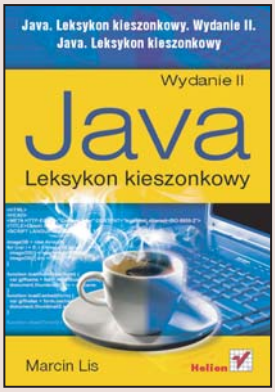
Java. Leksykon kieszonkowy. Wydanie II

Autor: Marcin Lis
ISBN: 978-83-246-1063-1
Format: B6, stron: 200



Doskonałe źródło wiedzy o języku Java i platformie Java 6

- Chcesz poznać składnię języka Java?
- Chcesz dowiedzieć się, jak wykonywać podstawowe zadania w tym języku?
- Szukasz zwięzłego, a przy tym wszechstronnego źródła informacji o Javie?

Java zasłużenie jest jednym z najbardziej popularnych języków programowania. Cechuje go między innymi wysoka przenośność, co pozwala uruchamiać kod napisany w nim w różnych systemach operacyjnych oraz na rozmaitych urządzeniach, niezawodność oraz łatwość obsługi rozwiązań sieciowych. Sprawia to, że Java ma bardzo wiele zastosowań – od prostych programów na telefony komórkowe, przez aplikacje internetowe, po rozbudowane projekty korporacyjne.

„Java. Leksykon kieszonkowy. Wydanie II" pozwoli Ci szybko rozpocząć programowanie w tym języku. Poznasz podstawy programowania obiektowego, strukturę kodu, najważniejsze typy danych, instrukcje oraz inne niezbędne elementy składni. Nauczysz się poprawiać stabilność kodu za pomocą wyjątków, a także pobierać i zapisywać dane przy użyciu strumieni wejścia-wyjścia. Dowiesz się też, jak tworzyć aplety oraz kompletne aplikacje z interfejsem graficznym.

- Składnia języka JavaScript
- Najważniejsze typy danych i instrukcje
- Tworzenie klas i obiektów oraz korzystanie z nich
- Zwiększanie niezawodności kodu przy użyciu wyjątków
- Pobieranie i zapisywanie danych
- Tworzenie apletów i umieszczanie ich na stronach
- Tworzenie aplikacji z interfejsem graficznym

Miej najważniejsze informacje o języku Java zawsze pod ręką

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Klasy i obiekty

Tworzenie klas

Klasy są opisami obiektów, czyli bytów programistycznych, które mogą przechowywać dane oraz wykonywać polecone przez programistę zadania. Każdy obiekt jest instancją, czyli wystąpieniem jakiejś klasy. Inaczej mówiąc, klasa to definicja typu danych. Schematyczny szkielet klasy wygląda następująco:

```
class nazwa_klasy {
  //treść klasy
}
```

W treści klasy są definiowane pola i metody. Pola służą do przechowywania danych, metody do wykonywania różnych operacji. Przy nadawaniu nazw klasom występują takie same ograniczenia jak w przypadku nazewnictwa zmiennych i innych identyfikatorów, czyli nazwa klasy może składać się jedynie z liter (zarówno małych, jak i dużych), cyfr oraz znaku podkreślenia, ale nie może zaczynać się od cyfry. Nie zaleca się również stosowania polskich znaków diakrytycznych, zwłaszcza że nazwa klasy publicznej musi być zgodna z nazwą pliku, w którym została zapisana.

Aby utworzyć zmienną typu obiektowego (klasowego, referencyjnego), należy skorzystać z konstrukcji:

```
nazwa_klasy nazwa_zmiennej;
```

Do tak zadeklarowanej zmiennej można następnie przypisać obiekt utworzony za pomocą operatora `new`:

```
new nazwa_klasy();
```

Jednoczesna deklaracja zmiennej, utworzenie obiektu i przypisanie go zmiennej odbywa się za pomocą schematycznej konstrukcji[5]:

```
nazwa_klasy nazwa_zmiennej = new nazwa_klasy();
```

Pola klas

Definicje pól

Pola definiowane są we wnętrzu klasy (stosowany jest też termin „w ciele klasy") w sposób identyczny jak zwykłe zmienne. Najpierw należy podać typ pola, a po nim jego nazwę. Schematycznie wygląda to następująco:

```
class nazwa_klasy {
  typ_pola1 nazwa_pola1;
  typ_pola2 nazwa_pola2;
  //...
  typ_polaN nazwa_polaN;
}
```

Przykładowa klasa o nazwie `Punkt`, zawierająca trzy pola typu `int` o nazwach `x`, `y` i `z`, będzie miała taką postać:

```
class Punkt {
  int x;
  int y;
  int z;
}
```

Odwołania do pól obiektu

Po utworzeniu obiektu do jego pól można odwoływać się za pomocą operatora kropka (`.`). Schematycznie należałoby ująć to w ten sposób:

[5] Zapis `nazwa_klasy()` to nic innego jak wywołanie bezargumentowego konstruktora danej klasy.

```
nazwa_obiektu.nazwa_pola;
```

Przykład:

```
Punkt punkt1 = new Punkt();
punkt1.x = 100;
punkt1.y = 200;
```

Wartości domyślne pól

Każde niezainicjowane pole klasy otrzymuje wartość domyślną, zależną od jego typu. Wartości te zostały zaprezentowane w tabeli 13.

Tabela 13. Wartości domyślne pól

Typ	Wartość domyślna
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
obiektowy	null

Metody klas

Definicje metod

Metody definiowane są w ciele klasy pomiędzy znakami nawiasu klamrowego. Każda metoda może przyjmować argumenty oraz zwracać wynik. Schematyczna deklaracja metody wygląda następująco:

```
typ_zwracany nazwa_metody(argumenty_metody) {
  //instrukcje metody
}
```

Po umieszczeniu we wnętrzu klasy deklaracja taka będzie miała postać:

```
class nazwa_klasy {
  typ_zwracany nazwa_metody(argumenty_metody) {
    //instrukcje metody
  }
}
```

Jeśli metoda nie zwraca żadnej wartości, jako *typ_zwracany* należy zastosować słowo `void`, jeśli natomiast nie przyjmuje żadnych argumentów, w nawiasie okrągłym nie należy nic wpisywać. Przykładowa klasa `Punkt` zawierająca dwa pola `x` i `y` typu `int` i jedną metodę o nazwie `show`, nie zwracająca żadnego wyniku, a wyświetlająca wartości `x` i `y`, będzie miała postać:

```
class Punkt {
  int x, y;
  void show(){
    System.out.println("x = " + x + ", y = " + y);
  }
}
```

Odwołania do metod

Po utworzeniu obiektu do jego metod można odwoływać się analogicznie jak do pól, czyli za pomocą operatora kropka (`.`). Oto schematyczny zapis:

```
nazwa_obiektu.nazwa_metody();
```

Zakładając, że istnieje klasa `Punkt` zawierająca bezargumentową metodę o nazwie `wyswietlWspolrzedne` oraz zmienna referencyjna `punkt1` wskazująca na obiekt tej klasy, wywołanie metody będzie miało postać:

```
punkt1.wyswietlWspolrzedne();
```

Argumenty metod

Argumenty metody to inaczej dane, które można jej przekazać. Metoda może mieć dowolną liczbę argumentów umieszczonych w nawiasie okrągłym za jej nazwą. Poszczególne argumenty oddzielane są od siebie znakiem przecinka. Schematycznie wygląda to następująco:

```
typ_wyniku nazwa_metody(typ_argumentu_1 nazwa_argumentu_1,
typ_ argumentu_2 nazwa_ argumentu_2, ... , typ_argumentu_N
nazwa_argumentu_N)
```

Przykład:

```
void ustawXY(int wspX, int wspY) {
  x = wspX;
  y = wspY;
}
```

Argumentami mogą być zarówno dane typów prostych, jak i obiektowych.

Zwracanie wyniku

Typ wyniku należy podać przed nazwą metody, tak jak zostało to opisane wcześniej. Sam wynik jest natomiast zwracamy przez zastosowanie instrukcji `return`. Schematycznie należałoby to ująć tak:

```
typ_zwracany nazwa_metody(argumenty_metody) {
  //instrukcje metody
  return wartość;
}
```

Przykładowo metoda przyjmująca dwa argumenty typu `int` i zwracająca w wyniku wartość typu `int` będącą wynikiem ich dodawania miałaby postać:

```
int dodaj(int arg1, int arg2){
  return arg1 + arg2;
}
```

Przeciążanie metod

W każdej klasie mogą istnieć dwie lub więcej metod, które mają takie same nazwy, o ile tylko różnią się argumentami. Mogą — ale nie muszą — również różnić się typem zwracanego wyniku. Technika ta nazywa się przeciążaniem metod. Przykładowa klasa zawierająca dwie przeciążone metody wygląda następująco:

```
public class Punkt {
  int x;
  int y;
  void ustawXY(int wspX, int wspY) {
    x = wspX;
    y = wspY;
  }
  void ustawXY(Punkt punkt) {
    x = punkt.x;
    y = punkt.y;
  }
}
```

Klasa ta zawiera dwie przeciążone metody o nazwie `ustawXY`. Jest to możliwe, ponieważ przyjmują one różne argumenty: pierwsza metoda dwie wartości typu `int`, druga — jeden obiekt klasy `Punkt`. Obie metody realizują takie samo zadanie, tzn. ustawiają nowe wartości w polach `x` i `y`.

Metoda specjalna main

Każdy program musi zawierać punkt startowy, czyli miejsce, od którego zacznie się jego wykonywanie. W Javie takim miejscem jest metoda o nazwie `main` i następującej ogólnej postaci:

```
public static void main(String[] args) {
  //kod metody main
}
```

Jeśli w danej klasie znajdzie się metoda o takiej deklaracji, od niej właśnie zacznie się wykonywanie kodu programu. Metoda `main` musi być publiczna i statyczna oraz nie może zwracać wyniku (typ

zwracany `void`). Parametr `args` to tablica zawierająca obiekty typu `String` odzwierciedlające listę argumentów, z jakimi program został wywołany. Przykładowy program wyświetlający listę argumentów przekazanych mu w wierszu poleceń będzie miał zatem postać:

```
class Main {
  public static void main (String args[]) {
    System.out.println("Parametry wywołania:");
    for(int i = 0; i < args.length; i++){
        System.out.println(args[i]);
    }
  }
}
```

Metoda `main` może zostać umieszczona praktycznie w dowolnej klasie wchodzącej w skład danej aplikacji. Co więcej, nawet w ramach jednej aplikacji każda z klas wchodzących w jej skład może mieć swoją własną metodę `main`. O tym, która z tych metod zostanie wykonana, decyduje to, która klasa zostanie klasą uruchomieniową. Klasa uruchomieniowa to ta klasa, której nazwa jest podawana jako argument dla maszyny wirtualnej. Zatem uruchomienie aplikacji przez wydanie komendy:

```
java Main
```

powoduje, że klasą uruchomieniową jest klasa `Main`, natomiast komendy:

```
java Punkt
```

powoduje, że klasą uruchomieniową jest klasa `Punkt`. W praktyce zazwyczaj stosuje się jednak tylko jedną metodę `main` dla całej aplikacji.

Konstruktory klas

Tworzenie konstruktorów

Konstruktor to specjalna metoda, która jest wywoływana podczas tworzenia obiektu. Metoda będąca konstruktorem nigdy nie zwraca żadnego wyniku i musi mieć nazwę zgodną z nawą klasy. Schematycznie budowa konstruktora wygląda następująco:

```
class nazwa_klasy {
  nazwa_klasy() {
    //kod konstruktora
  }
}
```

Należy zwrócić uwagę na to, że przed konstruktorem nie występuje słowo `void`.

Przykładowa klasa o nazwie `Punkt` zawierająca prosty konstruktor została przedstawiona poniżej:

```
class Punkt {
  int x;
  int y;
  Punkt() {
    x = 1;
    y = 1;
  }
}
```

Jak widać, wszystko jest tu zgodne z podanym wyżej schematem. Konstruktor nie zwraca żadnej wartości i ma nazwę zgodną z nazwą klasy. Przed nazwą nie występuje słowo `void`. W jego wnętrzu następuje natomiast proste przypisanie wartości polom obiektu.

Argumenty konstruktorów

Konstruktor nie musi być bezargumentowy, może on również przyjmować argumenty, które zostaną wykorzystane, bezpośrednio lub pośrednio, np. do zainicjowania pól obiektu. Argumenty przekazuje się dokładnie w taki sam sposób, jak w przypadku zwykłych metod, czyli schematyczna konstrukcja takiego konstruktora ma następującą postać:

```
class nazwa_klasy {
  nazwa_klasy(typ1 argument1, typ2 argument2,...,
  typN argumentN) {
  }
}
```

Oczywiście jeśli w klasie występuje tylko jeden konstruktor i przyjmuje on argumenty, przy tworzeniu obiektu należy je podać. W takim wypadku trzeba zatem zastosować konstrukcję:

```
nazwa_klasy zmienna = new
nazwa_klasy(argumenty_konstruktora);
```

Przykład klasy zawierającej konstruktor przyjmujący dwa argumenty typu `int`:

```
public class Punkt {
  int x;
  int y;
  Punkt(int wspX, int wspY) {
    x = wspX;
    y = wspY;
  }
}
```

Po takiej definicji podczas każdej próby utworzenia obiektu klasy `Punkt` niezbędne będzie podawanie jego współrzędnych. Przykładowo: jeśli początkowa współrzędna `x` ma mieć wartość `100`, a `y` — `200`, należy zastosować konstrukcję:

```
Punkt punkt = new Punkt(100, 200);
```

Przeciążanie konstruktorów

Konstruktory, tak jak zwykłe metody, mogą być przeciążane, tzn. każda klasa może mieć kilka konstruktorów, o ile tylko różnią się one przyjmowanymi argumentami. Przykładowa klasa `Punkt` zawierająca trzy konstruktory:

- pierwszy — bezargumentowy,
- drugi — przyjmujący dwa argumenty typu `int`,
- trzeci — przyjmujący jeden argument typu `Punkt`,

będzie miała postać:

```
class Punkt {
  int x;
  int y;
  Punkt() {
    x = 1;
    y = 1;
  }
  Punkt(int wspX, int wspY) {
    x = wspX;
    y = wspY;
  }
  Punkt(Punkt punkt) {
    x = punkt.x;
    y = punkt.y;
  }
}
```

Taka budowa klasy `Punkt` pozwala na niezależne wywoływanie każdego z trzech konstruktorów, w zależności od tego, który z nich jest najbardziej odpowiedni w danej sytuacji, np.:

```
Punkt punkt1 = new Punkt();
Punkt punkt2 = new Punkt(100, 100);
Punkt punkt3 = new Punkt(punkt1);
```

Konstruktor domyślny

Jeżeli w klasie nie zostanie zdefiniowany żaden konstruktor, zostanie do niej automatycznie dodany bezargumentowy konstruktor domyślny. Klasa taka będzie się zatem zachowywała tak, jakby miała schematyczną postać:

```
class nazwa_klasy {
  nazwa_klasy() {
  }
  //pola i metody klasy
}
```

Jeżeli w klasie zostanie jawnie zdefiniowany konstruktor bezargumentowy, automatycznie stanie się on konstruktorem domyślnym (jest to ważne przy dziedziczeniu), niezależnie od tego, czy istnieją inne konstruktory.

Wywoływanie metod w konstruktorach

We wnętrzu konstruktora (tak jak w każdej innej metodzie) można wywoływać inne metody. Przykładowo zamiast bezpośrednio przypisywać w konstruktorze wartości polom klasy, można do tego celu użyć innej metody. Taka klasa mogłaby mieć np. następującą postać:

```
class Punkt {
  int x;
  int y;
  Punkt() {
    ustawXY(0, 0);
  }
  Punkt(int x, int y) {
    ustawXY(x, y);
  }
  void ustawXY(int wspX, int wspY) {
    x = wspX;
    y = wspY;
  }
}
```

Słowo kluczowe this

Użycie odwołania this

Słowo kluczowe `this` to odwołanie do obiektu bieżącego. Można je traktować jako referencję do aktualnego obiektu. Odwołanie do pól i metod poprzez obiekt `this` odbywa się za pomocą operatora kropka (`.`):

```
this.nazwa_pola = wartość;
this.nazwa_metody(argumenty);
```

Umożliwia to m.in. stosowanie w metodach i konstruktorach argumentów o nazwach identycznych z nazwami pól klasy, przykładowo:

```
Punkt(int x, int y) {
  this.x = x;
  this.y = y;
}
```

Instrukcję `this.x = x` należy w takim przypadku rozumieć następująco: przypisz polu `x` wartość przekazaną jako argument o nazwie `x`, a instrukcję `this.y = y` jako: przypisz polu `y` wartość przekazaną jako argument o nazwie `y`.

Wywoływanie konstruktorów

Z wnętrza konstruktora można wywołać inny konstruktor. Taka możliwość może być przydatna w sytuacji, kiedy w klasie istnieje kilka przeciążonych konstruktorów, a zakres wykonywanego przez nie kodu pokrywa się. Nie zawsze takie wywołanie jest możliwe i niezbędne, niemniej taka możliwość istnieje. Należy w tym celu wykorzystać słowo kluczowe `this`. Jeżeli za tym słowem zostanie podana lista argumentów umieszczonych w nawiasie okrągłym, czyli konstrukcja będzie miała ogólną postać:

```
this(argument1, argument2, ... , argumentN)
```

to zostanie wywołany konstruktor, którego argumenty pasują do wymienionych. W przypadku klasy `Punkt` mogłoby to wyglądać następująco:

```
class Punkt {
  int x;
  int y;
  Punkt() {
    this(0, 0);
  }
  Punkt(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Należy przy tym pamiętać, że konstruktor jawnie można wywołać tylko w innym konstruktorze i musi on być pierwszą wykonywaną instrukcją. Oznacza to, że wolno wywołać tylko jeden konstruktor, a przed nim nie może znaleźć się żadna inna instrukcja.

Niszczenie obiektów

W Javie nie ma, znanych z języków takich jak C i C++, typowych destruktorów ani instrukcji pozwalających na usuwanie obiektów z pamięci. Tymi zadaniami zajmuje się maszyna wirtualna i proces tak zwanego odśmiecacza (ang. *garbage collector*). Jest to wyjątkowo wygodne podejście dla programisty, zwalania go bowiem z obowiązku zarządzania pamięcią, choć zwiększa to nieco narzuty czasowe związane z wykonaniem programu, jako że sam proces odśmiecania musi zająć czas procesora. Niemniej dzisiejsze maszyny wirtualne są na tyle dopracowane, że w większości przypadków nie ma najmniejszej potrzeby zaprzątania myśli tym problemem.

W nielicznych jednak przypadkach, np. w sytuacji, gdyby stworzony obiekt wykorzystywał mechanizmy alokacji pamięci specyficzne dla danej platformy systemowej czy też odwoływał się

do modułów napisanych w innych językach programowania, o posprzątanie systemu i zwolnienie zarezerwowanej pamięci musi zadbać programista. Java udostępnia w tym celu metodę `finalize`, która jest wykonywana zawsze, kiedy obiekt jest niszczony, usuwany z pamięci. Wystarczy więc, jeśli klasa będzie zawierała taką metodę, a przy niszczeniu obiektu naszej klasy zostanie ona wykonana. We wnętrzu tej metody można wykonać dowolne instrukcje sprzątające. Deklaracja metody `finalize` wygląda następująco:

```
public void finalize() {
  //tu treść metody
}
```

Trzeba jednak wiedzieć, że nie ma żadnej gwarancji, że ta metoda zostanie wykonana w trakcie działania programu. Dzieje się tak dlatego, że proces odzyskiwania pamięci, a więc niszczenia nieużywanych już obiektów, zaczyna się wtedy, kiedy *garbage collector* uzna to za stosowne. Czyli wtedy, kiedy uzna, że ilość wolnej pamięci dostępnej dla programu zbytnio się zmniejszyła. Może się więc okazać, że pamięć zostanie zwolniona dopiero po zakończeniu pracy aplikacji. Dlatego jeśli niezbędne jest wykonanie dodatkowych czynności porządkowych w jakimś określonym miejscu podczas działania programu, może być konieczne napisanie dodatkowej metody sprzątającej i jawne jej wywołanie.

Dziedziczenie

Konstrukcje podstawowe

W Javie dziedziczenie jest wyrażane za pomocą słowa `extends`, a cała definicja schematycznie wygląda następująco:

```
class klasa_potomna extends klasa_bazowa {
  //wnętrze klasy
}
```

Zapis taki oznacza, że klasa potomna dziedziczy z klasy bazowej. Na przykład, jeśli klasą bazową będzie `Punkt`, a potomną — `Punkt3D`, cała konstrukcja przyjmie postać:

```
class Punkt {
  //składowe klasy Punkt
}
class Punkt3D extends Punkt {
  //składowe klasy Punkt3D
}
```

W takiej sytuacji klasa `Punkt3D` oprócz własnych składowych będzie posiadała również składowe przejęte z klasy `Punkt`.

Konstruktory klas potomnych

Podczas tworzenia obiektu klasy potomnej zawsze wywoływany jest domyślny, bezargumentowy konstruktor klasy bazowej. Jeżeli w klasie bazowej nie istnieje konstruktor domyślny, wymagane jest jawne wywołanie jednego z pozostałych konstruktorów. Wywołanie to wymaga zastosowania konstrukcji ze słowem kluczowym `super`. Słowo `super` oznacza w tym przypadku wywołanie konstruktora klasy bazowej. Schematycznie można by to ująć tak:

```
class klasa_potomna extends klasa_bazowa {
  klasa_potomna() {
    super(argumenty);
    /*
    ...dalszy kod konstruktora...
    */
  }
}
```

Ważne jest, aby wywołanie metody `super` było pierwszą instrukcją konstruktora klasy potomnej. Przykładowe jawne wywołanie konstruktora klasy bazowej widoczne jest na poniższym przykładzie:

```
class Punkt {
  int x;
  int y;
  Punkt(int x, int y) {
```

```
    this.x = x;
    this.y = y;
  }
}

class Punkt3D extends Punkt {
  int z;
  Punkt3D(int x, int y, int z) {
    super(x, y);
    this.z = z;
  }
}
```

Przesłanianie pól i metod

Przesłanianie metod

Kiedy w klasie potomnej znajduje się metoda o takiej samej nazwie i argumentach jak metoda znajdująca się w klasie bazowej występuje zjawisko przesłaniania metod. Wtedy bezpośrednio dostępna jest jedynie metoda z klasy potomnej. W przypadku przykładowych dwóch klas:

```
class A {
  public void f() {
    System.out.println("A");
  }
}
class B extends A {
  public void f() {
    System.out.println("B");
  }
}
```

po utworzeniu obiektu klasy `B` i przypisaniu go do zmiennej `b` (`B b = new B()`) oraz po wywołaniu:

```
b.f()
```

zostanie wykonana metoda `f` klasy `B`. W klasie potomnej można jednak wywołać metodę przesłoniętą, czyli w tym przypadku metodę `f` z klasy `A`. Odwołanie do przesłoniętej metody klasy

bazowej uzyskuje się poprzez wykorzystanie składni ze słowem kluczowym `super`, w schematycznej postaci:

```
super.nazwa_metody(argumenty);
```

Takie wywołanie najczęściej stosowane jest w metodzie przesłaniającej, czyli (zakładając przedstawioną wyżej postać klasy `A`):

```
class B extends A {
  public void f() {
    super.f();
    System.out.println("B");
  }
}
```

choć oczywiście może się również znaleźć w dowolnej innej metodzie klasy potomnej.

Przesłanianie pól

Pola klas bazowych są przesłaniane podobnie jak w przypadku metod. Jeśli więc w klasie pochodnej zostanie zdefiniowane pole o takiej samej nazwie jak w klasie bazowej, bezpośrednio dostępne będzie tylko pole klasy pochodnej. Przykładowe dwie klasy zostały przedstawione poniżej:

```
class A {
  int liczba;
}
class B extends A {
  int liczba;
}
```

Warto zauważyć, że pole przesłaniające nie musi być tego samego typu co pole przesłaniane, choć tworzenie takich konstrukcji bardzo zaciemnia kod aplikacji i stanowczo nie jest zalecane.

W klasie potomnej można odwołać się do przesłoniętego pola klasy bazowej za pomocą składni ze słowem kluczowym `super`, podobnie jak w przypadku przesłanianych metod. Zakładając zatem klasę `A` w powyższej postaci, w klasie `B` można odwołać się do przesłoniętego pola następująco:

```
class B extends A {
  int liczba;
  public void f() {
    liczba = super.liczba;
  }
}
```

Modyfikatory dostępu

Przed każdym polem i metodą może wystąpić modyfikator dostępu określający prawa dostępu do składowych klasy. Wyróżnia się cztery typy dostępu:

- publiczny,
- prywatny,
- chroniony,
- pakietowy.

Domyślnie, jeżeli przed składową klasy nie występuje żadne określenie, dostęp jest pakietowy, co oznacza, że dostęp do tej składowej mają wszystkie klasy pakietu, w którym się ona znajduje (patrz podrozdział „Pakiety"). Dostęp publiczny określany jest słowem `public`, prywatny — słowem `private`, a chroniony — słowem `protected`.

Dostęp publiczny

Jeżeli dana składowa klasy jest publiczna, oznacza to, że mają do niej dostęp wszystkie inne klasy, czyli nie istnieją żadne ograniczenia w dostępie to tej składowej. W przypadku pól klasy specyfikator dostępu `public` należy zatem umieścić przed nazwą typu, co schematycznie wygląda następująco:

```
public nazwa_typu nazwa_pola;
```

Podobnie jest z metodami, specyfikator dostępu powinien być pierwszym elementem deklaracji:

```
public typ_zwracany nazwa_metody(argumenty){
  //treść metody
}
```

Przykładowa klasa o nazwie `Punkt` zawierająca dwa publiczne pola typu `int` o nazwach `x` i `y` oraz dwie publiczne metody o nazwach `pobierzX` i `pobierzY`, obie zwracające wartości typu `int`, będzie miała postać:

```
class Punkt {
  public int x;
  public int y;
  public int pobierzX() {
    return x;
  }
  public int pobierzY() {
    return y;
  }
}
```

Dostęp prywatny

Składowe oznaczone słowem `private` to takie, które dostępne są jedynie z wnętrza danej klasy. To znaczy wszystkie metody danej klasy mogą je dowolnie odczytywać i zapisywać, natomiast dostęp z zewnątrz jest zabroniony. W przypadku pól klasy specyfikator dostępu `private` należy umieścić przed nazwą typu:

```
private nazwa_typu nazwa_pola;
```

Podobnie jest z metodami, specyfikator dostępu powinien być pierwszym elementem deklaracji:

```
private typ_zwracany nazwa_metody(argumenty){
  //treść metody
}
```

Przykładowa klasa o nazwie `Punkt` zawierająca dwa prywatne pola typu `int` o nazwach `x` i `y` oraz dwie prywatne metody o nazwach `pobierzX` i `pobierzY`, obie zwracające wartości typu `int`, będzie miała postać:

```
class Punkt {
  private int x;
  private int y;
  private int pobierzX() {
    return x;
  }
  private int pobierzY() {
    return y;
  }
}
```

Dostęp chroniony

Składowe klasy oznaczone słowem `protected` to składowe chronione. Są one dostępne jedynie dla metod danej klasy, klas potomnych oraz klas z tego samego pakietu. W przypadku pól klasy specyfikator dostępu `protected` należy umieścić przed nazwą typu, co schematycznie wygląda następująco:

```
protected nazwa_typu nazwa_pola;
```

Podobnie jest z metodami, specyfikator dostępu powinien być pierwszym elementem deklaracji:

```
protected typ_zwracany nazwa_metody(argumenty){
  //treść metody
}
```

Przykładowa klasa o nazwie `Punkt` zawierająca dwa chronione pola typu `int` o nazwach `x` i `y` oraz dwie chronione metody o nazwach `pobierzX` i `pobierzY`, obie zwracające wartości typu `int`, będzie miała postać:

```
class Punkt {
  protected int x;
  protected int y;
```

```
  protected int pobierzX() {
    return x;
  }
  protected int pobierzY() {
    return y;
  }
}
```

Dostęp pakietowy

Dostęp pakietowy jest dostępem domyślnym i ma miejsce, kiedy przed składową klasy nie występuje żaden modyfikator dostępu. Konstrukcja taka oznacza, że dostęp do składowej mają wszystkie klasy pakietu, w którym się ona znajduje.

Przykładowa klasa o nazwie `Punkt` zawierająca dwa pakietowe pola typu `int` o nazwach `x` i `y` oraz dwie pakietowe metody o nazwach `pobierzX` i `pobierzY`, obie zwracające wartości typu `int`, będzie miała postać:

```
class Punkt {
  int x;
  int y;
  int pobierzX() {
    return x;
  }
  int pobierzY() {
    return y;
  }
}
```

Pakiety

Tworzenie pakietów

Klasy w Javie grupowane są w jednostki nazywane pakietami. Pakiet to inaczej biblioteka, zestaw powiązanych ze sobą tematycznie klas. Do tworzenia pakietów służy słowo kluczowe `package`,

po którym następuje nazwa pakietu zakończona znakiem średnika. Schematyczny zapis tego przedstawia się w ten sposób:

```
package nazwa_pakietu;
```

Instrukcja ta musi znajdować się na początku pliku, przed nią nie mogą znajdować się żadne inne instrukcje. Przed `package` mogą występować jedynie komentarze:

```
//pakiet i klasa pakietowa
package nazwa_pakietu;
class nazwa_klasy {
  /*
    treść klasy
  */
}
```

W celu skorzystania z klasy zawartej w pakiecie w innej klasie należy użyć dyrektywy `import` w postaci:

```
import nazwa_pakietu.nazwa_klasy;
```

Dyrektywa `import` musi znajdować się na początku pliku.

Aby zaimportować wszystkie klasy z danego pakietu, dyrektywa `import` powinna mieć postać:

```
import nazwa_pakietu.*;
```

Nazwy pakietów

Nazwy pakietów powinny być pisane w całości małymi literami, a jeśli pakiet ma być udostępniony publicznie, należy poprzedzić go odwróconą nazwą domenową twórcy pakietu. Nie jest to obligatoryjne, ale stwarza duże prawdopodobieństwo utworzenia nazwy unikalnej w skali globu. Na przykład jeżeli domeną autora jest *marcinlis.com* i ma powstać pakiet o nazwie *grafika*, jego pełna nazwa będzie brzmieć: *com.marcinlis.grafika*. Wszystkie z kolei klasy tego pakietu będą musiały być umieszczone w strukturze katalogów odpowiadających tej nazwie.

Rodzaje klas

Klasy w Javie możemy podzielić na trzy typy:

- publiczne,
- pakietowe,
- wewnętrzne.

Klasy wewnętrzne zostaną omówione w dalszej części leksykonu. Ten podrozdział zawiera informacje o klasach pakietowych i publicznych.

Jeśli przed nazwą klasy nie znajduje się modyfikator `public`, jest to klasa pakietowa, czyli klasa dostępna jedynie dla innych klas z tego samego pakietu. Jeśli natomiast przed nazwą klasy znajduje się modyfikator `public`, jest to klasa publiczna, czyli klasa dostępna dla wszystkich innych klas.

Zatem deklaracja klasy pakietowej ma postać:

```
class nazwa_klasy {
  //pola i metody klasy
}
```

natomiast klasy publicznej:

```
public class nazwa_klasy {
  //pola i metody klasy
}
```

Należy pamiętać, że w jednym pliku *java* (czyli w jednej jednostce kompilacji) może znaleźć się tylko jedna klasa o dostępie publicznym. Można jednak w takim pliku umieścić dowolną liczbę klas pakietowych.

Statyczne składowe klas

Składowe statyczne to pola i metody klasy, które mogą istnieć, nawet jeśli nie istnieje obiekt tej klasy. Każda taka metoda lub pole jest wspólna dla wszystkich obiektów tej klasy. Składowe te oznaczane są słowem `static`.

Statyczne metody

Metody statyczne oznacza się słowem `static`, które zwyczajowo powinno znaleźć się zaraz za modyfikatorem dostępu, a przed typem zwracanego wyniku, co schematycznie ma postać:

```
modyfikator_dostępu static typ_zwracany
nazwa_metody(argumenty) {
  //treść metody
}
```

Przykład:

```
public class A {
  public static void f() {
    //treść metody f
  }
}
```

Tak napisaną metodę można wywołać klasycznie, to znaczy po utworzeniu obiektu klasy `A`, np. w postaci:

```
A a = new A();
a.f();
```

czyli stosując standardowy schemat:

```
nazwa_obiektu.nazwa_metody(argumenty_metody);
```

Ponieważ jednak metody statyczne istnieją nawet wtedy, kiedy nie ma żadnego obiektu danej klasy, możliwe jest również wywołanie w postaci:

```
nazwa_klasy.nazwa_metody(argumenty_metody);
```

W przypadku wymienionej wyżej klasy `A` wywołanie tego typu miałoby postać:

```
A.f();
```

Warto zauważyć, że metoda `main`, od której rozpoczyna się wykonywanie kodu aplikacji, właśnie dlatego musi być metodą statyczną, może bowiem zostać wykonana, mimo że w trakcie uruchamiania aplikacji nie ma jeszcze żadnych obiektów.

Należy również zwrócić uwagę na to, że metoda statyczna jest umieszczana w specjalnie zarezerwowanym do tego celu obszarze pamięci i jeśli powstaną obiekty danej klasy, to będzie ona dla nich wspólna. To znaczy, że nie są tworzone kopie metody statycznej dla każdego obiektu klasy.

Statyczne pola

Do pól oznaczonych jako statyczne można odwoływać się, podobnie jak w przypadku statycznych metod, nawet jeśli nie istnieje żaden obiekt danej klasy. Pola takie deklaruje się, umieszczając przed typem słowo `static`. Schematycznie deklaracja taka wygląda zatem następująco:

```
static typ_pola nazwa_pola;
```

lub

```
specyfikator_dostępu static typ_pola nazwa_pola;
```

Przykład:

```
public class A {
  public static int liczba;
}
```

Do pól statycznych można się odwoływać w sposób klasyczny, tak jak i do innych pól klasy, czyli poprzedzając je nazwą obiektu (oczywiście o ile wcześniej dany obiekt został utworzony):

```
nazwa_obiektu.nazwa_pola;
```

bądź też poprzedzając je nazwą klasy:

```
nazwa_klasy.nazwa_pola;
```

Podobnie jak metody statyczne, również i pola tego typu znajdują się w wyznaczonym obszarze pamięci i są wspólne dla wszystkich obiektów danej klasy. Niezależnie więc od liczby obiektów danej klasy pole statyczne o danej nazwie będzie tylko jedno.

Klasy i składowe finalne

Finalne klasy

Klasa finalna to taka, z której nie wolno wyprowadzać innych klas. Pozwala to na tworzenie klas, które nie będą miały klas pochodnych — ich postać będzie po prostu z góry ustalona. Jeśli klasa ma stać się finalną, należy przed jej nazwą umieścić słowo kluczowe `final` zgodnie ze schematem:

```
specyfikator_dostępu final class nazwa_klasy {
  //pola i metody klasy
}
```

Przykład:

```
public final class Example {
  //treść klasy
}
```

Podobnie jak w przypadku słowa kluczowego `static`, nie ma znaczenia, czy zastosowany zostanie zapis `public final class` czy `final public class`, niemniej dla przejrzystości i ujednolicenia notacji najczęściej stosuje się pierwszy z przedstawionych sposobów.

Finalne pola

Pole klasy oznaczone słowem `final` staje się polem finalnym, czyli takim, którego wartość jest stała i nie można jej zmieniać. Słowo kluczowe `final` umieszcza się zwyczajowo przed nazwą typu danego pola, stosując zapis:

```
final typ_pola nazwa_pola;
```

lub ogólniej:

```
specyfikator_dostępu [static] final typ_pola nazwa_pola;
```

Poprawne są zatem wszystkie poniższe deklaracje:

```
final int liczba;
public final double liczba;
public static final char znak;
```

W rzeczywistości specyfikator dostępu oraz słowa `final` i `static` mogą występować w dowolnej kolejności, jednak dla zachowania spójnego stylu i przejrzystości kodu należy konsekwentnie stosować jeden schemat nazewnictwa. Najczęściej przyjmuje się zasadę, że na pierwszym miejscu jest umieszczany specyfikator dostępu, słowo `final` występuje zawsze tuż przed typem pola, natomiast słowo `static` zawsze tuż przed słowem `final`.

Inicjalizacja pól finalnych

Pola finalne mogą być inicjalizowane już w momencie ich deklaracji, w przypadku pól typów prostych:

```
final nazwa_typu nazwa_pola;
```

a w przypadku typów referencyjnych:

```
final nazwa_klasy nazwa_pola = new nazwa_klasy();
```

Pola takie mogą być jednak również deklarowane bez inicjalizacji. Wartością takiego pola staje się wtedy ta, użyta w pierwszym przypisaniu, i dopiero od chwili tego przypisania wartości tej nie

wolno zmieniać. Tak zdeklarowane pole musi też zostać zainicjalizowane w konstruktorze danej klasy, np.:

```
public final class A {
  final int i = 0;
  final int j;
  public A(){
    j = 10;
  }
}
```

Finalne metody

Metoda oznaczona słowem `final` staje się finalna, co oznacza, że jej przesłonięcie w klasie potomnej nie będzie możliwe. Słowo `final` umieszcza się przed typem wartości zwracanej przez metodę, czyli schematycznie konstrukcja taka wygląda następująco:

```
final typ_zwracany nazwa_metody(argumenty)
```

lub ogólniej:

```
specyfikator_dostępu [static] final typ_zwracany
nazwa_metody(argumenty).
```

Prawidłowe będą więc wszystkie wymienione przykładowe deklaracje:

```
final void metoda(){/*kod metody*/};
public final int metoda(){/*kod metody*/};
public static final void metoda(){/*kod metody*/};
public static final int metoda(int argument){/*kod
metody*/};
```

Ponieważ w jednej klasie nie mogą znaleźć się dwie metody o tej samej nazwie i takich samych argumentach, metody finalne będą się różniły od zwykłych metod tylko w przypadku dziedziczenia. Dokładniej, zgodnie z tym, co zostało napisane powyżej, metod finalnych nie można przesłaniać w klasach potomnych.

Finalne argumenty

Argument finalny to taki, którego nie wolno zmieniać w ciele metody. Aby uczynić argument finalnym, należy umieścić słowo `final` przed jego typem. Schematycznie konstrukcja taka wyglądać będzie w taki sposób:

```
specyfikator_dostępu [static][final] typ_zwracany
nazwa_metody(final typ_argumentu nazwa_argumentu).
```

Przykładowo deklaracja publicznej metody o nazwie `metoda`, nie zwracającej żadnej wartości, przyjmującej natomiast jeden finalny argument typu `int` o nazwie `argument`, będzie miała postać:

```
public void metoda(final int argument){
  /*treść metody*/
}
```

Interfejsy i klasy abstrakcyjne

Klasy abstrakcyjne

Klasa abstrakcyjna to taka, która została zadeklarowana z użyciem słowa kluczowego `abstract`. Przy czym klasa, w której przynajmniej jedna metoda jest abstrakcyjna (oznaczona słowem kluczowym `abstract`), musi być zadeklarowana jako abstrakcyjna[6]. Oto ogólny zapis takiej konstrukcji:

```
[public] abstract class nazwa_klasy {
  [specyfikator_dostępu] abstract typ_zwracany
   nazwa_metody(argumenty);
}
```

[6] Nie wyklucza to oczywiście istnienia klas abstrakcyjnych, w których żadna z metod nie jest abstrakcyjna.

Metoda abstrakcyjna posiada jedynie definicję, nie może zawierać żadnego kodu. Przykładowa publiczna i abstrakcyjna klasa `Shape` zawierająca abstrakcyjną metodę `draw` będzie miała postać:

```
public abstract class Shape {
  public abstract void draw();
}
```

Po takiej deklaracji nie można będzie tworzyć obiektów klasy `Shape`. Próba wykonania przykładowej instrukcji:

```
Shape shape = new Shape();
```

skończy się komunikatem o błędzie: `Shape is abstract; cannot be instantiated`.

Zadeklarowanie metody jako abstrakcyjnej wymusza jej redeklarację w klasie potomnej. W przedstawionym przykładzie oznacza to, że każda klasa wyprowadzona, czyli dziedzicząca, z `Shape` musi zawierać metodę `draw`. Jeżeli w klasie pochodnej tej metody zabraknie, programu nie uda się skompilować.

Interfejsy

Budowa interfejsu

Interfejs to klasa czysto abstrakcyjna, czyli taka, w której wszystkie metody są traktowane jako abstrakcyjne. Deklaruje się go za pomocą słowa kluczowego `interface`. Interfejs może być publiczny, o ile jest zdefiniowany w pliku o takiej samej nazwie jak nazwa interfejsu, lub pakietowy. W tym drugim przypadku jest dostępny jedynie dla klas wchodzących w skład danego pakietu. Schematyczna konstrukcja interfejsu wygląda następująco:

```
[public] interface nazwa_interfejsu {
  typ_zwracany nazwa_metody1(argumenty);
  typ_zwracany nazwa_metody2(argumenty);
  /*...dalsze metody interfejsu...*/
  typ_zwracany nazwa_metodyn(argumenty);
}
```

Przykładowy interfejs o nazwie `Drawable` zawierający deklarację jednej tylko metody o nazwie `draw` będzie miał postać:

```
public interface Drawable {
  public void draw();
}
```

Tak zdefiniowany interfejs może być implementowany przez dowolną klasę. Jeśli mowa o tym, że dana klasa implementuje interfejs, oznacza to, że zawiera definicje wszystkich zadeklarowanych w nim metod. Jeśli choć jedna metoda zostanie pominięta, kompilator zgłosi błąd. To, że klasa ma implementować dany interfejs, zaznacza się, wykorzystując słowo kluczowe `implements`, co schematycznie przedstawia się tak:

```
[specyfikator dostępu][abstract] class nazwa_klasy
implements nazwa_interfejsu {
  /*
  ... pola i metody klasy ...
  */
}
```

Jeśli więc przykładowa klasa `Example` ma implementować przedstawiony wyżej interfejs `Drawable`, powinna mieć postać:

```
public class Example implements Drawable {
  public void draw() {
    /* wnętrze metody draw */
  }
}
```

Pola interfejsów

Interfejsy, oprócz deklaracji metod, mogą również zawierać pola. Pola interfejsu są zawsze publiczne, statyczne oraz finalne, czyli trzeba im przypisać wartości już w momencie ich deklaracji. Można stosować zarówno typy proste, jak i złożone. Pola interfejsów są najczęściej wykorzystywane do tworzenia wyliczeń. Deklaracja pola interfejsu nie różni się od deklaracji pola klasy, schematycznie zatem można to ująć:

```
[public] interface nazwa_interfejsu {
  typ_pola1 nazwa_pola1 = wartość_pola1;
  typ_pola2 nazwa_pola2 = wartość_pola2;
  /* ... */
  typ_polaN nazwa_polaN = wartość_polaN;
}
```

Przykładowy interfejs zawierający dwa pola typu prostego i jedno typu obiektowego jest przedstawiony poniżej. Warto zwrócić uwagę na konwencję zapisu: ponieważ pola interfejsów są zawsze statyczne i finalne, przyjmuje się, że ich nazwy są pisane wielkimi literami, a poszczególne człony nazwy są oddzielane znakiem podkreślenia.

```
public interface NowyInterfejs {
  int POLE_TYPU_INT = 100;
  double POLE_TYPU_DOUBLE = 1.0;
  Object POLE_TYPU_OBJECT = new Object();
}
```

Klasy wewnętrzne

Budowa klas wewnętrznych

Klasa wewnętrzna to taka, która została zdefiniowana we wnętrzu innej klasy. W praktyce pozwala to na wygodne tworzenie różnych konstrukcji programistycznych. Schematyczna deklaracja klasy wewnętrznej wygląda następująco:

```
[specyfikator_dostępu] class klasa_zewnętrzna {
  [specyfikator_dostepu] class klasa_wewnętrzna {
    /* ... pola i metody klasy wewnętrznej ... */
  }
  /* ... pola i metody klasy zewnętrznej ... */
}
```

Jeśli zatem ma zostać utworzona klasa o nazwie `Outside`, która będzie zawierała w sobie klasę `Inside`, należy zastosować kod:

```
public class Outside {
  class Inside {
  }
}
```

W klasie zewnętrznej można bez problemów oraz bez jakichkolwiek dodatkowych zabiegów programistycznych korzystać z obiektów klasy wewnętrznej. Można je tworzyć, a także bezpośrednio odwoływać się do zdefiniowanych w nich ich pól i metod. W jednej klasie wewnętrznej może istnieć dowolna liczba klas wewnętrznych, nie ma w tym względzie ograniczeń.

Klasa wewnątrz metody

Klasa wewnętrzna może się również znaleźć w dowolnym miejscu: między polami, między metodami, a także wewnątrz metod. Ta ostatnia sytuacja jest rzadko spotykana, z reguły bowiem jedynie zaciemnia czytelność kodu, nie przynosząc wielu praktycznych korzyści. Niemniej taka możliwość jest faktem. Taki przypadek został zobrazowany na poniższym listingu:

```
public class Outside {
  public void f() {
    class Inside {
      /* ... pola klasy Inside ... */
      public void g() {
        /*... instrukcje metody g ...*/
      }
      /*... dalsze metody klasy Inside ...*/
    }
    /*... dalsze instrukcje metody f ...*/
  }
  /*... dalsze metody klasy Outside ...*/
}
```

Należy pamiętać, że przy takiej definicji widoczność klasy `wewnętrznej` została ograniczona wyłącznie do metody, `w której została zdefiniowana`. Poza tą metodą jest ona nieznana i nie można się do niej odwoływać.

Typy klas wewnętrznych

Ponieważ klasy wewnętrzne są definiowane wewnątrz innych klas, w pewnym sensie można je traktować jako składowe tych klas, a zatem można też w stosunku do nich stosować modyfikatory dostępu. W związku z tym klasy wewnętrzne mogą być:

- pakietowe,
- publiczne,
- prywatne,
- chronione

i należy je traktować tak jak składowe wymienionych typów.

Wyjątki

Wyjątki w Javie

Do przechwytywania wyjątków służy blok instrukcji `try...catch` o schematycznej, podstawowej postaci:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku identyfikatorWyjątku){
  //obsługa wyjątku
}
```

W nawiasie klamrowym występującym po słowie `try` należy umieścić instrukcję lub instrukcje, które mogą spowodować wystąpienie błędu. W bloku występującym po `catch` należy umieścić kod, który ma zostać wykonany, kiedy wyjątek wystąpi. Przykładowe przechwycenie wyjątku powstającego przy próbie odwołania się do nieistniejącego indeksu tablicy będzie miało postać:

```
public static void main (String args[]) {
  int tab[] = new int[10];
  try{
    //przekroczenie indeksu tablicy
    tab[10] = 100;
  }
  catch(ArrayIndexOutOfBoundsException e){
    //przechwycenie wyjątku
    System.out.println("Nieprawidłowy indeks tablicy!");
  }
}
```

Wyjątek to nic innego jak obiekt, który powstaje, kiedy w programie wystąpi sytuacja wyjątkowa. Skoro wyjątek jest obiektem, to typ wyjątku (np. `ArrayIndexOutOfBoundsException`, `ArithmeticException`) jest klasą opisującą tenże obiekt, natomiast *`identyfikatorWyjątku`* to zmienna obiektowa, wskazująca na obiekt wyjątku. Na tym obiekcie można wykonywać operacje zdefiniowane w klasie wyjątku. Można np. uzyskać systemowy komunikat o błędzie. Wystarczy wywołać w tym celu metodę `getMessage()`. Obrazuje to poniższy przykład:

```
public static void main (String args[]) {
  try{
    int liczba = 10 / 0;
  }
  catch(ArithmeticException e){
    System.out.println("Wystąpił wyjątek arytmetyczny...");
    System.out.println("Komunikat systemowy:" +
    e.getMessage())
  }
}
```

Hierarchia wyjątków

Każdy wyjątek jest obiektem pewnej klasy. Klasy podlegają z kolei regułom dziedziczenia, zgodnie z którymi powstaje hierarchia klas. Wszystkie standardowe wyjątki, które możemy przechwytywać w aplikacjach za pomocą bloku `try...catch`, dziedziczą z klasy `Exception`, która z kolei dziedziczy z klas `Throwable` oraz `Object`.

Wynika z tego ważna właściwość: jeżeli dana instrukcja może wygenerować wyjątek typu `XYZ`, to można zawsze przechwycić wyjątek ogólniejszy, czyli wyjątek, którego typem będzie jedna z klas nadrzędnych do `XYZ`.

Przechwytywanie wielu wyjątków

W jednym bloku `try...catch` można przechwytywać wiele wyjątków. Konstrukcja taka zawiera wtedy jeden blok `try` i wiele bloków `catch`. Schematycznie wygląda ona następująco:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(KlasaWyjątku1 identyfikatorWyjątku1){
  //obsługa wyjątku 1
}
  catch(KlasaWyjątku2 identyfikatorWyjątku2){
  //obsługa wyjątku 2
}
/*
... dalsze bloki catch ...
*/
catch(KlasaWyjątkuN identyfikatorWyjątkuN){
  //obsługa wyjątku n
}
```

Po wygenerowaniu wyjątku jest sprawdzane, czy jest on klasy *`KlasaWyjatku1`*, jeśli tak — są wykonywane instrukcje obsługi tego wyjątku i blok `try...catch` jest opuszczany. Jeżeli jednak wyjątek nie jest klasy *`KlasaWyjatku1`*, jest sprawdzane, czy jest on klasy *`KlasaWyjątku2`* itd.

Przy tego typu konstrukcjach należy jednak pamiętać o hierarchii wyjątków, nie jest bowiem obojętne, w jakiej kolejności będą one przechwytywane. Ogólna zasada jest taka, że nie ma znaczenia kolejność, o ile wszystkie wyjątki są na jednym poziomie hierarchii. Jeśli jednak są przechwytywane wyjątki z różnych poziomów,

najpierw muszą to być wyjątki bardziej szczegółowe, czyli stojące niżej w hierarchii, a dopiero po nich wyjątki bardziej ogólne, czyli stojące wyżej w hierarchii.

Zagnieżdżanie bloków try...catch

Bloki `try...catch` można zagnieżdżać. To znaczy, że w jednym bloku przechwytującym wyjątek `X` może istnieć drugi blok, który będzie przechwytywał wyjątek `Y`. Schematycznie taka konstrukcja prezentuje się w taki sposób:

```
try{
  //instrukcje mogące spowodować wyjątek 1
  try{
    //instrukcje mogące spowodować wyjątek 2
  }
  catch (TypWyjątku2 identyfikatorWyjątku2){
    //obsługa wyjątku 2
  }
}
catch (TypWyjątku1 identyfikatorWyjątku1){
  //obsługa wyjątku 1
}
```

Zagnieżdżenie takie może być wielopoziomowe, czyli w już zagnieżdżonym bloku `try` można umieścić kolejny blok `try`, choć w praktyce takich piętrowych konstrukcji zazwyczaj się nie stosuje. Zwykle też nie ma takiej potrzeby. Maksymalny poziom zagnieżdżenia z reguły nie przekracza dwóch poziomów.

Zgłaszanie wyjątków

Zgłoszenie własnego wyjątku polega na utworzeniu nowego obiektu jednej z klas wyjątków i wykorzystaniu go jako argumentu instrukcji `throw`. Za pomocą instrukcji `new` należy utworzyć nowy obiekt klasy, która dziedziczy pośrednio lub bezpośrednio z klasy `Throwable`. W najbardziej ogólnym przypadku będzie to klasa

`Exception`. Tak utworzony obiekt powinien stać się parametrem instrukcji `throw`, np.:

```
throw new Exception();
```

Utworzenie obiektu wyjątku nie musi mieć miejsca bezpośrednio w instrukcji `throw`. Można utworzyć go wcześniej, przypisać zmiennej obiektowej i dopiero tę zmienną wykorzystać jako parametr dla `throw`, czyli zastosować konstrukcję:

```
Exception exception = new Exception();
//dalsze instrukcje
throw exception;
```

Jeśli taki wyjątek zostanie obsłużony przez znajdujący się w danym bloku (danej metodzie) blok `try...catch`, nie trzeba robić nie więcej. Jeśli jednak nie zostanie obsłużony, w specyfikacji metody, w której powyższa instrukcja wystąpi, niezbędne jest zaznaczenie, że będzie w niej zgłaszany taki wyjątek. Wymaga to zastosowania instrukcji `throws`, w ogólnej postaci:

```
specyfikator_dostępu [static] [final] typ_zwracany
nazwa_metody(argumenty)
throws KlasaWyjątku1, KlasaWyjątku2, ..., KlasaWyjątkun {
  //treść metody
}
```

Przykładowo jeśli w metodzie `main` będzie zgłaszany wyjątek klasy `Exception`, należy zastosować konstrukcję:

```
public static void main (String args[]) throws Exception {
  throw new Exception();
}
```

Jeżeli zgłaszany wyjątek ma otrzymać własny komunikat, należy przekazać go jako argument konstruktora klasy `Exception`:

```
throw new Exception("komuniat");
```

lub

```
Exception exception = new Exception("komuniat");
throw exception;
```

Ponowne zgłaszanie wyjątków

Raz przechwycony wyjątek można zgłosić ponownie (pot. „wyrzucić"), wykorzystując instrukcję `throw` w schematycznej postaci:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(typTyjątku identyfikatorWyjątku){
  //instrukcje obsługujące sytuację wyjątkową
  throw identyfikatorWyjątku
}
```

Sytuację taką prezentuje poniższy listing. W bloku `try` jest wykonywana niedozwolona instrukcja dzielenia przez zero. W bloku `catch` najpierw na ekranie jest wyświetlana informacja o przechwyceniu wyjątku, a następnie za pomocą instrukcji `throw` jest ponownie zgłaszany już przechwycony wyjątek. Ponieważ w programie nie ma innego bloku `try...catch`, który mógłby przechwycić ten wyjątek, zostanie on obsłużony standardowo przez maszynę wirtualną.

```
public static void main (String args[]) {
  try{
    int liczba = 10 / 0;
  }
  catch(ArithmeticException e){
    System.out.println("Tu wyjątek został przechwycony");
    //ponowne zgłoszenie wyjątku
    throw e;
  }
}
```

Tworzenie klas wyjątków

Java umożliwia tworzenie własnych klas wyjątków. Wystarczy napisać klasę pochodną pośrednio lub bezpośrednio z klasy `Throwable`, aby móc wykorzystywać ją do zgłaszania własnych wyjątków. W praktyce wyjątki najczęściej wyprowadzane są z klasy `Exception` i klas od niej pochodnych. Klasa taka w najprostszej postaci wygląda następująco:

```
public class nazwa_klasy extends Exception {
  //treść klasy
}
```

Przykładowo można utworzyć bardzo prostą klasę o nazwie `GeneralException` (wyjątek ogólny) w postaci:

```
public class GeneralException extends Exception {
}
```

Wykorzystanie tej klasy do zgłoszenia nowego wyjątku typu `GeneralException` obrazuje poniższy listing:

```
public static void main (String args[]) throws
GeneralException {
  throw new GeneralException();
}
```

Sekcja finally

Do bloku `try` można dołączyć sekcję `finally`, która będzie wykonana zawsze, niezależnie od tego, co będzie działo się w bloku `try`. Oznacza to, że instrukcje znajdujące się w tej sekcji zostaną wykonane nawet wtedy, gdy wyjątek nie zostanie zgłoszony. Schematycznie taka konstrukcja ma postać:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(){
  //instrukcje sekcji catch
}
finally{
  //instrukcje sekcji finally
}
```

Sekcję `finally` można zastosować również w sytuacji, gdy nie ma potrzeby przechwytywania wyjątku w bloku `catch`. Stosowana jest wtedy konstrukcja `try...finally` w postaci:

```
try{
  //instrukcje
}
finally{
```

```
    //instrukcje
  }
```

Również w tym przypadku instrukcje z bloku `finally` zostaną zawsze wykonane, niezależnie od tego, jakie instrukcje znajdą się w bloku `try`.