Rozhraní a polymorfismus

4

Docela přirozeným způsobem jsme si v jednom z předchozích seminářů zavedli pojem *zapouzdření*. Připomeňme si, že to znamená, že všechny atributy objektu jsou skryty; okolní objekty k nim nemají přístup, a chtějí-li spolu komunikovat dva objekty, musí použít volání metod. Za takových okolností jednotlivé (veřejné, `public`) metody představují pomyslné *rozhraní* mezi vnějškem objektu a jeho atributy skrytými uvnitř. Proto soubor veřejných metod bývá někdy označován jako (veřejné) *rozhraní* objektu.

Rozhraní je pro objektově orientované programování klíčový pojem. Víme-li totiž, jak vypadá rozhraní objektu, tj. jaké má objekt metody, můžeme s objektem libovolně manipulovat. Nemusíme vědět nic o tom, co se děje uvnitř s jeho atributy, a nemusíme ani vědět, s objektem které třídy pracujeme. Vždy přistupujeme k objektu přes jeho rozhraní. Díky tomu, že jednotlivé objekty jsou zapouzdřeny, nepotřebujeme znát nic dalšího. To nám umožňuje pracovat s objekty různých typů jednotným způsobem. Tato schopnost pracovat s daty různých typů se označuje jako *polymorfismus*.

Poznámka 1 *S určitým zjednodušením lze opět najít analogii v reálném světě. Automobil má také své rozhraní (volant, pedály, řadící páku), kterým je ovládán. Abychom se automobilem dostali z místa A do místa B, v zásadě nám stačí znát, jak ovládat toto rozhraní, tj. jak zatočit, zrychlit, zabrzdit atd. O tom, co se děje v motoru auta v průběhu jízdy, nemusíme vědět nic.*[1] *Když umíme ovládat toto rozhraní, můžeme řídit jakékoliv auto se stejným rozhraním a to bez ohledu na to, zda se jedná o malé osobní auto, SUV nebo třeba dodávku.*

V objektovém programování se pojem *rozhraní* používá ve dvou významech. (i) V širším významu je to soubor všech veřejných metod, které daný objekt má, jak jsme tento pojem použili výše. (ii) V úzkém slova smyslu *rozhraní* označuje prvek programovacího jazyka, který umožňuje přesně určit, jaké metody objekt má mít. V tomto významu chápeme rozhraní jako pojmenovaný výčet metod, které objekt s daným rozhraním určitě musí mít, a je možné je tedy použít.

U jednotlivých tříd se specifikuje, jaké rozhraní mají, tj. jaké metody určitě budou mít objekty dané třídy. Chceme-li pak pracovat s nějakým objektem, stačí nám vědět, že daný objekt má námi požadované rozhraní a nemusíme se spoléhat na to, že se jedná o objekt konkrétní třídy. Tím se dá sjednotit práce s objekty různých tříd. Ukažme si to nyní prakticky.

Deklarace rozhraní je v Javě podobná deklaraci třídy. Rozhraní se deklaruje v samostatném souboru, je použito klíčové slovo `interface` (místo `class`) a jsou deklarovány pouze veřejné metody bez svého těla.

Uvažujme rozhraní pro rovinné geometrické útvary.

[1] Pro názornost silně zjednodušujeme.

```java
/** Rozhrani objektu rovinnych geom. utvaru */
public interface Shape {

  /** vypise informace o geom. utvaru */
  public void printOut();

  /** vrati plochu geom. utvaru */
  public double getArea();
}
```

Rozhraní `Shape` nám říká, že každý objekt s tímto rozhraním má minimálně dvě metody – `printOut()`, která vypíše informaci o objektu, a `getArea()`, která vrací informaci o velikosti plochy daného geometrického útvaru.

Pokud má třída nějaké rozhraní (v úzkém slova smyslu), říkáme, že jej *implementuje*. Tato informace se zapisuje za klíčové slovo `implements` v deklaraci třídy. Použití si ukážeme na dvou třídách představujících obdélník a kruh. Z důvodu úspory místa jsou z kódu vypuštěny deklarace atributů a konstruktoru.

```java
public class Rectangle implements Shape {

  // ... atributy a konstruktor

  public void printOut() {
    System.out.println("Obdelnik o stranach " + a + ", " + b);
  }

  public double getArea() {
     return a * b;
  }
}
```

```java
public class Circle implements Shape {

  // ... atributy a konstruktor

  public void printOut() {
    System.out.println("Kruh o polomeru " + r);
  }
```

```
  public double getArea() {
     return Math.PI * r * r;
  }
}
```

Důležité je, že na *rozhraní* se můžeme dívat také jako na typ, který zastřešuje třídy se stejným rozhraním. Jinými slovy, v kódu můžeme použít jako typ (např. atributu, proměnné) také název rozhraní a jako hodnotu použít libovolný objekt s tímto rozhraním. Můžeme tedy napsat následující:

```
Shape x = new Rectangle(10, 20);
Shape y = new Circle(10);

// urci, ktery objekt ma vetsi plochu
Shape largest = null;
if (x.getArea() > y.getArea()) largest = x;
else largest = y;

// vypise popis objektu
largest.printOut();
```

Všimněte si, že ve výše zmíněném kódu pracujeme jen s metodami, které zaručuje rozhraní `Shape`. Prakticky nás nezajímá, jestli pracujeme s kružnicí, či obdélníkem. Rozhodující pro nás je, že můžeme získat velikost plochy rovinného útvaru a vypsat informace o tomto útvaru.

Ve druhém příkladu si vytvoříme metodu, která vypíše informace o všech předaných rovinných útvarech a jejich ploše.

```
/** vypise jednotlive geom. utvary a jejich plochu */
public static void printShapes(Shape[] shapes) {
  for (Shape s: shapes) {
    s.printOut();
    System.out.println(s.getArea());
  }
}
```

Tuto metodu můžeme volat s parametry různých typů.

```
Rectangle[] rects = { new Rectangle(10, 20), new Rectangle(3, 7) };
Circle[] circles  = { new Circle(10), new Circle(3) };
Shape[] shapes    = { new Rectangle(4, 8), new Circle(10), new Rectangle(15, 20) };
```

```
printShapes(rects);
printShapes(circles);
printShapes(shapes);
```

I když jsou objekty typu `Rectangle` a `Circle` úplně odlišné, může s nimi metoda `printShapes` bezpečně pracovat, protože od nich požaduje jen, aby měly rozhraní `Shape`, což je splněno. Jak vidíme, polymorfismus nám umožňuje vytvářet obecné algoritmy pracující s různými typy dat. Ukažme si ještě složitější příklad, metodu, která najde největší geometrický útvar v poli.

```
/** vraci pozici v poli, kde lezi nejvetsi geom. utvar */
public static int largest(Shape[] shapes) {

  // index nejvetsiho nalezeneho objektu
  int index = -1;

  // velikost nejvetsiho nalezeneho objektu
  // na zacatku se nastavi na specialni
  // hodnotu, ktera odpovida zapornemu nekonecnu
  double max = Double.NEGATIVE_INFINITY;

  // projdeme vsechny geom. utvary
  for (int i = 0; i < shapes.length; i++) {
    if (shapes[i].getArea() > max) {
      index = i;
      max = shapes[i].getArea();
    }
  }
  return index;
}
```