9 Generické třídy a metody

1 Generické třídy

1.1 Motivace: neomezený seznam objektů

Uvažujme, že bychom chtěli vytvořit třídu `Receipt` reprezentující účtenku z obchodu. Při řešení tohoto úkolu se nabízí použít pole pro uložení jednotlivých položek účtenky. Nejpřímočařejší řešení by mohlo vypadat přibližně takto:

```
public class Receipt {

  /** pole s jednotlivymi polozkami */
  private ReceiptItem[] items;

  /** aktualni pocet polozek na seznamu */
  private int size;

  public Receipt() {
    // inicializace seznamu
    items = new ReceiptItem[10];
    size = 0;
  }

  /** vlozi polozku do seznamu */
  public void add(ReceiptItem item) {
    this.items[this.size] = item;
    this.size++;
  }
}
```

Toto řešení má několik nedostatků. Pole má pevně stanovenou délku, v tomto případě na 10 prvků. Pokusíme-li se do pole vložit jedenáctou položku, dojde k výjimce a program bude ukončen, protože pro jedenáctý prvek pole není vyhrazeno v paměti místo, a jedná se tedy o chybu v programu. Pokud bychom

zvýšili počet položek v poli (třeba na 100 nebo 1000), problém by to neřešilo, jen dočasně odsunulo. Pořád by hrozilo, že někdo na účet zapíše více položek než je možné. Navíc by se začal projevovat další problém – nadměrná spotřeba paměti. Vytvoříme-li pole, je pro něj vždy alokávano místo v paměti tak, aby bylo možné do něj uložit všechny položky. Máme-li tedy pole o velikosti 1000 prvků (co kdyby náhodou byly potřeba) a v něm uloženy dvě nebo tři hodnoty, je to špatně obhajitelné plýtvání pamětí.

Možným řešením by bylo udělat pole „nafukovací" tak, aby se průběžně přizpůsobovalo množství uložených položek. Toho se dá dosáhnout úpravou metody `add`.

```
1  public void add(ReceiptItem item) {
2    if (size == items.length) {
3      ReceiptItem[] newArray = new ReceiptItem[items.length * 2];
4      for (int i = 0; i < items.length; i++) {
5        newArray[i] = items[i];
6      }
7      items = newArray;
8    }
9
10   items[size] = item;
11   size++;
12 }
```

Na druhém řádku ověříme, zda je v poli `items` místo pro ještě jeden prvek. Pokud není, je potřeba pole rozšířit tak, aby pojmulo i další prvky. Proto si na třetím řádku vytvoříme pole `newArray` o dvojnásobné velikosti a v cyklu na řádcích 4 až 6 jednotlivé prvky překopírujeme z původního pole do pole nového.[1] Následně určíme, že atribut `this.items` bude odkazovat na nově vytvořené pole. Díky tomu bude naše třída budit dojem, že je „nafukovací" a je schopna pojmout libovolné množství prvků.[2] Jedná se však pouze o dojem, dojde-li místo v aktuálním poli, vždy vytvoříme pole nové, a to původní je z paměti následně odstraněno garbage collectorem.

Už tedy víme, že se při práci se souborem hodnot nemusíme omezovat na pole pevné velikosti. Co kdybychom místo seznamu položek nákupu chtěli mít podobný „nafukovací" seznam přečtených knih, zhlédnutých filmů, telefonních čísel, oblíbených prvočísel, bodů na přímce? Znamená to, že bychom si pro každý typ uložených dat měli vytvářet vlastní variantu takového seznamu? Mohli bychom, ale není to dobrý nápad. Při programování bychom se měli vyhnout psaní stejného nebo podobného kódu. Je to zbytečně strávený čas a hlavně, pokud je potřeba udělat nějakou změnu, znamená to měnit všechny varianty opakujícího se kódu.

[1] Pro překopírování obsahu pole bychom mohli ještě použít metodu `System.arraycopy`.

[2] Pochopitelně množství prvků je vždy omezeno dostupnou pamětí.

1.2 Seznam obecných objektů

Obecné řešení můžeme postavit na předpokladu, že všechny objekty mají společného předka, kterým je třída `Object`. S každým objektem, bez ohledu na to jaké je třídy, můžeme nakládat jako s objektem třídy `Object`. Máme-li tedy pole typu `Object[]`, můžeme do něj uložit libovolné objekty. Není proto složité převést třídu `Receipt` na obecný seznam, stačí místo typu `ReceiptItem` použít obecnější typ, například ten nejobecnější typ `Object`.

```
1  public class MyList {
2    private Object[] items;
3    private int size;
4
5    public MyList(int capacity) {
6      items = new Object[capacity];
7      size = 0;
8    }
9
10   public void add(Object item) {
11     if (size  == items.length) {
12       Object[] newArray = new Object[items.length * 2];
13       for (int i = 0; i < items.length; i++)
14         newArray[i] = items[i];
15       items = newArray;
16     }
17     items[size++] = item;
18   }
19
20   public Object get(int index) {
21     return items[index];
22   }
23
24   public int size() {
25     return size;
26   }
27 }
```

Třída `MyList` se od svého vzoru liší v použitém typu hodnot, které uchovává. Abychom si mohli ukázat některá úskalí tohoto řešení, doplnili jsme do třídy ještě metodu `size()`, která vrací aktuální počet prvků v seznamu, a metodu `get(int)` vracející položku s daným číslem.

Když si třídu `MyList` vyzkoušíme, zjistíme, že v základních obrysech plní naše potřeby. Dejme tomu, že bychom chtěli mít seznam objektů třídy `Light`, se kterými jsme pracovali v předchozích seminářích.

```
MyList lights = new MyList(10);
lights.add(new Light("red"));
lights.add(new Light("yellow"));
lights.add(new Light("green"));
```

Vložení prvků je přímočaré, protože metoda `add(Object)` bere jako svůj parametr libovolného potomka třídy `Object`, tj. jakýkoliv objekt. Chceme-li však přistoupit k některé z položek seznamu, máme zaručené pouze to, že se jedná o objekt, který je potomkem třídy `Object`. To znamená, že daný objekt musíme přetypovat na odpovídající typ.

```
Light redLight = (Light) lights.get(0);
redLight.turnOn();
```

Kdybychom si vytvořili metodu, která pro seznam objektů třídy `Light` projde všechny objekty na seznamu a zavolá metodu `turnOn()`, mohla by vypadat následovně:

```
1 public static void turnOnLights(MyList lights) {
2   for (int i = 0; i < lights.size(); i++) {
3     ((Light) lights.get(i)).turnOn();
4   }
5 }
```

Tato metoda funguje správně, ale pouze pokud seznam obsahuje objekty třídy `Light`. Co by se stalo, kdybychom zkusili do seznamu vložit hodnotu jiného typu. Například:

```
MyList lights = new MyList(10);
lights.add(new Light("red"));
lights.add(new Light("yellow"));
lights.add(new Light("green"));
lights.add(new Car("1A24816", "blue"));
```

Na posledním řádku jsme do seznamu vložili místo světla automobil. I když to na první pohled vypadá jako zjevná chyba, program půjde přeložit a spustit. Objekt třídy `Car` je totiž potomkem třídy `Object` a metoda `add` akceptuje libovolný takový objekt jako svůj parametr. Kdybychom tento seznam předali metodě `turnOnLights(lights)`, její provádění skončí výjimkou, protože na řádku č. 3 při přetypování počítáme s tím, že v seznamu jsou pouze objekty třídy `Light`. Jinými slovy naše řešení je použitelné, ale musíme si sami hlídat, abychom nemíchali objekty různých tříd v jednom seznamu.

Podrobme naši třídu `MyList` ještě jedné zatěžkávací zkoušce. Zkusme do ní vložit čísla.

```
MyList numbers = new MyList(10);
numbers.add(1);
```

```
numbers.add(42);
numbers.add(6.6);
```

Pozorného čtenáře asi napadne, že takový kód nedává smysl, jelikož čísla[3] nejsou objekty, ale primitivní hodnoty. Tato úvaha je správná, avšak překladač jazyka Java ve skutečnosti provádí tzv. *autoboxing*. To znamená, že pokud narazí na primitivní hodnotu na místě, kde se očekává objekt, vezme tuto hodnotu a zabalí ji do objektu, tzv. *obalové třídy*. Předchozí kód pak přibližně odpovídá:

```
MyList numbers = new MyList(10);
numbers.add(new Integer(1));
numbers.add(new Integer(42));
numbers.add(new Double(6.6));
```

Poznámka 1 *Ve výše zmíněném příkladu jsme pro lepší názornost použili přímo operátor* `new` *a volání konstrukturu obalové třídy. Ve skutečnosti se takto objekty nevytváří, ale místo toho se používá metoda* `Integer.valueOf(int)` *nebo* `Double.valueOf(double)`*, která zajistí, že pro nejčastější hodnoty se objekt použije opakovaně a dojde tak k úspoře paměti.*

Jinými slovy, na místech, kde se očekává objekt, můžeme použít i primitivní hodnotu, protože je automaticky převedena na vhodný objekt. Pro každý primitivní datový typ existuje obalová třída, kromě zmíněných `Integer` a `Double` jsou to například `Float`, `Long`, `Boolean` nebo `Character`.

Aby práce s objekty obalových tříd byla co nejpohodlnější, podporuje Java ještě tzv. automatický *unboxing*. Jinými slovy narazí-li překladač na objekt obalové třídy na místě, kde se očekává primitivní hodnota, provede automatickou konverzi, například:

```
Integer x = Integer.valueOf(41);
int y = x + 1;
```

Překladač převede na:

```
Integer x = Integer.valueOf(41);
int y = x.intValue() + 1;
```

Poznámka 2 *Jazyk Java byl navržen v první polovině devadesatých let, kdy počítače byly nesrovnatelně pomalejší (procesory pracovaly na frekvenci o desítkách MHz) a měly nesrovnatelně menší paměť (jednotky MB). Aby šlo jazyk Java na takových počítačích efektivně implementovat, rozhodli se jeho autoři, že nezavedou čísla, znaky a pravdivostní hodnoty, jako objekty, ale jako speciální primitivní datové typy. Aby se setřel rozdíl mezi objekty a těmito primitivními datovými typy byly do jazyka přidány obalové třídy, autoboxing a unboxing.*

[3] Stejně jako třeba znaky nebo pravdivostní hodnoty.

Poznámka 3 *I když díky autoboxingu a autounboxing se stírají rozdíly mezi hodnotami typu* `double` *a* `Double` *nebo* `boolean` *a* `Boolean`*, je potřeba mezi těmito typy důsledně rozlišovat. V prvním případě se jedná o primitivní datový typ, se kterým umí JVM efektivně pracovat, v druhém případě se jedná o třídu, jejíž instance jsou (neměnitelné) objekty, se kterými může být práce náročnější z pohledu spotřebované paměti i procesorového času. Proto je vhodnější používat spíše primitivní datové typy a obalové třídy používat pouze v situaci, kdy neexistuje alternativa.*

1.3 Seznam jako generická třída

Předchozí řešení má dvě nepříjemné vlastnosti. (i) Do seznamu je možné předat jakýkoliv objekt. (ii) Ze seznamu jsme schopni získat pouze objekty bez informace o jejich typu a musíme tak hodnoty explicitně přetypovávat. Aby se takovým problémům dalo předejít, byly do jazyka Java zavadeny tzv. *generické třídy*. To jsou třídy u nichž se dají specifikovat typy hodnot, se kterými jejich objekty budou pracovat. Na generickou třídu se můžeme dívat jako na určitou „šablonu", kterou lze přizpůsobovat různým typům dat.

Deklarace generické třídy je podobná deklaraci běžné třídy, jen s tím rozdílem, že za název třídy uvedeme do špičatých závorek seznam tzv. *typových parametrů*, které pak můžeme používat na libovolných místech třídy, kde je očekáván typ. Všechny výskyty tohoto typového parametru, budou následně při použití třídy nahrazeny konkrétním datovým typem. Jaký tento typ bude, je určeno buď při deklaraci proměnné nebo při volání konstruktoru. Naši třídu `MyList` bychom tedy mohli převést na generickou následovně.

```
public class MyList<T> {

  private T[] items;

  public void add(T item) { /* ... */ }
  public T get(int index) { /* ... */ }

  /* ... */
}
```

V tomto případě jsme vytvořili třídu s jedním typovým parametrem – `T`. Budeme-li chtít vytvořit objekt, je nutné uvést, jaký typ se má použít místo `T`. Tato informace se opět uvádí za název třídy do špičatých závorek, např.

```
MyList<Light> lights = new MyList<Light>(10);
MyList<String> strings = new MyList<String>(10);
```

Každý takto vytvořený objekt můžeme vnímat jako specializovanou variantu seznamu pro daný typ, tj. přibližně následovně.

```
public class MyList {
```

```
  private Light[] items;

  public void add(Light item) { /* ... */ }
  public Light get(int index) { /* ... */ }

  /* ... */
}
```

```
public class MyList {

  private String[] items;

  public void add(String item) { /* ... */ }
  public String get(int index) { /* ... */ }

  /* ... */
}
```

Velkou výhodou tohoto řešení je, že není možné míchat hodnoty různých typů.

```
MyList<String> strings = new MyList<String>(10);
strings.add("text");
strings.add(10); // chyba

MyList<Light> lights = new MyList<Light>(10);
lights.add(new Light("red"));
lights.add(new Car("1A24816", "blue")); // chyba
```

Jelikož víme, jaké hodnoty jsou v seznamu, můžeme se zbavit i zbytečných přetypování:

```
1 public static void turnOnLights(MyList<Light> lights) {
2   for (int i = 0; i < lights.size(); i++) {
3     lights.get(i).turnOn();
4   }
5 }
```

U této metody jsme upřesnili typ parametru jako `MyList<Light>` a na řádku č. 3 jsme odstranili explicitní přetypování.

Poznámka 4 *Dají-li se typové parametry odvodit z typu proměnné, nemusíme je v operátoru* `new` *uvádět a můžeme použít jen* `<>` *(tzv. diamant). Například místo:*

```
MyList<Light> lights = new MyList<Light>(10);
```

Lze psát jen:

```
MyList<Light> lights = new MyList<>(10);
```

Poznámka 5 *Informace o použitých hodnotách typových parametrů jsou k dispozici pouze během překladu programu, z technických důvodů jsou odstraněny a běžící program o nich již nemá žádnou informaci. To má za následek, že nejsme schopni udělat například* `new T[10]`*. Tento nedostatek se dá obejít tak, že si vytvoříme pole objektů a přetypujeme jej, tj.* `(T[]) new Object[10]`*. To si vyžádá i drobnou změnu v konstruktoru a metodě* `add`*.*

```
1 public MyList(int capacity) {
2   items = (T[]) new Object[capacity];
3   size = 0;
4 }
```

Je možné, že překladač zobrazí upozornění, že přetypování na řádku č. 2 je nekorektní. To ale můžeme ignorovat, v tomto případě se bude jednat o planý poplach. Úplný kód našeho obecného seznamu je přiložen jako třída `MyListGeneric`*.*

1.4 Upřesnění typových parametrů

Dosud jsme uvažovali, že za typový parametr můžeme použít libovolnou třídu, což může v extremním případě znamenat, že jako typový parametr bude použita třída `Object`, a tedy o objektech, se kterými pracujeme, nemáme příliš specifické informace.

Uvažujme třídu pro reprezentaci bodu v dvourozměrném systému souřadnic. Ta by mohla být definovaná následovně.

```
1  public class Point<T> {
2    private final T x;
3    private final T y;
4    public Point(T x, T y) {
5      this.x = x;
6      this.y = y;
7    }
8    public double distance(Point<T> that) {
9      return ???;
10   }
11 }
```

Jak v takové třídě spočítat vzdálenost dvou bodů, když za `T` může být dosazena libovolna třída? Typový systém jazyka Java nám umožňuje závést omezení na typové paremetry pomocí dvou klíčových slov `extends`

a `super`, kdy uvedeme, že typ musí být alespoň daného typu (`extends`) nebo nanejvýš daného typu (`super`) v hierarchii typů.

V našem ukázkovém příkladu bychom to využili následovně. V deklaraci třídy bychom za typový parametr T uvedli `extends Number`, což nám zajistí, že jako typový parametr můžeme použít pouze třídu nebo některého z jejich potomků, např. `Integer`, `Double`.

```
public class Point<T extends Number> { /* ... */ }
```

Díky tomu budeme mít zajištěné, že atributy `x` a `y` budou mít metodu `doubleValue()`, kterou můžeme použít pro výpočet vzdálenosti.

```
public double distance(Point<T> that) {
  double dx = this.x.doubleValue() - that.x.doubleValue();
  double dy = this.y.doubleValue() - that.y.doubleValue();
  return Math.sqrt(dx * dx - dy * dy);
}
```

Úkoly k procvičení 1 *Vyzkoušejte vytvořit instance třídy* `Point` *s různými typovými paremetry.*

1.5 Metody s generickými typy

Na každou generickou třídu s typovými parametry je potřeba nahlížet jako na samostatný datový typ. Vezměme si například statickou metodu, která vypíše obsah seznamu `MyList`. Přirozeně chceme, aby tato funkce byla co nejobecnější. Proto se nabízí použit třídu `Object`, jako typový parametr.

```
public static void printOut(MyList<Object> list) {
  for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
  }
}
```

Pokud tuto metodu zkusíme použít pro různé typové parametry, bude tato metoda fungovat jen pro typ `Object`.

```
MyList<Object> objects = new MyList<Object>(10);
MyList<Integer> integers = new MyList<Integer>(10);
printOut(objects);
printOut(integers); // nejde přeložit, protože MyList<Integer> != MyList<Object>
```

Abychom se vypořádali s tímto problémem, můžeme místo konkrétního datového typu použít, tzv. *žolíka* (wildcard), tj. symbol `?` (otazník). Který nám říká, že nezáleží na typu konkrétního typového parametru. V našem případě by to bylo:

```
public static void printOut(MyList<?> list) {
  for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
  }
}
```

Pokud použijeme žolíka, může být jako typový parametr použit libovolný typ, v extremním případě je to typ `Object`, takže opět nemáme příliš specifické informace o objektech. Avšak podobně jako v případě typových paremtrů tříd, můžeme rozsah typů upřesnit.

Pokud bychom chtěli udělat metodu, které spočítá průměr všech čísel v seznamu `MyList`, musíme upřesnit, že této metodě je možné předat pouze seznam čísel. Toho dosáhneme uvedením omezení typu `extends` za žolíka. Například následovně:

```
public static double average(MyList<? extends Number> numbers) {
  if (numbers.size() == 0) return Double.NaN;
  double total = 0;
  for (int i = 0; i < numbers.size(); i++) {
    Number value = numbers.get(i);
    total += value.doubleValue();
  }
  return total / numbers.size();
}
```

Pokud bychom z nějakého důvodu chtěli, aby byly jako typový parametr použity typy nejvýše `Double`, použili bychom `super`.

```
MyList<? super Double> numbers
```

2 Generické metody

Typové parametry nemusí být uvedeny jen u tříd. V případě potřeby můžeme typové parametry uvádět i u jednotlivých metod. Typový parametr si uvádí při deklaraci metody před typ návrátové hodnoty. Jak ukazuje následující příklad.

```
public static <T> boolean areEqual(T objectA, T objectB) {
  return objectA.equals(objectB);
}
```

Tato metoda má jeden typový parametr T, který určuje typ obou argumentů. Jaký typ se konkrétně bude uvažovat při volání, určí překladač tak, aby se jednalo, o co nejkonkrétnější typ, avšak v tomto případě můžeme uvažovat i s typem `Object`.

Proto dává smysl jednotlivé typové parametry upřesnit pomocí `extends` nebo `super`, jak ukazuje následující funkce pro výpočet minima dvou čísel.

```
public static <T extends Number> T min(T a, T b) {
  if (a.doubleValue() < b.doubleValue()) return a;
  return b;
}
```

Vyzkoušejme si, že typy argumentů i návratových hodnot musí být kompatibilní.

```
int min1 = min(1, 2);
int min2 = min(1, 2.0);     // nepůjde přeložit
double min3 = min(1, 2.0);  // nepůjde přeložit
double min4 = min(1.0, 2.0
```

Typové parametry metod můžeme kombinovat i s typovými parametry argumentů metod, jak ukazuje následující metoda, která naplní seznam `MyList` zadanou hodnotou.

```
public static <T> void fill(int count, T object, MyList<T> list) {
  for (int i = 0; i < count ; i++)
    list.add(object);
}
```

Díky takto určeným argumentům, máme zaručené, že nepůjde do seznamu vložit nekompatibilní hodnota.

```
MyList<String> list = new MyList<String>(10);
fill(10, "abc", list);
fill(10, false, list); // nepůjde přeložit
```