Tutorial 4 – Parte B

Iteradores

André Souto

Unidade Curricular de
Laboratórios de Programação

2020/2021

Resumo

Este é o tutorial para as aulas de LabP 2020/2021 sobre iteradores.

O guião aqui descrito é uma atualização (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 tarefa muito importante em programação é a de percorrer os elementos de uma estrutura de dados por uma dada ordem, isto é, é *iterar* sobre a estrutura de dados.

A criação de *iteradores* e a forma de iterar sobre estruturas de dados é matéria de estudo aprofundado na disciplina de AED.

É comum as classes JAVA que representam estruturas de dados terem associada a capacidade de definir iteradores.

A *interface* `Iterator` do pacote `java.util` define a capacidade de percorrer a informação guardada no objeto. Um iterador é um objeto que implementa os serviços definidos na *interface* `Iterator`.

Cada iterador possui duas operações base:

1. `hasNext( )` que verifica se ainda há elementos sobre os quais iterar;
2. `next( )` que devolve o próximo elemento do objeto iterado;

Utilização de Iteradores

A interface `Iterable` do pacote `java.lang` define a capacidade de ser iterado. 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.add(1);
list.add(2);
list.add(3);

//definir o iterador para o objeto 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:

```
1
2
3
```

Existe uma alternativa para iterar sobre uma estrutura sem usar o iterador diretamente. É através de ciclos **for-each** cuja sintaxe é mais simples. Tomando ainda o exemplo anterior, o ciclo `while` pode ser substituído por:

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

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

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

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

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

sendo que neste caso o resultado impresso na consola é:

```
3
2
1
```

Iteradores sobre dados do tipo `enum`

Os tipos de dados enumerados (`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
```

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` cujas instâncias são compostas de vários elementos. Para que possamos percorrer esses elementos, é necessário criar um método que devolva um objeto da classe `Iterator` e declarar que a classe `Pacote` implementa a interface `Iterable`:

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

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

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

Note que a classe `Pacote` tem que declarar que implementa a interface `Iterable`. É esta declaração que indica que a classe é capaz de produzir iteradores sobre a sua própria estrutura.

```
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 next()
       }
}
```

O que o método `iterator` faz é retornar um objeto da classe `PacoteIterador` que serve para definir instâncias de iteradores sobre os objetos 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 que implementar o interface `Iterator` com os métodos associados `hasNext` e `next`.

É importante frisar que cada objeto 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 atual do objeto iterador). Assim, é possível criar dois ou mais iteradores sobre a estrutura de dados, cada um independente dos restantes, dado que são vistos como objetos diferentes.

Exemplo

Desde a introdução das classes genéricas que os iteradores devem preferencialmente indicar que tipo de informação iteram (através da variável de 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 objetos iteradores. A ideia é utilizar um atributo que atravesse o vetor `elems[]` à procura dos valores ocupados. Consideramos neste exemplo que

o `next( )` devolve o próximo índice ocupado e vai posicionar o índice na próxima posição ocupada, caso exista. O método `hasNext( )` limita-se a verificar se já se atingiu a última posição do vetor.

```
/**
 * Classe que representa um iterador de lugares ocupados
 */
private class PlaceIterator implements Iterator<Integer> {

	private int index;

	// Construtor privado
	private PlaceIterator() {
		index = -1;
	}

	/**
	 * Qual o proximo lugar ocupado?
	 * @return O proximo lugar ocupado; se não existir retorna null
	 */
	private Integer nextOccupy() {
		Integer next = index + 1;
		boolean found = false;
		while (!found && next < elems.length) {
			// verifica se é uma posição ocupada
			if (elems[next] == Places.POSITIONS.OCCUPY) {
				found = true;
			} else {
				// avança o index até encontrar posição ocupada.
				next++;
			}
		}
		if (!found) {
			next = null;
		}
		return next;
	}

	/**
	 * Retorna o proximo lugar ocupado, se existir, e atualiza o
	 * estado do iterador
	 * @throws NoSuchElementException
	 *             se nao existir mais nenhum lugar ocupado
	 */
	public Integer next() {
		Integer next = nextOccupy();
		if (next != null) {
			index = next;
			return next;
		} else {
			throw new NoSuchElementException();
		}

	}

	/**
	 * Existe mais algum lugar ocupado?
	 */
	public boolean hasNext() {
		return nextOccupy() != null;
	}
}
```