

Notas sobre Java

António Anjos

aanjos@uevora.pt

Departamento de Informática
Universidade de Évora

4 de maio de 2021



Tópicos

1 Capítulo 1 e 2

2 Capítulo 3

3 Capítulo 4

4 Capítulo 6



Convenções

- Nomes de classe: iniciam com maiúscula, utilizam CamelCase, e são nomes no singular
`Pessoa`, `Animal`, `Carro`, `CarroEletrico`
- Nomes de variável/atributo: iniciam com minúscula, e utilizam camelCase
`ano`, `idade`, `pesoLiquido`, `notaFrequencia1`, `notaExameRecurso`
- Nomes de método: iniciam com minúscula, e utilizam camelCase
`getAno`, `getIdade`, `setPesoLiquido`, `getNotaExameRecurso`
- Nomes de constante: tudo em maiúsculas, e utilizam snake_case
`PI`, `TAMANHO_MAXIMO`, `MAX_LARGURA_JANELA`
- Nomes de packages: uma palavra em minúsculas
`java`, `javafx`, `swing`, `org`, `opencv`

NOTA: Packages são um tipo de pastas que contêm classes (ou outras pastas). E.g, `import java.util.Scanner`, quer dizer: “Carregar a classe Scanner que está na ‘pasta’ util que, por sua vez, se encontra na ‘pasta’ java”



Tipos/classes e variáveis

- Exemplos de tipos primitivos: `int`, `float`, `double`, `char`, `boolean`
- Exemplos de “tipos” não primitivos (AKA classes): `String`, `StringBuilder`
- Para se declarar uma variável, primeiro indica-se o tipo e, só depois, o nome da variável, e.g.:

```
int a;  
String s;
```

Atenção!

- Tipos primitivos não requerem instanciação
- Classes obrigam a instanciação (i.e. utilização de `new`)
- `String` é exceção à regra. Temos as 2 hipóteses, com e sem utilização explícita de `new`:

```
String s1 = new String("Hello"); String s2 = "Hello"
```

Declaração vs inicialização

Declaração:

- Informa o compilador sobre a existência de uma variável e o tipo de dados que esta vai referir

- Exemplos:

```
int n;  
StringBuilder sb;
```

Inicialização:

- Atribui um valor à variável (já declarada)

- Exemplos:

```
n = 10;  
sb = new StringBuilder();
```

- Declaração e inicialização num só passo:

```
int n = 10;  
StringBuilder sb = new StringBuilder();
```

Atenção!

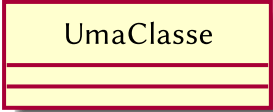
- Variáveis dentro de métodos devem de ser inicializadas (ou podem referenciar “lixo”)
- Atributos de classes são automaticamente inicializados com o *vazio* do tipo respetivo



class

- O código Java vive dentro de classes
- A classe mais pequena possível:

```
class UmaClasse {  
  
}
```

A diagram of a Java class box. It is a yellow rectangle with a red border. The text 'UmaClasse' is written in black in the top section. Below it are two empty horizontal lines, also within the red border.

UmaClasse

- Notar que o nome da classe inicia com maiúscula e, depois, CamelCase
- Esta classe não pode ser executada diretamente porque não tem o método `main`



main()

- Para ser possível executar um programa Java, uma das classes do programa tem de ter o método `main`
- Quando mandamos correr o programa, a JVM vai à procura do `main` dentro classe que tiver o mesmo nome que ficheiro `.java` que estamos a correr
- O `main` tem uma sintaxe especial que a JVM espera:

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

- `public`: permite à JVM aceder ao `main`
- `static`: para a JVM poder executar o `main` (i.e., sem instanciar a classe)
- `void`: para indicar que o `main` não vai retornar nenhum valor
- `String[] args`: contém as strings passadas pelo terminal (o nome `args` é utilizado frequentemente, mas pode ser o que quisermos)



Programa mínimo (assumir ficheiro com nome Treta.java)

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



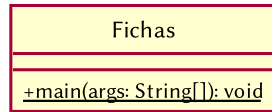
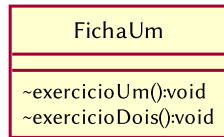
- Notar que:

- ▶ O main encontra-se dentro de uma classe que tem o mesmo nome que o ficheiro
- ▶ Ao executarmos o programa Treta.java, a JVM “vai” à classe treta e executa o main
- ▶ Métodos estáticos aparecem sublinhados em UML



Programa com 2 classes (ficheiro Fichas.java)

```
class FichaUm {  
    void exercicioUm() {  
        System.out.println("Olá mundo!");  
    }  
  
    void exercicioDois() {  
        System.out.println("Adeus mundo!");  
    }  
}  
  
class Fichas {  
    public static void main(String[] args) {  
        FichaUm f1 = new FichaUm();  
        f1.exercicioUm();  
    }  
}
```



NOTA: Este é um exemplo estúpido já que, geralmente, apenas faz sentido criar uma classe se estivermos a pensar criar mais do que uma instância da mesma. Inclusivamente, o nome `Fichas` viola a convenção.

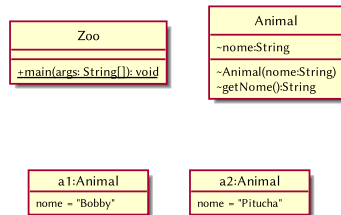


Programa com mais sentido (ficheiro Zoo.java)

```
class Animal {  
    String nome;  
  
    Animal(String nome) { // construtor  
        this.nome = nome;  
    }  
  
    String getNome() {  
        return nome;  
    }  
}
```

```
class Zoo {  
    public static void main(String[] args) {  
        Animal a1 = new Animal("Bobby");  
        Animal a2 = new Animal("Pitucha");  
        System.out.println("Um dos animais chama-se ", a1.nome);  
    }  
}
```

- 2 classes e 2 objetos:



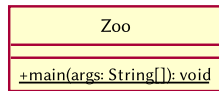
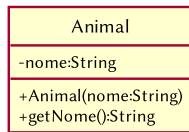
PERIGO! No main, estamos a aceder diretamente ao atributo nome!

NOTA: Em UML 1.0 sublinhava-se o nome e tipo dos objetos (i.e., das instâncias).



Programa com *ainda* mais sentido (ficheiro Zoo.java)

```
class Animal {  
    private String nome;  
  
    public Animal(String nome) { // construtor  
        this.nome = nome;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}  
  
class Zoo {  
    public static void main(String[] args) {  
        Animal a1 = new Animal("Bobby");  
        Animal a2 = new Animal("Pitucha");  
        System.out.println("Um dos animais chama-se ", a1.getNome());  
    }  
}
```



- a1.nome agora não é permitido!
- Notar privado (-) vs público (+)



Se o nome do animal não é conhecido à partida (ficheiro Zoo.java)

```
class Animal {  
    private String nome;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}  
  
class Zoo {  
    public static void main(String[] args) {  
        Animal a1 = new Animal();  
        a1.setNome("Bobby");  
        System.out.println("O bicho chama-se ", a1.getNome());  
    }  
}
```

Animal
-nome:String
+getNome():String +setNome(nome:String):void

Zoo
<u>+main(args: String[]): void</u>



Atributos (i.e., *variáveis*) e métodos de instância vs classe

- Um atributo de instância:
 - ▶ é uma *variável* exclusiva de cada objeto (i.e., cada instância tem a sua própria cópia)
 - ▶ só existe se a instância tiver sido criada
 - ▶ se for público, acede-se através do objeto, e.g., `a1.nome`
- Um atributo de classe (i.e., **static**):
 - ▶ é uma *variável* comum a todos os objectos instanciados a partir dessa classe
 - ▶ acede-se, geralmente, através do nome da classe, e.g., `JFrame.ABORT`
 - ▶ não requer a existência de um objeto dessa classe
 - ▶ define, frequentemente, uma constante dessa classe (como no exemplo anterior)
- Os métodos de instância e de classe são acedidos da mesma forma que os atributos

UmaClasse
-variavelDeInstancia: int <u>-variavelDeClasse: int</u> <u>+CONSTANTE DE CLASSE: int = 100</u>
+metodoDeInstancia(): int <u>+metodoDeClasse(): int</u>

Literais numéricos vs Tipos numéricos (mais comuns)

- **int**: não tem parte decimal
-5, 4, 8, 2000
- **float**: tem parte decimal (basta sufixar com um f)
-5f, 4f, 8.0f, 2000.5f
- **double**: tem parte decimal e mais precisão que **float**
-5.0, 4.0, 8.0, 2000.5



Operações entre diferentes tipos

- O valor final de uma operação é do tipo mais geral
- Exemplo (com operador `+`, mas pode ser outro operador aritmético):
 - ▶ `int + float → float`
 - ▶ `int + double → double`
 - ▶ `float + double → double`



Atribuição de valores

- Para se atribuir um valor a uma variável, a variável tem de ter capacidade para receber o valor atribuído

- Exemplos OK:

```
double n = 2.5;    // double cabe num double
double n = 2.5f;   // float cabe num double
double n = 2;      // int cabe num double
float n = 2.5f;    // float cabe num float
float n = 2;       // int cabe num float
int n = 2;         // int cabe num int
```

- Quando os tipos não são iguais, ocorre um **cast implícito** (i.e., uma conversão de tipo)



Atribuição de valores: Not OK

- Exemplos que vão dar erro:

```
int n = 2.5;      // double não cabe num int
```

```
int n = 2.5f;     // float não cabe num int
```

```
float n = 2.5;    // double não cabe num float
```

- Para “forçar” a atribuição tem de se fazer um **cast explícito** para o tipo da variável recetora (o mesmo que dizer “*trust me I know what I’m doing*”)

```
int n = (int)2.5;    // double não cabe num int
```

```
int n = (int)2.5f;   // float não cabe num int
```

```
float n = (float)2.5; // double não cabe num float
```

- Se o valor a atribuir não “encaixar” na variável recetora:

- ▶ **int**, a parte da informação que não encaixa é descartada, por exemplo:

```
int n = (int)2.5;    // n contém apenas o valor 2!!!
```

- ▶ **float**, a variável toma o valor Infinity

- A operação de cast tem precedência sobre as operações aritméticas



Operadores de divisão

- A divisão é efetuada pelo operador /

Cuidado!

- ▶ Referido anteriormente: operações aritméticas entre dois tipos, resultam no tipo mais geral
- ▶ Assim, se ambos operandos forem **int**, o resultado será **int**

```
5/2    // O resultado é 2 em vez de 2.5
```

```
1/2    // O resultado é 0 em vez de 0.5
```

- ▶ A solução passa por converter um dos operandos:

```
1f/2    // O resultado é 0.5f
```

```
1/2.0    // O resultado é 0.5 (i.e. double)
```

- O resto da divisão é dado pelo operador %
 - ▶ Em Java (ao contrário do C), pode-se operar não inteiros



Limites dos tipos primitivos

- Quantas “caixas de memória” são necessárias para representar o resultado de $\frac{1}{3}$?
- Temos de decidir: valores máximos, mínimos, e precisão que queremos representar
- Decidir, quer dizer: “escolher que tipo vamos usar, `int`, `float`, etc.”
- Se o resultado de uma operação com *não inteiros* não “caber” no tipo que estamos a utilizar, o resultado será Infinity (e.g., cast de um `double` muito grande para `float`)
- No caso dos inteiros, temos um overflow (não é fácil de detetar)
- As *wrapper classes* permitem consultar os valores máximos (e/ou mínimos) possíveis:

```
Integer.MAX_VALUE // maior inteiro possível (o mais negativo é MIN_VALUE)
Float.MAX_VALUE   // maior float possível (o mais negativo é -MAX_VALUE)
Double.MAX_VALUE  // como no float
Float.MIN_VALUE    // o valor mais pequeno que é possível representar
Double.MAX_VALUE   // como no float
```



Resultados estranhos? (tipos com vírgula flutuante)

Operação	Resultado	Comentário
$\frac{n}{\pm\text{Infinity}}$	0	Algo dividido infinitamente, tende para 0
$\text{Infinity} + \text{Infinity}$	Infinity	
$\pm\text{Infinity} \times \pm\text{Infinity}$	$\pm\text{Infinity}$	
$\frac{\pm n}{0}, \quad n \neq 0$	$\pm\text{Infinity}$	Se n e 0 forem int , temos uma <code>ArithmeticException</code>
$\frac{\pm 0}{\pm 0}$	NaN	
$\pm\text{Infinity} - \text{Infinity}$	NaN	
$\frac{\pm\text{Infinity}}{\pm\text{Infinity}}$	NaN	
$\pm\text{Infinity} \times 0$	NaN	

Decorar?

- Muito mais importante que decorar cada situação é saber que `Infinity` e `NaN` podem advir duma destas operações

Formatar números com DecimalFormat

- Para valores numéricos, podemos criar um formatador

```
double altura = 1.785;  
String padrao = "##.##"; // ou "00.00", aparece 0 se não existir dígito  
DecimalFormat df = new DecimalFormat(padrao);  
System.out.println("A altura é " + df.format(altura));
```

Detalhe sobre a sintaxe dos padrões



Output formatado

- Em vez de se criar uma string a partir de concatenação:

```
int idade = 25; double alt = 1.785;  
System.out.print("Idade é " + idade + ", e altura é " + altura + "cm");
```

- é possível indicar onde, na string, devem aparecer os valores:

```
System.out.format("A idade é %d, e a altura %f cm", idade, altura);
```

- O %d e %f, são chamados conversores para inteiros (i.e. decimal integers) e não inteiros (i.e. floating point), respetivamente
- O formatador %f permite controlar o número de casas decimais do número, e.g. 2 casas:

```
System.out.format("A a altura é %.2f cm", altura);
```
- `System.out.format()` é equivalente a `System.out.printf()`



Alinhamento de Strings (com conversor %s)

- Pode definir-se o espaço mínimo e máximo de caracteres que uma String deve ocupar, bem como o seu alinhamento nesse espaço
- Espaço exato, texto alinhado à direita (o que acontece por omissão nas Strings):

```
System.out.format("%s", "Olá") // ou System.out.println("Olá")
```

Output:

Olá

- Espaço mínimo 10 caracteres, texto alinhado à direita:

```
System.out.format("%10s", "Olá");
```

Output:

Olá

- Espaço mínimo 10 caracteres, texto alinhado à esquerda:

```
System.out.format("%-10s", "Olá");
```

Output:

Olá



String: Métodos úteis

- **int** `length()`: Devolve o tamanho da string

```
String nome = "Manuel";  
System.out.println(nome.length());           // imprime 6  
System.out.println("Maria".length());        // imprime 5
```

- **char** `charAt(pos)`: Devolve o carater na posição pos

```
System.out.println(nome.charAt(2));           // imprime 'n'
```

- String `substring(posI, posF)`: Devolve a sub-string de posI até posF (não inc.)

```
int pos = 5;  
System.out.println(nome.substring(2, pos));   // imprime "nue"
```

- String `concat(String)`: Concatena as duas strings

```
System.out.println("Olá".concat(" Maria"));  // "Olá Maria"
```



concat vs +

- O código

```
System.out.println("Olá".concat(" Maria")); // "Olá Maria"
```

é parecido com:

```
System.out.println("Olá" + " Maria"); // "Olá Maria"
```

- Isto quer dizer que, ao se utilizar o +, um dos operandos **tem de ser** uma String, para que o resultado seja uma String
- Situações problemáticas:

```
System.out.println('A' + 'b'); // Adição de chars resulta em int!
```

```
System.out.println("Olá".charAt(1) + "bom".charAt(0) + "dia");
```

- ▶ No último exemplo, somamos 2 **chars** que resultam num **int** e, depois, esse número concatena com a String "dia"
- ▶ Relembrar que charAt devolve um **char** e não uma String



Math: Métodos úteis

- `static double pow(double a, double b)`: Devolve a^b
`System.out.println(Math.pow(2, 3)); // imprime 8`
- `static double floor(double a)`: Devolve a arredondado para baixo
`System.out.println(Math.floor(2.9)); // imprime 2`
- `static double ceil(double a)`: Devolve a arredondado para cima
`System.out.println(Math.ceil(2.1)); // imprime 3`
- `static int round(float a)`: Devolve a arredondado ao inteiro mais próximo
`System.out.println(Math.round(2.1)); // imprime 2`

Atenção!

- Notar que os métodos (e atributos) da classe `Math` são todos **static**, i.e., de classe
- Assim nunca instanciamos a classe `Math` (i.e., nunca se criam objetos desta classe)
- Todos os membros da classe são acedidos através nome da classe, e não através do objeto

Scanner: Métodos úteis

- String `next()`: Lê até ao próximo espaço e devolve uma string

```
Scanner sc = new Scanner(System.in);
String frase = sc.next();           // Digitar: "Olá bom dia"
System.out.println(frase);         // imprime "Olá"
```
- String `nextLine()`: Lê até ao próximo enter e devolve uma string

```
String frase = sc.nextLine();      // Digitar: "Olá bom dia"
System.out.println(frase);         // imprime "Olá bom dia"
```
- String `nextInt()`: Lê e devolve um número inteiro

```
int valor = sc.nextInt();
```
- float `nextFloat()`: Lê e devolve um número *floating point* (vírgula flutuante)

```
float valor = sc.nextFloat();
```
- double `nextDouble()`: Lê e devolve um número *floating point* de precisão dupla

```
double valor = sc.nextDouble();
```
- void `close()`: Fecha o scanner (nunca fechar antes de ler tudo o que queremos!!!)

```
sc.close();
```



Ler do teclado

- A forma mais fácil de ler valores introduzidos através do teclado é utilizando um objeto da classe Scanner (temos de importar do package `java.util`)
- Os objetos da classe Scanner precisam de saber onde queremos fazer o *scan*
- Em Java, “teclado” diz-se: `System.in`
- Portanto, podemos ler do teclado assim:

```
Scanner sc = new Scanner(System.in); // instância p/ler no teclado
String nome = sc.next();
String morada = sc.nextLine();
int idade = sc.nextInt();
float altura = sc.nextFloat(); // ou double altura = sc.nextDouble()
sc.close(); // fechar quando não for necessário ler mais nada
```



Escrever no ecrã

- Em Java, “ecrã” diz-se: `System.out`¹
- Podemos escrever no ecrã por enviar a mensagem `print` ao “ecrã”
`System.out.print("Olá");`
- O “ecrã” sabe responder a outras mensagens, p.ex., `println` que, além de imprimir a mensagem desejada, também muda de linha

- São equivalentes:

```
System.out.println("Olá");
```

```
System.out.print("Olá\n");
```

`'\n'` é um carácter invisível que representa uma nova linha (new line)

¹`out` é, um atributo da classe `System`, instância da classe `PrintStream` e que a JVM ‘canaliza’ para o ecrã 🔍 ↻



Importar?

- Porque é que temos acesso às classes `String` e `Math` sem ter de importar?
 - ▶ Estão ambas no package `java.lang` que é importado automaticamente pela JVM
- As classes no package `java.util` não são importadas automaticamente
 - ▶ Por isso é que, para utilizar a classe `Scanner`, precisamos de importar:

```
import java.util.Scanner;
```



Controlo de acesso aos membros da classe

- Os membros de uma classe são os:
 - ▶ Atributos (i.e., as variáveis disponíveis na classe)
 - ▶ Métodos (i.e., as “funções” definidas na classe)
- Podemos controlar o acesso aos membros usando as palavras chave:
 - ▶ **private** (– em UML): Ninguém fora da classe pode aceder
 - ▶ **public** (+ em UML): Todos fora da classe podem aceder
 - ▶ Há mais 2, mas ficam para mais tarde

Qual usar?

- Regra geral (para simplificar):
 - ▶ Atributos devem ser privados
 - ▶ Métodos devem ser públicos
- Dito isto, há atributos que faz sentido serem públicos e métodos que faz sentido serem privados (quando tiverem mais experiência)

public VS private

```
class Animal {
    public String especie;

    public String getEspecie(){
        return especie;
    }

    public void setEspecie(String e) {
        especie = e;
    }
}

class Treta {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.especie = "Carapau";           // OK
        a.setEspecie("Galinha");         // OK
        System.out.println(a.getEspecie()); // OK
    }
}
```

```
class Animal {
    private String especie;

    private String getEspecie(){
        return especie;
    }

    public void setEspecie(String e) {
        especie = e;
    }
}

class Treta {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.especie = "Carapau";           // NÃO OK !!!
        a.setEspecie("Galinha");         // OK
        System.out.println(a.getEspecie()); //NÃO OK !!!
    }
}
```



Controlo de acesso a classes

- É possível controlar o acesso à classe por parte de código noutras classes e packages
- Este é conteúdo para mais tarde, mas fica aqui um aviso muito importante:

AVISO!!!

- ▶ Os exercícios no Moodle requerem que a classe onde está o main seja pública

- Funciona no Moodle:

```
public class Treta { // OK
    public static void main(){
        System.out.println("Olá");
    }
}
```

- Não funciona no Moodle:

```
class Treta { // Falta o public!
    public static void main(){
        System.out.println("Olá");
    }
}
```



Tipo de dados **boolean**

- Apenas 2 valores possíveis (ou **null** se não for inicializado):
 - ▶ Verdadeiro: **true**
 - ▶ false: **false**
- São chamados de valores lógicos
- Existem operadores para operar estes valores



Operadores lógicos

- **And** lógico: `&&`

- ▶ Resulta em **true** se **ambos** operadores forem **true**, ou falso caso contrário

```
System.out.print(true && true); // imprime true
```

```
System.out.print(true && false); // (ou false && true) imprime false
```

- **Or** lógico: `||`

- ▶ Resulta em **true** se **um** dos operandos for **true**, ou **false** caso contrário

```
System.out.print(true || true); // imprime true
```

// ou true||false; ou false||true

```
System.out.print(false || false); // false (apenas se ambos false)
```

- **Not** lógico: `!`

- ▶ Operador unário que nega o valor lógico

```
System.out.print(!true); // imprime false
```

```
System.out.print(!false); // imprime true
```



Operadores lógicos: Resumo

Operação	Res
<code>false && false</code>	<code>false</code>
<code>false && true</code>	<code>false</code>
<code>true && false</code>	<code>false</code>
<code>true && true</code>	<code>true</code>

Operação	Res
<code>false false</code>	<code>false</code>
<code>false true</code>	<code>true</code>
<code>true false</code>	<code>true</code>
<code>true true</code>	<code>true</code>

Operação	Res
<code>!false</code>	<code>true</code>
<code>!true</code>	<code>false</code>

- Precedências:

- ▶ 1st `!`; 2nd `&&`; 3rd `||`
- ▶ Pensem no negativo (menos unário), multiplicação e adição
- ▶ Pode alterar-se com a utilização de parêntesis

- Operadores lógicos têm precedência inferior à dos operadores relacionais (a seguir)



Operadores relacionais

- Não é frequente escrever-se **true** ou **false** diretamente em expressões
- Valores booleanos resultam, frequentemente, de operações relacionais (i.e., comparações)
- Um operador relacional estabelece a relação entre 2 valores
- Por exemplo: “10 é menor que 20?”

```
System.out.print(10 < 20); // imprime: true
```



Operadores relacionais: Resumo

Operador	Operação	Exemplo 1	Exemplo 2
<	Menor que	10 < 20 → true	20 < 10 → false
>	Maior que	5 > 2 → true	2 > 5 → false
<=	Menor ou igual que	8 <= 10 → true	8 <= 8 → true
>=	Maior ou igual que	4 >= 1 → true	4 >= 4 → true
==	Igual a	6 == 6 → true	3 == 6 → false
!=	Não igual a (diferente)	2 != 7 → true	8 != 8 → false

- Operadores relacionais têm precedência mais elevada que os lógicos
- Pensar nos operadores relacionais como uma pergunta à qual a resposta apenas pode ser **true** ou **false**



Executar ou não executar

- Dependendo do resultado de uma operação relacional, podemos decidir se queremos, ou não executar uma (ou mais) instrução
- A instrução **if** é uma das que permite fazer isso
- Sintaxe:

```
if (valorLógico)  
    instruçãoAExecutar;
```

a instrução só será executada se o valor lógico for **true**

- Recordar que os valores lógicos resultam de operações relacionais (i.e., comparações)
- Se tivermos mais do que uma instrução a executar, utilizamos chavetas:

```
if (valorLógico) {  
    instrução1;  
    instrução2;  
}
```



Executar de entre duas alternativas

- É possível executar alternativas dependendo do valor lógico

```
if (valorLógico)
    instruçãoAExecutar;    // executa se valorLógico for true
else
    instruçãoAlternativa;  // executa se nenhum dos valores
                           // lógicos anteriores for true
```

- Notar que se uma das instruções (ou bloco de instruções) for executada, mais nenhuma no **if** será
- Aplica-se a mesma regra da utilização de chavetas para executar grupos de instruções



Executar de entre **mais de duas** alternativas

- É possível executar várias alternativas dependendo de vários valores lógicos

```
if (valorLógico1)
    instruçãoAExecutar;    // executa se valorLógico for true
else if (valorLógico2)
    instruçãoAlternativa1;
else
    instruçãoAlternativa2; // executa se nenhum dos valores
                          // lógicos anteriores for true
```

- Podemos ter quantos **else if** desejarmos
- Notar que sempre que aparece **if**, tem de haver um valor lógico (i.e., condição)
- O **else** isolado apenas pode aparecer no fim (como alternativa final)
- Aplica-se a mesma regra da utilização de chavetas para executar grupos de instruções



Muitas comparações com a mesma variável `int` (`==`)

```
if (a == 5) {  
    s = 50;  
    a = 0;  
} else if (a == 10) {  
    s = 2;  
    a = 5;  
} else if (a == 20) {  
    s = 30;  
    a = 200;  
} else {  
    s = 34;  
    a = 15;  
}
```

- `if` é complicado de ler se tivermos de ver se uma variável é um de muitos valores inteiros (ou strings)
- Nesta situação recomenda-se a utilização da instrução `switch`
- Notar que o `default` funciona como o `else` final, e também é opcional
- O `break` impede que, depois de executar o código, a comparação continue e outro código no `switch` seja executado

```
switch (a) {  
    case 5:  
        s = 50;  
        a = 0;  
        break;  
    case 10:  
        s = 2;  
        a = 5;  
        break;  
    case 20:  
        s = 30;  
        a = 200;  
        break;  
    default:  
        s = 34;  
        a = 15;  
}
```

Impressionar os/as amigos/amigas

```
int a = 10;

if (a == 10) {
    System.out.print("Dez");
} else {
    System.out.print("Outro");
}
```

- O `if...else` pode ser substituído pelo operador ternário condicional
`System.out.print(a==10 ? "Dez" : "Outro");`



while

- Quando não sabemos à partida o número de vezes que queremos repetir a instrução (ou bloco de instruções)
- Sintaxe:

```
while (condição) {  
    instrução1;  
    instrução2;  
}
```

- As instruções repetem-se enquanto a condição for **true**
- Alguma das instruções dentro do ciclo, terá de alterar o resultado da condição, senão repete para sempre



do ... while

- Quando não sabemos à partida o número de vezes que queremos repetir as instruções
- Mas queremos que execute as instruções **pelo menos 1 vez**
- Sintaxe:

```
do {  
    instrução1;  
    instrução2;  
} while (condição);
```

- As instruções repetem-se enquanto a condição for **true**
- Alguma das instruções dentro do ciclo, terá de alterar o resultado da condição, senão repete para sempre
- Esta tem um ponto-e-vírgula no fim



for

- Quando sabemos à partida o número de vezes que queremos executar as instruções

```
for (variável; condição; atualização) {  
    instruções;  
}
```
- Executa enquanto a condição for **true**
- Podemos utilizar a variável:
 - ▶ para definir a condição
 - ▶ dentro do ciclo
 - ▶ na secção atualização (e.g., alterar a variável a cada passo)

Importante!

- ▶ a variável é 'executada' antes de executar a condição e o bloco
- ▶ a condição é 'executada' antes de executar o bloco
- ▶ a atualização é feita depois de executar o bloco

for: exemplos

```
for (int i=0; i < 10; i=i+1) {  
    System.out.println(i);  
}  
  
for (int i=0; i < 10; i+=1) {  
    System.out.println(i);  
}  
  
for (int i=0; i < 10; i++) {  
    System.out.println(i);  
}
```

```
for (int i=0; i < 10; i=i+5) {  
    System.out.println(i);  
}  
  
for (int i=0; i < 10; i+=5) {  
    System.out.println(i);  
}  
  
for (int i=10; i >= 0; i--) {  
    System.out.println(i);  
}
```



for: mais um exemplo

- Calcular o valor da seguinte série:

$$S = 1000 + 100 + 10 + 1 + 0.1 + 0.01 + 0.001$$



for: mais um exemplo

- Calcular o valor da seguinte série:

$$S = 1000 + 100 + 10 + 1 + 0.1 + 0.01 + 0.001$$

```
double s = 0.0;
for (double x = 1000; x >= 0.001; x /= 10) {
    s += x;
}
System.out.format("%.2f\n", s);
```

