest jednak bardziej efektywna i nie ma powodu do używania starej klasy `Vector`.

Nowe elementy do listy są dodawane za pomocą metody `add`. Poniższy fragment kodu zapełnia listę tablicową pracownikami:

```
staff.add(new Employee("Henryk Kwiatek", . . .));
staff.add(new Employee("Waldemar Kowalski", . . .));
```

Lista tablicowa zawiera wewnętrzną tablicę referencji do obiektów. Kiedy skończy się miejsce w tej tablicy, lista wykonuje swoje magiczne sztuczki. Jeśli wewnętrzna tablica jest pełna i zostanie wywołana metoda `add`, lista automatycznie utworzy większą tablicę i skopiuje do niej wszystkie obiekty.

Jeśli liczba elementów, które będą przechowywane w tablicy, jest znana, przynajmniej w przybliżeniu, przed zapełnieniem listy należy wywołać metodę `ensureCapacity`.

```
staff.ensureCapacity(100);
```

Powyższe wywołanie zarezerwuje miejsce dla wewnętrznej tablicy mogącej przechowywać 100 elementów. Dzięki temu 100 pierwszych wywołań metody `add` nie spowoduje czasochłonnej realokacji.

Początkową pojemność listy można także przekazać do konstruktora klasy `ArrayList`:

```
ArrayList<Employee> staff = new ArrayList<>(100);
```

Alokacja listy tablicowej:

```
new ArrayList<>(100)     // Pojemność 100
```

nie jest tym samym co alokacja nowej tablicy:

```
new Employee[100]        // Rozmiar 100
```

Pomiędzy pojemnością listy tablicowej a rozmiarem tablicy jest duża różnica. Tablica o rozmiarze 100 zawiera 100 miejsc, w których może przechowywać dane. Lista tablicowa o pojemności 100 ma *możliwość* przechowywania 100 elementów (a nawet więcej kosztem dodatkowej realokacji). Jednak na początku, nawet po jej utworzeniu, lista tablicowa nie zawiera żadnych elementów.

Metoda `size` sprawdza liczbę elementów w liście tablicowej. Na przykład wywołanie:

```
staff.size()
```

zwróci bieżącą liczbę elementów w liście tablicowej `staff`. To wywołanie jest odpowiednikiem wywołania:

```
a.length
```

dla tablicy `a`.

Kiedy jest już pewne, że rozmiar listy tablicowej się nie zmieni, można wywołać metodę `trimToSize`. Metoda ta dostosowuje rozmiar bloku pamięci przechowującego listę dokładnie do aktualnego rozmiaru listy. Nadmiar pamięci zostanie wyczyszczony przez system zbierania nieużytków.

Jeśli po dopasowaniu rozmiaru listy zostaną dodane do niej nowe elementy, blok ponownie zostanie przeniesiony, co zabiera czas. Metody `trimToSize` należy używać wyłącznie wtedy, gdy jest pewne, że do listy tablicowej nie będą dodawane już nowe elementy.

> **C++** Klasa `ArrayList` przypomina dostępny w C++ szablon `vector`. Jedno i drugie jest typem generycznym. Ale szablon `vector` przeciąża operator `[]`, dzięki czemu dostęp do elementów jest wygodniejszy. Ponieważ w Javie nie można przeciążać operatorów, trzeba używać do tego celu metod. Ponadto wektory w C++ są kopiowane przez wartość. Jeśli `a` i `b` są wektorami, przypisanie `a = b` tworzy nowy wektor `a` o takiej samej długości jak `b` i kopiuje wszystkie elementy z `b` do `a`. Takie samo przypisanie w Javie powoduje, że zarówno `a`, jak i `b` odwołują się do tej samej listy tablicowej.

`java.util.ArrayList<E>` **1.2**

- `ArrayList<E>()`

  Tworzy pustą listę tablicową.

- `ArrayList<E>(int initialCapacity)`

  Tworzy pustą listę tablicową o podanej pojemności.

- `boolean add(E obj)`

  Dodaje obiekt na końcu listy. Zawsze zwraca wartość `true`.

- `int size()`

  Zwraca liczbę elementów aktualnie przechowywanych w liście (wartość ta nie może być większa niż pojemność listy).

- `void ensureCapacity(int capacity)`

  Zapewnia, że lista będzie miała wystarczającą pojemność do przechowywania danej liczby elementów, bez potrzeby realokacji wewnętrznej tablicy.

- `void trimToSize()`

  Redukuje pojemność listy do jej aktualnego rozmiaru.

5.3.2. Dostęp do elementów listy tablicowej

Niestety nie ma nic za darmo. Ceną za wygodę związaną z automatycznym powiększaniem się listy tablicowej jest bardziej skomplikowany dostęp do jej elementów. Powód jest taki, że klasa `ArrayList` nie należy do języka Java — została utworzona przez jakiegoś programistę i dodana do standardowej biblioteki.

Zamiast składni z operatorem [], aby uzyskać dostęp do obiektów w celu ich odczytania lub modyfikacji, konieczne jest stosowanie metod `get` i `set`.

Aby na przykład ustawić wartość *i*-tego elementu, należy napisać:

```
staff.set(i, harry);
```

Powyższy zapis jest odpowiednikiem poniższego:

```
a[i] = harry;
```

dla tablicy `a` (tak samo jak w tablicach, numerowanie indeksów zaczyna się od zera).

Nie należy wywoływać `list.set(i, x)`, jeśli *rozmiar* tablicy nie jest większy od `i`. Na przykład poniższy kod jest błędny:

```
var list = new ArrayList<Employee>(100);      // Pojemność 100, rozmiar 0
list.set(0, x);                               // Nie ma jeszcze elementu 0
```

Do zapełniania tablicy używaj metody `add` zamiast `set`, którą należy stosować tylko w celu podmiany wcześniej dodanego elementu.

Aby pobrać wartość elementu listy tablicowej, należy napisać:

```
Employee e = staff.get(i);
```

Zapis ten jest odpowiednikiem poniższego:

```
Employee e = a[i];
```

Gdy nie było generycznych klas, metoda `get` surowej klasy `ArrayList` nie miała innego wyjścia, jak zwracać obiekty klasy `Object`. Z tego powodu wywołujący tę metodę musiał rzutować zwróconą wartość na odpowiedni typ:

```
Employee e = (Employee) staff.get(i);
```

Ponadto surowa klasa `ArrayList` nie jest w pełni bezpieczna. Jej metody `add` i `set` przyjmują obiekty każdego typu. Poniższe wywołanie:

```
staff.set(i, "Henryk Kwiatek");
```

w czasie kompilacji spowoduje tylko ostrzeżenie, a problemy zaczną się dopiero po uzyskaniu obiektu i próbie rzutowania go. Przy użyciu `ArrayList<Employee>` kompilator wykryje ten błąd.

Czasami możliwe jest wzięcie tego, co najlepsze, z tablic i list tablicowych — elastyczności i wygodnego dostępu do elementów. Trzeba zastosować następującą sztuczkę. Należy utworzyć listę tablicową i wstawić do niej wszystkie potrzebne elementy:

```
var list = new ArrayList<X>();
while (. . .)
{
   x = . . .;
   list.add(x);
}
```

Następnie wszystkie elementy należy skopiować do tablicy za pomocą metody `toArray`:

```
var a = new X[list.size()];
list.toArray(a);
```

Aby dodać element w środku listy, należy użyć metody `add` z parametrem określającym indeks:

```
int n = staff.size() / 2;
staff.add(n, e);
```

Elementy znajdujące się na pozycjach od `n` do góry są przesuwane, aby zrobić miejsce dla nowego elementu. Jeśli nowy rozmiar listy przekracza jej pojemność, następuje realokacja wewnętrznej tablicy.

Podobnie ze środka listy można usunąć dowolny element:

```
Employee e = staff.remove(n);
```

Elementy znajdujące się nad nim zostaną skopiowane w dół, a rozmiar tablicy zostanie zmniejszony o jeden.

Operacje wstawiania i usuwania elementów nie należą do najefektywniejszych. W przypadku małych list nie ma raczej czym się przejmować, ale jeśli elementów jest dużo i są one często wstawiane do środka kolekcji i z niej usuwane, należy rozważyć użycie listy dwukierunkowej (ang. *linked list*). Zagadnienia związane z listami dwukierunkowymi zostały poruszone w rozdziale 9.

Zawartość listy tablicowej można przemierzać za pomocą pętli typu `for each`:

```
for (Employee e : staff)
   działania związane z e
```

Ta pętla daje taki sam rezultat jak poniższa:

```
for (int i = 0; i < staff.size(); i++)
{
   Employee e = staff.get(i);
   działania związane z e
}
```

Listing 5.7 przedstawia zmodyfikowaną wersję programu *EmployeeTest* z rozdziału 4. Tablica `Employee[]` została zastąpiona listą tablicową `ArrayList<Employee>`. Należy zwrócić uwagę na następujące zmiany:

- Nie ma konieczności określenia rozmiaru tablicy.
- Za pomocą metody `add` można dodać dowolną liczbę elementów.
- Do sprawdzenia liczby elementów została użyta metoda `size()` zamiast metody `length`.
- Dostęp do elementu daje wywołanie `a.get(i)` zamiast `a[i]`.

Listing 5.7. arrayList/ArrayListTest.java

```
package arrayList;

import java.util.*;
```

```
/**
 * Ten program demonstruje użycie klasy ArrayList.
 * @version 1.11 2012-01-26
 * @author Cay Horstmann
 */
public class ArrayListTest
{
   public static void main(String[] args)
   {
      // Wstawienie do listy staff trzech obiektów klasy Employee.
      var staff = new ArrayList<Employee>();

      staff.add(new Employee("Karol Krakowski", 75000, 1987, 12, 15));
      staff.add(new Employee("Henryk Kwiatek", 50000, 1989, 10, 1));
      staff.add(new Employee("Waldemar Kowalski", 40000, 1990, 3, 15));

      // Zwiększenie pensji wszystkich pracowników o 5%.
      for (Employee e : staff)
         e.raiseSalary(5);

      // Drukowanie informacji o wszystkich obiektach Employee.
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary()
         ↪+ ",hireDay="
               + e.getHireDay());
   }
}
```

`java.util.ArrayList<E>` **1.2**

- `E set(int index, T obj)`

  Wstawia wartość `obj` do listy tablicowej w miejscu o określonym indeksie, zwraca poprzednią zawartość.

- `E get(int index)`

  Pobiera wartość zapisaną w określonym indeksie.

- `void add(int index, E obj)`

  Przesuwa elementy, aby dodać `obj` w określonym miejscu.

- `E remove(int index)`

  Usuwa element o określonym indeksie i przesuwa w dół wszystkie elementy, które znajdują się nad nim. Usunięty element jest zwracany.

5.3.3. Zgodność pomiędzy typowanymi a surowymi listami tablicowymi

Pisząc własny program, ze względów bezpieczeństwa należy zawsze używać parametrów typu. W tej części dowiesz się, jak korzystać ze starego kodu, w którym parametry te nie zostały użyte.

Mając poniższą starą klasę:

```
public class EmployeeDB
{
   public void update(ArrayList list) { ... }
   public ArrayList find(String query) { ... }
}
```

listę tablicową z typem można przekazać do metody `update` bez rzutowania.

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

Do metody `update` przekazywany jest obiekt `staff`.

Mimo że kompilator nie zgłasza żadnego błędu ani ostrzeżenia, to wywołanie nie jest w pełni bezpieczne. Metoda `update` może dodać do listy tablicowej elementy, które nie są typu `Employee`. Kiedy te elementy są pobierane, powstaje wyjątek. Mimo że brzmi to strasznie, działanie to jest dokładnie takie samo jak przed Java SE 5.0. Integralność maszyny wirtualnej nigdy nie jest zagrożona. W tej sytuacji nie zostaje naruszone bezpieczeństwo, ale nie ma też żadnych korzyści z testów przeprowadzanych w czasie kompilacji.

Jeśli natomiast obiekt surowej klasy `ArrayList` zostanie przypisany do typu z parametrem, zostanie wyświetlone ostrzeżenie.

```
ArrayList<Employee> result = employeeDB.find(query);    // Powoduje ostrzeżenie
```

Aby ostrzeżenie się pojawiło, należy podać kompilatorowi opcję `-Xlint:unchecked`.

Zastosowanie rzutowania nie powoduje zniknięcia ostrzeżenia.

```
ArrayList<Employee> result = (ArrayList<Employee>)
   employeeDB.find(query);    // Powoduje kolejne ostrzeżenie
```

Zostanie wyświetlone inne ostrzeżenie informujące, że rzutowanie może wprowadzać w błąd.

Jest to spowodowane niezbyt udanym ograniczeniem typów generycznych w Javie. Ze względów zgodności kompilator konwertuje wszystkie listy tablicowe z typem na surowe obiekty klasy `ArrayList` po uprzednim sprawdzeniu, że reguły dotyczące typów nie zostały złamane. W działającym programie wszystkie listy tablicowe są takie same — w maszynie wirtualnej nie ma parametrów określających typ. W związku z tym rzutowania `(ArrayList)` i `(ArrayList<Employee>)` powodują przeprowadzenie identycznych sprawdzeń w trakcie działania programu.

Niewiele można z tym zrobić. Pracując ze starszym kodem, należy przyglądać się ostrzeżeniom zgłaszanym przez kompilator i pocieszać się tym, że nie mają one wielkiego znaczenia.

Gdy już osiągniesz zadowalający rezultat, możesz oznaczyć rzutowaną zmienną adnotacją `@SuppressWarnings("unchecked")`:

```
@SuppressWarnings("unchecked") ArrayList<Employee> result =
   (ArrayList<Employee>) employeeDB.find(query); // Powoduje zgłoszenie kolejnego ostrzeżenia
```

5.4. Opakowania obiektów i automatyczne pakowanie

Od czasu do czasu konieczna jest konwersja typu podstawowego, jak `int`, na obiekt. Każdy typ podstawowy ma swój odpowiednik w postaci klasy. Na przykład typowi `int` odpowiada klasa `Integer`. Tego typu klasy często są nazywane klasami **opakowującymi** (ang. *wrapper*). Nazwy klas opakowujących są oczywiste: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` i `Boolean` (sześć pierwszych dziedziczy po wspólnej nadklasie `Number`). Klasy te są niezmienne, tzn. nie można zmienić opakowanej wartości po utworzeniu opakowania. Ponadto są finalne, co oznacza, że nie można tworzyć ich podklas.

Załóżmy, że potrzebujemy tablicy liczb całkowitych. Niestety parametr typu w nawiasach ostrych nie może określać typu podstawowego. Nie można utworzyć listy `ArrayList<int>`. W takiej sytuacji przydatna okazuje się klasa opakowująca. Listę obiektów klasy `Integer` można zadeklarować bez problemu.

```
var list = new ArrayList<Integer>();
```

Lista `ArrayList<Integer>` jest znacznie mniej wydajna niż tablica `int[]`, ponieważ każda wartość jest osobno zapakowana wewnątrz obiektu. Tego typu konstrukcji należy używać wyłącznie w przypadku małych kolekcji, kiedy wygoda programisty jest ważniejsza od wydajności.

Na szczęście istnieje technika umożliwiająca łatwe dodawanie i pobieranie elementów z tablicy. Wywołanie:

```
list.add(3);
```

jest automatycznie konwertowane na:

```
list.add(new Integer(3));
```

Tego typu konwersja nazywa się **automatycznym opakowywaniem** (ang. *autoboxing*).

Mogłoby się wydawać, że bardziej odpowiednim terminem byłoby **autowrapping**, ale człon **boxing** został przejęty z języka C#.

Jeśli natomiast obiekt klasy `Integer` zostanie przypisany do wartości `int`, zostanie automatycznie odpakowany. To znaczy kompilator przekonwertuje:

```
int n = list.get(i);
```

na:

```
int n = list.get(i).intValue();
```

Automatyczne opakowywanie i odpakowywanie działa nawet w przypadku operacji arytmetycznych. Można na przykład zastosować do referencji do obiektu opakowującego operator inkrementacji:

```
Integer n = 3;
n++;
```

Kompilator automatycznie wstawi instrukcje odpakowujące obiekt, zwiększające opakowaną w nim wartość i opakowujące ją z powrotem.

W większości przypadków wydaje się, że typy podstawowe i ich opakowania są jednym i tym samym. Jest między nimi tylko jedna znacząca różnica: tożsamość. Jak wiadomo, operator == zastosowany do obiektów opakowujących sprawdza tylko, czy obiekty te mają identyczne lokalizacje w pamięci. W związku z tym poniższe porównanie prawdopodobnie zakończyłoby się niepowodzeniem:

```
Integer a = 1000;
Integer b = 1000;
if (a == b) ...
```

Jednak implementacja Javy *może*, jeśli tak zdecyduje, opakować często pojawiające się wartości w identyczne obiekty i wtedy takie porównanie zakończyłoby się powodzeniem. Taka dwuznaczność nie jest pożądana. Rozwiązaniem problemu jest porównywanie obiektów opakowujących za pomocą metody `equals`.

Specyfikacja automatycznego opakowywania wymaga, aby typy `boolean`, `byte`, `char` o wartościach mniejszych bądź równych 127 oraz `short` i `int` w przedziale od –128 do 127 były opakowywane w ustalone obiekty. Jeśli na przykład w powyższym fragmencie kodu `a` i `b` zostałyby zainicjalizowane wartością 100, porównywanie musiałoby się zakończyć powodzeniem.

Nigdy nie polegaj na tożsamości obiektów opakowujących. Nie porównuj ich za pomocą operatora == i nie używaj ich jako blokad (rozdział 14.).

Nie używaj konstruktorów klas opakowujących. Są wycofywane i przeznaczone do usunięcia. Na przykład pisz `Integer.valueOf(1000)`, a nie `new Integer(1000)`. Ewentualnie polegaj po prostu na automatycznym opakowywaniu: `Integer a = 1000`.

Z automatycznym pakowaniem wiążą się jeszcze pewne inne subtelne kwestie. Przede wszystkim referencje do obiektów klasy opakowującej mogą mieć wartość `null`, więc operacja automatycznego odpakowywania może spowodować zgłoszenie wyjątku `NullPointerException`.

```
Integer n = null;
System.out.println(2 * n); // powoduje wyjątek NullPointerException
```

A jeśli zmiesza się typy `Integer` i `Double` w wyrażeniu warunkowym, to wartość `Integer` zostanie odpakowana, przekonwertowana na `double` i spakowana w `Double`:

```
Integer n = 1;
Double x = 2.0;
System.out.println(true ? n : x); // drukuje 1.0
```

Należy również zaznaczyć, że opakowywanie i odpakowywanie zawdzięczamy „uprzejmości" *kompilatora*, a nie maszyny wirtualnej. Kompilator wstawia odpowiednie wywołania, kiedy generuje kod bajtowy klasy. Rola maszyny wirtualnej sprowadza się tylko do wykonywania tego kodu.

W przyszłej wersji Javy programista będzie mógł definiować własne typy danych o takich samych cechach jak podstawowe typy danych, czyli nie przechowujące wartości w obiektach. Na przykład wartość podstawowego typu danych `Point` zawierającego pola `x` i `y` typu `double` jest w pamięci 16-bajtowym blokiem obejmującym dwie przylegające do siebie wartości typu `double`. Można go skopiować, ale nie można utworzyć referencji do niego.

Jeśli potrzebna jest referencja, należy użyć automatycznie wygenerowanej klasy towarzyszącej (zgodnie z obecnymi propozycjami miałaby nazwę `Point.ref`). Pakowanie i rozpakowywanie odbywa się automatycznie, tak samo jak w przypadku wszystkich typów podstawowych.

Kiedyś klasy opakowujące typów podstawowych zostaną zunifikowane z tymi klasami. Na przykład `Double` będzie aliasem `double.ref`.

Obiekty opakowujące typów liczbowych są także często używane do innego celu. Projektanci Javy odkryli, że obiekty opakowujące są dobrym miejscem na przechowywanie niektórych podstawowych metod, jak te, które służą do konwersji łańcuchów cyfr na liczby.

Aby przekonwertować łańcuch na liczbę, należy użyć następującej instrukcji:

```
int x = Integer.parseInt(s);
```

Kod ten nie ma nic wspólnego z obiektami klasy `Integer` — metoda `parseInt` jest statyczna. Jednak klasa `Integer` była dobrym miejscem na przechowywanie tej metody.

W opisie API znajdują się informacje o innych ważniejszych metodach klasy `Integer`. Pozostałe klasy odpowiadające typom liczbowym zawierają podobne metody.

Niektórzy programiści uważają, że klas opakowujących można używać do implementacji metod, które mogą modyfikować parametry liczbowe. Są jednak w błędzie. Pamiętamy z rozdziału 4., że w Javie nie można napisać metody zwiększającej parametr liczbowy, ponieważ parametry są zawsze przekazywane do metod przez wartość.

```
public static void triple(int x)        // nie zadziała
{
   x = 3 * x;                           // modyfikacja lokalnej zmiennej
}
```

Czy można to ominąć, stosując typ `Integer` zamiast `int`?

```
public static void triple(Integer x)    // nie zadziała
{
   ...
}
```

Problem polega na tym, że obiekty klasy `Integer` są **niezmienialne** — informacje zawarte w obiekcie opakowującym nie mogą być zmieniane. Nie można użyć tych klas opakowujących do tworzenia metod modyfikujących parametry liczbowe.

`java.lang.Integer` **1.0**

- `int intValue()`

  Zwraca wartość obiektu `Integer` jako liczbę typu `int` (przesłania metodę `intValue` z klasy `Number`).

- `static String toString(int i)`

  Zwraca nowy obiekt klasy `String` reprezentujący liczbę `i` w systemie dziesiętnym.

- `static String toString(int i, int radix)`

  Zwraca reprezentację liczby `i` w systemie określonym przez parametr `radix`.

- `Static int parseInt(String s)`
- `Static int parseInt(String s, int radix)`

  Zwraca liczbę całkowitą utworzoną z cyfr w łańcuchu `s`. Łańcuch ten musi reprezentować liczbę w systemie dziesiętnym (w przypadku pierwszej metody) lub w systemie określonym przez parametr `radix` (druga metoda).

- `static Integer valueOf(String s)`
- `static Integer valueOf(String s, int radix)`

  Zwraca obiekt klasy `Integer` zainicjalizowany liczbą całkowitą reprezentowaną przez łańcuch `s`. Łańcuch ten musi reprezentować liczbę w systemie dziesiętnym (w przypadku pierwszej metody) lub w systemie określonym przez parametr `radix` (druga metoda).

`java.text.NumberFormat` **1.1**

- `Number parse(String s)`

  Zwraca wartość liczbową, jeśli łańcuch `s` reprezentuje liczbę.

5.5. Metody ze zmienną liczbą parametrów

Programista może tworzyć metody, które da się wywoływać z różną liczbą parametrów (można spotkać ich angielską nazwę **varargs**).

Znamy już metodę `printf`. Na przykład wywołania:

```
System.out.printf("%d", n);
```

i

```
System.out.printf("%d %s", n, "widgets");
```

dotyczą tej samej metody, mimo że pierwsze z nich ma dwa parametry, a drugie trzy.

Definicja metody `printf` wygląda następująco:

```
public class PrintStream
{
   public PrintStream printf(String fmt, Object... args)
   {
     return format(fmt, args);
   }
}
```

W powyższym kodzie wielokropek (...) jest częścią kodu Javy. Określa on, że metoda może przyjmować dowolną liczbę obiektów (parametr `fmt` jest obowiązkowy).

Metoda `printf` w rzeczywistości przyjmuje dwa parametry — łańcuch formatu i tablicę `Object[]`, która zawiera wszystkie pozostałe parametry (jeśli zostanie podana wartość typu podstawowego, jak `int`, mechanizm automatycznego opakowywania zamieni ją na obiekt). Musi ona przeskanować łańcuch `fmt` i dopasować *i*-ty specyfikator formatu do wartości `args[i]`.

Innymi słowy, z punktu widzenia programisty implementującego metodę `printf` typ parametru `Object...` jest dokładnie tym samym co `Object[]`.

Kompilator musi przekonwertować każde wywołanie metody `printf`, pakując parametry do tablicy i wykonując w razie potrzeby automatyczne opakowywanie:

```
System.out.printf("%d %s", new Object[] { Integer.valueOf(n), "widgets" } );
```

Można definiować własne metody ze zmienną liczbą parametrów. Parametry te mogą być każdego typu, także podstawowego. Poniżej znajduje się prosty przykład takiej funkcji — zwraca największą liczbę w zbiorze o zmiennych rozmiarach:

```
public static double max(double... values)
{
   double largest = Double.NEGATIVE_INFINITY;
   for (double v : values) if (v > largest) largest = v;
   return largest;
}
```

Należy ją wywołać w następujący sposób:

```
double m = max(3.1, 40.4, -5);
```

Kompilator przekazuje tablicę `new double[] {3.1, 40.4, -5}` do funkcji `max`.

Ostatnim parametrem metody o zmiennej liczbie parametrów może być tablica. Na przykład:

```
System.out.printf("%d %s", new Object[] { Integer.valueOf(1), "widgets" } );
```

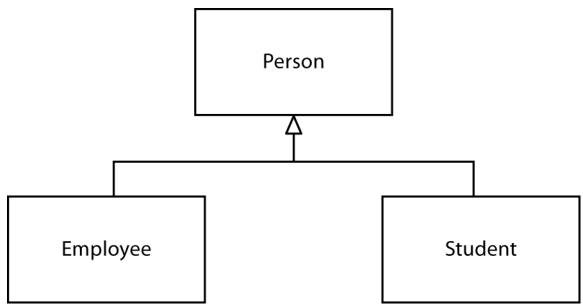
W związku z tym można przedefiniować istniejącą funkcję, której ostatni parametr jest tablicą, na metodę ze zmienną liczbą parametrów, nie uszkadzając istniejącego kodu. Na przykład w ten sposób rozszerzono metodę `MessageFormat.format` w Java SE 5.0. Można nawet zadeklarować metodę `main` w następujący sposób:

```
public static void main(String... args)
```

5.6. Klasy abstrakcyjne

Im bliżej wierzchołka hierarchii dziedziczenia, tym klasy są bardziej ogólne i często bardziej abstrakcyjne. W pewnym momencie klasa nadrzędna staje się tak abstrakcyjna, że zaczyna być traktowana nie jako klasa do tworzenia obiektów o określonym przeznaczeniu, a jako podstawa do tworzenia innych klas. Przeanalizujmy na przykład rozszerzenie hierarchii klasy `Employee`. Pracownik (ang. *employee*), podobnie jak student, jest osobą. Poszerzymy naszą hierarchię klas o klasy `Person` (osoba) i `Student`. Rysunek 5.2 przedstawia relacje dziedziczenia zachodzące pomiędzy tymi klasami.

Rysunek 5.2. Diagram dziedziczenia klasy Person i jej podklas



Po co w ogóle zaprzątać sobie głowę takim poziomem abstrakcji? Niektóre cechy ma każda osoba, np. nazwisko. Zarówno studenci, jak i pracownicy mają nazwiska, a więc wprowadzenie wspólnej nadklasy umożliwia wydzielenie metody `getName` i przeniesienie jej na wyższy poziom hierarchii dziedziczenia.

Dodamy teraz nową metodę o nazwie `getDescription`, która zwraca krótki opis osoby, np.:

```
pracownik zarabiający 50 000,00 zł
student specjalizacji informatyka
```

Implementacja tej metody dla klas `Employee` i `Student` jest łatwa. Jakie natomiast informacje można podać w klasie `Person`? Klasa ta ma jedynie informacje na temat nazwiska osoby. Oczywiście można zaimplementować metodę `Person.getDescription()`, która zwraca pusty łańcuch.

Istnieje jednak lepsze rozwiązanie. Dzięki użyciu słowa kluczowego `abstract` w ogóle nie ma potrzeby implementowania tej metody.

```
public abstract String getDescription();
   // nie jest potrzebna żadna implementacja
```

Klasa zawierająca przynajmniej jedną metodę abstrakcyjną sama musi być abstrakcyjna.

```
public abstract class Person
{ . . .
   public abstract String getDescription();
}
```

Poza metodami abstrakcyjnymi klasy abstrakcyjne mogą zawierać pola i metody konkretne. Na przykład klasa `Person` przechowuje nazwisko osoby i ma metodę konkretną, która zwraca te dane.

```
public abstract class Person
{
   private String name;

   public Person(String name)
   {
      this.name = name;
   }
   public abstract String getDescription();
   public String getName()
   {
     return name;
   }
}
```

Niektórzy programiści nie wiedzą, że klasy abstrakcyjne mogą zawierać metody konkretne. Należy zawsze przenosić wspólne pola i metody (bez względu na to, czy są abstrakcyjne, czy nie) do nadklasy (abstrakcyjnej lub nie).

Metody abstrakcyjne pełnią rolę symbolu zastępczego dla metod, które są implementowane w podklasach. Przy rozszerzaniu klasy abstrakcyjnej programista ma do wyboru jedną z dwóch opcji: może pozostawić niezdefiniowane niektóre lub wszystkie metody nadklasy — wtedy podklasa również musi być abstrakcyjna, albo zdefiniować wszystkie metody i wtedy podklasa nie jest abstrakcyjna.

Jako przykład zdefiniujemy klasę `Student`, która będzie rozszerzała abstrakcyjną klasę `Person` i implementowała metodę `getDescription`. Ponieważ żadna z metod klasy `Student` nie jest abstrakcyjna, klasa ta również nie musi być abstrakcyjna.

Klasę można zdefiniować jako abstrakcyjną, nawet jeśli nie zawiera żadnych metod abstrakcyjnych.

Nie można tworzyć obiektów klas abstrakcyjnych. To znaczy, że jeśli klasa ma w deklaracji słowo `abstract`, nie może mieć obiektów. Na przykład poniższe wyrażenie:

```
new Person("Wincenty Witos")
```

jest błędne. Można natomiast tworzyć obiekty podklas konkretnych.

Nadal można tworzyć zmienne obiektów klas abstrakcyjnych, ale takie zmienne muszą się odnosić do obiektu podklasy nieabstrakcyjnej. Na przykład:

```
Person p = new Student("Vince Vu", "Economics");
```

Tutaj zmienna p jest typu abstrakcyjnego Person, ale odnosi się do egzemplarza nieabstrakcyjnej podklasy Student.

W C++ metoda abstrakcyjna nazywa się **funkcją czysto wirtualną** i jest oznaczana końcowymi znakami =0:

```
class Person   // C++
{
public:
   virtual string getDescription() = 0;
   . . .
};
```

W C++ klasa jest abstrakcyjna, jeśli zawiera co najmniej jedną funkcję czysto wirtualną. Nie ma w tym języku specjalnego słowa kluczowego określającego klasę abstrakcyjną.

Zdefiniujemy konkretną podklasę Student, która rozszerza abstrakcyjną klasę Person:

```
public class Student extends Person
{
   private String major;

   public Student(String name, String major)
   {
      super(name);
      this.major = major;
   }
   public String getDescription()
   {
      return "student specjalizacji " + major;
   }
}
```

Klasa Student zawiera definicję metody getDescription. W związku z tym wszystkie metody tej klasy są konkretne, czyli nie jest ona abstrakcyjna.

Program przedstawiony na listingach 5.8, 5.9, 5.10 i 5.11 definiuje abstrakcyjną nadklasę o nazwie Person i dwie konkretne podklasy o nazwach Employee i Student. Do tablicy referencji typu Person wstawiane są obiekty klas Employee i Student:

```
var people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

Listing 5.8. abstractClasses/PersonTest.java

```
package abstractClasses;

/**
 * Ten program demonstruje klasy abstrakcyjne.
 * @version 1.01 2004-02-21
 * @author Cay Horstmann
 */
public class PersonTest
{
   public static void main(String[] args)
   {
      var people = new Person[2];

      // Wstawienie do tablicy people obiektów Student i Employee.
      people[0] = new Employee("Henryk Kwiatek", 50000, 1989, 10, 1);
      people[1] = new Student("Maria Mrozowska", "informatyka");

      // Drukowanie imion i nazwisk oraz opisów wszystkich obiektów klasy Person.
      for (Person p : people)
         System.out.println(p.getName() + ", " + p.getDescription());
   }
}
```

Listing 5.9. abstractClasses/Person.java

```
package abstractClasses;

public abstract class Person
{
   public abstract String getDescription();
   private String name;

   public Person(String name)
   {
      this.name = name;
   }

   public String getName()
   {
      return name;
   }
}
```

Listing 5.10. abstractClasses/Employee.java

```
package abstractClasses;

import java.time.*;

public class Employee extends Person
{
   private double salary;
   private LocalDate hireDay;
```

```
   public Employee(String n, double s, int year, int month, int day)
   {
      super(name);
      this.salary = salary;
      hireDay = LocalDate.of(year, month, day);
   }

   public double getSalary()
   {
      return salary;
   }

   public Date getHireDay()
   {
      return hireDay;
   }

   public String getDescription()
   {
      return String.format("pracownik zarabiający  %.2f zł", salary);
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }
}
```

Listing 5.11. abstractClasses/Student.java

```
package abstractClasses;

public class Student extends Person
{
   private String major;

   /**
    * @param n imię i nazwisko studenta
    * @param m specjalizacja studenta
    */
   public Student(String name, String major)
   {
      // Przekazanie name do konstruktora nadklasy.
      super(name);
      this.major = major;
   }

   public String getDescription()
   {
      return "student specjalizacji " + major;
   }
}
```

Poniższy fragment kodu odpowiada za wydruk imion i nazwisk oraz opisów powyższych obiektów:

```
for (Person p : people)
   System.out.println(p.getName() + ", " + p.getDescription());
```

Wątpliwości może budzić poniższe wywołanie:

```
p.getDescription()
```

Czy nie jest to wywołanie niezdefiniowanej metody? Należy pamiętać, że zmienna `p` nie odwołuje się nigdy do obiektu `Person`, ponieważ nie można utworzyć obiektu klasy abstrakcyjnej `Person`. Zmienna `p` zawsze odwołuje się do obiektu konkretnej podklasy, jak `Employee` lub `Student`. Dla tych obiektów metoda `getDescription` jest zdefiniowana.

Czy można by było pominąć abstrakcyjną metodę nadklasy abstrakcyjnej `Person` i zdefiniować metody `getDescription` w podklasach `Employee` i `Student`? Gdybyśmy tak zrobili, niemożliwe byłoby wywołanie metody `getDescription` na rzecz zmiennej `p`, ponieważ kompilator zezwala na wywoływanie tylko tych metod, które są zdefiniowane w danej klasie.

Metody abstrakcyjne są bardzo ważnym elementem języka programowania Java. Najczęściej występują w *interfejsach*. Więcej informacji na temat interfejsów zawiera rozdział 6.

5.7. Klasy wyliczeniowe

W rozdziale 3. nauczyliśmy się definiować typy wyliczeniowe. Oto typowy przykład:

```
public enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE}
```

Typ zdefiniowany przez powyższą deklarację jest w rzeczywistości klasą. Ma ona dokładnie cztery egzemplarze — nie można tworzyć jej nowych obiektów.

W związku z tym do porównywania typów wyliczeniowych nie trzeba używać metody `equals`. Wystarczy operator `==`.

Do typu wyliczeniowego można dodać konstruktory, metody i pola. Oczywiście konstruktory są wywoływane tylko wówczas, gdy są konstruowane stałe wyliczeniowe. Na przykład:

```
public enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private String abbreviation;

   Size(String abbreviation) { this.abbreviation = abbreviation; }
      // prywatna automatycznie
   public String getAbbreviation() { return abbreviation; }
}
```

Konstruktor wyliczenia zawsze jest prywatny. Modyfikator `private` można opuścić, jak pokazano w poprzednim przykładzie. Zadeklarowanie konstruktora wyliczenia jako publicznego lub chronionego stanowi błąd składni.

Wszystkie typy wyliczeniowe są podklasami abstrakcyjnej klasy `Enum`. Dziedziczą po niej kilka metod. Do najbardziej przydatnych należy metoda `toString`, która zwraca nazwę stałej wyliczeniowej. Na przykład wywołanie `Size.SMALL.ToString()` zwraca łańcuch `"SMALL"`.

Przeciwieństwem metody `toString` jest statyczna metoda `valueOf`. Na przykład poniższa instrukcja ustawia `s` na `Size.SMALL`.

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

Każdy typ wyliczeniowy ma statyczną metodę `values`, która zwraca wszystkie wartości wyliczenia. Na przykład wywołanie:

```
Size[] values = Size.values();
```

zwraca tablicę zawierającą następujące elementy: `Size.SMALL`, `Size.MEDIUM`, `Size.LARGE` oraz `Size.EXTRA_LARGE`.

Metoda `ordinal` zwraca położenie stałej wyliczeniowej w deklaracji `enum`, zaczynając liczenie od zera. Na przykład wywołanie `Size.MEDIUM.ordinal()` zwraca wartość 1.

Krótki program przedstawiony na listingu 5.12 demonstruje zastosowanie typów wyliczeniowych.

Klasa `Enum` ma parametr typu, który dla uproszczenia pominęliśmy. Na przykład typ wyliczeniowy `Size` w rzeczywistości rozszerza `Enum<Size>`. Parametr typu jest używany przez metodę `compareTo` (metodę `compareTo` opisujemy w rozdziale 6., a parametry typu w rozdziale 8.).

Listing 5.12. enums/EnumTest.java

```
package enums;

import java.util.*;

/**
 * Ten program demonstruje typy wyliczeniowe.
 * @version 1.0 2004-05-24
 * @author Cay Horstmann
 */
public class EnumTest
{
   public static void main(String[] args)
   {
      var in = new Scanner(System.in);
      System.out.print("Podaj rozmiar: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
      String input = in.next().toUpperCase();
      Size size = Enum.valueOf(Size.class, input);
      System.out.println("rozmiar=" + size);
      System.out.println("skrót=" + size.getAbbreviation());
      if (size == Size.EXTRA_LARGE)
         System.out.println("Dobra robota -- nie pominąłeś znaku podkreślenia _.");
   }
}

enum Size
{
   SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

   private Size(String abbreviation) { this.abbreviation = abbreviation; }
```

```
    public String getAbbreviation() { return abbreviation; }

    private String abbreviation;
}
```

`java.lang.Enum<E>` **5**

- `static Enum valueOf(Class enumClass, String name)`

  Zwraca stałą wyliczeniową danej klasy o podanej nazwie.

- `String toString()`

  Zwraca nazwę stałej wyliczeniowej.

- `int ordinal()`

  Zwraca położenie w deklaracji `enum` (licząc od zera) stałej wyliczeniowej.

- `int compareTo(E other)`

  Zwraca ujemną liczbę całkowitą, jeśli stała wyliczeniowa występuje przed `other`, zero — jeśli `this == other`, lub liczbę dodatnią w przeciwnym przypadku. Kolejność stałych jest określana przez deklarację `enum`.

5.8. Klasy zapieczętowane

Jeśli klasa nie jest zadeklarowana jako finalna, to każdy może utworzyć jej podklasę. Czy można mieć nad tym bardziej szczegółową kontrolę? Powiedzmy na przykład, że chcemy napisać własną bibliotekę JSON, ponieważ żadna z istniejących nie odpowiada w pełni naszym potrzebom.

W standardzie JSON znajduje się zapis informujący, że wartością JSON jest tablica, liczba, łańcuch, wartość logiczna, obiekt lub null. Oczywistym wyborem do reprezentowania tych wartości są klasy `JSONArray`, `JSONNumber` itd., które rozszerzają klasę `JSONValue`:

```
public abstract class JSONValue
{
    // Metody dotyczące wszystkich wartości JSON
}
public final class JSONArray extends JSONValue
{
    . . .
}
public final class JSONNumber extends JSONValue
{
    . . .
}
```

Przez dodanie słowa `final` do deklaracji klas `JSONArray`, `JSONNumber` itd. uniemożliwiamy komukolwiek utworzenie ich podklas. Nie możemy natomiast zabronić utworzenia kolejnej podklasy klasy `JSONValue`.

Do czego mogłaby być nam potrzebna taka możliwość? Spójrz na poniższy kod:

```
JSONValue v = . . .;
if (v instanceof JSONArray a) . . .
else if (v instanceof JSONNumber n) . . .
else if (v instanceof JSONString s) . . .
else if (v instanceof JSOBoolean b) . . .
else if (v instanceof JSONObject o) . . .
else . . . // Musi być JSONNull
```

W tym przypadku przepływ sterowania implikuje, że znamy wszystkie bezpośrednie podklasy klasy `JSONValue`. To nie jest otwarta hierarchia. Standard JSON się nie zmieni, a jeśli tak, to my, jako implementatorzy biblioteki, dodamy siódmą podklasę. Nie chcemy, aby ktokolwiek inny mieszał w tej hierarchii klas.

Mechanizmem pozwalającym kontrolować dziedziczenie w Javie jest **pieczętowanie** klas. Technika ta została wprowadzona na próbę w wersji 15 i ostatecznie zaimplementowana w wersji 17 języka.

Poniżej znajduje się przykład zapieczętowania klasy `JSONValue`:

```
public abstract sealed class JSONValue
    permits JSONArray, JSONNumber, JSONString, JSONBoolean, JSONObject, JSONNull
{
    . . .
}
```

Próba zdefiniowania klasy spoza dozwolonej listy jest błędem:

```
public class JSONComment extends JSONValue { . . . } // Błąd
```

W tym przypadku to bardzo dobrze, bo standard JSON nie przewiduje komentarzy. Zatem klasy zapieczętowane umożliwiają precyzyjne modelowanie ograniczeń dziedziny.

Dopuszczalne podklasy klasy zapieczętowanej muszą być dostępne. Nie mogą być prywatnymi konstrukcjami zagnieżdżonymi w innej klasie ani jednostkami widocznymi tylko w obrębie innego pakietu.

Reguły dotyczące dozwolonych podklas publicznych są jeszcze surowsze. Te klasy muszą się znajdować w tym samym pakiecie co klasa zapieczętowana. Jeśli jednak w użyciu są moduły (rozdział 9. tomu drugiego), to wystarczy, aby taka klasa znajdowała się w tym samym module.

Klasę zapieczętowaną można zadeklarować bez klauzuli `permits`. W takim przypadku wszystkie jej bezpośrednie podklasy muszą być zadeklarowane w tym samym pliku. Programiście nie mający do niego dostępu nie mogą tworzyć podklas.

W pliku może się znajdować tylko jedna klasa publiczna, a więc zasada ta wydaje się dotyczyć wyłącznie sytuacji, w której podklasy nie są przeznaczone do publicznego użytku.

Z drugiej strony w następnym rozdziale poznasz techniki używania klas wewnętrznych jako publicznych podklas.

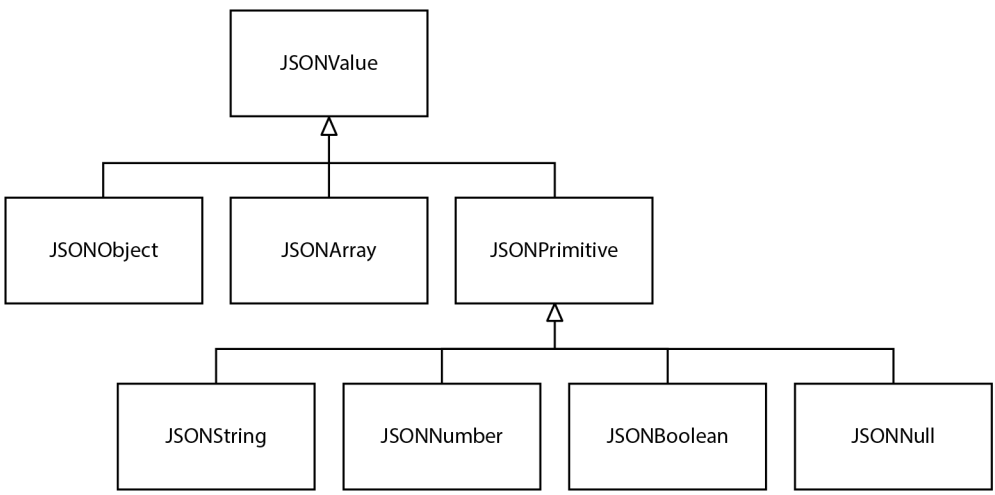
Ważnym argumentem przemawiającym za klasami zapieczętowanymi jest możliwość przeprowadzania kontroli przez kompilator. Spójrz na poniższą metodę z klasy `JSONValue`, która zawiera wyrażenie `switch` z dopasowywaniem wzorca (próbny dodatek do Javy 17):

```
public String type()
{
    return switch (this)
    {
        case JSONArray j -> "array";
        case JSONNumber j -> "number";
        case JSONString j -> "string";
        case JSONBoolean j -> "boolean";
        case JSONObject j -> "object";
        case JSONNull j -> "null";
        // Klauzula domyślna jest niepotrzebna
    };
}
```

Skoro wszystkie bezpośrednie podklasy klasy `JSONValue` zostały ujęte w klauzulach `case`, kompilator będzie „wiedział", że klauzula `default` jest niepotrzebna.

Powyższa metoda `type` nie wygląda na konstrukcję w technice obiektowej. Zgodnie z założeniami tej metodyki należałoby raczej utworzyć osobną metodę `type` w każdej z sześciu klas, z wykorzystaniem polimorfizmu, a nie konstrukcji `switch` — takie podejście dobrze się sprawdza w przypadku hierarchii otwartych. Kiedy natomiast zestaw klas jest określony na sztywno, często wygodniej jest zdefiniować wszystkie alternatywne opcje w jednej metodzie.

Na pierwszy rzut oka można odnieść wrażenie, że podklasa klasy zapieczętowanej musi być finalna. Jednak na potrzeby testów kompletności musimy znać tylko wszystkie bezpośrednie podklasy. Nie jest problemem, jeśli mają one jakieś dalsze podklasy. Naszą hierarchię klas JSON możemy na przykład przeorganizować w sposób pokazany na rysunku 5.3.



Rysunek 5.3. Kompletna hierarchia klas reprezentujących wartości JSON

W tej hierarchii klasa `JSONValue` może mieć trzy podklasy:

```
public abstract sealed class JSONValue permits JSONObject, JSONArray, JSONPrimitive
{
    . . .
}
```

Klasa `JSONPrimitive` też jest zapieczętowana:

```
public abstract sealed class JSONPrimitive extends JSONValue
permits JSONString, JSONNumber, JSONBoolean, JSONNull
{
    . . .
}
```

Podklasa zapieczętowanej klasy musi określać, czy jest zapieczętowana, finalna czy otwarta na dalsze tworzenie podklas. W drugim z wymienionych przypadków konieczna jest deklaracja, że klasa nie jest zapieczętowana (ang. `non-sealed`).

Słowo kluczowe `non-sealed` to pierwsze słowo kluczowe z łącznikiem w języku Java. Może zwiastować początek przyszłego trendu. Dodawanie słów kluczowych do języka programowania zawsze jest ryzykowne, ponieważ może spowodować trudności z kompilacją istniejącego kodu. Dlatego też `sealed` jest słowem „kontekstowym", tzn. nadal można go używać do nazywania zmiennych i metod:

```
int sealed = 1; // Można użyć kontekstowego słowa kluczowego jako identyfikatora
```

W przypadku gdy zostanie użyty łącznik, nie trzeba się o to martwić. Jedyna dwuznaczność może dotyczyć operacji odejmowania:

```
int non = 0;
non = non-sealed; // Odejmowanie, nie słowo kluczowe
```

Po co w ogóle ktoś miałby tworzyć niezapieczętowaną podklasę? Wyobraź sobie klasę węzła XML, która ma sześć bezpośrednich podklas:

```
public abstract sealed class Node permits Element, Text, Comment,
CDATASection, EntityReference, ProcessingInstruction
{
    . . .
}
```

Zdejmujemy ograniczenie dotyczące podklas z klasy `Element`:

```
public non-sealed class Element extends Node
{
    . . .
}
public class HTMLDivElement extends Element
{
    . . .
}
```

W tym podrozdziale opisałem klasy zapieczętowane, a w następnym rozdziale poznasz *interfejsy*, które są uogólnieniem klas abstrakcyjnych. W Javie interfejsy także mogą mieć podtypy.

Zapieczętowane interfejsy mają dokładnie takie same cechy jak zapieczętowane klasy w zakresie kontroli nad tworzeniem bezpośrednich podtypów.

Przykładowy program przedstawiony na listingu 5.13 ilustruje implementację hierarchii klas reprezentujących typy JSON. W implementacji klasy `JSONObject` została wykorzystana klasa `HashMap`, której opis znajduje się w rozdziale 9. W przykładzie użyto interfejsów zamiast klas abstrakcyjnych, dzięki czemu klasy `JSONNumber` i `JSONString` mogą być rekordami, a klasy `JSON↪Boolean` i `JSONNull` mogą być wyliczeniami. Rekordy i wyliczenia mogą implementować interfejsy, ale nie mogą rozszerzać klas.

Listing 5.13. sealed/SealedTest.java

```
package sealed;

import java.util.*;

sealed interface JSONValue permits JSONArray, JSONObject, JSONPrimitive
{
    public default String type()
    {
        if (this instanceof JSONArray) return "array";
        else if (this instanceof JSONObject) return "object";
        else if (this instanceof JSONNumber) return "number";
        else if (this instanceof JSONString) return "string";
        else if (this instanceof JSONBoolean) return "boolean";
        else return "null";
    }
}

final class JSONArray extends ArrayList<JSONValue> implements JSONValue {}

final class JSONObject extends HashMap<String, JSONValue> implements JSONValue
{
    public String toString()
    {
        StringBuilder result = new StringBuilder();
        result.append("{");
        for (Map.Entry<String, JSONValue> entry : entrySet())
        {
            if (result.length() > 1) result.append(",");
            result.append(" \"");
            result.append(entry.getKey());
            result.append("\": ");
            result.append(entry.getValue());
        }
        result.append(" }");
        return result.toString();
    }
}

sealed interface JSONPrimitive extends JSONValue
      permits JSONNumber, JSONString, JSONBoolean, JSONNull
{
}

final record JSONNumber(double value) implements JSONPrimitive
```

```
{
    public String toString() { return "" + value; }
}

final record JSONString(String value) implements JSONPrimitive
{
    public String toString() { return "\"" + value.translateEscapes() + "\""; }
}

enum JSONBoolean implements JSONPrimitive
{
    FALSE, TRUE;
    public String toString() { return super.toString().toLowerCase();}
}

enum JSONNull implements JSONPrimitive
{
    INSTANCE;
    public String toString() { return "null"; }
}

public class SealedTest
{
    public static void main(String[] args)
    {
        JSONObject obj = new JSONObject();
        obj.put("name", new JSONString("Harry"));
        obj.put("salary", new JSONNumber(90000));
        obj.put("married", JSONBoolean.FALSE);
        JSONArray arr = new JSONArray();
        arr.add(new JSONNumber(13));
        arr.add(JSONNull.INSTANCE);

        obj.put("luckyNumbers", arr);
        System.out.println(obj);
        System.out.println(obj.type());
    }
}
```

5.9. Refleksja

Biblioteka refleksyjna dostarcza bogaty i zaawansowany zestaw narzędzi do pisania programów, które dynamicznie zarządzają kodem Javy. Z mechanizmów refleksji w Javie korzystają kreatory interfejsów, mapery obiektowo-relacyjne i inne narzędzia programistyczne, które dynamicznie sprawdzają właściwości klas.

Program, który może analizować funkcjonalność klas, nazywa się **programem refleksyjnym**. Mechanizm refleksji ma bardzo duże możliwości. W kolejnych podrozdziałach opisujemy jego następujące zastosowania:

- Analiza właściwości klasy w trakcie działania programu.
- Inspekcja obiektów w czasie działania programu, na przykład do napisania jednej metody `toString`, która działa we *wszystkich* klasach.
- Implementacja generycznego kodu manipulującego tablicami.
- Wykorzystanie obiektów `Method`, które działają tak jak wskaźniki do funkcji w innych językach, np. C++.

Refleksja to wszechstronny i skomplikowany mechanizm. Jest interesująca przede wszystkim dla twórców narzędzi, mniej dla programistów aplikacji. Osoby, które są zainteresowane pisaniem aplikacji, a nie narzędzi dla innych programistów Javy, mogą pominąć resztę rozdziału i wrócić do niego kiedy indziej.

5.9.1. Klasa Class

Kiedy uruchomiony jest program, system wykonawczy Javy cały czas przechowuje informacje o typach wszystkich obiektów. Do informacji tych zaliczają się nazwy klas, do których należą obiekty. Informacje o typach czasu wykonywania programu są wykorzystywane przez maszynę wirtualną do wyboru odpowiednich metod do wykonania.

Do informacji tych można jednak uzyskać dostęp także dzięki specjalnej klasie. Klasa przechowująca te informacje ma nazwę `Class`. Metoda `getClass()` klasy `Object` zwraca egzemplarz klasy `Class`.

```
Employee e;
. . .
Class cl = e.getClass();
```

Podobnie jak obiekt klasy `Employee` opisuje cechy określonego pracownika, obiekt klasy `Class` opisuje cechy określonej klasy. Chyba najczęściej używaną metodą klasy `Class` jest metoda `getName`, która zwraca nazwę klasy. Na przykład poniższa instrukcja:

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

drukuje:

```
Employee Henryk Kwiatek
```

jeśli e jest zwykłym pracownikiem, lub

```
Manager Henryk Kwiatek
```

jeśli e jest kierownikiem.

Jeśli klasa należy do jakiegoś pakietu, nazwa tego pakietu stanowi część nazwy tej klasy:

```
var generator = new Random();
Class cl = generator.getClass();
String name = cl.getName(); // name ustawiono na "java.util.Random"
```

Obiekt klasy `Class` odpowiadający nazwie wybranej klasy można utworzyć za pomocą statycznej metody `forName`.

```
String className = "java.util.Date";
Class cl = Class.forName(className);
```

Metody tej należy użyć, jeśli nazwa klasy jest przechowywana w łańcuchu, który zmienia się w czasie działania programu. Powyższy kod działa, jeśli `className` jest nazwą klasy lub interfejsu. W przeciwnym przypadku metoda `forName` powoduje **wyjątek kontrolowany** (ang. *checked exception*). Informacje na temat **obsługi wyjątków** podczas używania tej metody znajdują się w punkcie 5.9.2, „Podstawy deklarowania wyjątków".

Trzecia metoda tworzenia obiektu typu `Class` jest wygodnym skrótem. Jeśli `T` jest dowolnym typem Javy (lub słowem kluczowym `void`), `T.class` jest odpowiadającym mu obiektem klasy `Class`. Na przykład:

```
Class cl1 = Random.class;     // Należy zaimportować java.util.*;.
Class cl2 = int.class;
Class cl3 = Double[].class;
```

Należy zauważyć, że obiekt klasy `Class` w rzeczywistości oznacza *typ*, który może, ale nie musi być klasą. Na przykład `int` nie jest klasą, ale `int.class` jest z pewnością obiektem typu `Class`.

Klasa `Class` jest sparametryzowana. Na przykład `Employee.class` jest typu `Class<Employee>`. Nie będziemy drążyć tego tematu, ponieważ jeszcze bardziej skomplikowalibyśmy i tak już wystarczająco abstrakcyjną koncepcję. Dla praktycznych celów można zignorować parametr typu i pracować na surowym typie `Class`. Więcej informacji na ten temat znajduje się w rozdziale 8.

Z powodów historycznych metoda `getName` zwraca nieco dziwne nazwy typów tablicowych:

- `Double[].class.getName()` zwraca `[Ljava.lang.Double;`,
- `int[].class.getName()` zwraca `[I`.

Maszyna wirtualna obsługuje unikatowy obiekt `Class` dla każdego typu. W związku z tym do porównywania obiektów `class` można używać operatora `==`. Na przykład:

```
if (e.getClass() == Employee.class) . . .
```

Wynik tego testu będzie pozytywny, gdy e będzie egzemplarzem klasy `Employee`. W odróżnieniu od warunku `e instanceof Employee`, w tym przypadku wynik będzie negatywny, jeśli e będzie egzemplarzem podklasy, np. `Manager`.

Jeśli masz obiekt typu `Class`, przy jego użyciu możesz tworzyć egzemplarze tej klasy. Wywołaj metodę `getConstructor`, aby otrzymać obiekt typu `Constructor`, a następnie za pomocą metody `newInstance` utwórz nowy egzemplarz. Na przykład:

```
var className = "java.util.Random"; // lub nazwa jakiejkolwiek innej klasy
                                    // z konstruktorem bezargumentowym
Class cl = Class.forName(className);
Object obj = cl.getConstructor().newInstance();
```

Jeżeli klasa nie ma konstruktora bezargumentowego, metoda `getConstructor` zgłasza wyjątek. W punkcie 5.9.7, „Wywoływanie dowolnych metod i konstruktorów”, dowiesz się, jak wywoływać inne konstruktory.

Istnieje wycofywana z użycia metoda `Class.toInstance`, która także tworzy egzemplarz przy użyciu konstruktora bez argumentów. Jeśli jednak konstruktor zgłosi wyjątek kontrolowany, zostaje on ponownie zgłoszony bez sprawdzenia. To stanowi złamanie zasady sprawdzania wyjątków w czasie kompilacji. Natomiast `Constructor.newInstance` opakowuje każdy wyjątek konstruktora w obiekt `InvocationTargetException`.

Metoda `newInstance` jest odpowiednikiem **konstruktora wirtualnego** w C++. Jednak konstruktory wirtualne w tym języku nie są właściwością języka, a tylko idiomem, który musi być obsługiwany przez specjalną bibliotekę. Klasa `Class` jest podobna do klasy `type_info` w C++, a metoda `getClass` jest odpowiednikiem operatora `typeid`. Klasa `Class` Javy jest jednak nieco bardziej wszechstronna niż `type_info`. Ta druga potrafi tylko zwrócić łańcuch z nazwą typu. Nie tworzy nowych obiektów tego typu.

`java.lang.Class` **1.0**

- `static Class forName(string className)`

  Zwraca obiekt klasy `Class` reprezentujący klasę o nazwie `className`.

- `Constructor getConstructor(Class... parameterTypes)` **1.1**

  Zwraca obiekt opisujący konstruktor za pomocą podanych typów parametrów. Zobacz punkt 5.9.7 „Wywoływanie dowolnych metod i konstruktorów”, aby się dowiedzieć, jak dostarczać te typy parametrów.

`java.lang.reflect.Constructor` **1.1**

- `Object newInstance(Object[] args)`

  Tworzy nowy egzemplarz klasy zawierającej deklarację konstruktora, przekazując do niego parametry. Więcej informacji na temat przekazywania parametrów znajduje się w punkcie 5.9.7.

`java.lang.Throwable` **1.0**

- `void printStackTrace()`

  Drukuje obiekt `Throwable` i dane ze stosu do standardowego strumienia błędów.

5.9.2. Podstawy deklarowania wyjątków

Techniki przechwytywania wyjątków zostały opisane w rozdziale 7., ale zanim do niego dojdziemy, napotkamy po drodze kilka metod, które grożą, że mogą spowodować wyjątek.

Kiedy w czasie działania programu występuje błąd, program może spowodować wyjątek. Mechanizm wyjątków zapewnia większą elastyczność niż kończenie programu, ponieważ można napisać procedurę, która *przechwyci taki* wyjątek i go odpowiednio obsłuży.

Jeśli nie ma procedury obsługi wyjątku, program zostaje zakończony, a w konsoli zostaje wydrukowany komunikat informujący o jego typie. Komunikat taki może się pojawić w wyniku przypadkowego użycia referencji `null` lub przekroczenia rozmiaru tablicy.

Są dwa rodzaje wyjątków: **niekontrolowane** (ang. *unchecked*) i **kontrolowane** (ang. *checked*). W przypadku tych drugich kompilator sprawdza, czy programista wie o wyjątku i czy jest przygotowany na konsekwencje jego zgłoszenia. Jednak wiele wyjątków, na przykład błędy zakresu i dostęp do referencji `null`, jest niekontrolowanych, tzn. kompilator nie oczekuje, że programista dostarczy dla nich procedurę obsługi. W ich przypadku energię należy poświęcić na uniknięcie ich powstania, a nie na tworzenie procedur do ich obsłużenia.

Nie wszystkich błędów można jednak uniknąć. Jeśli mimo najlepszych chęci programisty wyjątek może się pojawić, większość API Javy zgłasza wyjątek kontrolowany. Przykładem jest metoda `Class.forName`. Nie da się zagwarantować, że będzie istniała klasa o określonej nazwie. W rozdziale 7. znajduje się opis kilku technik obsługi wyjątków. Poniżej pokazana jest tylko najprostsza strategia.

Jeśli metoda zawiera instrukcję mogącą zgłosić kontrolowany wyjątek, do nazwy tej metody należy dodać klauzulę `throws`.

```
public static void doSomethingWithClass(String name)
     throws ReflectiveOperationException
{
    Class cl = Class.forName(name); // może zgłosić wyjątek
    jakieś operacje na cl
}
```

Każda metoda wywołująca tę metodę również musi mieć deklarację `throws`. Dotyczy to także metody `main`. Jeśli rzeczywiście wystąpi wyjątek, metoda `main` zakończy działanie, wyświetlając dane ze stosu. (W rozdziale 7. dowiesz się, jak przechwytywać wyjątki, zamiast pozwalać im doprowadzić do zamknięcia programu).

Klauzula `throws` jest wymagana tylko dla wyjątków kontrolowanych. To, które metody zgłaszają takie wyjątki, można łatwo sprawdzić – kompilator zawsze informuje o próbie wywołania metody grożącej wyjątkiem kontrolowanym, gdy programista nie dostarczy odpowiedniej procedury obsługi tego wyjątku.

5.9.3. Zasoby

Klasy często mają powiązane pliki danych, np.:

- pliki graficzne i dźwiękowe,
- pliki tekstowe z komunikatami i etykietami przycisków.

W Javie takie pliki powiązane nazywają się **zasobami**.

Spójrz na przykład na okno dialogowe z napisem widoczne na rysunku 5.4.

Rysunek 5.4. Wyświetlony obraz i tekst

Oczywiście tytuł książki i informacja o prawach autorskich w następnym wydaniu książki zostaną zmienione. Aby ułatwić sobie zmianę tych informacji, umieścimy ten tekst w pliku, zamiast kodować go w łańcuchu.

Tylko gdzie umieścić taki plik jak *about.txt*? Najwygodniej byłoby go wrzucić razem z pozostałymi plikami programu do archiwum JAR.

Klasa `Class` zawiera praktyczną usługę do znajdowania plików zasobów. Poniżej znajduje się lista czynności, jakie należy wykonać, aby z niej skorzystać:

1. Zdobądź obiekt klasy `Class` zawierający interesujący Cię zasób — na przykład `ResourceTest.class`.
2. Niektóre metody, takie jak `getImage` z klasy `ImageIcon`, przyjmują adresy URL określające położenie zasobów. Wówczas należy zastosować wywołanie `URL url = cl.getResource("about.gif");`.
3. W przeciwnym przypadku należy użyć metody `getResourceAsStream`, aby uzyskać strumień wejściowy do wczytywania danych z pliku.

Maszyna wirtualna Javy potrafi znaleźć klasę, a więc potem może także poszukać powiązanych z nią zasobów *w tej samej lokalizacji*. Załóżmy, że klasa `ResourceTest` znajduje się w pakiecie `resources`. Wówczas plik *ResourceTest.class* znajduje się w katalogu *resources*, w którym należy też umieścić plik ikony.

Ewentualnie, zamiast umieszczać plik zasobu w tym samym katalogu, w którym znajduje się klasa, można podać względną lub bezwzględną ścieżkę, na przykład:

```
data/about.txt
/corejava/title.txt
```

Jedynym zadaniem funkcji ładowania zasobów jest automatyzacja ładowania plików. Nie ma żadnych standardowych metod do interpretacji zawartości plików zasobów. Każdy program musi mieć zdefiniowane własne techniki.

Zasoby są często wykorzystywane także do internacjonalizacji programów. Teksty, które powinny występować w wielu językach, na przykład powiadomienia i nazwy elementów interfejsu użytkownika, przechowuje się w plikach zasobów — po jednym dla każdego języka. **API internacjonalizacji**, którego opis znajduje się w rozdziale 7. tomu II, obsługuje standardową metodę organizacji i dostępu do plików lokalizacyjnych.

Na listingu 5.14 znajduje się program demonstrujący zastosowanie techniki ładowania zasobów. (Nie przejmuj się kodem wczytującym tekst i wyświetlającym okna dialogowe — wszystko wyjaśni się później). Skompiluj program, utwórz plik JAR i uruchom go:

```
javac resource/ResourceTest.java
jar cvfe ResourceTest.jar resources.ResourceTest \
resources/*.class resources/*.gif resources/data/*.txt corejava/*.txt
java -jar ResourceTest.jar
```

Przenieś plik JAR do innego katalogu i uruchom go ponownie, aby sprawdzić, czy program wczytuje zasoby z archiwum, a nie z bieżącego katalogu.

Listing 5.14. resources/ResourceTest.java

```
package resources;

import java.io.*;
import java.net.*;
import java.nio.charset.*;
import javax.swing.*;
/**
 * @version 1.5 2018-03-15
 * @author Cay Horstmann
 */
public class ResourceTest
{
    public static void main(String[] args) throws IOException
    {
        Class cl = ResourceTest.class;
        URL aboutURL = cl.getResource("about.gif");
        var icon = new ImageIcon(aboutURL);

        InputStream stream = cl.getResourceAsStream("data/about.txt");
        var about = new String(stream.readAllBytes(), "UTF-8");

        InputStream stream2 = cl.getResourceAsStream("/corejava/title.txt");
        var title = new String(stream2.readAllBytes(),
        ↪StandardCharsets.UTF_8).trim();

        JOptionPane.showMessageDialog(
           null, about, title, JOptionPane.INFORMATION_MESSAGE, icon);
    }
}
```

java.lang.Class 1.0

- `URL getResource(String name)` **1.1**
- `InputStream getResourceAsStream(String name)` **1.1**

  Znajduje zasób w tym samym miejscu, w którym znajduje się klasa, i zwraca URL lub strumień wejściowy, przy użyciu którego można załadować ten zasób. Jeśli nie znajdzie zasobu, zwraca `null`, a więc nie zgłasza wyjątku w przypadku błędu wejścia-wyjścia.

5.9.4. Zastosowanie refleksji w analizie funkcjonalności klasy

Poniżej znajduje się zwięzły opis najważniejszych funkcji mechanizmu refleksji, które umożliwiają analizę struktury klasy.

Trzy klasy — `Field`, `Method` i `Constructor` — dostępne w pakiecie `java.lang.reflect` opisują odpowiednio pola, metody i konstruktory klasy. Każda z nich ma metodę o nazwie `getName`, która zwraca nazwę odpowiedniego elementu. Klasa `Field` ma metodę `getType`, która zwraca obiekt typu `Class`, zawierający informacje o typie pola. Klasy `Method` i `Constructor` mają metody informujące o typach parametrów, a klasa `Method` informuje dodatkowo o typie zwrotnym. Każda z trzech wymienionych klas ma metodę `getModifiers`, która zwraca liczbę całkowitą z włączonymi i wyłączonymi różnymi bitami, określającą użyte modyfikatory, jak `public` i `static`. Do analizy liczby zwróconej przez tę metodę można użyć metod statycznych klasy `Modifier` dostępnej w pakiecie `java.lang.reflect`. Aby sprawdzić, czy metoda lub konstruktor miał modyfikator `public`, `private` lub `final`, należy użyć metod `isPublic`, `isPrivate` lub `isFinal` dostępnych w klasie `Modifier`. Jedyne, co jest potrzebne, to odpowiednia metoda w klasie `Modifier` działająca na liczbie zwróconej przez metodę `getModifiers`. Modyfikatory można także drukować za pomocą metody `Modifier.toString`.

Metody `getFields`, `getMethods` i `getConstructors` klasy `Class` zwracają tablice *publicznych* pól, metod i konstruktorów klasy. Do tego wliczają się publiczne składowe nadklasy. Metody `getDe` ↪`claredFields`, `getDeclaredMethods` i `getDeclaredConstructors` klasy `Class` zwracają tablice zawierające wszystkie pola, metody i konstruktory zadeklarowane w klasie. Wliczają się do nich składowe prywatne, pakietów i chronione, jak również składowe z dostępem pakietowym, ale nie nadklasy.

Listing 5.15 prezentuje sposób wydrukowania wszystkich informacji o klasie. Ten program monituje o podanie nazwy klasy, po czym drukuje sygnatury wszystkich metod i konstruktorów oraz nazwy wszystkich pól klasy. Jeśli na przykład programowi zostanie podana klasa `java.lang.Double`, wydrukuje on następujące dane:

```
public final class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(double);
    public java.lang.Double(java.lang.String);

    public boolean equals(java.lang.Object);
```

```
    public static java.lang.String toString(double);
    public java.lang.String toString();
    public static int hashCode(double);
    public int hashCode();
    public static double min(double, double);
    public static double max(double, double);
    public static native long doubleToRawLongBits(double);
    public static long doubleToLongBits(double);
    public static native double longBitsToDouble(long);
    public int compareTo(java.lang.Double);
    public volatile int compareTo(java.lang.Object);
    public static int compare(double, double);
    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static java.lang.Double valueOf(java.lang.String);
    public static java.lang.Double valueOf(double);
    public static java.lang.String toHexString(double);
    public volatile java.lang.Object resolveConstantDesc(
        java.lang.invoke.MethodHandles$Lookup);
    public java.lang.Double resolveConstantDesc(java.lang.invoke.MethodHandles$Lookup);
    public java.util.Optional describeConstable();
    public boolean isNaN();
    public static boolean isNaN(double);
    public static double sum(double, double);
    public boolean isInfinite();
    public static boolean isInfinite(double);
    public static boolean isFinite(double);
    public static double parseDouble(java.lang.String);

    public static final double POSITIVE_INFINITY;
    public static final double NEGATIVE_INFINITY;
    public static final double NaN;
    public static final double MAX_VALUE;
    public static final double MIN_NORMAL;
    public static final double MIN_VALUE;
    public static final int MAX_EXPONENT;
    public static final int MIN_EXPONENT;
    public static final int SIZE;
    public static final int BYTES;
    public static final java.lang.Class TYPE;
    private final double value;
    private static final long serialVersionUID;
}
```

Należy zauważyć, że program ten potrafi przeanalizować każdą klasę, którą interpreter Javy potrafi załadować, a nie tylko te klasy, które były dostępne w czasie kompilacji. Ten program będziemy wykorzystywać w kolejnym rozdziale, w którym będziemy zaglądać do wnętrza klas wewnętrznych generowanych automatycznie przez kompilator.

`java.lang.Class` **1.0**

- `Field[] getFields()` **1.1**

Listing 5.15. reflection/ReflectionTest.java

```
package reflection;

import java.util.*;
import java.lang.reflect.*;

/**
 * Ten program wykorzystuje technikę refleksji do wydrukowania pełnych informacji o klasie.
 * @version 1.12 2021-06-15
 * @author Cay Horstmann
 */
public class ReflectionTest
{
   public static void main(String[] args)
   {
      // Wczytanie nazwy klasy z argumentów wiersza poleceń lub danych od użytkownika.
      String name;
      if (args.length > 0) name = args[0];
      else
      {
         var in = new Scanner(System.in);
         System.out.println("Podaj nazwę klasy (np. java.util.Date): ");
         name = in.next();
      }

      // drukowanie modyfikatorów i nazwy klasy oraz nazwy nadklasy (if != Object)
        Class cl = Class.forName(name);
    String modifiers = Modifier.toString(cl.getModifiers());
    if (modifiers.length() > 0) System.out.print(modifiers + " ");
    if (cl.isSealed())
        System.out.print("sealed ");
    if (cl.isEnum())
        System.out.print("enum " + name);
    else if (cl.isRecord())
        System.out.print("record " + name);
    else if (cl.isInterface())
        System.out.print("interface " + name);
    else
        System.out.print("class " + name);
    Class supercl = cl.getSuperclass();
    if (supercl != null && supercl != Object.class) System.out.print(" extends "
            + supercl.getName());

    printInterfaces(cl);
    printPermittedSubclasses(cl);

    System.out.print("\n{\n");
    printConstructors(cl);

    System.out.println();
    printMethods(cl);
    System.out.println();
    printFields(cl);
    System.out.println("}");
}
```

```
/**
 * Drukowanie wszystkich konstruktorów klasy.
 * @param cl klasa
 */
public static void printConstructors(Class cl)
{
   Constructor[] constructors = cl.getDeclaredConstructors();

   for (Constructor c : constructors)
   {
      String name = c.getName();
      System.out.print("   ");
      String modifiers = Modifier.toString(c.getModifiers());
      if (modifiers.length() > 0) System.out.print(modifiers + " ");
      System.out.print(name + "(");

      // Drukowanie typów parametrów.
      Class[] paramTypes = c.getParameterTypes();
      for (int j = 0; j < paramTypes.length; j++)
      {
         if (j > 0) System.out.print(", ");
         System.out.print(paramTypes[j].getName());
      }
      System.out.println(");");
   }
}

/**
 * Drukuje wszystkie metody klasy.
 * @param cl klasa
 */
public static void printMethods(Class cl)
{
   Method[] methods = cl.getDeclaredMethods();

   for (Method m : methods)
   {
      Class retType = m.getReturnType();
      String name = m.getName();

      System.out.print("   ");
      // Drukowanie modyfikatorów, typu zwrotnego i nazwy metody.
      String modifiers = Modifier.toString(m.getModifiers());
      if (modifiers.length() > 0) System.out.print(modifiers + " ");
      System.out.print(retType.getName() + " " + name + "(");

      // Drukowanie typów parametrów.
      Class[] paramTypes = m.getParameterTypes();
      for (int j = 0; j < paramTypes.length; j++)
      {
         if (j > 0) System.out.print(", ");
         System.out.print(paramTypes[j].getName());
      }
      System.out.println(");");
   }
}
```

```
   /**
    * Drukowanie wszystkich pól klasy.
    * @param cl klasa
    */
   public static void printFields(Class cl)
   {
      Field[] fields = cl.getDeclaredFields();

      for (Field f : fields)
      {
         Class type = f.getType();
         String name = f.getName();
         System.out.print("   ");
         String modifiers = Modifier.toString(f.getModifiers());
         if (modifiers.length() > 0) System.out.print(modifiers + " ");
         System.out.println(type.getName() + " " + name + ";");
      }
   }

/**
* Drukuje wszystkie dozwolone podtypy zapieczętowanej klasy
* @param cl a class
*/
public static void printPermittedSubclasses(Class cl)
{
    if (cl.isSealed())
    {
        Class<?>[] permittedSubclasses = cl.getPermittedSubclasses();
        for (int i = 0; i < permittedSubclasses.length; i++)
        {
            if (i == 0)
                System.out.print(" permits ");
            else
                System.out.print(", ");
                System.out.print(permittedSubclasses[i].getName());
            }
        }
    }

    /**
     * Drukuje wszystkie bezpośrednio zaimplementowane interfejsy klasy
     * @param cl a class
     */
    public static void printInterfaces(Class cl)
    {
        Class<?>[] interfaces = cl.getInterfaces();
        for (int i = 0; i < interfaces.length; i++)
    {
        if (i == 0)
            System.out.print(cl.isInterface() ? " extends " : " implements ");
        else
            System.out.print(", ");
            System.out.print(interfaces[i].getName());
        }
    }
}
```

- `Field[] getDeclaredFields()` **1.1**

  Metoda `getFields` zwraca tablicę zawierającą obiekty `Field` reprezentujące pola publiczne klasy lub nadklasy. Metoda `getDeclaredFields` zwraca tablicę obiektów `Field` reprezentujących wszystkie pola klasy. Obie metody zwracają tablicę o zerowej długości, jeśli nie ma takich pól lub obiekt `Class` reprezentuje typ podstawowy bądź tablicowy.

- `Method[] getMethods` **1.1**
- `Method[] getDeclaredMethods()` **1.1**

  Zwraca tablicę obiektów `Method`. Metoda `getMethods` zwraca metody publiczne, wliczając metody odziedziczone. Metoda `getDeclaredMethods` zwraca wszystkie metody klasy lub interfejsu, ale nie uwzględnia metod odziedziczonych.

- `Constructor[] getConstructors()` **1.1**
- `Constructor[] getDeclaredConstructors()` **1.1**

  Zwraca tablicę obiektów `Constructor` reprezentujących wszystkie konstruktory publiczne (`getConstructors`) lub wszystkie konstruktory w ogóle (`getDeclared`
  ↪`Constructors`) klasy reprezentowanej przez obiekt `Class`.

- `isInterface()`

  Zwraca `true`, jeśli ten obiekt `Class` opisuje interfejs (interfejsy są opisane w rozdziale 6.).

- `isEnum()` **1.5**

  Zwraca `true`, jeśli ten obiekt `Class` opisuje wyliczenie.

- `isRecord()` **16**

  Zwraca `true`, jeśli ten obiekt `Class` opisuje rekord.

- `RecordComponent[] getRecordComponents()` **16**

  Zwraca tablicę obiektów `RecordComponent`, które opisują pola rekordu, lub `null`, jeśli ta klasa nie jest rekordem.

- `String getPackageName()` **9**

  Pobiera nazwę pakietu zawierającego ten typ lub pakietu typu elementu, jeśli jest to typ tablicowy, lub `"java.lang"`, jeśli jest to typ podstawowy.

```
java.lang.reflect.Field 1.1
java.lang.reflect.Method 1.1
java.lang.reflect.Constructor 1.1
```

- `Class getDeclaringClass()`

  Zwraca obiekt klasy `Class` reprezentujący klasę, która definiuje dany konstruktor, metodę lub pole.

- `Class[] getExceptionTypes()` (tylko klasy `Constructor` i `Method`)

  Zwraca tablicę obiektów `Class`, które reprezentują typy wyjątków powodowanych przez metodę.

- `int getModifiers()`

  Zwraca liczbę całkowitą opisującą modyfikatory konstruktora, metody lub pola. Do analizy zwróconej wartości służą metody klasy `Modifier`.

- `String getName()`

  Zwraca w postaci łańcucha nazwę konstruktora, metody lub pola.

- `Class[] getParameterTypes()` (tylko klasy `Constructor` i `Method`)

  Zwraca tablicę obiektów klasy `Class` reprezentujących typy parametrów.

- `Class getReturnType()` (tylko w klasie `Method`)

  Zwraca obiekt klasy `Class` reprezentujący typ zwrotny.

`java.lang.reflect.RecordComponent` **16**

- `String getName()`
- `Class<?> getType()`

  Pobiera nazwę i typ tego komponentu rekordu.

- `Method getAccessor()`

  Zwraca obiekt `Method` dający dostęp do tego komponentu rekordu.

`java.lang.reflect.Modifier` **1.1**

- `static String toString(int modifiers)`

  Zwraca w postaci łańcucha modyfikatory odpowiadające bitom ustawionym przez metodę `modifiers`.

- `static boolean isAbstract(int modifiers)`
- `static boolean isFinal(int modifiers)`
- `static boolean isInterface(int modifiers)`
- `static boolean isNative(int modifiers)`
- `static boolean isPrivate(int modifiers)`
- `static boolean isProtected(int modifiers)`
- `static boolean isPublic(int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict(int modifiers)`
- `static boolean isSynchronized(int modifiers)`

- `static boolean isVolatile(int modifiers)`

  Sprawdza bit w wartości `modifiers` odpowiadający modyfikatorowi znajdującemu się w nazwie metody.

5.9.5. Refleksja w analizie obiektów w czasie działania programu

W poprzednim podrozdziale nauczyliśmy się sprawdzać *nazwy* i *typy* pól egzemplarza:

- Tworzymy odpowiedni obiekt klasy `Class`.
- Wywołujemy na rzecz obiektu `Class` metodę `getDeclaredFields`.

Teraz pójdziemy o krok dalej i dobierzemy się do **zawartości** pól danych. Oczywiście zawartość określonego pola obiektu o znanych w trakcie pisania programu typie i nazwie można podejrzeć bez trudu. Jednak refleksja umożliwia uzyskanie informacji o polach obiektów, które w czasie kompilacji nie były jeszcze znane.

Kluczowe znaczenie ma w tym przypadku metoda `get` z klasy `Field`. Jeśli `f` jest obiektem typu `Field` (na przykład utworzonym za pomocą metody `getDeclaredFields`), a `obj` jest obiektem klasy, której polem jest `f`, wywołanie `f.get(obj)` zwraca obiekt, którego wartością jest aktualna wartość pola obiektu `obj`. Przeanalizujmy to nieco skomplikowane zagadnienie na przykładzie.

```
var harry = new Employee("Henryk Kwiatek", 35000, 10, 1, 1989);
Class cl = harry.getClass();
   // Obiekt Class reprezentujący pracownika.
Field f = cl.getDeclaredField("name");
   // Pole name klasy Employee.
Object v = f.get(harry);
   // Wartość pola name obiektu harry
   // tj. obiekt klasy String "Henryk Kwiatek".
```

Oczywiście wartości, które można pobrać, można też ustawiać. Wywołanie `f.set(obj, value)` ustawia wartość pola reprezentowanego przez `f` obiektu `obj` na nową wartość.

Ten kod sprawia jednak jeden problem. Ponieważ pole `name` jest prywatne, metody `get` i `set` spowodują wyjątek `IllegalAccessException`. Za pomocą tych metod można uzyskać dostęp tylko do wartości dostępnych pól. Zabezpieczenia w Javie zezwalają na sprawdzenie, jakie pola zawiera obiekt, ale nie pozwalają na sprawdzenie ich wartości bez odpowiednich uprawnień dostępu.

Przy standardowych ustawieniach mechanizm refleksji honoruje mechanizmy ochronne Javy. Można jednak ominąć ustawienia ochrony dostępu. W tym celu należy wywołać metodę `set` ↪`Accessible` na rzecz obiektu klasy `Field`, `Method` lub `Constructor`. Na przykład:

```
f.setAccessible(true);     // Teraz można wywołać f.get(harry);.
```

Metoda `setAccessible` należy do klasy `AccessibleObject`, która jest wspólną nadklasą klas `Field`, `Method` i `Constructor`. Została ona utworzona z myślą o debuggerach, schowkach i podobnych mechanizmach. Nieco dalej używamy tej metody dla generycznej metody `toString`.

Wywołanie metody `setAccessible` powoduje zgłoszenie wyjątku, jeśli dostęp nie zostanie udzielony. Dostępu może odmówić system modułów (rozdział 9. w tomie II) lub menedżer zabezpieczeń (rozdział 10. w tomie II). Menedżery zabezpieczeń są używane rzadko i w Javie 17 otrzymały status wycofywanych. Natomiast od Javy 9 każdy program zawiera moduły, ponieważ API został na nie podzielony.

Na przykład program znajdujący się na końcu tego punktu zagląda do wnętrza obiektów `Array` `↪List` i `Integer`. Kiedy go uruchomisz w wersji Javy od 9 do 16, pojawi się następujące złowieszcze ostrzeżenie:

```
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by objectAnalyzer.ObjectAnalyzer
↪(file:/home/cay/books/cj11/code/v1ch05/bin/) to field
↪java.util.ArrayList.serialVersionUID
WARNING: Please consider reporting this to the maintainers of
↪objectAnalyzer.ObjectAnalyzer
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective
↪access operations
WARNING: All illegal access operations will be denied in a future release
```

Jeśli uruchomisz ten program w Javie 17, zostanie zgłoszony wyjątek `InaccessibleObjectEx` `↪ception`.

Aby program nadal działał, „otwórz” pakiety `java.util` i `java.lang` w module `java.base` do „modułu bez nazwy”. Szczegółowe informacje znajdują się w rozdziale 9. tomu II. Oto cała składnia:

```
java --add-opens java.base/java.util=ALL-UNNAMED \
--add-opens java.base/java.lang=ALL-UNNAMED \
objectAnalyzer.ObjectAnalyzerTest
```

Przyszłe wersje bibliotek mogą używać *uchwytów do zmiennych* zamiast refleksji w celu odczytu i zapisu pól. Obiekt `VarHandle` jest podobny do `Field`. Przy jego użyciu możesz odczytać lub zapisać określone pole dowolnego egzemplarza wybranej klasy. Aby jednak uzyskać `VarHandle`, biblioteka musi mieć obiekt `Lookup`:

```
public Object getFieldValue(Object obj, String fieldName, Lookup lookup)
throws NoSuchFieldException, IllegalAccessException
{
Class<?> cl = obj.getClass();
    Field field = cl.getDeclaredField(fieldName);
    VarHandle handle = MethodHandles.privateLookupIn(cl, lookup)
    .unreflectVarHandle(field);
    return handle.get(obj);
}
```

To zadziała, pod warunkiem że obiekt `Lookup` zostanie wygenerowany w module, który ma uprawnienia dostępu do pola. Metoda w module po prostu wywołuje `MethodHand` `↪les.lookup()`, która zwraca obiekt i posiada prawa dostępu wywołującego. W ten sposób jeden moduł może udzielić prawa dostępu do prywatnych składowych innego modułu. Praktycznym problemem jest to, jak te uprawnienia można nadać jak najmniejszym wysiłkiem.

Dopóki można jeszcze korzystać z tej techniki, przyjrzymy się generycznej metodzie `toString`, która działa dla każdej klasy (listing 5.16). Generyczna metoda `toString` pobiera wszystkie pola egzemplarza za pomocą metody `getDeclaredFields`. Następnie metoda `setAccessible` umożliwia dostęp do wszystkich tych pól. Pobierane są nazwa i wartość każdego pola. Program przedstawiony na listingu 5.16 rekurencyjnie wywołuje metodę `toString`, zamieniając każdą wartość na łańcuch.

Listing 5.16. objectAnalyzerTest/ObjectAnalyzerTest.java

```
package objectAnalyzer;

import java.util.*;

/**
 * Ten program analizuje obiekty za pomocą refleksji.
 * @version 1.13 2018-03-16
 * @author Cay Horstmann
 */
public class ObjectAnalyzerTest
{
   public static void main(String[] args)
      throws ReflectiveOperationException
   {
      var squares = new ArrayList<Integer>();
      for (int i = 1; i <= 5; i++)
         squares.add(i * i);
      System.out.println(new ObjectAnalyzer().toString(squares));
   }
}
```

Generyczna metoda `toString` musi rozwiązywać kilka problemów. Cykliczne odwołania mogą spowodować nieskończoną rekurencję. Dlatego klasa `ObjectAnalyzer` (listing 5.17) zapamiętuje obiekty, które były już odwiedzane. Aby zajrzeć do tablic, potrzebne jest zastosowanie innej metody. Więcej szczegółów na ten temat znajduje się w kolejnym podrozdziale.

Listing 5.17. objectAnalyzer/ObjectAnalyzer.java

```
package objectAnalyzer;

import java.lang.reflect.AccessibleObject;
import java.lang.reflect.Array;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.ArrayList;

class ObjectAnalyzer
{
   private ArrayList<Object> visited = new ArrayList<>();

   /**
    * Konwertuje obiekt na łańcuch zawierający listę wszystkich pól.
    * @param obj obiekt
    * @return łańcuch zawierający nazwę klasy obiektu oraz nazwy i wartości wszystkich pól.
    */
```

```
   public String toString(Object obj)
         throws ReflectiveOperationException
   {
      if (obj == null) return "null";
      if (visited.contains(obj)) return "...";
      visited.add(obj);
      Class cl = obj.getClass();
      if (cl == String.class) return (String) obj;
      if (cl.isArray())
      {
         String r = cl.getComponentType() + "[]{";
         for (int i = 0; i < Array.getLength(obj); i++)
         {
            if (i > 0) r += ",";
            Object val = Array.get(obj, i);
            if (cl.getComponentType().isPrimitive()) r += val;
            else r += toString(val);
         }
         return r + "}";
      }

      String r = cl.getName();
      // Inspekcja pól tej klasy i wszystkich nadklas.
      do
      {
         r += "[";
         Field[] fields = cl.getDeclaredFields();
         AccessibleObject.setAccessible(fields, true);
         // Pobranie nazw i wartości wszystkich pól.
         for (Field f : fields)
         {
            if (!Modifier.isStatic(f.getModifiers()))
            {
               if (!r.endsWith("[")) r += ",";
               r += f.getName() + "=";
               Class t = f.getType();
               Object val = f.get(obj);
               if (t.isPrimitive()) r += val;
               else r += toString(val);
            }
         }
         r += "]";
         cl = cl.getSuperclass();
      }
      while (cl != null);

      return r;
   }
}
```

Za pomocą metody `toString` można zajrzeć do środka każdego obiektu. Na przykład wywołanie:

```
var squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

zwraca:

```
java.util.ArrayList[elementData=class java.lang.Object[]{java.lang.Integer[value=1][][],
java.lang.Integer[value=4][][],java.lang.Integer[value=9][][],
java.lang.Integer[value=16][][],
java.lang.Integer[value=25][][],null,null,null,null,null},size=5][modCount=5][][]
```

Przy użyciu generycznej metody `toString` można zaimplementować metody `toString` w poszczególnych klasach:

```
public String toString()
{
   return new ObjectAnalyzer().toString(this);
}
```

Jest to bezproblemowa metoda na utworzenie uniwersalnej metody `toString`. Zanim jednak zaczniesz się cieszyć, że to koniec z samodzielnym implementowaniem metody `toString`, przypomnij sobie, że dni niekontrolowanego dostępu do mechanizmów wewnętrznych są już policzone.

`java.lang.reflect.AccessibleObject` 1.2

- `void setAccessible(boolean flag)`

  Ustawia lub czyści znacznik dostępności obiektu lub zgłasza wyjątek `IllegalAccessException` w przypadku odmowy dostępu.

- `boolean trySetAccessible()` 9

  Ustawia znacznik dostępności dla tego dostępnego obiektu lub zwraca fałsz w przypadku odmowy dostępu.

- `boolean canAccess()`

  Sprawdza, czy wywołujący ma dostęp do `obj` przez ten obiekt pola, metody lub konstruktora. Dla statycznego pola lub metody albo dla konstruktora należy przekazać `null`.

- `static void setAccessible(AccessibleObject[] array, boolean flag)`

  Ustawia znacznik dostępności obiektów tablicowych.

`java.lang.Class` 1.1

- `Field getField(String name)`
- `Field[] getFields()`

  Zwraca pole publiczne o danej nazwie lub tablicę wszystkich pól.

- `Field getDeclaredField(String name)`
- `Field[] getDeclaredFields()`

  Zwraca pole o danej nazwie zadeklarowane w klasie lub tablicę wszystkich pól.

`java.lang.reflect.Field` **1.1**

- `Object get(Object obj)`

  Zwraca wartość pola reprezentowanego przez obiekt `Field` w obiekcie `obj`.

- `void set(Object obj, Object newValue)`

  Ustawia pole reprezentowane przez obiekt `Field` w obiekcie `obj` na nową wartość.

5.9.6. Zastosowanie refleksji w generycznym kodzie tablicowym

Klasa `Array` dostępna w pakiecie `java.lang.reflect` umożliwia dynamiczne tworzenie tablic. Możliwość ta jest wykorzystana na przykład w implementacji metody `copyOf` w klasie `Array`. Przypomnijmy sobie, jak użyć tej metody do powiększenia tablicy, która została zapełniona.

```
var a = new Employee[100];
. . .
// Tablica jest pełna.
a = Arrays.copyOf(a, 2 * a.length);
```

Jak napisać taką metodę? Pomocny jest fakt, że tablicę `Employee[]` można przekonwertować na tablicę `Object[]`. Brzmi obiecująco. Oto pierwsza próba.

```
public static Object[] badCopyOf(Object[] a, int newLength) // nieprzydatna
{
   var newArray = new Object[newLength];
   System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
   return newArray;
}
```

Niestety jest problem z *użyciem* powstałej tablicy. Ten kod zwraca tablicę *obiektów* (`Object[]`). Odpowiada za to poniższy wiersz:

```
new Object[newLength]
```

Tablicy obiektów *nie można* rzutować na tablicę pracowników (`Employee[]`) — w czasie działania programu zostałby spowodowany wyjątek `ClassCastException`. Problem polega na tym, że jak nam wiadomo, tablice w Javie pamiętają typ swoich elementów, to znaczy typ elementu, który był użyty w wyrażeniu `new` podczas ich tworzenia. Można tymczasowo przekonwertować tablicę `Employee[]` na `Object[]`, a później wrócić do poprzedniego stanu. Niemożliwe jednak jest rzutowanie tablicy, która od chwili powstania jest typu `Object[]`, na typ `Employee[]`. Do napisania tego rodzaju generycznego kodu tablicy konieczna jest możliwość utworzenia nowej tablicy *tego samego* typu co oryginalna tablica. Do tego celu potrzebne są metody dostępne w klasie `Array` z pakietu `java.lang.reflect`. Kluczowe znaczenie ma metoda `newInstance` klasy `Array`, która tworzy nową tablicę. Metoda ta przyjmuje jako argumenty typ elementów i żądany rozmiar tablicy.

```
Object newArray = Array.newInstance(componentType, newLength);
```

Aby tego dokonać, trzeba znać typ elementów i rozmiar nowej tablicy.

Pierwszą wartość można uzyskać za pomocą wywołania `Array.getLength(a)`. Statyczna metoda `getLength` klasy `Array` zwraca rozmiar tablicy. Aby sprawdzić typ elementów nowej tablicy:

1. Utwórz obiekt klasowy `a`.
2. Sprawdź, czy to na pewno jest tablica.
3. Znajdź odpowiedni typ dla tablicy za pomocą metody `getComponentType` (która jest zdefiniowana tylko dla obiektów klasowych reprezentujących tablice) klasy `Class`.
4. Z drugiej strony dla każdego obiektu `Class` reprezentującego klasę `C` metoda `arrayType` zwraca obiekt `Class` reprezentujący `C[]`.

Dlaczego metoda `getLength` należy do klasy `Array`, a `getComponentType` do klasy `Class`? Nie wiadomo — czasami wydaje się, że rozkład w klasach metod refleksyjnych jest nieco przypadkowy.

Oto potrzebny kod:

```
public static Object goodCopyOf(Object a, int newLength)
{
   Class cl = a.getClass();
   if (!cl.isArray()) return null;
   Class componentType = cl.getComponentType();
   int length = Array.getLength(a);
   Object newArray = Array.newInstance(componentType, newLength);
   System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
   return newArray;
}
```

Należy zauważyć, że metoda `copyOf` może powiększać tablice każdego typu, nie tylko przechowujące obiekty.

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodCopyOf(a, 10);
```

Aby to było możliwe, parametr metody `goodCopyOf` jest typu `Object`, a *nie tablicą obiektów* (`Object[]`). Tablicę typu `int[]` można przekonwertować na `Object`, ale nie na tablicę obiektów!

Listing 5.18 demonstruje obie metody powiększania tablicy. Zauważ, że rzutowanie typu zwrotnego metody `badCopyOf` spowoduje wyjątek.

Listing 5.18. arrays/CopyOfTest.java

```
package arrays;

import java.lang.reflect.*;
import java.util.*;

/**
 * Ten program demonstruje zastosowanie refleksji do manipulacji tablicami.
 * @version 1.2 2012-05-04
 * @author Cay Horstmann
 */
public class CopyOfTest
{
   public static void main(String[] args)
```

```
   {
      int[] a = { 1, 2, 3 };
      a = (int[]) goodCopyOf(a, 10);
      System.out.println(Arrays.toString(a));

      String[] b = { "Tomek", "Daniel", "Henryk" };
      b = (String[])goodCopyOf(b, 10);
      System.out.println(Arrays.toString(b));

      System.out.println("Poniższe wywołanie spowoduje wyjątek.");
      b = (String[]) badCopyOf(b, 10);
   }

   /**
    * Ta metoda próbuje powiększyć tablicę, tworząc nową tablicę i kopiując wszystkie elementy.
    * @param a tablica, która ma być powiększona.
    * @param newLength nowa długość tablicy
    * @return większa tablica zawierająca wszystkie elementy tablicy a. Zwrócona tablica jest
    * typu Object[], a nie takiego samego jak a.
    */
   public static Object[] badCopyOf(Object[] a, int newLength) // nieprzydatna
   {
      var newArray = new Object[newLength];
      System.arraycopy(a, 0, newArray, 0, Math.min(a.length, newLength));
      return newArray;
   }

   /**
    * Ta metoda powiększa tablicę, tworząc nową tablicę tego samego typu
    * i kopiując wszystkie elementy.
    * @param a tablica, która ma być powiększona. Może to być tablica obiektów lub
    * elementów typu podstawowego.
    * @return większa tablica zawierająca wszystkie elementy tablicy a.
    */
   public static Object goodCopyOf(Object a, int newLength)
   {
      Class cl = a.getClass();
      if (!cl.isArray()) return null;
      Class componentType = cl.getComponentType();
      int length = Array.getLength(a);
      Object newArray = Array.newInstance(componentType, newLength);
      System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
      return newArray;
   }
}
```

`java.lang.Class` **1.1**

- `boolean isArray()`

  Zwraca `true`, jeśli ten obiekt reprezentuje typ tablicowy.

- `Class<?> getComponentType()`
- `Class<?> componentType()` **12**

  Zwraca obiekt `Class` opisujący typ komponentu, jeśli ten obiekt reprezentuje typ tablicowy, lub `null` w przeciwnym razie.

- `Class<?> arrayType()` **12**

  Zwraca obiekt `Class` opisujący typ tablicy, którego typ komponentu jest reprezentowany przez ten obiekt.

`java.lang.reflect.Array` **1.1**

- `static Object get(Object array, int index)`
- `static xxx getXxx(Object array, int index)`

  xxx to jeden z typów podstawowych: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`. Te metody zwracają wartość przechowywaną w określonym indeksie tablicy.

- `static void set(Object array, int index, Object newValue)`
- `static setXxx(Object array, int index, xxx newValue)`

  xxx to jeden z typów podstawowych: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`. Te metody zapisują nową wartość w danej tablicy w określonym indeksie.

- `static int getLength(Object array)`

  Zwraca długość tablicy.

- `static Object newInstance(Class componentType, int length)`
- `static Object newInstance(Class componentType, int[] lengths)`

  Zwraca nową tablicę danego typu o określonych rozmiarach.

5.9.7. Wywoływanie dowolnych metod i konstruktorów

W językach C i C++ można wywołać dowolną funkcję, posługując się ustawionym na nią wskaźnikiem. Na pierwszy rzut oka wydaje się, że w Javie nie ma wskaźników do metod — umożliwiają one podanie lokalizacji metody innej metodzie, dzięki czemu ta druga może później wywołać tę pierwszą. Projektanci języka Java stwierdzili nawet, że wskaźniki do metod nie są bezpieczne, mogą bowiem stanowić źródło błędów, a lepszym od nich rozwiązaniem są **interfejsy** i wyrażenia lambda (opisane w kolejnym rozdziale). Jednak dzięki refleksji także w Javie można wywoływać dowolne metody.

Przypomnijmy sobie, że za pomocą metody `get` klasy `Field` można sprawdzić dowolne pole obiektu. Podobnie klasa `Method` zawiera metodę `invoke`, która umożliwia wywołanie metody zapakowanej w bieżący obiekt klasy `Method`. Sygnatura metody `invoke` wygląda następująco:

```
Object invoke(Object obj, Object... args)
```

Pierwszy jest parametr niejawny, a pozostałe obiekty to parametry jawne.

W przypadku metody statycznej pierwszy parametr jest ignorowany — można go ustawić na `null`.

Jeśli na przykład `m1` reprezentuje metodę `getName` klasy `Employee`, poniższy kod pokazuje, jak można ją wywołać:

```
String n = (String) m1.invoke(harry);
```

Jeśli typ zwrotny jest podstawowy, metoda `invoke` zwróci typ opakowania. Załóżmy na przykład, że `m2` reprezentuje metodę `getSalary` klasy `Employee`. Zwrócony obiekt będzie typu `Double` i trzeba wykonać odpowiednie rzutowanie. Można do tego celu zastosować automatyczne odpakowywanie.

```
double s = (Double) m2.invoke(harry);
```

Jak uzyskać obiekt klasy `Method`? Można oczywiście wywołać metodę `getDeclaredMethods` i przeszukać zwróconą tablicę w celu znalezienia żądanej metody. Można również wywołać metodę `getMethod` klasy `Class`. Jest ona podobna do metody `getField`, która przyjmuje nazwę pola w postaci łańcucha i zwraca obiekt typu `Field`. Jednak metod o takiej samej nazwie może być kilka, więc należy uważać, aby się nie pomylić. Z tego powodu konieczne jest podanie dodatkowo typów parametrów żądanej metody. Sygnatura metody `getMethod` jest następująca:

```
Method getMethod(String name, Class... parameterTypes)
```

Poniższy kod przedstawia sposób uzyskania wskaźników do metod `getName` i `raiseSalary` klasy `Employee`:

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

W podobny sposób można wywoływać dowolne konstruktory. Należy przekazać typy parametrów wybranego konstruktora do metody `Class.getConstructor` i dostarczyć ich wartości do metody `Constructor.newInstance`:

```
Class cl = Random.class; // lub dowolna inna klasa z konstruktorem
                         // przyjmującym parametr typu long
Constructor cons = cl.getConstructor(long.class);
Object obj = cons.newInstance(42L);
```

Klasy `Method` i `Constructor` rozszerzają klasę `Executable`. Od Javy 17 klasa `Execu`
`↪table` jest zapieczętowana oraz zezwala na tworzenie tylko podklas `Method` i `Constructor`.

Skoro znamy już zasady używania obiektów klasy `Method`, wykorzystajmy je w praktyce. Listing 5.19 przedstawia program drukujący tabelę wartości funkcji matematycznych, jak `Math.sqrt` lub `Math.sin`. Wydruk wygląda następująco:

```
public static native double java.lang.Math.sqrt(double)
1.0000  |   1.0000
2.0000  |   1.4142
3.0000  |   1.7321
4.0000  |   2.0000
5.0000  |   2.2361
6.0000  |   2.4495
7.0000  |   2.6458
8.0000  |   2.8284
9.0000  |   3.0000
10.0000 |   3.1623
```

Oczywiście kod drukujący tabelę jest niezależny od samej funkcji, dla której zastosowano wcięcia.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
   double y = (Double) f.invoke(null, x);
   System.out.printf("%10.4f | %10.4f%n", x, y);
}
```

W powyższym kodzie `f` jest obiektem typu `Method`. Pierwszy parametr metody `invoke` ma wartość `null`, ponieważ wywoływana jest metoda statyczna.

Wyrównanie funkcji `Math.sqrt` zostało uzyskane poprzez ustawienie `f` na:

```
Math.class.getMethod("sqrt", double.class)
```

czyli metodę klasy `Math` o nazwie `sqrt`, z parametrem typu `double`.

Listing 5.19 przedstawia pełny kod generycznego tabulatora i kilka przykładowych wywołań.

Listing 5.19. methods/MethodTableTest.java

```
package methods;

import java.lang.reflect.*;

/**
 * Ten program demonstruje sposób wywoływania metod poprzez refleksję.
 * @version 1.2 2012-05-04
 * @author Cay Horstmann
 */
public class MethodTableTest
{
   public static void main(String[] args) throws ReflectiveOperationException
   {
      // Pobranie wskaźników do metod square i sqrt.
      Method square = MethodTableTest.class.getMethod("square", double.class);
      Method sqrt = Math.class.getMethod("sqrt", double.class);

      // Drukowanie tabel wartości x i y.
      printTable(1, 10, 10, square);
      printTable(1, 10, 10, sqrt);
   }

   /**
    * Zwraca kwadrat liczby.
    * @param x liczba
    * @return x podniesione do kwadratu
    */
   public static double square(double x)
   {
      return x * x;
   }

   /**
    * Drukuje tablicę wartości x i y dla danej metody.
```

```
     * @param od dolnej granicy wartości x
     * @param do górnej granicy wartości x
     * @param n liczba wierszy w tabeli
     * @param f metoda z parametrem i typem zwrotnym typu double
     */
    public static void printTable(double from, double to, int n, Method f)
          throws ReflectiveOperationException
    {
       // Drukowanie metody jako nagłówka tabeli
       System.out.println(f);

       double dx = (to - from) / (n - 1);

       for (double x = from; x <= to; x += dx)
       {
          double y = (Double) f.invoke(null, x);
          System.out.printf("%10.4f | %10.4f%n", x, y);
       }
    }
}
```

Powyższy program pokazuje, że z obiektami klasy Method można zrobić wszystko to co ze wskaźnikami do funkcji w języku C (i delegacjami w C#). Podobnie jak w C, taki styl programowania jest niewygodny i zawsze podatny na błędy. Co się stanie, jeśli metoda zostanie wywołana przy użyciu nieprawidłowych parametrów? Metoda `invoke` spowoduje wyjątek.

Dodatkowo parametry i typy zwrotne metody `invoke` muszą być typu `Object`. Oznacza to konieczność częstego rzutowania w obie strony. W ten sposób kompilator zostaje pozbawiony możliwości sprawdzenia kodu. Przez to błędy ujawniają się tylko podczas testów, kiedy są trudniejsze do naprawienia. Ponadto kod tworzący wskaźnik do metody przy użyciu refleksji jest znacznie wolniejszy niż kod, który wywołuje metody bezpośrednio.

Z wymienionych powodów zalecamy używać obiektów klasy `Method` wyłącznie wtedy, kiedy jest to absolutnie niezbędne. Prawie zawsze lepiej jest używać interfejsów lub (od Java 8) wyrażeń lambda (które są tematem kolejnego rozdziału). Zgadzamy się zwłaszcza ze stanowiskiem projektantów języka Java i nie polecamy używania obiektów typu `Method` dla funkcji zwrotnych. Dzięki użyciu interfejsów dla metod zwrotnych (zobacz następny rozdział) kod jest znacznie szybszy i łatwiejszy w utrzymaniu.

`java.lang.reflect.Method` **1.1**

- `public Object invoke(Object implicitParameter, Object[] explicitParameters)`

  Wywołuje metodę reprezentowaną przez obiekt, przekazując podane parametry, oraz zwraca wartość, którą zwraca ta metoda. W przypadku metod statycznych parametr niejawny powinien mieć wartość `null`. Wartości typów podstawowych należy przekazywać w obiektach opakowujących. Wartości zwrotne typów podstawowych muszą być odpakowywane.

5.10. Porady projektowe dotyczące dziedziczenia

Na zakończenie tego rozdziału przedstawiamy kilka porad dotyczących dziedziczenia.

1. **Wspólne metody i pola umieszczaj w nadklasie.**

   Z tego powodu pole `name` umieszczamy w klasie `Person` zamiast w klasach `Employee` i `Student`.

2. **Nie używaj pól chronionych.**

   Niektórzy programiści uważają, że zdefiniowanie większości pól jako chronionych jest dobrym pomysłem, ponieważ dzięki temu podklasy mają w razie potrzeby do nich dostęp. Jednak mechanizm stojący za słowem kluczowym `protected` nie daje dobrej ochrony, i to z dwóch powodów. Po pierwsze, liczba podklas jest nieskończona — każdy może napisać podklasę naszej klasy, a następnie napisać kod uzyskujący bezpośredni dostęp do chronionych pól egzemplarzy, co stanowi złamanie zasad hermetyzacji. Po drugie, w Javie wszystkie klasy znajdujące się w jednym pakiecie mają dostęp do pól chronionych pozostałych klas, bez względu na to, czy są podklasami.

   Użyteczne mogą natomiast być metody chronione, które nie są gotowe do użytku i powinny zostać ponownie zdefiniowane w podklasach.

3. **Za pomocą dziedziczenia imituj relację „jest”.**

   Dziedziczenie umożliwia zaoszczędzenie wielu wierszy kodu, ale niestety jest często nadużywane. Wyobraźmy sobie na przykład, że potrzebujemy klasy o nazwie `Contractor` (pracownik kontraktowy). Pracownik kontraktowy ma imię i nazwisko oraz datę zatrudnienia, ale nie ma stałej pensji. Zamiast tego otrzymuje wynagrodzenie zależne od liczby przepracowanych godzin i nie pracuje na tyle długo, aby dostać podwyżkę. Istnieje więc pokusa, aby utworzyć podklasę `Contractor` klasy `Employee` i dodać pole `hourlyWage` (stawka godzinowa).

   ```
   class Contractor extends Employee
   {
      private double hourlyWage;
      . . .
   }
   ```

   Nie jest to jednak dobre rozwiązanie, ponieważ każdy obiekt pracownika kontraktowego będzie miał zarówno pole pensji, jak i stawki godzinowej. Stanie się to źródłem niekończących się problemów podczas implementacji metod generujących rachunki lub formularze podatkowe. Będzie konieczne napisanie większej ilości kodu, niż gdyby zrezygnowano na początku z dziedziczenia.

   Relacja pracownik – pracownik kontraktowy nie jest typu „jest”. Pracownik kontraktowy nie jest specjalnym typem pracownika.

4. **Nie używaj dziedziczenia, jeśli któraś z metod nie działa odpowiednio.**

   Wyobraźmy sobie, że chcemy napisać klasę o nazwie `Holiday`. Jak wiadomo, każde święto jest jakimś dniem, a dni można reprezentować jako obiekty klasy `GregorianCalendar`. W związku z tym możemy wykorzystać dziedziczenie.

   ```
   class Holiday extends GregorianCalendar { . . . }
   ```

   Niestety zbiór dni wolnych nie jest zamknięty dla wszystkich odziedziczonych metod. Jedną z publicznych metod klasy `GregorianCalendar` jest `add`. Za jej pomocą dni świąteczne można zamienić w dni nieświąteczne:

   ```
   Holiday christmas;
   christmas.add(Calendar.DAY_OF_MONTH, 12);
   ```

   Z tego powodu dziedziczenie nie jest właściwą techniką w tym przypadku.

   Problem ten nie wystąpiłby w przypadku rozszerzenia klasy niezmiennej. Powiedzmy, że mamy niezmienną klasę reprezentującą datę, podobną do `localDate`, ale nie finalną. Jeśli utworzymy podklasę `Holiday`, to nie będzie takiej metody, która mogłaby zamienić święto na dzień nieświąteczny.

5. **Przesłaniając metodę, nie zmieniaj jej spodziewanego działania.**

   Zasada zamienialności ma zastosowanie nie tylko do składni, ale co ważniejsze — do zachowania. Przesłaniając metodę, nie należy bez powodu zmieniać jej zachowania. Kompilator w takiej sytuacji nie pomoże, ponieważ nie może sprawdzić, czy nowa definicja metody jest właściwa. Na przykład problem z metodą `add` w klasie `Holiday` można rozwiązać, definiując tę metodę ponownie w taki sposób, aby nic nie robiła lub powodowała wyjątek czy też przechodziła do kolejnego święta.

   Niestety ta poprawka łamie zasadę zamienialności. Poniższe instrukcje:

   ```
   int d1 = x.get(Calendar.DAY_OF_MONTH);
   x.add(Calendar.DAY_OF_MONTH, 1);
   int d2 = x.get(Calendar.DAY_OF_MONTH);
   System.out.println(d2 - d1);
   ```

   powinny *działać przewidywalnie*, bez względu na to, czy `x` jest typu `Gregorian`
   ↪`Calendar`, czy `Holiday`.

   Oczywiście na ten temat można toczyć zażarte spory dotyczące tego, co należy uważać za przewidywalne działanie. Na przykład niektórzy programiści stwierdzą, że zasada zamienialności wymaga, aby metoda `Manager.equals` ignorowała pole `bonus`, ponieważ robi to także metoda `Employee.equals`. Takie dyskusje są bezcelowe, jeśli prowadzi się je bez odpowiedniego kontekstu. Przede wszystkim należy pamiętać, aby przesłaniając metody w podklasach, nie zacierać przeznaczenia oryginalnego projektu.

6. **Wykorzystuj polimorfizm zamiast informacji o typach.**

   Kiedy napotkasz kod typu:

```
if (x jest typu 1)
działanie1(x);
else if (x jest typu 2)
działanie2(x);
```

zawsze pomyśl o polimorfizmie.

Czy *działanie*$_1$ i *działanie*$_2$ reprezentują wspólną koncepcję? Jeśli tak, zamień tę koncepcję na metodę nadklasy lub interfejsu wspólnego dla obu typów. Dzięki temu wystarczy wywołanie typu:

```
x.działanie();
```

Polimorficzny mechanizm dynamicznego przydzielania zadań wywoła odpowiednią metodę.

Kod wykorzystujący metody polimorficzne lub interfejsy jest zdecydowanie łatwiejszy do utrzymania i rozszerzania niż kod zawierający wiele sprawdzeń typów.

7. **Nie nadużywaj refleksji.**

   Możliwość wykrywania pól i metod w czasie działania programu pozwala na pisanie programów o zadziwiającym stopniu uogólnienia. Ta funkcjonalność jest przydatna w programowaniu systemów, ale nie sprawdza się w aplikacjach. Refleksja jest wrażliwym mechanizmem — kompilator nie może pomóc w znajdowaniu błędów. Wszystkie błędy są odkrywane w czasie działania programu i wtedy powodują wyjątki.

Znamy już zasady działania fundamentów programowania obiektowego: klas, dziedziczenia i polimorfizmu. W kolejnym rozdziale opisujemy dwa zaawansowane zagadnienia, bardzo ważne dla tych, którzy chcą efektywnie wykorzystywać możliwości Javy. Są to interfejsy i wyrażenia lambda.

Skorowidz

B

C

D

E

F

G

H

I

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

V

W

X

Z

JAVA:

biegle opanuj język mistrzów programowania!

W 1995 roku świat ujrzał przyszłą gwiazdę programowania: Javę. Dziś to język dojrzały i elastyczny, dzięki czemu może służyć do pisania dużych systemów, małych programów, aplikacji mobilnych i internetowych. Java została zaprojektowana z ogromną starannością. W język wbudowano wysublimowane zabezpieczenia, a także pewne zaawansowane funkcje, które docenia każdy programista tworzący systemy o skomplikowanej architekturze.
Ta książka jest kolejnym, zaktualizowanym i uzupełnionym wydaniem kultowego podręcznika dla profesjonalnych programistów Javy. To pierwszy tom, w którym opisano podstawy języka i najważniejsze zagadnienia związane z programowaniem interfejsu użytkownika, a także kolekcje, wyrażenia lambda, techniki programowania współbieżnego i funkcyjnego. W tym wydaniu poszczególne zagadnienia zoptymalizowano pod kątem Javy 17, opisano też takie nowości jak bloki tekstu, rozszerzenia konstrukcji *switch*, rekordy, dopasowywanie wzorców operatora *instanceof*, klasy zapieczętowane i wiele więcej. Podręcznik zawiera mnóstwo przykładów kodu obrazujących zasady działania niemal każdej opisywanej funkcji czy biblioteki. Aby nauka najważniejszych zagadnień była jeszcze łatwiejsza, przykładowe programy są proste i realistyczne.

W książce między innymi:

- składnia i najlepsze praktyki pisania kodu w języku Java
- interfejsy, klasy wewnętrzne i wyrażenia lambda
- obsługa wyjątków i skuteczne techniki debugowania
- korzystanie z typów generycznych i standardowych kolekcji Javy
- nowoczesne graficzne interfejsy użytkownika przy użyciu komponentów Swing
- stosowanie modelu współbieżności Javy

Cay S. Horstmann jest autorem najpopularniejszych w Polsce podręczników do nauki Javy, zdobywcą tytułu Java Champion i częstym gościem konferencji programistycznych. Urodził się w północnych Niemczech, obecnie mieszka w USA, gdzie pracuje jako profesor informatyki na Uniwersytecie Stanowym w San José.

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 978-83-283-9479-7

Cena: 129,00 zł

Java™
Pearson