Java. Programowanie obiektowe

Autor: Marek Wierzbicki
ISBN: 83-246-0290-9
Format: B5, stron: 264

Doskonałe wprowadzenie w świat obiektowości

- Podstawowe zasady programowania obiektowego
- Programowanie sterowane zdarzeniami
- Obsługa wyjątków i wielowątkowości

Programowanie obiektowe to technologia, która zdobyła już bardzo mocną pozycję wśród twórców oprogramowania. Nadal jednak wielu programistów, którzy zdobywali doświadczenie, używając języków proceduralnych, ma problemy z jej zrozumieniem i wszechstronnym stosowaniem. Wiele języków programowania określanych mianem „obiektowe" wywodzi się z języków proceduralnych, co ogranicza możliwości wykorzystywania wszystkich zalet obiektowości. Ograniczeń tych pozbawiona jest Java — stworzony od podstaw, nowoczesny, bezpieczny, niezależny od typu komputera i systemu operacyjnego, w pełni obiektowy język programowania.

Książka „Java. Programowanie obiektowe" opisuje wszystkie aspekty programowania obiektowego w Javie. Początkujący użytkownicy tego języka znajdą w niej wyjaśnienia nawet najbardziej skomplikowanych mechanizmów obiektowości, a ci, którzy posiadają już pewne doświadczenie, mogą wykorzystać ją w charakterze podręcznego kompendium wiedzy. Można znaleźć w niej omówienie zarówno podstawowych zagadnień, jak i zaawansowanych technik obsługi błędów, programowania wielowątkowego i sterowanego zdarzeniami. W książce przedstawiono również metody tworzenia wydajnie działających programów, które do uruchomienia nie wymagają maszyn o potężnej mocy obliczeniowej.

- Cechy programowania obiektowego
- Obiektowość w Javie
- Tworzenie i stosowanie klas i obiektów
- Budowanie pakietów
- Tworzenie apletów
- Komunikacja apletów ze skryptami Java Script
- Obiekty nasłuchujące i obsługa zdarzeń
- Przechwytywanie wyjątków
- Synchronizacja wątków

Poznaj możliwości technologii obiektowej w praktyce

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

Klasy i obiekty w Javie

Poprzedni rozdział wprowadzał ogólnie pojętą ideę programowania obiektowego oraz jej modyfikacje na potrzeby Javy. W tym rozdziale zajmę się tym samym problemem, ale tutaj pokażę środki realizacji idei opisanych wcześniej. Będziesz mógł dowiedzieć się, jak w praktyce realizuje się programowanie obiektowe z użyciem kodu źródłowego w Javie. Poszczególne konstrukcje języka są omówione z perspektywy kodowania oraz działania wirtualnej maszyny Javy, czyli JVM (*Java Virtual Machine*). Pomijam większość rozważań teoretycznych nad cechami poszczególnych konstrukcji, które opisałem wcześniej, dlatego liczę, że w dostateczny sposób zapoznałeś się z treścią poprzedniego rozdziału.

2.1. Klasy

Klasy określają postać, strukturę i działanie obiektów, które są egzemplarzami klas. W związku z zastosowaniem w Javie skrajnie ortodoksyjnego podejścia program napisany z użyciem tego języka musi mieć, poza kilkoma wyjątkami (czyli prostymi podstawowymi typami danych), strukturę oraz działanie lub algorytm, który wykonuje, zaprojektowane z użyciem klas (a zrealizowane z użyciem ich egzemplarzy, czyli obiektów).

2.1.1. Tworzenie klas

Najprostsza możliwa do stworzenia klasa ma postać:

```
class Simple {}
```

Charakteryzuje ją słowo kluczowe `class`, nazwa klasy (w tym wypadku `Simple`) oraz para nawiasów klamrowych, które reprezentują jej ciało (w tym przypadku są puste). Klasa ta musi być umieszczona w pliku *Simple.java*. Tak utworzony plik może zostać poddany poprawnej kompilacji i stanowić zupełnie poprawną (choć całkiem nieprzydatną) klasę Javy. Należy pamiętać, że każda klasa publiczna musi być zapisana w osobnym pliku, którego nazwa musi być dokładnie taka sama (oczywiście plus rozszerzenie

.java) jak nazwa klasy zdefiniowanej wewnątrz (włącznie z rozróżnieniem na duże i małe litery). Teoria mówi, że nazwy klas mogą zawierać tak zwane znaki narodowe, ale ze względu na różne standardy kodowania (nawet w obrębie jednego systemu operacyjnego) nie powinno się stosować liter innych niż łacińskie.

Definicja klasy podstawowej musi być tworzona według szablonu zaprezentowanego na listingu 2.1 (elementy ujęte w nawiasy kwadratowe są opcjonalne i nie muszą występować).

Listing 2.1. *Szablon definicji klasy*

```
[modyfikator] class NazwaKlasy {
  [modyfikator] typ nazwa_pola_1;
  ...
  [modyfikator] typ nazwa_pola_k;

  [modyfikator] typ nazwa_metody_1([lista_parametrów])
  {
    ciało_metody_1
  }
  ...
  [modyfikator] typ nazwa_metody_L([lista_parametrów])
  {
    ciało_metody_L
  }
}
```

Klasa może posiadać dowolną liczbę pól i metod (w tym zero, nawet łącznie dla pól i metod, jak pokazałem to wcześniej w najprostszej klasie `Simple`).

Poniżej umieszczam objaśnienie poszczególnych elementów zaprezentowanych w szablonie na listingu 2.1.

- `class` — słowo kluczowe określające definicję klasy.
- `NazwaKlasy` — identyfikator określający nazwę klasy.
- `modyfikator` — słowo lub słowa kluczowe oddzielone od siebie spacją określające sposób traktowania elementu, do którego się odnoszą. Modyfikator może też oznaczać ograniczenie lub rozszerzenie dostępu do elementu. Pełne wyjaśnienie znaczenia tego elementu języka znajduje się w podrozdziale 2.9. „Modyfikatory".
- `typ` — typ pola lub metody — może to być typ prosty (`byte`, `short`, `int`, `long`, `char`, `float`, `double` lub `boolean` oraz `void` — tylko w odniesieniu do metody), klasa bądź tablica (*array*) elementów jednego typu.
- `nazwa_pola_x` — identyfikator jednoznacznie określający pole konstruowanej klasy.
- `nazwa_metody_x` — identyfikator, który wraz z listą parametrów jednoznacznie określi metodę.
- `lista_parametrów` — lista par rozdzielonych przecinkami składających się z określenia typu i nazwy egzemplarza danego typu. Jeśli nie zamierzamy przekazać do metody żadnych parametrów, jej deklaracja powinna zawierać

parę pustych nawiasów. Zwracam tu uwagę na odstępstwa od C++, które w takim przypadku powinno (zamiast pustych nawiasów) zawierać słowo `void`, oraz różnice w stosunku do Object Pascala niezawierającego w takim przypadku nawiasów.

- `ciało_metody_x` — zbiór instrukcji języka Java określający funkcjonalność danej metody.

2.1.2. Pola

Pola są to miejsca, w których przechowywane są informacje charakterystyczne dla całej klasy bądź dla jej konkretnego egzemplarza. O polach mówi się też czasami, że są to egzemplarze zmiennych należące do konkretnego egzemplarza klasy. W praktyce możemy traktować pola jako lokalne zmienne danej klasy z zastrzeżeniem, że zakres ich widzialności i zachowania jest określony przez modyfikatory poszczególnych pól. Klasyczna deklaracja pola odbywa się według schematu:

```
[modyfikator] typ nazwa_pola_k;
```

Przykład klasy zawierającej tylko dwa pola pokazany jest na listingu 2.2.

Listing 2.2. *Klasa posiadająca tylko dwa pola*

```
class Point {
  int x;  // położenie na osi 0X
  int y;  // położenie na osi 0Y
}
```

W przykładzie tym pola są zmiennymi prostymi. Nie ma jednak żadnych przeciwwskazań, żeby były zmiennymi złożonymi, w tym również obiektami.

2.1.3. Metody

Inaczej niż inne języki obiektowe takie jak C++ czy Object Pascal, Java nie tylko gromadzi wszystkie informacje w plikach jednego rodzaju (tekstowych, z rozszerzeniem *.java*), ale również stara się je przechowywać w możliwie najbardziej skoncentrowany sposób. W C++ istnieją pliki nagłówkowe, które przechowują strukturę obiektów, i właściwe pliki z programem przechowujące między innymi obiekty. W Object Pascalu informacje te są co prawda zawarte w jednym pliku, jednak część jest w sekcji `interface`, część w `implementation`. W Javie wszystko jest w jednym miejscu. Cała informacja o metodzie zawarta jest tuż przed jej ciałem, tak jak to widać na listingu 2.3.

Listing 2.3. *Szablon definicji metody*

```
[modyfikator] typ nazwa_metody([lista_parametrów])
{
  // blok instrukcji
}
```

Jeśli typ metody jest różny od `void` (czyli funkcja zwraca jakąś wartość), powinna ona być zakończona wierszem:

```
return wyliczonaWartosc;
```

gdzie `wyliczonaWartosc` musi być takiego samego typu jak typ metody.

Po zaprezentowaniu schematu tworzenia klas mogę przystąpić do przedstawienia przykładu prostej klasy, która umożliwia przechowywanie informacji o położeniu punktu na płaszczyźnie wraz z metodami umożliwiającymi określenie położenia początkowego punktu i przemieszczenia go. Klasa ta pokazana jest na listingu 2.4.

Listing 2.4. *Klasa opisująca punkt*

```
class Point {
  int x;   // położenie na osi 0X
  int y;   // położenie na osi 0Y
  // ustawienie nowej pozycji
  public void newPosition(int newX, int newY) {
    x = newX;
    y = newY;
  }
  // przemieszczenie punktu
  public void changePosition(int dX, int dY) {
    x = x+dX;
    y = y+dY;
  }
}
```

W dalszej części tego rozdziału będę rozszerzał definicję tej klasy i precyzował jej znaczenie.

2.1.4. Hermetyzacja i modyfikator private

Wprowadzając ideę programowania obiektowego, zwracałem uwagę na jej podstawową cechę (i zarazem bardzo ważną zaletę), czyli hermetyzację. Klasa (a wraz z nią obiekt) miała gromadzić w jednym miejscu dane i procedury ich obsługi. Jednak miało to być zgromadzone w taki sposób, aby osoba używająca obiektu miała jak najmniejszy dostęp do danych (tylko do tych niezbędnych). Miało to zapewnić zarówno zmniejszenie liczby błędów popełnianych w czasie kodowania, jak i podniesienie przejrzystości programu. Przedstawiona wcześniej klasa `Point` nie stanowiła idealnej reprezentacji hermetycznej klasy, gdyż udostępniała na zewnątrz wszystkie, a nie tylko niezbędne elementy. Aby uniemożliwić dostęp do pól, które w idei klasy nie muszą być dostępne z zewnątrz, należy je oznaczyć modyfikatorem `private`. Na listingu 2.5 przedstawiam poprawioną, bardziej hermetyczną wersję klasy `Point`.

Listing 2.5. *Poprawiona klasa opisująca punkt*

```
class Point {
  private int x;   // położenie na osi 0X
  private int y;   // położenie na osi 0Y
```

```
  // odczyt wartości
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  // ustawienie nowej pozycji
  public void newPosition(int newX, int newY) {
    x = newX;
    y = newY;
  }
  // przemieszczenie punktu
  public void changePosition(int dX, int dY) {
    x = x+dX;
    y = y+dY;
  }
}
```

Na listingu 2.5 wytłuściłem różnice w stosunku do wcześniejszej wersji klasy, czyli ukrycie bezpośrednich wartości `x` i `y` oraz udostępnienie w zamian ich wartości przez metody `getX` i `getY`. Zaleta takiego rozwiązania jest widoczna. Nie można, nawet przez przypadek, odwołać się bezpośrednio do `x` i `y`, dzięki czemu nie może nastąpić przypadkowa ich modyfikacja. Aby je odczytać, trzeba jawnie wywołać `getX` lub `getY`. Aby je ustawić, trzeba jawnie wywołać `newPosition` (można też utworzyć metody `setX` i `setY`, aby ustawiać te parametry pojedynczo). Dopiero tak skonstruowana klasa spełnia warunki hermetyzacji.

2.1.5. Przeciążanie metod

Istnieją sytuacje, w których niektórzy programiści uważają, że wskazane jest, aby można było utworzyć kilka metod o tych samych nazwach, lecz o różnym zestawie parametrów. Jako przykład można pokazać kolejne rozszerzenie naszej klasy `Point` o nową wersję metody `newPosition`. Rozszerzenie to pokazane jest na listingu 2.6.

Listing 2.6. *Kolejna wersja klasy opisującej punkt*

```
class Point {
  private int x;   // położenie na osi 0X
  private int y;   // położenie na osi 0Y

  // ustawienie nowej pozycji
  public void newPosition(int newX, int newY) {
    x = newX;
    y = newY;
  }
  // ustawienie nowej pozycji na (0,0)
  public void newPosition() {
    x = 0;
    y = 0;
  }

  // pozostałe metody klasy Point
  // ...
}
```

Pokazana na listingu 2.6 klasa ma dwie metody `newPosition`. Jedna, wywołana z parametrami, ustawia współrzędne punktu na wartości podane jako parametry. Druga, bez parametrów, ustawia współrzędne punktu na wartość domyślną (0,0). Można próbować wyobrazić sobie sytuację, w której nie da się zastosować innego rozwiązania. Często jednak przeciążanie nie jest konieczne. Osobiście uważam, że kiedy tylko nie ma takiej potrzeby, nie powinno się go stosować. Jednak w standardowych bibliotekach Javy wiele funkcji jest przeciążonych, co powoduje, że programiści chętnie trzymają się takiego standardu kodowania. Na przykład w projektowanej przez nas klasie zamiast przeciążania metody `newPosition` można by zastosować dwie różne metody — `newPosition` oraz `defaultPosition`. Jeżeli jednak decydujemy się na przeciążanie metod, powinniśmy pamiętać o następujących uwagach:

- Metody rozróżniane są wyłącznie na podstawie liczby i typów przekazywanych do nich parametrów. Wywołanie metody powinno odbyć się z właściwym zestawem parametrów, gdyż w przeciwnym wypadku kompilator zgłosi błąd.
- Metody nie są rozróżniane na podstawie nazw parametrów formalnych, w związku z tym próba stworzenia dwóch metod o tym samym zestawie typów parametrów i różnych ich nazwach zakończy się błędem.
- Metody nie są również rozróżniane na podstawie typów zwracanej wartości. W związku z tym dwie metody o takim samym zestawie parametrów, lecz o różnym typie zwracanego wyniku zostaną potraktowane jak jedna metoda i kompilator również zgłosi błąd.
- Jak wszędzie, w Javie wielkość liter ma znaczenie. W związku z tym istnienie metod `newPosition` i `NewPosition` nie jest żadnym przeciążeniem, gdyż mają one różne nazwy (według mnie stosowanie nazw różniących się wyłącznie wielkością liter to bardzo zły pomysł).

2.1.6. Słowo kluczowe this

Java zawiera w swojej składni ciekawe, choć pozornie nieprzydatne słowo `this`. Z punktu widzenia formalnego wszystkie odwołania do własnych pól i metod są dokonywane w stosunku do tej klasy, w której się znajdujemy (czyli po angielsku właśnie `this`). Podobny mechanizm stosowany jest na przykład w Object Pascalu, który domyślnie zakłada, że wszystkie nieprzekierowane odwołania wykonywane są w stosunku do siebie (w Pascalu do przekierowań używa się słowa `Self`). Przykład wcześniej używanej metody `newPosition` może być (a w zasadzie z punktu widzenia skrajnego formalizmu powinien być) zapisany w postaci zaprezentowanej na listingu 2.7.

Listing 2.7. *Bardzo formalna definicja metody w klasie opisującej punkt*

```
public void newPosition(int newX, int newY) {
  this.x = newX;
  this.y = newY;
}
```

Oczywiście nikt tego nie robi, gdyż poza niepotrzebnym nakładem pracy nie zyskuje się w ten sposób żadnego ciekawego efektu. Nie zawsze jednak stosowanie tego przedrostka nie daje żadnego efektu. Istnieją sytuacje, kiedy kod źródłowy programu

bez słowa this nie determinuje poprawnie elementu, do którego zamierzaliśmy się odwołać. Dzieje się tak wtedy, gdy parametry metody mają takie same nazwy jak pola klasy. Na listingu 2.8 przedstawiam zmodyfikowaną wersję metody newPosition, w której użycie słowa this jest już jak najbardziej uzasadnione.

Listing 2.8. *Uzasadnione użycie słowa this w klasie opisującej punkt*

```
public void newPosition(int x, int y) {
  this.x = x;
  this.y = y;
}
```

Patrick Naughton, jeden ze współtwórców Javy, uważa, że taka konstrukcja upraszcza tekst źródłowy oraz czyni go bardziej przejrzystym i mniej podatnym na błędy. W związku z taką tezą stawianą przez współautora języka wiele osób nagminnie stosuje takie konstrukcje. Według mnie jest to niepotrzebny manieryzm, który zaciemnia obraz sytuacji i jest przyczyną dużej liczby drobnych i zupełnie niepotrzebnych błędów. Warto popatrzeć na hipotetyczną metodę newPosition pokazaną na listingu 2.9, która przelicza przed ustawieniem wartość położenia z cali na centymetry, aby można było zobaczyć, że łatwo jest się pomylić, stosując te same nazwy dla parametrów i pól klasy.

Listing 2.9. *Przykład popełnienia błędu zasięgu zmiennych*

```
classFloatPoint {
  float x, y;
  public void newPosition(float x, float y) {
    float xcm, ycm;
    xcm = 2.51*x;
    ycm = 2.51*y;
    x = xcm;    // zły zakres
    y = ycm;    // zły zakres
  }
}
```

Oczywiście kompilator nie zgłosi żadnego błędu, gdyż konstrukcja jest jak najbardziej poprawna, a my będziemy się zastanawiać, dlaczego pola obiektu nie są inicjowane we właściwy sposób. Otóż w wierszach oznaczonych na listingu 2.9 komentarzem *zły zakres* podstawiamy wartości do zmiennych, które posłużyły nam do przekazania wartości do metody, a które nie są widoczne na zewnątrz od niej (przykryły nazwy pól).

Rozszerzenie użycia słowa this pokazałem w paragrafach 2.1.8. „Przeciążenie konstruktorów", 2.4.3. „Zastosowanie interfejsów" oraz 2.3.4. „this w klasach wewnętrznych".

2.1.7. Konstruktor

Mimo iż zaprezentowana klasa Point jest w pełni funkcjonalna w zakresie, jakiego od niej oczekujemy, w praktyce brakuje jej elementu, który znacznie ułatwiłby jej (i każdej innej klasy) wykorzystanie. Otóż bezpośrednio po utworzeniu obiektu, czyli egzemplarza tej klasy (co przedstawię w dalszej części tego rozdziału), położenie nowego

punktu jest nieokreślone. Dopiero po użyciu metody `newPosition`, która jawnie deklaruje nowe położenie punktu, przestaje ono być nieokreślone, a zaczyna być takie, jak to zostało w niej ustawione. W związku z tym po każdorazowym utworzeniu takiego obiektu należałoby pamiętać o zainicjowaniu jego położenia. Znacznie wygodniej byłoby, gdyby inicjacja położenia punktu odbywała się automatycznie w czasie tworzenia obiektu. Jest to możliwe, pod warunkiem że skorzystamy z możliwości stosowania specjalnej metody zwanej konstruktorem, wywoływanej automatycznie w czasie tworzenia egzemplarza klasy. Od zwykłej metody odróżniają konstruktor dwie kwestie — nazwa zgodna z nazwą klasy oraz brak typu. W stosunku do konstruktora można stosować deklaracje zasięgu, przy czym dobra praktyka sugeruje, aby zasięg widzialności konstruktora był dokładnie taki sam jak samej klasy. Byłoby to bowiem dużym błędem, gdyby klasa była widziana, a jej konstruktor nie. Przykładowy konstruktor dla klasy `Point` pokazywanej wcześniej będzie miał postać zaprezentowaną na listingu 2.10.

Listing 2.10. *Konstruktor klasy opisującej punkt*

```
// konstruktor klasy Point
Point(int newX, int newY) {
  x = newX;
  y = newY;
}
```

Brak typu w deklaracji konstruktora wynika z tego, że w praktyce zwraca on wartość typu dokładnie takiego samego jak klasa, w której jest umieszczony, czyli domyślnie jego typ jest dokładnie taki jak nazwa klasy. Gdyby więc twórcy Javy chcieli być bardzo pedantyczni, deklaracja konstruktora powinna wyglądać jak na listingu 2.11.

Listing 2.11. *Hipotetyczna deklaracja konstruktora*

```
// teoretyczna deklaracja konstruktora
// (uwaga: błędna formalnie)
Point Point(int newX, int newY) {
  x = newX;
  y = newY;
}
```

Na szczęście nie ma potrzeby, aby tak utrudniać sobie życie.

2.1.8. Przeciążanie konstruktorów

O ile przeciążenia metod można uniknąć, stosując różne nazwy metod (na przykład dodając różne przyrostki), o tyle przeciążenie konstruktorów może okazać się niezbędne. Konstruktor to specyficzna, wywoływana w czasie tworzenia obiektu metoda o nazwie zgodnej z nazwą klasy. Ograniczenie takie (niewystępujące na przykład w Object Pascalu, gdzie konstruktor może mieć dowolną nazwę) wymusza stosowanie przeciążenia konstruktorów, jeśli chcemy korzystać z nich w sposób bardziej uniwersalny. Jako przykład weźmy pokazywaną wcześniej klasę `Point`. Sugeruję dodanie do niej drugiego konstruktora bez parametrów, który będzie ustawiał położenie punktu na początku układu współrzędnych (`0`,`0`), tak jak na listingu 2.12.

Listing 2.12. *Deklaracja dwóch konstruktorów o tej samej nazwie*

```
class Point {
  private int x;  // położenie na osi 0X
  private int y;  // położenie na osi 0Y
  // pierwszy konstruktor klasy Point
  Point() {
    x = 0;
    y = 0;
  }
  // drugi konstruktor klasy Point
  Point(int newX, int newY) {
    x = newX;
    y = newY;
  }
  //...
}
```

W tym przypadku nie jest możliwe ominięcie przeciążenia ze względu na konieczność zastosowania dla obu konstruktorów tej samej nazwy (czyli `Point`).

Udogodnienie wprowadzone przez mechanizm przeciążania metod wprowadza bocznymi drzwiami możliwość zastosowania metod nazywających się tak samo jak klasy. Na pierwszy rzut oka wydaje się, że będziemy mieli do czynienia z konstruktorem, choć w rzeczywistości będzie to zwykła metoda o nazwie takiej jak klasa. W szczególnym przypadku możemy więc zastosować konstrukcję pokazaną na listingu 2.13.

Listing 2.13. *Deklaracja metody i klasy o tej samej nazwie*

```
class Klasa {
  Klasa(){ /* konstruktor Klasa*/ }
  // metoda o nazwie Klasa:
  public int Klasa(int i) { return i; }
}
```

Użycie konstruktora i metody (trochę wybiegam tu w przyszłość, lecz mam nadzieję, że mi to wybaczysz) będzie miało postać jak na listingu 2.14.

Listing 2.14. *Użycie konstruktora i metody o tej samej nazwie*

```
// wykorzystanie konstruktora
Klasa k = new Klasa();
// wykorzystanie metody
int i = k.Klasa(11);
```

Jakkolwiek taka konstrukcja jest możliwa, nie polecam jej ze względu na wysoką podatność na generowanie błędów w tym miejscu. Jeśli użyjemy kompilatora z opcją pedantycznej kompilacji (na przykład *JIKES*), w czasie przetwarzania tej konstrukcji zgłosi on co do niej zastrzeżenie, lecz wykona proces kompilowania. Oto przykład błędnego użycia zaprezentowanej klasy:

```
Klasa k = new Klasa(11);
```

Na pierwszy rzut oka wydaje się, że wszystko jest w porządku. Odwołanie takie nie skutkuje jednak wywołaniem konstruktora, tylko metody. Dlatego jak wcześniej napisałem, nie powinno się stosować tej konstrukcji, chyba że szczególne zależy nam na zaciemnieniu struktury programu (na przykład w celu utrudnienia dekompilacji).

Warto zauważyć, że stosowanie konstruktora i metody o tej samej nazwie jest pewną nieścisłością w stosunku do kwestii przeciążania metod. Zwykłe metody nie są rozróżniane na podstawie typu zwracanego wyniku. Natomiast konstruktor i metoda o tej samej nazwie i tym samym zestawie parametrów są dla kompilatora różne. Dzięki temu możliwe jest totalne zaciemnienie kodu klasy, jak to pokazałem na listingu 2.15.

Listing 2.15. *Metoda udająca domyślny konstruktor*

```
class Klasa {
  public int Klasa() {
    return 1;
  }
}
```

Pokazana na listingu 2.15 metoda umożliwia napisanie fragmentu programu zaprezentowanego na listingu 2.16.

Listing 2.16. *Użycie konstruktora i metody o takiej samej liście parametrów*

```
// domyślny, bezparametrowy konstruktor
Klasa k = new Klasa();
// metoda zwracająca wynik typu int
int i = k. Klasa();
```

Jakkolwiek są osoby, które lubują się w stosowaniu takich konstrukcji, twierdząc że jest to esencja programowania obiektowego, ja uważam to za złe rozwiązanie.

Na marginesie przeciążenia konstruktorów można pokazać użycie słowa kluczowego `this` w formie innej, niż pokazano w paragrafie 2.1.6. „Słowo kluczowe this". Otóż odwołanie do samego tego słowa jest równoważne odwołaniu do konstruktora klasy, w której się znajdujemy. Oczywiście ma to sens jedynie w przypadku, gdy klasa ma kilka przeciążonych konstruktorów i jeden z nich, zamiast jawnie wykonywać jakiś blok instrukcji, odwołuje się do innego. Na listingu 2.17 przedstawiam ten sam fragment klasy `Point`, jednak z użyciem wywołania jednego z konstruktorów przez drugi za pomocą słowa `this`.

Listing 2.17. *Użycie słowa this zamiast konstruktora*

```
class Point {
  private int x;  // położenie na osi 0X
  private int y;  // położenie na osi 0Y
  // pierwszy konstruktor klasy Point
  Point() {
    this(0,0);
  }
  // drugi konstruktor klasy Point
  Point(int newX, int newY) {
```

```
      x = newX;
      y = newY;
    }
    //...
  }
```

Takie zastosowanie `this` rzeczywiście upraszcza kod źródłowy i czyni go bardziej przejrzystym.

2.1.9. Dziedziczenie

Zanim przejdziemy dalej, należy wprowadzić pojęcie dziedziczenia. Jak zwracałem na to uwagę w poprzednim rozdziale, dziedziczenie jest jedną z podstawowych cech programowania obiektowego. Mechanizm ten umożliwia rozszerzanie możliwości wcześniej utworzonych klas bez konieczności ich ponownego tworzenia. Zasada dziedziczenia w Javie ma za podstawę założenie, że wszystkie klasy dostępne w tym języku bazują w sposób pośredni lub bezpośredni na klasie głównej o nazwie `Object`. Wszystkie klasy pochodzące od tej oraz każdej innej są nazywane, w stosunku do tej, po której dziedziczą, podklasami. Klasa, po której dziedziczy własności dana klasa, jest w stosunku do niej nazywana nadklasą. Jeśli nie deklarujemy w żaden sposób nadklasy, tak jak jest to pokazane w przykładowej deklaracji klasy `Point`, oznacza to, że stosujemy domyślne dziedziczenie po klasie `Object`. Formalnie deklaracja klasy `Point` mogłaby mieć postać zaprezentowaną na listingu 2.18.

Listing 2.18. *Dziedziczenie po klasie głównej*

```
class Point extends Object {
  // ...
  // ciało klasy Point
  // ...
}
```

Wytłuszczony fragment listingu 2.18 deklaruje dziedziczenie po klasie `Object`. Jak wcześniej pisałem, jest ono opcjonalne, to znaczy, że jeśli go nie zastosujemy, `Point` również będzie domyślnie dziedziczył po `Object`.

Przedstawiony sposób jest używany w przypadku dziedziczenia po innych klasach, tak jak na listingu 2.19.

Listing 2.19. *Praktyczne użycie dziedziczenia*

```
class Figura extends Point {
  ...
}

class Wielokat extends Figura {
  ...
}
```

W przykładzie tym klasa `Wielokat` dziedziczy po klasie `Figura`, która z kolei dziedziczy po `Point`, a ta po `Object`. W Javie nie ma żadnych ograniczeń co do zagnieżdżania poziomów dziedziczenia. Poprawne więc będzie dziedziczenie na stu i więcej poziomach. Jakkolwiek takie głębokie dziedziczenie jest bardzo atrakcyjne w teorii, w praktyce wiąże się z niepotrzebnym obciążaniem zarówno pamięci, jak i procesora. To samo zadanie zrealizowane za pomocą płytszej struktury dziedziczenia będzie działało szybciej aż z trzech powodów:

- Wywołanie metod będzie wymagało mniejszej liczby poszukiwań ich istnienia w ramach kolejnych nadklas.
- Interpreter będzie musiał załadować mniej plików z definicjami klas (i mniej będzie ich później obsługiwał).
- System operacyjny (a przez to również interpreter Javy) ma więcej wolnej pamięci, a przez to pracuje szybciej.

Ponadto w przypadku apletów możemy liczyć na szybsze ładowanie się strony do przeglądarki, a więc będzie to kolejna pozytywna strona.

Poza dziedziczeniem w dowolnie długim łańcuchu od klasy głównej do najniższej klasy potomnej w niektórych językach programowania (na przykład C++) istnieje wielokrotne dziedziczenie jednocześnie i równorzędnie po kilku klasach. W Javie jest to niemożliwe, to znaczy w definicji każdej klasy może wystąpić co najwyżej jedno słowo `extends`. Zamiast wielokrotnego dziedziczenia w Javie dostępny jest mechanizm interfejsów opisany w podrozdziale 2.4. „Interfejsy”.

2.1.10. Inicjator klasy i obiektu

Wróćmy do rozważań na temat tego, co się dzieje w początkach życia obiektu. Poza konstruktorem Java udostępnia dwa inne mechanizmy wspomagające inicjację i tworzenie zarówno klas, jak i obiektów. Inicjator klasy jest to blok instrukcji wykonywany tylko raz, po załadowaniu przez JVM pliku z klasą przed pierwszym użyciem (jednak klasa musi być użyta, żeby blok ten wykonał się — sama deklaracja użycia bez inicjacji nie gwarantuje wykonania inicjatora klasy). Blok ten, zawarty między dwoma nawiasami klamrowymi, musi być poprzedzony słowem kluczowym `static` (dokładne znaczenie tego modyfikatora zostanie wyjaśnione dalej w tym rozdziale). Poza tym klasa może zawierać również inicjator obiektu, czyli egzemplarza klasy. Jest to też blok instrukcji zamknięty w nawiasach klamrowych, ale bez żadnego kwalifikatora. Zarówno inicjator klasy, jak i obiektu może wystąpić w każdej klasie kilkukrotnie. Jeśli jest ich większa ilość, zostaną wykonane zgodnie z kolejnością pojawienia się w kodzie źródłowym. Poniżej przedstawiony jest listing 2.20 z apletem, który zawiera różne elementy inicjacyjne wraz z instrukcjami umożliwiającymi sprawdzenie kolejności ich wykonywania się (szczegóły działania apletów zostaną wprowadzone w rozdziale 3. „Aplet jako obiekt na stronie HTML”). W komentarzach zaznaczono priorytet ważności od 1 (najważniejsze, wykonywane najpierw) do 3 (najmniej ważne, wykonywane na końcu). Elementy inicjacyjne o tym samym priorytecie wykonywane są zgodnie z kolejnością wystąpienia. Warto zauważyć, że inicjator obiektu i inicjator pól obiektu mają ten sam priorytet i wykonywane są zgodnie z kolejnością wystąpienia w klasie.

Listing 2.20. *Inicjatory klasy*

```
import java.applet.*;

public class Applet2 extends Applet {
  int i = setInt(1);  // priorytet 2

  static { // priorytet 1
    System.err.println("class init");
  }

  public Applet2() { // priorytet 3
    System.err.println("konstruktor");
  }

  { // priorytet 2
    System.err.println("instance init");
  }

  // dodatkowa funkcja wyświetlająca
  private int setInt(int i) {
    System.err.println("set int: " + i);
    return i;
  }

  int j = setInt(2); // priorytet 2
}
```

Zaprezentowany aplet generuje na konsolę Javy w przeglądarce zestaw komunikatów pokazanych na rysunku 2.1.

Rysunek 2.1. *Wydruk generowany przez program 2.20*

```
class init
set int: 1
instance init
set int: 2
konstruktor
```

Rozszerzenie tego tematu znajduje się w paragrafie 2.1.11. „Kolejność inicjacji klas".

Istnienie inicjatorów może uprościć tworzenie niektórych klas, zwłaszcza tych bardziej skomplikowanych. Wyobraźmy sobie, że tworzymy klasę z dużą liczbą przeciążonych konstruktorów. W każdym z nich poza działaniami związanymi z ich charakterystyczną pracą zależną od zestawu przekazanych parametrów należy wykonać czynności inicjujące — wspólne dla wszystkich konstruktorów. Można by to zrobić poprzez jawne wywołanie wspólnej metody inicjującej, jak to pokazałem na listingu 2.21.

Listing 2.21. *Jawne użycie jednej metody inicjującej*

```
class MultiKonstruktor {
  public MultiKonstruktor() { wspolnyInit(); }
  public MultiKonstruktor(int i) {
    wspolnyInit();
    // przetwarzanie i
  }
  public MultiKonstruktor(String s) {
```

```
    wspolnyInit();
    // przetwarzanie s
  }
  void wspolnyInit() { /*inicjacja*/ }
  }
}
```

Działanie takie jest poprawne, ale wymaga od nas pamiętania o dodaniu jednego wywołania metody w każdym kolejnym konstruktorze oraz zwiększa wielkość kodu wynikowego. Każdy konstruktor zawiera bowiem wywołanie tej metody. Zastosowanie inicjatora obiektu uwalnia nas od konieczności każdorazowego dodawania tego wywołania oraz usuwa ten fragment kodu z konstruktora. Biorąc pod uwagę to, że aplet jest programem ładowanym do komputera użytkownika przez internet (czasami, gdy korzysta się z dość wolnej linii telefonicznej), każde kilkadziesiąt czy kilkaset bajtów może być ważne (ten sam problem dotyczy telefonów komórkowych, które są najczęściej wyposażone w bardzo małą pamięć). Jeśli tylko jest to możliwe, wspólny kod inicjacyjny powinno umieszczać się w bloku inicjacyjnym. Dla klasy z listingu 2.21 rozwiązanie takie miałoby postać zaprezentowaną na listingu 2.22.

Listing 2.22. *Kod inicjacyjny we wspólnym bloku*

```
class MultiKonstruktor {
  public MultiKonstruktor(int i) {
    // przetwarzanie i
  }
  public MultiKonstruktor(String s) {
    // przetwarzanie s
  }
  { /* tu wspólna inicjacja */ }
}
```

Warto zauważyć, że zaoszczędziliśmy nie tylko na dwóch wywołaniach metody, ale mogliśmy zrezygnować nawet z bezparametrowego konstruktora, którego jedyną pracą było uruchomienie procedury wspólnej inicjacji. Tak więc argumentem na stosowanie takiego rozwiązania jest nie tylko dbanie o użytkowników wdzwaniających się do internetu, ale również elegancja i prostota, które prawie zawsze skutkują zmniejszeniem liczby popełnianych błędów.

Warto zwrócić uwagę na to, że (inaczej niż w przypadku apletów) w przypadku programów, które mogą egzystować samodzielnie, przed jakąkolwiek inicjacją obiektu przeprowadzana jest inicjacja statyczna, a następnie wykonywana jest metoda `main` i to dopiero w niej może być tworzony egzemplarz obiektu. Dzieje się tak, gdyż metoda ta jest statyczna i publiczna i może być użyta przed stworzeniem egzemplarza obiektu (dokładniejsze wyjaśnienia co do natury metod statycznych zostaną zamieszczone dalej w tym rozdziale). Maszyna wirtualna Javy jest tak skonstruowana, że wywołuje tę metodę jako pierwszą. W przypadku samodzielnych programów należy więc wziąć pod uwagę ten aspekt i zmodyfikować kolejność wywoływania bloków inicjacyjnych.

2.1.11. Kolejność inicjacji klas

Powróćmy do kwestii kolejności, w jakiej wykonuje się inicjacja klas. Załóżmy, że nasze dziedziczące klasy będą skonstruowane według schematu pokazanego na listingu 2.23.

Listing 2.23. *Bloki inicjujące w klasach dziedziczących*

```
class A {
  A() { System.err.println("konstruktor A"); }
  { System.err.println("inicjator obiektu A"); }
  static { System.err.println("inicjator klasy A"); }
}

class B extends A {
  B() {System.err.println("konstruktor B"); }
  { System.err.println("inicjator obiektu B"); }
  static { System.err.println("inicjator klasy B"); }
}
```

Pierwsze użycie pokazanej na listingu 2.23 klasy `B`, przy założeniu, że wcześniej nie używaliśmy klasy `A`, spowoduje wyświetlenie kolejnych napisów, które pokazane są na rysunku 2.2.

Rysunek 2.2. *Wydruk generowany przez program z listingu 2.23*

```
inicjator klasy A
inicjator klasy B
inicjator obiektu A
konstruktor A
inicjator obiektu B
konstruktor B
```

Jak więc widać, najpierw — w kolejności dziedziczenia — inicjowane są klasy. Po nich następuje sekwencja charakterystyczna dla inicjacji obiektów typu klasy nadrzędnej. Obiekty te nazywam egzemplarzami wirtualnymi. W praktyce JVM rezerwuje od razu pamięć na cały rzeczywisty obiekt, jednak tworzenie go jest przeprowadzane sekwencyjnie. Najpierw inicjowany jest wirtualny obiekt bazowy, później uzupełniane są braki przez inicjacje kolejnych klas pochodnych. Inicjacja obiektów wirtualnych, zaznaczona na rysunku 2.2 kursywą, jest blokiem nie do rozłączenia. Jeśli klasa używająca `B` ma blok inicjujący klasę, to zostanie on wykonany przed inicjatorem klasy `A`. Sytuacja nie zmieni się, jeśli w pierwszej linii konstruktora klasy `B` dodamy jawne wywołanie konstruktora klasy nadrzędnej (z użyciem słowa `super`), czyli klasy `A`, tak jak pokazałem to na listingu 2.24. Odwołanie do klasy nadrzędnej dokładnie zostanie wyjaśnione w paragrafie 2.1.14. „Odwołanie do klas nadrzędnych”.

Listing 2.24. *Rozszerzenie inicjacji klas z listingu 2.23*

```
class B extends A {
  B() {
    super();
    System.err.println("konstruktor B");
  }
  { System.err.println("inicjator obiektu B"); }
  static { System.err.println("inicjator klasy B"); }
}
```

Zgodnie z tym, co wcześniej napisałem, konstruktor klasy nadrzędnej wywoływany jest domyślnie jako pierwsze działanie konstruktora danej klasy.

Z sekwencyjnym tworzeniem i inicjacją obiektów dziedziczących związany jest pewien ważny problem. Pokażę go na przykładzie klasy nadrzędnej o postaci zaprezentowanej na listingu 2.25.

Listing 2.25. *Klasa, po której trudno dziedziczyć*

```
public class A {
  private Object o;
  public A(Object o) {
    this.o = o;
  }
}
```

Klasa taka nie umożliwia utworzenia dziedziczenia w postaci zaprezentowanej na listingu 2.26.

Listing 2.26. *Błędne dziedziczenie po klasie z listingu 2.25*

```
public class B extends A {
  Object oo = new Object();
  public B() {
    super(oo);  // błąd
  }
}
```

W takim przypadku konstruktor klasy nadrzędnej, reprezentowany przez linię `super(oo)`, jest wywoływany, zanim utworzony zostanie egzemplarz `oo` klasy `Object`. Przed konstruktorem klasy nadrzędnej nie może bowiem być wykonywana żadna inna akcja poza ewentualnym wywołaniem innego konstruktora przeciążonego, który wywoła konstruktor `super`. Podobnie niepoprawna będzie też konstrukcja pokazana na listingu 2.27.

Listing 2.27. *Błędne dziedziczenie po klasie z listingu 2.25*

```
public class B extends A {
  public B() {
    super(this);  // błąd
  }
}
```

Błąd wynika z tego, że egzemplarz obiektu tej klasy, reprezentowany przez `this`, będzie znany dopiero po jego utworzeniu, a więc najwcześniej po zakończeniu pracy konstruktora klasy nadrzędnej.

Mimo takiego podejścia, to znaczy kolejnego tworzenia egzemplarzy obiektów klas dziedziczących, metody tych klas są formalnie dostępne w obiektach nawet przed ich utworzeniem. Może to spowodować powstanie błędnego, przynajmniej w naszym pojęciu, działania niektórych konstruktorów. Na listingu 2.28 zaprezentowane zostały dwie klasy — `A` i `B`. W klasach tych metoda `doSth` została zadeklarowana i wykorzystana niepoprawnie.

Listing 2.28. *Błędne deklaracje metody w klasach dziedziczących*

```
class A {
  A() {
    doSth();
  }
  void doSth() {
    System.err.println("A.doSth");
  }
}

class B extends A {
  B(){
    super();
    doSth();
  }
  void doSth() {
    System.err.println("B.doSth");
  }
}
```

Jeśli zadeklarujemy użycie klasy `B` i utworzenie z niej obiektu

```
B b = new B();
```

otrzymamy niespodziewany dla większości osób wynik (wydruk na konsoli Javy):

```
B.doSth
B.doSth
```

Zaobserwowany efekt działania jest jednak poprawny. Jest on skutkiem działania polimorfizmu. W zaprezentowanym przykładzie konstruktor w klasie `A` wywołuje metodę `doSth` tworzonego obiektu (czyli klasy `B`). Tak więc to metodę tej klasy wywoła konstruktor klasy `A`, mimo iż twórca miał zapewne co innego na myśli. Aby wywołanie `doSth` zawsze dotyczyło własnej klasy, metoda ta musi być prywatna (modyfikator `private`). Warto na to zwrócić uwagę, gdyż może to być przyczyną wielu podobnych nieporozumień. Inicjacja klasy:

```
B b = new B(3);
```

której definicja pokazana jest na listingu 2.29, może przynieść nieoczekiwany efekt.

Listing 2.29. *Użycie metod w konstruktorze*

```
public class A {
  public A() {
    System.out.println("wewnątrz konstruktora A");
    doSth();
  }
  public void doSth() {
    System.out.println("nic nie robię");
  }
}

public class B extends A {
  private int p1;
  public B(int p) {
```

```
    p1 = p;
    System.out.println("wewnątrz konstruktora B");
  }
  public void doSth() {
    System.out.println("p1=" + p1);
    // obliczenia z użyciem p1
  }
}
```

Pozornie nieoczekiwany wynik działania klasy z listingu 2.29 zaprezentowałem na rysunku 2.3.

Rysunek 2.3. *Wydruk generowany przez program 2.29*

```
wewnątrz konstruktora A
p1=0
wewnątrz konstruktora B
```

Czyli tak jak napisałem wcześniej, przed uruchomieniem konstruktora klasy `B` (czyli przed powstaniem egzemplarza tej klasy) system potrafi już użyć jego metody `doSth`. Oczywiście skoro dzieje się to przed uruchomieniem konstruktora `B`, prywatne pole `p1` nie jest jeszcze zainicjowane, więc jest równe zero. Więcej na temat polimorfizmu znajdziesz w paragrafie 2.2.6. „Efekty polimorfizmu".

2.1.12. Destruktor

Podchodząc formalnie do specyfikacji JVM, można powiedzieć, że klasy Javy nie wymagają stosowania specjalizowanych metod zwalniających zajętą przez siebie pamięć. Wynika to z założenia przyjętego w czasie tworzenia tego systemu, a mianowicie braku jawnego zwalniania pamięci zajmowanej przez obiekty. Podejście klasyczne stosowane w C++ i Object Pascalu zakłada, że to programista, w chwili kiedy uznaje to za stosowne lub gdy wymusza to struktura programu, zwalnia pamięć, korzystając z jawnych funkcji systemowych bądź specjalizowanych metod wbudowanych w obiekty (destruktory). Podejście takie ma tę zaletę, że umożliwia zwalnianie pamięci natychmiast, kiedy jest to możliwe. Ma jednak tę wadę, że może powodować próby odwołania się do obiektu omyłkowo i przedwcześnie zwolnionego. W Javie zrezygnowano więc z tego mechanizmu. Nie oznacza to jednak braku metody, która byłaby namiastką destruktora. Jest nią metoda `finalize` wywoływana w trakcie czyszczenia pamięci. Ma ona za zadanie wykonać wszystkie konieczne działania przed całkowitym zastopowaniem oraz zwolnieniem pamięci przez obiekt, do którego należy (jednak zwalnianie pamięci nie należy już do obowiązków tej metody). Typowa deklaracja funkcji kończącej działanie obiektu powinna mieć postać pokazaną na listingu 2.30.

Listing 2.30. *Przykładowa deklaracja destruktora*

```
public void finalize() {
  zatrzymajWatki();
  usunPowiazania();
}
```

W ramach działania tej metody poza zatrzymaniem wątków przynależnych do tego obiektu (o wątkach napiszę szerzej w rozdziale 6. „Programowanie wielowątkowe") powinno się dokonać wszelkiego sprzątania po działającym obiekcie. Jeśli obiekt sam siebie wyświetlał na ekranie, należy ten obraz usunąć (w przeciwnym razie może pozostać tam na zawsze). Jeśli obiekt był wykorzystywany jako nasłuchujący w procesie zarządzania zdarzeniami, należy go usunąć z kolejki obsługi zdarzeń obiektu, który był przez niego obsługiwany. Jeśli z działaniem obiektu związana była obsługa plików bądź dostęp do baz danych, należy zakończyć tę obsługę, aby pliki nie były blokowane przez nieistniejący już obiekt. Po zakończeniu tej metody sterowanie oddawane jest do wirtualnej maszyny Javy, która wykonuje fizyczne zwalnianie pamięci (*garbage collection*). Nie wszystkie maszyny dokonują tego natychmiast po zakończeniu pracy destruktora, ale nie wpływa to na sposób działania programu.

2.1.13. Przykrywanie metod

W miarę jak tworzymy kolejne klasy pochodne od innych, pojawia się często sytuacja, że musimy zastąpić działanie pewnej metody inną, o takiej samej nazwie i tym samym zestawie parametrów. Działanie takie nazywa się przykrywaniem, pokrywaniem bądź nadpisywaniem metod. Na listingu 2.31 przedstawiam 3 kolejne klasy w łańcuchu dziedziczenia, które przykrywają po kolei swoją metodę `info`.

Listing 2.31. *Trzy klasy pokrywające jedną metodę*

```
class A {
  public void info() {
    System.err.println("Jestem w klasie A");
  }
}

class B extends A {
  public void info() {
    System.err.println("Jestem w klasie B");
  }
}

class C extends B {
  public void info() {
    System.err.println("Jestem w klasie C");
  }
}
```

Odwołanie do metody `info` obiektu klasy `C` powoduje wyświetlenie na konsoli Javy w przeglądarce napisu:

```
Jestem w klasie C
```

Nie jest to nic dziwnego, bo w klasie `C` metoda `info` przykryła swoją odpowiedniczkę z klasy `B`, która z kolei przykryła swoją odpowiedniczkę z klasy `A`. Należy jednak pamiętać, że domyślnie wszystkie metody w Javie są wirtualne. Związana jest z tym kwestia polimorfizmu, to znaczy wywołanie metody ustalane jest na podstawie rzeczywiście używanego obiektu, a nie jego deklaracji typu (chyba że są to metody statyczne). Więcej na ten temat znajdziesz w paragrafie 2.2.6. „Efekty polimorfizmu".

2.1.14. Odwołanie do klas nadrzędnych

Nie zawsze jednak przykrywanie metod jest pożądanym efektem. Mogą zajść sytuacje, w których mimo tego że mamy przykryte pewne metody, chcielibyśmy odwołać się do nich, gdyż wiemy, że wykonują przydatne dla nas działanie. Dokonuje się tego poprzez referencję `super` (jest to to samo słowo kluczowe, którego używa się do wywołania konstruktora klasy nadrzędnej). Na listingu 2.32 przedstawiona jest zmodyfikowana klasa `C`, która pokaże, jak odwołać się do metody swojej klasy nadrzędnej, mimo że metoda ta jest przykryta.

Listing 2.32. *Sposób odwołania do metody klasy nadrzędnej*

```
class C extends B {
  public void show() {
    info();
    super.info();
    // super.super.info();   <- błąd kompilacji
  }
  public void info() {
    System.err.println("Jestem w klasie C");
  }
}
```

Efekt użycia w aplecie klasy `C` i jej metody `show` przedstawiony jest na rysunku 2.4.

Rysunek 2.4. *Wydruk generowany przez program 2.32*

```
Jestem w klasie C
Jestem w klasie B
```

Do poprawnej pracy trzeba było zastosować słowo kluczowe `super`, które umożliwia cofnięcie się wstecz łańcucha dziedziczenia, niestety tylko o jeden poziom (zaznaczyłem to listingu 2.32 wytłuszczonym komentarzem w kodzie). Jedyną możliwość cofnięcia się wstecz dalej niż poza najbliższą klasę jest stosowanie metod bądź klas statycznych. Możliwość taka nie wynika wtedy jednak z efektu cofania się w łańcuchu dziedziczenia, a z ogólnych cech modyfikatora `static`. Dalsze cofanie jest również możliwe w przypadku potraktowania obiektów jako refleksyjnych. Należy jednak zaznaczyć, że próba dalekiego i głębokiego cofania się świadczy o nie najlepszym projekcie, który posłużył do stworzenia naszej klasy. Jeśli więc czujemy potrzebę takiego głębokiego cofania się, powinniśmy, zamiast próbować ją zrealizować, ponownie przemyśleć projekt klasy.

Słowo `super` może być również wykorzystane w konstruktorze, bardzo podobnie jak `this`. Jednak w przypadku `this` chodziło o odwołanie do innego konstruktora w tej samej klasie, na tym samym poziomie dziedziczenia, ale z inną liczba parametrów (czyli do jednego z przeciążonych konstruktorów). W przypadku słowa `super` odwołujemy się do konstruktora klasy nadrzędnej. Sposób użycia pokazany jest we fragmencie konstruktora klasy `Test`:

```
public Test(int a, string b) {
  super(a, b);
  System.err.println(b);
}
```

W tym przykładzie konstruktor klasy `Test` wykonuje dokładnie to samo, co konstruktor jego klasy nadrzędnej, oraz dodatkowo wyświetla ciąg tekstowy przekazywany w parametrze `b`. Należy pamiętać, że odwołanie do konstruktora klasy nadrzędnej, jeśli występuje, musi być pierwszą instrukcją w konstruktorze. Jedynym wyjątkiem jest odwołanie do innego konstruktora w tej samej klasie z użyciem słowa `this`. Jeśli chcemy jawnie wywołać konstruktor nadklasy, wtedy ten drugi konstruktor, do którego odnosimy się przez `this`, musi już użyć odwołania `super` w pierwszym wierszu metody.

2.1.15. Odwołanie do pól klas nadrzędnych

Pewną namiastką odwołania do klas nadrzędnych na dowolną głębokość łańcucha dziedziczenia jest odwołanie do pól pokrywanych w kolejnych podklasach, co postaram się pokazać na listingu 2.33.

Listing 2.33. *Klasa dziedzicząca cechy bez modyfikacji*

```
class A {
  String info = "klasa A";
}

class B extends A { }
```

Formalnie, jeśli pole nie jest prywatne, konstrukcja zaprezentowana na listingu 2.33 oznacza, że klasa `B` posiada pole o nazwie `info` typu `String` zawierające tekst `"klasa A"`. Nie stosuje się więc ponownego definiowania tego pola, nawet jeśli chcemy je wykorzystać w inny sposób, niż pierwotnie zakładaliśmy w klasie `A`. Możemy jednak zastosować pokrycie pól w klasie, tak jak pokazano to na listingu 2.34.

Listing 2.34. *Pokrywanie pól w klasie dziedziczącej*

```
class A {
  String info = "klasa A";
}

class B extends A {
  String info = "klasa B";
}

class C extends B {
  String info = "klasa C";
  public void show() {
    System.err.println(((A)this).info);
  }
}
```

Jeśli wywołamy metodę `show` obiektu klasy `C` na konsoli Javy, zobaczymy napis:

```
klasa A
```

Takie zachowanie, różniące się od zachowania metod, wynika z faktu, że odwoływanie się do metod zależy od faktycznego typu obiektu, którego w danym momencie używamy. Natomiast w stosunku do pól używany jest zadeklarowany typ obiektu. Jest to

wynik dobrze przemyślanej metodologii realizacji idei obiektowej i nazywa się przesłanianiem pól. Mechanizm ten pojawił się jako rozszerzenie możliwości dziedziczenia. Tworząc nową klasę, jej autor może stworzyć nowe pole o nazwie wykorzystywanej już w klasie nadrzędnej (specjalnie bądź przez pomyłkę). Gdyby tak utworzone pole uniemożliwiało dostęp do pola klas nadrzędnych (tak jak to się dzieje w przypadku metod), wewnętrzne odwołania do pól klasy nadrzędnej zaczęłyby się teraz odnosić do pól nowej klasy, być może nieznanej nawet autorowi klasy nadrzędnej. Mogłoby to skutecznie zakłócić działanie nadklasy, a co za tym idzie i podklasy, w której zadeklarowano pole o używanej już nazwie. Biorąc pod uwagę fakt, że Java nie ogranicza w żaden sposób głębokości dziedziczenia, mogłoby się okazać, że w pewnym momencie zabraknie nam logicznych nazw do określenia kolejnych pól.

Problem ten jest kolejnym argumentem przemawiającym za stosowaniem ortodoksyjnie pojętej hermetyzacji klas, to znaczy maksymalnie dużą liczbę pól powinno deklarować się jako prywatne. Problem taki nie miałby wtedy prawa się pojawić.

Należy również pamiętać, że mimo stosowania referencji do pól zgodnych z deklarowanym typem, problem nie pojawi się w przypadku stosowania podręcznikowej konstrukcji klas `B` i `C` pokazanej na listingu 2.35.

Listing 2.35. *Inicjacja pól w klasach dziedziczących w inicjatorze*

```
class B extends A {
  { info = "klasa B"; }
}

class C extends B {
  { info = "klasa C"; }
  public void show() {
    System.err.println(((A)this).info);
  }
}
```

Stosując wyłącznie inicjację pola `info` właściwą wartością, bez deklarowania tego pola na nowo, po użyciu obiektu klasy `C` na konsoli zobaczymy napis:

```
klasa C
```

Wynik będzie więc zgodny z oczekiwaniami.

2.1.16. Klasy abstrakcyjne

Klasy abstrakcyjne to szczególny rodzaj klas, który przenosi obowiązek implementacji niektórych bądź wszystkich metod do swoich podklas. Mechanizm ten jest przydatny w przypadku, gdy część zachowania jest wspólna dla wszystkich klas pochodnych, a część jest charakterystyczna tylko dla klas pochodnych, natomiast dla wspólnej nadklasy nie miałaby żadnego sensu. W takim przypadku części tej nie implementuje się w klasie nadrzędnej z założeniem, że zostanie to zaimplementowane w podrzędnej. Wyróżnikiem tego, że klasa jest abstrakcyjna, jest słowo kluczowe `abstract` określające

klasę zawierającą przynajmniej jedną „pustą” deklarację oraz wszystkie metody, które są wyłącznie deklarowane. Przykład klasy abstrakcyjnej pokazany jest na listingu 2.36. Fakt, że klasa jest abstrakcyjna,, wynika z braku implementacji metody `getInfo`.

Listing 2.36. *Przykładowa klasa abstrakcyjna*

```
abstract class Obliczenia {
  abstract public String getInfo();
  public void doMath() {
    // ciało metody
  }
}
```

W przypadku próby dziedziczenia po takiej abstrakcyjnej klasie wszystkie metody abstrakcyjne muszą zostać pokryte. W przeciwnym wypadku klasa dziedzicząca również będzie abstrakcyjna. Klasa abstrakcyjna nie może posłużyć jako wzorzec do utworzenia egzemplarza klasy. Tak więc zastosowanie klasy abstrakcyjnej to sposób na wymuszenie implementacji pewnych metod w programie używającym naszych klas. Poza klasami abstrakcyjnymi podobnym mechanizmem są interfejsy, które zostaną opisane dalej w tym rozdziale. Przeprowadzę tam krótkie porównanie klas abstrakcyjnych i interfejsów oraz przypadków wskazanych do ich zastosowania. Jedyną z zalet niedostępnych w interfejsach, którą ma mechanizm klas abstrakcyjnych, jest możliwość uczynienia abstrakcyjnymi niektórych klas mających wcześniej pełnoprawną funkcjonalność. Można tego dokonać, pokrywając w klasie pochodnej jedną lub więcej metod metodami abstrakcyjnymi, czyli bez implementacji. Dzięki temu możemy „usunąć” standardową implementację metod, wymuszając ich przyszłą, właściwszą dla danej klasy implementację.

2.2. Obiekty

Jak kilkukrotnie zwracałem już uwagę, klasy są tylko definicją i określeniem sposobu działania bądź przechowywania informacji. Aby klasa mogła być użyta, musimy utworzyć jej egzemplarz, którym jest obiekt. Istnieją dwa podstawowe sposoby tworzenia obiektów zdefiniowanych typów — tak aby były jawnie dostępne oraz w sposób niejawny. Poniżej przedstawię różnice między nimi. Ponadto spróbuję zwrócić uwagę na wszystkie ważne zagadnienia związane z tworzeniem i egzystowaniem obiektów w Javie. W dalszej części tego rozdziału pokażę również sposób tworzenia obiektów z użyciem refleksji (typy tych obiektów nie muszą być znane w czasie tworzenia programu).

2.2.1. Rozważania o adresie

Przed przystąpieniem do praktycznych przykładów tworzenia obiektów powinieneś, zwłaszcza jeśli znasz inne języki programowania, dowiedzieć się, co wiąże się z powstaniem pojedynczego egzemplarza klasy. W C++ czy Object Pascalu utworzenie egzemplarza zmiennej typu obiektowego wiązało się z przydzieleniem jej pewnego

obszaru pamięci (jawnie bądź nie, ręcznie bądź automatycznie). Zmienna typu obiektowego przechowywała wyłącznie wskazanie na obszar, w którym znajdował się obiekt, bądź na miejsce, gdzie znajdował się opis obiektu i dalsze adresy pól oraz metod. W Javie, wbrew obiegowym opiniom, jest dokładnie tak samo. Zmienna reprezentująca obiekt to rzeczywiście wskazanie (albo według innego nazewnictwa adres) na blok rozpoznawany przez maszynę wirtualną Javy jako zbiór wszystkich potrzebnych informacji do jednoznacznego zidentyfikowania obiektu i poprawnego używania go. Na tym jednak kończy się podobieństwo między wskazaniem na obiekt w Javie a w innych językach. W Javie nie da się wykonać kilku podstawowych operacji dostępnych w C++ czy Object Pascalu. Nie da się utworzyć samoistnego obszaru pamięci i wymusić potraktowanie go przez kompilator bądź interpreter jako obiekt. Nie da się dodać do wskaźnika liczby będącej wielokrotnością długości słowa maszynowego, aby odwołać się bezpośrednio do pola, metody bądź następnego obiektu w pamięci. Najwięcej trudności sprawia natomiast przyzwyczajenie się do tego, że obiekt, który nie jest już przez nas używany, nie może być zwolniony na nasze żądanie, tylko musimy zdać się przy tym na łaskę JVM. Jednak te trzy ograniczenia sprawiają, że Java jest językiem, który wynosi bezpieczeństwo kilka kroków naprzód, przed inne języki. Dostajemy zalety pracy na wskaźnikach i pozbawieni jesteśmy możliwości świadomego (bądź nie) zniszczenia sobie działającego programu. W dalszej części książki będę używał zamiennie nazwy obiekt, egzemplarz, adres bądź wskazanie. Nie powinno Cię to jednak zwieść. Będę cały czas mówił o tym samym — o symbolicznej reprezentacji pojedynczego egzemplarza klasy.

Zanim przejdziemy do praktycznego stosowania obiektów, chciałbym przypomnieć, że w Javie udostępniony jest mechanizm kompatybilności typów obiektowych wynikających z dziedziczenia klas. To znaczy możemy zadeklarować obiekt jako egzemplarz klasy dokładnie tego samego typu, jaki tworzymy, albo jako egzemplarz dowolnej klasy nadrzędnej tego obiektu, czyli takiej, która znajdowała się w łańcuchu jego dziedziczenia. W skrajnym przypadku możemy więc deklarować w Javie wszystkie obiekty jako egzemplarze klasy Object, po której dziedziczą wszystkie klasy w Javie. Takie podejście, jakkolwiek formalnie słuszne i niezmniejszające (dzięki mechanizmowi polimorfizmu) funkcjonalności programu, w praktyce jest bardzo uciążliwe. Wymaga bowiem od programisty ciągłego umieszczania referencji typu obiektu w momencie odwoływania się do pól czy metod istniejących wyłącznie w klasach pochodnych. Interpreter zna ściśle typ obiektu w czasie uruchomienia programu, ale kompilator w czasie kompilacji nie jest tego w stanie stwierdzić. Tak więc prościej jest używać zmiennych właściwego typu. Możliwość stosowania typu nadrzędnego powinno pozostawić się wyłącznie do sytuacji wyjątkowych, kiedy jest to wytłumaczone potrzebą uproszczenia bądź zwiększenia funkcjonalności projektu.

2.2.2. Jawne użycie obiektów

Rozpoczęcie tworzenia obiektu należy zacząć od deklaracji użycia egzemplarza konkretnego typu. Poza typem i modyfikatorem musi ona zawierać unikalną nazwę, która umożliwi przyszłą identyfikację obiektu. Przykład takiej deklaracji podany jest poniżej:

```
Button b;
```

W przykładzie tym pod nazwą b będzie się w przyszłości ukrywał obiekt klasy Button (należący do standardowej biblioteki AWT). Będzie, ponieważ bezpośrednio po deklaracji zmienna b nie reprezentuje jeszcze żadnego konkretnego wskazania. Sama deklaracja powoduje przypisanie zmiennej wartości null. To znaczy kompilator wie już, że zmienna b jest odpowiedniego typu i możemy z jej pomocą odwołać się do pól i metod klasy Button, jednak brak nam jeszcze samego obiektu. Dopiero konstrukcja:

```
b = new Button();
```

fizycznie tworzy jej pojedynczy egzemplarz. W praktyce operator new rezerwuje pamięć potrzebną dla danego obiektu, inicjuje właściwe pola w tym egzemplarzu oraz zwraca do zmiennej b adres zarezerwowanego bloku pamięci. Warto wiedzieć, że wielkość i sposób rezerwacji pamięci nie zależy od typu zmiennej zadeklarowanej wcześniej, lecz od konstruktora. To on, a nie deklaracja zmiennej, determinuje wynik tworzenia egzemplarza. Ponadto powinno się też pamiętać o tym, że wywołanie new skojarzone jest również ze sprawdzeniem, czy wskazana klasa była wcześniej używana i czy jest załadowana do pamięci. Jeśli nie, jest ona znajdowana na dysku bądź w sieci i ładowana do pamięci. Po załadowaniu, jeśli w klasie występuje jej inicjator, uruchamiany jest jego kod. Dopiero po tym procesie następuje przejście do faktycznej realizacji zadań skojarzonych z operatorem new.

Poza konstrukcją pokazaną powyżej, czyli osobno deklarujemy zmienną typu obiektowego i osobno tworzymy egzemplarz klasy, możemy te operacje wykonać w sposób spójny, umieszczając je w jednym wierszu programu:

```
Button b = new Button();
```

Z konwencji takiej korzysta się najczęściej wtedy, gdy zmienna obiektowa używana jest wyłącznie w obrębie jednej metody bądź obiekt może być utworzony globalnie i deklaracja zmiennej znajduje się poza ciałem jakiejkolwiek metody. Jeśli jednak obiekt ma być dostępny w całej klasie, lecz jego inicjacja uzależniona jest od innych dodatkowych czynników bądź czynności, należy zastosować rozłączną deklarację i tworzenie obiektu. Najczęściej obiekt tworzy się wtedy w konstruktorze, tak jak na listingu 2.37.

Listing 2.37. *Rozłączne deklarowanie i tworzenie obiektu*

```
class DemoPanel extends Panel {
  Button b;
  public DemoPanel() {
    b = new Button();
    b.setLabel("przycisk");
  }
}
```

Moment tworzenia obiektów jest dobry, aby przypomnieć o kwestii przeciążenia konstruktorów. Stosowany w tym paragrafie już trzy razy przykład bazował na wykorzystaniu bezparametrowego konstruktora, który tworzy przycisk ekranowy bez żadnego napisu. Dodanie napisu na tym przycisku wymaga użycia metody setLabel, co pokazano na ostatnim przykładzie. Obie operacje można połączyć, korzystając z innego konstruktora, który tworzy od razu przycisk z opisem:

```
b = new Button("przycisk");
```

Jak widać, jest to prostsze rozwiązanie, co wcale nie oznacza, że trzeba je zawsze stosować.

2.2.3. Kopiowanie obiektów

Egzemplarz obiektu identyfikowany przez nazwę symboliczną w wielu przypadkach może być traktowany dokładnie tak samo jak egzemplarz zmiennej prostej. Istnieją jednak wyjątki od tej reguły. Konstrukcja poprawna dla typów prostych:

```
int i, j;
i = 1;
j = i;
```

nie jest dla obiektów merytorycznie słuszna, chociaż pod względem formalnym jest jak najbardziej poprawna. Przypisanie jednej zmiennej prostej do innej skutkowało wyłącznie przypisaniem wartości jednej zmiennej do drugiej. Konstrukcja:

```
Button b = new Button();
Button c;
c = b;
```

powoduje wyłącznie przepisanie adresów, a więc zmienne `b` i `c` wskazują na jeden, ten sam obiekt. Nie mamy więc dwóch obiektów, jak to się pozornie wydaje, a tylko jeden, do którego mamy dwie referencje. Wykonując operację zmiany napisu na przycisku:

```
c.setLabel("przycisk 2");
```

tak naprawdę zmieniamy tekst w obu referencjach. Istnienie dwóch referencji do jednego obiektu możemy zredukować do jednej bez zmiany drugiej, przypisując do jednej z nich wartość pustą `null`:

```
b = null;
```

Po tej akcji zmienna `b` nie wskazuje już na żaden obiekt. Natomiast `c` wskazuje na przycisk ekranowy, który był wcześniej utworzony na potrzeby zmiennej `b`, czyli `c` nie utraciło referencji. Co więcej, obiekt wskazywany przez `c` nie zostanie zwolniony przez system zwalniania wolnej pamięci Javy. JVM zwalnia bowiem pamięć po zajmowanych obiektach, ale pod warunkiem że nie są one już używane, to znaczy żadna nazwa symboliczna znajdująca się w aktualnej przestrzeni nazw na nie nie wskazuje. W działaniu takim uwidacznia się bezpieczeństwo Javy, która nie umożliwia ręcznego zwolnienia pamięci. Gdyby bowiem po powieleniu adresu obiektu na jednej z kopii dało się wykonać działania destruktora, zniszczyłby on również drugą. Użytkownik, nie mając świadomości tego faktu, poprzez użycie zwolnionego obiektu mógłby spowodować błędy w działaniu.

Jednak poprawne kopiowanie obiektów, to znaczy tworzenie ich kopii bez wywoływania konstruktora klasy, jest możliwe. Do tego celu stworzona została metoda `clone`. Tworzy ona nowy egzemplarz obiektu, podobnie jak operator `new`, i wypełnia wszystkie jego pola wartościami ustawionymi w tym obiekcie, którego kopię tworzymy. Jednak metoda `clone` nie jest dostępna w prosty sposób z dowolnego obiektu. Aby można było jej użyć, obiekt, który zamierzamy powielać, musi spełniać dwa warunki — pokrywać oryginalną metodę `clone` z uwzględnieniem możliwości powstania wyjątków oraz implementować interfejs `Cloneable` (o interfejsach i implementacji przeczytasz dalej w tym rozdziale). Przykładowa klasa ze szczątkową obsługą błędów, która może być powielana za pomocą swojej metody `clone`, przedstawiona jest na listingu 2.38.

Listing 2.38. *Definicja klasy, której obiekty mogą być powielane*

```
class A implements Cloneable {
  int i1, i2;
  public Object clone() {
    try { return super.clone(); }
    catch(Exception e)
    { return null; }
  }
}
```

Jej użycie przedstawione jest na listingu 2.39.

Listing 2.39. *Powielanie obiektów z listingu 2.38*

```
public class Applet2 extends Applet {
  A a1 = new A();
  A a2;
  public void init() {
    a1.i1 = 1;
    a1.i2 = 2;
    a2 = (A)a1.clone();
    a2.i1 = 3;
    a2.i2 = 4;
    System.err.println(a1.i1);
    System.err.println(a1.i2);
  }
}
```

Dla upewnienia się, że powstały dwa różne obiekty, zastosowałem tu zmianę wartości pól jednego z nich i sprawdzenie, czy pola drugiego nie zmieniły się. W wyniku działania tego programu na konsoli przeglądarki zobaczymy liczby `1` i `2`. Oznacza to, że rzeczywiście mamy do czynienia z dwoma różnymi egzemplarzami. Gdyby zamiast użycia metody klonującej zastosować proste podstawienie obiektów, o którym już wcześniej pisałem, że jest niepoprawne:

```
a2 = a1;
```

na konsoli zobaczylibyśmy liczby `3` i `4`. Oznaczałoby to, że mamy co prawda dwie referencje do obiektów, ale obie do jednego egzemplarza. To znaczy nie mamy dwóch kopii obiektu, ale dwa wskazania na jeden obiekt.

2.2.4. Niejawne używanie obiektów

Rezerwowanie dla obiektu nazwy symbolicznej ma sens jedynie wtedy, gdy w dalszej części programu zamierzamy go wykorzystywać. Są jednak sytuacje, kiedy obiekt tworzony jest doraźnie wyłącznie na potrzeby jednorazowej akcji i nie zamierzamy się już do niego nigdy odwoływać. Tak na przykład może być w przypadku etykiet umieszczanych w programie w celach informacyjnych. Raz umieszczoną etykietę zamierzamy pozostawić w tym miejscu, gdzie ją umieściliśmy, bez żadnych zmian.

Nie interesuje nas odwołanie do niej (chociaż w praktyce wygląda to tak, że system będzie to robił za nas w sposób niejawny). W takim przypadku możemy skorzystać z niejawnej metody tworzenia obiektów pokazanej na listingu 2.40.

Listing 2.40. *Inicjacja etykiety bez nadania nazwy jej zmiennej*

```
class DemoPanel extends Panel {
  public DemoPanel() {
    this.add(new Label("napis informacyjny"), null);
  }
}
```

W pokazanym przykładzie obiekt tworzony jest wyłącznie po to, aby można go było użyć w metodzie `add`, która dodaje napis informacyjny do naszego panelu. Korzystamy tu z wiedzy o sposobie wywoływania metod w Javie. Otóż parametry przekazywane do tych metod są wyliczane przed ich wywołaniem. W związku z tym przedstawiona konstrukcja powoduje, że przed przekazaniem w postaci parametru uruchamiany jest konstruktor nowego obiektu klasy `Label`, którego adres przekazywany jest jako parametr. Oczywiście niejawne użycie nie oznacza wcale, że obiekt istnieje tylko przez czas wykonywania metody `add`. Wewnątrz tej metody obiekt przypisywany jest do pojedynczego elementu nowej zmiennej tablicowej typu umożliwiającego przechowywanie takich obiektów. W związku z tym, korzystając ze spostrzeżenia poczynionego wcześniej, wytłuszczona linia mogłaby być rozwinięta bez zmiany funkcjonalności do postaci:

```
Label l = new Label("napis informacyjny");
this.add(l, null);
l = null;
```

W związku z niepotrzebną rozwlekłością kodu i niecelowym zablokowaniem nazwy zmiennej na potrzeby jednorazowej akcji naturalne jest użycie niejawnej formy utworzenia obiektu, która nie wymaga rezerwacji nazwy zmiennej.

2.2.5. Typ zmiennej i obiektu. Operator instanceof

W dotychczasowych przykładach używania obiektów typ zmiennej, który służył do ich przechowywania, był dokładnie taki, jak ich typ. Takie rozwiązanie jest jak najbardziej słuszne i najczęściej używane. Nie oznacza to jednak, że jest to jedyny sposób na deklarowanie zmiennych obiektowych. Często zdarza się, że w celu zwiększenia funkcjonalności programu deklarujemy zmienną innego typu, niż w efekcie będziemy wykorzystywać. W bezpieczny sposób możemy deklarować typ zmiennej jako typ nadklasy klasy, której będziemy używali. Możemy więc zadeklarować, a następnie utworzyć zmienną z użyciem konstrukcji:

```
Object b;
b = new Button("przycisk");
```

Taka konstrukcja nazywa się bezpieczną konwersją typu, konwersją w górę lub rozszerzaniem zakresu zmiennej. Dalsze użycie zmiennej `b` może okazać się utrudnione, gdy

będziemy chcieli wykorzystać cechy obiektu charakterystyczne nie dla typu zmiennej, a dla rzeczywistego typu obiektu. Bez problemów moglibyśmy użyć dla tej zmiennej metody `toString` zadeklarowanej w klasie `Object`, która zwraca tekstową reprezentację tego obiektu. Jednak wstawienie obiektu typu przycisk ekranowy (`Button`) do graficznego obrazu programu wymaga użycia funkcji `add`, która oczekuje, że parametr jej wywołania będzie obiektem typu `Component` lub obiektem typu pochodnym. W związku z tym, że `Object` jest klasą nadrzędną dla `Component`, użycie zmiennej typu `Object` nie zostanie zaakceptowane:

```
add(b, null);  // jeśli b typu Object, to błąd
```

Jeśli mamy pewność, że `b` jest typu `Button` (lub `Component`, po którym dziedziczy `Button`), wtedy możemy użyć jawnej konwersji typu `Object` na typ akceptowany przez metodę `add`:

```
add((Component)b, null);
```

Takie wywołanie zostanie zaakceptowane przez kompilator. W wywołaniu tym zastosowaliśmy niebezpieczną konwersję typu, konwersję w dół bądź zwężenie typu zmiennej. Jeśli konwersja jest poprawna, to znaczy zmienna `b` reprezentuje obiekt typu `Component` lub pochodny od niego, wtedy wszystko zadziała dobrze. Gdyby jednak okazało się, że `b` nie przechowuje obiektu typu `Component` ani żadnego innego od niego pochodzącego, wtedy wystąpi błąd (konkretnie będzie to wyjątek `ClassCastException`), który wstrzyma wykonanie programu wykonanie. Gdyby konwersja była niepoprawna już na etapie kompilacji, to znaczy próbowalibyśmy konwertować typ obiektów między dwoma gałęziami dziedziczenia, wtedy błąd wystąpiłby już na etapie kompilacji.

Zdarza się, że aby podnieść uniwersalność niektórych fragmentów programu, z założenia stosuje się klasę nadrzędną jako typ zmiennej. Często, niemal nagminnie, stosuje się takie rozwiązanie w przypadku przekazywania obiektów poprzez parametry do wnętrza metod. Nie mamy wtedy żadnej pewności, jakiego typu obiekt odbieramy. Do obsługi takich sytuacji Java posiada operator `instanceof`, który umożliwia zbadanie typu egzemplarza klasy. W praktyce dokonuje on porównania typu egzemplarza klasy z typem klasy. Przykład jego użycia pokazany jest na listingu 2.41.

Listing 2.41. *Przykładowe badanie typu obiektu*

```
public void UseObject(Object comp) {
  if (comp instanceof Button) {
    // obsługa obiektu typu Button
  } else if (comp instanceof Label) {
    // obsługa obiektu typu Label
  } else {
    // obsługa innych obiektów
  }
}
```

Takie podejście zapewnia zabezpieczenie programu przed nieoczekiwanymi przerwami w pracy oraz umożliwia wprowadzenie do niego większej funkcjonalności.

2.2.6. Efekty polimorfizmu

Polimorfizm to zamierzone działanie w idei programowania obiektowego. Zapewnia ono wielopostaciowość działania zależną od wykorzystywanego obiektu. Dokładnie opisałem to w poprzednim rozdziale. Ponadto wspominałem już o tym zarówno przy opisywaniu dziedziczenia, jak i przykrywania pól i metod. Ominę więc rozważania teoretyczne, a przedstawię tylko efekt działania polimorfizmu. Do prezentacji wykorzystam dwie klasy o definicji podanej na listingu 2.42.

Listing 2.42. *Klasy dziedziczące z pokrytymi metodami*

```
class A {
  public void info1() {
    System.err.println("klasa A(1)");
  }
  public void info2() {
    System.err.println("klasa A(2)");
    info3();
  }
  public void info3() {
    System.err.println("klasa A(3)");
  }
}

class B extends A {
  public void info1() {
    System.err.println("klasa B(1)");
  }
  public void info3() {
    System.err.println("klasa B(3)");
  }
}
```

Efekty działania polimorfizmu będę mógł zaobserwować dzięki apletowi przedstawionemu na listingu 2.43.

Listing 2.43. *Użycie klas z pokrytymi metodami*

```
public class Applet2 extends Applet {
  public void init() {
    A a = new A();
    a.info1();
    a.info2();
    a = new B();   // a staje się klasy B
    a.info1();
    a.info2();
  }
}
```

Na konsoli Javy przeglądarki internetowej pojawi się ciąg napisów zaprezentowany na rysunku 2.5.

Rysunek 2.5. *Wydruk generowany przez listing 2.43 z użyciem klas z listingu 2.42*

```
klasa A(1)
klasa A(2)
klasa A(3)
klasa B(1)
klasa A(2)
klasa B(3)
```

Należy zwrócić uwagę na następujące sprawy:

- Wywoływana jest metoda `info1` zawsze tej samej klasy, jakiego typu jest obiekt w danym momencie przechowywany w zmiennej `a`. Dzieje się tak mimo tego, że typ zmiennej `a` to `A`. Tak więc odwołanie dotyczy rzeczywistego typu obiektu, a nie wymienionego w referencji.
- Możemy wywołać metodę `info2` obiektu typu `B`, mimo że nie została ona w nim zaimplementowana. Oznacza to, że metoda ta została odziedziczona po klasie `A` i jest traktowana jak własna metoda obiektu `B`. Jakkolwiek jej wywołanie nie jest polimorficzne, w przypadku dalszego dziedziczenia i pokrycia jej może powstać ten efekt.
- Metoda `info3`, jakkolwiek wywoływana z metody zaimplementowanej tylko w klasie `A`, również pochodzi z tej klasy, której typ reprezentuje obiekt. Tak więc tworząc klasę `A`, możemy przewidzieć użycie nieistniejącej jeszcze metody z klasy pochodnej, która zostanie utworzona dopiero w przyszłości.

Przypominam, że inny przykład działania polimorfizmu pokazany został wcześniej w paragrafie 2.1.11. „Kolejność inicjacji klas". Problemy związane z polimorfizmem i klasami wewnętrznymi omówione są też w paragrafie 2.3.2. „Polimorfizm i zmienne klasy zawierającej".

Omawiając efekt polimorfizmu, warto też wspomnieć, że niektórzy autorzy pod pojęcie polimorfizmu podciągają możliwość przeciążania metod. Wydaje mi się to zbyt rozszerzoną interpretacją. Metody przeciążone to zazwyczaj takie, które działają w bardzo podobny sposób, tylko przekazywanie do nich parametry są innego rodzaju bądź wykorzystuje się domyślne. Ponadto istnieje wiele języków programowania, które nie udostępniają przeciążenia metod, i nikt nie zwraca uwagi na to, że ich polimorfizm jest zubożony. Według mnie przeciążanie nie należy do zakresu polimorfizmu, ale zwracam uwagę na to, że możesz zetknąć się z taką tezą.

2.3. Klasy wewnętrzne i lokalne

W klasycznym podejściu każda klasa w Javie zawarta jest w osobnym pliku albo stanowi odrębny, łatwy do wydzielenia fragment większego. Dzięki temu klasy mogą tworzyć biblioteki dostępne z dowolnego miejsca w całym programie. Od reguły tej istnieją pewne wyjątki wprowadzone w celu podniesienia stopnia hermetyzacji programu. Wyjątkami tymi są klasy wewnętrzne i lokalne. Klasa wewnętrzna to zdefiniowana w miejscu, w którym może wystąpić definicja pola lub metody. Musi więc ona wystąpić między pierwszym i ostatnim nawiasem klamrowym ograniczającym ciało

klasy na prawach równych polu lub metodzie. Klasa, w której definiowana jest klasa wewnętrzna, nazywana jest zawierającą. Klasy lokalne to klasy zdefiniowane w bloku programu Javy. Ta kwestia może więc dotyczyć inicjatora klasy, inicjatora obiektu, konstruktora bądź (najczęściej) metody. Klasa lokalna może odwoływać się do wszystkich zmiennych widocznych w miejscu wystąpienia jej definicji. Jest ona widoczna i może zostać użyta tylko w bloku, w którym została zdefiniowana. Poza tymi cechami charakterystycznymi klasy wewnętrzne i lokalne są najzwyklejszymi klasami. Można je tworzyć przy użyciu reguł obowiązujących wobec zwykłych klas. Główne zalety to możliwość ukrywania nazw klas w strukturze bibliotek i pakietów Javy (poza szczególnymi sposobami użycia są one niedostępne z zewnątrz) oraz używania zmiennych lokalnych w tych klasach. Daje to czasami możliwość lepszej optymalizacji niektórych fragmentów kodu zarówno ze względu na ułatwienia w implementacji algorytmu, jak i pracy kompilatora. Ponadto udostępnia mechanizm zbliżony do typu proceduralnego, który jest dostępny w większości obiektowych języków programowania. Przykład pokazujący umiejscowienie klasy wewnętrznej i lokalnej pokazany jest na listingu 2.44.

Listing 2.44. *Przykład klasy wewnętrznej i lokalnej*

```
class Zawierajaca {
  Wewnetrzna w = new Wewnetrzna();
  class Wewnetrzna {
    // ciało klasy wewnętrznej
  }
  public void metodaZ() {
    Lokalna l = new Lokalna();
    class Lokalna {
      // ciało klasy lokalnej
    }
    // ciało metody MetodaZ
  }

  // ciało klasy zawierającej
}
```

Mimo tego, że klasy te są pełnoprawne pod względem konstrukcyjnym, sugeruje się, aby nie były one zbyt rozbudowane. Wynika to przede wszystkim z chęci uniknięcia pomyłek typu: *który fragment jest metodą klasy zawierającej, który wewnętrznej, a który lokalnej*. Oczywiście możemy jednak tworzyć w ten sposób nawet bardzo skomplikowane konstrukcje — formalnie nie istnieją bowiem żadne ograniczenia w budowaniu klas wewnętrznych i lokalnych.

Kompilacja kodu utworzonego na podstawie szablonu pokazanego powyżej spowoduje powstanie plików:

```
Zawierajaca.class
Zawierajaca$Wewnetrzna.class
Zawierajaca$Lokalna.class
```

Tak więc zastosowanie klas wewnętrznych zmniejszy tylko liczbę plików źródłowych. Liczba plików skompilowanych `class` pozostaje stała.

2.3.1. Dostęp do zmiennych klasy zawierającej

Jakkolwiek ogólne zasady tworzenia klas wewnętrznych i lokalnych są takie same, tak jak dla zwykłych klas istnieje kilka szczegółów, na które należy zwrócić uwagę, bo rozszerzają możliwości udostępnione w czasie tworzenia tych klas. Jednym z takich udogodnień jest bezpośredni dostęp przez klasy wewnętrzne i lokalne do pól klas zawierających. Umożliwia to uproszczenie niektórych konstrukcji i przybliżenie ich do natury programowania strukturalnego. Kilkakrotnie zwracałem już uwagę na to, że Java w niektórych przypadkach bywa ortodoksyjnie obiektowa. Stosowanie klas wewnętrznych nieznacznie łagodzi te restrykcje. Warto jednak pamiętać, że bezpośredni dostęp do zmiennych spoza klasy jest niezgodny z ideą programowania obiektowego. Jeśli jednak założymy, że klasy lokalne powstają wyłącznie na potrzeby klasy zawierającej, w której są potem używane, możemy przyjąć, że nie łamie to zbyt mocno reguł hermetyzacji. Przykład użycia pól przez klasy wewnętrzne i lokalne pokazany jest na listingu 2.45, we fragmencie większej klasy.

Listing 2.45. *Używanie pól klasy zawierającej przez klasy wewnętrzne i lokalne*

```
int i;

class Wewnetrzna {
  double wynik() { return (i+0.01) * 0.99; }
}

void button_actionPerformed(ActionEvent e) {
  class Lokalna {
    double wynik() { return (i+0.05) * 0.8; }
  }
  Lokalna l = new Lokalna();
  Wewnetrzna w = new Wewnetrzna();
  for (i=0; i<1000; i++){
    showStatus("w: " + l.wynik() + ", " + w.wynik());
}
```

Rozwiązanie takie ma między innymi tę zaletę, że nie ma konieczności przekazywania czy synchronizowania wartości pola klasy zawierającej z działaniami klasy wewnętrznej czy lokalnej. Ściśle koresponduje to z celem tworzenia klas wewnętrznych, które z założenia mają wykonać jakieś zadanie wewnątrz klasy zawierającej, czyli z dostępem do jej pól. Należy jednak zwrócić uwagę na niebezpieczeństwa mogące się pojawić w przypadku częściowego odwoływania się do pól klasy zawierającej w przypadku istnienia większej liczby egzemplarzy klas wewnętrznych tego samego typu. Przykład niebezpiecznych interakcji pokazano na listingu 2.46.

Listing 2.46. *Błędne użycie pól klasy zawierającej*

```
int i;
class Wewnetrzna {
  int j;
  void setSth(int a) {
    i = a;
    j = a;
  }
```

```
  int suma() { return i+j; }
}

void metodaX() {
  int wynik;
  Wewnetrzna w = new Wewnetrzna();
  Wewnetrzna w2 = new Wewnetrzna();
  w.setSth(10);
  w2.setSth(100);
  wynik = w.suma();  // 110,
  // choć spodziewaliśmy się 20
  wynik = w2.suma(); // 200
}
```

Gdyby w klasie `Wewnetrzna` nie było interakcji z polem klasy zewnętrznej, wtedy działanie tego kodu byłoby zupełnie inne, niż jest. Takie błędy trudno wyśledzić. W związku z tym sugeruję jawne odwoływanie się do pól klas zewnętrznych wyłącznie w przypadku, gdy mamy pewność, że będziemy używać wyłącznie jednego egzemplarza klasy wewnętrznej.

2.3.2. Polimorfizm i zmienne klasy zawierającej

Używanie pól klasy zawierającej powinno odbywać się z dużą ostrożnością w przypadku, gdy klasa wewnętrzna dziedziczy po tak zwanej normalnej klasie. Poniżej przedstawiony jest przykład, który mimo tego, że bezbłędnie przechodzi kompilację (nawet z użyciem kompilatora *jikes* i jego opcji pedantycznej kompilacji `+P`), nie pracuje poprawnie. Najważniejsze fragmenty, na które należy zwrócić uwagę, wytłuściłem na listingu 2.47.

Listing 2.47. *Błędne użycie pól klasy zawierającej*

```
import java.applet.*;

public class Aaa extends Applet {
  public Integer p1 = new Integer(2);
  public void init() {
    System.out.println("wewnatrz konstruktora apletu");
    B b = new B();
  }
  private class B extends A {
    public void doSth() {
      System.out.println("p1=" + p1);
    }
  }
}

abstract class A {
  public A() {
    System.out.println("wewnatrz konstruktora A");
    doSth();
  }
  abstract public void doSth();
}
```

Wewnętrzna klasa `B` korzysta z pola `p1` pozornie w sposób poprawny. Pozornie, gdyż jeśli cofniesz się do paragrafu 2.1.11. „Kolejność inicjacji klas", przypomnisz sobie zapewne, że zanim zostanie zainicjowana klasa `B`, wcześniej inicjowana jest `A`. W czasie jej inicjacji w jej konstruktorze wywoływana jest metoda `doSth`. A metoda ta, zgodnie z zasadami polimorfizmu, zostanie zaczerpnięta już z klasy `B`. Jednak nie stanie się to w obszarze wewnętrznym apletu `Aaa`, lecz poza nim. I efektem działania będzie błąd (konsola Javy z Internet Explorera) pokazany na rysunku 2.6.

Rysunek 2.6. *Efekt działania programu 2.47 w Internet Explorerze*

```
wewnatrz konstruktora apletu
wewnatrz konstruktora A
java.lang.NullPointerException
  at aaa$B.doSth (aaa.java:11)
  at A.<init> (aaa.java:20)
  at aaa$B.<init> (aaa.java:9)
  at aaa.init (aaa.java:8)
  at com/ms/applet/AppletPanel.securedCall0 (AppletPanel.java)
  at com/ms/applet/AppletPanel.securedCall (AppletPanel.java)
  at com/ms/applet/AppletPanel.processSentEvent
  ➥(AppletPanel.java)
  at com/ms/applet/AppletPanel.processSentEvent
  ➥(AppletPanel.java)
  at com/ms/applet/AppletPanel.run (AppletPanel.java)
  at java/lang/Thread.run (Thread.java)
```

Opera (a w zasadzie jej konsola Javy) pokaże ten błąd nieco inaczej, co zaprezentowane jest na rysunku 2.7.

Rysunek 2.7. *Efekt działania programu 2.47 w Operze*

```
wewnatrz konstruktora apletu
wewnatrz konstruktora A
java.lang.NullPointerException
  at aaa$B.doSth(aaa.java:11)
  at A.<init>(aaa.java:19)
  at aaa$B.<init>(aaa.java:9)
  at aaa.init(aaa.java:7)
  at opera.PluginPanel.run(opera/PluginPanel.java:308)
  at java.lang.Thread.run(Unknown Source)
```

Przyczyny błędu lepiej widoczne są po dekompilacji przedstawionego programu z użyciem dekompilatora *jad* (należy użyć opcji `-noinner`). Przedstawiony wcześniej kod został przetłumaczony nieco inaczej niż wygląd źródła (pominąłem kod klasy `A`, który jest dokładnie taki sam jak w oryginale). Wynika to z jawnego wyniesienia klasy wewnętrznej na zewnątrz. Na listingu 2.48 pokazany jest sposób, w jaki widzi ten kod maszyna Javy.

Listing 2.48. *Kod 2.47 po kompilacji i dekompilacji*

```
import java.applet.Applet;
import java.io.PrintStream;

public class Aaa extends Applet {
  public void init() {
    System.out.println("wewnątrz konstruktora apletu");
    Aaa$B aaa$b = new Aaa$B(this);
  }
```

```
  public Aaa() {
    p1 = new Integer(2);
  }
  public Integer p1;
}

class Aaa$B extends A {
  public void doSth() {
    System.out.println("p1=" + this$0.p1);
  }
  Aaa$B(Aaa aaa1){
    this$0 = aaa1;
  }
  private final Aaa this$0; /* synthetic field */
}
```

Jak widać, wewnątrz klasy `B` dodane jest pole `this$0`, które przechowuje wskazanie do używającego jej obiektu. Wskazanie to jest inicjowane w konstruktorze. Jak widać, zarówno konstruktor, jak i inicjacja klasy odbywa się inaczej, niż jest to jawnie zapisane. Jeśli przypomnimy sobie paragraf 2.1.11. „Kolejność inicjacji klas", jasne się stanie, dlaczego pojawia się błąd. W czasie tworzenia obiektu `b` klasy `B`, czyli w czasie wykonania wiersza:

```
B b = new B();
```

JVM rozpoczyna wykonywanie konstruktora `B` (w kodzie dekompilowanym `Aaa$B`). W ramach wykonania tego konstruktora przed wykonaniem jego pierwszego wiersza, czyli przed wykonaniem kodu:

```
this$0 = aaa1;
```

uruchamiany jest konstruktor klasy `A`. W konstruktorze tym uruchamiana jest metoda `doSth`, w ramach której (w związku z polimorfizmem metod) wykonywany jest wiersz:

```
System.out.println("p1=" + this$0.p1);
```

Oczywiście pole `p1` apletu `Aaa` jest już zainicjowane. Ale pole `this$0` obiektu `b` klasy `B` jeszcze nie. W związku z tym naturalne jest pojawienie się wyjątku `NullPointerException`. Oczywiście problemowi temu można zapobiec, przekazując kaskadowo do klasy nadrzędnej wartość tego parametru. W praktyce jednak widać, że w takim przypadku nie powinno się używać klasy wewnętrznej. W końcu została ona wprowadzona generalnie w celu rozwiązywania problemów, które wymagają tworzenia małych klas na potrzeby jednorazowego użycia. Przypadek, kiedy używamy zagnieżdżonego dziedziczenia, raczej powinno się zrealizować z użyciem klasycznego rozwiązania, czyli klasycznego tworzenia klas prywatnych.

2.3.3. Zmienne lokalne w klasie lokalnej

Powyższe rozważania dotyczyły zarówno klas wewnętrznych, jak i lokalnych. Dla klas lokalnych istnieje dodatkowa możliwość korzystania ze zmiennych lokalnych i parametrów przekazywanych do wnętrza metod. Jednak dostęp do tych zmiennych wymaga

specyficznego ich deklarowania, to znaczy muszą one posiadać status finalny (dokładne wytłumaczenie znaczenia tego statusu znajdziesz w podrozdziale 2.9. „Modyfikatory"). Przykład użycia zmiennej przekazywanej przez parametr pokazany jest na listingu 2.49.

Listing 2.49. *Przekazywanie zmiennej przez parametr*

```
void showInfo(final int i) {

  class Lokalna {
    int wynik() { return 2*i; }
  }

  Lokalna l = new Lokalna();
  showStatus("wynik: " + l.Wynik());
}
```

Jeśli na zmiennej musimy wykonywać jakieś obliczenia, należy zastosować inne rozwiązanie, a mianowicie wyłącznie na potrzeby klasy lokalnej musimy zadeklarować zmienną finalną, tak jak pokazałem to na listingu 2.50.

Listing 2.50. *Używanie zmiennej finalnej*

```
void showInfo() {
  final int i2;
  int i = 100;
  // tu działania na i
  class Lokalna {
    int wynik() { return 2*i2; }
  }
  i2 = i;
  Lokalna l = new Lokalna();
  showStatus("wynik: " + l.Wynik());
}
```

Przypisanie wartości zmiennej finalnej musi odbyć się przed utworzeniem egzemplarza klasy lokalnej, gdyż wykorzystuje ona tę zmienną w momencie tworzenia egzemplarza klasy, a nie jak nam się wydaje w chwili użycia metody `wynik`, która tę zmienną wykorzystuje. Szerzej na ten temat napiszę w rozdziale dotyczącym programowania sterowanego zdarzeniami, w podrozdziale 4.5. „Zdarzenia z parametrem".

2.3.4. this w klasach wewnętrznych

Podobnie jak w innych klasach również w wewnętrznych i lokalnych z powodzeniem można używać słowa kluczowego `this`, które służy jako referencja do własnej klasy macierzystej. Jednak `this` w klasach zagnieżdżonych nie jest tak bardzo jednoznaczne, jak w zwykłych. W związku z tym możliwe jest poprzedzenie słowa `this` nazwą klasy i oddzielenie jej od niego kropką. Na listingu 2.51 pokazany jest przykład użycia `this` w stosunku do klasy zawierającej.

Listing 2.51. *Użycie this w stosunku do klasy zawierającej*

```
class SingApplet extends Applet {
  int i = 100;
  class Wewnetrzna {
    int i = 222;
    void setValue(int i) {
      SingApplet.this.i = i;
    }
    int wynik() {
      return SingApplet.this.i;
    }
  }
  //...
}
```

Na listingu 2.52 słowo `this` zostało użyte w stosunku do klasy wewnętrznej.

Listing 2.52. *Użycie this w stosunku do klasy wewnętrznej*

```
int i = 100;
class Wewnetrzna {
  int i = 222;
  void setValue(int i) {
    Wewnetrzna.this.i = i;
  }
  int wynik() {
    return Wewnetrzna.this.i;
  }
}
```

Kod ten jest równoważny zapisowi bez żadnej referencji przed `this` pokazanej na listingu 2.53.

Listing 2.53. *Oszczędna wersja programu 2.52*

```
int i = 100;
class Lokalna {
  int i = 222;
  void setValue(int i) {
    this.i = i;
  }
  int wynik() {
    return i;
  }
}
```

Wynika to ze sposobu poszukiwania i ustalania przez kompilator nazw pól i metod. W pierwszej kolejności poszukuje on nazw lokalnych (deklarowanych wewnątrz metod), następnie (jeśli nie znalazł właściwej nazwy) rozszerza zakres poszukiwań, by w drugim podejściu sprawdzić (w naszym przykładzie) klasę lokalną. Jako że tam znajduje odpowiednie pole, jednoznacznie je identyfikuje. Mimo takiej jednoznacznej identyfikacji dobrą praktyką jest (jeśli już musimy powielać nazwy) stosowanie pełnych referencji do nazwy, co zapobiega przypadkowym pomyłkom.

2.3.5. Korzystanie z klas wewnętrznych

Klasy wewnętrzne i lokalne to, jak wcześniej pisałem, klasy o własnościach niemal nieróżniących się od zwykłych klas. W związku z tym klasyczny sposób ich użycia jest dokładnie taki sam, jak pokazany wcześniej i dotyczący zwykłych klas. Dla przykładu, użycie klasy lokalnej można przeprowadzić w sposób pokazany na listingu 2.54.

Listing 2.54. *Użycie klasy lokalnej*

```
public void metodaZKlasa() {
  class Lokalna {
    int wynikLokalny() {
      return 0;
    }
  }
  Lokalna l = new Lokalna();
  int i = l.wynikLokalny();
  //...
}
```

Mimo tego, że klasy wewnętrzne projektowane są z myślą o używaniu ich wewnątrz innych klas, czasami może okazać się, że musimy wykorzystać je na zewnątrz, poza klasami, wewnątrz których umieszczony został ich kod. Pomijam przypadek, gdy kod klasy wewnętrznej da się w łatwy sposób wykopiować z zewnętrznej i użyć w sposób samodzielny. Jeśli jednak korzystaliśmy z pełni możliwości dostarczonych przez mechanizm zagnieżdżania klas, operacja taka może być niemożliwa do zrealizowania. W takim przypadku klasy wewnętrznej możemy użyć w specyficzny sposób, traktując klasę zawierającą podobnie do biblioteki, w której znajduje się nasza klasa. Aby utworzyć egzemplarz klasy wewnętrznej, musimy najpierw utworzyć egzemplarz zawierającej. Dla przykładu posłużę się najprostszymi klasami o postaci:

```
class A {
  class A1 { }
}
```

Użycie wewnętrznej klasy `A1` wymaga utworzenia wcześniej egzemplarza klasy zawierającej `A`:

```
// utworzenie egzemplarza klasy zawierającej:
A a = new A();
// utworzenie egzemplarza klasy wewnętrznej:
A.A1 a1 = a.new A1();
```

bądź w formie skróconej:

```
A.A1 a1 = (new A()).new A1();
```

Warto zauważyć dziwną sytuację, a mianowicie typ klasy wewnętrznej jest widziany poprzez referencje nazwy klasy zawierającej i kropki. Natomiast użycie samej klasy wewnętrznej jest niemożliwe. Bardzo podobna sytuacja wystąpi w przypadku próby dziedziczenia. Wykorzystam wprowadzoną klasę `A1` zadeklarowaną wewnątrz `A` do próby dziedziczenia:

```
class B extends A.A1 {
  B(A a) { a.super(); }
}
```

Wyłącznie tak zadeklarowany konstruktor klasy `B` zostanie zaakceptowany przez kompilator. Wyjaśnienie wymaga przypomnienia sobie tego, co napisałem w podrozdziale 2.1.10. „Inicjator klasy i obiektu" oraz 2.1.11. „Kolejność inicjacji klas". Otóż przed utworzeniem klasy przeprowadzane jest budowanie wszystkich klas nadrzędnych z całego łańcucha dziedziczenia. W związku z tym, że do utworzenia egzemplarza klasy wewnętrznej potrzebny jest istniejący egzemplarz klasy zewnętrznej, wywołanie konstruktora klasy bazowej może się odbyć jedynie w odniesieniu do tego egzemplarza. Stąd konieczność wcześniejszego utworzenia klasy zewnętrznej i podanie jej jako parametru konstruktora klasy dziedziczącej po wewnętrznej. Na skutek tego korzystanie z klasy dziedziczącej po wewnętrznej będzie musiało mieć postać:

```
// utworzenie egzemplarza klasy zawierającej
A a = new A();
// utworzenie egzemplarza klasy dziedziczącej
B b = new B(a);
```

lub w formie skróconej:

```
B b = new B(new A());
```

Jeśli użyliśmy pełnej formy inicjacji klasy wewnętrznej bądź dziedziczącej po wewnętrznej, wtedy po jej użyciu możemy zwolnić nazwę klasy zewnętrznej bez żadnych negatywnych konsekwencji:

```
A a = new A();
A.A1 a1 = a.new A1();
B b = new B(a);
a = null;
```

Konstrukcja taka nie spowoduje błędów w pracy programu, nawet jeśli klasa wewnętrzna odwoływała się w bezpośredni sposób do pól klasy zawierającej. Przyczyn należy się doszukiwać w sposobie korzystania z klas wewnętrznych. W specyfikacji do Javy 1.1, w której po raz pierwszy wprowadzono klasy wewnętrzne, znalazłem informacje, że kompilator tłumaczy kod źródłowy zawierający klasy wewnętrzne w taki sposób, aby kod wynikowy nie różnił się od kodu zapisanego z użyciem zwykłych (rozłącznych) klas. Klasa wewnętrzna zawiera niejawną referencję do obiektu klasy zawierającej, który stanowi jego otoczkę. Tak więc nawet jawne zwolnienie obiektu otaczającego nie spowoduje fizycznego usunięcia obiektu, gdyż referencje do niego przechowuje sam obiekt klasy wewnętrznej. Ponadto według tej samej specyfikacji program napisany z użyciem klas wewnętrznych, zawierający poza tym wyłącznie konstrukcje językowe charakterystyczne dla Javy 1.0 może zostać uruchomiony w środowisku JVM 1.0. Informacja ta jest o tyle wiarygodna, że większość początkowych wersji języków obiektowych bazujących na językach strukturalnych (dotyczy to przede wszystkim C++) w czasie kompilacji tłumaczona była najpierw z kodu obiektowego na strukturalny na poziomie źródła, a następnie kompilowana zwykłym, strukturalnym kompilatorem. Trudno powiedzieć, o ile zmienił się ten mechanizm obecnie. Biorąc jednak pod uwagę, że kod programu skompilowany z użyciem różnych kompilatorów zgodnych z wersją 1.4 jest bezproblemowo wykonywany w przeglądarkach internetowych zawierających różne wersje Javy 1.1 (oczywiście pod warunkiem że nie zawiera konstrukcji i bibliotek z wyższych wersji), można powiedzieć, że sytuacja zmieniła się niewiele lub wcale.

2.4. Interfejsy

Jak już wcześniej wspominałem, Java nie udostępnia mechanizmu wielokrotnego dziedziczenia. Ze względu na brak nieobiektowych elementów takie rozwiązanie skutkowałoby sporymi utrudnieniami w tworzeniu bardziej zaawansowanych programów. Skutkowałoby, gdyby nie interfejsy. Jest to specjalnie zaprojektowany mechanizm, który umożliwia klasom wykorzystanie innych klas, nawet jeśli były one nieznane w czasie tworzenia fragmentów wykorzystującego je kodu. Analogicznie do sprzętowego interfejsu ten z Javy umożliwia łączenie ze sobą klas za pomocą struktury pośredniczącej stworzonej niezależnie od łączonych klas. Interfejsy sprawdzają się wszędzie tam, gdzie istnieje czasowa rozbieżność pomiędzy powstaniem klasy używającej pewnych obiektów, a powstaniem tej, która będzie używana. Tworząc na przykład taki mechanizm w dużym zespole programistycznym, musimy zadbać o właściwy obieg informacji i dostarczenie pełnego opisu oczekiwań. Musielibyśmy przedstawić specyfikację używanej przez nas klasy w postaci:

> *Musi posiadać pola o następujących nazwach i typach, musi posiadać metody o następujących nazwach i typach oraz formalnej liście parametrów oraz musi dziedziczyć własności po konkretnej klasie.*

Zamiast pisać to wszystko, wystarczy, że podamy:

> *Musi implementować interfejs XXX.*

I sprawa jest oczywista. Nie grozi nam żadne przekłamanie w nazewnictwie pól czy metod. Interfejs jest dostarczany razem z naszą klasą i to my jesteśmy odpowiedzialni za jego kształt. Kompilator nie dopuści do powstania błędu. Ponadto interfejs ma tę zaletę, że może być wykorzystany w czasie testowania programu nawet wtedy, gdy właściwa klasa, której będziemy używać w przyszłości, jeszcze nie powstała. Dodatkowo korzystając z zalet interfejsu, uwalniamy autora klasy używanej przez nas od ograniczenia typu *twoja klasa musi dziedziczyć własności po klasie YYY*. Nie ma potrzeby stosować tego ograniczenia. Całkowitą zgodność typów zapewnia właśnie interfejs, bez względu na to, jaką klasę nadrzędną wybierze użytkownik. Dzięki temu do mechanizmu obsługi zdarzeń jako obiekt nasłuchujący może być przekazany zarówno obiekt typu `Applet`, jak i `Button`, mimo że oba należą do różnych łańcuchów dziedziczenia. A wszystko dzięki temu, że implementują właściwy interfejs (w pełniejszym zrozumieniu idei interfejsów może pomóc przestudiowanie rozdziału 4. „Programowanie sterowane zdarzeniami"). Ponadto dzięki interfejsowi możemy na przykład dodać klasy zakupione od jednego producenta do mechanizmu ich wykorzystania zakupionego gdzie indziej, nawet jeśli obie firmy nie wiedziały o swoim istnieniu (nie mówiąc o tym, że na pewno nie znały struktury i działania swoich produktów). Poza obsługą zdarzeń interfejsy są używane w sposób systemowy w programowaniu wielowątkowym (patrz podrozdział 4.2. „Klasyczna obsługa zdarzeń").

2.4.1. Definicja interfejsu

Ogólna postać deklaracji interfejsu jest bardzo podobna do deklaracji klasy i jest pokazana na listingu 2.55.

Listing 2.55. *Szablon deklaracji interfejsu*

```
interface Nazwa {
  final typ nazwa_zmiennej_finalnej_1;
  ...
  final typ nazwa_zmiennej_finalnej_K;
  typ nazwa_metody_1([lista_parametrów]);
  ...
  typ nazwa_metody_L([lista_parametrów]);
}
```

Główna różnica w stosunku do klas to:

- Brak ciała metod deklarowanych w interfejsie — po nazwie metody zamiast bloku instrukcji ujętego w nawiasy klamrowe znajduje się średnik.
- Brak konstruktora, czyli specjalizowanej metody używanej w przypadku tworzenia egzemplarza interfejsu (oczywiście gdyby było to możliwe, gdyż jak wspominałem, interfejs nie może być wzorcem do powstania egzemplarza).
- Możliwość stosowania wyłącznie zmiennych finalnych, czyli odpowiedników stałych z innych języków.

Przykładowy interfejs o nazwie `Przekaznik`, który zawiera jedną metodę `wykonajZestawInstrukcji` z argumentem typu ciąg tekstowy pokazany jest poniżej.

```
interface Przekaznik {
  void wykonajZestawInstrukcji(String instrukcje);
}
```

2.4.2. Implementacje

Jak wcześniej wspominałem, sam interfejs z punktu widzenia metod jest tylko deklaracją ich wystąpienia. Samo ciało metody musi zostać zaimplementowane w innym miejscu. Wspominałem również, że sam interfejs nie może być miejscem tej implementacji. Podobnie jak wirus musi się on „dokleić" do klasy, aby był w pełni funkcjonalny. Takie doklejanie odbywa się z użyciem słowa kluczowego `implements`. Po dodaniu tego fragmentu mogę wreszcie na listingu 2.56 zaprezentować pełny format nagłówka definicji klasy.

Listing 2.56. *Pełny format nagłówka klasy*

```
class Nazwa [extends NazwaNadklasy]
[implements Interfejs1 [, Interfejs2 [...]]]
{
  public typ metoda_z_interfejsu([parametry]) {
```

```
    ciało_implementowanej_metody
  }

  // część główna klasy
  // według wcześniejszego schematu
}
```

Już wskazuje pobieżna analiza nagłówka, klasa może implementować większą liczbę interfejsów. W części głównej klasy musi znaleźć się deklaracja i implementacja metody, której istnienie jest zadeklarowane w interfejsie (z wyjątkiem przypadku, kiedy klasę deklarujemy jako abstrakcyjną). Należy pamiętać, że do ciała klasy należy dodać metody zadeklarowane w implementowanych interfejsach. Metody te muszą być deklarowane z modyfikatorem `public`. Przykładowa implementacja interfejsu `przekaznik` zadeklarowanego wcześniej z użyciem klasy `UrzadzenieZdalnieSterowane` będzie miała postać pokazaną na listingu 2.57.

Listing 2.57. *Przykładowa implementacja interfejsu*

```
class UrzadzenieZdalnieSterowane
  extends Urzadzenie implements Przekaznik {
  private int dekodujRozkaz(String instrukcje, int polozenie) {
    // tu dekodowanie instrukcji
    // o numerze polozenie na Rozkaz typu int
  }
  private int dekodujParametr(String instrukcje, int polozenie) {
    // tu dekodowanie parametru o numerze polozenie
    // z ciągu tekstowego na typ int
  }
  private void wykonajRozkaz(int rozkaz, int parametr) {
    // tu wykonanie instrukcji rozkaz
    // z parametrem parametr
  }

  public void WykonajZestawInstrukcji(String instrukcje) {
    int n = 0;
    int r, p;
    r = dekodujRozkaz(instrukcje, n);
    p = dekodujParametr(instrukcje, n);
    while (r>0) {
      wykonajRozkaz(r, p);
      n++;
      r = dekodujRozkaz(instrukcje, n);
      p = dekodujParametr(instrukcje, n);
    }
  }
}
```

Na listingu 2.57 wytłuszczona została metoda `wykonajZestawInstrukcji` pochodząca z interfejsu `Przekaznik`. Można też zauważyć, że poza `dekodujRozkaz`, `dekodujParametr` i `wykonajRozkaz` klasa nie zawiera żadnych innych metod charakterystycznych dla samej klasy `UrzadzenieZdalnieSterowane`.

Stosując interfejsy, należy pamiętać o mogącym pojawić się problemie wynikającym z konfliktu nazw metod. Jakkolwiek Java umożliwia przeciążanie metod, może się zdarzyć, że interfejsy implementowane w danej klasie dostarczają metody o tej samej nazwie, tym samym zestawie parametrów i różnym wyniku. Kompilator nie jest w stanie obsłużyć takiego przeciążenia metod, więc generuje błąd semantyki języka. Problem ten szerzej opisałem w paragrafie 2.4.6. „Dziedziczenie interfejsów".

2.4.3. Zastosowanie interfejsów

Wyobraźmy sobie, że tworzymy oprogramowanie do zarządzania inteligentnym budynkiem. Inwestor zakupił do tego budynku bardzo wymyślne urządzenie klimatyzacyjne, do którego dołączony był sterownik obsługi napisany w Javie. Oczywiście zgodnie z konwencją języka jest to w pełni funkcjonalna klasa, która udostępnia programiście metody `ustawTemperature`, `ustawWilgotnosc` i `ustawSileNadmuchu`. Klasa ta jest w pełni funkcjonalna, to znaczy możemy użytkownikowi naszego programu udostępnić korzystanie z każdej funkcji klimatyzatora, ale tylko w sposób manualny, to znaczy może on zmieniać tylko jeden parametr w danym momencie i nie może zażądać, aby na przykład temperatura zmieniła się, ale dopiero za godzinę. Nie ma się co dziwić, w końcu jest to tylko sterownik do fizycznego urządzenia. Pełną funkcjonalność można uzyskać dopiero po wprowadzeniu własnego, bardziej zaawansowanego programu. Moglibyśmy oczywiście utworzyć nową klasę dziedziczącą własności po klasie sterownika klimatyzacji i w jej rozszerzeniu zamieścić nowe metody, które umożliwiłyby stworzenie programatora zmian. Nie jesteśmy jednak początkującymi programistami. Poprzednio wykonywaliśmy bardzo podobną pracę dla małej lokalnej kotłowni. Mamy więc już opracowaną klasę, która doskonale nadaje się do tworzenia listy instrukcji typu *o tej godzinie mam wykonać taką akcję*. Projektując klasę `EdyorPolecen`, przewidzieliśmy jego przyszłe wykorzystanie za pośrednictwem pokazanego wcześniej interfejsu `Przekaznik`. Skorzystaliśmy z możliwości tworzenia zmiennych o typie takim jak nazwa interfejsu. Oczywiście nie oznacza to, że jest to egzemplarz typu interfejs. Jest to egzemplarz dowolnej klasy (implementującej ten interfejs), której na dodatek nie musimy znać w chwili deklarowania jej użycia. Na listingu 2.58 przedstawiam fragment klasy `EdytorPolecen` używający klasy implementującej interfejs `Przekaznik`.

Listing 2.58. *Przykład użycia interfejsu*

```
class EdytorPolecen {
  // deklaracja użycia klasy
  // implementującej Przekaznik
  Przekaznik p;
  // konstruktor klasy
  EdytorPolecen(Przekaznik p) {
    this.p = p;
  }
  // metoda pokazująca okno dialogu z użytkownikiem
  // oraz zdalnie uruchamiająca urządzenie
  // reprezentowane przez obiekt p
  public void showDialog() {
    boolean kolejnykrok = true;
    String instrukcje;
```

```
    while (kolejnykrok) {
      // tu odbywa się cała komunikacja z użytkownikiem,
      // w tym ustawienie wartości zmiennej kolejnykrok
      // oraz instrukcje
      p.WykonajZestawInstrukcji(instrukcje);
    }
  }
  // dalsza część klasy
}
```

Jak widać, do zmiennej `p` odwołujemy się tak, jak do najzwyklejszego obiektu. Jest to jak najbardziej słuszne, gdyż jest to obiekt takiego typu, który implementuje interfejs `Przekaznik`. Oczywiście należy pamiętać, aby zainicjować go we właściwy sposób i przekazać go w konstruktorze tworzącym obiekt typu `EdytorPolecen`. Fragment kodu programu, który wykonuje taką akcję, pokazany jest na listingu 2.59 (kod klasy `UrzadzenieZdalneSterowane` znajduje się wcześniej).

Listing 2.59. *Przykład odwołania do obiektu implementującego interfejs*

```
UrzadzenieZdalneSterowane k;
EdytorPolecen p;
k = new UrzadzenieZdalneSterowane();
p = new EdytorPolecen(p);
p.showDialog();
```

Jeśli odpowiednio dobrze zaprojektowaliśmy klasę `EdytorPolecen`, może się okazać, że nie musimy jej modyfikować w razie wykorzystania z zupełnie nowym, nieznanym w czasie jej tworzenia urządzeniem.

Odszukiwanie metod zdefiniowanych w interfejsie odbywa się w sposób dynamiczny, to znaczy interpreter Javy w czasie napotkania na odwołanie do metody należącej do interfejsu przegląda listę wszystkich metod obiektu, który jest użyty w miejscu wskazania na interfejs. Na skutek takiego odwoływania się do metod ten fragment programu może być nawet w znaczny sposób spowolniony. W związku z tym nie powinno się stosować interfejsów wszędzie tam, gdzie jest to wskazane ze względu na krytyczne szybkości wykonania oraz możliwość zastąpienia ich klasami.

2.4.4. Stałe symboliczne

Interfejsów można używać do deklarowania często wykorzystywanych wartości, które w innych językach nazywane są stałymi. Java nie udostępnia mechanizmu deklarowania stałych globalnych podobnego do deklaracji `const` w Object Pascalu czy instrukcji preprocesora `#define` w C++. Jednak stosowanie stałych symbolicznych w wielu przypadkach ułatwia modyfikowanie programów. Można dać wiele przykładów, w których pewna stała wartość determinująca działanie algorytmu występuje w całym programie kilkadziesiąt bądź kilkaset razy. Gdyby używać jej jawnie w postaci wartości liczbowej, jej zmiana w całym programie mogłaby być niezmiernie uciążliwa. Naprzeciw temu zapotrzebowaniu wychodzą specyficzne możliwości stosowania interfejsów. Jak wspominałem w trakcie wprowadzania informacji o strukturze interfejsu, poza deklaracją istnienia metod może on posiadać również pola, które mają status

zmiennych finalnych. Można więc zadeklarować stałe w podobny sposób, jak działo się to w innych językach obiektowych, tak jak na listingu 2.60.

Listing 2.60. *Deklaracja stałych symbolicznych w interfejsie*

```
interface PrzyciskiPl {
  String mbYes = "Tak";
  String mbNo = "Nie";
  String mbCancel = "Anuluj";
}
```

Ten sam zestaw przycisków, lecz w wersji angielskiej znajduje się na listingu 2.61.

Listing 2.61. *Deklaracja stałych symbolicznych w interfejsie*

```
interface PrzyciskiEn {
  String mbYes = "Yes";
  String mbNo = "No";
  String mbCancel = "Cancel";
}
```

Wykorzystanie stałych zadeklarowanych w interfejsie wymaga dodania do obiektu słowa `implements` z właściwą nazwą interfejsu, tak jak na listingu 2.62.

Listing 2.62. *Użycie stałych symbolicznych*

```
public class DemoApplet extends Applet
  implements PrzyciskiPl {

  Button buttonYes = new Button();
  Button buttonNo = new Button();
  Button buttonCancel = new Button();

  public void init() {
    buttonYes.setLabel(mbYes);
    add(button1);
    buttonNo.setLabel(mbNo);
    add(button2);
    buttonCancel.setLabel(mbCancel);
    add(button3);
  }
}
```

Zmiana wersji językowej wymaga w tym przykładzie wyłącznie zmiany nazwy używanego interfejsu w deklaracji klasy. Aplet z angielską wersją językową przycisków wymagałby jedynie deklaracji, co pokazałem na listingu 2.63 poprzez wytłuszczenie zmiany w nagłówku.

Listing 2.63. *Zmiana języka poprzez niewielką modyfikację nagłówka klasy*

```
public class DemoApplet extends Applet
  implements PrzyciskiEn {
  // ciało dokładnie takie samo
}
```

Takie podejście do stałych jest znacznie wygodniejsze od stosowanego w innych językach, gdzie wymagana jest fizyczna podmiana wartości stałych, zamiast stosowanej w Javie podmiany interfejsu, którego używamy do stworzenia klasy. Jeśli jednak programista przyzwyczajony jest do klasycznego układu stosowania stałych w programie, może to zrobić bez deklarowania implementacji interfejsu w tworzonej klasie, tak jak na listingu 2.64.

Listing 2.64. *Użycie stałych symbolicznych bez implementacji interfejsu*

```
interface Interfejs {
  int JEDEN = 1;
  int DWA = 2;
}

class TestInt {
  int i = Interfejs.JEDEN;
}
```

Takie zastosowanie, pozornie mniej atrakcyjne, ma zaletę grupowania stałych w pewnego rodzaju kontenery, którymi są interfejsy. Można z tego skorzystać, na przykład tworząc dwa interfejsy — z nazwami miesięcy i z liczbą dni w miesiącu. Oba będą miały pola o dokładnie takich samych nazwach, lecz różnym typie i znaczeniu. Może to w niektórych sytuacjach bardzo uprościć kod źródłowy:

```
print(Nazwy.STYCZEN + " ma " + Dni.STYCZEN + " dni");
```

Jak się można domyślić, interfejs `Nazwy` deklaruje nazwy miesięcy, a `Dni` liczbę dni w miesiącu.

2.4.5. Trochę kodu w interfejsie

Wprowadzając interfejsy, zwracałem uwagę na to, że jest to szczególna konstrukcja, która nie zawiera żadnego kodu, a jedynie deklaracje istnienia metod. Nie jest to do końca prawda, ponieważ wewnątrz interfejsów można definiować klasy statyczne. Wewnątrz tych klas może znajdować się dość duży kod, a nawet odwołania do metod innych klas, pod warunkiem że są one statyczne. Tak więc niejako bocznymi drzwiami możemy wprowadzić do interfejsu trochę kodu, tak jak w przykładzie zaprezentowanym na listingu 2.65.

Listing 2.65. *Przemycanie kodu do interfejsu*

```
import java.applet.*;

interface Interfejs {
  class S {
    { System.err.println("static S"); }
    S() {
      System.err.println("konstruktor S");
    }
  }
  S s = new S();
}
```

```
public class Applet2 extends Applet {
  class A implements Interfejs { }
  A a1 = new A();
  A a2 = new A();
  static {
    System.err.println("inicjator klasy");
  }
  {
    System.err.println("inicjator obiektu");
  }
  public Applet2() {
    System.err.println("konstruktor obiektu");
  }
}
```

Ten fragment kodu wygeneruje na konsoli Javy w przeglądarce komunikaty w kolejności pokazanej na rysunku 2.8.

Rysunek 2.8. *Wydruk generowany przez program 2.65*

```
inicjator klasy
static S
konstruktor S
inicjator obiektu
konstruktor obiektu
```

Zmiana miejsca implementacji interfejsu, jak to pokazałem na listingu 2.66, będzie skutkowała inną kolejnością wykonania kodu.

Listing 2.66. *Inne umieszczenie kodu w interfejsie*

```
public class Applet2 extends Applet
  implements Interfejs{
  static {
    System.err.println("inicjator klasy");
  }
  {
    System.err.println("inicjator obiektu");
  }
  public Applet2() {
    System.err.println("konstruktor obiektu");
  }
}
```

Efekt działania zmiany pokazałem na rysunku 2.9.

Rysunek 2.9. *Wydruk generowany przez program z listingu 2.66*

```
static S
konstruktor S
inicjator klasy
inicjator obiektu
konstruktor obiektu
```

Należy jednak pamiętać, że zmienne w interfejsach są traktowane jako finalne, tak więc umieszczony w nich kod będzie się wykonywał tylko jeden raz w czasie ładowania klasy. Jeśli klasa implementuje większą liczbę interfejsów zawierających kod,

to kolejność wykonania kodu będzie zgodna z kolejnością wymienionych do implementacji interfejsów. Powyższe użycie kodu w interfejsie może być użyteczne na przykład jako wyświetlanie nazwy naszej biblioteki w sprzedawanym pakiecie.

Innym sposobem wstawienia kodu do interfejsu, który może być wykonany bez jego implementacji w metodzie, jest wstawienie do niego egzemplarza klasy rozszerzającego inny interfejs ze zdefiniowanym wewnątrz kodem. Przykład takiego łamańca wraz z jego użyciem pokazany jest na listingu 2.67.

Listing 2.67. *Kolejny sposób na umieszczenie kodu w interfejsie*

```
import java.applet.*;

interface InsideInt {
  interface Method {
    int run(int i);
  }

  Method stub = new Method() {
    public int run(int i) {
      System.err.println("Parametr przekazany: " + i);
      return i+1;
    }
  };
}

public class AppletTst extends Applet
  implements InsideInt {

  public void init() {
    int i;
    i = this.stub.run(17);
    i = this.stub.run(i);
  }
}
```

Wytłuszczony fragment w interfejsie odpowiada za stworzenie metody, którą będziemy w przyszłości wykorzystywać, co pokazane jest w aplecie również z użyciem wytłuszczenia. Jak widać, mimo tego że klasa `AppletTst` implementuje interfejs `InsideInt`, możemy w niej korzystać z metod w nim zadeklarowanych. Zaletą takiego udziwnionego stosowania kodu jest to, że nie możemy modyfikować tej metody, a więc zmieniać jej działania, ale możemy implementować różne funkcjonalności w jednej metodzie, co lubią niektórzy programiści.

2.4.6. Dziedziczenie interfejsów

Podobnie jak klasy, interfejsy mogą podlegać dziedziczeniu. Jednak dziedziczenie interfejsów jest bardziej rozbudowane, a mianowicie w procesie tym nie występuje ograniczenie co do jednokrotnego dziedziczenia. Tak jak na listingu 2.68 interfejs może więc dziedziczyć po większej liczbie interfejsów.

Listing 2.68. *Dziedziczenie interfejsu po wielu interfejsach*

```
interface Interfejs {
  void metoda(int i);
}
interface Interfejs1 {
  void metoda1(int i);
}
interface Interfejs2
  extends Interfejs, Interfejs1 {
  void metoda2(int i);
}
```

Tak jak w przypadku klasycznego dziedziczenia ostatni w łańcuchu interfejs, czyli w naszym przykładzie `Interfejs2`, posiada wszystkie cechy interfejsów, po których dziedziczy, to znaczy jego implementacja wymaga uwzględnienia wszystkich metod w równorzędny sposób, tak jak to pokazałem na listingu 2.69.

Listing 2.69. *Implementacja interfejsu o wielu przodkach*

```
import java.applet.*;

public class Applet2 extends Applet
  implements Interfejs2 {
  public void metoda(int i) {
    // ciało metody
  }
  public void metoda1(int i) {
    // ciało metody
  }
  public void metoda2(int i) {
    // ciało metody
  }
}
```

Gdyby okazało się, że w łańcuchu wielokrotnego dziedziczenia któraś z metod ma dokładnie taką samą nazwę i dokładnie taki sam zestaw parametrów, w obiekcie implementującym umieszcza się ją tylko jeden raz. Problem pojawia się w przypadku, gdy interfejsy implementowane w danej klasie dostarczają metody o tej samej nazwie, tym samym zestawie parametrów i różnym typie wartości zwracanej. Kompilator nie jest w stanie stworzyć takiego przeciążenia metody, więc generuje błąd semantyki języka. Warto jednak zauważyć, że błąd jest generowany dopiero na etapie kompilacji klasy. Tak więc konstrukcja pokazana na listingu 2.70 będzie poprawna z punktu widzenia kompilatora, choć całkowicie bezsensowna z punktu widzenia języka.

Listing 2.70. *Błędne wielokrotne dziedziczenie interfejsów*

```
interface Interfejs {
  void metoda(int i);
}
interface Interfejs1 {
  int metoda(int i);
}
interface Interfejs2
  extends Interfejs, Interfejs1 { }
```

Dzieje się tak dlatego, że formalnie rzecz biorąc, powyższy zapis jest równoważny deklaracji z listingu 2.71.

Listing 2.71. *Pozorna postać interfejsu z listingu 2.70*

```
interface Interfejs2 {
  void metoda(int i);
  int metoda(int i);
}
```

Kompilator na szczęście tego już nie zaakceptuje, gdyż przeciążenie z punktu widzenia typu wyniku nie jest dopuszczalne.

2.4.7. Egzemplarz interfejsu

Pośrednio wspominałem już o tym, wprowadzając ideę przyświecającą powstaniu interfejsów. Przedstawię to teraz dokładnie, zwracając uwagę na konsekwencje tego faktu. Interfejs może zostać wykorzystany jako identyfikator typu deklarowanej zmiennej. „Może" to mało powiedziane. Idea użycia interfejsów bazuje właśnie na tym, że w miejscu, gdzie będziemy potrzebowali użyć klasy, która będzie implementować jakiś interfejs jako typ zmiennej, użyjemy interfejsu, a nie nieznanej sobie klasy. Użycie interfejsu jako typu zmiennej wynika z tego, że nie znamy rzeczywistego typu klasy, którą będziemy wykorzystywać, a może się też zdarzyć, że nie wiemy nawet, po jakiej klasie ona dziedziczy. Deklarujemy więc zmienną typu interfejs według klasycznego wzorca (`ActionListener` jest nazwą interfejsu):

```
ActionListener lst;
```

Oczywiście to, że zmienna jest typu zgodnego z interfejsem, nie oznacza, że możemy stworzyć zmienną z użyciem interfejsu. Konstrukcja pokazana poniżej, używająca interfejsu wykorzystywanego przy tworzeniu obiektów nasłuchujących w AWT jest błędna i nie da się jej użyć.

```
ActionListener a = new ActionListener();  // błąd
```

Jeśli stworzyliśmy klasę, która implementuje ten interfejs:

```
class ALclass extends Button
  implements ActionListener {
  //...
}
```

wtedy dopiero możemy jej użyć do stworzenia obiektu:

```
ActionListener a = new ALclass();
```

Najczęściej jednak w takim przypadku nie tworzymy obiektu od zera z użyciem operatora `new`, ale odbieramy go od użytkownika końcowego z użyciem parametru jakiejś metody. Możemy zadeklarować, że metoda ta będzie przekazywała obiekt typu interfejs:

```
void dodajLst(ActionListener lst) {
  addActionListener(lst);
}
```

W takim przypadku dbaniem o to, czy przekazujemy do tej metody właściwy obiekt, zajmie się kompilator. Dokładnie w taki sposób ustawiane jest wskazanie na pole `stub` w klasie `Applet`:

```
private AppletStub stub;

public final void setStub(AppletStub stub) {
  this.stub = (AppletStub)stub;
}
```

W zaprezentowanym przykładzie występuje bardzo ciekawa sytuacja. Otóż pole `stub` przechowuje wskazanie na obiekt implementujący interfejs `AppletStub`, który znajduje się poza apletem, czyli poza całym programem uruchamianym w środowisku przeglądarki. Obiekt ten istnieje bezpośrednio tylko w JVM i poza przechowywanym wskazaniem nie mamy do niego żadnego dostępu (nie jest jawnie tworzony w żadnym miejscu programu, ani w żadnej części biblioteki).

Poza scedowaniem na kompilator sprawdzania typu obiektu możemy postąpić inaczej i samodzielnie przejąć rozpoznanie odbieranego obiektu pod kątem implementacji konkretnego interfejsu. Należy do tego wykorzystać operator `instanceof` w sposób analogiczny jak w przypadku zwykłej klasy, na przykład tak jak na listingu 2.72.

Listing 2.72. *Sprawdzanie typu interfejsu*

```
void dodajLst(Object lst) {
  if (lst instanceof ActionListener) {
    addActionListener(lst);
  }
  if (lst instanceof MouseListener) {
    addMouseListener(lst);
  }
}
```

W zaprezentowanym przykładzie typ interfejsu rozpoznawany jest wewnątrz metody `dodajLst`. Na podstawie tego typu dodawany jest właściwy obiekt obsługi zdarzeń. Jeśli obiekt nie da się rozpoznać jako żaden z obiektów nasłuchujących, metoda nie wykonuje żadnej akcji.

Więcej przykładów używania zmiennych o typie interfejsu znajduje się w rozdziale 4. „Programowanie sterowane zdarzeniami”. W zrozumieniu tego problemu pomoże Ci też paragraf 2.5.2. „Jawna klasa anonimowa”.

2.5. Klasy anonimowe

Klasy anonimowe są bardzo silnie powiązane z procesem tworzenia ich egzemplarzy (czyli obiektów). Jako że kwestia ta silnie łączy ze sobą definicję klasy z jej egzemplarzem (czyli obiektem), postanowiłem opisać ją samodzielnie, nie włączając tego problemu ani do podrozdziału opisującego tworzenie klas, ani obiektów. Klasa anonimowa to pełnoprawna klasa wewnętrzna lub lokalna, niezawierająca nazwy oraz

konstruktora, która posiada jedynie niewielkie ograniczenia w stosunku do zwykłej (nazwanej) klasy. Jednak sens jej stosowania występuje wyłącznie w przypadku, gdy klasa ma być niewielka, prosta w użyciu oraz stosowana jednorazowo bądź w bardzo ograniczony sposób. Ze względu na ograniczenia formalne nadaje się ona wyłącznie do implementacji interfejsów bądź reimplementacji metod istniejących w klasach bazowych. Stosowanie dużych i rozbudowanych klas anonimowych oczywiście jest możliwe, w praktyce jednak znacznie zaciemnia kod źródłowy. Jest to raczej w sprzeczności z ideą programowania obiektowego, która zakłada upraszczanie kodu przez ukrywanie szczegółów implementacyjnych. Stosowanie klas anonimowych nie wpływa również w żaden sposób na wielkość czy efektywność kodu wynikowego (czasami nawet można poprawić sytuację, rezygnując z klas anonimowych, co pokażę w rozdziale 4. „Programowanie sterowane zdarzeniami"). Mimo tych zastrzeżeń klasy anonimowe stosuje się dość często, zwłaszcza w mechanizmach obsługi zdarzeń.

2.5.1. Klasyczne użycie klasy anonimowej

Klasa anonimowa to klasa definiowana w miejscu, w którym tworzymy jej egzemplarz. Podobnie jak w przypadku klasycznych obiektów możemy je tworzyć w sposób jawny, tak aby następnie były dostępne przez nazwę, bądź niejawny, na przykład w czasie przekazywania ich do dalszego użycia wewnątrz innych klas. Najczęściej wykorzystuje się tę drugą metodę, gdyż w naturalny sposób łączy się ona z cechami klasy anonimowej. Tworzona jest na jednorazowy użytek za pomocą jednostkowej definicji, która nigdy więcej w tej postaci nie będzie używana. Klasa anonimowa jest tworzona według szablonu pokazanego na listingu 2.73.

Listing 2.73. *Schemat tworzenia klasy anonimowej*

```
new
KlasaBadzInterfejsBazowy
([lista argumentów])
{
ciało klasy anonimowej
}
```

Jak widać, klasa anonimowa może zostać zdefiniowana wyłącznie w chwili tworzenia jej egzemplarza, po wystąpieniu operatora `new`. Po tym operatorze musi wystąpić definicja typu, po którym dziedziczy klasa anonimowa. Może to być nazwa klasy bazowej bądź interfejsu. Jeśli podajemy nazwę interfejsu, domyślnie oznacza to, że klasa anonimowa dziedziczy po klasie `Object` i implementuje wymieniony interfejs. Nie jest możliwa implementacja większej liczby interfejsów niż jeden ani dziedziczenie po klasie innej niż `Object` i jednoczesna implementacja interfejsu. Przykład klasycznej definicji klasy anonimowej pokazany jest na listingu 2.74.

Listing 2.74. *Klasyczna definicja klasy anonimowej*

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

```
public class Applet2 extends Applet {
  Button b = new Button("przycisk");
  public void init() {
    b.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          // obsługa zdarzenia e
        }
      }
    );
    add(b, null);
  }
}
```

Wytłuszczony fragment to definicja klasy anonimowej dziedziczącej (niejawnie) po klasie `Object` i implementującej interfejs `ActionListener`. Interfejs ten deklaruje istnienie metody `actionPerformed`, która jest zdefiniowana w pokazanej klasie anonimowej. Klasa ta będzie wykorzystana w obiekcie typu `Button` jako klasa nasłuchująca zdarzeń pochodzących od przyciśniętego przycisku ekranowego. W chwili pojawienia się tego zdarzenia wywoływana będzie metoda `actionPerformed` klasy podawanej w metodzie `addActionListener`. W innych popularnych językach obiektowych, takich jak Object Pascal czy C++, w takim wypadku przekazywalibyśmy adres procedury, która byłaby wywoływana we właściwym momencie. Java jest tak mocno obiektowa, że jest to niemożliwe. Zamiast tego przekazuje się adres obiektu, który w tym wypadku jest nienazwanego typu. W związku z tym, że takie działanie wykonywane jest jednorazowo dla przyspieszenia tworzenia kodu, stosuje się klasę anonimową. Konstrukcja taka nie wpływa na czytelność kodu, jednak przyjęła się wśród osób używających Javy. Więcej na temat użycia klasy anonimowej i przykłady eliminacji anonimowości znajdziesz w rozdziale 4. „Programowanie sterowane zdarzeniami". Na listingu 2.75 zaprezentowałem jeszcze jeden przykład użycia klasy anonimowej. Pozornie nie wnosi on nic nowego poza zaciemnieniem kodu. W praktyce umożliwia wyodrębnienie i zgrupowanie kodu w jeden blok, podobnie jak podprogramy czy procedury lokalne w klasycznym programowaniu strukturalnym.

Listing 2.75. *Użycie klasy anonimowej do zgrupowania kodu*

```
(new Object() {
  // definicja pól
  public void make() {
    // działania w metodzie make
  }
}).make();
```

Przedstawiona klasa anonimowa gromadzi pewne instrukcje wewnątrz metody `make`, umożliwiając dostęp do lokalnych pól tej klasy. Stosowania takiej konstrukcji ma sens w przypadku, gdy chcemy stworzyć silniejszą hermetyzację wewnątrz pojedynczej metody.

2.5.2. Jawna klasa anonimowa

Najczęściej stosuje się klasy anonimowe, które nie posiadają nazwanych egzemplarzy. Nic jednak nie stoi na przeszkodzie temu, aby tworzyć klasy anonimowe z ich użyciem. Fakt, że egzemplarz klasy jest nazwany, nie wpływa oczywiście na anonimowość jej typu, co nie zawsze i nie przez wszystkich jest zrozumiałe. W praktyce więc pokazany na listingu 2.76 przykład również zawiera klasę anonimową (wytłuszczony fragment jest deklaracją klasy dziedziczącej po `Object` i implementującej interfejs `ActionListener`).

Listing 2.76. *Klasa anonimowa o nazwanym egzemplarzu*

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

public class Applet2 extends Applet {
  Button b = new Button("przycisk");
  ActionListener a = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      // obsługa zdarzenia e
    }
  };
  public void init() {
    b.addActionListener(a);
    add(b, null);
  }
}
```

Pokazany przykład ułatwi rozważania na temat sposobu rozszerzania klas za pomocą mechanizmu klas anonimowych. Wprowadzając interfejsy, zwracałem uwagę na to, że możemy stosować zmienne typu zgodnego z typem interfejsu. Tak właśnie stało się w przykładzie pokazanym na listingu 2.76. Może on być użyty w miejscu, gdzie wymagany jest taki typ (na przykład w metodzie `addActionListener` we wcześniejszym przykładzie), bądź jawnie, na przykład:

```
a.actionPerformed(null);
```

Ważne jest, aby struktura klasy anonimowej była znana kompilatorowi. Oznacza to, że oprócz interfejsu klasa anonimowa może redefiniować istniejącą klasę. Nie ma natomiast większego sensu wprowadzanie nowych metod do klasy anonimowej, gdyż później nie da się ich wykorzystać. Przykład klasy definiującej metodę `make`, której nie da się następnie użyć, pokazany jest na listingu 2.77.

Listing 2.77. *Niecelowe nazywanie egzemplarza klasy anonimowej*

```
Object a = new Object() {
  public void make() {
    // ciało metody make
  }
};
// błąd: make nie istnieje w klasie Object
a.make();
```

Możemy próbować różnych sztuczek w celu użycia metody `make`, na przykład zastosować refleksje bądź używać nazwy klas tworzonej przez kompilator typu `Applet$1` jako określającej typ klasy, jednak rozwiązania te są bardzo czasochłonne i skomplikowane w kodzie, tak że nie opłaca się używać.

2.5.3. Konstruktor klasy anonimowej

Większość osób jest przekonana, że klasa anonimowa nie może posiadać konstruktora. Tezy takie wynikają najczęściej z porównania klasy anonimowej ze zwykłą. W zwykłej (nazwanej) klasie konstruktor jest to metoda, która posiada nazwę zgodną z nazwą klasy. Oczywiste jest, że skoro klasa nie ma nazwy, to nie będzie mogła mieć metody o tej samej nazwie. Należy jednak pamiętać, że poza konstruktorem istnieją takie narzędzia jak inicjator klasy i inicjator egzemplarza klasy, czyli obiektu. Z powodzeniem możemy zastosować go jako namiastkę konstruktora. Biorąc pod uwagę, że klasa anonimowa jest najczęściej bardzo zubożona, trudno spodziewać się w jej przypadku nadmiernie rozbudowanych konstruktorów. W związku z tym niemal zawsze w zastępstwie konstruktora może występować inicjator egzemplarza, tak jak na listingu 2.78.

Listing 2.78. *Namiastka konstruktora w klasie anonimowej*

```
interface Anonim {
  void make();
}

Anonim a = new Anonim() {
  // pola klasy implementującej Anonim
  {
    // inicjator obiektu (w tym pól klasy anonimowej)
  }
  public void make() {
    // ciało klasy Make
  }
};
```

W przypadku konieczności przekazania parametrów do inicjatora klasy anonimowej możemy skorzystać z lokalnych zmiennych finalnych, jak na listingu 2.79.

Listing 2.79. *Przekazanie parametrów do inicjatora klasy anonimowej*

```
final int i = 111;
Anonim a = new Anonim() {
  int j;
  {
    j = i;
  }
  public void make() {
    System.err.println("i ma wartość: " + i);
  }
};
```

Jak to pokazałem w metodzie `make`, do niej również możemy przekazać dane z zewnątrz klasy anonimowej z użyciem zmiennej finalnej spoza klasy. Taka możliwość dodatkowo ogranicza liczbę przypadków, kiedy używanie inicjatora obiektu jest konieczne.

2.6. Obiekty refleksyjne

W podrozdziale 2.2. „Obiekty" pokazałem sposoby używania obiektów o typie znanym i zdefiniowanym w czasie tworzenia kodu programu. Znajomość typu oznacza w tym wypadku to, że kompilator jest w stanie odwołać się w czasie kompilacji do pliku typu `class` zawierającego definicję klasy, która posłuży do utworzenia jej egzemplarza. Nie zawsze sytuacja taka jest możliwa. Może się okazać, że właściwy plik z definicją będzie dostępny w późniejszym terminie (jest na przykład tworzony przez drugi zespół programistyczny). Innym przykładem zastosowania może być wykorzystanie gotowych klas w wizualnych środowiskach programistycznych. Środowisko takie powinno obsługiwać dowolne komponenty, nawet jeśli powstały już po utworzeniu całego korzystającego z nich programu. Obiekty refleksyjne wykorzystuje się również jako sterowniki do komunikacji z urządzeniami i programami zewnętrznymi. Dzięki tej metodzie tworzenia obiektów skompilowany program może się porozumieć z innym skompilowanym programem interaktywnie — bez konieczności tworzenia w tym celu dedykowanego kodu. Innym ważnym zastosowaniem refleksji jest możliwość uruchomienia odrębnego programu napisanego w Javie znanego wyłącznie z nazwy. Refleksji należy używać wszędzie tam, gdzie zamierzamy tworzyć egzemplarze klas, które znamy wyłącznie z nazwy.

Dzięki wprowadzeniu mechanizmu refleksji Java, jako jeden z niewielu języków programowania, umożliwia tworzenie i używanie obiektów na przykład na podstawie nazw klas wpisanych ręcznie z użyciem klawiatury w trakcie działania programu. Należy jednak pamiętać, że obiekty refleksyjne nie są czystym wykorzystaniem możliwości języka, lecz biblioteki Javy. Mimo tego zdecydowałem się umieścić opis tego problemu w tym miejscu ze względu na pokrewną tematykę zagadnienia.

2.6.1. Obiekt tworzony refleksyjnie

Najprostszym sposobem wykorzystania refleksji jest skorzystanie z konstrukcji:

```
Class.forName("nieznana_klasa").newInstance();
```

Zaprezentowany wiersz programu tworzy obiekt typu określonego przez klasę znajdującą się w pliku `nieznana_klasa.class`. W celu utworzenia tego obiektu wykorzystuje bezparametrowy konstruktor tej klasy. W związku z tym, że nazwa pliku jest zgodna z nazwą klasy, która jest główną klasą tego pliku, utworzony obiekt jest typu właśnie `nieznana_klasa`. Twórcy klasy `Class`, która odpowiada za kreowanie obiektów w trybie refleksyjnym, musieli poradzić sobie z problemem obsługi klas dowolnych, niezaprojektowanych jeszcze typów. Metoda `newInstance` zwraca więc wskazanie

na nowo utworzony obiekt, jednak jest ono typu `Object`. Autorzy tej klasy skorzystali tu z bezpiecznej konwersji typu zwanej też rozszerzeniem zakresu zmiennej. To znaczy otworzony obiekt typu `nieznana_klasa` zostaje przekazywany jako obiekt typu `Object`. W praktyce więc powinno się używać wcześniejszej konstrukcji w postaci:

```
Object o;
o = Class.forName("nieznana_klasa").newInstance();
```

Jakkolwiek konstrukcja ta wydaje się bardziej przydatna, w praktyce wymaga bardziej zaawansowanych działań opisanych dalej. Samo wskazanie na obiekt typu `Object` jest bowiem w tej prostej postaci niemal nieprzydatne. Nieprzydatne, gdyż klasa `Object` nie posiada własności, które tak naprawdę nas interesują. Aby można było użyć tego wskazania, powinno się dokonać tak zwanej niebezpiecznej konwersji typu, czyli zawężenia zakresu zmiennej:

```
Nieznana_klasa nk;
nk = (Nieznana_klasa)o;
```

Rozwiązanie takie jest jednak niemożliwe, ponieważ w chwili tworzenia programu typ `nieznana_klasa` nie jest dla nas (i dla kompilatora) dostępny. Nie możemy utworzyć typu, którego nie znamy. Często więc tworzy się obiekty refleksyjne wyłącznie w celu skorzystania z kodu, który umieszczony jest w bezparametrowym konstruktorze. Jeśli używamy obiektów refleksyjnych jako sterowników, można temu zaradzić, stosując wytyczne dla twórców klas, które będą używane w ten sposób. Tak na przykład dzieje się w przypadku sterowników JDBC (*Java DataBase Connectivity*). W specyfikacji opisującej ten standard wymagane jest, aby tworzący się obiekt sterownika utworzył wskazanie do obiektu `DriverManager`, który będzie w przyszłości wykorzystywany do obsługi bazy danych. Przykład praktycznego wykorzystania obiektu refleksyjnego w obsłudze baz danych pokazany jest na listingu 2.80.

Listing 2.80. *Użycie obiektu refleksyjnego w obsłudze baz danych*

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

public class sqlApplet extends Applet {
  public ResultSet rst = null;
  public List sqlWynik = new java.awt.List(10);
  Button button1 = new Button();

  public void init() {
    String url = "jdbc:AvenirDriver://marekwi_nt5:1433/mwi";
    Class.forName("net.avenir.jdbcdriver7.Driver").newInstance();
    DriverManager.setLoginTimeout(30);
    Connection ctn = DriverManager.getConnection(url, "konto", "haslo");
    Statement stmt = ctn.createStatement();
    boolean moreResult = stmt.execute("select * from tabela_SQL");
    int updateCount = stmt.getUpdateCount();
    rst = stmt.getResultSet();
    sqlWynik.add("nagłówek bazy");
    button1.setLabel("Naciśnij, żeby pobrać kolejny rekord");
    button1.addActionListener(new java.awt.event.ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
          button1_actionPerformed(e);
        }
      });
      this.add(sqlWynik, null);
      this.add(button1, null);
    }

    void button1_actionPerformed(ActionEvent e) {
      String str;
      try {
        if(rst.next()) {
          str = rst.getObject(1).toString() + "-" + rst.getObject(2).toString();
          sqlWynik.add(str);
        }
      } catch(Exception ee) {
        ee.printStackTrace();
      }
    }
  }
```

Jak widać, w całym aplecie tylko jeden wiersz (wytłuszczony) odpowiada za powstanie obiektu refleksyjnego. Pozostała część kodu bazuje na specyfikacji JDBC znanej twórcom sterownika obsługi baz danych (no i oczywiście programistom). Takie rozwiązanie jest nagminnie stosowane właśnie w trakcie tworzenia sterowników zewnętrznych urządzeń czy programów. Dzięki niemu użytkownik dowolnej bazy danych musi znać wyłącznie nazwę sterownika skojarzoną z posiadaną bazą, by po modyfikacji dwóch wierszy programu móc używać go w nowej wersji.

2.6.2. Ogólne rozpoznawanie klasy

Poprzedni paragraf pokazał, jak wykorzystać ideę refleksji do stworzenia własnej specyfikacji (na przykład sterowników) i wykorzystania nieznanych obiektów do z góry zaprojektowanej akcji. Nie zawsze jednak jesteśmy w tak komfortowej sytuacji, że tworzymy standard, który wypełniają inni programiści. Czasami bywa tak, że musimy wykorzystać klasę, której zupełnie nie znamy. Do pełnej analizy nieznanego pliku możemy wykorzystać cztery specjalizowane w tym celu klasy: `Class`, `Method`, `Constructor` i `Field`. Przykład najprostszego użycia tych klas pokazany jest na listingu 2.81.

Listing 2.81. *Metody rozpoznawania obiektów refleksyjnych*

```
import java.lang.reflect.*;
import java.applet.*;

public class aplet extends Applet {

  public void init() {
    int i;
    try {
      Class c = Class.forName("net.avenir.jdbcdriver7.Driver");

      System.out.println("");
      System.out.println("Lista metod z argumentami:");
```

```
      Method m[] = c.getDeclaredMethods();
      for (i=0; i<m.length; i++)
        System.err.println(m[i].toString());

      System.out.println("");
      System.out.println("Lista konstruktorów z argumentami:");
      Constructor ctrs[] = c.getDeclaredConstructors();
      for (i=0; i<ctrs.length; i++)
        System.err.println(ctrs[i].toString());

      System.out.println("");
      System.out.println("Lista pól:");
      Field f[] = c.getDeclaredFields();
      for (i=0; i<f.length; i++)
        System.err.println(f[i].toString());
    } catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

Jak widać, ważną rolę w rozpoznawaniu struktury nieznanej klasy odgrywają metody `getDeclaredXxxx` (wytłuszczone na wydruku) należące do klasy `Class`, które zwracają jako wynik odpowiednio listę metod, konstruktorów i pól występujących w badanej klasie. Klasa `Class` jest nadzorcą klas refleksyjnych, która umożliwia dość dokładne zbadanie właściwości analizowanej klasy. Poza najpopularniejszym sposobem tworzenia obiektu tego typu (wytłuszczonego na ostatnim wydruku) można go utworzyć dla typów podstawowych z użyciem konstrukcji:

```
Class c = int.class;
```

Ponadto dla klas usługowych, czyli obudowujących typy podstawowe w klasy, można to zrobić, korzystając z predefiniowanego pola `TYPE`:

```
Class c = Integer.TYPE;
```

Oba sposoby są raczej rzadziej spotykane ze względu na to, że typy te są zawsze dostępne dla kompilatora i nie ma potrzeby używania ich z użyciem refleksji.

2.6.3. Przykład użycia refleksji

Rozpoznanie struktury klasy to tylko pośredni krok do faktycznego użycia obiektu stworzonego refleksyjnie. Pełne wykorzystanie obiektu wymaga używania jego pól i metod. Do wykorzystania w dalszym przykładzie stworzyłem klasę, która (listing 2.82) zawiera tylko dwa pola typu `double`.

Listing 2.82. *Definicja klasy testowej*

```
public class Test {
  public double d1 = 0.1;
  public double d2 = 0.2;
}
```

Najprostszy przykład odczytu, a następnie ustawienia tych pól w klasie `Test` będzie miał postać zaprezentowaną na listingu 2.83.

Listing 2.83. *Refleksyjne operacje na polach klasy testowej*

```
// egzemplarz klasy nadzorującej
Class c = Class.forName("Test");
// egzemplarz obiektu refleksyjnego
Object o = Class.forName("Test").newInstance();

// nadzorcy poszczególnych pól
Field fd1 = c.getField("d1");
Field fd2 = c.getField("d2");

// odczyt pól obiektu klasy test
System.out.println("d1 = " + fd1.getDouble(o));
System.out.println("d2 = " + fd2.getDouble(o));

// ustawienie pól
fd1.setDouble(o, 1.1);
fd2.setDouble(o, 2.2);

// sprawdzenie, czy operacja powiodła się
System.out.println("d1 = " + fd1.getDouble(o));
System.out.println("d2 = " + fd2.getDouble(o));
```

W komentarzach zaznaczyłem znaczenie poszczególnych fragmentów programu. Tworząc przykład, założyłem, że wiem, jakie będą nazwy i typy poszczególnych pól. Oczywiście w praktyce trzeba sprawdzać zarówno ich typy, jak i nazwy (co pokazałem wcześniej) oraz przewidzieć bardziej zaawansowane techniki wychwytywania błędów. Warto zwrócić uwagę na dość dziwną konstrukcję stosowaną w powyższym przykładzie:

```
fd1.getDouble(o);
```

Umożliwia ona odczyt pola obiektu `o` poprzez obiekt nadzorcę refleksji pola. Konstrukcja ta jest o tyle dziwna, że w przypadku większej liczby pól argumentem jest zawsze ten sam obiekt — egzemplarz refleksyjny. Rozróżnienie poszczególnych pól odbywa się poprzez rozróżnienie nadzorcy ich refleksji. Jest to konwencja odwrotna do naturalnego wykorzystania pól w obiekcie, gdzie zawsze używamy tego samego obiektu, natomiast zmieniają się referencje do pól w nim.

Znacznie ciekawszym problemem jest wywoływanie metod z obiektów tworzonych refleksyjnie. W dalszej części tego paragrafu pokażę, jak wywołać metodę `add` klasy z listingu 2.84.

Listing 2.84. *Definicja klasy testowej z metodą*

```
public class Testm {
  public int add(int a, int b) {
    return (a + b);
  }
}
```

Na listingu 2.85 wykorzystam wiedzę o strukturze tej klasy (jednak bez dostępu do definicji jej typu) do uruchomienia jej jedynej metody z użyciem obiektów refleksyjnych.

Listing 2.85. *Refleksyjne użycie klasy testowej z metodą*

```
// egzemplarz klasy nadzorującej
Class cls = Class.forName("Testm");
// egzemplarz obiektu refleksyjnego
Object obj = Class.forName("Testm").newInstance();
// nadzorcy parametrów
Class partypes[] = new Class[2];
partypes[0] = Integer.TYPE;
partypes[1] = Integer.TYPE;
// nadzorca metody
Method meth = cls.getMethod("add", partypes);

// egzemplarze parametrów
Object arglist[] = new Object[2];
arglist[0] = new Integer(11);
arglist[1] = new Integer(22);

// wywołanie metody
Object retobj = meth.invoke(obj, arglist);

// odbiór wyniku
Integer retval = (Integer)retobj;
System.err.println(retval.intValue());
```

Jak widać, technika jest podobna do stosowanej w przypadku obsługi pól, chociaż oczywiście jest nieco bardziej skomplikowana ze względu na konieczność dodatkowej obsługi parametrów wywoływanych funkcji. Zaprezentowany przykład korzystał z wiedzy na temat struktury używanej metody (parametrów i wyniku). W praktyce powyższy przykład powinien być rozszerzony o badanie tych elementów i obsługę ewentualnych błędów. Można na przykład stworzyć z użyciem tej metody zaawansowany kalkulator, w którym użytkownik końcowy mógłby dodawać swoje funkcje automatycznie rozpoznawane na podstawie nazwy.

2.6.4. Związek refleksji z obiektowością

Możliwości refleksji nie rozszerzają w żaden istotny sposób idei programowania obiektowego. Z punktu widzenia tej idei jedyną, acz wątpliwą zaletą jest możliwość dogłębnego sprawdzenia struktury klasy, której nie jesteśmy autorami. Jednak jeśli klasa została dobrze zaprojektowana, to analiza taka nie powinna nam przynieść żadnych dodatkowych korzyści. Wszak hermetyzację i ukrywanie niektórych cech implementacyjnych wprowadzono po to, by nie zaciemniać programu niepotrzebnymi szczegółami, których znajomość nie wnosi żadnej nowej wiedzy do procesu działania tego obiektu.

Stosując refleksje, należy pamiętać o różnicy między egzemplarzem klasy nadzorującej refleksje, a samym egzemplarzem obiektu refleksyjnego. Jeśli nie nastąpi pomieszanie obu pojęć, to nie powinno być problemów z właściwym stosowaniem tej ciekawej cechy Javy udostępnianej przez jej bibliotekę.

2.7. Metody

Metody to połączone w jedną całość zbiory instrukcji, które umożliwiają wykonywanie akcji charakterystycznych dla danego obiektu bądź zmieniają jego stan. Przy opisywaniu sposobów tworzenia klas wprowadziłem informacje dotyczące używania metod jako całości, nie wgłębiając się w szczegółowe zasady działania rządzące ich tworzeniem. Teraz zajmę się kwestiami związanymi z wewnętrzną strukturą metod i sposobem porozumiewania się kodu wewnętrznego metody z kodem, który ją wywołuje. Większość pokazanych tu kwestii dotyczy również konstruktorów, które niemal nie różnią się w swojej budowie od metod.

Metody to kolejny element układanki składającej się na obiektowy sposób programowania. Pozwalają ukrywać bardziej rozbudowane operacje pod nazwą metody. Bardzo podobne znaczenie mają funkcje i procedury w zwykłym programowaniu strukturalnym. Jednak cechą metod jest to, że mogą działać na danych należących do obiektu, czyli do konkretnego egzemplarza klasy. Jeśli w zwykłym programowaniu strukturalnym funkcja odwołuje się do pewnej zmiennej zadeklarowanej poza jej ciałem, to zawsze odwołuje się do tej samej zmiennej (do jednego jej egzemplarza). W programowaniu obiektowym jest inaczej. Tworząc egzemplarze klas, tworzymy zestawy danych związanych z pewną metodą. Możemy tworzyć więcej takich samych egzemplarzy i wywołanie metod wiązać z konkretnymi zestawami danych, czyli obiektami. Dzięki temu metoda może działać za każdym razem na danych zewnętrznych, ale przechowywanych specjalnie na potrzeby tej i innych metod tego obiektu. Kolejnym z celów stosowania metod jest, poza ukrywaniem większej grupy działań pod jedną nazwą, zwiększenie hermetyzacji obiektów. Klasyczna sugestia programowania obiektowego (o której już kilkukrotnie wspominałem w tym rozdziale) mówi, że we wszystkich możliwych sytuacjach w klasach należy stosować pola o statusie prywatnym, to znaczy niedostępnym i niewidzialnym spoza obiektu. Dostęp do tych metod zapewnia się właśnie przez odpowiednie metody ustawiające i zwracające wartość pola.

2.7.1. Zwracanie wartości przez metodę

Metody są uruchamiane jako działania na danym obiekcie. Wykorzystuje się w tym celu referencję obiektu i nazwę tej metody z listą jej parametrów:

```
obiekt.metoda(lista parametrów);
```

bądź

```
zmienna = obiekt.metoda(lista parametrów);
```

To, czy metoda zwraca jakąś wartość czy nie i jakiego typu jest ta wartość, deklarujemy w czasie jej definiowania. Metoda, która nie zwraca żadnej wartości, musi być deklarowana jako posiadająca typ `void`:

```
void metoda() {
  // ciało metody
}
```

Poza deklaracją typu w ciele metody nie muszą być wykonywane żadne czynności związane ze zwrotem parametrów (nie trzeba dodawać żadnych instrukcji). Nieco inaczej jest w metodach zwracających wartość. Na listingu 2.86 jest to wartość typu `int`.

Listing 2.86. *Sposób zwracania wartości przez metodę*

```
int metoda() {
  int cur;
  // ciało metody ustawiające wartość cur
  return cur;
}
```

Poza deklaracją typu konieczne jest przekazanie do miejsca wywołania wartości wyznaczonej w ciele metody z użyciem instrukcji `return`. Przypisanie wartości w miejscu wywołania metody polega na skorzystaniu z kopii zmiennej wymienionej w instrukcji `return`.

Specjalnym wypadkiem jest konstruktor, który deklarujemy bez żadnego typu, ale również bez słowa `void`. W rzeczywistości konstruktor zwraca wartość będącą wskazaniem na egzemplarz klasy, wewnątrz której jest zadeklarowany. Zwracaniem tego parametru zajmuje się jednak JVM, więc nie musimy się o to martwić, a tym bardziej nie musimy używać w nim instrukcji `return`, która jest odpowiedzialna za taką operację.

2.7.2. Przekazywanie parametrów przez wartość

Wywołanie każdej metody wymaga dostarczenia określonej w jej definicji liczby parametrów, przy czym muszą one być dokładnie tego samego typu, jak jest to określone. Pod pojęciem dokładnie tego samego typu ukrywa się też przypadek, gdy na parametrach może zostać zastosowana konwersja typu (przypadek ten dotyczy przekazywania obiektów). Pewnym pozornym wyłomem są metody przeciążone, gdzie występuje pewna dowolność w liczbie i typach parametrów. Dowolność występuje jednak wyłącznie w zakresie przewidzianym w czasie projektowania metod o tej samej nazwie.

Standardowo parametry metod w Javie przekazywane są przez wartość. Oznacza to, że w czasie wywołania metody tworzone są kopie zmiennych, które przekazuje się do wnętrza metody pod postacią parametrów. Jeśli więc wywołamy metodę `dodaj1` z parametrem typu `int`, to zmiana wartości tego parametru wewnątrz metody nie wpłynie na wartość zmiennej użytej w wywołaniu. Do zaprezentowania problemu posłuży przykład z listingu 2.87.

Listing 2.87. *Obiekt prezentujący zasięg zmiennych*

```
import java.applet.*;

public class Applet2 extends Applet {
  public void init() {
    int i = 1;
    System.err.println("wartość przed:    " + i);
    dodaj1(i);
    System.err.println("wartość po:       " + i);
  }
```

```
  void dodaj1(int j) {
    j = j + 1;
    System.err.println("wartość wewnątrz: " + j);
  }
}
```

Uruchomienie tego przykładu spowoduje wyświetlenie na konsoli Javy trzech wierszy tekstu pokazanych na rysunku 2.10.

Rysunek 2.10. *Wydruk generowany przez program 2.87*

```
wartość przed:    1
wartość wewnątrz: 2
wartość po:       1
```

Jak widać, zmiana wartości kopii nie wpłynęła na wartość oryginału przekazywaną do metody. Sytuacja nie zmieniłaby się nawet wtedy, gdyby nazwa parametru i pola pokrywały się.

2.7.3. Zmiana wartości parametru

Formalnie rzecz ujmując, parametry są przekazywane do metod przez wartość. Nie jest więc możliwa zmiana pewnej wartości wewnątrz metody tak, aby było to widziane na zewnątrz. Można tego dokonać, zwracając ten sam parametr w postaci wartości funkcji. Można też skorzystać z cechy Javy, o której pisałem w podrozdziale 2.2.3. „Kopiowanie obiektów". Otóż w momencie wywołania tworzona jest kopia zmiennej przekazywanej do metody. Jeśli jest to obiekt, to tworzona jest jego kopia (a nie jest on klonowany). Obiektu tego nie możemy zmodyfikować jako całości, to znaczy nie możemy podstawić w to miejsce innego, bądź go zwolnić tak, aby efekt był widoczny poza metodą. Jak jednak wcześniej pokazałem, modyfikowanie wartości pól kopii obiektu wpływa na zmianę wartości również oryginału. Stąd już bliska droga do mechanizmu znanego w innych językach, czyli zmiany wartości parametru wewnątrz metody. W przykładzie z listingu 2.88 wykorzystam specjalnie w tym celu utworzoną klasę `Int`, choć można również korzystać z klas dostępnych w bibliotekach Javy, w tym z usługowych powielających typy proste.

Listing 2.88. *Zmiana parametru wewnątrz metody widziana na zewnątrz*

```
class Int { int i; }

Int i = new Int();
i.i = 1;
System.err.println("wartość przed:    " + i.i);
dodaj1(i);
System.err.println("wartość po:       " + i.i);
//...

void dodaj1(Int j) {
  j.i = j.i + 1;
  System.err.println("wartość wewnątrz: " + j.i);
}
```

Wynik działania tego programu to wyświetlone na konsoli Javy napisy (rysunek 2.11):

Rysunek 2.11. *Wydruk generowany przez program 2.88*

```
wartość przed:    1
wartość wewnątrz: 2
wartość po:       2
```

Ważne jednak jest, że do wnętrza metody `dodaj1` należy przekazać wskazanie do obiektu, a nie samo pole. Na listingu 2.89 zastosowano błędne podejście.

Listing 2.89. *Błędne podejście do modyfikacji wartości przez metodę*

```
class Int { int i; }

Int i = new Int();
i.i = 1;
System.err.println("wartość przed:    " + i.i);
dodaj1(i.i);  // błędne przekazanie wartości pola
System.err.println("wartość po:       " + i.i);
//...

void dodaj1(int j) { // konieczność modyfikacji
  j = j + 1;
  System.err.println("wartość wewnątrz: " + j);
}
```

Efekt działania takiego błędnego podejścia byłby taki, jak w klasycznym przekazywaniu parametru przez wartość (rysunek 2.12).

Rysunek 2.12. *Wydruk generowany przez program z listingu 2.89*

```
wartość przed:    1
wartość wewnątrz: 2
wartość po:       1
```

Należy pamiętać o tym, że to nie przynależność do obiektu powoduje inne traktowanie parametru, lecz przekazywanie adresu do kopii obiektu, a następnie działanie na jej polach, co jest równoznaczne z działaniami na samym obiekcie. Gdybyśmy chcieli przekazywać do metody obiekt w taki sposób, aby efekt działania na nim nie był widoczny na zewnątrz metody, należałoby przekazywać do niej jego klon.

Identyczne działanie jak przekazywanie referencji do obiektu można osiągnąć, przekazując referencję do tablicy (nawet jednoelementowej). Dzięki takiej sztuczce można zmusić Javę do modyfikacji wartości parametru bez przekazywania go jako obiekt, co pokazałem na listingu 2.90.

Listing 2.90. *Modyfikacja wartości parametru niebędącego obiektem*

```
int[] a = new int[1];
a[0] = 15;
razy2(a);
// a[0] ma teraz wartość 30

void razy2(int[] a) {
  a[0] = 2 * a[0];
}
```

Jak widać, jest to prostsze rozwiązanie niż zastosowanie obiektów i doskonale sprawdza się zwłaszcza w prostych działaniach. Trzeba jednak pamiętać, że stosując to rozwiązanie, należy bardzo uważać, gdyż nie kontrolujemy w nim parametrów (w metodzie `razy2` nie sprawdziłem, jaka wielka jest przekazywana do niej tablica).

2.7.4. Metody ze zmienną liczbą parametrów

Przeciążanie metod jest sposobem na konstruowanie obiektów z większą liczbą metod o tej samej nazwie, lecz różnym zestawie parametrów. Kwestia ta była już omówiona w paragrafie 2.1.5. „Przeciążanie metod". Z punktu widzenia konstrukcji metod ich przeciążanie nie jest jednak żadnym szczególnym przypadkiem. Polega po prostu na budowaniu kilku metod o różnym zestawie parametrów. Ciekawszym rozwiązaniem są metody o zmiennej liczbie parametrów, choć formalnie, o czym wcześniej już pisałem, Java nie umożliwia ich stosowania. Możemy jednak zastosować rozwiązanie, które umożliwi przesłanie ich w takiej liczbie, jaka jest akurat wymagana w momencie wywołania takiej nietypowej metody. Klasyczny sposób przekazywania parametrów bazuje na ich stałej, określonej w czasie definiowania metody liczbie. Niewielkim nakładem pracy można jednak skorzystać z mechanizmu dynamicznego przekazywania parametrów, który umożliwi oderwanie się od stosowania stałej ich liczby. Metody takie są nagminnie wykorzystywane w innych językach bez szkody, a często nawet z korzyścią dla przejrzystości, łatwości i szybkości tworzenia kodu. W Javie mechanizm taki, jakkolwiek dostępny, nie jest zbytnio eksponowany. W związku z tym, że uważam metody ze zmienną liczbą parametrów za mniej niebezpieczne od mechanizmu ich przeciążania, przedstawię na listingu 2.91 sposób realizacji tego przypadku.

Listing 2.91. *Przekazywanie do metody zmiennej liczby parametrów*

```
void zmienneParametry(Object params[]) {
  for (int i=0; i<params.length; i++) {
    System.err.println(params[i].getClass());
    System.err.println(params[i]);
  }
}
```

Formalnie zaprezentowana metoda odbiera tablicę obiektów. Jeśli jednak uwzględni się fakt, że standardowa biblioteka klas Javy, obowiązkowo znajdująca się w każdym środowisku tego języka, zawiera zestaw klas usługowych odpowiadających typom prostym, można przyjąć, że do metody możemy przekazać każdy rodzaj parametrów:

```
ZmienneParametry(new Object[]
  {"tekst", new Boolean(true), new Integer(5)});
```

Wynik działania tak wywołanej funkcji zaprezentowany jest na rysunku 2.13.

Rysunek 2.13. *Efekt przekazania różnych parametrów do metody*

```
class java.lang.String
tekst
class java.lang.Boolean
true
class java.lang.Integer
5
```

Jak widać, wewnątrz metody można określić typ przekazanych do niej parametrów, co wydatnie podnosi funkcjonalność tego rozwiązania. Do pełnego określenia typu obiektu przesyłanego do metody powinno użyć się operatora `instanceof`, o którym pisałem w podrozdziale 2.2.5. „Typ zmiennej i obiektu. Operator instanceof".

2.7.5. Zakres nazw zmiennych

Opisując sposób tworzenia metod, chciałbym wspomnieć też o zakresie widzialności w ich obrębie nazw zmiennych i pól. Zmienna zadeklarowana w obrębie metody bądź parametr o nazwie takiej jak jej pole są wewnątrz niej „ważniejsze" niż nazwa pola klasy. Na listingu 2.92 znajduje się podwojona deklaracja o nazwie `use`.

Listing 2.92. *Przesłanianie zmiennych o tej samej nazwie*

```
class Test {
  int use;
  void testM() {
    int use;
    // operacje na use
  }
}
```

Powoduje ona, że operacje na zmiennej `use` wewnątrz metody `testM` nie będą miały wpływu na stan pola `use` obiektu. Gdybyśmy chcieli mieć dostęp jednocześnie do zmiennej i do pola, musielibyśmy użyć (tak jak ja w wytłuszczonym wierszu na listingu 2.93) znanej już referencji `this` do bieżącego obiektu.

Listing 2.93. *Jednoczesny dostęp do zmiennej i pola o tej samej nazwie*

```
class Test {
  int use;
  void testM() {
    int use = 0;
    // podstawienie zmiennej lokalnej do pola:
    this.use = use;
  }
}
```

Takie podejście ma zapewnić bezpieczeństwo w czasie rozszerzania i dziedziczenia klas. Może się bowiem okazać, że w czasie rozszerzenia w jednej z nowych metod wymagane jest zastosowanie nazwy zmiennej zgodnej z nazwą pola. Aby nie zmodyfikować przez przypadek wartości pola, Java przykrywa tę nazwę z użyciem lokalnej zmiennej, na której wykonywane są wszystkie lokalne operacje.

2.8. Pakiety

Zaprezentowane dotychczas przykłady klas w Javie były bardzo skromne. W związku z tym nie pojawiły się jak na razie żadne problemy wynikające z konfliktów nazw między różnymi klasami. W rzeczywistych dużych projektach niepowtarzalność nazw jest trudna do zrealizowania. Często może się okazać, że potrzebujemy stworzyć dwie klasy lub więcej, dla których najbardziej adekwatna nazwa jest już gdzieś użyta. Gdyby nie mechanizm zarządzania nazwami poszczególnych klas szybko okazałoby się, że nie możemy już stworzyć żadnej nowej o logicznej nazwie. Jeśli uwzględnić możliwość importu klas napisanych przez jednego z wielu twórców udostępniających zasoby w internecie, wkrótce mogłoby również zabraknąć nazw złożonych z zupełnie przypadkowych znaków. Aby zaradzić temu problemowi, wprowadzony został mechanizm pakietów. Umożliwia on stosowanie takich samych nazw klas z zastrzeżeniem, że należą do różnych pakietów. W praktyce oznacza to na przykład, że możemy utworzyć trzy klasy o nazwie `Grid` tworzące i obsługujące siatki elementów różnego rodzaju. Jedną dla danych tekstowych, jedną dla obrazków i jedną dla przycisków (siatka przycisków ekranowych to na przykład kalkulator). Pierwsza z klas należałaby do pakietu `texts`, druga `pictures`, a trzecia `buttons`. Dzięki temu nie tylko możemy zastosować takie same nazwy, ale nazwa pakietu lepiej identyfikuje naszą klasę.

2.8.1. Tworzenie pakietów

Każdy plik z klasą Javy poza nią samą może zawierać łącznie cztery różne sekcje tworzące kompletny i w pełni funkcjonalny plik. Sekcje te nie są w żaden sposób wyróżnione. Podział ten jest tworzony wyłącznie na podstawie zawartości pliku pogrupowanej według schematu z listingu 2.94.

Listing 2.94. *Struktura pliku definiującego klasę*

```
[deklaracja przynależności do pakietu]
[deklaracje użycia innych pakietów]
deklaracja klasy publicznej
[deklaracje klas prywatnych]
```

Elementy ujęte w nawiasy kwadratowe są opcjonalne. Tak więc klasa nie musi należeć do żadnego pakietu i nie musi żadnego z nich importować, a przynajmniej tak się wydaje. Jednak powiedzenie, że klasa nie należy do żadnego pakietu, nie jest prawdziwe, bo wszystkie do jakiegoś należą. Jeśli nie jest on jawnie nazwany, wtedy klasa należy do nienazwanego pakietu domyślnego. Ogólna postać deklaracji przynależności klasy do pakietu wygląda tak:

```
package pakiet1[.pakiet2[.pakiet3]];
```

Wytłuszczony fragment linii to słowo kluczowe informujące kompilator, że następująca po tym słowie sekwencja jest nazwą pakietu. Pakiety mogą być grupowane w struktury hierarchiczne z użyciem kolejnych pakietów według podobnej zasady, jak należą do nich klasy. Tak więc w skrajnym przypadku kilka klas należy do pakietu rzędu pierwszego. Kilka pakietów rzędu pierwszego należy do drugiego. Strukturę można

zagłębiać dalej. Kolejne poziomy hierarchii oddzielone są w kodzie źródłowym kropkami. Struktura nazw pakietów musi odpowiadać strukturze i położeniu plików w podkatalogach. Tak więc jeśli główny plik aplikacji znajduje się w katalogu `c:\java\src`, wtedy pliki z klasami należącymi do pakietu `grafika` muszą leżeć w katalogu `c:\java\src\grafika`, a pliki z pakietu `grafika.obrazki` muszą znajdować się w katalogu `c:\java\src\grafika\obrazki`. W praktyce może okazać się, że pliki z klasami należącymi do tych pakietów leżą w rzeczywistości w innych katalogach. Katalog źródłowy ustalany jest bowiem na podstawie zmiennej środowiskowej bądź parametru `CLASSPATH`. Jeśli więc zmienna ta wskazuje na dwa różne katalogi, na przykład `c:\java\ibm\src` i `c:\java\sun\src`, wtedy klasy pakietu `grafika` mogą znajdować się zarówno w katalogu `c:\java\ibm\src\grafika`, jak i w `c:\java\sun\src\grafika`. Jeśli nie wymienimy jawnie nazwy pakietu, do którego należy dana klasa, oznacza to, że klasy będą szukane wyłącznie w katalogach wskazywanych przez `CLASSPATH`, a w niektórych wersjach Javy i jej kompilatorów także w katalogu bieżącym. W praktyce bardzo często stosuje się kompresję wszystkich tworzonych przez nas klas do jednego archiwum (typu `jar` bądź `zip`) z użyciem hierarchicznej struktury katalogów. Zmienne `CLASSPATH` bądź odpowiedni parametr uruchomienia apletu wskazują wtedy na konkretny plik `jar`, wewnątrz którego znajdują się właściwe klasy we właściwych podkatalogach. Aby pokazać zagnieżdżenie pakietów oraz możliwość stosowania w nich takich samych nazw bez wprowadzania konfliktów, prezentuję na listingu 2.95 zestaw pakietów dostępnych w Javie 1.1.

Listing 2.95. *Lista pakietów dostępnych w Javie 1.1*

```
java.applet
java.awt
java.awt.datatransfer
java.awt.event
java.awt.image
java.beans
java.io
java.lang
java.lang.reflect
java.math
java.net
java.rmi
java.rmi.dgc
java.rmi.registry
java.rmi.server
java.security
java.security.acl
java.security.interfaces
java.sql
java.text
java.util
java.util.zip
```

Warto zauważyć, że często zdarza się, że na tym samym poziomie zagnieżdżenia występują zarówno klasy, jak i kolejne pakiety. Dla przykładu mogą wystąpić pliki należące do pakietu `java.util` oraz pakiet do niego należący. Daje to możliwość lepszego wydzielania i gromadzenia klas w sposób bardziej oczywisty i zgodny z modelem hermetyzacji obiektowej.

2.8.2. Używanie pakietów

Po wprowadzeniu informacji, w jaki sposób tworzyć klasy, aby nie występowały między nimi konflikty nazw i znajdowały się w różnych pakietach, pokażę, w jaki sposób użyć tych pakietów. Jak to wcześniej pokazałem, plik zawierający klasę Javy posiada sekcję deklarującą używanie pakietów. Sekcja ta składa się z dowolnej liczby linii o postaci:

```
import pakiet[.pakiet[.pakiet]].klasa;
```

Każda z takich linii informuje kompilator oraz maszynę Javy o miejscu, gdzie należy szukać elementów używanych przez projektowaną klasę. Ostatnim słowem w hierarchii pakietów dołączanej do bieżącego pliku jest klasa, której zamierzamy używać. Może to być konkretna klasa, której nazwę znamy, bądź * (gwiazdka) oznaczająca, że importujemy wszystkie elementy z tego pakietu (to znaczy wszystkie klasy i wszystkie interfejsy). Najczęściej stosuje się gwiazdkę, jednak dla bardzo dużych projektów nie jest to rozwiązanie optymalne. Zastosowanie importu wszystkich klas z pakietu powoduje, że kompilator w czasie tworzenia programu przegląda wszystkie pakiety, z których umożliwiamy użycie wszystkich klas. Najczęściej trwa to dość długo, dlatego warto maksymalnie ograniczyć zakres importowanych klas. Nadmierny import nie ma jednak wpływu na wielkość wynikową klasy i szybkość działania programu. Klasy należące do tego samego pakietu co bieżąca nie muszą być importowane.

Hierarchia pakietów jest odnoszona w stosunku do ustawienia zmiennej `CLASSPATH` oraz rzeczywistej struktury katalogów. Oznacza to, że pakiet:

```
grafika.przyciski
```

przy ustawieniu:

```
CLASSPATH=c:\java\src\
```

powinien znajdować się w katalogu:

```
c:\java\src\grafika\przyciski\
```

Jakkolwiek wszystkie klasy pakietu, do którego należy bieżąca klasa, są widziane z poziomu jej pliku, to widzialność ta nie przenosi się na podpakiety. To znaczy jeśli (przy zaprezentowanym wcześniej ustawieniu zmiennej `CLASSPATH`) mamy klasę:

```
package aa.bb;
class B { }
```

znajdującą się w katalogu `c:\java\src\aa\bb`, wtedy klasa `A` dziedzicząca po `B`, znajdująca się w pakiecie `bb` i umieszczona w katalogu `C:\java\src\aa` musi mieć postać:

```
package aa;
import aa.bb.*;
class A extends B { }
```

W deklaracji `import` musimy wymienić pełny dostęp do pakietu `aa.bb`, gdzie znajduje się klasa `B`, po której dziedziczy `A`, mimo tego iż formalnie znajduje się ona tylko o jeden szczebel niżej. Wydawałoby się więc, że można dokonać importu z użyciem deklaracji:

```
import bb.*;
```

Byłoby to możliwe, gdyby zmienna CLASSPATH zawierała poza wskazaniem na konkretne katalogi wskazanie na katalog bieżący. Można tego dokonać, dodając do tej zmiennej kropkę, która reprezentuje właśnie katalog bieżący. Warto wiedzieć, że niektóre kompilatory Javy domyślnie działają tak, jakby użytkownik korzystał z takiego ustawienia. Powinno się o tym pamiętać przy zmianie kompilatora, może się bowiem okazać, że projekt, który dotychczas kompilował się zupełnie poprawnie, przestaje się kompilować ze względu na zmianę traktowania bieżącego katalogu.

Warto też zauważyć, że import nie dotyczy podkatalogów i podpakietów. Gdybyśmy więc chcieli w programie użyć zarówno klasy A, jak i jej nadrzędnej B, musimy zaimportować oba pakiety:

```
import aa.*;   // aa.bb.* nie jest importowane
import aa.bb.*;
```

Stosowanie deklaracji import nie jest jednak jedynym sposobem na skorzystanie z klas zdefiniowanych w odrębnych pakietach. Klas tych można używać, stosując pełne nazwy z zachowaniem całej hierarchii pakietów. Pokazany wcześniej przykład klasy A dziedziczącej własności po B, która zdefiniowana została w pakiecie aa.bb bez deklaracji importu, będzie miał postać:

```
package aa;
class A extends aa.bb.B { }
```

Jak widać, można było zrezygnować z deklaracji importu na rzecz rozszerzenia nazwy klasy o nazwę pakietu (na przykładzie wytłuszczone).

Proste programy i aplety, które zawierają stosunkowo niewielką liczbę nowych projektowanych komponentów, mogą być tworzone w ramach pakietów bez nazwy. Wszystkie klasy należące do projektu z domyślnym pakietem będą wtedy pozbawione linii z deklaracją package, która deklaruje przynależność klasy do pakietu. Wszystkie te klasy będą także widoczne w całym projekcje bez konieczności importowania ich. W takim przypadku wszystkie klasy muszą znajdować się w jednym katalogu dyskowym. Często spotyka się konwencję, że klasy użytkowe stanowiące bibliotekę użytecznych komponentów umieszczane są w pakietach, które tworzą biblioteki użytkownika, natomiast ostateczny projekt zawiera się w pakiecie bez nazwy. Takie podejście jest realizacją myślenia obiektowego na poziomie całych komponentów. Projekt główny jest widoczny dla programisty w sposób naturalny, natomiast wszystkie jego potrzebne elementy, których szczegóły implementacji nie są dla niego istotne, są ukryte w pakietach. Korzystając z modyfikatorów zakresu widzialności, możemy te szczegóły udostępniać bądź dalej ukrywać zgodnie z własną koncepcją hermetyzacji.

2.8.3. Lista pakietów

Jeśli tworzymy aplikacje bądź aplety z wykorzystaniem Javy 1.3 lub wyższej, możemy skorzystać z klasy Package, która umożliwi nam wyświetlenie pożytecznych informacji na temat wszystkich używanych pakietów. Może ona być na przykład użyta do sprawdzenia, które klasy należy dołączyć do archiwum jar zawierającego niestan-

dardowe klasy (niedostępne w przeglądarkach). Metoda wyświetlająca listę używanych pakietów powinna być skonstruowana według schematu z listingu 2.96.

Listing 2.96. *Metoda wyświetlająca listę używanych pakietów*

```
void listPackages() {
  Package[] allPackages = Package.getPackages();
  System.out.println("All loaded packages:");
  for(int i=0; i < allPackages.length; i++) {
    System.err.println(allPackages[i].getName());
    System.err.println("  " + allPackages[i].getImplementationTitle());
    System.err.println("  " + allPackages[i].getImplementationVersion());
}}
```

Informacje na temat pakietów pobierane są przez obiekt typu `ClassLoader` z archiwum `jar`, z pliku `manifest.mf`. Przykładowy opis może wyglądać tak, jak to pokazuję na rysunku 2.14.

Rysunek 2.14. *Przykładowa zawartość manifestu pliku jar*

```
Name: java/mwi/grafika/3d
Specification-Title: Grafika 3D
Specification-Version: 1.0
Specification-Vendor: Sun
Implementation-Title: Najlepsza grafika 3D
Implementation-Version: 1.2.1
Implementation-Vendor: Helion
```

Tekst znajdujący się przed dwukropkiem definiuje nazwę parametru, a ten po dwukropku — jego wartość. W pakiecie możemy przekazywać łącznie 7 parametrów, przy czym zdarza się, że `ClassLoader` nie czyta ostatniej linii specyfikacji. Formalnie, jeśli brakuje danego wiersza, zwracana wartość jest ciągiem pustym. Przy całkowitym braku opisu pakietu zwracane wartości mogą mieć postać `null`.

2.9. Modyfikatory

Po wprowadzeniu wszystkich elementów obiektowych mogę przystąpić teraz do wyjaśnienia znaczenia poszczególnych modyfikatorów, które wpływają na rozszerzenie funkcjonalności Javy bądź też uczynienie jej jeszcze bardziej obiektowej. W dotychczasowych rozważaniach, poza drobnymi wyjątkami, nie brałem w żaden sposób pod uwagę kwestii istnienia modyfikatorów. Są to słowa kluczowe występujące przed deklaracjami klas, pól oraz metod uściślające sposób działania lub zakres widzialności elementu, którego dotyczą. Modyfikatory te są opcjonalne. Jeśli ich nie ma, kompilator przyjmuje domyślne ustawienia języka dla poszczególnych fragmentów kodu. Ich stosowanie powoduje zmianę własności poszczególnych elementów i dzięki temu jest to silne narzędzie wpływające na realizację wszystkich trzech elementów idei programowania obiektowego, czyli hermetyzacji, polimorfizmu i dziedziczenia. W związku z brakiem modyfikatorów niektóre z wcześniej wprowadzonych pojęć mogły się wydawać niepełne bądź niespełniające różnorodnych oczekiwań użytkowników. Dopiero

wprowadzenie modyfikatorów umożliwi rozwinięcie pełnych możliwości programowania obiektowego. Przedstawiając modyfikatory, będę się starał zwracać Ci uwagę na miejsca, w których wniosą one nowe możliwości i gdzie należy zweryfikować swoje poglądy na temat już wprowadzonych konstrukcji. W Javie możemy rozróżnić modyfikatory zmieniające przestrzeń nazw oraz funkcjonalność elementu. Na przestrzeń nazw wpływają modyfikatory `private`, `public` i `protected`. Na działanie i własności wpływają słowa `static`, `final`, `volatile`, `synchronized`, `transient` i `native`. Modyfikatory mogą występować w dowolnej kolejności i łączyć się we w miarę dowolne grupy, z zastrzeżeniem, że modyfikator przestrzeni może wystąpić tylko raz, to znaczy nie można stworzyć pola, które będzie jednocześnie na przykład rodzaju `public` i `private`.

2.9.1. Modyfikatory dostępu

Brak modyfikatorów jest równoznaczny ze stosowaniem dla klas, pól i metod domyślnego sposobu ich traktowania. W praktyce oznacza to, że sterowanie dostępnością do poszczególnych elementów odbywa się za pomocą właściwego rozmieszczania tych elementów w klasach i pakietach. W Javie rozróżniamy pięć obszarów i zakresów dostępności elementów. Tabela 2.1 określa zakres widzialności elementu w zależności od zastosowanego modyfikatora.

Tabela 2.1 *Zakres widzialności elementów w zależności od modyfikatora*

Modyfikator	Wnętrze klasy, w której zadeklarowany jest dany element	Podklasy znajdujące się w tym samym pakiecie	Inne klasy znajdujące się w tym samym pakiecie	Podklasy znajdujące się w innych pakietach	Inne klasy znajdujące się w innych pakietach
private	Tak	Nie	Nie	Nie	Nie
(bez modyfikatora)	Tak	Tak	Tak	Nie	Nie
protected	Tak	Tak	Tak	Tak	Nie
public	Tak	Tak	Tak	Tak	Tak

Z niewiadomych powodów twórcy Javy zrezygnowali w nowszej specyfikacji języka z możliwości używania ciekawego modyfikatora, który był kombinacją `private` i `protected` i był dostępny wyłącznie w Javie 1.0. Umożliwiał on ustawienie widzialności elementu wyłącznie w podklasach, bez względu na to, czy należały one do tego samego pakietu czy do innych. Nieznacznie różniło się to od standardu `protected`, który poza widzialnością w podklasach zapewnia widzialność w innych klasach tego samego pakietu.

Analizując modyfikator `private` w odniesieniu do klas lokalnych, należy zwrócić uwagę, że nie działa on w pełni tak, jakbyśmy oczekiwali. Listing 2.97 ukazuje klasę lokalną zadeklarowaną z metodą prywatną.

Listing 2.97. *Klasa lokalna z metodą prywatną*

```
public int i;
private class Lokalna {
  private int wynik() { return i; }
}
```

Takie rozwiązanie powoduje, że metoda `wynik` — prywatna dla tej klasy — jest jednak widoczna poza nią. Kompilator nie zgłosi błędu przy kompilacji fragmentu programu z listingu 2.98.

Listing 2.98. *Niekonsekwencja prywatności w klasie z listingu 2.97*

```
void doSth() {
  i = 100;
  Lokalna l = new Lokalna();
  System.err.println("wynik: " + l.wynik());
}
```

Metody i pola prywatne z założenia nie powinny być widoczne poza klasą, tak więc osobiście oczekiwałem, że metoda `wynik` nie będzie widzialna dla kompilatora. We wczesnych subwersjach Javy 1.1 pola i metody prywatne klas lokalnych były traktowane inaczej, to znaczy pola prywatne były niewidoczne, a metody prywatne widoczne. Błąd ten naprawiono i w wyższych wersjach zarówno pola, jak i metody prywatne są widoczne poza klasą. Moim zdaniem jest to niezgodne z konwencją modyfikatora `private`. Korzystając z niego w klasach lokalnych, należy więc pamiętać o tej nieścisłości. Na modyfikator `private` zwracałem też uwagę w paragrafach 2.1.4. „Hermetyzacja i modyfikator private” oraz 2.1.11. „Kolejność inicjacji klas”.

2.9.2. Pokrywanie modyfikatorów dostępu

Raz nadany zakres dostępu może zostać w przyszłości rozszerzony. Oznacza to, że jeśli zadeklarowaliśmy metodę bez modyfikatora, metoda pokrywająca może również tak być zadeklarowana oraz z modyfikatorami `protected` lub `public`. Jeśli deklarujemy metodę jako `protected`, pokrywając ją, możemy (poza tym samym modyfikatorem) użyć modyfikatora `public`. Oczywiście możemy również przedeklarować zakres `private` na dowolny inny. Jakkolwiek konwencja jest zachowana w stosunku do wymienionych wcześniej układów, wynika to z faktu, że element prywatny jest niewidoczny poza swoją klasą. Odwrotna sytuacja nie jest możliwa, to znaczy zakres widzialności nie może zostać zawężony. Tak więc, jeśli deklarujemy klasę:

```
public class A {
  int mi() {
    return 0;
  }
}
```

wtedy poniższa deklaracja jest poprawna:

```
public class B extends A {
  public int mi() { return 0; }
}
```

natomiast następna już nie:

```
public class B extends A {
  private int mi() { return 0; } // błąd kompilacji
}
```

Wyjątek dotyczy jedynie zakresu widzialności klas. Oznacza to, że jeśli mamy w pewnym pakiecie deklarację klasy:

```
package ppp;
public class P { }
```

wtedy możemy zawęzić widzialność klasy potomnej w następujący sposób:

```
package ppp;
class Q extends P { }
// zakres widzialności tylko w obrębie pakietu
```

Przy tak zadeklarowanych klasach próba użycia:

```
import ppp.*;
class Z {
  P p = new P();
  Q q = new Q();   // klasa niewidoczna
}
```

w przypadku klasy `Q` zakończy się niepowodzeniem.

Wróćmy jednak do widzialności metod. Zadeklarujmy klasy `A` i `B` w sposób podany na listingu 2.99.

Listing 2.99. *Przykładowa deklaracja dwóch klas*

```
package ppp;
public class A {
  int mi() { return 0; }
}

package ppp;
public class B extends A {
  public int mi() { return 0; }
}
```

Jak się należało spodziewać, próba użycia metody `mi` obiektu `a` pokazana na listingu 2.100 nie powiedzie się, gdyż metoda ta będzie niedostępna.

Listing 2.100. *Błędna próba użycia klas z listingu 2.99*

```
import ppp.*;

class C {
  A a = new A();
  B b = new B();
  void init() {
    int i;
    i = a.mi();   // metoda niedostępna
```

```
    i = b.mi();
  }
}
```

Jednak bardzo dziwne jest to, że również w przypadku próby pokazanej na listingu 2.101 dostęp do tej metody będzie niemożliwy.

Listing 2.101. *Inna próba użycia klas z listingu 2.99*

```
import ppp.*;

class C {
  A a = new B();   // inny sposób tworzenia
  B b = new B();
  void init() {
    int i;
    i = a.mi();   // metoda niedostępna
    i = b.mi();
  }
}
```

Mam wrażenie, że w tym przypadku, jakkolwiek referencja do metody `mi` dotyczy klasy `B`, prawo dostępu do niej jest pobierane z typu zadeklarowanego, a nie rzeczywiście używanego. Jest to według mnie pewna niespójność pomiędzy rzeczywistym typem rozpoznawanym przez JVM, a jego dostępnością rozpoznawaną przez kompilator, do której należy się dostosować.

2.9.3. Metody i pola statyczne

Poza modyfikatorami zakresu widzialności w Javie występują modyfikatory określające sposoby zachowania się bądź umożliwiające określenie nietypowych cech pól lub metod. Jednym z takich modyfikatorów jest słowo kluczowe `static`. Może ono być stosowane łącznie z modyfikatorami zakresu, to znaczy pole lub metoda statyczna może być zarówno publiczna, prywatna, jak i domyślna. Jakkolwiek istnieją pewne różnice w znaczeniu, które nadaje to słowo polom i metodom, jedno jest wspólne. Otóż zarówno pola, jak i metody statyczne mogą być użyte bez konieczności tworzenia egzemplarza klasy. To znaczy zamiast tworzyć obiekt, możemy odwoływać się do tych pól i metod poprzez nazwę klasy i nazwę pola bądź metody. Takie rozwiązanie upodabnia metody statyczne do funkcji globalnych w programowaniu strukturalnym, a pola do zmiennych globalnych. Zaletą jest jednak grupowanie tych zmiennych i funkcji w grupy o wspólnej nazwie głównej. Umożliwia to lepsze zarządzanie tymi elementami, a co za tym idzie czytelniejszą strukturę programu, co jest jednym z głównych celów obiektowego sposobu tworzenia aplikacji. Weźmy deklarację klasy przedstawioną na listingu 2.102.

Listing 2.102. *Metoda statyczna, która może być użyta jak zwykła funkcja*

```
class Globalne {
  static int version = 12;
  public static void libName() {
```

```
    System.err.println("Funkcje globalne");
  }
}
```

Klasy `Globalne` możemy użyć w sposób pokazany na listingu 2.103.

Listing 2.103. *Proste użycie metody statycznej*

```
public void showLibName() {
  Globalne.libName();
  System.err.println("wersja: " + Globalne.version);
}
```

Wytłuszczoną czcionką zaznaczyłem odwołanie bezpośrednio do klasy, bez tworzenia jej egzemplarza.

Poza niestandardowym użyciem klasy pola statyczne są wspólne dla wszystkich egzemplarzy klas, czyli obiektów zawierających pola statyczne. Dzięki tej własności umożliwiają nam swoiste komunikowanie się między obiektami oraz zmianę wartości we wszystkich egzemplarzach klasy za pomocą jednego przypisania. Jeśli mamy przykładową klasę o definicji:

```
class A {
  static int a = 10;
}
```

wtedy przykład z listingu 2.104 działa w sposób niespotykany dla innych przypadków.

Listing 2.104. *Charakterystyczne zachowanie pola statycznego*

```
public void init() {
  // jeden egzemplarz klasy A:
  A a = new A();
  // drugi egzemplarz klasy A:
  A b = new A();
  // zmiana we wszystkich egzemplarzach klasy A:
  a.a = 11;
  System.err.println(a.a);  // 11
  // tu również zmiana we wszystkich egzemplarzach:
  b.A = 12;
  System.err.println(a.a);  // 12
}
```

Pierwsze przypisanie jest bardziej oczywiste i naturalne. Drugie może powodować w niektórych sytuacjach trudne do wykrycia błędy (zmieniamy wartość w jednej z klas, a efekt pojawia się również w innych). Jest to o tyle ważne, że dokładnie taki sam efekt osiągniemy w przypadku zadeklarowania pola `a` klasy `A` jako prywatnego, jak to zrobiłem na listingu 2.105.

Listing 2.105. *Zachowanie prywatnego pola statycznego*

```
class A {
  private static int a = 10;
  public static void setA(int a) { A.a = a; }
```

```
  public static int getA() { return A.a; }
}

public void init() {
  A a = new A();
  A b = new A();
  A.setA(11);
  System.err.println(a.setA()); //11
  b.setA(12);
  System.err.println(a.setA()); //12
}
```

Dzieje się tak dlatego, że na pole statyczne w JVM miejsce jest rezerwowane tylko jeden raz, w chwili definicji klasy. I właśnie tam wpadają wszystkie wartości tego pola bez względu na to, w którym egzemplarzu klasy są używane. Można to traktować jako pewnego rodzaju zmienną globalną dla wszystkich egzemplarzy klasy jednego typu.

W stosunku do pól statycznych należy również pamiętać, że powinno się je inicjować w inicjatorze klasy, czyli bloku instrukcji opatrzonym deskryptorem `static`. Ponadto konstrukcja metod statycznych powinna być zaprojektowana w taki sposób, aby wykorzystywały one wyłącznie inne metody statyczne oraz nie używały wskazań `this` i `super`. Warto też wiedzieć, że deklarując metodę jako statyczną, blokujemy działanie polimorfizmu. Jeśli więc jakaś klasa używa metody statycznej, to będzie używała zawsze „swojej" wersji metody.

2.9.4. Pola finalne

Kolejnym modyfikatorem, który może występować ze słowami określającymi zakres widzialności i zmieniać charakter pola, jest `final`. Umożliwia on tworzenie pól, których wartość nadawana jest tylko jeden raz i nie może być już później zmieniana. Taka cecha jest podobna do zachowania się stałych deklarowanych z użyciem słów `const` w C++ czy Object Pascalu. Pola finalne muszą być inicjowane w chwili ich tworzenia, to znaczy wytłuszczony fragment listingu 2.106 musi występować w deklaracji pola.

Listing 2.106. *Definicja pola finalnego*

```
class A {
  final int a = 10;
}
```

Pola finalne charakteryzują się tym, że wszystkie są przechowywane w pamięci w postaci jednego egzemplarza, dokładnie tak samo jak pola statyczne. Trudno to zauważyć w codziennym użyciu, gdyż pola te muszą być inicjowane, a jak pokazałem to we wcześniejszej części tego rozdziału, dzieje się to przed wywołaniem konstruktora. Ponadto nie mogą one zostać pokryte, jak dzieje się to w przypadku zwykłych pól. Można to wykazać w stosunkowo karkołomnym przykładzie na listingu 2.107.

Listing 2.107. *Błędna próba pokrycia pól finalnych*

```
public void init() {

  class A {
    final int a = 10;
  }

  A[] a = new A[10];
  int i;
  for(i=0; i<10; i++) {
    final int j=i;
    a[i] = new A() {
      final int a = j;
    };
  }
  for (i=0; i<10; i++)
     System.err.println(a[i].a);
}
```

Wbrew oczekiwaniom przykład ten spowoduje wyświetlenie kolumny samych dziesiątek. Warto pamiętać o takim zachowaniu w przypadku stosowania bardziej skomplikowanych konstrukcji językowych.

2.9.5. Metody i klasy finalne

W stosunku do metod i całych klas modyfikator `final` wprowadza cechę podobną do tej, która wystąpiła również dla pól, to znaczy metody nie mogą być pokrywane, a klasy nie mogą być rozszerzane przez dziedziczenie. Cecha ta umożliwia deklarowanie specyficznych klas, które na skutek krytycznej konstrukcji mogłyby działać niepoprawnie w przypadku modyfikacji. Zabezpieczenie przed dalszym rozszerzaniem możemy też stosować w przypadku rozprowadzania własnych obiektów jako bibliotek komponentów w celu wymuszenia pewnych specyficznych sposobów użycia.

Ponadto metody finalne mogą się wykonywać szybciej w przypadku, gdy program został skompilowany z opcją kompilatora `-O`. Opcja ta umożliwia rozwinięcie metod finalnych w miejscu ich wywołania podobnie jak dyrektywa `inline` w C++ czy Borland Pascalu. Taka rozwinięta metoda oszczędza czas interpretera, gdyż nie ma konieczności poszukiwania jej w czasie wykonania programu wśród kodu klas. Ponadto zaoszczędza czas potrzebny na przekazanie parametrów do wnętrza metody i na samo jej wywołanie. W związku z tym warto stosować tę konstrukcję. Powinno się jednak wiedzieć, że takie działanie zwiększa kod wynikowy, co nie zawsze jest korzystne. Poza tym nie ma gwarancji, że kod zostanie rozwinięty, gdyż może się okazać, że kompilator uzna taką akcję za nieopłacalną i zrezygnuje z niej.

2.9.6. Pola podlegające zmianie

W rozdziale 6. „Programowanie wielowątkowe" zaprezentowane zostaną kwestie związane z tworzeniem programów wielowątkowych. W ramach tej części książki należy tylko wspomnieć, że istnieją modyfikatory, które można, a czasami trzeba wykorzystywać w czasie tworzenia programów podzielonych na kilka wątków wykonywanych w tym samym czasie (współbieżnie). Jednym z takich słów jest `volatile`. Modyfikator ten stosuje się wyłącznie w stosunku do pól. Oznacza on, że pole może być modyfikowane w każdym momencie i w każdym fragmencie kodu, nawet jeśli nie wynika to z samej jego treści. Jeśli pole oznaczone jest słowem `volatile`, kompilator, nawet jeśli wykonuje optymalizację, uwzględnia możliwość jego modyfikacji „z zewnątrz". Fragment kodu, w którym stosowanie tego modyfikatora jest szczególnie wskazane, pokazany jest na listingu 2.108.

Listing 2.108. *Kod, w którym powinno się użyć modyfikatora volatile*

```
public void show() {
  volatile current = 1;
  while (true) {
    System.err.println(current);
    // tu wstrzymanie wykonania programu
    // na określony czas
  }
}
```

`current` jest polem klasy, do której należy metoda `show`, widocznym przez wątki wykonywane współbieżnie do tej metody. Gdyby pole to było zadeklarowane bez modyfikatora `volatile`, wtedy kompilator w ramach optymalizacji kodu mógłby zamienić wywołanie:

```
System.err.println(current);
```

na:

```
System.err.println(1);
```

Ta zamiana wyniknęłaby z tego, że wartość `current` jest niezmienna w czasie wykonywania pętli. Umieszczenie w deklaracji pola opisywanego modyfikatora spowoduje, że kompilator uwzględni możliwość, że między kolejnymi wywołaniami wyświetlania wartości `current` może się ona zmienić, mimo iż nie wynika to z kodu wewnątrz pętli.

2.9.7. Metody synchronizowane

Kolejny modyfikator stosowany wyłącznie w programowaniu współbieżnym to `synchronized`. Nie wdając się w szczegóły (zaprezentuję je dalej), powiem tylko, że w czasie wykonywania się metody synchronizowanej inne wątki odwołujące się do obiektu, w którym jest ona zadeklarowana, zostają wstrzymane do momentu zakończenia jej działania. W praktyce oznacza to, że synchronizowana metoda zostanie wykonana w całości, jako jedna porcja kodu. Ma to znaczenie na przykład w przypadku wykonywania

skomplikowanych i długotrwałych obliczeń, dla których przechowujemy kilka wyników pośrednich. Wyobraźmy sobie, że program analityczny wyznacza korelację cech *X* i *Y* według schematu pokazanego na listingu 2.109.

Listing 2.109. *Przykładowy kod wymagający synchronizacji*

```
WyznaczWariancjeX();
WyznaczWariancjeY();
WyznaczKowarianceXY();
WyznaczKorelacjaXY();
```

Każda z używanych i wywoływanych w tym fragmencie metod zapisuje wynik swojego działania, który jest wynikiem pośrednim tych obliczeń w polu dostępnym dla innych wątków. Gdyby działanie zaprezentowanego fragmentu zostało przerwane przez inny wątek, który odwoływałby się zarówno do wyników pośrednich, jak i końcowych mogłoby się okazać, że część wyników pochodzi jeszcze ze „starych" wyliczeń, a część już z „nowych". Efekt byłby tragiczny dla poprawności działania wątku wykorzystującego te wyniki. Aby temu zapobiec, powinniśmy zadeklarować metodę zawierającą przedstawiony wcześniej fragment jako synchronizowaną. Da to gwarancję, że wykona się ona cała w jednej porcji, co zapewni spójność wyników pośrednich z końcowym.

2.9.8. Pola ulotne

Java daje możliwość zapisywania stanu obiektów w sposób umożliwiający ich przyszłe odtworzenie. Umożliwia to na przykład zawieszenie pracy programu w dowolnym momencie w celu późniejszego odtworzenia go w dokładnie takim samym miejscu. Zdarzają się jednak sytuacje, kiedy działanie takie nie jest konieczne, a może nawet być niebezpieczne i szkodliwe. Może to dotyczyć przypadku, gdy przechowujemy w obiekcie pośrednie wartości ułatwiające szybkie uwiarygodnienie hasła bądź deszyfrację wiadomości. Zapisanie tych wartości w sposób łatwo dostępny mogłoby ułatwić złamanie zabezpieczeń, które inaczej byłyby dostatecznie wiarygodne. Aby można było zabezpieczyć się przed taką sytuacją, w Javie istnieje specjalny modyfikator `transient` wyłączający z zapisu to pole. W czasie zapisu obiektu na dysk pole oznaczone tym modyfikatorem wypełniane jest wartością pustą (`null`), mimo że zawiera pewną wartość. W czasie odczytu pole to jest więc niewypełnione i musimy zainicjować je ponownie specjalnie skonstruowaną w tym celu metodą.

2.9.9. Metody rodzime

Specyficzne znaczenie ma modyfikator `native`. Umożliwia on wywoływanie funkcji wewnętrznych systemu operacyjnego. Standardowo program w Javie tłumaczony jest na specjalny pseudokod, który jest w swoim charakterze bardzo podobny do kodu powszechnie stosowanych procesorów. Różnica między Javą a innymi powszechnymi rozwiązaniami jest taka, że w większości przypadków tłumaczenie kodu źródłowego odbywa się z użyciem zasad obowiązujących konkretny procesor i konkretny system operacyjny. Kod w Javie tłumaczony jest dokładnie tak samo, jednak stosowane są

zasady ustalone dla wirtualnego, a nie konkretnego docelowego procesora i systemu operacyjnego. Połączenie tych dwóch nierzeczywistych urządzeń współistnieje w wirtualnej maszynie Javy (JVM). Maszyna ta to nic innego jak aplikacja, już dla konkretnego systemu operacyjnego i procesora, która umożliwia uruchomienie programu wynikowego w Javie na dowolnym komputerze. W związku z takim rozwiązaniem program w Javie jest w trakcie działania na bieżąco tłumaczony z wewnętrznych instrukcji JVM na instrukcje procesora. Może to w niektórych przypadkach spowolnić jego działanie. Aby temu zaradzić, można zastosować metody napisane w czystym kodzie maszynowym konkretnego procesora. Często ma to zaletę szybszego wykonania i uproszczenia dostępu do niektórych specyficznych funkcji systemu operacyjnego. Wadą jest konieczność pisania takich funkcji dla wszystkich rodzajów komputerów, na których ma pracować nasz program. Pełny opis znaczenia i możliwości modyfikatora `native` znajduje się w specyfikacji JNI (*Java Native Interface*).

W apletach nie można zastosować takich funkcji z dwóch powodów. Po pierwsze nigdy nie wiadomo, do jakiego komputera zostanie załadowana strona HTML, która zawiera nasz aplet. Może się okazać, że nie przewidzieliśmy użycia konkretnego systemu, co spowodowałoby, że aplet nie pracowałby poprawnie. Druga kwestia, ważniejsza ze względów bezpieczeństwa użytkownika, to fakt, że metody w kodzie maszynowym mogą mieć nieograniczony dostęp do zasobów komputera, na którym się wykonują. Bez większych problemów mogłyby więc odczytywać bądź zapisywać plik na lokalnym dysku osoby przeglądającej strony internetowe. Takie rozwiązanie jest oczywiście nie do zaakceptowania przez większość serfujących po internecie, zostało więc zablokowane w apletach. Moim zdaniem, jeśli nasz program w Javie zaczyna wymagać funkcji natywnych, oznacza to, że zrezygnowaliśmy w nim z przenośności i uniwersalności na rzecz celów, które znacznie łatwiej zrealizować w innych środowiskach typu RAD, jakimi są na przykład Delphi, Kylix, C++Builder czy VisualAge for C.

2.10. Podsumowanie

W rozdziale tym pokazałem środki do tworzenia klas i obiektów w Javie, jednak samo pokazanie sposobu używania języka jest niewystarczające do tego, by osiągnąć zadowalające rezultaty. Dopiero trening czyni mistrza. Wychodząc z tego założenia, w następnych rozdziałach zaprezentuję i wyjaśnię kolejne ważniejsze i mniej ważne kwestie związane programowaniem obiektowym już na konkretnych przykładach i problemach, które pojawiają się w czasie tworzenia apletów i programów w Javie.