Módulo 5 - Iteradores e protocolos de interface

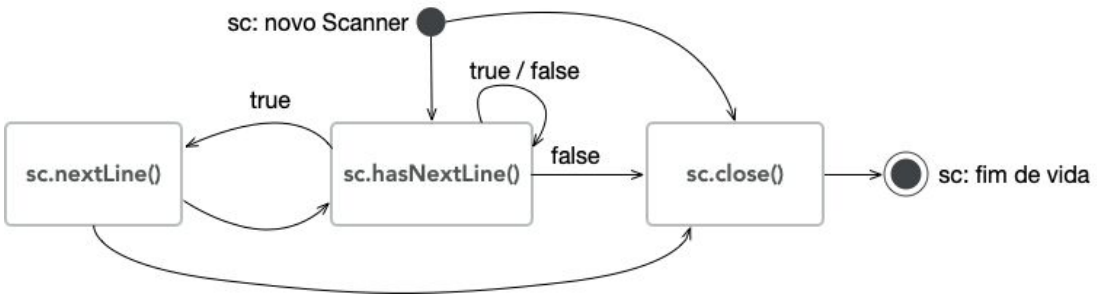
Objetivos

- Entender a importância de garantir a compatibilidade comportamental na utilização de classes e implementação de interfaces.
- Entender e saber utilizar a abstração de iterador.

Motivação: garantir compatibilidade comportamental

Existem objetos onde a execução das suas operações corresponde a eventos independentes que não afetam o seu correto funcionamento. Desta forma, quaisquer sequências de invocações neste tipo de objetos nunca irá causar um erro (desde que se utilizem argumentos válidos), por exemplo ao manipular uma lista. O caso dos objetos imutáveis é outro exemplo, pois dado que o estado do objeto nunca muda, as operações terão sempre o mesmo comportamento. Um caso particular de objetos imutáveis é o caso dos objetos sem estado (i.e. sem atributos), como por exemplo as implementações típicas de comparadores (java.util.Comparator).

Porém, algumas classes de objetos mutáveis têm uma lógica de funcionamento associada onde não é irrelevante a ordem pela qual são invocadas as operações. Nestes casos, a classe foi desenhada assumindo um *protocolo* de utilização dos seus objetos. Se não for respeitado o protocolo ao utilizar os objetos deste tipo de classes irão ocorrer erros (exceções). Um exemplo já utilizado anteriormente trata-se da classe Scanner para leitura de um ficheiro linha a linha. O protocolo implica que a operação nextLine() não pode ser invocada a qualquer momento, pois só pode ser invocada quando hasNextLine() retorna verdadeiro. Por outro lado, deverá ser invocado close() ao terminar a utilização. Não respeitar estas regras (protocolo) irá resultar numa incorreta utilização do objeto. O seguinte diagrama de estados ilustra com mais detalhe e precisão o protocolo de utilização de objetos Scanner, com foco apenas no caso de leitura linha a linha (o caso de palavra a palavra com as operações next() e hasNext() é análogo).



A importância de respeitar o protocolo aplica-se igualmente ao caso de estarmos a manipular uma referência polimórfica (através de uma interface). Porém, neste caso temos outra questão, relativa ao facto de haver várias implementações da interface (classes compatíveis). O processo de compilação apenas assegura *compatibilidade estrutural* da classe que se declara compatível, isto é, que a classe tem os elementos necessários para a estrutura "encaixar" (os métodos batem certo). Porém, não é garantida *compatibilidade comportamental*, isto é, que os métodos se comportam como esperado. Caso exista um protocolo associado à utilização da interface, as classes compatíveis terão que o ter em conta na sua implementação. Caso contrário, a execução do código que se baseia na interface não irá funcionar corretamente.

Padrão Iterador (*Iterator*)

O padrão de desenho Iterador consiste em possibilitar a iteração sobre uma coleção de elementos de determinado tipo, abstraindo a forma de obtenção e proveniência desses elementos. Por exemplo, podemos desenvolver uma função que itera sobre um conjunto de números para fazer cálculos (p.e. somatório, média) sem saber como são fornecidos os números. Estes poderão ser provenientes de um vetor, de uma lista (podendo esta ser de vários tipos), de um ficheiro, ou simplesmente calculados através de uma série (os números vão sendo calculados sucessivamente). Através de iteradores é possível definir apenas uma função para este objetivo.

Dada a utilidade generalizada deste padrão, quase todas as linguagens de programação modernas incorporam e integram-no com as bibliotecas e sintaxe da linguagem. Tipicamente, existe um tipo de objeto que representa um iterador que tem o papel de gerir o "varrimento" pelos elementos da fonte de dados em questão. A lógica deste tipo de objeto baseia-se em ter um objeto (iterador) que após a sua criação está “posicionado” no primeiro elemento da iteração, fornecendo duas operações relacionadas para:

- saber se existem mais elementos para processar (booleana); caso a iteração não seja vazia, esta operação devolverá verdadeiro após o objeto ter sido criado;

- obter o elemento atual, sendo que o iterador avança para o elemento seguinte (nalgumas linguagens esta operação divide-se em duas independentes: obter / avançar)

A lógica da classe Scanner aplica no fundo a ideia de um iterador de Strings (palavras ou linhas), podendo estas ser provenientes de várias fontes diferentes (outras Strings, teclado, ficheiros de texto).

Interface java.util.Iterator

As bibliotecas do Java fazem uma utilização extensiva da interface java.util.Iterator para representar um iterador de elementos de uma coleção. A interface define duas operações correspondentes às descritas na secção anterior. Adicionalmente, define uma operação para integrar a remoção com o processo de interação, permitindo que sejam removidos elementos durante o processo de interação. Esta operação de remoção é opcional, significando que poderá não estar disponível nalgumas coleções (causando um erro/exceção), e logo a sua utilização deverá ser feita com cautela. Por exemplo, numa coleção de elementos imutável esta operação não está disponível (não é autorizada), pois a coleção pretende-se impossível de alterar. De seguida é apresentada a definição da interface, junto com excertos da sua documentação oficial. A documentação neste caso é essencial, pois é nela que está descrito o protocolo de utilização do iterador.

```
/** <E> the type of elements */
public interface Iterator<E> {
  /**
   Returns true if the iteration has
   more elements.(In other words, returns
   true if next() would return an element
   rather than throwing an exception.) */
  boolean hasNext();
```

Se for devolvido verdadeiro, significa que ainda há pelo menos um elemento por processar, e que é seguro invocar next() para obter esse elemento.

```
/**
  Returns the next element in the iteration.
  @throws NoSuchElementException if the
  iteration has no more elements */
  E next();
```

Obtém o elemento onde o iterador está posicionado, sendo que o estado do iterador avança para o elemento seguinte. É lançada uma exceção caso não hajam elementos por processar (hasNext() → falso).

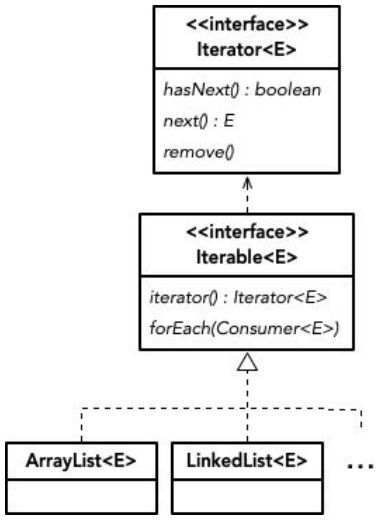
```
 /**
  Removes from the underlying collection the
  last element returned by this iterator
  (optional operation). This method can be
  called only once per call to next(). */
  void remove();
}
```

Remove o último elemento obtido através de next(). Só pode ser invocada uma vez por invocação de next(). Operação opcional: pode não estar disponível em todos os iteradores.

Interface java.util.Iterable

Associada à interface java.util.Iterator, as bibliotecas do Java definem a interface java.util.Iterable para representar objetos iteráveis, ou seja, objetos sobre os quais podemos obter um iterador para os mesmos.

```
public interface Iterable<E> {
  Iterator<E> iterator();
}
```



Todos os tipos de lista e outras coleções implementam esta interface, e logo, a partir dos seus objetos podemos obter um iterador. O seguinte exemplo ilustra a obtenção e utilização de um iterador para um ArrayList (outra coleção seria análogo).

```
ArrayList<String> list = ...
Iterator<String> it = list.iterator();
while(it.hasNext()) {
  String s = it.next();
  ...
}
```

Dada a frequente necessidade de iteração, a interface java.util.Iterable está integrada com a sintaxe da linguagem através do ciclo *for-each* (o qual pode também ser utilizado com um vetor). Desta forma, qualquer objeto cuja classe implementa Iterable pode ser utilizado neste ciclo, abreviando a sintaxe acima, embora o que executará de facto será equivalente.

```
Iterable<String> it = ... // ArrayList, LinkedList, etc: {"a", "b", "c"}
for(String s : it)
  System.out.println(s);

> a
> b
> c
```

Interface java.util.function.Consumer

As bibliotecas do Java definem várias interfaces funcionais que são utilizadas em operações de manipulação de coleções. Uma dessas interfaces é a java.util.function.Consumer para representar um consumidor de objetos de determinado tipo, i.e. uma operação que utiliza o objeto em questão.

```
public interface Consumer<E> {
  void accept(E object);
}
```

Podemos criar um objeto compatível com esta interface através de uma expressão lambda:

```
Consumer<Object> printer = obj -> System.out.println(obj);
```

Todos os objetos iteráveis têm uma operação forEach(Consumer) que efetua a iteração, e para cada elemento *e* invoca a operação accept(*e*). O exemplo seguinte utiliza esta operação para imprimir os elementos de uma lista (equivalente ao exemplo acima).

```
Iterable<String> it = ...     // {"a", "b", "c"}
it.forEach(printer);
```

Interfaces e protocolos

As interfaces são muitas vezes utilizadas para propósitos de permitir a adaptação de uma classe por terceiros, isto é, código cliente da classe que não é desenvolvido pelos autores da mesma. Os algoritmos de ordenação das bibliotecas do Java são exemplo disso, onde a adaptação é permitida mediante a interface java.util.Comparator. Porém, este caso é simples e não tem nenhum protocolo associado, pois apenas existe uma função compare(...), a qual

devolve um inteiro, sendo qualquer valor desse inteiro válido (negativo, zero, ou positivo). Desta forma, uma implementação da operação compare(...) que compile irá ser válida em termos comportamentais porque é impossível devolver um valor inválido.

No entanto, há casos em que ao desenvolvermos uma classe compatível com uma interface, esta poderá ter um protocolo de utilização associado, o qual vai ser assumido pelos módulos que utilizam referências polimórficas do tipo da interface. A interface java.util.Iterator é um desses casos. Ao implementar uma classe compatível com esta interface teremos que obedecer ao protocolo.

Exercício 5.1 - Protocolo de utilização de iterador

Desenhe um diagrama de estados, semelhante ao apresentado anteriormente para a classe Scanner, para descrever o protocolo da interface java.util.Iterator com base na sua documentação (descrita anteriormente). Pode ignorar a operação remove() numa primeira fase.

Exercício 5.2 - Cálculo de média de valores provenientes de um iterável

Desenvolva uma função para calcular a média de um conjunto de valores inteiros provenientes de um objeto iterável. Teste a função utilizando como argumentos objetos ArrayList e LinkedList.

```
public static double average(Iterable<Integer> iterable) { ...
```

Exercício 5.3 - Classe para representar intervalo de inteiros iterável

a) Desenvolva uma classe Interval para representar intervalos de inteiros representados através de um valor mínimo e valor máximo. Os intervalos deverão ser imutáveis, e construídos fornecendo os valores mínimo e máximo. Deverá ser possível saber se o intervalo é vazio (i.e., não tem elementos) e quantos elementos tem o intervalo. Inclua métodos fábrica (estáticos) para a criação de um intervalo de números naturais até um dado máximo, intervalo de índices válidos de um vetor, e um intervalo vazio.

```
Interval test = new Interval(5, 10);                    // [5, 10]
Interval nat = Interval.naturals(10);                   // [1, 10]
Interval indexes = Interval.arrayIndexes(new int[5]);   // [0, 4]
Interval empty = Interval.empty();                      // vazio
```

b) Implemente um iterador de inteiros para o intervalo como classe interna.

```
public class Interval {
  private class IntervalIterator implements Iterator<Integer> { ...
```

c) Altere a classe Interval para implementar Iterable, fazendo com o que o método exigido devolva uma instância do iterador da alínea anterior. Experimente a iteração com um ciclo for-each.

```
Interval test = new Interval(5, 7);
for(Integer i : test)
   System.out.println(i);

> 5
> 6
> 7
```

d) Experimente invocar a função do Exercício 5.2 com objetos Interval.

Remoção durante a iteração

Se durante a utilização de um iterador de uma coleção (utilizando for-each ou não) a mesma for alterada, irá ser lançada uma exceção. O motivo deve-se ao funcionamento interno dos iteradores, e para evitar que o seu comportamento seja inesperado nestas situações, as bibliotecas proíbem a remoção de elementos de uma coleção durante a iteração da mesma (lançando a exceção). No seguinte exemplo, a lista está a ser iterada com um for-each, e no caso de encontrar strings vazias, estas são removidas das lista.

```
List<String> list = ...
for(String s : list)
  if(s.isEmpty())
    list.remove(s);
```

É lançada a exceção
java.util.ConcurrentModificationException

Para contornar este problema podemos iterar a lista utilizando o objeto iterador diretamente, e quando é identificado um elemento para remoção invocar a operação remove() do iterador.

```
List<String> list = ...
Iterator<String> it = list.iterator();
while(it.hasNext()) {
  String s = it.next();
  if(s.isEmpty())
    it.remove();
}
```

Interface java.util.function.Predicate

Outra interface das bibliotecas do Java para assistir à manipulação de coleções é java.util.function.Predicate e representa predicados para um objeto de determinado tipo.

```
public interface Predicate<T> {
  boolean test(T object);
}
```

Podemos criar um objeto compatível com esta interface através de uma expressão lambda:

```
Predicate<String> emptyString = s -> s.isEmpty();
```

Uma das utilizações desta interface alivia o problema da verbosidade da solução para a remoção durante a iteração apresentada anteriormente. As coleções das bibliotecas do Java fornecem a operação removeIf(Predicate) que remove *todos* os elementos *e* da coleção para os quais o predicado devolva verdadeiro (resultante da invocação de test(*e*)).

```
List<String> list = ...          // {"a", "", "b", "", "c"}
list.removeIf(emptyString);      // {"a", "b", "c"}
```

Exercício 5.4 - Filtro de Strings (*select*)

Desenvolva uma função que dado um objeto iterável e um predicado, devolve uma lista contendo todos os elementos para os quais o predicado se verifica.

```
static Iterable<String> select(Iterable<String> it, Predicate<String> pred) { ...

List<String> list = ...                                          // {"a","","b",""}
Iterable<String> nonEmpty = select(list, s -> !s.isEmpty());     // {"a","b"}
```

Exercício 5.5 - Filtro genérico (*select*)

Desenvolva uma versão da função anterior, mas genérica em termos do tipo dos elementos.

```
static <T> Iterable<T> select(Iterable<T> it, Predicate<T> pred) { ...
```

Exercício 5.6 - Classe genérica para iteradores de vetores de objetos

Desenvolva uma classe genérica para iteradores de vetores de objetos, cujos objetos possam ser criados fornecendo um vetor de qualquer tipo de referência.

```
public class ArrayIterator<T> implements Iterator<T> {
  public ArrayIterator(T[] array) { ...
```