INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL

DELEGAÇÃO REGIONAL DO CENTRO

CENTRO DE EMPREGO E FORMAÇÃO PROFISSIONAL DE VISEU

Programação orientada a objetos

Aprendizagem + Tecnologias e Programação de Sistemas de Informação

UC00608 – Desenvolver programas em linguagem orientada a objetos

FORMADOR/A
Bruno Silva

DATA
novembro de 2026

PESSOAS 2030 · PORTUGAL 2030 · Cofinanciado pela União Europeia

Os Fundos Europeus mais próximos de si.

Índice

Capítulo 1 – Introdução a linguagem de programação Java 3
Capítulo 2 – Passos de criação de um novo projeto no Eclipse.....4
Capítulo 3 – Estrutura de um programa 9
Capítulo 4 – Tipos de dados, variáveis e constantes 11
4.1 – Regras dos Nomes Variáveis e Constantes.....11
4.3.1 - Caracteres ou cadeias de caracteres (texto).....12
4.3.2 - Numéricos (inteiros / decimais).....13
5.3.3 - Lógicos ou booleanos 13
4.3.4 – Declaração e inicialização de variáveis 13
Capítulo 5 – Ler valores do teclado.....15
Capítulo 6 – Tipos de Operadores 18
6.1 - Aritméticos 18
6.2 - Comparação.....18
6.3 - Lógicos.....18
Capítulo 7 – Estruturas de Controlo 19
7.1 – Estrutura de decisão simples.....20
7.2 – Estrutura de decisão dupla 21
7.3 – Estrutura de decisão dupla encadeada 23
7.4 – Estrutura de decisão seletiva.....24
7.5 – Estruturas de repetição For (contador automático).....26
7.6 – Estruturas de repetição While (com contador manual) 29
7.7 – Estruturas de repetição Do...While (com contador manual) 30
7.8 - Diagrama do mapa das estruturas de controlo 32
Capítulo 8 – Funções e subprogramas - conceitos e utilidades 33
8.1 – Subprogramas: procedimentos, funções e passagem de parâmetros 35
8.1.1 – Procedimentos (com e sem passagem de parâmetros).....35
8.1.2 – Funções (com e sem passagem de parâmetros) 36
Capítulo 9 – Introdução à Programação orientada por objetos 37
9.1 - Programação estruturada (sequencial) 37
.....37
9.2 - Programação Orientada a Objetos (POO) 38
9.3 – Conceitos de Programação Orientada a Objetos.....39
9.3.1 - Encapsulamento de informação 39
9.3.2 - Composição / Herança.....39
9.3.3 - Polimorfismo.....40

Paradigmas de Programação: Objetos, Classes e Instâncias 40
9.5 - Paradigmas de Programação 40
Capítulo 10 – Classes 41
10.1 – Introdução as classes 41
10.2 – Definição de Classe 42
10.2.1 – Declaração de uma Classe e Modificadores de Acesso 42
10.2.2 – Variáveis de instância de uma Classe 43
10.2.3 – Construtores de uma classe 43
10.2.4 – Métodos de instância de uma Classe 44
10.2.5 – Overload de Métodos de uma Classe 45
10.2.6 – Classes e as Palavras reservadas: new e this 45
10.2.7 – Definição de Classe – Exemplo de uma classe com métodos 46
10.2.8 – Exemplo de criação de um novo objeto a partir de uma Classe 47
10.2.9 – Exemplo de criação de um novo objeto a partir de uma Classe 47
11 - Heranças 48
11.1 – Constituição e Manipulação Entre Classes 48
11.2 - HERANÇA – criar uma classe derivada de Produto – Livro 50
11.3 - HERANÇA – criar uma classe derivada de Produto - Software 51
12 – Interfaces 52
12.1 – Estruturas e comportamentos nas Heranças 52
12.2 – Interfaces 53
12.3 – Criação de uma Interface 54
12.4 – Tipos de Assinaturas na Interface 55
12.5 – Exemplo de interfaces 56
13 – Classes Abstratas 57
14 – Exceções 61
14.1 – Construção do Bloco de Excepções Try-Catch 62
14.2 – Exemplo Prático da Construção do Bloco de Exceções Try-Catch 63
14.3 – Tratamento de Excepções e a Classe Throwable 64
14.4 – Lançamento de exceções (Por iniciativa do Programador) 66
15 – Coleções em Java – Collections (Coleções) 67
15.1 – Construtores da classe ArrayList 68
15.2 – Exemplo de 1 Arraylist com 1 tipo de dados 69
15.3 – Exemplo de 1 Arraylist com Classes 70

Capítulo 1 – Introdução a linguagem de programação Java

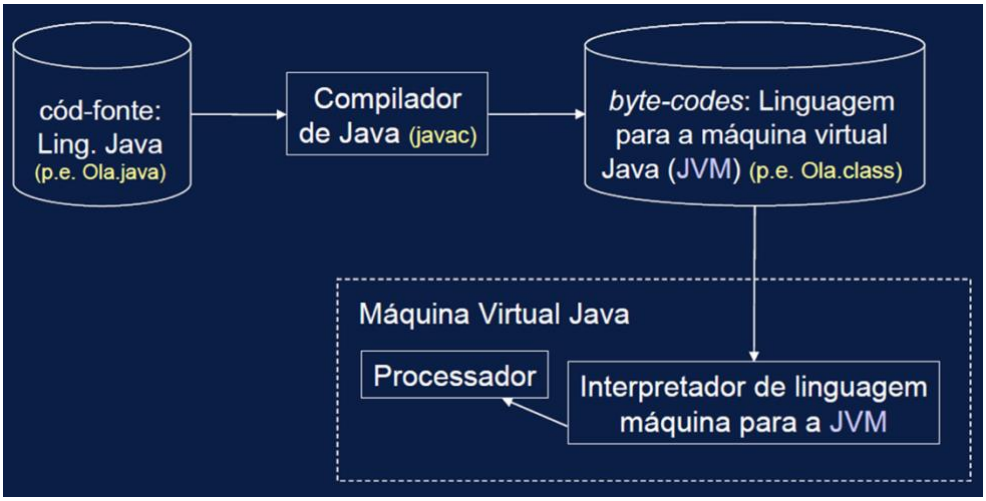
A linguagem JAVA foi desenvolvida pela SUN. Inicialmente começou a pensar na área da eletrónica, mas rapidamente estendeu-se para a Web;

Esta linguagem consiste numa linguagem orientada por objetos e constituída por uma vasta coleção de classes. Esta linguagem tem a vantagem de ser multiplataforma e as aplicações são executadas em cima de uma máquina virtual chamada JVM;



A linguagem também é caracterizada por ser uma linguagem robusta e fortemente tipificada;

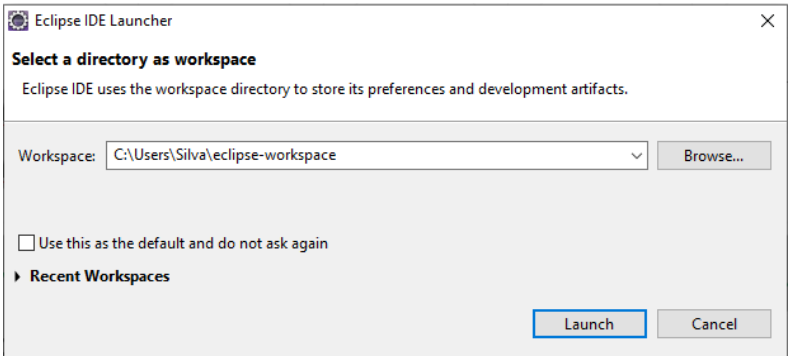
Mas, apesar de ser uma linguagem rápida, não pode ser comparada a linguagens como o C++ porque o Java trata-se de uma linguagem interpretada (interpretado pelo JVM).



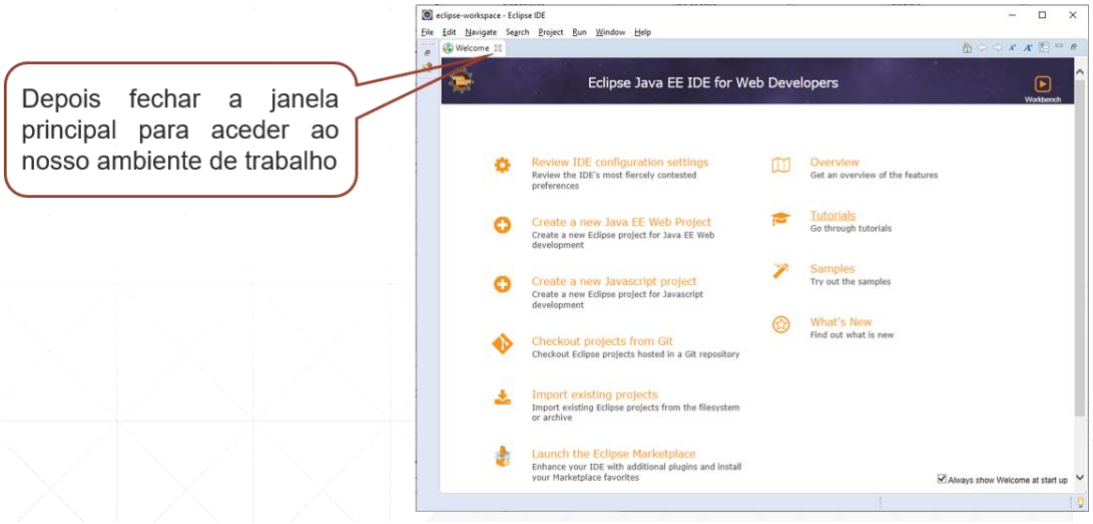
Capítulo 2 – Passos de criação de um novo projeto no Eclipse

1. ***Iniciar o programa Eclispe;***

2. ***Podem deixar a <u>workspace</u> (espaço de trabalho) predefinida pelo programa. Aqui vão ficar os vossos projetos que irão realizar ao longo do modulo.***

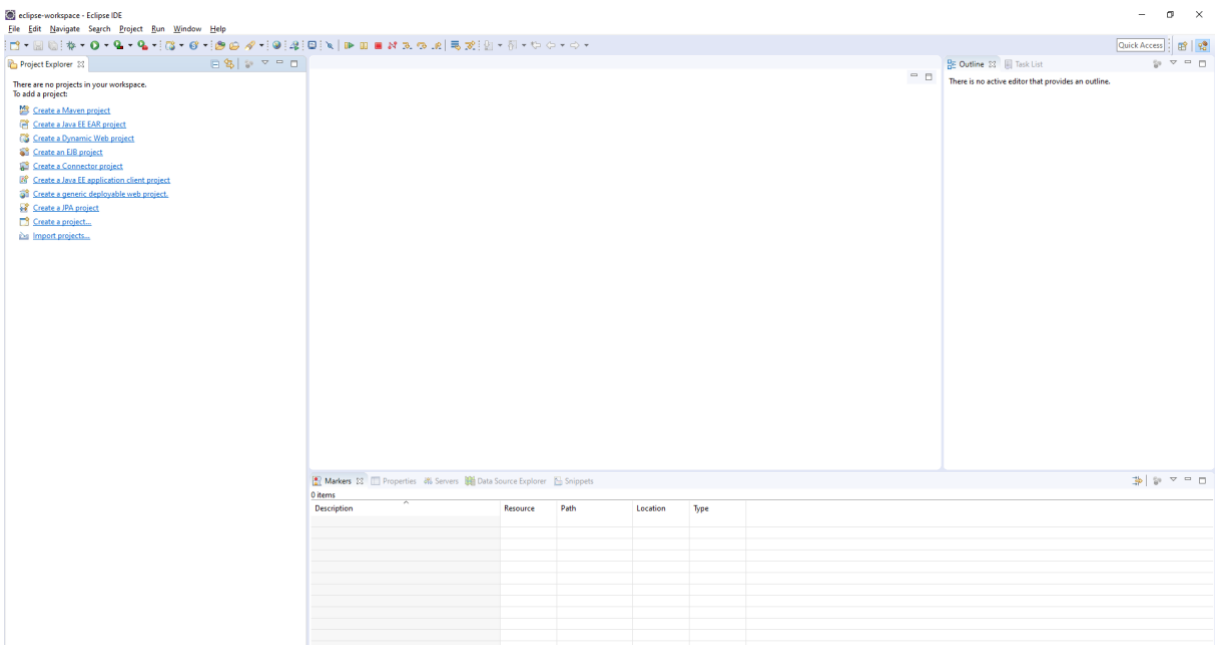


De seguida, irá abrir o seguinte ambiente de trabalho:



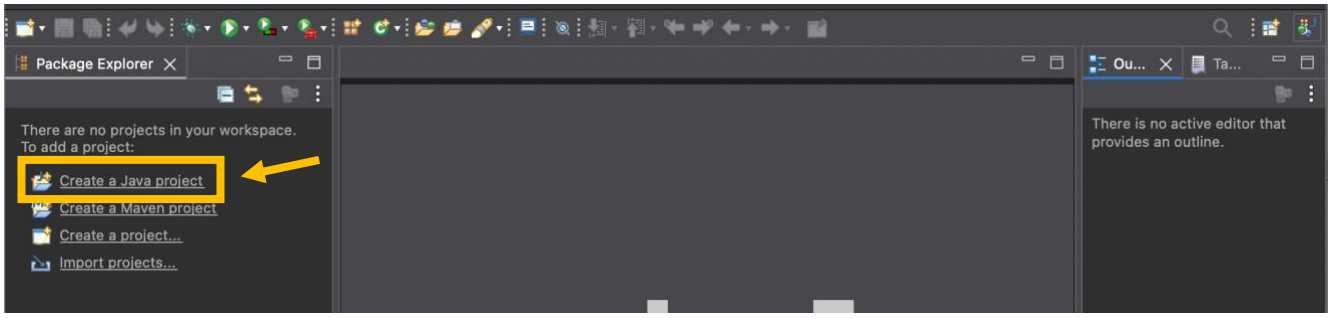
3. ***De seguida, vamos explorar o nosso ambiente de trabalho:***

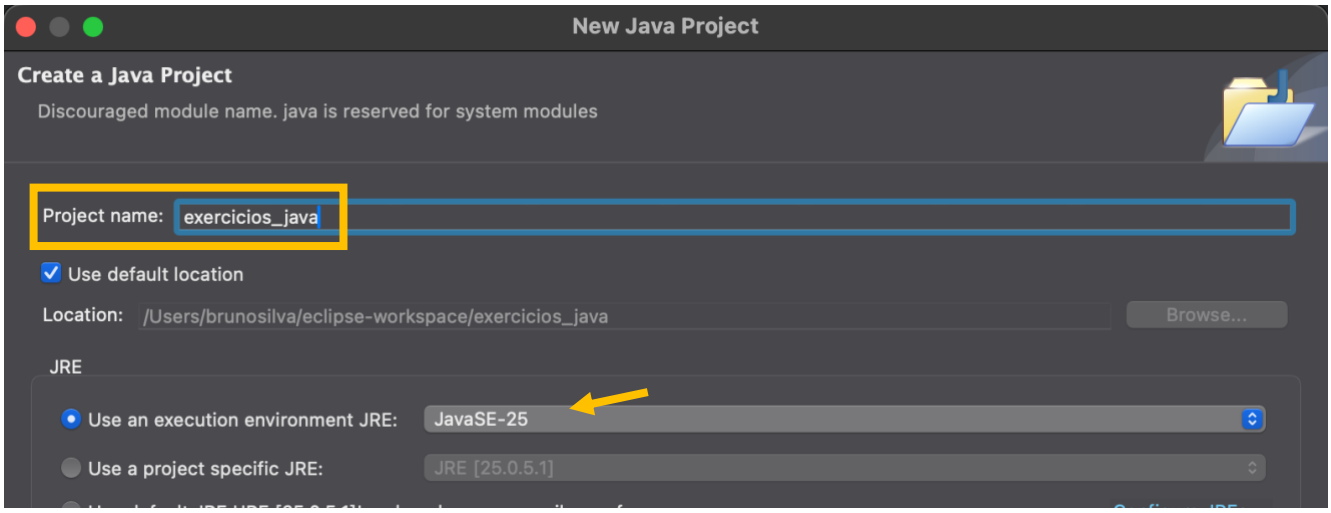
    1. No lado esquerdo, vamos ter o navegador de ficheiros, do qual irá mostrar todos os packages, classes, ficheiros, etc… (depois vamos estudar com mais pormenor);
    2. No meio vai estar o nosso editor de texto / código, do qual vamos escrever o conteúdo desejado;
    3. Do lado direito, tem várias janelas opcionais, mas por defeito irá mostrar uma área com a informação do métodos, funções, etc… que foram escritos no código, mas uma consulta rápida;



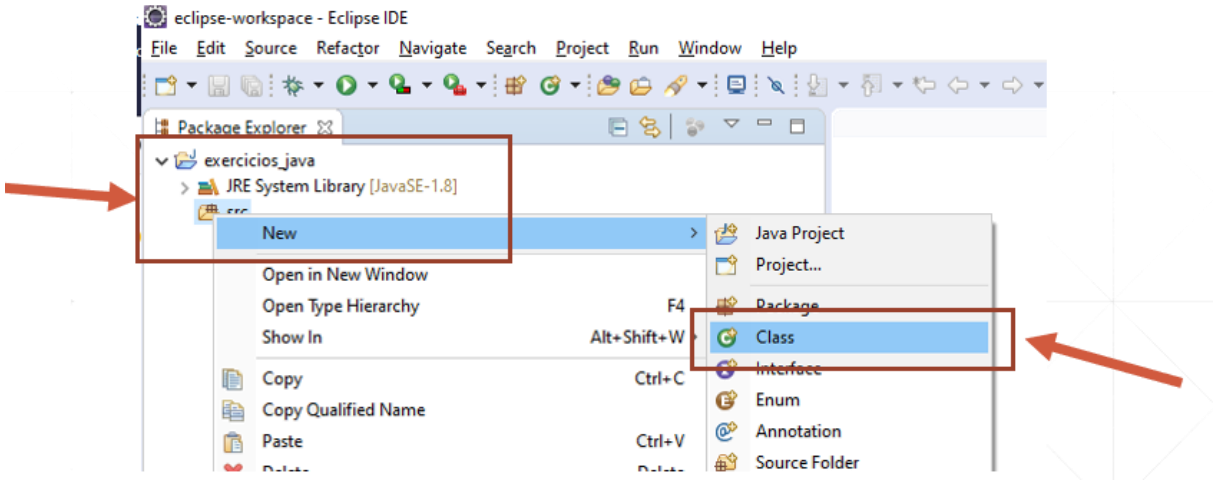
4. ***Agora vamos criar um novo projeto em JAVA:***

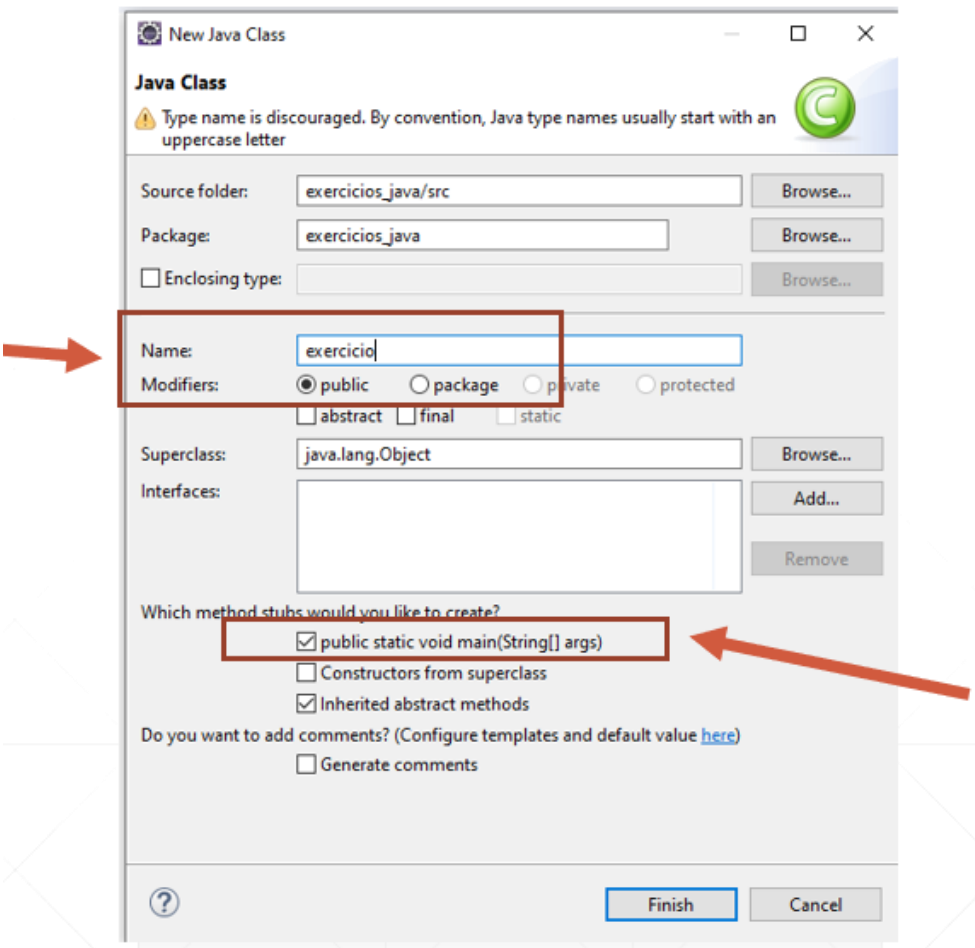
No lado esquerdo, vamos ter o navegador de ficheiros:



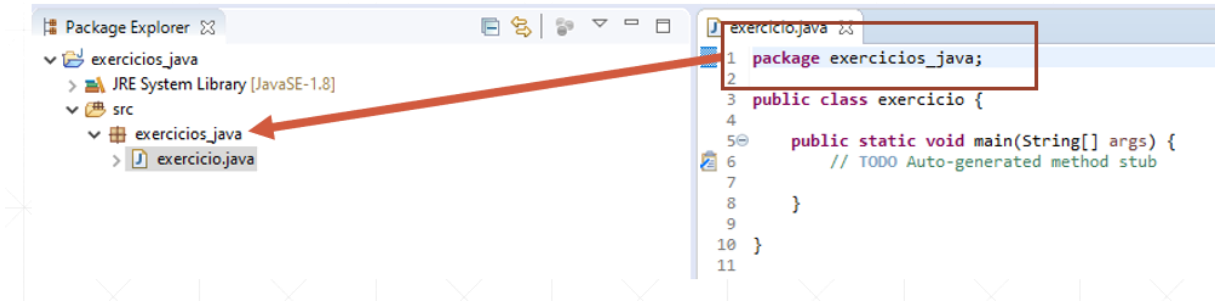


5. Já temos o projeto criado. Agora falta criar a classe principal (que será o ***main()***). É aqui que começa a execução do nosso programa. Vamos desmembrar o explorador de ficheiros até a pasta ***src*** e de seguida com o lado direito do rato clicar em ***Class***:





6. E cá está o ***resultado final***. Falta agora perceber a estrutura do mesmo. Logo no cimo vamos ter o ***package*** (se reparem bem, está associado com o mesmo nome de onde o ficheiro ***exercício.java*** pertence).



Tudo em Java está organizado por ***classes*** e algumas classes são desenvolvidas pelo ***programador da aplicação*** enquanto que outras já estão disponíveis através do núcleo de funcionalidades da plataforma Java.

As classes que compõem o núcleo de funcionalidades Java estão organizadas por pacotes (***packages***). Um ***package*** Java é um mecanismo para agrupar classes de finalidades afins ou de uma mesma aplicação.

Para ***facilitar a localização e o acesso às diversas classes existentes***, a plataforma JAVA ***organiza as classes em packages***. Os ***packages*** permitem que o programador importe apenas as classes necessárias à aplicação em questão.

Por outro lado, os packages permitem organizar as diversas classes produzidas em compartimentos ou pacotes distintos em função da sua funcionalidade;

Seguindo as ***orientações teóricas da PPO*** conjugado com a ***organização das classes em packages***, estes potenciam a reutilização de código/classes em diversas aplicações;

Os ***nomes dos packages*** devem ser ***únicos*** para que ***não haja conflitos com outros packages.***

Para acrescentar uma classe a um ***package***, a primeira definição da classe deve ser a declaração do próprio ***package***. É o mesmo que dizer que a classe a definir faz parte do package;

Qualquer ficheiro que ***pretenda utilizar o package*** deve utilizar a ***directiva "import"*** para indicar a necessidade de aceder às classes do package, ou seja:

- Podemos efetuar ***uma importação qualificada***, quando especificamos ***qual a classe pretendemos importar*** (ex: ***import java.lang.String;***)
- Ou podemos ***recorrer à importação global*** (ex: ***import java.util.*;***) e assim engloba todo;



Entre os principais pacotes oferecidos como parte do núcleo Java estão:

- ***java.lang*** (utilizado para evocar as classes String, System, (…))
- ***java.util*** (utilizado para estrutura, coleções, classes de datas, (…))
- ***java.io*** (utilizado para criar e manipulação de ficheiros, (…))
- ***java.awt*** (utilizado para criar de ecrãs gráficos, (…))
- ***java.net*** (utilizado para serviços de rede, (…))
- ***(…)***

Para consultar mais informações sobre a documentação da linguagem JAVA, devemos ir sempre a aplicação online, para consultar e esclarecer as dúvidas sobre esses mesmo componentes:

https://docs.oracle.com/javase/8/docs/ (***versão 8***)

https://docs.oracle.com/javase/7/docs/ (***versão 7***)

Capítulo 3 – Estrutura de um programa

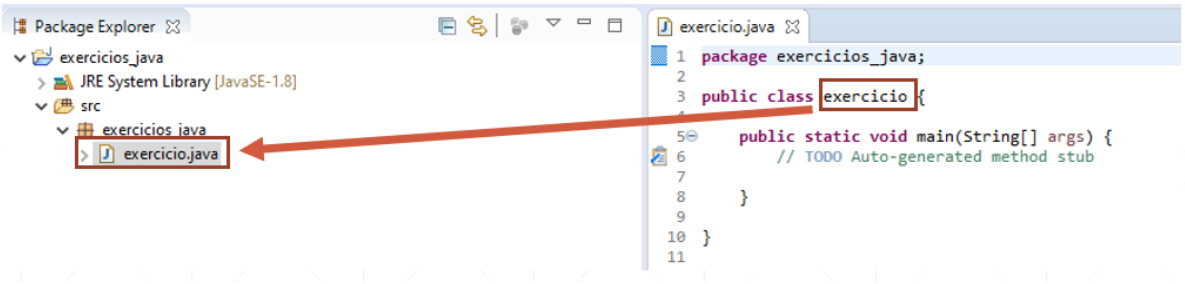
A parte operativa dum subprograma, função ou programa é composto por um conjunto de **instruções**. Numa linguagem de programação as **instruções** correspondem à codificação dum algoritmo (pseudolinguagem ou fluxograma), de acordo com a sintaxe específica da linguagem.

Tipos de Instruções:

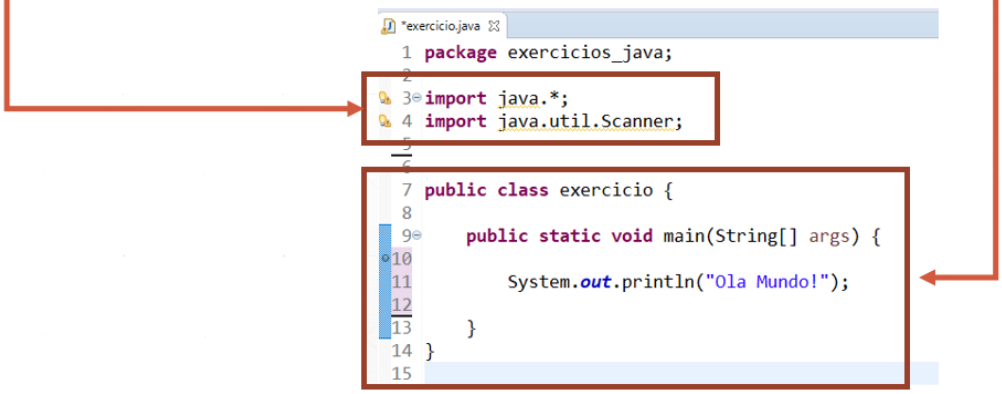
- ***Leitura e Escrita***
- ***Atribuição***
- ***Estruturas de controlo***

Em cada ficheiro de Java só pode existir uma classe e ***o nome do ficheiro tem de coincidir ao nome da classe com extensão java***.

Para esta estrutura o ficheiro tem de se chamar algo parecido como: "BemVindo***.java***".



A estrutura de um ficheiro JAVA, tem 2 seções: a zona de ***importação das bibliotecas*** e de seguida vamos ter o ***código propriamente dito*** (que será composto pela classe e pelas funções):



3.1 – Atribuição de valores

O ***operador de atribuição*** é constituído simplesmente ***por igual "="***.

Tal como o símbolo utilizado na descrição algorítmica ("←"), também tem o significado "toma o valor de ". A instrução em pseudo-linguagem i ← 3 seria codificada em JAVA através da instrução **= *(igual)***:

***int* resultado =** 3 + 5;

***String* palavra =** "Programação";

3.2 – Escrever mensagens no ecrã

Para escrever informação no ecrã utilizamos um dos dois comandos:

System.out.print();

ou

System.out.println(); ***

*** A última versão, este comando efetua mudança de linha depois de escrever a mensagem no ecrã;

Exemplos:

- ***System.out.println(***375***);***
- ***int x =*** 12***;***
- ***System.out.println(***x***);***
- ***System.out.println(*** " " + 375 + "-" + x***);***

Ecrã:

375
12
375-12

```
1  package exercicios_java;
2
3  import java.*;
4
5  public class exercicio {
6
7      public static void main(String[] args) {
8          System.out.println(375);
9
10         int x = 12;
11         System.out.println(x);
12         System.out.println(" " + 375 + "-" + x);
13
14     }
15
16 }
17
```

```
<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (05/09/2020, 18:15:31)
375
12
 375-12
```

Capítulo 4 – Tipos de dados, variáveis e constantes

Uma variável corresponde a uma posição da memória onde vamos guardar a informação, cujo conteúdo pode ser alterado durante a execução de um programa, ou seja, é onde guardamos os valores.

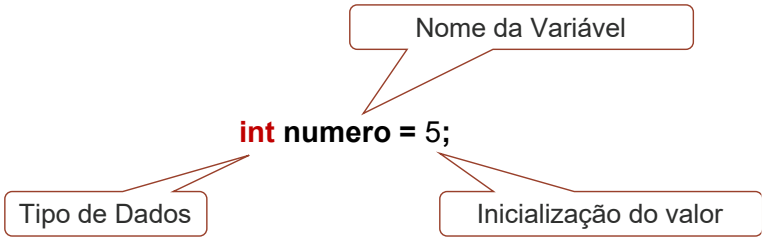
Embora uma variável possa assumir diferentes tipos de dados, ela só pode armazenar um valor a cada instante. A linguagem JAVA possui 8 tipos de dados primitivos:

Inteiros	Virgula flutuante	Caractere	Verdadeiro / Falso

`byte (8bits)`
`short (16bits)`
`int (32bits)`
`long (64bits)`
	`float (8bits)`
	`double(16bits)`
		`char (16bits)`	`boolean (1bit)`

-1,-345,2 07 0xFF	1.2 .78645 6.234E22	'g' '\uFFFF' '\n'	true False

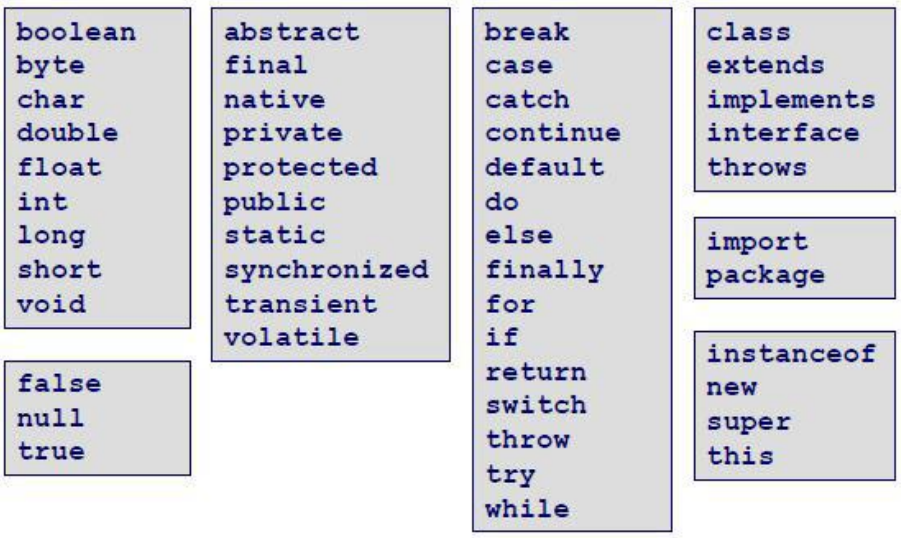
Na linguagem de programação em java, temos de definir ***3 partes*** para declarar uma variável:



4.1 – Regras dos Nomes Variáveis e Constantes

O nome de uma variável/constante **não pode**:

- ***conter espaços e operadores (+, -, +, /, %) (ex: soma+dois);***
- ***começar por um número (ex: 1numero);***
- ***conter acentos (ex: preparação);***
- ***ser igual a uma palavra reservada da linguagem de programação, como por exemplo:***



4.3 – Tipos de dados elementares

- ***Numéricos*** (Inteiro, Decimais);
- ***Strings*** (caracter ou cadeia de carateres (texto));
- ***Booleanos*** (apenas contém um de dois valores lógicos): (**true** (verdadeiro) ou **false** (falso))

4.3.1 - Caracteres ou cadeias de caracteres (texto)

Existem dados que são do tipo caracteres (char - de character em inglês);

O tipo char serve para armazenar UM, e somente UM caracter. Para declarar, usamos a seguinte sintaxe:

char texto1 = 'b'

Se for preciso armazenar vários caracteres (texto), temos que usar os dados que são do tipo texto ou cadeias de caracteres (strings).

Existem dados que são do tipo texto ou cadeias de caracteres (strings);

String texto2 = "Hoje é dia 14"

Para este tipo de dados também se utiliza, por vezes, a designação de dados alfanuméricos – caracteres alfabéticos, alfanuméricos e também algarismos.

Exemplo: **String** texto = "A1B2c3#5!6".

4.3.2 - Numéricos (inteiros / decimais)

Existem dados que são do tipo numéricos e podem ser de vários tipos:

int → Inteiros (Exemplo: 10, -5, 0, 5, 10...)

Float / Double → Decimais (Exemplo: 1.5, 0, 2.5, - 4.1...)

Exemplos: int numero1 = 123 ***ou*** **double** numero1 = 123.12

Nota Importante: na programação é utilizado o "." como ***sinal decimal***

5.3.3 - Lógicos ou booleanos

Existem dados que são do tipo lógicos ou booleanos. São utilizados com muita frequência em algoritmia e programação. Este tipo de dados caracteriza-se por admitir de cada vez um de dois resultados:

Verdadeiro(true, 1) ***ou*** Falso(false, 0)

boolean estado = true ***ou*** ***boolean*** estado = false

4.3.4 – Declaração e inicialização de variáveis

As variáveis armazenam valores que podem ser alterados com a execução do programa com exceção das contastes.

- Sintaxe típica de ***declaração de variáveis*** e ***inicialização de variáveis***:

<***Tipo_dados***> ***nome_variável = valor***;

Ex: ***int idade =*** 32***;***

Ex: ***string texto =*** "ola mundo!"***;***

4.4 – Declaração e inicialização de constantes

Uma constante, é como uma variável, mas o seu valor deve manter-se constante (apenas de leitura) enquanto o pograma estiver a correr, ***isto é***, não devem ser alteradas. Qualquer alteração do valor, deve ser efetuada no código.

Como boa regra de programação, os nomes das constantes têm de ***começar com a primeira letra do nome em letra maiúscula***

- Sintaxe típica de declaração de variáveis e ***inicialização de variáveis constantes***:

final <*Tipo_dados*> *nome_variável* **= valor**;

Ex: **final** float Pi **= *3.14;***

4.5 – Casting e a conversão de variáveis

Os tipos de dados primitivos numéricos são compatíveis entre si dentro de certos limites:

byte → short → int → long → float → double

Basicamente podemos converter um tipo numérico de menor precisão (à esquerda) por um tipo numérico de maior precisão (à direita).

Ainda existe o "***casting explicito***" quando pretendermos ***converter um tipo numérico de maior precisão em um de menor precisão***:

int **x = (*int*) 93.75;** (***x*** fica com o valor de 93)

Capítulo 5 – Ler valores do teclado

A classe ***Scanner (que provém da biblioteca java.util),*** facilita a ***interação de obtenção de dados provenientes*** *do teclado.* Logo, para usar o scanner é preciso importar o package (pacote) dessa classe, ou seja:

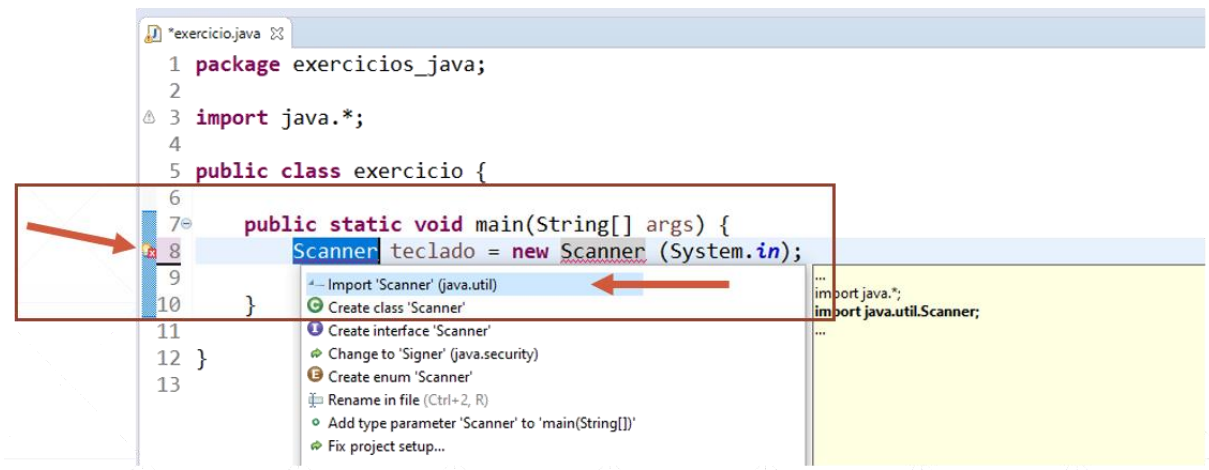
import java.util.Scanner;

De seguida vamos ter de criar um objeto do tipo scanner para conseguirmos aceder as funções do pacote que foi importado:

Scanner teclado = new Scanner (System.in);

Como vamos ter de criar um objeto do tipo scanner para conseguirmos aceder as funções do pacote que foi importado (***Scanner teclado = new Scanner (System.in);)***, temos de verificar se a palavra ***Scanner*** não dá erro.

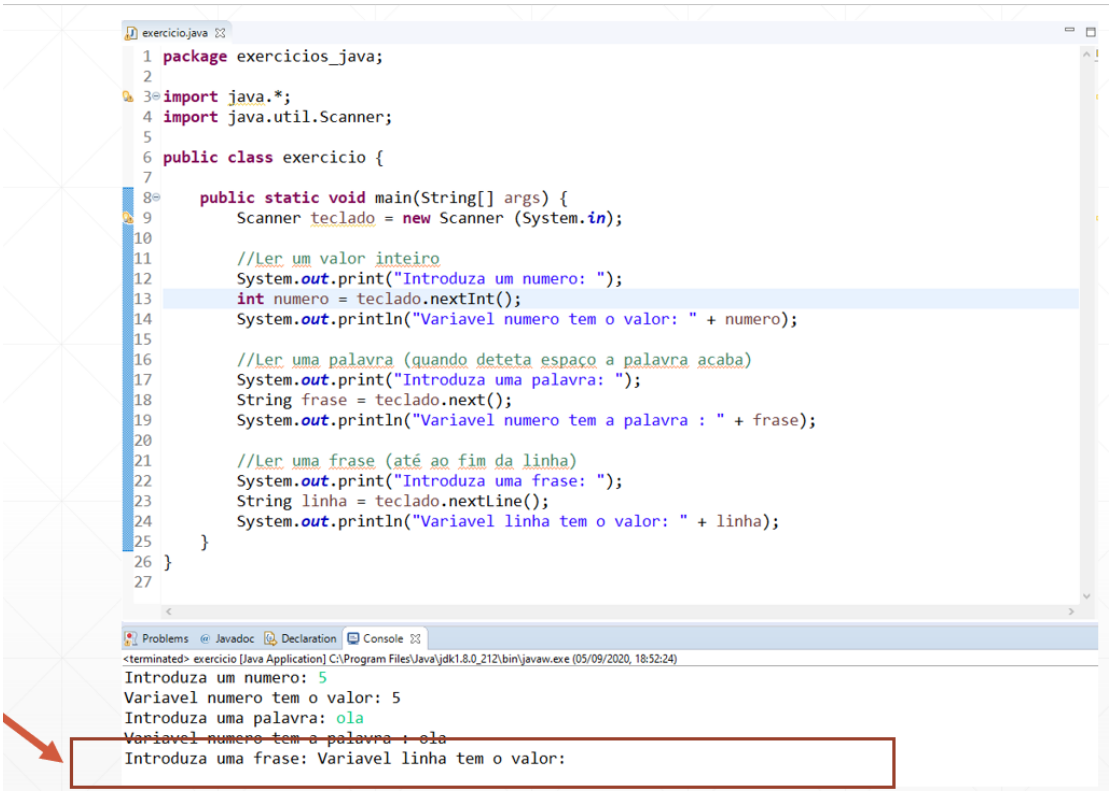
Caso isso aconteça, devemos ***importar a biblioteca que contém essa função***, bastando clicar na lâmpada amarela ao pé dos números das linhas e clicar na opção ***import 'Scanner'*** (…), como podemos ver a seguir:



Depois de feitos os passos anteriores, podemos ler diferentes tipos de dados (***int***, ***float***, ***double***, ***char***, ***string***, etc…), como por exemplo:

- ***Um valor inteiro*** → ***int n = teclado.nextInt();***
- ***Uma palavra*** → ***String n = teclado.next();***
- ***Uma frase (ate ao fim da linha)*** → ***String n = teclado.nextLine();***

De seguida, vamos perceber que ***quando queremos ler um valor inteiro ou uma string*** e ***de seguida*** se for ***lida uma linha inteira***, vamos ter um pequeno problema.



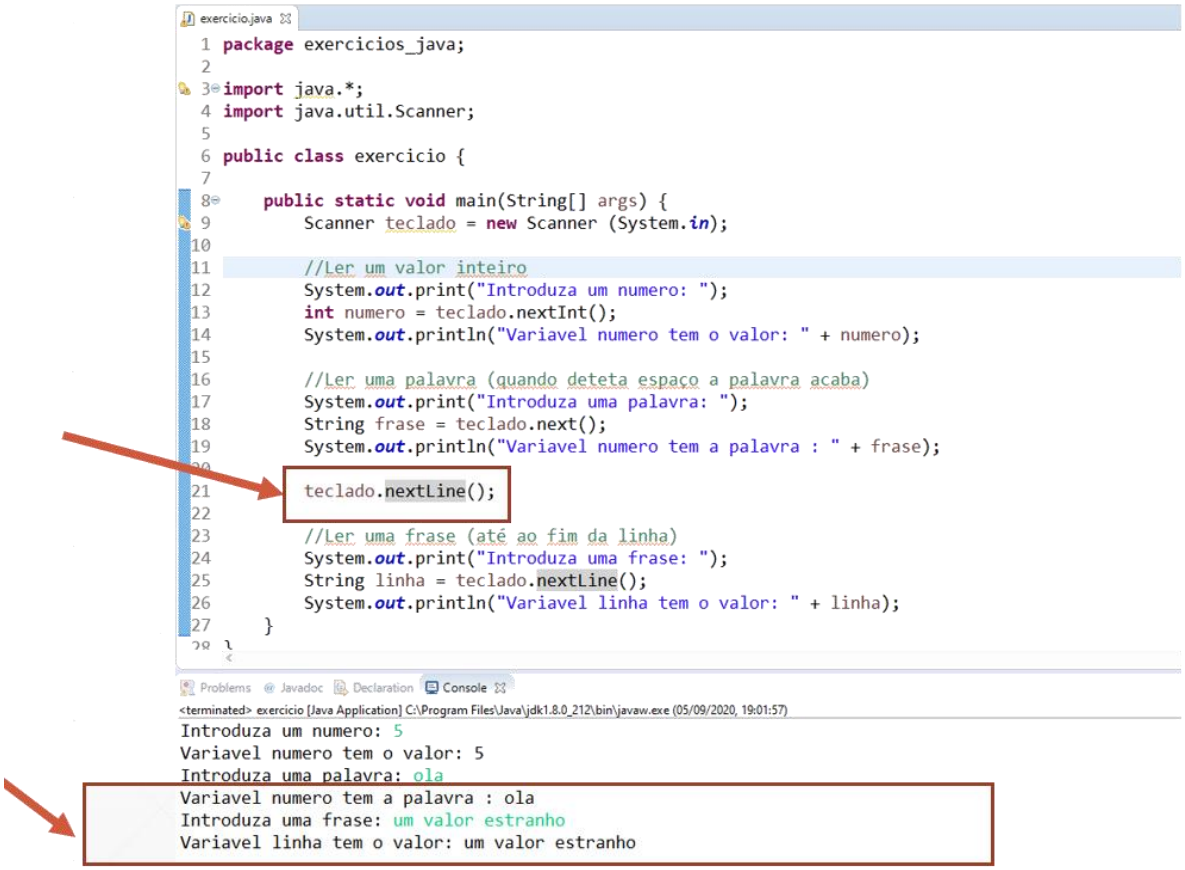
Conseguimos ler o número inteiro e uma palavra, mas quando chega a terceira operação para ler uma linha de texto completa, o programa finalizou!

Isto acontece, porque o método ***Scanner.nextInt*** ou ***Scanner.next*** não lê o caracter da nova linha quando carregamos na tecla "Enter“ e, com isso, a chamada para ***Scanner.nextLine*** retorna após a leitura dessa nova linha.

Logo o programa assumiu o ***"Enter“,*** que foi digitado anteriormente, como uma linha e finalizou o programa

Para resolver este tipo de situações, vamos fazer ***teclado.nextLine()***, para que seja possível ler o valor pretendido na 3 operação.

Basicamente estamos a enganar o sistema, do qual estamos a dizer que o programa deve descartar o ***"Enter“*** que foi digitado anteriormente.



As classes de **Scanner** e **BufferedReader** são usadas para ler a entrada de dados externos. O scanner é normalmente usado quando sabemos que a entrada é de tipo string ou de tipos primitivos.

No entanto, temos a classe BufferedReader ***(que provem da biblioteca java.io),*** que é usado para a leitura de carateres de forma mais eficiente.

Resumindo, a biblioteca Scanner é ideal para ler números e a BufferedReader seria ótima para ler carateres e texto.

Exemplo com as duas classes:



Capítulo 6 – Tipos de Operadores

6.1 - Aritméticos

Adição (**+**), ***Subtração*** (**-**), ***Multiplicação*** (*), ***Divisão*** (/), ***Resto da divisão*** (**%**) e ***Expoente*** (**).

Operador	Descrição	Exemplo	Resultado
+	Soma	21 + 4	25
-	Subtração	21 – 4	17
*	Multiplicação	21 * 4	84
/	Divisão	21 / 4	5,25
%	Resto da Divisão	21 % 4	1
**	Expoente	2 ** 4	16

6.2 - Comparação

Operador	Descrição	Exemplo	Resultado
>	Maior	5 > 3	5 é **maior que** 3?
>=	maior ou igual	5 >= 4	5 é **maior ou igual que** 4?
<	menor	2 < 9	2 **é menor que** 9?
<=	menor ou igual	9 <= 9	9 **é menor ou igual que** 9?
==	igual / comparação	4 == 4	4 **é igual a** 4?
!=	diferença	2 != 2	2 **é diferente qu**e 2?

6.3 - Lógicos

Operador	Descrição	Exemplo	Resultado
&&	**E** lógico	(2 > 1 **&&** 3 > 1)	**Verdadeiro** (porque satisfez as duas condições)
||	**Ou** lógico	(2 > 1 **||** 3 < 1)	**Verdadeiro** (porque satisfez as uma das duas condições)
!	negação	**!**true	O contrário de **Verdadeiro** é **Falso**

Negação

A	~A
V	F
F	V

Operador "E"

A	B	A ∧ B
V	V	V
F	V	F
V	F	F
F	F	F

Operador "OU"

A	B	A ∨ B
V	V	V
F	V	V
V	F	V
F	F	F

Capítulo 7 – Estruturas de Controlo

Neste capítulo, vamos entender como funcionam as várias estruturas de controlo, dos quais podemos classificar em 2 grupos:

- ***Estruturas de decisão condicional***

  Permitem, com base na análise de uma condição (geralmente, o resultado duma expressão booleana), decidir sobre a execução ou não de determinada instrução ou bloco de instruções. Estas podem ser classificadas como:

  - Estruturas de Decisão Simples ***(If)***
    - Condições e os operadores lógicos
  - Estruturas de Decisão Dupla ***(If / else);***
  - Estruturas de Decisão Dupla Encadeada ***(If / else if / else)***
  - Estruturas de Decisão Seletiva ***(case / when / default);***

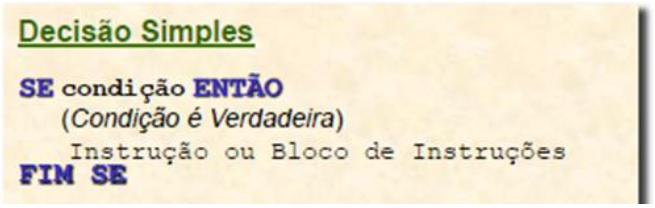
- ***Estruturas de repetição***

  Permitem repetir a mesma instrução ou bloco de instruções quantas vezes sejam necessárias. Estas podem ser classificadas como:

  - Repetição com contador automático ***For***
  - Repetição com contador manual***:***
    - ***While***
    - ***Do ... While***
    - ***Foreach***

7.1 – Estrutura de decisão simples

Estruturas de Decisão Simples: nesta estrutura de controlo, a instrução ou bloco de instruções serão executadas ***apenas se a condição testada for verdadeira***.



Caso seja, a instrução ou bloco de instruções que serão executadas devem estar ***dentro de chavetas*** { código… } :

If (nota >= 9.5) **{** /***inicio** do bloco de instruções*/

System.out.print(“Nota **positiva**”);

} /***fim** do bloco de instruções*/

*exercicio.java

```
1  package exercicios_java;
2
3  import java.*;
4  import java.util.Scanner;
5
6  public class exercicio {
7
8      public static void main(String[] args) {
9          Scanner teclado = new Scanner (System.in);
10
11         int numero = 0;
12
13         //Ler um valor inteiro
14         System.out.print("Introduza um numero: ");
15         numero = teclado.nextInt();
16
17         if( numero <= 5 ) {
18             System.out.println("Inserir um numero válido!");
19         }
```

7.2 – Estrutura de decisão dupla

Estruturas de Decisão Dupla: permitem, com base na análise de uma condição, decidir sobre a execução **ou não** de determinada instrução ou bloco de instruções, ***mas neste caso dá para testar quando a condição é verdadeira e o que acontece quando a condição é falsa***

If (nota >= 9.5) **{** /***inicio** do bloco de instruções do caso **verdadeiro***/

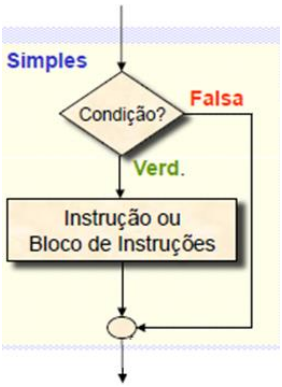
System.out.print(“Nota ***positiva***”);

} /***fim** do bloco de instruções do caso **verdadeiro** */

else **{** /***inicio** do bloco de instruções do caso **falso***/

System.out.print(“Nota ***negativa***”);

} /***fim** do bloco de instruções do caso **falso** */



Exemplo:

*exercicio.java

```
1  package exercicios_java;
2
3  import java.*;
4  import java.util.Scanner;
5
6  public class exercicio {
7
8      public static void main(String[] args) {
9          Scanner teclado = new Scanner (System.in);
10
11         int numero = 0;
12
13         //Ler um valor inteiro
14         System.out.print("Introduza um numero: ");
15         numero = teclado.nextInt();
16
17         if( numero <= 5 ) {
18             System.out.println("Inserir um numero válido!");
19         }
20         else {
21             System.out.println("Inseriu um numero fora dos limites!");
22         }
23
24     }
25 }
26
```

Problems Javadoc Declaration Console

```
<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (05/09/2020, 22:17:05)
Introduza um numero: 9
Inseriu um numero fora dos limites!
```

Também podemos conjugar condições e operadores lógicos nas estruturas de decisão condicional. Neste caso vamos utilizar o operador lógico ***&& (e)*** do qual obriga que as 2 condições têm de ser verdadeiras (basta 1 falhar para a condição ser falsa):

Quando queremos juntar várias condições com o operador lógico ***E*** utilizamos os símbolos ***&&:***

Ex: **if(numero >= 1 && numero <= 5) { (...) }**

*exercicio.java

```java
package exercicios_java;

import java.*;
import java.util.Scanner;

public class exercicio {

    public static void main(String[] args) {
        Scanner teclado = new Scanner (System.in);

        int numero = 0;

        //Ler um valor inteiro
        System.out.print("Introduza um numero entre 1 e 5: ");
        numero = teclado.nextInt();

        if( numero >= 1 && numero <= 5) {
            System.out.println("Inserir um numero válido!");
        }
        else {
            System.out.println("Inseriu um numero fora dos limites!");
        }

    }
}
```

Quando queremos juntar várias condições com o operador lógico ***OU*** utilizamos os símbolos ***||*** ***:***

Ex: **if(numero >= 1 || numero <= 5) { (...) }**

*exercicio.java

```java
package exercicios_java;

import java.*;
import java.util.Scanner;

public class exercicio {

    public static void main(String[] args) {
        Scanner teclado = new Scanner (System.in);

        int numero = 0;

        //Ler um valor inteiro
        System.out.print("Introduza um numero: ");
        numero = teclado.nextInt();

        if( numero == 1 || numero <= 5) {
            System.out.println("Inserir um numero válido!");
        }
        else {
            System.out.println("Inseriu um numero fora dos limites!");
        }

    }
}
```

7.3 – Estrutura de decisão dupla encadeada

Estruturas de Decisão Dupla: permitem, com base na análise de uma condição, decidir sobre a execução **ou não** de determinada instrução ou bloco de instruções, ***mas neste caso dá para testar quando a condição é verdadeira e o que acontece quando a condição é falsa.***



As instruções ou blocos de instruções que serão executadas devem estar ***dentro de chavetas*** { código… }

*exercicio1.java

```java
package exercicios_java;

public class exercicio1 {

    public static void main(String[] args) {
        double nota = 14.75;

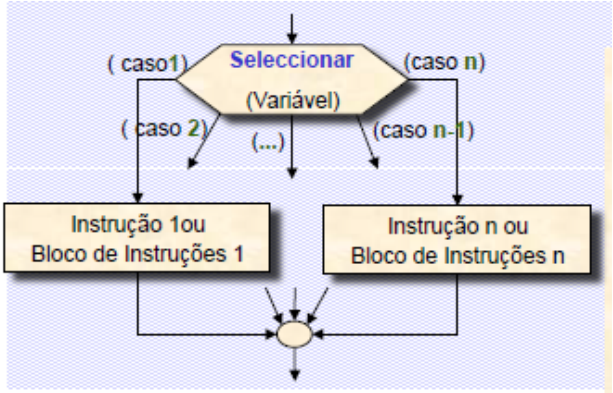
        if(nota >= 0 && nota < 9.5) {
            System.out.println("Nota Negativa");
        }
        else if (nota >= 14 && nota < 17) {
            System.out.println("Nota Média");
        }
        else if (nota >= 17 && nota < 20) {
            System.out.println("Nota Excelente");
        }
        else {
            System.out.println("Nota Invalida");
        }
    }
}
```

7.4 – Estrutura de decisão seletiva

Quando é necessário decidir entre **múltiplas** opções possíveis para a instrução ou bloco de instruções a executar, podemos utilizar **várias estruturas condicionais duplas** (**Se**, **Então**, **Senão**) **encadeadas**

ou então, quando conhecemos as opções a trabalhar, usamos

a **estrutura Condicional de Seleção**.



Regras do Switch:

1. A estrutura ***Switch*** (Caso) inicia a execução pela avaliação da expressão do caso.
2. Depois de calculada a expressão do ***Switch***, o controlo é transferido para o ***caso mais apropriado***, por comparação do resultado da expressão do ***Switch***.
3. A instrução ***break*** faz sair da instrução ***Switch***. Se um caso contiver uma ou mais instruções, deve terminar com um break.
4. Se não houver nenhum caso que satisfaça a condição do ***Switch***, o case ***default*** será executado automaticamente.

Pode-se especificar mais do que um valor para que um bloco de código seja executado, especificando os diversos casos e respetivos valores.

exercicio.java

```java
package exercicios_java;

import java.*;
import java.util.Scanner;

public class exercicio {

    public static void main(String[] args) {
        Scanner teclado = new Scanner (System.in);

        int nota = 0;

        //Ler um valor inteiro
        System.out.print("Introduza a nota do teste: ");
        nota = teclado.nextInt();

        switch(nota) {
            case 1: System.out.println("Mediocre"); break;
            case 2: System.out.println("Mau"); break;
            case 3: System.out.println("Suficiente"); break;
            case 4: System.out.println("Bom"); break;
            case 5: System.out.println("Muito Bom"); break;

            default: System.out.println("Nota Invalida!"); break;
        }
    }
}
```

Problems Javadoc Declaration Console

<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (05/09/2020, 22:21:34)

```
Introduza a nota do teste: 4
Bom
```

7.5 – Estruturas de repetição For (contador automático)

Estruturas de repetição For (com Contador Automático): esta estrutura de controlo permite a repetição duma determinada instrução ou bloco de instruções, durante um número de vezes predefinido, e duma forma automática.

Cada repetição condicional é designada por **Ciclo**.

FOR(<condição de inicio>; <condição de fim>; <ação no final do ciclo>);

- ***condição de inicio -»*** iniciar a variável
- ***condição de fim -»*** qual a condição que irá satisfazer a condição
- ***ação no final do ciclo -»*** normalmente vamos dizer o que acontece a variável da condição de inicio: ***incrementa*** ou ***desincrementa;***
- ***Exemplo Prático:***

```
for (int contador = 1; contador <= 10; contador++)
{
      // calcula a soma
      soma = soma + contador;
}
```

Exemplo: Vamos simular o processo de repetição do cálculo da soma dos números de 1 a 5;

Quando começamos o nosso programa, vamos ter 2 valores a ter em conta e que serão determinantes para o ciclo do programa:

- ***A soma (acumulador):*** pois será o nosso acumulador das somas;
- ***O contador (contador):*** pois devemos dar +1 ao contador de cada vez que percorre a estrutura de repetição, ou seja, ***de cada vez que corre o ciclo*** da estrutura de repetição;
    - De cada vez que percorre a estrutura de repetição (***um ciclo***), vamos ***adicionar +1 ao valor do contador de forma automática***;

- Uma estrutura de repetição deve ser ***repetida***, ***tantas vezes quantas necessárias***, até que ***chegue a condição de saída*** da estrutura de repetição;

Valores Iniciais: Soma = ***0*** e Contador = ***1***

Ciclo	*Soma*	*Contador*
Inicio	***0***	***1***
1	***1***	***2***

1º Ciclo: ***Soma = 1*** (que é a soma atual ***(0)*** mais o contador atual ***(1)*** e que o novo valor será atribuído na variável soma = 1) e ***Contador = 2*** ***(***pois temos contador++ no 3 parâmetro do FOR, logo soma +1 ao contador atual que é ***1***, ou seja fica ***2****)*

Ciclo	*Soma*	*Contador*
Inicio	***0***	***1***
1	1	2
2	***3***	***3***

2º Ciclo: ***Soma = 3*** (que é a soma atual ***(1)*** mais o contador atual ***(2)*** e que o novo valor será atribuído na variável soma = 3) e ***Contador = 3*** ***(***pois temos contador++ no 3 parâmetro do FOR, logo soma +1 ao contador atual que é ***2***, ou seja fica ***3****)*

Ciclo	*Soma*	*Contador*
Inicio	***0***	***1***
1	1	2
2	3	3
3	***6***	***4***

3º Ciclo: ***Soma = 6*** (que é a soma atual ***(3)*** mais o contador atual ***(3)*** e que o novo valor será atribuído na variável soma = 6) e ***Contador = 4*** ***(***pois temos contador++ no 3 parâmetro do FOR, logo soma +1 ao contador atual que é ***3***, ou seja fica ***4****)*

Ciclo	*Soma*	*Contador*
Inicio	***0***	***1***
1	1	2
2	3	3
3	6	4
4	***10***	***5***

4º Ciclo: ***Soma = 10*** (que é a soma atual ***(6)*** mais o contador atual ***(4)*** e que o novo valor será atribuído na variável soma = 10) e ***Contador = 5*** ***(***pois temos contador++ no 3 parâmetro do FOR, logo soma +1 ao contador atual que é ***4***, ou seja fica ***5****)*

Ciclo	*Soma*	*Contador*
Inicio	***0***	***1***
1	1	2
2	3	3
3	6	4
4	10	5
5	***15***	***6***

5º Ciclo: ***Soma = 15*** (que é a soma atual ***(10)*** mais o contador atual ***(5)*** e que o novo valor será atribuído na variável soma = 15) e ***Contador = 6*** ***(***pois temos contador++ no 3 parâmetro do FOR, logo soma +1 ao contador atual que é 5, ou seja 6*)*

Se o ***Contador = 6***, quer dizer que ultrapassa a condição, ficando o resultado da ***soma = 15***.

- ***Soma : inteiro ← 0***
- ***PARA*** contador ← 1 ***ATÉ*** 5 ***FAZ***
  - ***soma ← soma + contador;***
- ***FIM PARA***
  - ***(+1 ao contador atual)***
  - ***(voltar ao inicio do PARA)***

Ciclos	Valor do Contador	Valor da Soma
Valores iniciais	1	0
1º Ciclo	2	1
2º Ciclo	3	3
3º Ciclo	4	6
4º Ciclo	5	10
5º Ciclo		15

Exemplo para somar todos os números de 1 até 5 (inclusive):

exercicio.java

```
1  package exercicios_java;
2
3  import java.*;
4  import java.util.Scanner;
5
6  public class exercicio {
7
8      public static void main(String[] args) {
9          Scanner teclado = new Scanner (System.in);
10
11         int soma = 0;
12
13         //Somar todas as notas de 1 a 5 (inclusive)
14         for(int i = 0; i <= 5; i++) {
15             soma = soma + i;
16         }
17
18         System.out.print("Soma total: " + soma);
19     }
20 }
21
```

Problems @ Javadoc Declaration Console

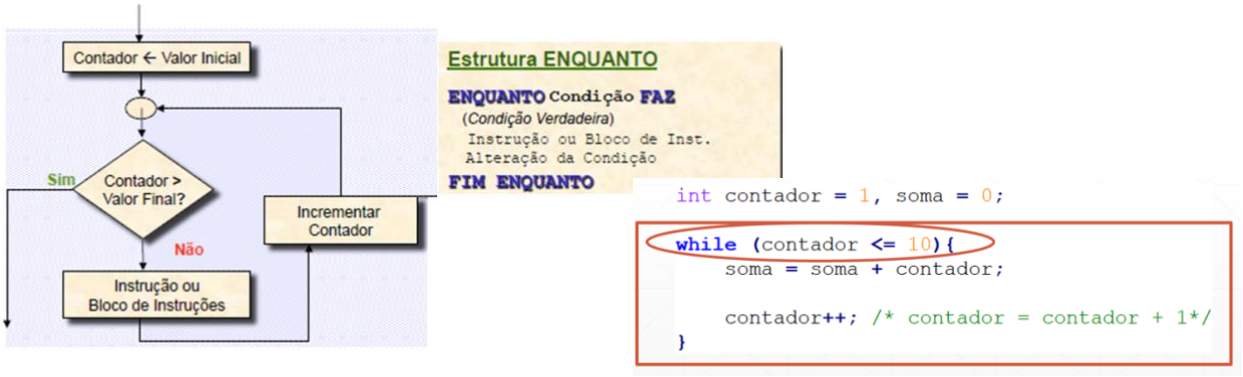
```
<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (05/09/2020, 22:24:00)
Soma total: 15
```

7.6 – Estruturas de repetição While (com contador manual)

Estruturas de repetição While: o teste à condição é realizado antes de executar as instruções dentro estrutura de repetição e ***enquanto a condição for verdadeira***, volta a executar a instrução ou bloco de instruções, caso contrário sai do ciclo.

O teste à condição é realizado ***no início do while*** **(**antes de executar qualquer instrução contida numa estrutura de repetição) e enquanto a condição for verdadeira, volta a executar a instrução ou bloco de instruções, caso contrário sai do ciclo.

Dentro do bloco de instruções terá que ocorrer uma alteração que interfira com a condição, caso contrário, entraria em ciclo infinito.



Exemplo para somar todos os números de 1 até 5 (inclusive):

```
exercicio.java
1  package exercicios_java;
2
3  import java.*;
4  import java.util.Scanner;
5
6  public class exercicio {
7
8      public static void main(String[] args) {
9          Scanner teclado = new Scanner (System.in);
10
11         int soma = 0, contador = 0;
12
13         //Somar todas as notas de 1 a 5 (inclusive)
14         while(contador <= 5) {
15             soma = soma + contador;
16
17             contador++; // contador = contador + 1;
18         }
19
20         System.out.print("Soma total: " + soma);
21     }
22 }
23
```

```
Problems  Javadoc  Declaration  Console
<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (05/09/2020, 22:26:57)
Soma total: 15
```

7.7 – Estruturas de repetição Do...While (com contador manual)

Estruturas de repetição Do...While: o teste à condição só é realizado depois de executar as instruções contidas numa estrutura de repetição ***e enquanto a condição for verdadeira***, volta a executar a instrução ou bloco de instruções, caso contrário sai do ciclo.

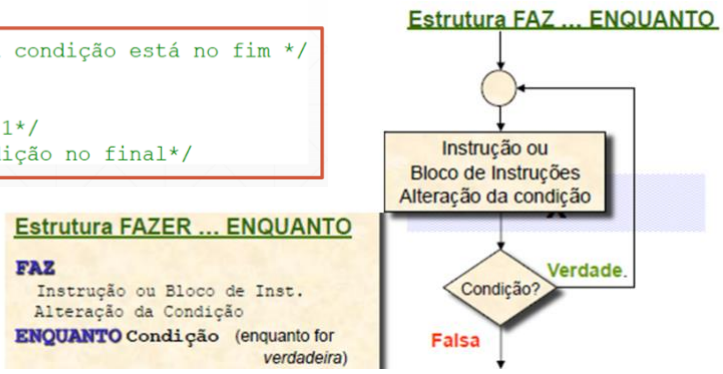
- Como esta estrutura ***só avalia a condição no final***, *temos a certeza de que as instruções indicadas serão executadas* ***pelo menos uma vez***.

Nesta estrutura de repetição ***a condição*** é ***avaliada apenas no final do ciclo***.

```
int contador = 1, soma = 0;

do{ /* vai funcionar sempre 1 vez, pois a condição está no fim */
    soma = soma + contador;

    contador++; /* contador = contador + 1*/
}while (contador <= 10) /*só testa a condição no final*/
```

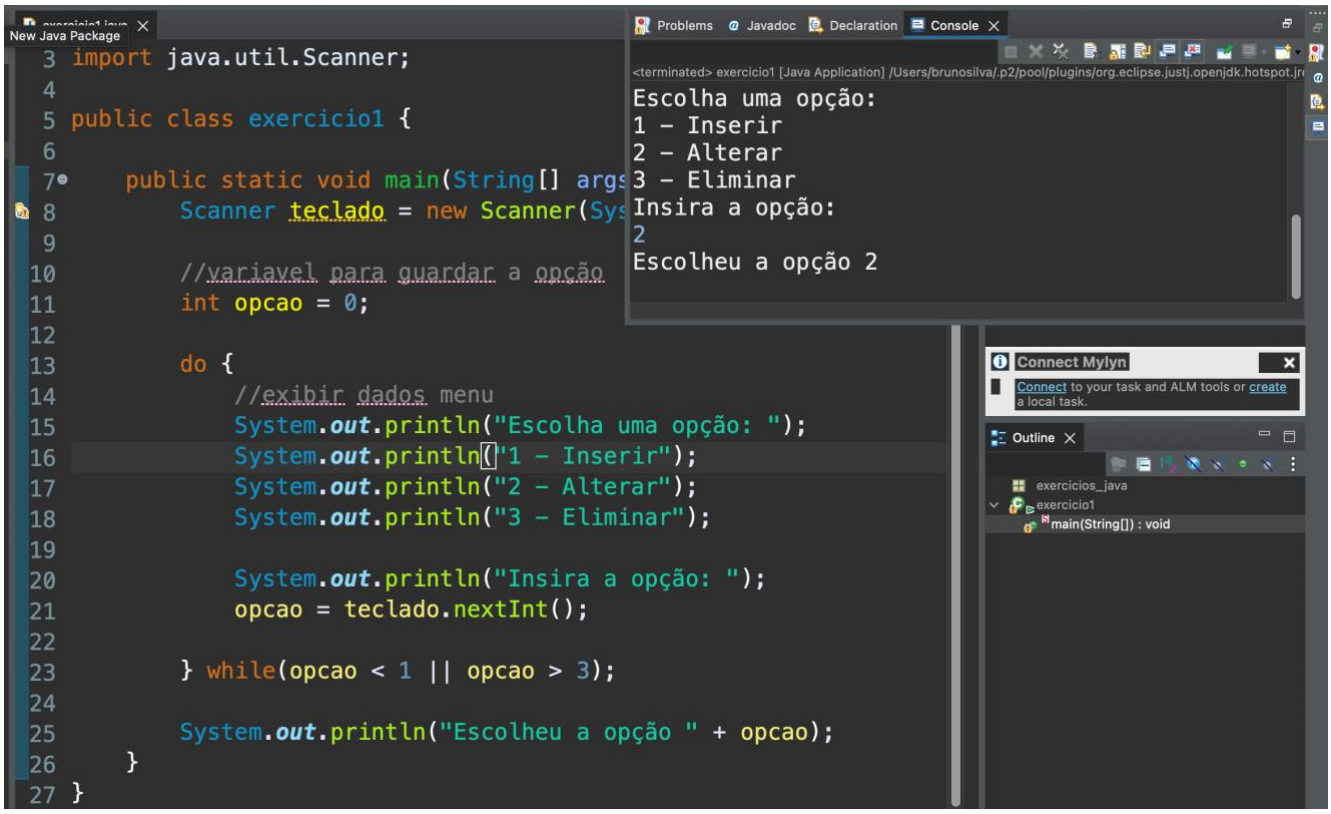


Exemplo para somar todos os números de 1 até 5 (inclusive):

```
exercicio.java
1  package exercicios_java;
2
3  import java.*;
4  import java.util.Scanner;
5
6  public class exercicio {
7
8      public static void main(String[] args) {
9          Scanner teclado = new Scanner (System.in);
10
11         int soma = 0, contador = 0;
12
13         //Somar todas as notas de 1 a 5 (inclusive)
14         do {
15             soma = soma + contador;
16
17             contador++; // contador = contador + 1;
18         }while(contador <= 5);
19
20         System.out.print("Soma total: " + soma);
21     }
22 }
23
```

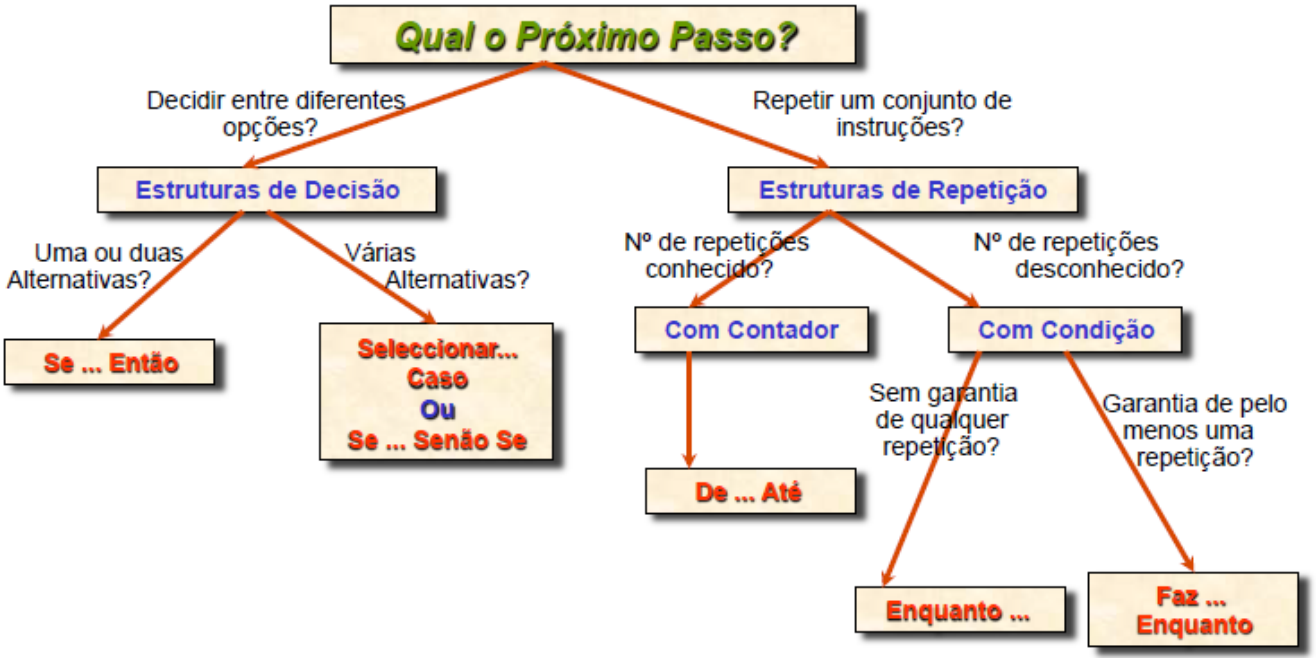
```
Problems  Javadoc  Declaration  Console
<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (05/09/2020, 22:31:42)
Soma total: 15
```

Exemplo para mostrar um menu de escolhas e escolher uma opção:



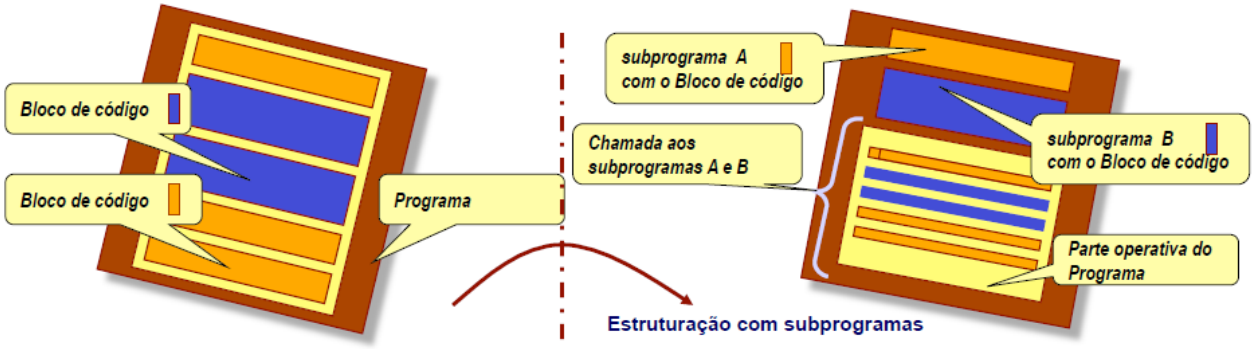
7.8 - Diagrama do mapa das estruturas de controlo

Construção dum algoritmo: Qual a decisão da estrutura a seguir?



Capítulo 8 – Funções e subprogramas - conceitos e utilidades

Um programa corresponde à implementação de uma solução para um determinado problema. Mas, muitas vezes, os problemas são demasiado complexos.



Exemplo de um programa tradicional (<u>sem</u> Subprogramas)

```
public static void main(String[] args) {
    int resultado = 0;

    resultado = 5 + 7;

    if(resultado >= 10) {
        System.out.println("Igual ou superior que 10");
    }
    else {
        System.out.println("Inferior a 10");
    }

    resultado = 9 + 12;

    if(resultado >= 10) {
        System.out.println("Igual ou superior que 10");
    }
    else {
        System.out.println("Inferior a 10");
    }
}
```

Numa análise rápida, reparem que:

- Estamos a repetir a operação de soma e estrutura de decisão;
- Se continuarmos com esta situação***, isto vai ficar muito confuso e repetitivo!***

É neste momento que as funções ajudam e têm uma solução mais simples permitindo assim reduzir trabalho e poupar linhas de código, em vez de repetir funções e procedimentos desnecessariamente.

Para tal, podemos dividir um problema grande em porções mais pequenas, por forma a facilitar a descoberta da solução.

Vantagens:

- Criam-se secções do programa que executam tarefas específicas, com um conjunto de instruções a cumprir muito bem definido;
- Contribuem para a resolução do problema geral e mais complexo.

Como funciona uma função?

O código de uma função ***só é executado quando esta é invocada em alguma parte do programa*** que está de algum modo ligada;

Sempre que uma função é invocada, o programa que a invoca, ***fica "suspenso" temporariamente***.

Em seguida são executadas as instruções presentes na função. Uma vez ***terminada a função*** invocada, ***o programa retorna ao local onde esta foi invocada***.

Vantagens dos Subprogramas:

- ***Clareza de código***: Criam-se secções do programa que executam tarefas específicas, com um conjunto de instruções a cumprir muito bem definido;
- ***Reutilização***: podemos usar o código devolvido em funções mais do que uma vez;
- ***Independência:*** uma função é relativamente independente do código que a chamou. Uma função pode modificar variáveis globais ou ponteiros, mas limitando-se aos dados fornecidos pela chamada de função;

O grande objetivo, será reduzir e evitar repetições de código, fazendo com que tenhamos uma melhor funcionalidade lógica e reutilização do mesmo.

8.1 – Subprogramas: procedimentos, funções e passagem de parâmetros

Um subprograma ***pode (ou não) ter que produzir algum resultado para o exterior*** (para quem lhe pediu que executasse a tarefa a que está destinado), podendo assim ser classificado como:

- ***Procedimento*** (subprograma ***que não retorna*** informação);
- ***Função*** (subprograma que ***retorna*** informação);

Em Java os ***procedimentos*** e ***funções*** chamam-se ***métodos***.

8.1.1 – Procedimentos (com e sem passagem de parâmetros)

Um **Procedimento** é um subprograma que contém um conjunto de instruções a executar, de forma unária e atómica, para cumprir uma determinada tarefa.

- ***O procedimento pode ou não ter de receber dados de entrada*** e executa as instruções que o compõem ***até chegarem ao fim***;
- Não devem esquecer que, ***o número de dados de entrada de um procedimento*** deve ser ***igual a mesma quantidade de parâmetros que foi enviado do programa que invocou o procedimento***.
- Um procedimento ***não retorna valor***, sendo isto identificado em JAVA pela palavra ***void,*** que é colocada ***antes do nome do método*** (sub-programa).

Vamos ver um exemplo prático!

```java
package exercicios_java;

import java.util.Scanner;

public class exercicio1 {

    public static void calcula_dobro(int valor) {
        int res = valor * valor;
        System.out.println("O dobro de número 1 é: " + res);
    }

    public static void main(String[] args) {

        int num1 = 3, num2 = 5, num3 = 8;

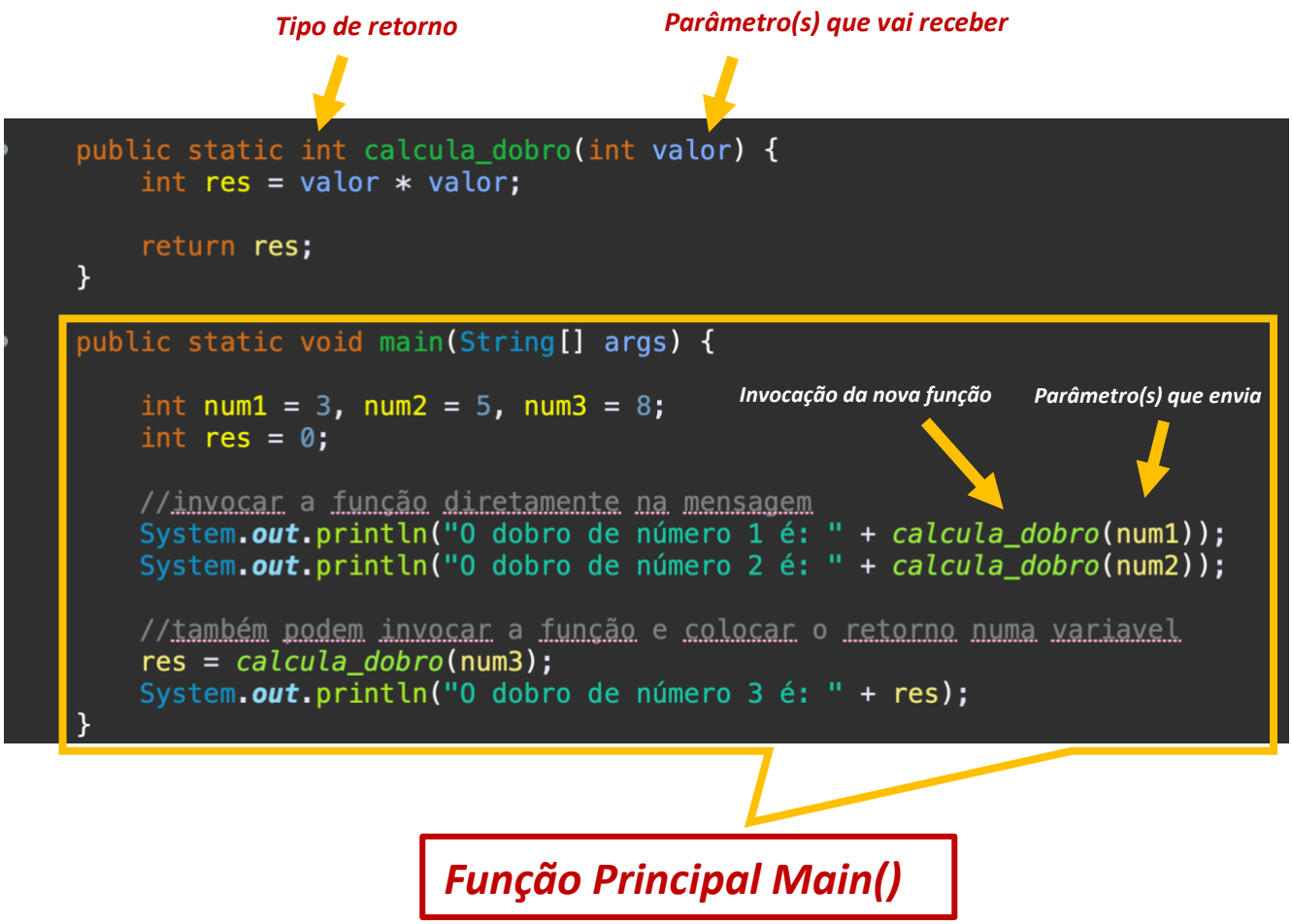
        calcula_dobro(num1);
        calcula_dobro(num2);
        calcula_dobro(num3);
    }
}
```

8.1.2 – Funções (com e sem passagem de parâmetros)

Uma ***função*** é um subprograma idêntico a um procedimento, só que neste caso, uma função ***devolve sempre um valor como resultado da sua execução***.

- Uma função é um método que ***tem um valor de retorno*** (ao contrário do procedimento). A forma de uma função apresentar um valor de retorno é recorrer à instrução ***return***;
- A instrução ***return interrompe a execução do sub-programa*** chamado e o fluxo de execução é devolvido ao programa (ou subprograma) chamante, continuando a execução na instrução seguinte à chamada do subprograma;
- Também não devem esquecer que, o ***número de dados de entrada de uma função*** deve ***ser igual a mesma quantidade de parâmetros que foi enviado do programa que invocou a função***.

Vamos ver um exemplo prático!



Capítulo 9 – Introdução à Programação orientada por objetos

Antes das linguagens orientadas a objetos (anos 70 e 80) as linguagens estruturadas como o ***C*** e o ***Pascal*** tiveram um enorme sucesso.

Inicialmente o paradigma de programação orientado por objetos (***POO***) surgiu por via do software de simulação nos ***anos 60*** (***SIMULA-67***) e mais tarde por via da ***Engenharia de Software***.

É nos ***anos 90*** que a ***POO*** ganhou relevância dominando até hoje na indústria da informática.
Entre as linguagens que seguiram o paradigma da POO está o ***C++, Java***, ***Delphi***, ***C#, VisualBasic,*** entre outras;





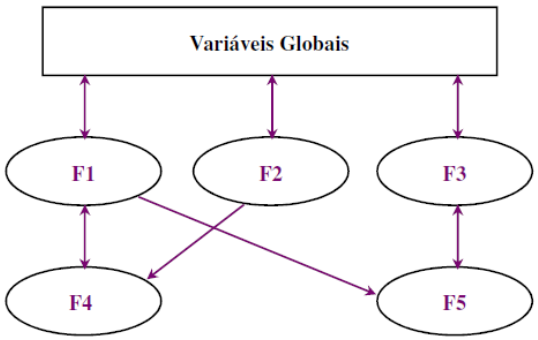


Então, em termos de paradigmas de programação, temos dois tipos de programação que são:

9.1 - Programação estruturada (sequencial)

Nas linguagens estruturadas, um programador pensa numa ou mais estruturas de dados para modelar o problema e em seguida desenvolve um conjunto de funções que atuam sobre estas estruturas de dados, ou seja, um ***pensamento mais voltado para o pensamento da máquina***.

Programa = *Estrutura de Dados + Algoritmos*



Características:

- Muito usada para aprender a programar;

- Funciona bem para pequenos programas, mas quando as dimensões do programa aumentam, começam a aparecer problemas;
- À medida que os programas crescem torna-se mais difícil manter o código;
- O problema que acontece é que todas as operações têm acesso a todos os dados e basta uma pequena alteração num dos módulos pode ter implicações em todo o programa.
- Para programar em ***linguagem estruturada*** usamos variáveis e funções que executam certas regras que são desenvolvidas pelos programadores;

9.2 - Programação Orientada a Objetos (POO)

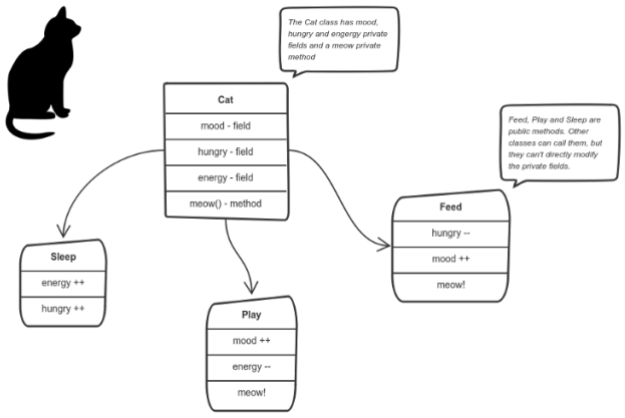
Nas ***linguagens estruturadas*** o problema era modelado em ***estruturas de dados*** e depois eram desenvolvidas ***operações / funções*** que atuavam sobre essas mesmas. O modelo funcionava (***e ainda funciona***) para programas com ***pouca complexidade***, ***mas***, qualquer ***alteração à estrutura*** implica que

- todas as ***funções*** / ***módulos*** ***tenham de ser revistos***;
- existe um ***elevado acoplamento (relacionamento) entre os módulos;***

A ***POO*** vem aliviar esse problema ***ao diminuir o acoplamento*** entre os módulos, onde as estruturas de dados são ***encapsuladas*** dentro dos ***objetos*** e dentro dos objetos continuam a existir funções que atuam sobre as estruturas de dados, ou seja,

defende um pensamento de programação mais voltado ao pensamento natural (humano).

Programa = *Objetos* + *Relações*

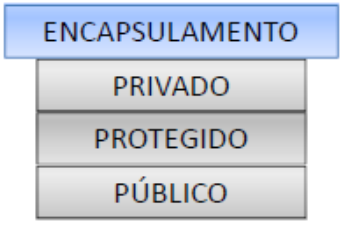


9.3 – Conceitos de Programação Orientada a Objetos

A programação orientada aos objetos assenta em três conceitos básicos fundamentais:

- **Encapsulamento de informação**
- **Composição / Herança**
- **Polimorfismo**

9.3.1 - Encapsulamento de informação

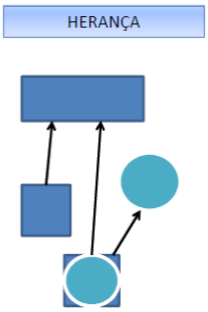


- O Objeto é uma embalagem de ***estados*** (propriedades), ***comportamentos*** (métodos) e tem uma ***identidade***;
- O estado dos objetos é acessível através de operações invocadas por intermédio de mensagens;
- Nem todos os estados e comportamentos podem ser livremente acedidos;

Os modificadores de acesso <u>controlam</u> que parte do programa ***pode ou não utilizar a classe***. Estes podem ser de ***3 tipos***:

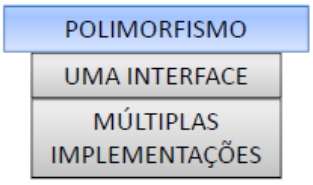
- ***Public*** (público) - é o nível de acesso que os atributos ou métodos podem ser acedidos em qualquer outro ponto do projeto e por outras classes;
- ***Private*** (privado) - é o nível de acesso mais restrito e permite definir que somente a própria classe pode aceder aos atributos ou métodos. Ou seja, nenhuma outra parte do projeto, nem mesmo as classes filhas (heranças), podem aceder a esse atributo ou método;
- ***Protected*** (protegido) – nível de acesso protegido que somente a própria classe e as classes que herdam da mesma é que podem aceder os atributos ou métodos. Dessa forma, ao instanciar a classe os elementos protegidos (protected) não podem ser acedidos diretamente, como ocorre com o public;

9.3.2 - Composição / Herança



- Classes derivadas têm um comportamento específico;
- Herança das propriedades e comportamento protegidos e públicos das classes derivadas;
- ***Todos herdamos características e comportamentos dos nossos pais!***

9.3.3 - Polimorfismo



- A fronteira/interface de acesso ao objeto é separada da própria implementação do objeto;
- Criação de classes com comportamento genérico;
- Um objeto pode pertencer a vários tipos com comportamentos específicos para cada tipo, ***mas tecnicamente diferentes, implementando a filosofia de "uma interface, múltiplos métodos".***

Paradigmas de Programação: Objetos, Classes e Instâncias

- ***Objetos*** – é a conjugação de dados e operações numa entidade;
- ***Classe*** – Descreve os objetos que têm características comuns;
- ***Instância*** – Objeto que é descrito por uma classe. Todas as instâncias de uma mesma classe partilham as mesmas operações e possuem uma estrutura de dados interna comum;

Características:

- **Produtividade** - Construir *software* é mais fácil e barato
- **Robustez** - O *software* é mais fiável (por reutilização e melhores regras)
- **Manutenção** - Muito mais fácil executar as melhorias e encontrar os "bugs"
- **Extensibilidade** - Muito mais fácil acomodar novos requisitos
- **Modularidade** - Abstração e encobrimento da informação

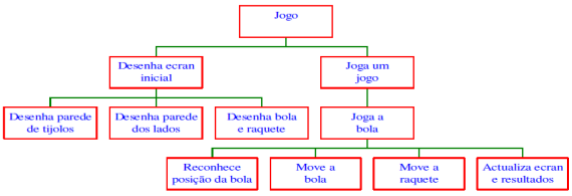
9.5 - Paradigmas de Programação

Em termos de paradigmas de programação, temos dois tipos de programação que são:

- ***Programação Estruturada (Sequencial)***
- ***Programação Orientada a Objetos*** **(POO)**

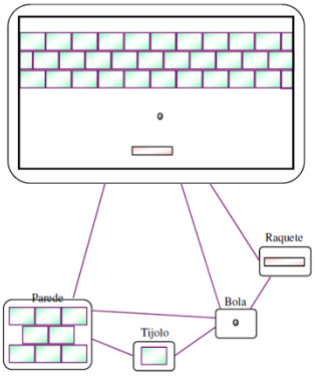
Solução Tradicional

- Identificar as funções
- Identificar as relações entre as funções
- Identificar as características das funções



Solução Orientada por Objectos

- Identificar os objectos
- Identificar as relações entre os objectos
- Identificar as características dos objectos



Capítulo 10 – Classes

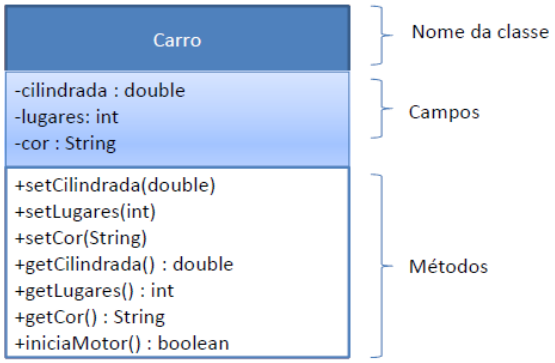
10.1 – Introdução as classes

Uma ***classe*** é um conceito que define uma ***estrutura para a criação de objeto***. Uma classe ***define um objeto com base*** nas suas propriedades (os campos ou variáveis) e as suas ações ou métodos que definem o seu comportamento (funções).

A classe é especificada pelo programador, definindo a interface e a implementação dos objetos

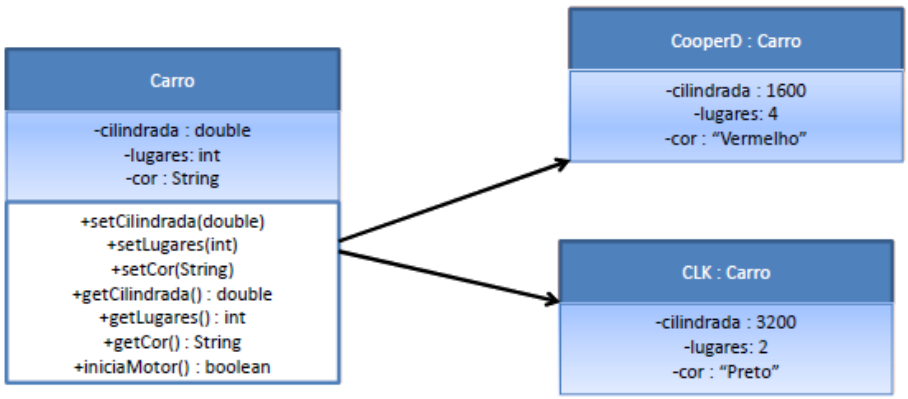
Nesta classe temos:

- a representação abstrata de um Carro (***classe***);
- que tem características (***campos***);
- e apresenta comportamentos (***métodos***);

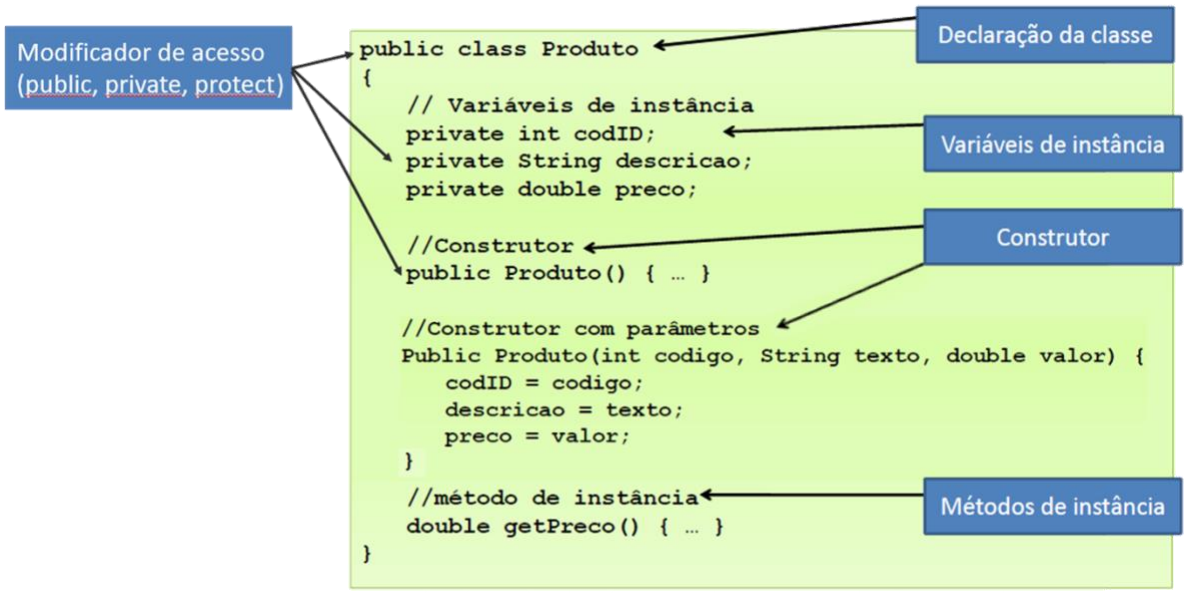


Como podemos analisar:

- A classe é um template a partir do qual "nasce" o objeto;
- O objeto é uma instância da classe, dá nome à classe e valores aos campos
- O objeto tem uma identidade e apesar do seu estado sofrer alterações, a identidade não é alterada!
- Se pintar o Mercedes CLK de Amarelo, continua a ser um CLK!



10.2 – Definição de Classe



10.2.1 – Declaração de uma Classe e Modificadores de Acesso

- A ***declaração de classes*** em JAVA ***começa com letra Maiúscula***;
- O ficheiro onde está a definição da ***classe tem de ter o mesmo nome da classe*** e com ***extensão*** ".java" (como já referido anteriormente);
- As palavras ***public*** / ***private*** são ***modificadores de acesso*** que ***controlam*** que parte do programa ***pode ou não utilizar a classe***.
  - ***Exemplo***: Uma classe ***pública*** pode ser utilizada em todo o lugar;

- ***Os modificadores de acesso controlam*** que parte do programa ***pode ou não utilizar a classe***. Estes podem ser de ***3 tipos***:
  - ***Public*** (público) - é o nível de acesso que os atributos ou métodos podem ser acedidos em qualquer outro ponto do projeto e por outras classes;
  - ***Private*** (privado) - é o nível de acesso mais restrito e permite definir que somente a própria classe pode aceder aos atributos ou métodos. Ou seja, nenhuma outra parte do projeto, nem mesmo as classes filhas (heranças), podem aceder a esse atributo ou método;
  - ***Protected*** (protegido) – nível de acesso protegido que somente a própria classe e as classes que herdam da mesma é que podem aceder os atributos ou métodos. Dessa forma, ao instanciar a classe os elementos protegidos (protected) não podem ser acedidos diretamente, como ocorre com o public;

10.2.2 – Variáveis de instância de uma Classe

- A ***declaração de variáveis de instância*** em JAVA é feita ***dentro do bloco da definição de classe*** e ***nunca*** dentro de ***métodos*** ou ***construtores***;
- Para ***prevenir*** que outras classes ***possam aceder às variáveis de instância***, devemos utilizar a palavra reservada ***private*** antes da declaração

```
public class Produto {

    private int codID;
    private  String descricao;
    private double preco;
```

10.2.3 – Construtores de uma classe

- O ***construtor é um método especial***, por norma é o ***primeiro "método"*** que se ***constrói dentro da classe***. Não são métodos de instância porque apenas são utilizados para criar instâncias de uma dada classe;
- Os ***construtores*** têm de utilizar o mesmo nome da ***classe e podem ser:***
    - ***Construtor com parâmetros:*** Igual ao anterior, mas ***vamos ter parâmetros de entrada no construtor*** e vai ***receber e passar a informação dos parâmetros de entrada*** para as ***variáveis de instância*** da classe

```
    //Construtor com parâmetros
    public Produto(int codID, String descricao, double preco) {
        this.codID = codID;
        this.descricao = descricao;
        this.preco = preco;
    }
```

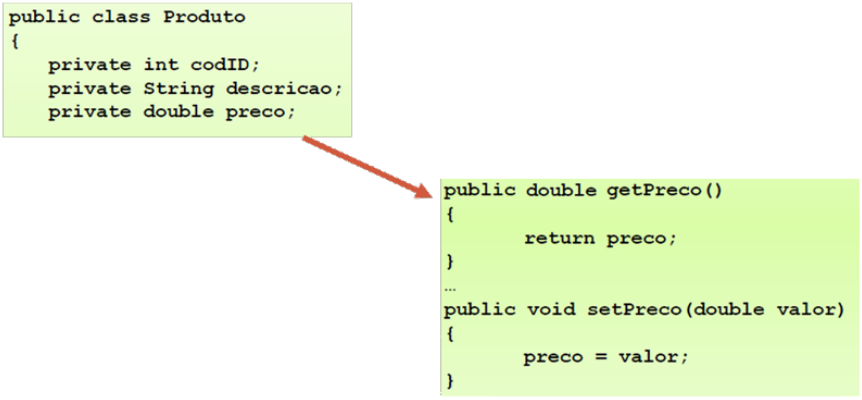
    - ***Se não especificarmos um construtor***, o JAVA ***cria um construtor por defeito*** que ***inicializa todas as variáveis de instância*** para ***valores por defeito*** (numéricos a 0, boolean a false/true e objetos a null)

    - Utilizamos a palavra reservada ***this*** para ***identificar*** o que é a **variável de instância do parâmetro** da informação **do parâmetro de entrada do construtor**;
        - A palavra reservada ***this*** permite ter acesso a partir do código do objeto às ***variáveis***, ***métodos*** e ***construtores*** do próprio objeto.

- A palavra reservada ***this*** é útil por exemplo, quando as variáveis dos parâmetros de entrada de um método são coincidentes com as variáveis de instância.

10.2.4 – Métodos de instância de uma Classe

- ***Métodos de instância*** são responsáveis ***pelo comportamento das instâncias/objetos quando recebem mensagens provenientes de alguma parte do programa***;
- ***Para que outras classes tenham acesso ao método*** é preciso designar o método com acesso público com a ***palavra reservada public***;
- ***Um método que não retorne qualquer tipo de dado*** deve ser colocada a palavra reservada ***void*** no ***tipo de dados de retorno***;
- É boa prática na codificação das classes JAVA construir métodos de instância ***set*** e ***get*** para ***cada variável de instância*** que existe definida na classe;




```
public int getCodID() {
    return this.codID;
}

public void setCodID(int codID) {
    this.codID = codID; // codID = codID ... não faz sentido!
}

public String getDescricao() {
    return descricao;
}

public void setDescricao(String descricao) {
    this.descricao = descricao;
}

public double getPreco() {
    return preco;
}

public void setPreco(double valor) {
    preco = valor;
}

public String MostraMensagem() {
    return new String("ID: " + codID + ", Descrição: " + descricao + ", Preco: " + preco);
}
```

10.2.5 – Overload de Métodos de uma Classe

- ***Sobrecarga semântica de métodos (method overloading)*** consiste na possibilidade de ***numa mesma classe existirem dois ou mais métodos com o mesmo nome***, desde que ***difiram na assinatura (dados que recebem nos parâmetros)***;

```
double aumentaPreco(int codigo, double aumento) { … }

double aumentaPreco(double aumento) { … }

double aumentaPreco(int aumento) { … }
```

10.2.6 – Classes e as Palavras reservadas: new e this

- A palavra reservada ***new*** trata-se de um ***operador que vai instanciar a classe***, ***alocando para isso espaço em memória para o objeto*** e retorna a referência/endereço desse objeto.
- ***Exemplo***: Vamos criar uma instância de uma classe, isto é, um novo objeto.

Empregado eng1 = ***Empregado***("Magda Dionísio", 25);

- O operador ***new*** também tem a responsabilidade de:
  - ***invocar o construtor da classe;***
  - ***para inicializar um objeto;***

```
//construtor sem parâmetros
Produto meuProduto1 = new Produto();

//construtor com recurso a parâmetros
Produto meuProduto2 = new Produto(1, "iPAD", 499.99);

//aplicado aos arrays
String meuArrayTexto = new String[21][10]; // 21 x 10
```

10.2.7 – Definição de Classe – Exemplo de uma classe com métodos

```
public class Produto {

    private int codID;
    private  String descricao;
    private double preco;

    //Construtor
    public Produto() {
        codID = 0;
        descricao = "nenhum";
        preco = 0.0;
    }

    //Construtor com parâmetros
    public Produto(int codID, String descricao, double preco) {
        this.codID = codID;
        this.descricao = descricao;
        this.preco = preco;
    }

    public int getCodID() {
        return this.codID;
    }

    public void setCodID(int codID) {
        this.codID = codID; // codID = codID ... não faz sentido!
    }

    public String getDescricao() {
        return descricao;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public double getPreco() {
        return preco;
    }

    public void setPreco(double valor) {
        preco = valor;
    }

    public String MostraMensagem() {
        return new String("ID: " + codID + ", Descrição: " + descricao + ", Preco: " + preco);
    }
```

10.2.8 – Exemplo de criação de um novo objeto a partir de uma Classe

- Em qualquer local do programa que se pretenda ter acesso às variáveis privadas da instância, deve ser feito por intermédio dos métodos próprios;

```
public static void main(String[] args) {
    Produto myProduto1 = new Produto(1, "PDA", 300.12);
    //obter dados
    System.out.println("\n\nDescricao do produto1 \n" +
            "ID: " + myProduto1.getCodID() +
            "\nDescrição: " + myProduto1.getDescricao() +
            "\nValor: " + myProduto1.getPreco());
}
```

10.2.9 – Exemplo de criação de um novo objeto a partir de uma Classe

```
public class Produto {

    public int codID;
    public  String descricao;
    public double preco;

    //Construtor
    public Produto() {
        codID = 0;
        descricao = "nenhum";
        preco = 0.0;
    }
}
```

```
public static void main(String[] args) {
    Produto myProduto1 = new Produto(1, "PDA", 300.12);
    //obter dados
    System.out.println("Descricao do produto1 \n" +
            "ID: " + myProduto1.codID +
            "\nDescrição: " + myProduto1.descricao +
            "\nValor: " + myProduto1.preco);
}
```

O que não está de acordo com as boas práticas de POO ?

Solução

- As variáveis de instância devem ***ser protegidas*** de acesso directo (***private***);
- As classes devem providenciar ***métodos públicos que se encarregam de aceder às variáveis de instância***;

```
private int codID;
private  String descricao;
private double preco;

public int getCodID() {
    return codID;
}

public void setCodID(int Id) {
    codID = Id ;
}
public String getDescricao() {
    return descricao;
}

public void setDescricao(String texto) {
    descricao = texto;
}
```

11 - Heranças

11.1 – Constituição e Manipulação Entre Classes

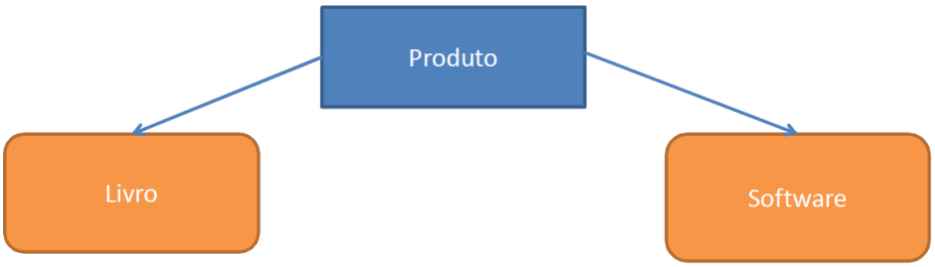
A herança é dos mecanismos mais importantes na PPO pois permite ***criar classes*** com ***base em outras classes***. Permite ***definir uma hierarquia de especialização de classes***, da classe mais genérica para a mais detalhada e ***reutilizando*** variáveis e métodos ao longo da especialização das classes;

Para utilizar os mecanismos de heranças nas classes ***é preciso saber classificar as classes.*** No JAVA os mecanismos de herança são de hierarquia simples, isto é, uma classe ***só pode herdar diretamente*** de uma ***única classe***;

- Uma classe que serve de base a outra classe é designada de ***classe base*** ou ***superclasse***;
- Uma classe que herda de outra classe é uma ***classe derivada*** ou ***subclasse***;

Uma classe derivada / subclasse estende a definição da classe base / superclasse ao adicionar campos, métodos e construtores à superclasse;

Podemos definir uma subclasse relativamente à superclasse como “é-do-tipo” ou “é-um”;

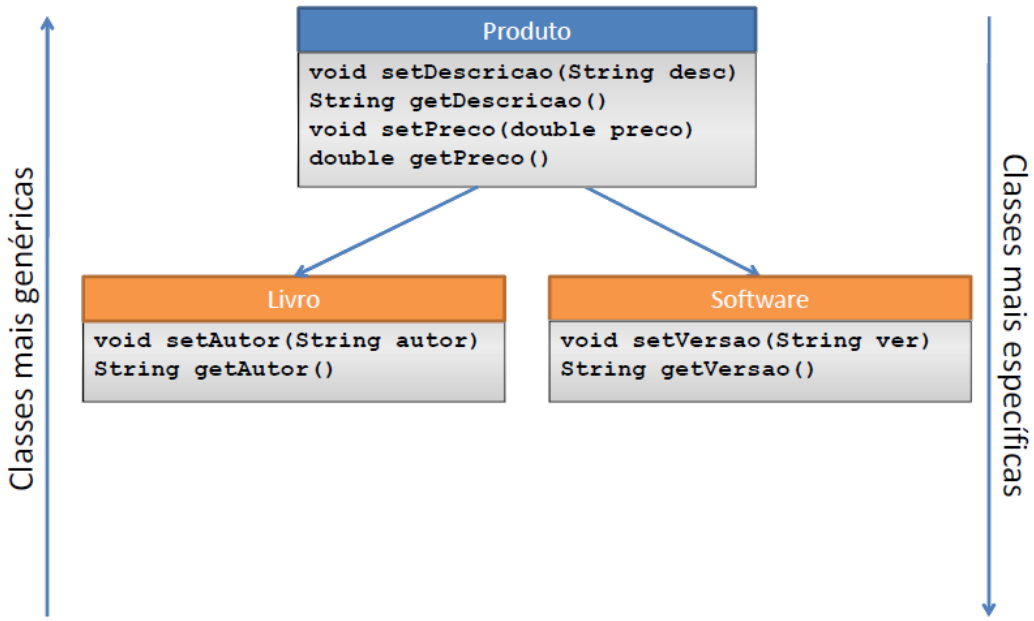


A classe Livro ou Software “é-um”/”é-do-tipo” classe Produto…

No exemplo anterior, existe uma ***superclasse*** “Produto” que vai servir de ***base à construção de diferentes classes de produtos***;

- As ***classes*** ***Livro e Software*** têm como ***ponto de partida*** a classe “Produto”.
- As ***subclasses*** ***Livro e Software herdam*** todos os métodos públicos da classe “Produto”;
- A ***subclasse*** Livro e Software definem métodos específicos de cada classe, que ***são únicos para cada classe***.

Exemplo: a classe Livro especifica um método para atribuir Autor e a classe Software especifica um método para definir a versão do software);



Para ***indicar que uma classe*** é ***derivada*** de ***outra classe base*** usamos a notação / instrução:

class <nome_subclasse> ***extends*** <nome_classe_base> {}

Exemplo: ***class*** Livro ***extends*** Produto { … }

Para ***chamar o construtor da classe base***:

super (<lista_argumentos>);

Exemplo: ***super()***;

Para ***chamar um método da classe base***:

super.nome_método (<lista_argumentos>);

Exemplo: ***super.MostraMensagem();***

- ***NOTA IMPORTANTE***: Podemos fazer o ***override*** (sobreescrita) de métodos definidos na classe base ao codificarmos na classe derivada com a mesma assinatura;

11.2 - HERANÇA – criar uma classe derivada de Produto – Livro

```
public class Livro extends Produto {

    //Definir as variáveis de instancia
    private String autor;

    //Definir o construtor sem parametros
    public Livro() {
        super();
        autor = "";
    }

    //Definir o construtor com parametros
    public Livro(int codigo, String texto, double preco, String autor, int nPaginas) {
        super(codigo, texto, preco);
        this.autor = autor;
    }

    //Metodos de instancia (gets e sets do autor)
    public String getAutor() {
        return this.autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    //Mostra Mensagem dos novos valores
    @Override
    public String MostraMensagem() {
        return new String(super.MostraMensagem() + "Autor: " + autor);
    }
}
```

11.3 - HERANÇA – criar uma classe derivada de Produto - Software

```
public class Software extends Produto {

    //Definir as variáveis de instancia
    private int versao;

    //Definir o construtor sem parametros
    public Software() {
        super();
        versao = 0;
    }

    //Definir o construtor com parametros
    public Software(int codigo, String texto, double preco, int versao) {
        super(codigo, texto, preco);
        this.versao = versao;
    }

    //Metodos de instancia (gets e sets da versao)
    public int getversao() {
        return this.versao;
    }

    public void setversao(int versao) {
        this.versao = versao;
    }

    //Mostra Mensagem dos novos valores
    @Override
    public String MostraMensagem() {
        return new String(super.MostraMensagem() + ", Versão: " + versao);
    }
}
```

12 – Interfaces

12.1 – Estruturas e comportamentos nas Heranças

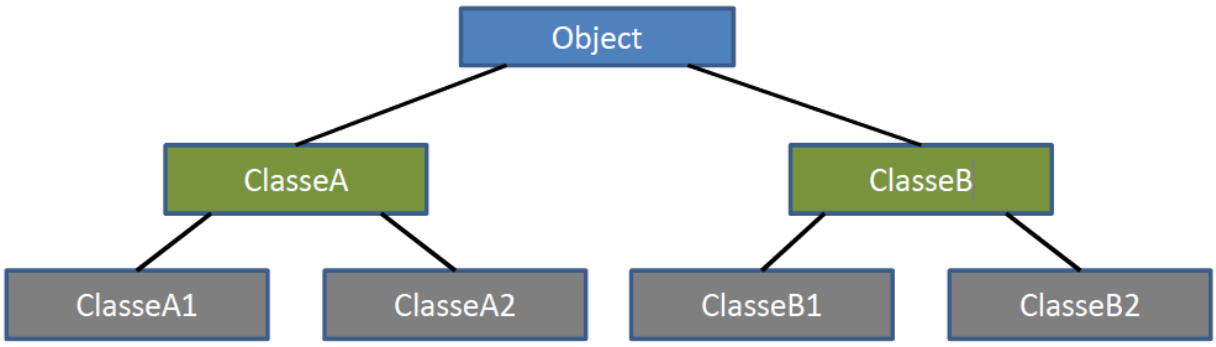
Em programação por objetos, as ***classes definem a estrutura e comportamento*** das suas instâncias. Por outro lado, as classes fazem parte de uma hierarquia que as relaciona entre si, cujo mecanismo de herança faz com que seja a hierarquia de especialização;

Uma classe ao declarar que faz o ***extends*** de ***outra classe***, é o mesmo que dizer que se torna ***numa subclasse da primeira.***

Em Java ***a primeira classe*** é a ***única superclasse possível*** para a segunda classe. A herança em Java é simples. Característico da Herança, ao ***descermos na hierarquia das classes***, temos classes cada vez mais ***especificas e detalhadas***;

O conceito de classe e subclasse estão associados a uma filosofia muito importante na conceção e construção de software que se baseia na procura de reutilização de componentes já existentes.

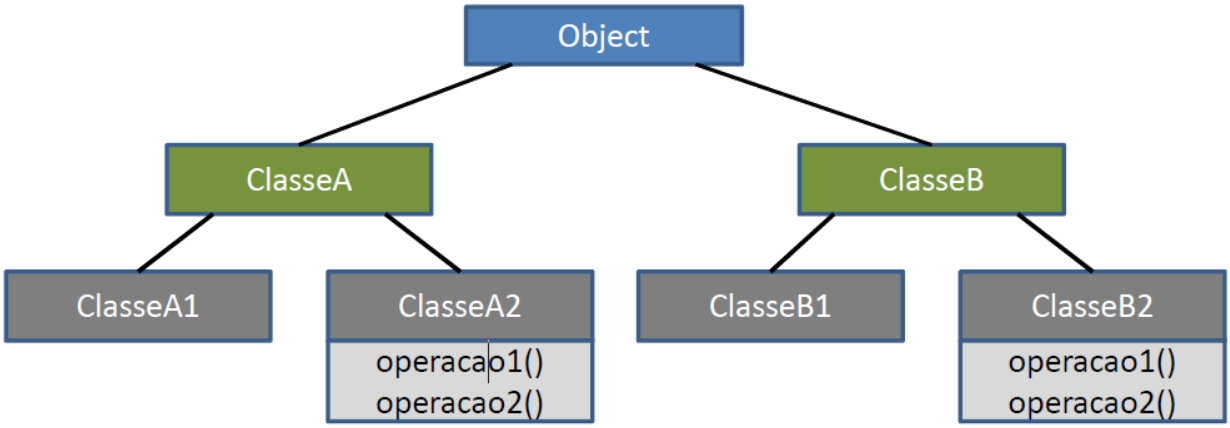
Consideremos a estrutura hierárquica das classes:



- Sabemos que a ***ClasseA1*** e ***ClasseA2 partilham*** algo em comum devido a ***derivarem de ClasseA***, o mesmo se passa entre a ***ClasseB1*** e ***ClasseB2 partilham*** algo em comum por ***derivarem de ClasseB***;

- Devido à ***posição hierárquica ClasseB1 e ClasseB2*** possuem uma ***"linguagem" distinta da ClasseA1 e ClasseA2***. As classes têm apenas de comum o que é definido por ***Object***;

Imaginem que ***segundo a hierarquia definida***, é necessário que ***apenas*** as ***classes ClasseA2 e ClasseB2 implementem os métodos operacao1() e operacao2()***:



- Recorrendo ***unicamente à herança***, para satisfazer este requisito a classe ***Object*** teria de ***especificar os métodos operacao1() e operacao2()…***
- Se ***Object*** definir os ***métodos operacao1() e operacao2()*** estes métodos ***ficam disponíveis em todas as classes da hierarquia.***
- ***Esta solução é inaceitável.***

12.2 – Interfaces

A solução anterior é inaceitável ***porque para além de não responder aos requisitos***, do ponto de vista da programação ***não é uma solução estável, genérica e extensível.***

Pela solução anterior (***recorrendo unicamente à herança***) sempre que pretendemos que classes mais distantes na hierarquia e não relacionadas ***implementem determinado comportamento***, é necessário alterar uma classe mais genérica ***afetando todas as subclasses desta classe***.

Em outras linguagens de programação orientadas a objectos ***que permitam herança múltipla este problema não é colocado***. Em Java, para responder a este tipo de requisitos sobre certas classes, sem recorrer unicamente à herança, ***existem as interfaces***.

A interface ***define um conjunto de métodos públicos que pode ser implementado por qualquer classe em qualquer ponto da hierarquia***. A interface em si ***não define qualquer código para implementar os métodos***, apenas define a ***assinatura do método***.

As classes que implementem uma determinada interface têm de ***providenciar uma implementação para cada método definido pela interface***.

A interface também pode ***definir variáveis constantes*** que ficam disponíveis ***às classes que implementam a interface***. As interfaces especificam um tipo de dados (métodos abstratos a implementar), e qualquer classe que implemente a interface passa a ser compatível com o tipo de dados da interface

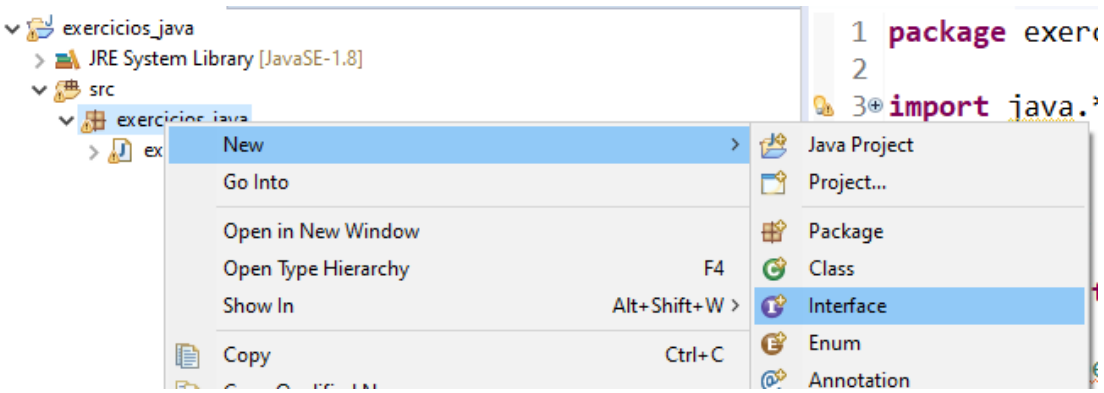
Como boa regra de programação, o ***nome do ficheiro de uma interface*** deve começar pelo a ***letra I (i maiúsculo)***, ***seguido do nome***;

Definição de uma interface "***ILeitorCd***" com um método ***playCD***():

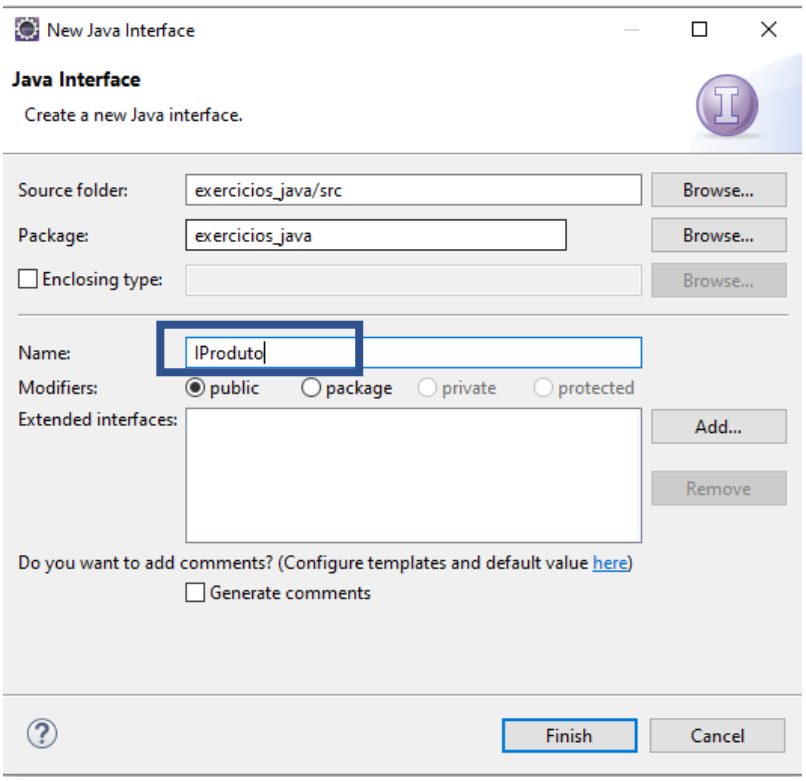
```
public interface ILeitorCD
{
    public void playCD();
}
```

12.3 – Criação de uma Interface

Para criar um ficheiro do tipo interface, basta clicar no botão direito do rato em cima do package -» ***New*** -» ***Interface***:



- E como referido anteriormente, o nome do ficheiro de uma interface deve começar pelo a letra I (i maiúsculo) seguido do nome;



12.4 – Tipos de Assinaturas na Interface

Dentro das interfaces existem somente assinaturas de métodos e propriedades, cabendo à classe que a utilizará realizar a implementação das assinaturas, dando comportamentos práticos aos métodos.

Abaixo é possível ver um exemplo de uma interface chamada ***ILeitorCD*** com três assinaturas de métodos que vão ter de ser implementados pelas classes referentes à implementação da interface.

```
public interface ILeitorCD {

    public void playCD();

    public int Contador_equipamentos();

    public String MostraMensagem();

}
```

Reparem que, podem utilizar várias assinaturas dentro de uma interface, e que vai de encontro com as necessidades da mesma.

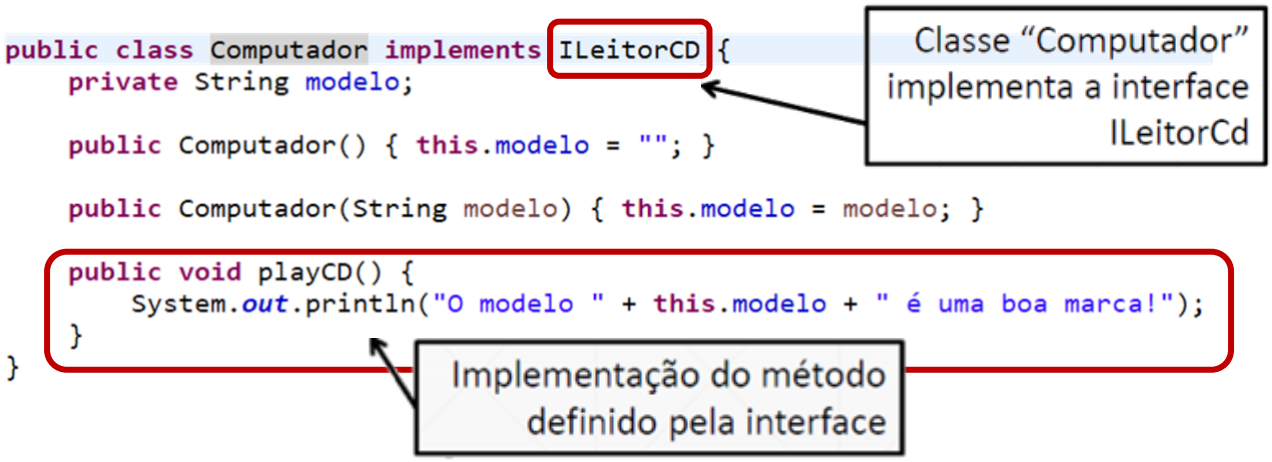
Resumindo, uma interface ***é como um contrato a ser seguido por um conjunto de classes que implementem o mesmo***. Este contrato pode ser entendido com a declaração do comportamento de um conjunto de classes.

As Interfaces possuem somente a declaração dos métodos, ou seja, o tipo de retorno, o nome do método, seus parâmetros e as exceções lançadas pelo método.

A implementação dos métodos será feita pelas ***classes que implementam*** a interface.

12.5 – Exemplo de interfaces

Uma ***classe*** "***Computador***" implementa a interface ***ILeitorCD***:



Utilização da classe "Computador" e do ***método definido na interface:***

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

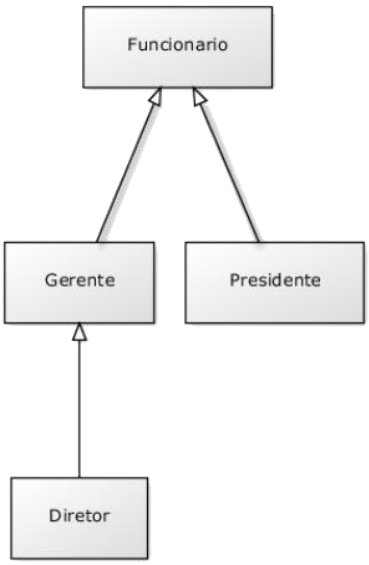
    Computador comp1 = new Computador("HP");
    comp1.playCD();

    Computador comp2 = new Computador("Toshiba");
    comp2.playCD();
}
```

13 – Classes Abstratas

As ***classes abstractas*** são classes que ***não podem ser instanciadas*** (ao contrário das classes normais), ou seja, ***não podemos criar um objeto de uma classe Abstrata.***

Imagem o seguinte caso: numa empresa temos Funcionários. Esses funcionários podem ser do tipo Diretores, Gerentes, Secretários, etc, ou seja, estamos na presença de uma herança.



A ***classe Funcionario*** apenas idealiza um tipo de funcionário.

```
public class Funcionario {

    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros atributos e métodos comuns a todos Funcionarios

}
```

Infelizmente, esta situação é inadmissível pois só temos um ***objeto apenas do tipo Funcionario.***

O objetivo é poder descrever os vários tipos de funcionários e poder haver uma herança entre classes.

Para evitar que haja uma invocação direta na classe dos funcionários, vamos considerar esta classe ***como uma classe abstrata*** (do qual não pode ser instanciada como um objeto).

Para tal vamos usar a palavra ***abstract*** entre o ***public*** (tipo de modificador) e ***o nome da classe***.

```
public abstract class Funcionario {

    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros atributos e métodos comuns a todos Funcionarios

}
```

Caso a classe seja instanciada, o mesmo irá dar um erro:

```
Funcionario f = new Funcionario(); // não compila!!!
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Cannot instantiate the type Funcionario

	at br.com.caelum.empresa.TestaFuncionario.main(TestaFuncionario.java:5)
```

Agora se o método ***getBonificacao()*** não fosse reescrito nas outras classes para além do funcionário, ele seria herdado da classe mãe, fazendo com que ***devolvesse o salário mais 20%.***

```
public abstract class Funcionario {

    protected double salario;

    public double getBonificacao() {
        return this.salario * 1.2;
    }

    // outros atributos e métodos comuns a todos Funcionarios

}
```

O objetivo é fazer com que cada classe que derive de ***funcionário*** tenha um ***método diferente de bonificação***, isto é, reescreva o método ***getBonificacao()*** mas com regras diferentes.

Existe um recurso em Java do qual em uma classe abstrata, podemos escrever que determinado método será sempre escrito pelas classes filhas. Isto é, ***um método abstrato***.

Ou seja, ***todas as classes filhas*** que ***não forem abstratas***, devem ***reescrever*** esse método ***ou não vão funcionar***.

É como herdar a responsabilidade de alguém ter aquele método, pois o método da classe funcionário nunca vai ser chamado, logo, vai cair em uma das classes filhas.

Por isso vamos reescrever a classe do funcionário e dizer que este terá um método abstrato ***getBonificacao(), mas só as classes filhas é*** que definem o tipo de bonificação a herdar.

Classe Funcionário:

```
public abstract class Funcionario {

    public abstract double getBonificacao();

    // outros atributos e métodos

}
```

Nas classes filhas vamos definir o método ***getBonificacao()*** e dizer qual o tipo de bonificação que o gerente, ou outro tipo de funcionário é que vai receber:

Classe Gerente:

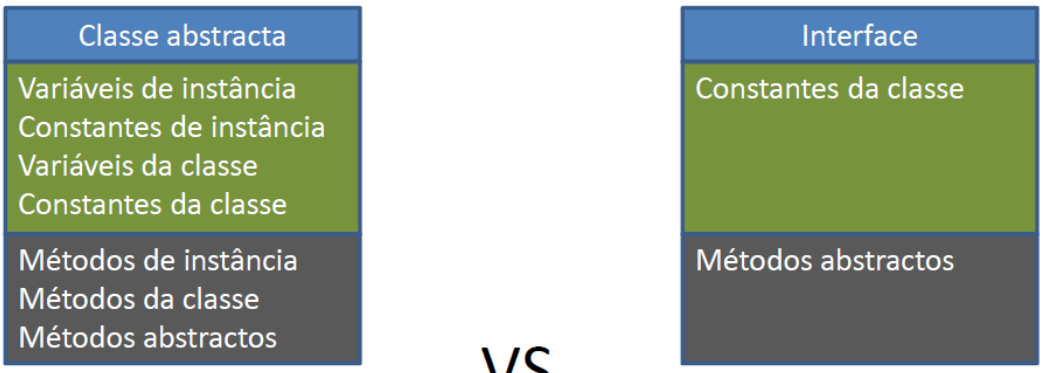
```
public class Gerente extends Funcionario {

    public double getBonificacao() {
        return this.salario * 1.4 + 1000;
    }
}
```

Resumo classes abstratas:

- Classes abstratas não podem ser instanciadas, ou seja, elas servem apenas para que outras classes usem a mesma como modelo (herdar os atributos/propriedades e métodos).
- Podem ter métodos abstratos ou não abstratos.

- Os métodos abstratos não podem ter corpo, ou seja, deve-se declarar apenas a assinatura do método e eles obrigatoriamente terão que ser implementados na classe filha (a classe que herda).




```
public abstract class LeitorCD
{
    public abstract void playCD();
}
```

```
public interface ILeitorCD
{
    public abstract void playCD();
}
```

14 – Exceções

O JAVA, como linguagem de programação por objetos é robusta, devido essencialmente a três fatores:

- sistema de tipos dados (fortemente tipificado);
- modelo de memória sem apontadores e sistema automático de gestão de memória (Garbage Collection);
- sistema de captura e tratamento de exceções;

As excepções são um mecanismo para detetar e tratar erros em tempo de execução real, ou seja, são anomalias que ocorrem durante a execução do programa.



Uma exceção representa uma situação anormal que tem que ser tratada nalgum ponto do código. A partir do momento que é ativada, aborta a execução normal do programa até existir forma de a tratar ou, em último caso, terminar a execução de todo o programa. Podem ser causadas pelo utilizador, pela lógica do programa ou por erros do sistema tais como:

- Pode esgotar-se o espaço em disco numa operação de escrita em ficheiro
- O acesso à rede pode ficar indisponível
- Falta de memória
- ***...***

14.1 – Construção do Bloco de Excepções Try-Catch

O tratamento de erros por exceções assenta no conceito dos blocos ***try-catch***. Aplicamos esta programação defensiva, para que o programa esteja preparado para correr código com operações sensíveis e que eventualmente possam gerar/ocorrer exceções (erros de processamento no código que o programa não conseguiu processar/tratar).

Exemplo da sintaxe try/catch/finally

```
try
{
    // código que pode provocar o lançamento de excepções
    MetodoA();
    MetodoB();
}
catch (ExcepcaoMaisEspecifica myEx1)
{
    // código encarregue de manipular a excepção
}
catch (ExcepcaoMenosEspecifica myEx2)
{
    // código encarregue de manipular a excepção
}
finally
{
    // código que é executado tenha ou não existido uma excepção
}
```

Analise do código:

- O código que pode provocar erro deve ficar entre as instruções ***TRY***.
- A instrução ***CATCH*** deve conter a exceção especifica que vai ser executada como resposta da exceção.
- A secção ***FINALLY*** deve implementar processos de “limpeza” ou para finalizar algo.
- As secções ***FINALLY*** é opcional.

14.2 – Exemplo Prático da Construção do Bloco de Exceções Try-Catch

- ***Vamos considerar o seguinte exemplo***: É pretendido dividir 2 números, em que o 1º número é inteiro e o 2º número tem o valor 0.
- Ora, não podemos dividir algo por 0. Se tentarem fazer o mesmo em programação, será gerada uma exceção. Para este caso específico seria gerada uma exceção do tipo:

ArithmeticException

(exceção específica para erros de operações aritméticas)

```
1  package exercicios_java;
2
3  import java.*;
5
6
7  public class exercicio {
8
9      public static void main(String[] args) {
10
11         int num1 = 96, num2 = 0;
12         float result = 0;
13
14         result = num1 / num2;
15
16         System.out.println("resultado: " + result);
17     }
18 }
19
```

Problems @ Javadoc Declaration Console

```
<terminated> exercicio [Java Application] C:\Program Files\Java\jdk1.8.0_212\bin\javaw.exe (28/09/2020, 19:26:07)
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exercicios_java.exercicio.main(exercicio.java:14)
```

- Para compor este tipo de problemas (exceções), vamos ter de criar um tratamento para exceção, para que este esteja preparado para lidar com estas situações.

- Como podemos verificar o código da divisão é a situação mais crítica, pois não podemos dividir nada por 0. Então uma vez que já avaliamos a zona crítica do código vamos colocar o tratamento da exceção.

```java
public class exercicio {

    public static void main(String[] args) {

        int num1 = 96, num2 = 0;
        float result = 0;

        try {
            result = num1 / num2;
            System.out.println("resultado: " + result);
        }
        catch (Exception e) {
            System.out.println("Erro: " + e);
        }
    }
}
```

Na imagem anterior podemos ver que é usada a palavra Exception para tratar da exceção. Mas, este tipo de tratamento é muito genérico.

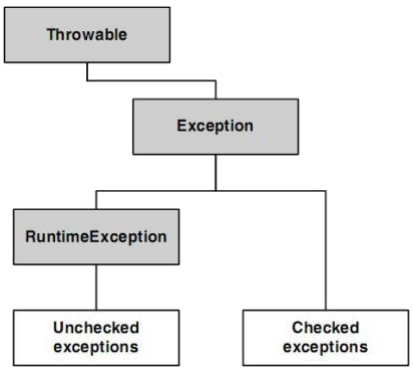
Felizmente o java já tem vários tipos de exceções e indicadas para vários casos. Mas antes disso vamos ver de onde é que as exceções derivam.

14.3 – Tratamento de Excepções e a Classe Throwable

A classe Exception deriva da classe ***Throwable***;

Uma excepção é um objecto da classe **Exception** ou de uma das suas subclasses;

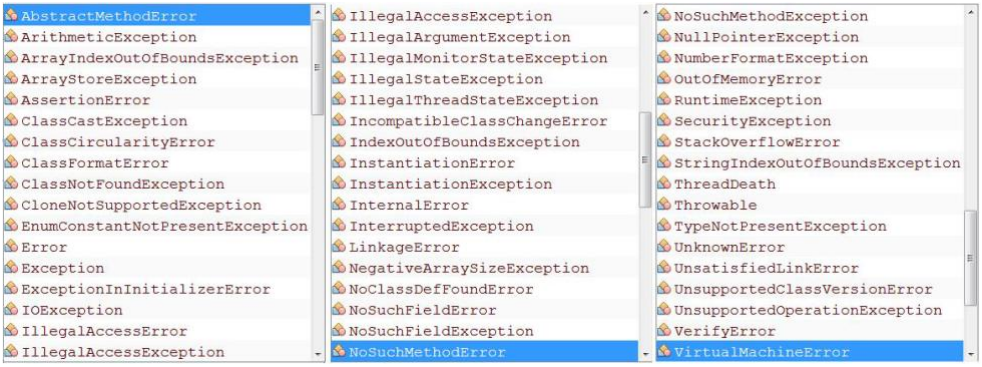
As **Unchecked exceptions**, se não forem previstas pelo programador, vão levar a que o programa termine abruptamente.



Já as exceções do tipo ***checked exceptions***, que derivam diretamente de ***Exception***, o compilador obriga que as exceções sejam tratadas.

Aqui as exceções são situações que estão fora do controlo do programador, como falha de ficheiros, falta de conectividade de rede…

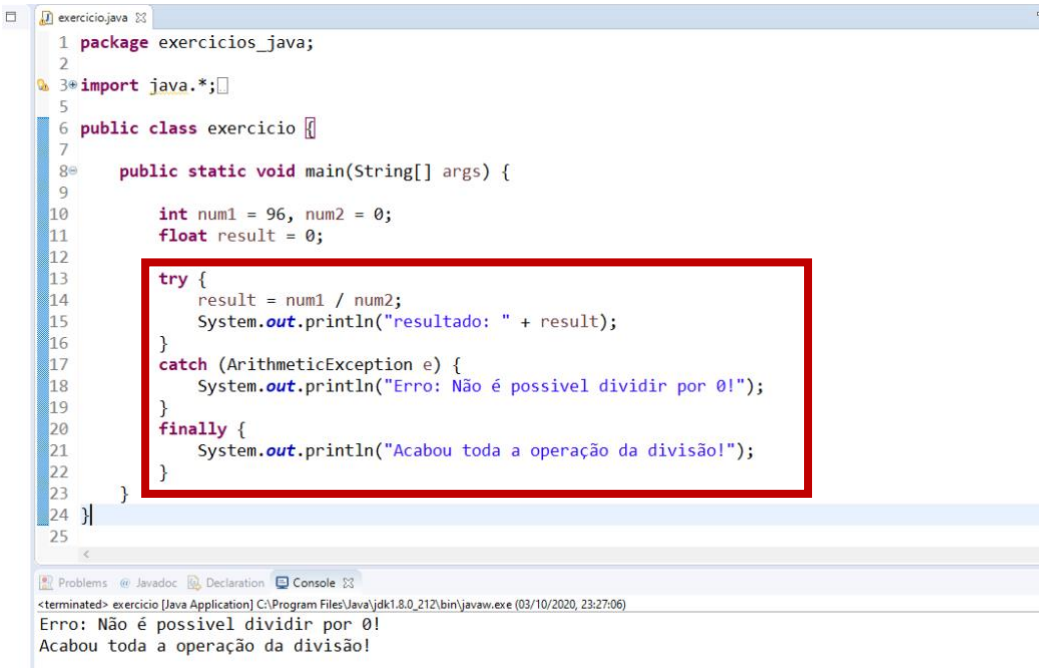
Como podemos ver, existem vários tipos de exceções (do tipo Throwable) que podem ser geradas a partir do código pelo programador:



Resumindo, podemos indicar a exceção mais pertinente para o erro em causa, como por exemplo:

- Tratamento de ***ficheiros*** usamos o ***IOException;***
- Tratamento ***aritméticos*** usamos o ***ArithmeticException;***
- Para testar ***ligação a base de dados*** usamos o ***SQLException:***
- ***Entre outras opções …***

Agora vamos compor o exercício anterior com a exceção mais adequada no caso da divisão por 0 (que também foi sugerido na consola):



14.4 – Lançamento de exceções (Por iniciativa do Programador)

Pode haver situações em que é útil ao programador lançar excepções. Para isso no decorrer do código pode utilizar a instrução:

throw new <throwable_exception>

```
public static String getProdutoByCodID(int codID)
{
    if (codID <= 0)
    {
        throw new IllegalArgumentException("O código de produto tem de ser " +
                "superior a 0!");
    }
    else{
        switch(codID){
            case 1:
```

O código que invocar o método que lança a exceção deve ter um bloco try/catch para tratar devidamente a exceção.

No exemplo anterior, o código deve pedir ao utilizador para voltar a introduzir um número sempre que é gerada a exceção (***IllegalArgumentException***).

A classe ***Throwable***, a partir da qual derivam todas as exceções, tem métodos que permitem obter a mensagem que descreve a exceção e o nome da própria exceção.

Quando é lançado no código uma nova exceção, é possível/recomendável passar uma mensagem descritiva da exceção no construtor da mesma

```
throw new IOException("Falha ao aceder ao ficheiro da BD!");

        } catch (IOException myIOException) {
            System.out.println(myIOException.getMessage());
        }

java.io.IOException: Falha ao aceder ao ficheiro da BD!
```

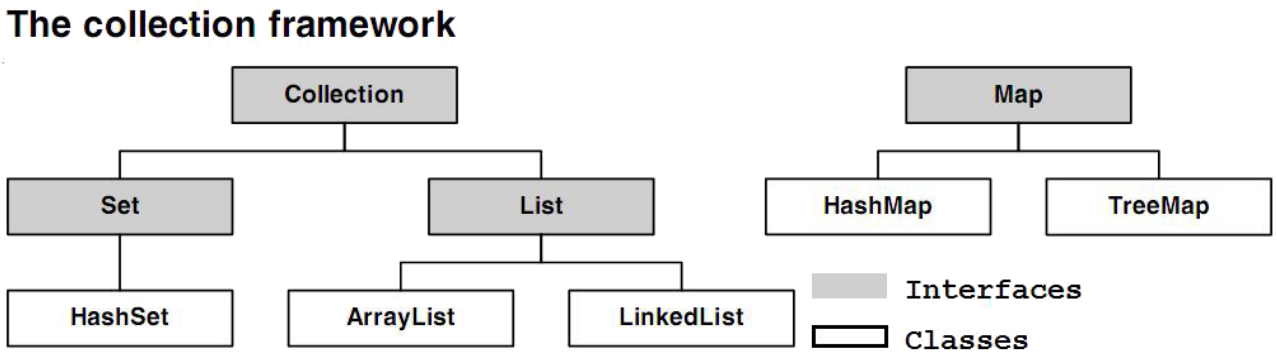
15 – Coleções em Java – Collections (Coleções)

"A Collections Framework é uma arquitetura unificada, constituída por interfaces, classes abstratas, implementações e métodos, visando representar e manipular coleções, normalizando as API, oferecendo conversões entre estruturas, reduzindo o esforço de aprendizagem e programação, e oferecendo grandes soluções para reutilização"

FCA - Java 8 - POO + Construções Funcionais

Benefícios da utilização das coleções:

- reduz o esforço de programação, uma vez que fornece estruturas de dados e algoritmos que permite abstrair de como são feitas as operações baixo nível;
- aumenta a qualidade e desempenho das aplicações, a JCF fornece algoritmos e estruturas de dados de alto desempenho e qualidade;
- permite interoperabilidade entre diversas API, como a JCF é aceite por diversas API a troca de dados entre API fica facilitada com a utilização das collections;
- reduz o esforço de aprendizagem e utilização de novas APIs, uma vez que as APIs aceitam e trabalham com estruturas de dados provenientes da JCF;
- promove a reutilização de software, as estruturas de dados que estão ajustadas à JCF são por natureza reutilizáveis;



A JCF consiste em ***2 hierarquias***: "***Collection***" e "***Map***":

- ***Collection*** → armazena objetos individuais como sendo elementos;
- ***Map*** → armazena pares de objetos com ***chave | valor***, onde a chave é única e permite ter acesso ao valor;

Existem mais de 30 classes que compõem a JCF, mas as mais utilizadas costumam ser a ***ArrayList***, ***LinkedList***, ***HashMap*** e ***TreeMap***.

Classe	Descrição
ArrayList	Funciona como um Array mas pode expandir. Eficiente para aceder a elementos de forma sequencial, ineficiente se for necessário inserir elementos no meio da lista.
LinkedList	Semelhante a um ArrayList mas com mais funcionalidades. Menos eficiente a aceder elementos sequencialmente, mais eficiente para inserir elementos no meio da lista.
HashSet	Collection com armazenamento de elementos únicos baseado no seu *hash code*. Os objectos a adicionar a um tipo Hash têm de implementar o método hashCode, para garantir que é único.
HashMap	Semelhante ao HashSet mas implementa a interface Map, logo armazena pares chave|valor em que as chaves têm de ser únicas.
TreeMap	Armazena pares chave|valor numa estrutura em árvore onde são automaticamente ordenadas com base na chave (key).

15.1 – Construtores da classe ArrayList

Construtor	Descrição
ArrayList<E>()	Cria um ArrayList vazio com capacidade para 10 Objectos do tipo <E>.
ArrayList<E>(*n*)	Cria um ArrayList vazio com *capacidade* para *n* elementos.

- Um ArrayList é um colecção semelhante a um Array, mas com capacidade decrescimento à medida que os elementos são adicionados/removidos.
- No construtor é possível definir o tipo de elementos que o ArrayList comporta.
- Internamente o ArrayList utiliza Array para armazenar os elementos, logo sempre que se aumenta/diminui a capacidade do ArrayList é criado um novo array com o tamanho pretendido e os elementos são copiados…

Métodos da classe ArrayList

Método	Descrição
add(*object*)	Adiciona um *object* ao fim da lista.
add(*index*, *object*)	Adiciona um *object* na posição *index*.
clear()	Remove todos os elementos da lista.
contains(*object*)	Devolve *true* se *object* está na lista.
get(*index*)	Retorna o *object* que está na posição *index*.
indexOf(*object*)	Devolve a posição do *object*.
isEmpty()	Devolve *true* se a lista está vazia.
remove(*index*)	Remove o objecto da lista na posição *index*.
set(*index*, *object*)	Coloca *object* na posição *index*.
size()	Devolve o número de elementos da lista.
toArray()	Retorna um array com os elementos da lista.

15.2 – Exemplo de 1 Arraylist com 1 tipo de dados

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

    ArrayList<String> modelosCarros = new ArrayList<String>();
    modelosCarros.add("scirocco");
    modelosCarros.add("A3");
    modelosCarros.add("John Cooper Works");

    //imprimir o ArrayList atraves do ciclo for
    for(int i=0; i < modelosCarros.size(); i++)
    {
        String modelo = modelosCarros.get(i);
        System.out.println(modelo);
    }

    //imprimir directamente, implicitamente é chamado o método toString()
    System.out.println(modelosCarros);
```

15.3 – Exemplo de 1 Arraylist com Classes

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

    Scanner teclado = new Scanner(System.in);

    //Criar uma nova lista de dados com a estrutura da classe Carro
    ArrayList<Carro> newCarro = new ArrayList<Carro>();

    //inserir informação para dentro da lista
    newCarro.add(new Carro("VW", "Scirocco", 3));
    newCarro.add(new Carro("BMW", "120D", 3));
    newCarro.add(new Carro("Mini", "John Cooper Works", 5));

    System.out.println(newCarro);

    for (Carro valor : newCarro) {
      System.out.println(valor.MostraMensagem());
    }

}
```