

# Linguagem Java

Continuação do estudo sobre coleções de dados

ArrayList e interface List

# Exercício

- Resolução do exercício 13 da ficha

**13.** Pretende-se uma aplicação para gerir os livros de uma biblioteca. Os livros são identificados por um código (um número inteiro positivo que representa a ordem de criação do registo dos livros na biblioteca). O registo de um livro, para além do referido código, tem obrigatoriamente informação sobre o título e os autores.

# Exercício

- Definir uma classe para representar livros
  - a. Defina a classe `Book` que representa este conceito de registo de um livro nesta biblioteca. Deve ser possível criar objectos da classe `Book`, dando informação sobre o título e autores, sendo, o código gerado automaticamente. As variáveis-membro devem ser privadas, podendo ser acedidas através de funções *get* e *set*. A variável-membro código não deve poder ser modificado a partir do exterior da classe. Nesta classe deve também implementar as funções:
    - i. `toString()` que retorna uma *string* com a descrição do objeto;
    - ii. `equals()` que representa o critério de identificação de um livro (dois livros são o mesmo se tiverem o mesmo código);
    - iii. `hashCode()` que retorna o *hash code* do objeto.
    - iv. `getDummyBook()` método estático que recebe o código de um livro e retorna uma instância da classe `Book` com o código em causa, mas sem título e sem autores (`null`).

# Método equals

- Um objeto que vá ser alvo de comparações com outros objetos similares deverá disponibilizar uma implementação do método equals, herdado a partir da classe Object
- O método equals recebe como parâmetro a referência do outro objeto com o qual se pretende realizar a comparação, retornando
  - true, se for considerado igual
  - false, se for considerado diferente

```
@Override  
public boolean equals(Object o) {  
    // ...  
    return <resultado_comparação>;  
}
```

- Ter em atenção que o parâmetro do método equals é do tipo Object, ou seja, pode referenciar qualquer tipo de objeto Java

# Método equals

- O método `equals` deverá verificar se a comparação está a ser realizada com o objeto adequado
  - Obrigar a que seja exatamente um objeto do mesmo tipo  
`this.getClass() == other.getClass()`
  - Permitir que o objeto possa ser uma instância de um tipo de objeto de determinada hierarquia (em princípio da mesma à qual o próprio objeto pertence)  
`other instanceof <classe_base_da_hierarquia>`
- Para além da verificação do tipo deverá depois ser verificado o conteúdo que se pretende realmente comparar
  - Necessário fazer a conversão (*cast*) para o tipo pretendido para aceder aos membros adequados

# Método hashCode

- Sempre que se redefine o método equals deve também ser redefinido o método hashCode
- O método hashCode deverá retornar um inteiro que represente a instância em causa
  - Pode corresponder a um identificador único atribuído ao objeto ou pode ser calculado tendo em conta uma ou mais características do objeto
    - Normalmente são consideradas as características usadas para a comparação no equals

```
@Override  
public int hashCode() {  
    return <valor_único_para_a_instância>;  
}
```

- Ter em consideração:
  - Dois objetos considerados iguais pelo método equals (true) devem possuir o mesmo hash code
  - Não é necessário que dois objetos considerados diferentes pelo equals (false) tenham hash codes diferentes

# Exercício

- Definir uma classe para representar a biblioteca

- b. Defina a classe `Library` que representa uma biblioteca que possui um conjunto de livros, geridos com o auxílio de um objecto `ArrayList`. Para além dos livros, a biblioteca tem um nome. Ao ser criado um objeto da classe `Library` deve ser dado o seu nome, ficando, à partida, sem registo de qualquer livro. Esta classe deve ter as seguintes funções:
- i. `addBook()` que recebe toda a informação que permite criar o registo de um livro, cria o registo e acrescenta-o à biblioteca. Deverá retornar o código do livro adicionado;
  - ii. `findBook()` que recebe o código de um livro e retorna uma referência para o livro se o encontrar ou `null` se não encontrar. 1ª versão: pesquisa iterativa do livro. 2ª versão: utilização do método `indexOf()` do `ArrayList`;
  - iii. `removeBook()` que recebe o código do livro, eliminando-o se o encontrar. Retorna o valor lógico do sucesso desta operação. 1ª versão: pesquisa iterativa do livro. 2ª versão: utilização do método `indexOf()` do `ArrayList`;
  - iv. `toString()` que retorna uma string com a descrição do objeto.

# interface

- Uma *interface* em Java é uma descrição de um protocolo que outras *interfaces* ou classes têm que respeitar
- Na sua utilização mais tradicional, as *interfaces* permitem "definir" um conjunto de métodos que uma ou mais classes terão que respeitar
- Exemplo de definição de uma *interface*

```
interface MyInterface {  
    void my_func();  
}
```



# interface

- Implementação de uma *interface*

```
class MyClass implements MyInterface {  
    @Override  
    public void my_func() {  
        // ...  
    }  
}
```

- Nota: uma instância de `MyClass` é também instância (`instanceof`) de `MyInterface`

# ArrayList<E> e List<E>

- Como já foi referido o tipo de objetos ArrayList<E> é apenas um dos muitos tipos de coleções de dados suportados em Java
- Mesmo na gestão de coleções que seguem o formato de uma lista existem diferentes implementações
  - ArrayList, LinkedList, Vector, Stack, ...
- Todas estas implementações respeitam um protocolo comum, ou seja, possui um conjunto de métodos idêntico, oferecendo funcionalidades idênticas, mesmo que a gestão interna seja otimizada para diferentes fins
- Esta compatibilização existe porque todos eles implementam uma mesma *interface*, no caso concreto, a List<E>
- A *interface* List<E> define os métodos que as classes que a implementam devem respeitar, tais como:
  - add, addAll, clear, contains, indexOf, isEmpty, remove, removeAll, sort, size, toArray, ...

# ArrayList<E> e List<E>

- A referência para uma instância de ArrayList<E> pode ser guardada e eventualmente retornada/passada entre funções através de variáveis do tipo da *interface* List<E>
- Dessa forma poderá ser ofuscada a forma de implementação interna de uma classe, sem perder funcionalidades e flexibilizando uma eventual alteração da forma de implementação
  - Por exemplo, passando de ArrayList<E> para LinkedList<E> ou vice-versa

# ArrayList<E> e List<E>

```
List<String> list1 = new ArrayList<>();  
list1.add("DEIS-ISEC");
```

```
List<Object> list2 = new ArrayList<>();  
list2.add("Prog. Avançada");  
list2.add(1234);
```

```
for( int i = 0 ; i < list2.size(); i++){  
    // ... list2.get(i)  
}
```

```
for(String s: list1){  
    // ... s;  
}
```