Tutorial 4 – Parte A

Tipos Genéricos de Dados

André Souto

Unidade Curricular de
Laboratórios de Programação

2020/2021

Resumo

Este é o tutorial para as aulas de LabP 2020/2021 sobre tipos de dados genéricos.

O guião aqui descrito é uma adaptação (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.

Classes Genéricas e Estruturas de Dados

O Java dispõe na sua API de diversas classes que providenciam serviços típicos de várias estruturas de dados. Exemplos incluem classes que implementam vetores, pilhas, filas de espera ou listas.

Desde a sua versão 5, a linguagem Java suporta classes genéricas. A ideia por detrás de uma classe genérica é permitir abstrair das estruturas de dados referidas o tipo de informação que irão armazenar/manipular.

Esta característica é útil pois, por exemplo, evita a necessidade de repetir a escrita de código semelhante (senão mesmo igual!) se tivermos de usar uma pilha de inteiros e uma pilha de *strings*.

Com este propósito, introduziu-se em Java a possibilidade de as classes serem parametrizadas por um tipo **`T`** que define qual o tipo de informação que os serviços da classe irão usar.

Tomando como exemplo a estrutura de dados de lista, a API do Java disponibiliza uma classe denominada **`ArrayList<T>`**, que pode ser instanciada como `ArrayList<Integer>` para armazenar inteiros ou `ArrayList<String>` para armazenar *strings* ou com qualquer outro tipo de dados.

Note que o parâmetro **`T`** pode ser um qualquer tipo de dados **não primitivo** de Java.

Para usar esta classe é necessário importar a seguinte classe do pacote `java.util` através da palavra reservada `import`: `import java.util.ArrayList;`

`ArrayList <T>`

A classe `ArrayList<T>` implementa uma lista cuja dimensão total é dinâmica (ao contrário dos *arrays* originais do Java que são sempre de dimensão fixa) permitindo o aumento ou diminuição de tamanho consoante a necessidade.

De forma semelhante aos `vetores`, sendo ela própria um vetor, a informação em cada uma das suas posições pode ser acedida através de um índice inteiro. Contudo, os objetos desta classe têm disponíveis outros métodos que um vetor tradicional não tem. Por exemplo, é possível inserir um novo elemento no fim de um objeto do tipo `ArrayList<T>`.

Sugere-se uma leitura da API para se familiarizar com esta classe. Em particular, observe os diferentes construtores disponíveis:

```
//constroi um ArrayList com capacidade inicial para 6 strings
ArrayList <String> sequencia = new ArrayList<String> (6);

//constroi um ArrayList sem indicacao da capacidade inicial
ArrayList <String> sequencia = new ArrayList<String> ( );
```

Note que a indicação da capacidade serve apenas para otimizar as operações sobre a lista, não para limitar o número de elementos que ela pode armazenar.

Chama-se a atenção para o facto de que um objeto acabado de criar está vazio (mesmo que se tenha indicado um valor para a capacidade no construtor). É importante lembrar este detalhe porque o comportamento não é intuitivo. Por exemplo, quando se cria um novo vetor de `Strings` com a

instrução `new String[5]` é reservado o espaço de memória para todos os índices do vetor e portanto, por exemplo, o seu tamanho é 5. Por seu turno, ao construirmos um `ArrayList<String>` com a instrução `new ArrayList<String>(6)`, o seu tamanho é 0.

(Alguns) Métodos da classe `ArrayList<T>`

Para adicionar elementos ao `ArrayList` pode-se usar o método `add`:

```
//acrescenta o elemento no fim do array (apos o ultimo elemento)
sequencia.add("banana");

//acrescenta um elemento na posicao indicada deslocando todos os elementos
//a partir da posicao j uma posicao i.e., para a posicao seguinte.
sequencia.add(j,"laranja");
```

Note que se a posição `j` não existir, é lançada uma exceção de tipo `java.lang.IndexOutOfBoundsException`. Note ainda que, à medida que se forem acrescentando mais elementos à estrutura, a dimensão do objeto vai aumentando em conformidade.

No contexto que se segue, assuma que `sequencia` é um objeto da classe `ArrayList<T>` para algum valor de `T`. Esta classe tem ainda muitos outros métodos dos quais se destacam os seguintes (para uma lista completa devem consultar a API):

- Interrogações:
    - `isEmpty()` - verifica se `sequencia` está vazia (i.e., devolve `true` se estiver vazio e `false` caso contrário).
    - `size()` - retorna o número de itens em `sequencia`.
    - `contains (Object o)` - verifica se o objeto `o` (que tem de ser de tipo `T`) é idêntico a algum objeto pertencente a `sequencia`.
    - `get(int index)` - devolve o item (do tipo `T`) em `sequencia` no índice `index` dado.
    - `toString()` - devolve uma descrição textual do conteúdo de `sequencia` (para representar cada um dos elementos usa o método `toString` da classe que instancia o tipo `T`).
- Comandos:
    - `add (T item)` - adiciona o novo elemento `item` ao fim de `sequencia`.
    - `add (int index, T item)` - adiciona o novo elemento `item` no índice `index` dado (há um *shift* da posição dos itens seguintes).
    - `remove (int index)` - remove o elemento no índice `index` dado (há um *shift* da posição de todos os elementos seguintes para a esquerda).
    - `set(T item, int index)` - substitui o elemento do índice `index` dado pelo novo elemento `item`.
    - `clear( )` - remove todos os itens de `sequencia`.

De notar que podemos usar um **`for-each`** para atravessar todos os itens de `sequencia`. Assim, tomando o exemplo em que instanciamos `T` com `String`, podemos imprimir todos os seu elementos através de:

```
for (String s: sequencia) {
    System.out.println(s);
}
```

Sobre o uso dos tipos primitivos no contexto das classes genéricas

Desde a versão 5 do Java que é possível guardar valores de tipos primitivos nestas classes. Pode eventualmente parecer estranho à partida uma vez que estas classes têm como parâmetro `T` um tipo de dados não primitivo – classe, enumerado ou *interface*. Torna-se então pertinente a questão: "*Dado que os valores dos tipos primitivos não são objetos, como é isto possível?*"

A resposta reside nos "embrulhos" de Java. Para cada tipo primitivo em Java existe uma classe capaz de **«embrulhar»** (do inglês, *wrap*) os valores do respetivo tipo primitivo. Por exemplo, para o tipo `int`, existe a classe `Integer`, para o tipo `double` existe a classe `Double`.

Cada vez que se pretende armazenar um valor de tipo primitivo numa classe genérica, o Java automaticamente "embrulha" esse valor num objeto da classe *wrapper* correspondente, permitindo cumprir a restrição sobre o tipo `T` que tem que ser não primitivo.

No que se segue, apresenta-se um exemplo ilustrativo deste conceito:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(1);
list.add(2);
```

é equivalente a:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(5));
list.add(new Integer(1));
list.add(new Integer(2));
```

Dado que a transformação é automática, podemos usar a primeira versão, que é mais sucinta. A este processo de transformação automática é costume dar-se o nome de *autoboxing*.

Note ainda que o contrário, o *auto-unboxing*, também acontece em Java, ou seja, também é automático o processo de desembrulhar o valor guardado num objeto da respetiva classe (que faz o *wrap* da classe primitiva) e colocá-lo numa variável de tipo primitivo.

No exemplo:

```
Integer obj = new Integer(10);
int i = obj;
```

o inteiro i toma o valor 10.