Programowanie Obiektowe i C++

Marcin Benke

8 stycznia 2007

Java

- Język obiektowy o składni podobnej do C++
- Niezależny od platformy (w zasadzie) — maszyna wirtualna
- Zarządzanie pamięcią — niepotrzebne obiekty automatycznie usuwane
- Program jest zbiorem klas — nie ma obiektów globalnych.
- Mechanizmy do programowania współbieżnego (wątki).
- Applety — mini-aplikacje wykonywane np. wewnątrz przeglądarki internetowej

Java vs C++

- Nie ma zmiennych/funkcji globalnych — zamiast nich używamy atrybutów/metod statycznych, np.

```
public static void main(String[] args) // ...
```

- Nie ma sekcji public/private — oznaczamy każdą składową.
- Nie ma wskaźników, tylko referencje.

```
String s = new String("Luna to surowa pani");
```

- Wszystkie klasy dziedziczą od `Object`.
- Nie ma przeciążania operatorów (ale jest przeciążanie metod).

Najprostszy program

```
public class Hello {
    public static void main(String[] args) {
    System.out.println("Hello!");
        // out jest atr. statycznym klasy System
    }
}
```

Każdą klasę umieszczamy w osobnym pliku o nazwie `NazwaKlasy.java` (w tym przypadku `Hello.java`).

Kompilacja i uruchomienie programu

Kompilacja programu

```
javac Hello.java
```

Uruchomienie

```
java Hello
```

Wyszukiwanie klas — zmienna CLASSPATH i opcja -classpath.

Referencje

- Do wszystkich obiektów programista uzyskuje dostęp poprzez referencje.
- Referencja może mieć wartość `null`, wtedy nie wskazuje na żaden obiekt.
- Odwołanie do referencji pustej powoduje wygenerowanie wyjątku java.lang.NullPointerException.
- Nie może istnieć referencja nie mająca wartości `null` i nie wskazująca na żaden obiekt.
- Referencja może wskazywać tylko na obiekt klasy, której jest typu, bądź jej nadklasy.

Referencje

- Nowe obiekty tworzy się przy pomocy new.
- Referencji można przypisać wartość w dowolnym momencie.
- Wiele referencji może wskazywać na ten sam obiekt.
- Usunięcie obiektu może nastąpić w momencie gdy ostatnia wskazująca go referencja zostanie usunięta, bądź zostanie jej przypisana wartość `null`.
- Za zwalnianie pamięci odpowiedzialna jest maszyna wirtualna. Brak operatora delete.

Przykład — referencje

```
public class Hello2 {
   public static void main(String[] args) {
      String greeting = new String("Hello");
      System.out.println(greeting);
      greeting = null;
   }
}
```

Przykład — wywoływanie metod

```
public class Hello3 {
  public static void main(String[] args) {
    String greeting = new String("Hello");
    for (int i = 0; i < greeting.length(); i++)
      System.out.println(greeting.charAt(i));
   // Nie możemy napisać greeting[i]
   // Nie ma przeciążania operatorów
  }
}
```

Pakiety

W C++ organizację programu wyznaczał podział na moduły (pliki); w Javie służą do tego **pakiety**.
Pakiety są przydatne z kilku powodów:

- pozwalają na grupowanie powiązanych ze sobą klas,
- klasy w pakiecie mogą korzystać z popularnych nazw, które inaczej mogłyby ze sobą kolidować
- mogą zawierać definicje klas i składowych, które są dostępne tylko wewnątrz pakietu.

Używanie pakietów

```
class Date1{
    public static void main(String[] args) {
        java.util.Date now = new java.util.Date()
        System.out.println(now);
    }
}
```

Używanie pakietów

```
import java.util.Date;
class Date2 {
    public static void main(String[] args) {
        Date now = new Date();
        System.out.println(now);
    }
}
```

Tworzenie pakietów

```
package geometry;
class Point {
    double x, y;
    public void moveto(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Przy tworzeniu własnych pakietów, pakiet `foo.bar.baz` powinien znaleźć się w katalogu foo/bar/baz.
Zmienna CLASSPATH i opcja -classpath.

Nazewnictwo pakietów

Jeżeli zamierzamy rozpowszechniać nasz pakiet, musimy zadbać aby jego nazwa nie kolidowała z już istniejącymi.
Sugerowana konwencja łączy nazwy pakietów z domenami internetowymi: firma `example.com` poprzedza nazwy swoich pakietów `com.example`.
Np. moja domena to `marcin.org`, więc powinienem napisać

```
package org.marcin.geometry;
class Point { // ...
```

Dziedziczenie i konstruktory

```
package mwt;
import javax.swing.JFrame;
import java.awt.Component;

public class MFrame extends JFrame {
 public MFrame(String title) {
  super(title);
  setDefaultLookAndFeelDecorated(true);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 }
 final void addContent(Component comp) {
  getContentPane().add(comp);
 }
}
```

Interfejsy

Zamiast klas abstrakcyjnych używamy *interfejsów*:

```
public interface Runny {
    int ANSWER = 42;
    void run();
}

public class DeepThought implements Runny {
    public void run() {}
}
```

Metody interfejsu są zawsze publiczne; nie mogą być statyczne.
Pola interfejsu należy rozumieć jako stałe.

Dziedziczenie interfejsów

Interfejs może dziedziczyć wiele interfejsów:

```
public interface Bar { void bar();}
public interface Baz { void baz();}
public interface Foo extends Bar,Baz {}
```

Klasy anonimowe

Czasem tworzymy klasę tylko po to, żeby przekazać jej obiekt jako parametr.
W Javie możemy utworzyć taką “jednorazówkę” w miejscu wywołania, np.

```
class Lola {
  public static void main(String[] args) {
    Thread lola;
    lola = new Thread(new Runnable() {
        public void run() {
          System.out.println("Run Lola, run");
        }
      });
    lola.start();
  }
}
```

Unicode

Dzięki Unicode możemy w programach używać znaków dowolnego języka

```
class Gżegżółka {
   public static void main(String[] args) {
      String gżegżółka = new String("Gżegżółka");
      System.out.println(gżegżółka);
   }
}
```

Wyjątki

Podobnie jak w C++, ale tylko podklasy klasy `Throwable`

```
public class Wyjątek extends Throwable {}
```

Metody muszą specyfikować wszystkie przekazywane wyjątki

```
public class Wyj {
    public static void main(String[] args)
                                    throws Wyjątek {
if(args.length <  1)
    throw new Wyjątek();
    }
}
```

Applety

```
import java.applet.*;
import java.awt.*;
public class HelloApplet extends Applet
{
    public void paint(Graphics g)
    {
      g.drawString("Hey hey hey",20,20);
      g.drawString("Hello World",20,40);
    }
    public void init() {} // Inicjalizacja
    public void stop() {} // Końcowe porządki
}
```

Biblioteki standardowe

- Java posiada bogaty zestaw bibliotek standardowych.
- Klasy, które dziś widzieliśmy: Applet, Thread, Date,. . .— to klasy biblioteczne
- Także standardowe biblioteki GUI: AWT i Swing (java.awt, javax.swing)
- Model GUI sterowany zdarzeniami

Zdarzenia

- Zdarzenie to przeważnie jakaś akcja użytkownika (np. kliknięcie myszą)
- Zdarzenie jest obiektem (pod)klasy Event
- Każdy komponet UI może zgłosić zainteresowanie pewną kategorią zdarzeń podając “słuchacza” — obiekt który będzie obsługiwał zdarzenia.
- Słuchacz musi implementować odpowiedni interface np. `MouseListener`.
- Można to osiągnąć np. rozszerzając klasy biblioteczne takie jak `MouseAdapter`.

ClickReporter

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

public class ClickReporter extends Applet
{
    public void init() {
        setBackground(Color.yellow);
        addMouseListener(new ClickListener());
    }
}
```

ClickListener

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

public class ClickListener extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
System.out.println("Mouse pressed at(" +
   event.getX() + "," +
   event.getY() + ").");
    }
}
```

CircleListener

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

public class CircleListener extends MouseAdapter
  private int radius = 25;
  public void mousePressed(MouseEvent event) {
    Applet app = (Applet)event.getSource();
    Graphics g = app.getGraphics();
    g.setColor(Color.blue);
    g.fillOval(event.getX()-radius,
               event.getY()-radius,
               2*radius, 2*radius);
  }
}
```

Słuchacz wewnątrz klasy

Tworzenie osobnej klasy dla jednej tylko metody może się nam wydawać przesadą, wolelibyśmy np. tak:

```
  public class CircleDrawer1
           extends Applet,MouseListener {
  public void init() {
    setBackground(Color.yellow);
    addMouseListener(this);
  }
  public void mousePressed(MouseEvent event) {
...
```

Niestety tak nie można (nie ma wielodziedziczenia).

Rozwiązania

Możemy zamiast tego:

- zaimplementować wszystkie metody `MouseListener`, lub
- zdefiniować lokalną klasę rozszerzającą `MouseAdapter`, lub
- użyć klasy anonimowej.

Nazwana klasa wewnętrzna

```
public class CircleDrawer2 extends Applet {
  private class CircleListener
                  extends MouseAdapter {
    private int radius = 25;
    public void mousePressed(MouseEvent event) {
      Graphics g = getGraphics();
      g.setColor(Color.blue);
      g.fillOval(event.getX()-radius,
                 event.getY()-radius,
                 2*radius, 2*radius);
    }}
  public void init() {
    setBackground(Color.yellow);
    addMouseListener(new CircleListener());
  }}
```

Anonimowa klasa wewnętrzna

```
public class CircleDrawer3 extends Applet {
  public void init() {
    setBackground(Color.yellow);
    addMouseListener(new MouseAdapter() {
    private int radius = 25;
    public void mousePressed(MouseEvent event) {
      Graphics g = getGraphics();
      g.setColor(Color.blue);
      g.fillOval(event.getX()-radius,
                 event.getY()-radius,
                 2*radius, 2*radius);
    }});
  }
}
```

Zaleta: zwięzłość; wada: słaba czytelność