Aula 3 – Conceitos avançados em Java

Algoritmos e Estruturas de Dados

Input e Output

Tipo String

- *Strings são cadeias de caracteres*
- *Strings* em Java são implementadas como uma classe que deriva de *Object*
- *Strings em Java são **imutáveis***
  - *Depois de criada, não podemos alterar os caracteres de uma string*

type	set of values	typical literals	operators	typical expressions	
				expression	value
String	character sequences	"AB" "Hello" "2.5"	+ (concatenate)	"Hi, " + "Bob" "12" + "34" "1" + "+" + "2"	"Hi, Bob" "1234" "1+2"

String tokenization

- Divisão de uma string em vários subcomponentes (tokens)

String[] split(String regex)	Separa a string em tokens tendo em conta a expressão regular recebida

```
String exemplo = "Isto é um exemplo: 35";
String[] tokens = exemplo.split(" ");
System.out.print(tokens[3]);
int valor = Integer.parseInt(tokens[4]);
System.out.print(10+valor);
```

separação da string pelo caracter espaço.
o espaço não faz parte do token

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
exemplo:45
Process finished with exit code 0
```

Escrita no ecrã

- System.out → Output Stream

```
public static Integer Inc(Integer x)
{
    return x + 1;
}

public static void main(String[] args) {
    String nome = "Jose";
    int idade = 20;
    System.out.println(nome + ":" + idade);
    System.out.println(nome + ":" + (idade + 5));
}
```

Concatenação de Strings

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Jose:20
Jose:25

Process finished with exit code 0
```

Leitura de terminal

- System.in → InputStream
- BufferedReader → classe leitor com um buffer que facilita a leitura

```
import java.io.*;
public class Main {
    public static void main (String[] args) {
        int primeiro, segundo, resultado;
        InputStreamReader sr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(sr);
        try {
            System.out.println("Introduza o primeiro número:");
            primeiro = Integer.parseInt(br.readLine());
            System.out.println("Introduza o segundo número:");
            segundo = Integer.parseInt(br.readLine());
            resultado = primeiro + segundo;
            System.out.println("A soma é: " + resultado);
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}
```

apenas lê caracteres/bytes

lê cadeias de caracteres (strings)

```
"C:\Program Files\Java\jdk1.8.0_18
Introduza o primeiro número:
20
Introduza o segundo número:
25
A soma é: 45

Process finished with exit code 0
```

Leitura de terminal (com Scanner)

- Scanner -> Classe usada para obter tipos simples e primitivos do input
  - Método de leitura de um ficheiro que é ineficiente
  - Mais é muito fácil de usar

```
import java.util.Scanner;

public class InputMain {

    public static void main (String[] args)
    {
        int primeiro, segundo, resultado;

        Scanner sc = new Scanner(System.in);
        System.out.println("Introduza o primeiro número:");
        primeiro = sc.nextInt();
        segundo = sc.nextInt();
        resultado = primeiro + segundo;
        System.out.println("A soma é: " + resultado);
    }
}
```

Leitura de Ficheiro

- FileReader → cria um leitor capaz de lêr de um ficheiro

```
try
{
    File file = new File(args[0]);
    FileReader fr = new FileReader(file);
    BufferedReader br = new BufferedReader(fr);
    String line;

    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
    fr.close();    //fecha o ficheiro
}
catch(Exception e)
{
    e.printStackTrace();
}
```

abre o ficheiro e cria um *reader* preparado para o lêr

Leitura de Ficheiro (com Scanner)

```
public static void main2(String[] args)
{
    try
    {
        File f = new File(args[0]);
        Scanner sc = new Scanner(f);

        while(sc.hasNextLine())
        {
            System.out.println(sc.nextLine());
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

Observação:
A utilização de um scanner é sempre mais simples

Escrita de ficheiro

- FileWriter → Classe capaz de escrever para um ficheiro
- BufferedWriter → Classe utilitária para tornar escrita mais fácil

```
try {
    BufferedWriter bw;
    String conteudo = "Esta frase será escrita no ficheiro!";

    File file = new File(args[0]);

    FileWriter fw = new FileWriter(file);
    bw = new BufferedWriter(fw);
    bw.write(conteudo);
    System.out.println("Escrita bem sucedida");
    bw.flush();
    bw.close();

} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Linguagem Java

Alguns Conceitos Avançados

Tipos e métodos genéricos

- Por vezes vemo-nos obrigados a escrever funções muito semelhantes para diferentes tipos

```c
void doubles2_exchange_rows(double **a, int x, int y)
{
  double *m = a[x];
  a[x] = a[y];
  a[y] = m;
}
```

exemplo em C

```c
void ints2_exchange_rows(int **a, int x, int y)
{
  int *m = a[x];
  a[x] = a[y];
  a[y] = m;
}
```

exemplo em C

É Chato ☹

E propenso a erros ☹

Tipos e métodos genéricos

- tipo/método genérico
  - Def: classe/método genérico que pode ser parametrizado com vários tipos não primitivos

parâmetro ou variável de tipo

```
public class Box<T> {
    // T stands for "Type"
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}


public static void main(String[] args) {
    Box<String> bS = new Box<String>();
    Box<Integer> bI = new Box<Integer>();

    bI.set(2);
    bS.set("ola");
}
```

não podemos usar o tipo int!

argumento de tipo

Exemplos de Métodos Genéricos

```
 public static <T> void imprimeLista(List<T> objList)
 {
     for(T t : objList)
     {
         System.out.println(t.toString());
     }
 }
public static <T extends Comparable<T>> T maxList(List<T> objList)
{
    T maxValue = objList.get(0);
    for(T element : objList)
    {
        if(element.compareTo(maxValue)>= 1)
        {
            maxValue = element;
        }
    }

    return maxValue;
}
```

Tipo parametrizado restrito

Não funciona para listas vazias

Tipos primitivos como objectos

- Java tem classes objecto para os vários tipos primitivos
  - usados em colecções (argumentos de tipos genéricos)
  - tipos invólucro (wrapper types)

Tipo primitivo	Invólucro
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

Tipos primitivos como objectos

- Autoboxing – conversão automática de um tipo primitivo para a respectiva classe invólucro
- Unboxing – converte um objeto invólucro no correspondente tipo primitivo

```
public static Integer Inc(Integer x)
{
   return x + 1;
}

public static void main(String[] args) {

 Integer[] inteiros = {1,2,3,4,5};

 int x = inteiros[0];
 Integer y = new Integer(1);
 Integer z = Inc(x);
}
```

autoboxing

unboxing

manual “boxing”

autoboxing

Funções de 1ª Classe

em Java

Funções de 1.ª classe

- Diz-se que uma linguagem tem funções de 1.ª classe se ela trata as funções como cidadãos de 1.ª classe
  - Podem ser nomeadas
  - Podem ser guardadas em variáveis
  - Podem ser guardadas em coleções e outras estruturas de dados
  - Podem ser passadas como argumentos de funções
  - Podem ser devolvidas como retorno de função
  - Podem ser criadas de forma dinâmica (a partir de outras funções)

Funções de 1.ª classe em Java

- Em Java, as funções podem ser:

✓ nomeadas
✓ guardadas em variáveis
✓ guardadas em coleções e outras estruturas de dados
✓ passadas como argumentos de funções
✓ devolvidas como retorno de função
✓ criadas de forma dinâmica (a partir de outras funções)

Observação:
A linguagem java é uma linguagem que cumpre a 100% os requisitos necessários para ter funções de 1.ª classe.

E consegue fazê-lo de forma ligeiramente mais elegante.

Funções como Ponteiros

- Em C, nomes de funções eram ponteiros para funções
- Em Java, também usamos ponteiros para trabalhar com funções.
- Mas como vos disse anteriormente, não podemos trabalhar explicitamente com ponteiros em Java.
- Mas podemos trabalhar com objetos.

Funções como Ponteiros

- Em Java, funções são objetos que implementam obrigatoriamente uma interface funcional

interface	Método funcional	descrição
`Function<T,R>`	`R apply(T t)`	função com um argumento e que retorna um resultado
`Consumer<T>`	`void accept(T t)`	função que recebe um argumento T e não retorna nada
`Supplier<T>`	`T get()`	função sem argumentos que retorna um objecto do tipo T
`Runnable`	`void run()`	função sem argumentos que não retorna nada
`Predicate<T>`	`boolean test(T t)`	recebe 1 argumento, e retorna um booleano (true ou false)

Exemplos de interfaces funcionais, para consultar a lista completa de interfaces funcionais, aceda a:
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html#package.description

Funções

de ordem superior

Função de Ordem Superior

- Função de Ordem Superior
  - ***Def:***

    Função que recebe uma função como argumento ou retorna uma função

Exemplo

```
public static <T> List<T> filtraLista(List<T> lista, Predicate<T> p)
{
    List<T> result = new ArrayList<>();

    for(T t : lista)
    {
        if(p.test(t))
        {
            result.add(t);
        }
    }

    return result;
}
```

Invocação de Método de Ordem Superior

- 3 Formas de passar um método como argumento
  - Referência para método
  - Classe anónima
  - Expressão lambda

Referência para método

Main::isEven

nome da classe onde o método está definido

nome do método

```
public static boolean isEven(Integer i)
{
    return i%2 == 0;
}

public static void main(String[] args)
{

    List<Integer> lista = Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9});
    filtraLista(lista,Main::isEven);
}
```

Exemplo de referências para métodos

- Main criado para o Mooshak correr os testes

```
public static void main(String[] args)
{
    HashMap<String,List<Runnable>> unitTests = new HashMap<>();

    unitTests.put("A",ClubeTest.getAllTests());
    unitTests.put("B",JogadorTest.getAllTests());
    unitTests.put("C",UtilsTest.getAllTests());

    String problem = args[0];
    int testNumber = Integer.parseInt(args[1])-1;

    unitTests.get(problem).get(testNumber).run();
}
```

Lista de métodos runnable

invocação de método funcional

Exemplo de referências para métodos

```
public class ClubeTest {

    public static List<Runnable> getAllTests()
    {
        List<Runnable> tests = new ArrayList<Runnable>();
        tests.add(ClubeTest::test1);
        tests.add(ClubeTest::test2);
        tests.add(ClubeTest::test3);
        tests.add(ClubeTest::test4);
        return tests;
    }

     …

}
```

```
public static void test1()
{
    System.out.println("Teste ao construtor");
    Clube c = new Clube("SPORTING",10);
    String nome = c.getNome();
    int interna = c.getInternacionalizacoes();
    System.out.println(nome);
    System.out.println(interna);
}
```

Classe anónima

- Definição de classe sem lhe darmos um nome

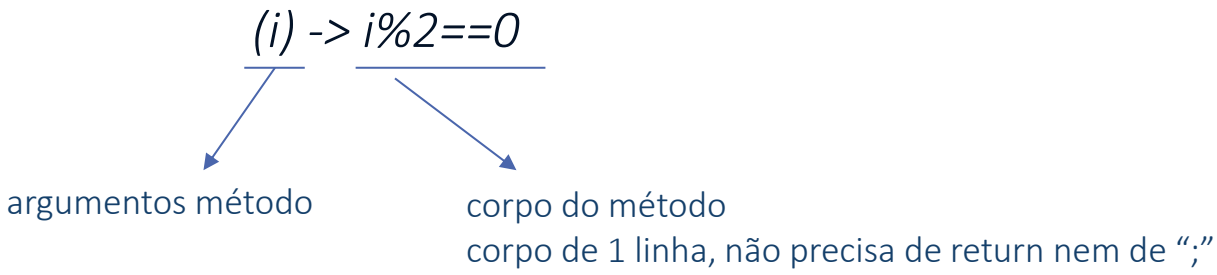
```
public static void main(String[] args)
{

    List<Integer> lista = Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9});
    ThreeSum.filtraLista(lista,
            new Predicate<Integer>() {
                @Override
                public boolean test(Integer i) {
                    return i%2;
                }}
                );
}
```

Cria uma instância de uma classe anónima que implementa a interface Predicate<Integer>

Lambda

- Expressão lambda
  - ***Def**: expressão que define um método ou função sem lhe dar um nome (função anónima)*




```
public static void main(String[] args)
{

    List<Integer> lista = Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9});

    filtraLista(lista, (i) -> i%2==0);
}
```

Lambda

- E os tipos do argumentos, e tipo de retorno?
  - *É inferido pelo compilador a partir da instrução onde o lambda é usado*

(i) -> {i++;return i%2==0;}

argumentos método

corpo do método
corpo de múltiplas linhas, precisa de ";" e return

```
public static void main(String[] args)
{

    List<Integer> lista = Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9});

    filtraLista(lista, (i) -> i%2==0);
}
```

Método filtraLista está à espera de um predicado que recebe um *Integer* e devolve um *boolean*