Immutable (neměnné) programování v Javě

Zjednodušeně řečeno, je immutable programování způsob programování, při kterém se nic (nebo téměř nic) nemění, ale neustále vzniká nové. Tento článek volně navazuje na článek „Immutable object (neměnné objekty)“.

Cíl

Hlavním cílem je čtenáři vysvětlit, jak se programuje immutable, a to hlavně programátorům, kteří o immutable programování neslyšeli, ale jsou z nějakého důvodu k němu tlačeni. Řekneme si, čím se vyznačují immutable třídy a metody, k čemu je to dobré.

Proč se takto programuje?

Immutable programování odstraňuje *side-efect* (vedlejší efekt). Ten nastává, pokud se něco v programu mění. To je problém hlavně u vláken, immutable třídy jsou thread-safe – žádné vlákno nemůže měnit nějaká data jinému vláknu.

Jak to funguje?

Nemění se vnitřní stav, což pro nás znamená žádné metody vracející void, každá metoda musí něco vracet. Pokud je potřeba něco změnit – nějaký atribut – vytvoříme novou instanci třídy, která bude totožná s předchozí, kromě změněného atributu. Příklad: pro třídu Person není metoda setAge(int age), jak se to nahrazuje, bude ukázáno dále. Kromě toho se nemění ani reference. Správná immutable třída má u každé proměnné final. Aby projekt fungoval, zůstává jedna třída – obvykle ta, která vše řídí – mutable.

Pokud nemůžeme používat void metody a nefinal proměnné, nemůžeme používat setry, for cyklus, while cyklus, foreach cyklus a samozřejmě kolekce jako List<T>. Co použít místo toho?

Mohli bychom si sice vytvořit nějakou knihovnu s immutable seznamy a cykly, ale proč nepoužít existující? **JavaSlang** je knihovna, která nám poskytuje immutable objekty, které nahrazují mutable cykly a kolekce.

Setry

Místo setrů se používají tzv. withry.

```
//metoda nastavi atribut name
public Person withName(final String name){
      return new Person(name, this);
}
```

Kromě této metody je potřeba i konstruktor. Lze použít buď hlavní konstruktor, nebo – jako v tomto případě – se použije privátní konstruktor:

```
private Person(final Person that,final String name){
      this.name = name;
      //další atributy
}
```

List<T>

Místo mutable Listu je v JavaSlang k dispozici interface IndexedSeq<T>. Funguje podobně jako list.

```
private final IndexedSeq<Person> persons;
```

Práce s ním vypadá asi takto:

```
//prazdna sekvence
this.persons = Array.empty();
//naplneni sekvence prvky
this.persons = Array.of(
          new Person("Jan", "Novak"),
          new Person("Josef","Svoboda")
     );
```

Tento interface předepisuje podobné metody jako list (get(i), size(),…), metody pro práci s prvky fungují takto:

```
//pridani prvku
this.persons.append(new Person("Petr","Pavel"));
//smazani prvku
this.persons.remove(new Person("Petr","Pavel"));
//smazani prvku na pozici
this.persons.removeAt(0);
```

Pamatujte si, že tyto tři metody nemění persons. Tento seznam po provedení zůstane stejný. Tyto metody vrátí nový seznam s provedenou změnou.

```
IndexedSeq<Person> persons2 = persons.method...
```

For cyklus

Místo for cyklu nám JavaSlang poskytuje Stream. Pro něj není potřeba další konstruktor. Má návratovou hodnotu.

```
IndexedSeq<Integer> numbers =
          Stream
               //vygeneruje posloupnost od 0 do 9
                .range(0, 10)
               //postupne projde radu vygenerovanych cisel
                .foldLeft(
                         //vyhozi hodnota je prazdne pole
                         Array.empty(),
                    //current - aktualne zpracovana hodnota
                    // i - index zpracovavaneho prvku
                    // i - totozne s i v for cyklu
                                   (current, i)->{

     //tady muze byt nejaky kod - podminky, dalsi stream....
                              //pokud jsme na konci streamu
                              // tak se toto vrati
                              //jinak je to current
                                   return current.append(i);
```

```
} );
```

Aby to bylo srozumitelnější, zkusím to ukázat na dalším konkrétnějším příkladu.

```
String outPrint =
Stream.range(0, persons.size()).foldLeft("Vypis osob: ",
            (current, i)->{
                return current + "\n" + persons.get(i).toString();
           });
```

Pokud byste chtěli procházek Stream po větších krocích než po jedné, má metoda range přetížení: range(od, do, krok). Stream má také metodu rangeClosed(od, doVčetně).

While

Pro while cyklus se mi nepodařilo najít náhradu v JavaSlang. Sám jej příliš nepoužívám, většinou místo něj stačí for cyklus potažmo Stream. Ukážeme si příklad pro prohledání sekvence čísel, kde chceme zjistit, zda obsahuje nějaké sudé číslo (omlouvám se za takový primitivní příklad, nic inteligentnějšího mě nenapadlo):

```
//sekvence
IndexedSeq<Double> numbers = Array.of(1.0,4.0,5.0, 7.8,9.7);

boolean result =
Stream.range(0, numbers.size()).foldLeft(false,
            (currentValue, i)->{
           //pokud uz byl patricny prvek nalezen
           // neni potreba kontrolovat dalsi
                if(currentValue==true)
                     return currentValue;
                if(numbers.get(i)%2==0)
                     return true;//hodnota se nepridava, nahrazuje
                return false;
     } );
```

Bez ohledu na index rozhodujícího prvku, má tento algoritmus složitost *n*. Proto bych ho nahradil rekurzí.

```
//spousteci metoda
public boolean isEvenThere(final IndexedSeq<Integer> num){
     return isEvenThere(num, 0);
}

//zde probiha rekurse
private boolean isEvenThere(final IndexedSeq<Integer> num,
                            final int i){
     if(i>= num.size())
                return false;
     if(num.get(i)%2==0)
          return true;
     return isEvenThere(num, i+1);
}
```

Při tomto postupu je PRŮMĚRNÁ složitost n/2.

Tuple

Velkou výhodou JavaSlangu je Tuple. Interface Tuple a několik jeho implementací jako Tuple2, Tuple3 a další, slouží k práci s více hodnotami jako s jednou. Kdy se to může hodit? Představme si toto: nějaká třída potřebuje seznam manželských párů. Zároveň potřebuje přístup k jednotlivým lidem. Začátečník by definoval dva listy – jeden pro všechny manžely, druhý pro manželky. Někdo jiný by vytvořil list dvouprvkových listů. Poslední by vytvořil třídu pro manželský pár a ty by pak tvořila list.
Právě poslední případ popisuje Tuple2.

```
Tuple2<T1, T2>
```

Jak je vidět, tato třída je generická, může obsahovat libovolné prvky. Současně nemusí prvky být stejného datového typu.

Práce s immutable třídou

Nakonec bych rád ukázal, jak se s immutable třídou pracuje. Kód je z třídy, která je mutable.

```
//narodil se jan novak a je mu 0 let
Person p = new Person("Jan", "Novák", 0);
//ten se rozhodl prejmenovat
p = p.withName("Petr");
p = p.withForName("Lebeda");
//nyni uz petr zestarl o rok
p = p.withAge(p.getAge()+1);
```

Do mutable proměnné se ukládají nové a nové instance Person, zatímco ty staré ničí Cargabe Collector.

Programovat immutable nebo ne?

Nebudu zde prosazovat jeden nebo druhý způsob, musíte se rozhodnout sami. Já sám se snažím immutable programovat, s tím, že jsou třídy, které takové být nemohou – například třídy, které ukládají dokument, pracují s databází a potomci JFrame.