IPS Instituto Politécnico de Setúbal

Escola Superior de Tecnologia de Setúbal

Programação Avançada 2021/22

1 Introdução aos Tipos Abstratos de Dados

Patricia Macedo , Bruno Silva

Sumário 📝

- Tipos abstratos de dados (ADTs)
  - Acrónimo anglo-saxónico: **A**bstract **D**ata **T**ype
- Metodologia de implementação: *C* vs *Java*
- ADT Stack
- Exercícios

Tipos abstratos de dados | O que são?

Podemos definir tipos abstratos de dados como unidades sintáticas que definem um tipo (de dados) tal que:

- a interface (ou especificação) do tipo **não depende** da (nem obriga a nenhuma) representação concreta dos dados;
- a representação concreta dos dados e a sua manipulação, i.e., a **estrutura de dados**, é ocultada do programa que a utiliza.

Tipos abstratos de dados | O que são?

- São tipos de dados que podemos utilizar em programas sabendo apenas a sua especificação e comportamento esperado, ignorando os detalhes da sua implementação.
- Se a implementação mudar, o tipo comporta-se exatamente da mesma forma, sem necessidade de alterar o programa.

ADTs do tipo Coleção

- Nas ciências de computação os tipos abstratos de dados mais "úteis" são conhecidos por **coleções**, i.e., tipos de dados que permitem armazenar um conjunto de elementos.
- O tipo de "relação" entre elementos permite classificar o tipo abstrato de dados / coleção, e.g.:
  - **Linear** (stack, queue, list)
  - **Não-linear**
    - **Hierárquico** (árvores)
    - **Não-hierárquico** (grafos)

ADTs do tipo Coleção | Utilidade

- São utilizados para:
  - guardar/representar informação diversa;
  - resolver mais facilmente determinados problemas (abstraindo a gestão interna dos elementos);
  - suportar a implementação de diversos algoritmos.

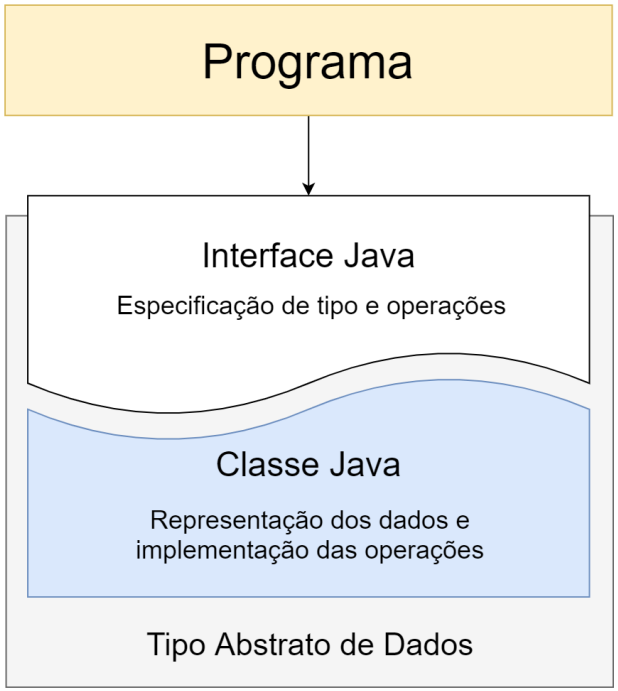
===

Exemplo para **ADT Stack**:

	C	Java
Especificação	stack.h	**interface** Stack.java
Tipo armazenado	StackElem	**Tipo genérico** <T>
Controlo de erros	retorno numérico	**Excepções** Java
Implementação	stackArrayList.c	StackArrayList.java

- A escolha da estrutura de dados é sempre "escondida" na implementação.
- Podem existir múltiplas implementações, sendo que em Java é possível escolher a mesma através de instanciação.

Metodologia ADT em Java



Q: Qual é o ponto de acesso conhecido pelo utilizador/programador?

ADTs (coleções) em Java

A *Java Collections Framework* contém um conjunto de interfaces e classes no *package* `java.util`; as coleções são especificadas através de três interfaces principais:

Interfaces	Implementações
List	ArrayList, LinkedList
Set	HashSet, TreeSet
Map	HashMap, TreeMap, Hashtable

Link: https://en.wikipedia.org/wiki/Java_collections_framework

- As **implementações variam nas estruturas de dados utilizadas**.
- Obedecem à metodologia apresentada.

ADTs (coleções) em Java | Exemplo

Simular um sorteio do **totoloto**:

```
List<Integer> numeros = new ArrayList();
for (int i = 1; i <= 50; i++) numeros.add(i);
Collections.shuffle(numeros);
List<Integer> sorteio = numeros.subList(0, 5);
int suplementar = numeros.get(6);
Collections.sort(sorteio);
System.out.println(sorteio + " + " + suplementar);
```

Resultado (será diferente a cada execução):

```
[12, 15, 33, 34, 43] + 7
```

Note a utilização do tipo genérico <Integer> para definir o tipo de elementos a armazenar; será verificado pelo compilador.

Exemplo de metodologia: ADT Stack

Note que o nosso propósito é o de "ignorar" as interfaces/classes existentes na *Java Collections Framework* e estudarmos a sua implementação na linguagem Java.

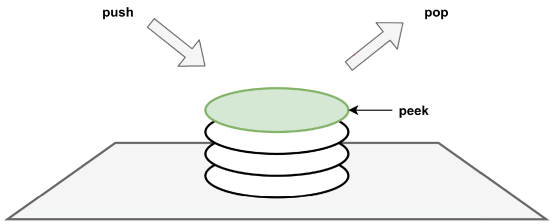
Como tal, vamos incluir as nossas coleções no package:

```
pt.pa.adts
```

Sempre que uma interface/classe tiver o mesmo nome que alguma da Java Collections Framework, a desambiguação será feita pelo nome do package!

ADT Stack | Política de acesso LIFO

- Podemos inserir elementos em qualquer altura,mas **só podemos aceder/retirar (a)o elemento mais recente na coleção**. ➡️ i.e., o elemento no "topo" da pilha.
- LIFO: *last in, first out*:
  - significa que o último elemento a entrar, é o primeiro a sair.
  - analogia a uma pilha de pratos, onde podemos ir empilhando partos, mas só podemos remover o prato que está no **topo**.



===

As operações fundamentais sobre uma *stack* **S** são:

- **push(e)** - insere o elemento **e** no topo de **S**; devolve *erro* se não houver capacidade/memória para mais elementos.
- **pop()** - remove e devolve o elemento no topo de **S**; devolve *erro* se vazia;
- **peek()** - devolve, sem remover, o elemento no topo de **S**; devolve *erro* se vazia;

Operações genéricas de coleções:

- **size()** - devolve o tamanho (nº de elementos) atualmente em **S**;
- **isEmpty()** - devolve um valor lógico (true/false) que indica se **S** está vazia, ou não;
- **clear()** - descarta todos os elementos de **S** voltando a estar vazia;

ADT Stack | Especificação em Java

Especificação do "comportamento" do ADT na interface `Stack.java` (inclui tipo genérico e erros possíveis):

```
public interface Stack<T>  {
        public void push(T element) throws FullStackException;
        public T pop() throws EmptyStackException;
        public T peek() throws EmptyStackException;

        public int size();
        public boolean isEmpty();
        public void clear();
}
```

O **tipo genérico** `<T>` representa qualquer tipo *referenciado* (não-primitivo) na linguagem, que deriva obrigatoriamente da classe `Object`, sendo encarado como desse tipo.

ADT Stack | Especificação em Java

Definição das excepções relativas aos erros possíveis:

```
public class EmptyStackException extends RuntimeException {
        public EmptyStackException(String message) {
                super(message);
        }

        public EmptyStackException() {
                super("The stack is empty.");
        }
}

public class FullStackException extends RuntimeException {
        public FullStackException(String message) {
                super(message);
        }

         public FullStackException() {
                super("The stack is full.");
        }
}
```

ADT Stack | Implementação em Java

Através de uma classe que implemente a interface `Stack.java`; **o nome deverá aludir à estrutura de dados utilizada**`, e.g.:

```
public class StackArrayList<T> implements Stack<T> {
        //representação da informação, i.e. estrutura de dados
        private T[] elements;
        private int size;

        public StackArrayList() {
          //inicialização dos atributos -> pilha vazia
          //...
        }

        @override
        public void push(T element) throws FullStackException {
          //...
        }

        //implementação dos restantes métodos da interface
}
```

ADT Stack |Exer. de implementação 💻

1. Faça *clone* do projeto base **ADTStack_Template** (projeto **IntelliJ**) do *GitHub*:

   https://github.com/estsetubal-pa-geral/ADTStack_Template

2. Forneça a documentação *Javadoc* para a interface `Stack` da forma mais completa possível (interface e métodos).
   - Ver: https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

3. Forneça a documentação para a classe `StackArrayList`, sabendo que utiliza uma estrutura de dados baseada em *array* (da classe e atributos; a documentação dos métodos da interface é herdada).

ADT Stack |Exer. de implementação 💻

4. Forneça o código dos métodos por implementar, i.e., os que estão a lançar `NotImplementedException`;
5. Compile e teste o programa fornecido, verificando que os resultados são os esperados; a excepção `FullStackException` deverá ser capturada com sucesso.
6. Modifique o método `push()` para aumentar dinamicamente o array `elements` sempre que necessário. Compile e teste novamente o programa; a excepção já não deverá ocorrer.

ADT Stack |Exer. de implementação 💻

7. Pretende-se uma diferente implementação baseada em *lista (simplesmente) ligada* na classe `StackLinkedList`. A definição de um nó é fornecida na *inner class* `Node`.

```
public class StackLinkedList<T> implements Stack<T> {
        private Node top; //sentinela
        private int size;

        public StackLinkedList() {
                this.top = new Node(null, null);
                this.size = 0;
        }

        private class Node { //inner class, só reconhecida neste contexto
                private T element;
                private Node next;
                public Node(T element, Node next) {
                        this.element = element;
                        this.next = next;
                }
        }
}
```

ADT Stack |Exer. de implementação 💻

[A realizar autonomamente extra aula]

8. Substitua a implementação de `Stack` utilizada no método `main()` por uma instância da classe anterior. Compile e teste o programa verificando que o comportamento do programa se mantém inalterado.

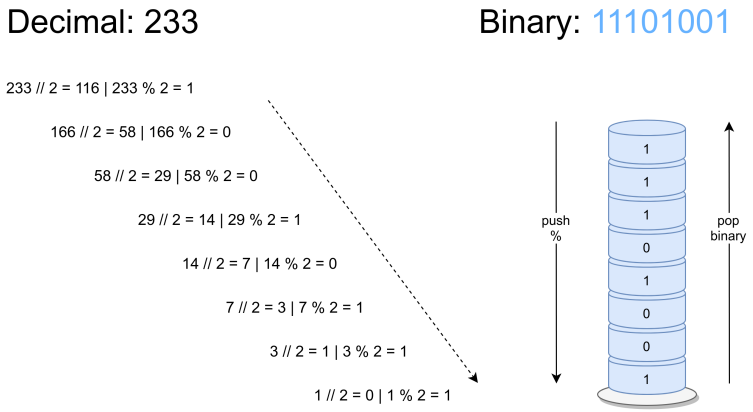
9. Quais as complexidades algorítmicas para as operações `push()` e `pop()` nas duas implementações obtidas?

10. (**Extra**) Para efeitos meramente pedagógicos, remova o atributo `size` da classe `StackLinkedList` e adapte o código existente para o tornar funcional.

ADT Stack |Exer. de implementação 💻

[A realizar autonomamente extra aula]

Um programa deverá solicitar um número ao utilizador, e.g., 233 e apresentar esse número em binário. O algoritmo divide sucessivamente o número por 2 (divisão inteira) até zero e guarda numa pilha o resto das divisões – ver figura. O número em binário é obtido removendo todos os elementos da pilha, i.e., pela ordem de saída. **Obedeça às solicitações do próximo slide**.



ADT Stack |Exer. de implementação 💻

[A realizar autonomamente extra aula]

1. Crie uma classe `DecimalToBinary` contendo um método `main`; implemente o algoritmo solicitado no método:

```
public static String decimal2Binary(int decimal)
```

- Utilize qualquer implementação existente de `Stack`.

2. No método `main` crie o programa que solicita ao utilizador um número decimal e apresente a sua representação em binário; invoque o método anterior.

Bibliografia 📖

pp 40.

António Adrego da Rocha, **Estruturas de Dados e Algoritmos em Java**, 2011. ISBN- 978-972-722-704-4, FCA