

Linguagem Java

Continuação do estudo sobre coleções de dados

Set e HashSet

Map e HashMap

Set<E>

- Um conjunto, `Set`, permite o armazenamento e gestão de uma série de elementos não repetidos
 - Dois itens, `i1` e `i2`, são considerados repetidos quando a comparação dos dois através do método `equals`, `i1.equals(i2)`, retorna o valor `true`
 - É imprescindível que os métodos `hashCode` estejam devidamente implementados para que as operações sobre um `Set` conduzam aos resultados esperados
 - Caso a indexação dos elementos não seja feita corretamente, as procuras de elementos iguais não funcionará e, consequentemente, o mecanismo que garante a não existência de elementos repetidos também não funcionará

Set<E> e HashSet<E>

- A interface Set<E> define o protocolo associado a este tipo de coleção
 - add, addAll, clear, contains, containsAll, isEmpty, iterator, remove, removeAll, size, toArray, ...
- Não é garantida a manutenção da ordem de inserção dos elementos de um Set
- O acesso aos elementos é realizado através de um processo iterativo sobre os elementos
- Existem várias implementações desta interface sendo uma das mais usadas a HashSet<E>
 - Outros: EnumSet, TreeSet, LinkedHashSet, ...

HashSet e hashCode

```
class Item {
    private static int count = 0;
    int i;

    public Item(int i) { this.i = i; }

    @Override
    public int hashCode() { return i;  }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Item other = (Item) obj;
        return (i == other.i);
    }
}

public class App {
    public static void main(String[] args) {
        HashSet<Item> set = new HashSet<>();
        System.out.println(set.add(new Item(1))); // true
        System.out.println(set.add(new Item(2))); // true
        System.out.println(set.add(new Item(1))); // false
        System.out.println(set.size()); // 2
    }
}
```

```
class Item {
    private static int count = 0;
    int i;

    public Item(int i) { this.i = i; }

    @Override
    public int hashCode() { return count++;  }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Item other = (Item) obj;
        return (i == other.i);
    }
}

public class App {
    public static void main(String[] args) {
        HashSet<Item> set = new HashSet<>();
        System.out.println(set.add(new Item(1))); // true
        System.out.println(set.add(new Item(2))); // true
        System.out.println(set.add(new Item(1))); // true
        System.out.println(set.size()); // 3
    }
}
```

Map<K,V>

- Um mapa, Map, permite o armazenamento e gestão de pares atributo-valor (*key-value*)
 - As chaves, K, e os valores, V, podem ser de qualquer tipo não primitivo
 - Não podem existir chaves repetidas, mas os valores podem ser repetidos
- A interface Map<K, V> define o protocolo associado a este tipo de coleção
 - `clear`, `containsKey`, `containsValue`, `get`, `getOrDefault`, `isEmpty`, `keySet`, `put`, `putAll`, `putIfAbsent`, `remove`, `replace`, `size`, `values`, ...
- Se for indicado um novo valor para uma chave já existente (através do `put` ou `replace`) o valor antigo será substituído pelo novo (o valor anterior é retornado em ambas as funções indicadas)

Map<K,V> e HashMap<K,V>




- Existem várias implementações da interface Map sendo uma das mais usadas a HashMap<K,V>
 - Outros: Hashtable, EnumMap, TreeMap, LinkedHashMap, ...

```
HashMap<String,Integer> colors = new HashMap<>();
```

```
colors.put("Orange",0xFF8C00); // 0xFF8C00==16747520
colors.put("Yellow",0xFFFF00); // 0xFFFF00==16776960
System.out.println(colors);      // Output: {Yellow=16776960, Orange=16747520}
```

```
colors.put("Orange",0xFF6500); // 0xFF6500==16737536
System.out.println(colors);      // Output: {Yellow=16776960, Orange=16737536}
```

```
System.out.println(colors.keySet()); // Output: [Yellow, Orange]
System.out.println(colors.values()); // Output: [16776960, 16737536]
```

RGB Colours	
0xFF8C00	
0xFFFF00	
0xFF6500	

Exercício

- Continuação da 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.

...

- d. Desenvolva uma nova versão da classe `Library` recorrendo a um objeto `HashSet` para gerir o conjunto de livros.
- e. Desenvolva uma nova versão da classe `Library` recorrendo a um objeto `HashMap` para gerir o conjunto de livros.

- *Para a versão com `HashMap` deve ser usado o código do livro como chave*

Injeção de dependências

- Na resolução do exercício 13 efetuou-se o desenvolvimento recorrendo a diferentes coleções de dados (ArrayList, HashSet e HashMap)
 - Apesar disso, as funcionalidades oferecidas pela biblioteca são idênticas
- As opções tomadas para a implementação da biblioteca não deverão afetar o desenvolvimento das restantes partes da aplicação
 - Por exemplo, a interface com o utilizador não deve ter de ser adaptada para que possa interagir com qualquer uma das implementações
- O *Software Design Pattern* designado por "Injeção de dependências" (*dependency injection*) ajuda-nos a organizar o código para tornar os módulos do nosso código mais independentes e adaptáveis

Injeção de dependências

- Nesse sentido, quando existem várias implementações possíveis para um mesmo conceito ou funcionalidade:
 - Define-se uma interface com as funcionalidades genéricas oferecidas
 - No exercício 13, deverá ser definida uma interface (ex.: `ILibrary`) que inclui os métodos comuns a todas as implementações realizadas para a classe biblioteca
 - As classes alternativas que oferecem as funcionalidades em causa deverão implementar essa interface e respeitar os nomes definidos
 - No exercício 13 isso corresponderá a que todas as diferentes implementações da biblioteca implementem a interface com as funcionalidades comuns
 - Exemplo: `class LibrarySet implements ILibrary { ... }`

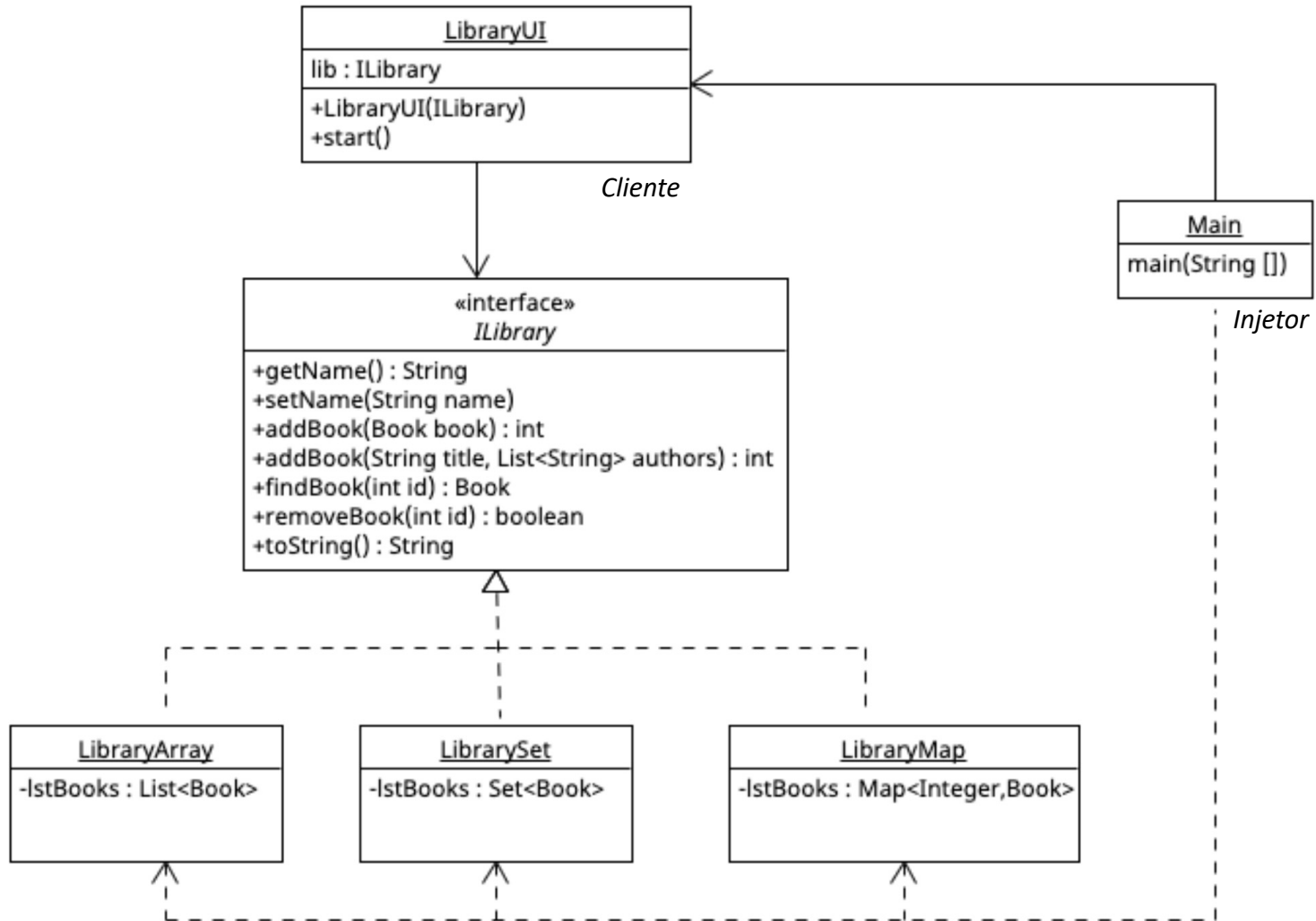
Injeção de dependências

- Após os passos referidos:
 - Os módulos que deverão interagir com os objetos em causa, os *Clientes*, receberão a referências para os mesmos através de uma referência do tipo da interface definida, ou seja, a referência de um objeto que respeita o protocolo definido pela interface (o objeto certamente disponibiliza as funcionalidades pretendidas)
 - No exercício 13 a classe `LibraryUI` receberá uma referência para um objeto `ILibrary`

Injeção de dependências

- Assumindo que um módulo *A* (*A*="injetor de dependências") invoca as funcionalidades de um outro módulo *B* que está dependente de uma versão do objeto criado (*B*="cliente"), então *A* ficará responsável por passar a versão do objeto ("injetar a dependência") com a qual *B* deverá trabalhar
 - No exercício 13 corresponderá a que a função `main` crie o tipo de objeto pretendido em cada execução e o passe para a classe `LibraryUI`
 - Em opção:
 - Criar várias classes `Main`: `MainArray`, `MainSet` e `MainMap`
 - Em cada uma dessas classes realizar código similar ao já existente para criação da biblioteca e passagem para a interface com o utilizador, mas criando objetos dos diferentes tipos de `ILibrary`
 - Criar configurações de execução para cada uma delas e testar individualmente

Injeção de dependências



Injeção de dependências

- Na explicação e exemplos anteriores assumiu-se aquilo que é designado por injeção de dependências pelo construtor
 - O construtor recebe a referência para as dependências necessárias
- Alternativas adicionais
 - Injeção através de *setter*
 - É fornecido um método no cliente que permite passar a dependência já depois de o cliente ter sido criado
 - Injeção através da definição de interface Java
 - É definido uma interface Java para definir o método que o cliente deverá disponibilizar para injetar a dependência
 - Vantagem: permite suportar diferentes tipos de cliente

Service Locator

- Em alternativa à injeção de dependências poder-se-á recorrer à disponibilização das referências para as dependências num serviço/objeto centralizado
 - Deverá existir uma entidade que cria inicialmente as dependências e as regista no *Service Locator*
 - Os clientes consultam o serviço para obter a referência para a dependência necessária
- Na implementação deste modelo normalmente recorre-se adicionalmente a outros padrões ou técnicas que serão estudadas mais tarde:
 - *Singleton*
 - *Multiton*
 - *Late init*

Aplicação destes conceitos

- Exemplos em que a injeção de dependências ou *Service Locator* é usado:
 - Aplicações com múltiplos tipos de fontes de informação
 - Por exemplo, aplicações que podem obter os seus dados através de ficheiros, bases de dados locais, base de dados remotas, web services, ...
 - Desde que seja indicada a dependência adequada para a situação em curso, os módulos poderão tratar os dados independentemente da sua origem
 - Teste de software
 - Gerar dados ou estados fictícios ou extremos para testar as aplicações com valores difíceis de obter numa execução normal

Desafio

- Altere a aplicação desenvolvida para garantir que as classes Library*...
 - não retornam referências para objetos mutáveis pertencentes ao modelo de dados
 - (dependendo da implementação realizada) não armazenam referências para os objetos originais passados por argumento