Semana 6

Iteradores

André Souto

Unidade Curricular de
Laboratório de Programação

2017/2018

Resumo

Este é o tutorial para as aulas de LabP 2017/1018 sobre iteradores. O guião aqui descrito é uma actualização (e cópia) de material já existente de outros anos e todos os créditos da originalidade do trabalho são devidos aos seus autores. Este documento apenas serve para suporte às aulas de LabP. De notar que apenas se foca o essencial sendo que o estudo deve ser complementado com as aulas de AED.

Iteradores

Uma das ferramentas mais úteis em programação é a possibilidade de percorrermos as estruturas com que trabalhamos de modo a obtermos o conteúdo dessa estrutura por uma dada ordem, isto é, é útil podermos iterar (de alguma forma) de modo a obtermos todos os elementos de um objecto de um tipo de dados complexo. A criação de iteradores e a forma de iterar sobre estruturas de dados é matéria de estudo aprofundado da disciplina de AED.

É comum que as classes JAVA que suportam estruturas de dados tenham associadas a capacidade de definir iteradores. Um iterador é um objecto que implementa os serviços de `Iterator`, i.e., a capacidade de percorrer a informação guardada no respectivo objecto que itera.

Cada iterador possui duas operações base:

1. `hasNext ( )` que verifica se ainda há elementos por percorrer e que portanto o iterador ainda pode percorrer mais elementos;
2. `next ( )` que devolve o próximo elemento do objecto iterado;

Utilização de Iteradores

As classes que implementam a interface `Iterable` são capazes de produzir iteradores da sua própria estrutura, acessíveis através do método `iterator( )`.

Um exemplo de uma dessas classes é a classe `LinkedList`:

```
LinkedList<Integer> list = new LinkedList<Integer>();

//adicionar tres elementos a list
list.addFirst(1);
list.addFirst(2);
list.addFirst(3);

//definir o iterador para o objecto list
Iterator<Integer> it = list.iterator();

//usar o iterador obtido para percorrer todos os
elementos de list
while (it.hasNext()){
    System.out.println(it.next());
}
```

Neste exemplo obtemos o seguinte resultado na consola:

```
3
2
1
```

Existe uma alternativa do uso de iteradores. É feita através de ciclos **for-each** cuja sintaxe é mais simples. Tomando ainda o exemplo anterior, o ciclo `while` é substituído por:

```
for (int elem[1] : list) {
    System.out.println(elem);
}
```

A classe `LinkedList`, como tem uma implementação de uma lista duplamente ligada, fornece um iterador extra, `descendingIterator( )`, que percorre os elementos do fim para o princípio do objecto que está a iterar:

```
Iterator<Integer> itr = list.descendingIterator();

while (itr.hasNext()){
      System.out.println(itr.next());
}
```

sendo que neste caso o resultado impresso na consola é:

```
1
2
3
```

Iteradores sobre dados do tipo `enum`

Os tipos de dados `enum` têm sempre associado um iterador. Este pode ser obtido através do método `values( )`:

```
public static enum Dias {Seg,Ter,Qua,Qui,Sex,Sab, Dom};

...

for (Dias d : Dias.values()) {
     System.out.println(d);
}
```

Neste exemplo o resultado na consola é:

```
Seg
Ter
Qua
Qui
Sex
Sab
Dom
```

[1] Note que `elem` é do tipo `int`, portanto houve um unboxing automático de `Integer` para `int`.

Implementação de Iteradores

É igualmente possível criar iteradores para as nossas classes. Recorde o assunto desenvolvido pormenorizadamente nas aulas de AED. Considere uma classe `Pacote` semelhante à que terá de implementar no exercício. Para esta classe, para que possamos percorrer os seus elementos, é necessário criar um método que devolva um objecto da classe `Iterator` e declarar que a classe implementa a interface `Iterable`:

```
public class Pacote<E> implements Iterable<E>{
      // ... resto da classe

      @Override
      //metodo publico que devolve um iterador para
      //objectos do tipo Pacote
      public Iterator<E> iterator () {
            return new PacoteIterador();
      }

      // ... resto da classe
}
```

Note que a classe `Pacote` tem de declarar que implementa a interface `Iterable`. É esta declaração que indica que a respectiva classe é capaz de produzir iteradores sobre a sua própria estrutura. O que o método iterator faz é retornar um objecto da classe `PacoteIterador` que serve para definir instâncias de iteradores sobre os objectos da classe `Pacote`. Por essa razão, esta classe tem de ser definida dentro da classe `Pacote` como uma classe privada. Note que esta classe por sua vez tem de implementar o interface `Iterator` com os métodos associados `hasNext` e `next`. No que se segue esquematiza-se a estrutura dessa classe interna adaptada a Java 8:

```
public class Pacote<E> implements Iterable<E>{
      // ... resto da classe

      @Override
      //metodo publico que devolve um iterador para
      //objectos do tipo Pacote
      public Iterator<E> iterator () {
        return new PacoteIterador();
      }

      private class PacoteIterador implements Iterator<E>{

        private PacoteIterator() {
            // codigo do construtor
            // so pode ser chamado pela classe Pacote
        }

        @Override
        public boolean hasNext () {
           // codigo do metodo hasNext()
        }
```

```
        @Override
        public E next () {
           // codigo do metodo hasNext()
        }
}
```

É importante frisar que cada objecto da classe `PacoteIterator` tem um estado próprio que indica em que posição se encontra no processo de iteração (por isso é necessário ter atributos privados que armazenam o estado actual do objecto iterador). Assim, é possível criar dois ou mais iteradores sobre a estrutura de dados, cada um independente dos restantes dado que são vistos como objectos diferentes.

Exemplo

Desde a introdução das classes genéricas que os iteradores devem preferencialmente indicar que tipo de informação iteram (seja o tipo `T`). Isso é feito ao concretizar as interfaces genéricas `Iterable<T>` e `Iterator<T>`.
No exemplo que se segue, vamos definir uma classe `Places` que guarda um conjunto de locais (indexados por um inteiro) que podem estar ocupados ou disponíveis. Esta classe vai ter um iterador que nos devolve os índices dos locais ocupados, i.e., um iterador sobre inteiros. Eis o início da classe:

```
import java.util.Iterator;

public class Places implements Iterable<Integer> {

    public enum POSITIONS {AVAILABLE, OCCUPY};

    private POSITIONS[] elems;

    public Places(int size) {
        elems = new POSITIONS[size];
    }

    public void occupy(int place) {
        elems[place] = Places.POSITIONS.OCCUPY;
    }

    public void available(int place) {
        elems[place] = Places.POSITIONS.AVAILABLE;
    }

    // constroi iterador que percorre todas as posicoes ocupadas
    public Iterator<Integer> iterator() {
        return new PlaceIterator();
    }
    // falta definir a classe PlaceIterator
}
```

Agora é necessário definir a classe capaz de criar objectos iteradores. A ideia é utilizar um atributo que atravesse o vector `elems[]` à procura dos valores ocupados. Consideramos neste exemplo que o `hasNext( )` procura o próximo índice ocupado devolvendo true se o encontra e colocando-se imediatamente antes dessa posição. O método `next( )` limita-se a invocar `hasNext( )` e avançar para essa posição (devolvendo -1 se já não houver índices). O código completo deste exemplo está disponível em Places.java disponibilizado com este enunciado.

```
private class PlaceIterator implements Iterator<Integer> {

        private int index;

        private PlaceIterator() {
           index = -1;
        }

        public Integer next(){
           if(hasNext()){
                     index ++;
                     return index;
           } else{
                     return -1;
           }
        }

        public boolean hasNext() {
           boolean found = false;
            // verifica se jah chegou ao final do vector
           while(!found && index != elems.length-1) {
                // verifica se eh uma posicao ocupada
                     if (elems[index + 1] == Places.POSITIONS.OCCUPY){
                     found = true;
                     } else{
                         //avanca o index ate encontrar posicao
ocupada.
                         index++;
                     }
           }
           return found;
        }
}
```