Vnořené třídy a lambda výrazy 11

Nejčastěji jsou jednotlivé třídy deklarovány v samostatných souborech, kde jedna třída odpovídá jednomu souboru, což napomáhá orientaci v projektu. Není to však jediný způsob, jak deklarovat třídy. Ukážeme si proto další možnosti.

Jednotlivé možnosti budeme demonstrovat s pomocí objektu, který se označuje jako *iterátor* a slouží k procházení kolekcí objektů. Pro tento typ objektů existuje v Javě standardní rozhraní `java.util.Iterator<E>` mající dvě metody:

```
public interface Iterator<E> {
  public boolean hasNext();
  public E next();
}
```

Metoda `hasNext()` vrací `true`, pokud je možné získat další prvek z kolekce, a metoda `next()` vrací další prvek z kolekce, pokud existuje. Objekty s tímto rozhraním se běžně používají k průchodu kolekcí pomocí cyklu `while`.

```
Iterator<Object> iterator = /* ... */;
while (iterator.hasNext()) {
  Object obj = iterator.next();
  System.out.println(obj);
}
```

V jednom z předešlých seminářů jsme si ukázali, jak vytvořit třídu `MyList` představující jednoduchý seznam hodnot. Nyní se podíváme na to, jak bychom mohli postupovat, kdybychom do této třídy chtěli přidat podporu iterátoru.

1 Vnořené třídy

Jazyk Java umožňuje deklarovat třídu uvnitř jiné třídy, tj. třída je deklarována na úrovni jednotlivých atributů nebo metod. Tento typ deklarace tříd nám umožňují sloučit (pomocné) třídy tak, aby věci, které spolu souvisí, byly pohromadě v jednom souboru se zdrojovými kódy.

1.1 Statické vnořené třídy

Třídu implementující rozhraní `Iterator` pro náš seznam můžeme chápat jako pomocnou a dává smysl ji přidružit do samotné třídy `MyList`, jak ukazuje následující kód.

```
public class MyList<T> {
  private T[] items;
  private int size;

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

  public Iterator<T> iterator() {
    return new MyIterator<>(this);
  }

  public static final class MyIterator<E> implements Iterator<E> {
    private final MyList<E> list;
    private int index;
    public MyIterator(MyList<E> list) {
      super();
      this.list = list;
      this.index = 0;
    }
    public boolean hasNext() {
      return (index < list.size);
    }
    public E next() {
      if (index >= list.size) throw new NoSuchElementException();
      E result = list.get(index);
      index++;
      return result;
    }
  }
}
```

Třída `MyIterator` je v tomto případě deklarovaná jako `static`, což znamená, že může přistupovat pouze ke statickým atributům a metodám nadřazené třídy. Z pohledu možností, které tato třída má, se nijak neliší od tříd, které jsou deklarovány v samostatných souborech. Všimněme si, že v metodě `MyList.iterator()`,

vytváříme instanci této třídy běžným způsobem.

Kdybychom chtěli metodu a třídu použít, budeme postupovat očekávetelným způsobem.

```
MyList<Integer> list = new MyList<Integer>(10);
list.add(10);
list.add(20);
list.add(30);

Iterator<Integer> iterator = list.iterator();
while (iterator.hasNext()) {
  System.out.println(iterator.next());
}
```

Protože je třída `MyIterator` deklarována jako `public`, můžeme instanci této třídy vytvořit i explicitně.

```
Iterator<Integer> iterator = new MyList.MyIterator<>(list);
```

Většinou však chceme pomocné třídy skrýt, proto by bylo vhodnější třídu `MyIterator` deklarovat jako `private`.

Poznámka 1 *V podobném duchu můžeme použít i deklarace výčtových typů (*`enum`*) a tříd typu záznam (*`record`*), které jsou implicitně brány jako* `static`*, pokud jsou deklarovány uvnitř třídy.*

1.2 Nestatické vnořené třídy

V předchozím případě jsme třídu `MyIterator` označili modifikátorem `static`, pokud toto označení vypustíme, získáme možnost přistupovat ke všem metodám a atributům nadřazené třídy. V našem případě, by se díky tomu kód mohl zjednodušit.

```
public Iterator<T> iterator() {
  return new MyIterator();
}
private final class MyIterator implements Iterator<T> {
  private int index = 0;
  public boolean hasNext() {
    return (index < size);
  }
  public T next() {
    if (index >= size) throw new NoSuchElementException();
    T result = items[index];
```

```
    index++;
    return result;
  }
}
```

Všimněme si především, že metody `hasNext()` a `next()` pracují přímo s atributy `size` a `items`, které jsou deklarovány ve třídě `MyList`.

Při vytvoření instance třídy `MyIterator` v metodě `iterator()` se do objektu (interně) vloží odkaz na objekt nadřazené třídy, díky kterému je možné přistupovat k jednotlivým atributům a metodám. Z toho také plyne, že nemůžeme vytvořit instance třídy `MyIterator` mimo tuto třídu.

1.3 Anonymní třídy

Pozornému čtenáři jistě neuniklo, že ve všech zatím uvažovaných případech třídu `MyIterator` používáme právě jednou. V situacích, kdy máme třídu použitou právě jednou, můžeme využít tzv. *anonymních tříd*, kdy deklarace třídy a samotné vytvoření instance této třídy jsou spojeny v jeden krok.

V kódu se to zapisuje:

```
Foo bar = new Foo() {
  // atributy a metody
};
```

Za operátor `new` uvedeme název rodičovské třídy a předáme argumenty konstruktoru rodičovské třídy. Následně do složených závorek uvedeme jednotlivé atributy a metody námi vytvořené třídy.

Při implementaci iterátoru bychom to využili následovně.

```
public Iterator<T> iterator() {
  return new Iterator<>() {
    int index = 0;
    public boolean hasNext() {
      return (index < size);
    }
    public T next() {
      if (index >= size) throw new NoSuchElementException();
      T result = items[index];
      index++;
      return result;
    }
  };
}
```

Zajímavou vlastností anonymních tříd je, že mají přístup k lokálním proměnným a argumentům metody, kde byly vytvořeny, ty však musí být deklarovány jako konstanty nebo se s nimi tak musí pracovat (tj. nesmí se v metodě ani vnořené třídě měnit.)

Prakticky to ukazuje následující kód vracející iterátor, který prochází hodnoty ze zadaného rozsahu, který je dán argumenty metody `rangeIterator`.

```
public static Iterator<Integer> rangeIterator(int from, int to) {
  return new Iterator<Integer>() {
    int currrent = from;
    public boolean hasNext() {
      return currrent < to;
    }
    public Integer next() {
      int result = currrent;
      currrent++;
      return result;
    }
  };
}
```

Poznámka 2 *Prakticky je možné uvnitř metody deklarovat třídu včetně jejího jména, jako jsme to viděli u nestatických vnořených tříd. Tato možnost se však v praxi nepoužívá.*

Ukázali jsme si několik způsobů, jak pomocí vnořených tříd implementovat podporu iterátoru. Drobnou změnou můžeme naši třídu `MyList` ještě významně vylepšit.

Protože explicitní práce s iterátorem je nepříliš pohodlná, byl do Javy přidám cyklus `for`-each. Tento cyklus umožňuje procházet kolekce implementující rozhraní `Iterable<T>`, což je rozhraní mající právě jednu abstraktní metodu `Iterator<T> iterator()`. Doplníme-li do deklarace třídy `MyList` toto rozhraní:

```
public class MyList<T> implements Iterable<T>
```

Bude možné tento seznam procházet pomocí cyklu `for`-each.

1.4 Lambda-výrazy

Při práci se seznami často narazíme na to, že je nutné je nějak transformovat, filtrovat hodnoty nebo prostě projít všechny hodnoty v seznam. Ukažme si, jak bychom postupovali, kdybychom do našeho seznamu chtěli přidat metodu `filter`, která vrátí nový seznam, kde budou pouze hodnoty splňující zadanou podmínku.

Předtím než se vůbec k návrhu takové metody dostaneme, je potřeba se zamyslet, jak reprezentovat podmínku. Náš seznam by měla být obecná třída, která může pracovat s libovolnými hodnotami, např. celými čísly,

řetězci, automobily apod. Podmínka by proto měla být formulovatelná taky obecně. Mohou nás zajímat sudá čísla, prvočísla, řetězce obsahující písmeno 'A', modrá auta atd. Nemůžeme proto podmínku vložit přímo do metody, ale potřebujeme ji do něčeho tzv. *zabalit*. Protože jazyk Java je objektový jazyk, nabízí se reprezentovat podmínku jako objekt. Bude se jednat o jednoduchý objekt mající právě jednu metodu, která nám vrátí `true`, pokud objekt splňuje danou podmínku, jinak vrátí `false`. Abychom s takovým objektem mohli pracovat, vytvoříme si rozhraní `Condition<T>`.

```
public interface Condition<T> {
  public boolean test(T value);
}
```

Nyní už můžeme metodu `filter` vytvořit.

```
public MyList<T> filter(Condition<T> condition) {
  MyList<T> result = new MyList<>(items.length);
  for (T item : items) {
    if (condition.test(item)) {
      result.add(item);
    }
  }
  return result;
}
```

V cyklu procházíme jednotlivé hodnoty a kažkou hodnotu `item` ze seznamu předáme objektu `condition`, aby otestoval, jestli objekt splňuje zadanou podmínku, či ne.

Při práci s metodou `filter` potřebujeme předat podmínku, která se použije pro filtrování. Jednou z možností je použít anonymní třídy:

```
MyList<Integer> numbers = new MyList<Integer>(10);
MyList<Integer> oddNumbers = numbers.filter(new Condition<Integer>() {
  public boolean test(Integer value) {
    return (value % 2) == 1;
  }
});
```

V tomto případě podmínka bude představovat test, jestli je zadané číslo liché, a daný kód vybere ze vstupního seznamu pouze lichá čísla.

Podobně můžeme chtít transformovat jeden seznam hodnot na jiný. Pro tento účel si vytvoříme rozhraní `Transform`:

```
public interface Transform<T, V> {
```

```
  public V apply(T value);
}
```

Objekt implementující toto rozhraní transformuje jednu hodnotu typu `T` na hodnotu typu `V`.

Metoda, která převede jeden seznam hodnot na druhý, by mohla vypadat následovně:

```
public <V> MyList<V> map(Transform<T, V> transformation) {
  MyList<V> result = new MyList<>(items.length);
  for (T item : items) {
    result.add(transformation.apply(item));
  }
  return result;
}
```

Všimněme si, že se jedná o generickou metodu, kde jednak pracujeme s typem `T`, který odpovídá hodnotám vstupního seznamu, a s typem `V`, který odpovídá typu hodnot výstupního seznamu.

Použití této metody ukazují následující dva příklady. V prvním příkladu všechny hodnoty v seznamu zvětšíme o jedna.

```
MyList<Integer> numbers = new MyList<Integer>(10);
MyList<Integer> incrementedNumbers = numbers.map(new Transform<Integer, Integer>() {
  public Integer apply(Integer value) {
    return value + 1;
  }
});
```

V druhém příkladu převedeme celá čísla na řetězce reprezentující čísla ve dvojkové soustavě.

```
MyList<String> binaryNumbers = numbers.map(new Transform<Integer, String>() {
  public String apply(Integer value) {
    return Integer.toBinaryString(value);
  }
});
```

Na závěr si ukážeme provedení nějaké operace s každým prvkem seznamu. Tuto operaci opět můžeme zabalit do objektu s jednou metodou, která provede nějakou operaci s právě jednou hodnotou ze seznamu. Vytvoříme si proto rozhraní `Action<T>`.

```
public interface Action<T> {
  public void perform(T value);
}
```

Metoda, která provede s každou hodnotou v seznamu zadanou operaci, je již přímočará.

```
public void forEach(Action<T> action) {
  for (T item : items) {
    action.perform(item);
  }
}
```

Stejně tak její použití kopíruje předchozí scénáře.

```
numbers.forEach(new Action<Integer>() {
  public void perform(Integer value) {
    System.out.println(value);
  }
});
```

Všimněme si společných rysů metod `filter`, `map` a `forEach`. Argumenty, které jsou těmto metodám předávány,

- jsou objekty implementující rozhraní, které má právě jednu metodu,
- jsou objekty jejichž smyslem je provedení nějaké operace (test, transformace, operace) a nepracují s vlastním stavem (nemají atributy),
- použití anonymních tříd je velmi nekomfortní, protože vyžaduje spoustu nadbytečného kódu.

Poznámka 3 *Čtenářům, kteří mají zkušenost s funkcionálním programováním, určitě neunikla podobnost s funkcemi vyššího řádu, které se používají ve funkcionálním programování.*

Aby bylo možné si práci s tímto typem objektů usnadnit a také přidat do Javy prvky funkcionálního programování, byl zaveden koncept tzv. *funkcionálního rozhraní*, což je rozhraní, které má právě jednu abstraktní metodu. Dále, na místech, kde se očekává objekt implementující funkcionální rozhraní, můžeme použít lambda-výraz.[1] Lambda-výraz můžeme zapsat několik způsoby:

```
argument -> tělo-vyrazu
(argument1, argument2, argument3) -> tělo-vyrazu
(Typ1 argument1, Typ2 argument2) -> tělo-vyrazu
```

Kde tělem výrazu může být buď běžný výraz, nebo blok kódu zakončený příkazem `return`, například:

```
foo -> foo * 2
(foo, bar, baz) -> foo * baz + baz
```

[1] Někdy se používá také označení *anonymní funkce*.

```
(int a, double b) -> {
   System.out.println(a);
   return a + b;
};
```

Všimněme si, že není nutné uvádět typy argumentů. Ty, pokud je to možné, jsou automaticky odvozeny překladačem z funkcionálního rozhraní, ke kterému se lambda-výraz vztahuje.

Můžeme tedy napsat:

```
Action<Integer> action = number -> System.out.println(number);
action.perform(20);
```

Protože objekt `action` má právě jednu abstraktní metodu `void perform(Integer)`, překladač je schopen odvodit, že typ argumentu `number` bude `Integer`.

S použitím lambda-výrazů můžeme operace se seznamem výrazně zestručnit.

```
MyList<Integer> oddNumbers         = numbers.filter(value -> (value % 2) == 1);
MyList<Integer> incrementedNumbers = numbers.map(value -> value + 1);
MyList<String>  binaryNumbers      = numbers.map(number -> Integer.toBinaryString(number));
numbers.forEach((Integer value) -> System.out.println(value));
```

Jako hodnoty, které implementují funkcionální rozhraní, nemusí být použity jen lambda-výrazy. Místo nich můžeme použít tzv. *odkazy na metody* (method references), které se zapisují s pomocí operátoru `::` (dvě dvojtečky):

```
Trida::statickaMetoda  // odkaz na statickou metodu
objekt::metoda         // odkaz na konkretni metodu daneho objektu
Trida::metoda          // odkaz na metodu dane tridy
Trida::new             // odkaz na konstruktor
```

Pochopitelně typy argumentů a návrátové hodnoty odkazované metody musí být shodné s typy uvedených u metody funkcionálního rozhraní.

Předchozí příklady by tedy šlo dále zjednodušit.

```
Action<Integer> action = System.out::println;
action.perform(20);
MyList<String> binaryNumbers = numbers.map(Integer::toBinaryString);
```

Komentář závěrem

Použití lambda-výrazů umožňuje kombinovat prvky objektově-orientovaného programovaní a funkcionálního programování, což může mít pozitivní dopad na kvalitu kódu. Jazyk Java je tomuto kombinování stylů nakloněn a má podporu pro tento styl programování i ve standardní knihovně. V balíčku `java.util.function`

je nachystaná řada nejběžněji používaných funkcionálních rozhraní (`Function`, `Consumer`, `Supplier` a další), které máme k dispozici a které se používají napříč standardní knihovnou Javy. Je na p.t. čtenářích, aby tyto rozhraní prošli v rámci samostudia a podle svých potřeb využili.