

# Notas sobre Java

António Anjos

[aanjos@uevora.pt](mailto:aanjos@uevora.pt)

Departamento de Informática  
Universidade de Évora

31 de maio de 2021



# Conteúdo

- 1 Conceitos introdutórios
- 2 Tipos e operadores
- 3 Formatação de output
- 4 Classes e métodos úteis
- 5 Controlo de acesso aos membros
- 6 Operadores lógicos, relacionais, e decidir
- 7 Ciclos
- 8 Revisões
- 9 Sobrecarga de construtores e de métodos
- 10 Enumerados
- 11 Packages
- 12 Exceções
- 13 Arrays e coleções
- 14 Herança



# 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!

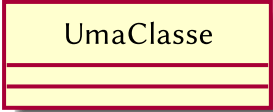
- Variáveis dentro de métodos devem de ser inicializadas (ou podem referenciar “lixo”)
- Atributos de classes são automaticamente inicializados com o *vazio* do tipo respetivo



# 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:

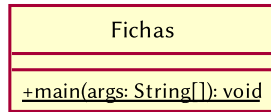
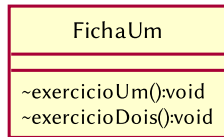
- ▶ O main encontra-se dentro de uma classe que tem o mesmo nome que o ficheiro
- ▶ Ao executarmos o programa Treta.java, a JVM “vai” à classe treta e executa o main
- ▶ Métodos estáticos aparecem sublinhados em UML





# 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.

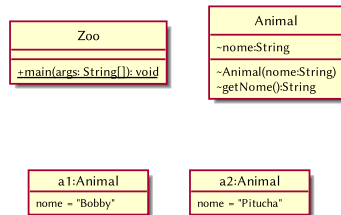


# Programa com mais sentido (ficheiro Zoo.java)

```
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:



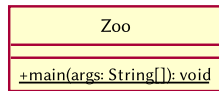
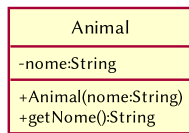
**PERIGO!** No main, estamos a aceder diretamente ao atributo nome!

**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 “couber” 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
$\text{Infinity} + \text{Infinity}$	$\text{Infinity}$	
$\pm\text{Infinity} \times \pm\text{Infinity}$	$\pm\text{Infinity}$	
$\frac{\pm n}{0}, \quad n \neq 0$	$\pm\text{Infinity}$	Se $n$ e 0 forem <code>int</code> , temos uma <code>ArithmeticException</code>
$\frac{\pm 0}{\pm 0}$	<code>NaN</code>	
$\pm\text{Infinity} - \text{Infinity}$	<code>NaN</code>	
$\frac{\pm\text{Infinity}}{\pm\text{Infinity}}$	<code>NaN</code>	
$\pm\text{Infinity} \times 0$	<code>NaN</code>	

### 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"
```

- **public int** `indexOf(char)`: Devolve posição do carácter (um **int** na realidade)

```
System.out.println("Olá".indexOf('o'));       // imprime 1
```



## 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`<sup>1</sup>
- 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)

---

<sup>1</sup>`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!!!

- ▶ Alguns dos exercícios no Moodle podem requerer que a classe onde está o main seja pública

- Se isto não funcionar:

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

- Experimentar isto:

```
public class Treta { // com 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 &amp;&amp; false</code>	<code>false</code>
<code>false &amp;&amp; true</code>	<code>false</code>
<code>true &amp;&amp; false</code>	<code>false</code>
<code>true &amp;&amp; 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:

- ▶ 1<sup>st</sup> `!`; 2<sup>nd</sup> `&&`; 3<sup>rd</sup> `||`
- ▶ 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:

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

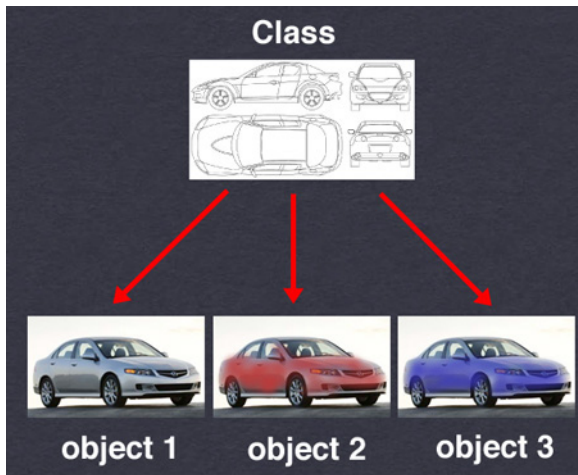
```
double s = 0.0; // obrigatório, senão s pode conter qualquer coisa <> 0
for (double x = 1000; x >= 0.001; x /= 10) {
    s += x;
}
System.out.format("%.2f\n", s);
```



- Podemos ver uma classe como se fosse uma planta de uma casa
  - ▶ A planta define as características das casas que serão feitas a partir desta (ninguém mora numa planta)
  - ▶ As casas feitas a partir desta são objetos dessa planta (a coisa concreta)



# Revisão: Classe vs Objeto



- **Classe** é uma espécie de desenho (i.e., planta) que define as características dos objetos que vamos produzir
- **Objectos** são as coisas que fazemos tendo como base o desenho (i.e., a classe)
- A classe define-se uma vez, e não queremos saber mais dela
- O que nos interessa, no fim, são os objetos (ninguém consegue viajar num desenho, tem de ter um carro concreto)



# O que já sabemos até agora

- Uma classe pode ser vista como a definição de um tipo (serve de molde)
- Um objeto é um exemplo (i.e., instância) de um elemento desse *tipo* (trabalhamos com os objetos)
- Por exemplo, "Olá" e "Adeus" são duas instâncias da classe `String`
- "Olá" tem as características definidas na classe `String`, e sabe responder às mensagens (i.e., métodos) definidas na classe `String`
- Para criar um objeto (i.e., instância) de uma classe, utilizamos a palavra `new` seguida do nome do tipo (i.e., classe) do objeto que queremos criar (`String` é um caso especial que não obriga a isso)



# Exemplo

- A “planta” (i.e., **classe**):

```
class Animal {  
    static int MAX_IDADE = 15;  
    String nome;  
  
    Animal(String nome) {  
        this.nome = nome;  
    }  
  
    String getName() {  
        return nome;  
    }  
  
    void setName(String nome) {  
        this.nome = nome;  
    }  
}
```

- Os **objetos**:

```
Animal a1 = new Animal("Bobby");  
Animal a2 = new Animal("Tareco");  
Animal a3 = new Animal("Pituxa");
```

- Cada objeto é totalmente independente e:

- ▶ tem o seu próprio atributo nome

```
System.out.println(a1.nome); // imprime "Bobby"
```

- ▶ sabe responder às mensagens getName e setName

```
System.out.println(a3.getName()); // imprime "Pituxa"
```

porque os objetos foram criados seguindo a informação na “planta”

## Importante!

- Atributos ou métodos declarados como **static** não fazem parte do objeto
- São da “planta”, e chamados a partir do nome da classe e não do objeto  
Por exemplo: `Animal.MAX_IDADE` vs `a3.nome`

# Sobrecarga de construtores (dentro da classe, claro)

```
public Animal() {  
    this.nome = "Bobby";  
}  
  
public Animal(String nome) {  
    this.nome = nome;  
}  
  
public Animal(String especie) {  
    this.especie = especie;  
}  
  
public Animal(int idade) {  
    this.idade = idade;  
}  
  
public Animal(String nome, int idade) {  
    this.nome = nome;  
    this.idade = idade;  
}
```

- O que o compilador vê:  

```
public Animal(void) {  
    this.nome = "Bobby";  
}  
  
public Animal(String) {  
    this.nome = nome;  
}  
  
public Animal(String) {  
    this.especie = especie;  
}  
  
public Animal(int) {  
    this.idade = idade;  
}  
  
public Animal(String, int) {  
    this.nome = nome;  
    this.idade = idade;  
}
```

- Para distinguir entre construtores, analisar:
  - ▶ Quantidade de parâmetros
  - ▶ Tipo dos parâmetros
  - ▶ Ordem dos parâmetros
- Se os 3 critérios forem iguais, não há forma de distinguir
- Por exemplo, não é possível distinguir entre o 2º e 3º construtores:  

```
Animal a = new Animal("Olá");
```
- Qual dos 2 que recebe uma String está a ser chamado?!



# Sobrecarga de métodos

- Vários métodos com mesmo nome
- Mesma lógica que para os construtores: métodos com o mesmo nome são considerados diferentes se:
  - ▶ O número de parâmetros for diferente
  - ▶ O tipo de parâmetros for diferente
  - ▶ A ordem dos (tipos dos) parâmetros for diferente
- Nesta análise, os nomes dos parâmetros são ignorados
- Compilador faz a mesma análise que um humano faria: Qual estou a chamar?  
`objeto.setInfo("Olá", 7);`

```
void setInfo(String) {  
}
```

```
void setInfo(String, String) {  
}
```

```
void setInfo(String, int) {  
}
```

```
void setInfo(int, String) {  
}
```

- Não há qualquer confusão entre os anteriores



# Construtores a chamar construtores

- Um construtor pode chamar outros construtores na mesma classe
- Dentro de um construtor, referimo-nos a outro construtor da classe utilizando:  
`this();`
- Dentro dos parêntesis, colocamos os argumentos requeridos pelo construtor que queremos chamar
- `this(...)` tem de ser a primeira coisa a acontecer no construtor
- O que vai acontecer?

```
Animal a = new Animal();
```

```
class Animal {  
    String nome;  
  
    Animal() {  
        // não pode haver código aqui  
        this("Desconhecido");  
        // pode haver mais código aqui  
    }  
  
    Animal(String nome) {  
        this.nome = nome;  
    }  
}
```





- Um enumerado pode ser visto como um tipo de dados, que apenas pode assumir os valores definidos nele
- Por exemplo:

```
enum Cor {  
    VERMELHO, VERDE, AZUL  
}
```

- Os valores são escritos em maiúsculas visto que são constantes (é implícito)
- Cada constante tem um valor inteiro que inicia em 0 (pode ser alterado)
- Experimente imprimir as constantes



## enum: exemplos de utilização

- Normalmente:

```
void printCor(Cor c) {  
    if (c == Cor.VERMELHO)  
        System.out.println("Cor vermelha");  
    else if (c == Cor.VERDE)  
        System.out.println("Cor verde");  
    else  
        System.out.println("Cor verde");  
}  
:  
:  
:  
printCor(Cor.AZUL);
```

- Num switch:

```
void printCor(Cor c) {  
    switch(c) {  
        VERMELHO: // não utilizar o nome do enum  
            System.out.println("Cor vermelha");  
            break;  
        VERDE:  
            System.out.println("Cor verde");  
            break;  
        default:  
            System.out.println("Cor azul");  
    }  
}
```



# Package

- De forma simples, uma pasta com ficheiros .java relacionados entre si
- Exemplo: pasta java/util contém os ficheiros Scanner.java, ArrayList.java, etc.
- Cada um desses ficheiros “diz” que faz parte dessa pasta por escrever

```
package java.util;
```

na primeira linha do código

- Resumindo, para criar um package<sup>2</sup>:
  - 1 Criar a estrutura de pastas desejada
  - 2 Colocar os ficheiros .java lá dentro
  - 3 Escrever na primeira linha de cada ficheiro `package nomes.das.pastas`
- O ponto a separar os nomes das pastas indicam que as pastas estão dentro das outras
- Por exemplo, o package java.util corresponde à estrutura de pastas java/util, i.e., a pasta util está dentro da pasta java

---

<sup>2</sup>No VSCode, se estivermos num projeto, apenas temos de escolher “New Package”, o VSCode trata de tudo.



# Utilizar um package

- Para utilizar os “ficheiros” que estão no package, temos de importá-los:  
`import nomepackage.NomeFicheiro;`
- Quando fazemos `import pasta.Ficheiro`, a MV do Java precisa de saber onde está a pasta (i.e., package)
- A MV vai procurar nas pastas que estiverem definidas numa variável de sistema chamada CLASSPATH
- No terminal podemos dizer para procurar noutros sítios, utilizando o parâmetro `-cp`
- No VSCode:
  - ▶ Se o package foi criado com “New Package”, o VSCode trata de tudo
  - ▶ Sem projeto criado, é-nos perguntado se queremos adicionar a pasta à CLASSPATH
  - ▶ Poderá ser necessário reiniciar o VSCode depois de adicionar-mos o package à CLASSPATH



# Lidar com situações excepcionais

- A forma habitual:

```
double dividir(int a, int b) {  
    if (b == 0) {  
        System.err.println("Divisor = 0!");  
        System.exit(0);  
    }  
    return (double)a/b;  
}
```

- ou, utilizando asserções (OK apenas em modo de debugging):

```
double dividir(int a, int b) {  
    assert b != 0;  
  
    return a/(double)b;  
}
```

- Em qualquer das situações, o programa termina sem dar oportunidade de recuperar do erro

- Uma forma mais elegante:

```
double dividir(int a, int b) throws Exception {  
    if (b == 0) {  
        Exception e = new Exception();  
        throw e;  
        // ou, numa só linha, throw new Exception()  
    }  
    return (double)a/b;  
}
```

- O código que chamou este método vai ter oportunidade de decidir o que fazer
- Simplificando muito, o `throw` é um `return` especial
- O que é devolvido por esse “`return`” tem de ser indicado no lado direito da assinatura do método

**Nota:** A classe `Exception` tem vários construtores



# Exceptions

- Condições anormais que interferem com a execução do programa (não durante a compilação)
- É uma instância da classe `Exception`, ou de uma das suas subclasses
- As exceções *apanhadas* podem ser tratadas
- Tratam-se através de blocos `try ... catch ... finally`
  - ▶ `try`: **Tenta** executar o bloco de código
  - ▶ `catch`: **Apanha** a exceção, caso não tenha sido possível executar o código com sucesso
  - ▶ `finally`: não é obrigatório mas, se definido, é sempre executado (i.e., quer tenha havido exceção, quer não; mesmo que haja um `return`)



# try ... catch ... finally

```
try {  
    // código que pode gerar exceção  
    // inclui chamadas a métodos que podem gerar exceção  
} catch (Exception e) {  
    // código que trata da exceção  
} finally {  
    // é sempre executado  
    // libertar recursos se necessário  
}
```

- É possível ter vários *catches*
- Quando isso acontece deve colocar-se os *catches* de exceções mais específicas primeiro (ver [hierarquia de exceções](#))
- Os objetos do *tipo* `Exception` sabem responder à mensagem `getMessage()`



# Exceções: Exemplo

```
double dividir(int a, int b) throws Exception {  
    if (b == 0) {  
        throw new Exception("Ups...");  
    }  
    return a/(double)b;  
}
```

- Se não terminarmos o programa dentro de um `catch`, a exceção considera-se tratada e o programa continua
- A única situação em que o `finally` não é executado é se terminarmos o programa (i.e., `exit()`), ou se falhar a luz
- O que acontece se trocarmos a ordem dos catches?

- No main:

```
int n;  
double d;  
  
try {  
    Scanner sc = new Scanner(System.in);  
    n = sc.nextInt();  
    d = dividir(5, n)  
} catch (InputMismatchException e) {  
    System.err.print("Não introduziste um int!");  
    System.exit(0); // decidi que é melhor parar  
} catch (Exception e) {  
    System.out.println(e.getMessage()); // ups...  
} finally {  
    sc.close();  
}
```





# Arrays: criação

- Declaração (1 linha de “caixas”):

```
int[] arrayInts; // menos legível: int arrayInts[]  
double[] arrayDoubles;  
Cat[] arrayGatos;
```

- Instanciação:

```
arrayInts = new int[5]; // apenas reservamos espaço para ints  
arrayDoubles = new double[5];  
arrayGatos = new Cat[5]; // apenas reservamos espaço p/objetos Cat
```

- De uma só vez:

```
int[] arrayInts = new int[5];  
double[] arrayDoubles = new double[5];  
Cat[] arrayGatos = new Cat[5];
```



# Array: utilização

- Armazenamento:

```
arrayInts[0] = 100; // na primeira pos do array  
arrayInts[arrayInts.length - 1] = 150; // notar atributo dos arrays  
arrayGatos[0] = new Cat("Patareco"); // notar criação do objeto
```

- Acesso:

```
int a = arrayInts[0]; // 'a' fica com 100  
Cat c = arrayGatos[0]; // 'c' passa a referir o objeto
```

- O que vai acontecer? (considere as instruções acima)

```
a = 20;  
System.out.print(arrayInts[0]);  
c.setName("Tico");  
System.out.print(arrayGatos[0].getName());
```



# Iterar sobre arrays

- Opção 1:

```
for (int i = 0; i < arrayGatos.length; i++) {  
    System.out.println(arrayGatos[i].getName());  
}
```

- Opção 2 (AKA *for-each*):

```
for (Cat c : arrayGatos) {  
    System.out.println(c.getName());  
}
```

```
for (int i : arrayInts) {  
    System.out.println(i);  
}
```



# Arrays multi-dimensionais

- Mesmo princípio que nos unidimensionais
- Declaração e instanciação (uma matriz de “caixas”):

```
int[] [] arrayInts = new int[3][4]; // 2D c/3 linhas e 4 colunas  
Cat[] [] arrayGatos = new Cat[3][4]; // dá para 12 gatos
```

- Cada nova dimensão (max. 255) necessita de um conjunto novo de parêntesis retos

```
double[] [] [] arrayD = new double[3][4][3] // um cubo c/36 'caixas'
```

- Armazenamento e acesso:

```
arrayGatos[0][0] = new Cat("Miau"); // primeira 'célula'  
Cat g = arrayGatos[0][0];  
System.out.print(arrayGatos[0][0]); // ou g
```



# Iterar sobre arrays multi-dimensionais

```
for (int l = 0; l < arrayGatos.length; l++) {  
    for (int c = 0; c < arrayGatos[0].length; c++) {  
        Cat gato = arrayGatos[l][c]; // na linha l e coluna c  
        System.out.print(gato.getName());  
    }  
}
```

- Para saber o tamanho da segunda dimensão, perguntar a uma das linhas se todas tiverem o mesmo comprimento, senão, perguntar a `arrayGatos[1].length`

```
for (Cat[] linha : arrayGatos) { // para cada linha de Cats  
    for (Cat gato : linha) { // p/cada Cat da linha (i.e. do array de Ca  
        System.out.print(gato.getName());  
    }  
}
```



## Outras formas de criar arrays

```
int[] a = {1, 2, 3}; // 'a' tem length 3, e os valores passados
```

```
Cat[] cats = {new Cat("Miau"), new Cat("Lulu")};
```

```
int[][] b = {{1, 2, 3}, {4, 5}}; // 2D, mas linhas c/lengths diferentes
```

```
int[][] c = new int[4][]; // 2D, pode referir 4 linhas, mas não existem  
for (int i = 0; i < c.length; i++)  
    c[i] = new int[5]; // criar linhas c/length=5, mas podem ser difs
```

- Uma forma mais estranha:

```
int[] a = new int[]{1, 2, 3};
```



- Das mais utilizadas: `ArrayList` e `LinkedList`
- Ambas fazem exatamente o mesmo, mas internamente funcionam de forma diferente
- Fazem o mesmo, porque ambas implementam a interface `List`
- Uma interface é algo que define o que fazer e não como fazer
- Uma classe que implementa uma interface, está a assinar um contrato:  
*Comprometo-me a implementar os comportamentos definidos na interface*
- Tanto a `ArrayList` como `LinkedList` se comprometem a funcionar como `Lists`
- Assim, ambas têm de implementar os comportamentos definidos em `List`:  
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html>



# ArrayList e LinkedList

- Declaração:

```
List<Integer> listaInts; // ou ArrayList<Integer>, não recomendado!  
List<Cat> listaGatos;    // ou LinkedList<Cat>, não recomendado!
```

- Instanciação:

```
listaInts = new ArrayList<>();  
listaGatos = new LinkedList<>(); // ou LinkedList<Cat>(), redundante!
```

- Ou, de uma só vez: `List<Cat> listaGatos = new LinkedList<>();`

- Notas:

- ▶ Diamond operator <> permite indicar que tipo de objetos serão armazenados na List
- ▶ Apenas podemos armazenar objetos nas Lists, i.e., não podemos armazenar tipos primitivos como `int`, daí termos `List<Integer>` e não `List<int>`





# Exemplos de utilização

```
List<Integer> numeros = new ArrayList<>(); // ou LinkedList<>()
numeros.add(new Integer(4));
```

```
if (numeros.size() > 0) {
    System.out.println(numeros.get(0));
}
```

```
List<Cat> gatos = new ArrayList<>(); // ou LinkedList<>()
gatos.add(new Cat("Miau"));
```

```
if (gatos.size() > 0) {
    System.out.println(gatos.get(0).getName());
}
```



# Iterar sobre Lists

- O mesmo princípio que iterar sobre qualquer outro iterável

```
for (int i = 0; i < numeros.size(); i++)  
    System.out.println(numeros.get(i));
```

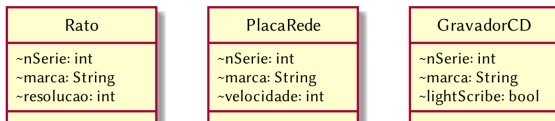
```
for (int n : numeros)  
    System.out.println(n);
```

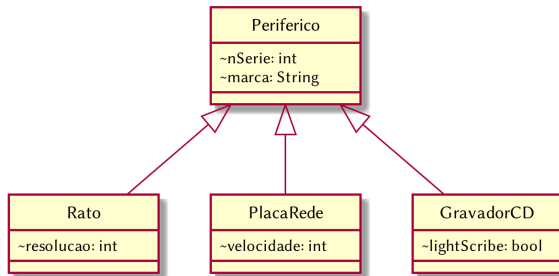
```
for (int i = 0; i < gatos.size(); i++)  
    System.out.println(gatos.get(i).getName());
```

```
for (Cat c : gatos)  
    System.out.println(c.getName());
```



- Mecanismo que permite criar “tipos de dados” (i.e., classes) mais especializados (mais específicos)
- Quando *especializar*?
  - ▶ Quando classes (dum mesmo domínio) começam a partilhar vários membros em comum
  - ▶ Normalmente, é óbvio:

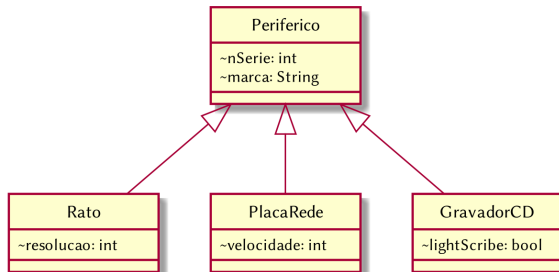




- A seta lê-se *is a*
- Rato *is a* Periférico, etc.
- A direção da seta é importante, ou seja, não podemos dizer “Periferico *is a* Rato”
- Se ao lermos o diagrama a relação *is a* não fizer sentido, cuidado!
- Periférico é um tipo mais geral
- Rato é um tipo especializado de Periférico
- As subclasses herdam os membros da superclasse
- Os objetos da subclasse sabem responder às mensagens definidas na superclasse, porque *is a*



# UML vs. Java



```
class Periferico {
    int nSerie;
    String marca;
}

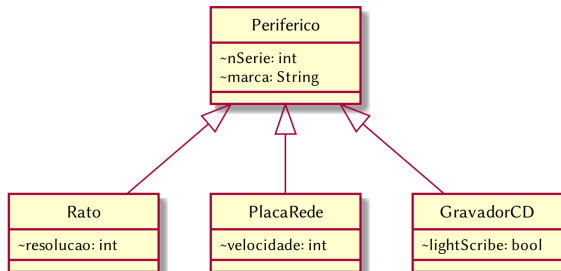
class Rato extends Periferico {
    int resolucao;
}

class PlacaRede extends Periferico {
    int velocidade;
}

class GravadorCD extends Periferico {
    boolean lightScribe;
}
```

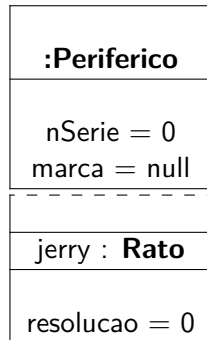


# Instanciar subclasse



`Rato jerry = new Rato();`

- Podemos imaginar um objeto em memória com o seguinte aspeto:



# Instanciar subclasse . . .

- Notas importantes:

- ▶ Quando instanciamos uma subclasse, se a superclasse:
  - ★ não tiver um construtor definido, a JVM chama o construtor vazio da superclasse automaticamente
  - ★ tiver construtor definido, é da responsabilidade da subclasse chamar o construtor da superclasse
- ▶ A chamada ao construtor da superclasse é feita:
  - ★ no construtor da subclasse, e é a primeira coisa a fazer antes de qualquer outra coisa
  - ★ através de `super()`, i.e., `super()` refere-se ao construtor da superclasse e, poderá ter, ou não, argumentos (i.e., depende do construtor da superclasse)



# Exemplo

```
class Periferico {  
    int nSerie;  
    String marca;  
  
    Periferico(int n, String m) {  
        nSerie = n;  
        marca = m;  
    }  
  
    int getSerie() {  
        return nSerie;  
    }  
  
    void setSerie(int n) {  
        nSerie = n;  
    }  
}
```

```
class Rato extends Periferico {  
    int resolucao;  
  
    Rato() {  
        super(0, "Nada");  
        resolucao = 0;  
    }  
  
    Rato(int n, String s, int res) {  
        super(n, s);  
        resolucao = res;  
    }  
}
```

- No main:

```
Rato r1 = new Rato();  
Rato r2 = new Rato(123, "Logitech", 300);  
System.out.print(r1.getSerie()); // 0  
System.out.print(r2.getSerie()); // 123
```





# Class abstrata

- Uma classe que tem um ou mais métodos sem implementação (i.e., abstratos)
- Uma classe abstrata tem de ser declarada como `abstract`, assim como todos os métodos sem corpo
- Quem herda dessa classe é obrigado a implementar os métodos abstratos, ou também tem de ser declarada como abstrata
- Uma classe abstrata não pode ser instanciada



# Exemplo

```
abstract class Periferico {  
    int nSerie;  
    String marca;  
  
    Periferico(int n, String m) {  
        nSerie = n;  
        marca = m;  
    }  
  
    int getSerie() {  
        return nSerie;  
    }  
  
    void setSerie(int n) {  
        nSerie = n;  
    }  
  
    abstract void imprimir();  
}
```

```
class Rato extends Periferico {  
    int resolucao;  
  
    Rato(int n, String s, int res) {  
        super(n, s);  
        resolucao = res;  
    }  
  
    void imprimir() {  
        System.out.println("Isto é um rato");  
        System.out.println("Marca: " + marca);  
        System.out.println("Resolução:" + resolucao);  
    }  
}  
  
● No main:  
Rato r2 = new Rato(123, "Logitech", 300);  
r2.imprimir();
```



## Coleções (e.g., ArrayList, LinkedList) de instâncias de subclasses

- Rato, PlacaRede, GravadorCD são todas subclasses de Periferico
- Por outras palavras, Rato, PlacaRede, GravadorCD *is a* Periferico
- Assim, instâncias destas podem “viver” numa coleção do tipo da superclasse:

```
Rato r = new Rato();  
PlacaRede p = new PlacaRede();  
List<Periferico> al = new ArrayList<>();  
al.add(r);  
al.add(p);  
al.add(new GravadorCD());
```

- O que permite fazer:

```
for (Periferico p : al):  
    p.imprimir();
```



# instanceof

- Permite “perguntar” ao objeto qual a classe da qual foi instanciado

```
Rato r = new Rato();  
PlacaRede p = new PlacaRede();  
List<Periferico> al = new ArrayList<>();  
al.add(r); // cast implícito de Rato p/Periférico  
al.add(p); // idem de PlacaRede  
al.add(new GravadorCD()); // idem de GravadorCD  
  
Periferico g = new GravadorCD(); // idem
```

```
for (Periferico p : al) {  
    if (p instanceof Rato)  
        System.out.println("É um rato");  
    else if (p instanceof GravadorCD)  
        System.out.println("É um gravador");  
    else  
        System.out.println("É o outro");  
}  
  
if (g instanceof GravadorCD)  
    System.out.println("É um gravador");
```



# Upcast vs. Downcast

## • Upcast

- ▶ conversão de um objeto de uma subclasse para o tipo da superclasse
- ▶ pode ser feito implicitamente ou explicitamente

```
Periferico g = new GravadorCD(); // upcast implícito  
Periferico h = (Periferico)new GravadorCD(); // upcast explícito
```

## • Downcast

- ▶ conversão de um objeto de uma superclasse para o tipo de uma das subclasses
- ▶ apenas pode ser feito explicitamente

```
Periferico p = new GravadorCD(); // upcast implícito  
GravadorCD i = p; // downcast implícito ERRO (Incompatible types)  
GravadorCD h = (GravadorCD)p; // downcast explícito OK  
Rato r = (Rato)p; // downcast explícito aceite, mas...
```

- ▶ O último caso é aceite pela JVM durante a compilação, mas vamos ter uma `ClassCastException` em runtime

