

Packages, Javadoc

Entrada e saída de dados

Strings e expressões regulares

Programação Orientada a Objetos, António J. R. Neves

(Agradecimento aos Docentes Ioulia Skliarova e José Luis Oliveira)

Pacotes

Um **pacote** (**package**) contém uma série de classes organizadas num único **espaço de nomes**. **Packages** permitem evitar conflitos de nomes.

As classes são referenciadas através dos seus nomes absolutos ou utilizando a cláusula **import**:

```
import java.util.Scanner; //importa a classe Scanner
import java.util.*; /*importa todas as classes da package
java.util*/
```

A cláusula **import** deve aparecer sempre na primeira linha do programa.

Uma vez importado o nome da classe, podemos referenciá-la por nome simples:

```
Scanner sc = new Scanner(System.in);
```

Sem **import** teríamos de escrever:

```
Scanner sc = new java.util.Scanner(System.in);
```

Criação de pacotes

Para criar um pacote (**package**) `poo`, devemos colocar na primeira linha de código:

```
package poo;
```

A classe pública dessa unidade de compilação (por exemplo, `Pessoa.java`) fará parte do *package* `poo`.

A sua utilização será na forma:

```
poo.Pessoa p = new poo.Pessoa();
```

ou

```
import poo.*  
Pessoa p = new Pessoa();
```

Caso não crie pacote nenhum, por omissão, todos os ficheiros `.java` numa pasta pertencem a um "*unnamed*" ou *default package*.

Javadoc

Java inclui um tipo de comentário especial que serve para **documentar** código. A documentação pode posteriormente ser extraída e representada num formato uniformizado.

A ferramenta que extrai comentários e gera documentos HTML é chamada **Javadoc**.

Todos os comandos de **Javadoc** ocorrem em comentários de tipo `/** * /`.

Existem 3 tipos de comentários de documentação: para classes, para atributos e para métodos. É possível utilizar código HTML dentro destes comentários (exceto *headings*).

O resultado é um ficheiro HTML que segue o mesmo formato que o resto da documentação Java – serviço para utilizadores.

É possível usar comandos adicionais, chamados **doc tags** que começam com o símbolo `@`.

Doc tags

Alguns exemplos de **doc tags**:

- `@see classname` – encaminhar para documentação da classe `classname`
- `@version version-information` – para indicar versão
- `@author author-information` – informação sobre autor do código
- `@since version` – para indicar a partir de que versão é que é suportada uma certa funcionalidade
- `@param parameter-name description` – para documentar parâmetros de métodos
- `@return description` – para descrever o valor de retorno dum método
- `@deprecated` – para indicar que uma funcionalidade não deve ser usada porque prevê-se que será removida no futuro

Exemplo de Javadoc

Exemplo:

```
package POO;
import java.util.*;

/**Displays a string and today's date
 * @author Iouliia Skliarova
 * @author Universidade de Aveiro
 * @version 1.0
 */
public class DateTime {
    /** @param strings an array of strings to display
     * @see java.util.Date
     * @see java.util.Arrays
     */
    public static void Test(String[] strings)
    {
        if (strings.length == 0)
            System.out.println("No info");
        else
            System.out.println(Arrays.toString(strings));
        System.out.println(new Date());
    }
    /** @param args an array of strings from command line
     */
    public static void main(String[] args)
    {
        DateTime.Test(args);
    }
}
```

Resultado

PACKAGE	CLASS	USE	TREE	DEPRECATED	INDEX	HELP
PREV PACKAGE	NEXT PACKAGE	FRAMES	NO FRAMES			

Package POO

Class Summary

Class	Description
DateTime	Displays a string and today's date

Method Detail

Test

```
public static void Test(java.lang.String[] strings)
```

Parameters:

strings - an array of strings to display

See Also:

Date, Arrays

main

```
public static void main(java.lang.String[] args)
```

Parameters:

args - an array of strings from command line

PACKAGE	CLASS	USE	TREE	DEPRECATED	INDEX	HELP
PREV CLASS	NEXT CLASS	FRAMES	NO FRAMES			
SUMMARY: NESTED FIELD CONSTR METHOD	DETAIL: FIELD					

POO

Class DateTime

java.lang.Object
POO.DateTime

```
public class DateTime  
extends java.lang.Object
```

Displays a string and today's date

Version:

1.0

Author:

Iouliia Skliarova, Universidade de Aveiro

Constructor Summary

Constructors

Constructor and Description

DateTime()

```
void POO.DateTime.Test(String[] strings)
```

Parameters:

strings an array of strings to display

See Also:

[java.util.Date](#)
[java.util.Arrays](#)

Escrita formatada

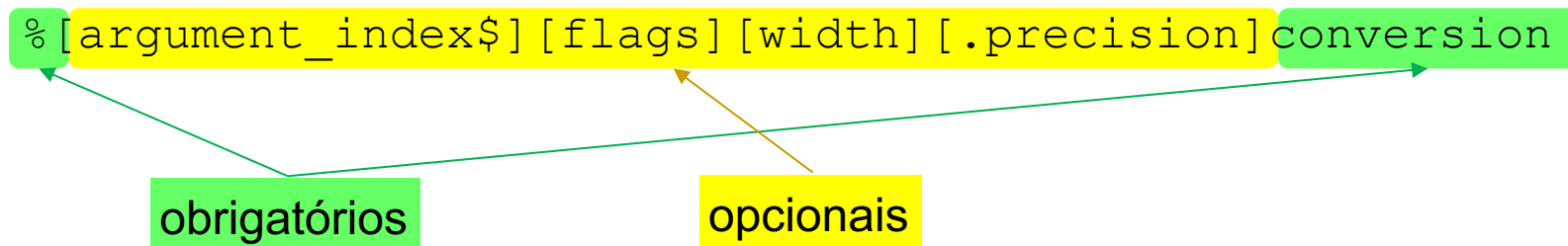
A classe **PrintStream** (que modela o *stream* de saída) inclui o método `printf(String format, Object... args)` que permite a escrita formatada.

Este método pode ser usado de modo seguinte:

```
System.out.printf("formato de escrita", lista de variáveis);
```

O formato de escrita é uma sequência de caracteres, que pode conter **especificadores de formato**.

O **especificador de formato** é composto pelo símbolo **%** seguido de um ou vários caracteres de acordo com o formato seguinte:



Especificadores de formato

`%[argument_index$][flags][width][.precision]conversion`

`conversion <obrigatório>` – é um carater que indica como formatar o argumento. O argumento formatado será inserido em vez do carater de conversão.

Alguns caracteres de conversão válidos:

b – valor booleano (**true** ou **false**)

s – **String**

c – carater

d – inteiro decimal (o – octal, x – hexadecimal)

f – valor real

Exemplo:

```
String str1 = "teste";
```

```
String str2 = "teste";
```

```
int a = 123;
```

```
double d = 1.234;
```

```
System.out.printf("%b\n%s\n%c\n%d\n%f\n",  
                  str1.equals(str2), str1, 'Y', a, d);
```

```
true  
teste  
Y  
123  
1.234000
```

Especificadores de formato (cont.)

O especificador de formato é composto pelo símbolo % seguido de um ou vários caracteres de acordo com o formato seguinte:

```
%[argument_index$][flags][width][.precision]conversion
```

`width <opcional>` - valor inteiro positivo que indica o número mínimo de caracteres que devem ser usados para o argumento. Se o comprimento do argumento é menor, os caracteres livres serão preenchidos com espaços.

Exemplo:

```
//continuação do exemplo anterior
System.out.printf("%10b\n%10s\n%10c\n%10d\n%10f\n",
str1.equals(str2), str1, 'Y', a, d);
System.out.printf("%1b\n%1s\n%1c\n%1d\n%1f\n",
str1.equals(str2), str1, 'Y', a, d);
```

```
true
teste
Y
123
1.234000
true
teste
Y
123
1.234000
```

Especificadores de formato (cont.)

O especificador de formato é composto pelo símbolo % seguido de um ou vários caracteres de acordo com o formato seguinte:

%[argument_index\$][flags][width][.precision]conversion

precision <opcional> - valor inteiro positivo que, quando usado para argumentos reais com conversão 'f', indica o número de dígitos na parte fracionária.

Exemplo:

```
double d = 1.234;  
System.out.printf("%5.2f\n", d);  
System.out.printf("%5.7f\n", d);
```

1.23
1.2340000

Especificadores de formato (cont.)

O especificador de formato é composto pelo símbolo % seguido de um ou vários caracteres de acordo com o formato seguinte:

```
%[argument_index$][flags][width][.precision]conversion
```

`argument_index <opcional>` - nº de argumento (da lista de variáveis) a usar neste ponto. Se não especificado, os argumentos serão usados pela ordem listada, um de cada vez.

Exemplo:

```
System.out.printf("%1$4d * %1$4d = %2$4d\n", 15, 15*15);  
//__15 * __15 = 225  
System.out.printf("%4d * %4d = %4d\n", 15, 15, 15*15);
```

Especificadores de formato (cont.)

`%[argument_index$][flags][width][.precision]conversion`

`flags <opcional>` – conjunto de caracteres que modificam o formato de saída.

- (– incluir números negativos em parênteses
- , – separar cada mil (em valores numéricos grandes) com vírgulas
- + – imprimir valores sempre com sinal
- 0 – preencher posições livres com 0
 - colocar um espaço à frente de valores positivos
- – ajustar à esquerda

Exemplo:

```
System.out.printf("%1$(10d\n%1$(10d\n%1$(10d\n%1$(10d\n%1$010d\n%1$ 10d\n%1$-10d\n", -1234);  
System.out.printf("%1$(7d\n%1$(7d\n%1$(7d\n%1$(7d\n%1$07d\n%1$ 7d\n%1$-3d\n", 23);
```

```
(1234)  
-1,234  
-1234  
-000001234  
-1234  
-1234  
23  
23  
+23  
0000023  
23  
23
```

Especificadores de formato (cont.)

Exemplo:

*/*Dado um tempo em segundos lido do teclado, mostre na consola o tempo com o formato hh:mm:ss. */*

```
Scanner sc = new Scanner(System.in);  
long segundos = sc.nextLong();  
System.out.printf("%02d:%02d:%02d\n", segundos / 3600,  
(segundos % 3600) / 60, segundos % 60);  
sc.close();
```

Especificadores de formato (cont.)

Exemplo:

*/*Dada uma tabela com temperaturas médias mínimas e máximas registadas em cada mes dum ano, imprima-a organizada em 12 linhas em formato ilustrado na figura*/*

```
double[][] tabelaTemperaturas =  
{ {-7.2, -6.7, -0.2, 4.6, 10.2, 13.8, 18.2, 18.0, 13.4, 8.3, 2.2, -0.5},  
  {-6.3, -5.5, 5.3, 7.7, 15.7, 19.9, 25.7, 24.5, 17.1, 13.3, 5.6, 0.5}};  
  
for (int mes = 0; mes < tabelaTemperaturas[0].length; mes++)  
{  
    System.out.printf("Mes %2d: ", mes + 1);  
    System.out.printf("%+8.2f%+8.2f",  
        tabelaTemperaturas[0][mes],  
        tabelaTemperaturas[1][mes]);  
    System.out.printf("\n");  
}
```

Mes 1:	-7.20	-6.30
Mes 2:	-6.70	-5.50
Mes 3:	-0.20	+5.30
Mes 4:	+4.60	+7.70
Mes 5:	+10.20	+15.70
Mes 6:	+13.80	+19.90
Mes 7:	+18.20	+25.70
Mes 8:	+18.00	+24.50
Mes 9:	+13.40	+17.10
Mes 10:	+8.30	+13.30
Mes 11:	+2.20	+5.60
Mes 12:	-0.50	+0.50

A classe **String**

A classe **java.lang.String** facilita a manipulação de cadeias de caracteres.

Objetos da classe **String** são **imutáveis** (constantes). Todos os métodos cujo objetivo é modificar uma **String**, na realidade constroem e devolvem uma **String** nova que incorpora as modificações pretendidas. A **String** original mantém-se inalterada.

Objetos de tipo **String** podem ser criados a partir dum vetor de **chars** ou de **bytes**.

Exemplo:

```
String str1 = "ABC";  
str1[0] = 'O'; //erro, String não é um vetor!  
char data[] = {'A', 'B', 'C'};  
String str2 = new String(data);
```

```
String str3 = new String("ABC"); /*não use esta construção,  
porque acaba por criar duas (!) Strings - pouco eficiente */
```


Teste de igualdade de Strings

Quando cria um literal de tipo **String**, JVM procura se este já existe na *pool* de **Strings**. Se existir, então é devolvida uma referência para a **String** existente.

Sendo assim, a *pool* de **Strings** permite reutilizá-las (i.e. ter na memória só uma instância da **String** a ser partilhada por várias referências). Isto é possível porque as **Strings** são imutáveis.

Exemplo:

```
String str1 = "ABC";  
char data[] = {'A', 'B', 'C'};  
String str2 = new String(data);  
String str3 = "ABC";  
  
System.out.println(str1 == str2); //false  
System.out.println(str1.equals(str2)); //true  
System.out.println(str1 == str3); /*true, ambas referenciam o mesmo  
objeto*/
```

Métodos da classe **String**

- Concatenação;
- Método **toString**;
- Comprimento e acesso a caracteres;
- Comparação
- “Modificação”
- Formatação
- Métodos que envolvem expressões regulares

Concatenação de Strings

Concatenação de **Strings** pode ser realizada com operadores `+` e `+=`. As primitivas são convertidas automaticamente para Strings.

```
String data = " feve" + "reiro ";  
data = 25 + data;  
data += "de " + 2015;  
System.out.println(data);
```

Para realizar a concatenação o compilador cria um objeto de tipo **StringBuilder** que representa uma sequência de caracteres **mutável**, chama para este métodos **append** várias vezes e finalmente o método **toString** para produzir o resultado final.

```
StringBuilder sb = new StringBuilder();  
sb.append(25);  
sb.append(" feve");  
sb.append("reiro ");  
sb.append("de ");  
sb.append(2015);  
data = sb.toString();  
System.out.println(data);
```

Concatenação de Strings (cont.)

O compilador nem sempre consegue fazer otimização com o **StringBuilder**. Em particular, se se pretende compor uma **String** dentro dum ciclo, é preferível o uso **explícito** de **StringBuilder** (com métodos **append**, **insert** e **toString**).

Exemplo:

```
String teste = "";  
for (int i = 0; i < 5; i++)  
    teste += i;          /* um novo objeto StringBuilder é  
construído em cada iteração do ciclo - pouco eficiente */  
System.out.println(teste);
```

```
StringBuilder sb = new StringBuilder();  
for (int i = 0; i < 5; i++)  
    sb.append(i);  
teste = sb.toString();  
System.out.println(teste);
```

Método toString

O método **toString** produz a versão “imprimível” dum objeto, i.e. converte um objeto numa **String**.

Todos os objetos (mas não as primitivas) têm este método implementado que é chamado sempre quando o compilador precisa duma **String** em vez do objeto, por exemplo:

```
Integer i = new Integer(22);  
System.out.println(i);
```

Nas classes criadas por utilizador o método **toString**, conforme implementado na classe **Object**, apenas imprime o nome da classe e o endereço do objeto respetivo.

Se pretender uma funcionalidade mais simpática, deve implementar o método **toString** explicitamente.

Método toString (cont.)

Exemplo:

```
Pessoa p = new Pessoa();  
p.setAge(17);  
p.setName("Paulo Ferreira");  
System.out.println(p); //Pessoa@55f96302 ☹
```

```
public String toString(){  
    return nome + " " + idade;  
}
```

```
System.out.println(p); //Paulo Ferreira 17 ☺
```

Comprimento e acesso a caracteres

O comprimento (número de caracteres) numa **String** pode ser determinado com o método **length**.

Acesso a caracteres numa **String** é feito com o método `charAt (int index)`.

Exemplo:

```
String s1 = "Aveiro";  
System.out.println(s1.length()); //6  
System.out.print(s1[0]); //erro  
System.out.print(s1.charAt(0)); //'A'
```

Comparação de Strings inteiras

As **Strings** completas podem ser comparadas usando métodos: **equals**, **equalsIgnoreCase**, **compareTo**.

Exemplo:

```
String s1 = "Aveiro";  
String s2 = "aveiro";
```

```
System.out.println(s1 == s2 ? "Iguais" : "Não Iguais");  
//Não Iguais  
System.out.println(s1.equals(s2) ? "Iguais" : "Não Iguais");  
//Não Iguais  
System.out.println  
(s1.equalsIgnoreCase(s2) ? "Iguais" : "Não Iguais"); //Iguais  
System.out.println(s1.compareTo(s2));  
// -1 (s1 menor), 0(iguais), 1 (s1 maior)
```


Comparação de subStrings

Pode-se analisar partes duma **String** com os métodos: **contains**, **regionMatches**, **startsWith**, **endsWith**.

Exemplo:

```
String s1 = "Aveiro";  
String s2 = "aveiro";
```

```
System.out.println(s1.contains("ve")); //true  
System.out.println(s1.regionMatches(1, s2, 1, 3)); //true  
System.out.println(s1.startsWith("ave")); //false  
System.out.println(s1.endsWith("ro")); //true
```

“Modificação” de Strings

Os métodos **replace**, **substring**, **toLowerCase**, **toUpperCase**, **trim** produzem uma **String** **nova** que inclui modificações pretendidas.

Exemplo:

```
String s1 = " Aveiro";  
System.out.println(s1.replace(" Aveir", "Port")); //Porto  
System.out.println(s1.substring(1, 3)); //Av  
System.out.println(s1.toLowerCase()); // aveiro  
System.out.println(s1.toUpperCase()); // AVEIRO  
System.out.println(s1.trim()); //Aveiro
```

Formatação de Strings

O método **format** retorna uma **String** nova formatada de acordo com os especificadores de formato (semelhante a **printf**).

Exemplo:

```
Scanner sc = new Scanner(System.in);  
long segundos = sc.nextLong();  
String s1 = String.format("%02d:%02d:%02d\n",  
    segundos / 3600, (segundos % 3600) / 60, segundos % 60);  
System.out.println(s1);  
sc.close();
```

Expressões regulares

Expressões regulares facilitam processamento de texto através de especificação de padrões que podem ser procurados em **Strings**.

A lista completa de construções suportadas está descrita na documentação da classe **java.util.regex.Pattern**.

Uma expressão regular pode incluir **carateres**, **símbolos** que especificam classes de carateres, **operadores lógicos** e **quantificadores**.

Exemplo:

`-?\d+` - padrão de um ou mais dígitos eventualmente precedidos com o sinal `'-'` (menos)

Carateres e classes de carateres

Uma expressão regular pode incluir **carateres** seguintes (e outros):

x	carater x
\\	carater \
\\t	carater <i>tab</i>
\\n	carater <i>newline (line feed)</i>
\\r	carater <i>carriage-return</i>
\\e	carater <i>escape</i>

Classes de carateres permitem especificar um conjunto (uma classe) de carateres:

[abc]	qualquer dos carateres a, b ou c
[^abc]	qualquer carater exceto a, b e c
[a-zA-Z]	qualquer carater das gamas (inclusivas) a-z ou A-Z
[a-d[m-p]]	qualquer carater da gama a-d ou m-p (união)
[a-z&&[def]]	qualquer dos carateres d, e ou f (interseção)
[a-z&&[^bc]]	qualquer dos carateres da gama a-z, exceto b e c (subtração)

Classes de caracteres predefinidas

Classes predefinidas especificam um conjunto de caracteres:

.	qualquer carater
\d	dígito de 0 a 9
\D	não dígito [^0-9]
\s	“espaço”: [\t\n\x0B\f\r]
\S	não “espaço”: [^\s]
\w	carater alfanumérico: [a-zA-Z_0-9]
\W	carater não alfanumérico: [^\w]

Delimitadores e operadores lógicos

Os **delimitadores** permitem separar logicamente entidades numa **String**:

<code>^</code>	início de linha
<code>\$</code>	fim de linha
<code>\b</code>	limite de palavra
<code>\B</code>	limite de não palavra
<code>\G</code>	fim do delimitador anterior
<code>\R</code>	qualquer mudança de linha

Os **operadores lógicos** controlam caracteres que serão procurados:

<code>XY</code>	X seguido de Y
<code>X Y</code>	X ou Y

Quantificadores

Os **quantificadores** especificam como um padrão é procurado no texto. Há vários tipos de quantificadores, iremos analisar apenas os de tipo **Greedy** que tencionam encontrar o maior pedaço de texto que satisfaça a um padrão.

$X?$ um ou nenhum X

X^* nenhum ou vários X

X^+ um ou vários X

$X\{n\}$ exatamente n X

$X\{n,\}$ pelo menos n X

$X\{n,m\}$ pelo menos n mas não mais de m X

Strings e expressões regulares

O método **matches** da classe **String** verifica se uma **String** inclui um dado padrão (especificado com a expressão regular).

Exemplo:

```
System.out.println("abcdefg".matches(".*"));  
//nenhum ou vários caracteres  
System.out.println("123".matches("\\d{2,4}"));  
//2-4 dígitos seguidos  
System.out.println("abcdefg".matches("\\w{3,}"));  
//pelo menos 3 caracteres alfanuméricos
```

Método `split`

O método **`split`** separa uma **`String`** em partes com base numa expressão regular e devolve o vetor de **`Strings`** resultantes.

Exemplo:

```
String frase = "Regular expressions are powerful and "
+ "flexible text-processing tools.";

String[] splitResult = frase.split("\\W");
//separar com base em caracteres não alfanuméricos
System.out.println(splitResult.length +
Arrays.toString(splitResult));
//9[Regular, expressions, are, powerful, and, flexible, text,
processing, tools]


splitResult = frase.split("ex");
System.out.println(splitResult.length +
    Arrays.toString(splitResult));
//4[Regular , pressions are powerful and fl, ible t, t-processing
tools.]
```

Substituição de subStrings

Os métodos **replaceFirst**, **replaceAll** permitem substituir partes de texto que correspondem a um dado padrão com outra **String**.

Casaco 49.89
Botas 64.99
Carteira 21.56

Talão para troca
Casaco *****
Botas *****
Carteira *****
Prazo: 3 dias



Exemplo:

```
String fatura = String.format
    ("%-10s%6.2f\n%-10s%6.2f\n%-10s%6.2f\n",
    "Casaco", 49.89, "Botas", 64.99, "Carteira", 21.56);
System.out.println(fatura);

String fatura_presente =
    fatura.replaceFirst("^", "Talão para troca\n");
fatura_presente =
    fatura_presente.replaceAll("\\d|\\.", "*");
fatura_presente =
    fatura_presente.replaceAll("$", "\nPrazo: 3 dias\n");
System.out.println(fatura_presente);
```

Separação de entrada

Por omissão a classe **Scanner** separa dados diferentes com base em “espaços”.

É entretanto possível definir outro delimitador usando a função **useDelimiter** e expressões regulares.

Exemplo:

```
Scanner sc = new Scanner
    ("Frases com muitos  espaços. No início  . E no fim.");
sc.useDelimiter("\\s*\\.\\s*"); //ponto com espaços à volta
while (sc.hasNext())
{
    String next = sc.next();
    System.out.println(next);
}
sc.close();
```

Frases com muitos espaços
No início
E no fim

Conceito de classe

Um programa = coleção de **objetos** que interagem entre si através de **mensagens**.

Cada **objeto** tem um tipo. Cada objeto é instância de uma **classe**. Classe = tipo.

Uma **classe** define os estados possíveis de objeto, o seu comportamento e o relacionamento com outros objetos.

Pode-se criar qualquer número de objetos numa classe. Vários objetos numa classe têm o mesmo comportamento mas guardam dados diferentes.

Todas as classes em Java são **derivadas** (i.e. são **uma espécie de**) da classe **Object** definida na biblioteca **java.lang**.

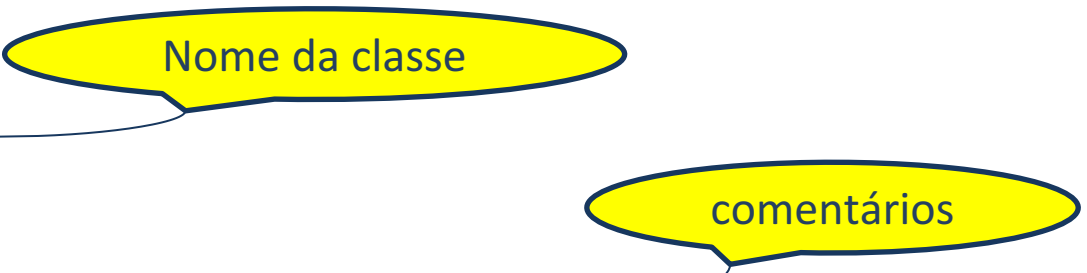
Exemplo:

Imagine que existe uma classe **Pessoa**. Cada **Pessoa** tem nome e data de nascimento. Mas **Pessoas** diferentes têm, em geral, nomes e datas diferentes.

Conceito de classe (cont.)

Definição duma classe (ficheiro `Pessoa.java`):

```
public class Pessoa
{
    //dados que vão compor o objeto
    /*métodos que vão responder às mensagens enviadas ao
       objeto*/
}
```



Java é uma linguagem **case-sensitive**.


O ficheiro que contém código fonte deve ter o nome `Xxx.java`, onde `Xxx` – é o nome da classe principal.

Esta classe deve ser declarada como pública (**public**) de modo a permitir que seja acessível a utilizadores (para que possam criar objetos desta classe).

Dados da classe

Vamos adicionar dois dados (*fields*) à classe **Pessoa**: nome e idade.

```
public class Pessoa
{
    //dados que compõem o objeto
    private String nome; // nome da pessoa
    private int idade;   // idade da pessoa
    /*métodos que vão responder às mensagens enviadas ao
    objeto*/
}
```



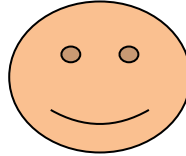
A yellow oval with a blue border contains the text "especificadores de acesso". Two blue arrows point from this oval to the `private` keywords in the code snippet above, specifically to the ones before `String nome` and `int idade`.

O atributo `nome` é uma **referência** para um objeto do tipo **String**. **String** é uma classe existente nas bibliotecas de Java (**java.lang**) que serve para modelar *strings* (sequências de caracteres).

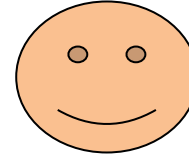
O atributo `idade` é uma variável de **tipo primitivo int** (que especifica um inteiro).

Intervenientes possíveis

Criador da classe

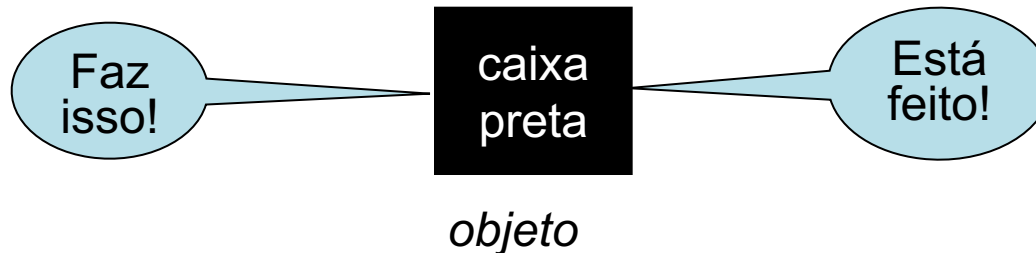


Utilizador da classe



- **Criador da classe** – uma pessoa ou uma equipa que desenvolve classes para serem usadas por outras pessoas/equipas.
- **Utilizador da classe** – pessoa/equipa que utiliza classes já existentes (desenvolvidas por outros) para poupar recursos, tempo, etc.

Em ideal, o cenário de utilização dum objeto deveria ser este:



Especificadores de acesso

Tal cenário é atingido com **especificadores de acesso** que servem para controlar acesso aos membros (atributos e métodos) de uma classe:

- **public** – significa que o membro pode ser acedido por **todos**
- **protected** – será considerado mais tarde
- **private** – significa que ninguém tem acesso para além dos métodos-membros da própria classe.

Os membros de uma classe por omissão (se se esquecer de colocar um especificador de acesso à frente dum membro) são de acesso **package**. Isso significa que o membro pode ser acedido por **todas as outras classes que pertencem ao mesmo pacote** (para já – que estejam definidas com ficheiros na mesma pasta).

Regra geral: deve declarar todos os atributos (membros-dados) como privados (**private**).

Porquê?

Há três razões principais que explicam porque se deve esconder ao máximo a estrutura e implementação duma classe:

- 1) Para garantir que os clientes, por engano, não estraguem algo importante. Se o cliente não tem acesso a um membro – torna-se impossível que mude o seu valor, por exemplo.
- 2) Para tornar claras as partes da classe que servem para o cliente usar e separar as partes que suportam a implementação interna da classe. Logo o cliente consegue facilmente filtrar o que é importante e separar a parte que pode ignorar.
- 3) O criador da classe tem a liberdade completa de mudar a estrutura e implementação interna da classe sem se preocupar que isto afete clientes. A única restrição: não mudar a interface da classe, i.e. mensagens a que a classe deve reagir (métodos públicos).

Métodos da classe

Vamos adicionar métodos que permitem consultar e alterar o nome e idade da pessoa.

```
public class Pessoa
{
    ...
    //métodos
    public String getName() { //lê nome
        return nome;
    }
    public void setName(String nome_novo) { //altera nome
        nome = nome_novo;
    }
    public int getAge() { //lê idade
        return idade;
    }
    public void setAge(int idade_nova) { //altera idade
        idade = idade_nova;
    }
}
```

Especificação de métodos em Java

Especificador de acesso. Já que o método é público pode ser usado por todos.

Tipo de valor que o método retorna.

Nome do método.

Parâmetros que o método recebe.

```
public int getAge() { //lê idade  
    return idade;  
}
```

```
public void setAge(int idade_nova) { //altera idade  
    idade = idade_nova;  
}
```

Implementação (corpo) do método.

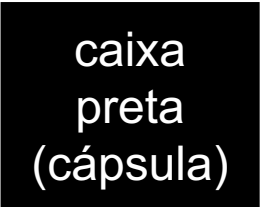
Regra geral: deve declarar métodos que definem a interface da classe (que sejam usados por clientes) como públicos (**public**).

Encapsulamento

Propriedade fundamental da programação orientada a objetos.

Encapsulamento = interligação de dados e métodos dentro de classes em combinação com controlo de acesso.

O resultado é criação de tipos de dados que possuam **caraterísticas** e **comportamentos**.



caixa
preta
(cápsula)

objeto

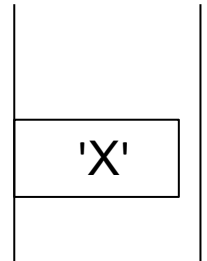
Objetos

Os **objetos** em Java são manipulados através de **referências**.

Para criar um objeto deve-se especificar o seu tipo e fornecer parâmetros necessários à sua construção.

Exemplo: para criar um objeto do tipo **Character**:

```
new Character('X');
```

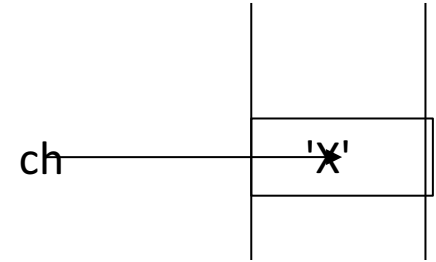


Este código reserva na **memória dinâmica** (*heap*) espaço necessário para armazenar um **Character** e a seguir executa um código especial responsável pela inicialização deste **Character** com o conteúdo fornecido na lista de parâmetros ('X'). Como resultado, será devolvida uma **referência** para o objeto criado.

Mas não a gravámos em lado nenhum. Logo não podemos utilizar o objeto criado ☹.

Objetos (cont.)

Solução:



```
Character ch = new Character('X');
```

Aqui `ch` é uma **referência** para o objeto **Character** criado dinamicamente (com o operador **new**).

Através de `ch` podemos facilmente localizar o nosso **Character** na memória.

Através da referência `ch` podemos agora manipular o objeto, i.e. enviar mensagens para ele.

Por exemplo o método `hashCode` permite determinar o código do **Character**:

```
int code = ch.hashCode(); //o resultado é 88
```

Objetos (cont.)

Tabela **Unicode** – *standard* para codificação e representação de símbolos usados em sistemas de computação:

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
0																				
20														!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~		€		,	f	†	‡	^	%o	Š	◀

Objetos (cont.)

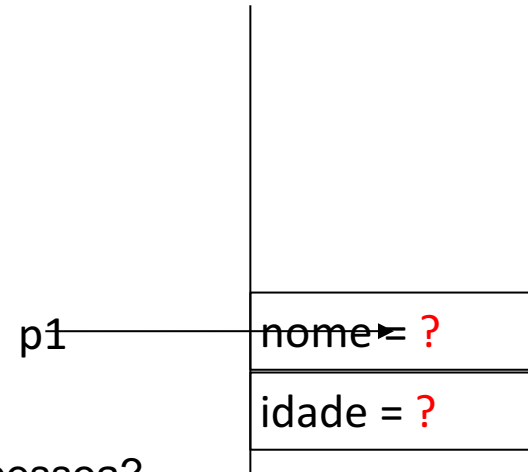
De maneira semelhante pode-se criar um objeto do tipo **Pessoa**:

```
Pessoa p1 = new Pessoa();
```

Aqui `p1` é uma **referência** para uma **Pessoa** criada dinamicamente (com o operador **new**).

Note que a lista de parâmetros está vazia.

```
public class Pessoa
{
    //dados que compõem o objeto
    private String nome;
    private int idade;
    ...
}
```



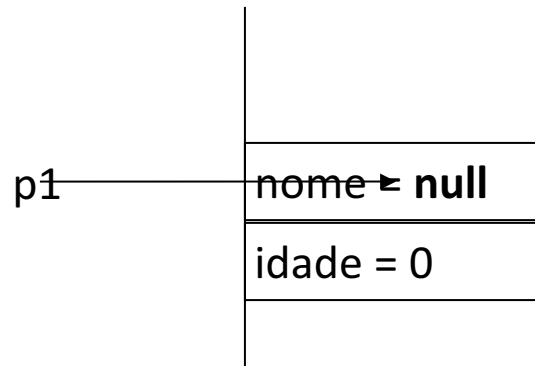
Que valores vão ter os atributos `nome` e `idade` desta pessoa?

Inicialização

Se não implementar nenhum método especial de inicialização (não temos), o compilador Java vai inicializar automaticamente todos os dados da classe com os seus **valores por omissão**.

O campo `nome` é uma referência (que deve referenciar uma **String**). O valor por omissão dum referência é **null** (i.e. não referencia nenhum objeto).

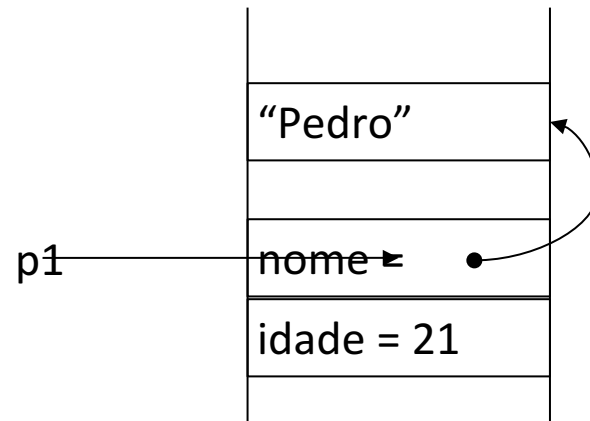
O dado `idade` é um inteiro (**int**). O valor por omissão dum inteiro é 0.



Envio de mensagens

Uma vez construído o objeto do tipo **Pessoa**, podemos enviar mensagens para ele, por exemplo, modificar o seu nome e idade:

```
Pessoa p1 = new Pessoa();  
p1.setName("Pedro");  
p1.setAge(21);
```



Tipo enum

A palavra reservada **enum** permite criar conjuntos de valores constantes.

Exemplo:

```
enum Mes
{
    JANEIRO, FEVEREIRO, MARÇO, ABRIL, MAIO, JUNHO,
    JULHO, AGOSTO, SETEMBRO, OUTUBRO, NOVENBRO, DEZEMBRO
}

...
Mes m = Mes.OUTUBRO; int dias;
switch (m)
{
    case ABRIL:
    case JUNHO:
    case SETEMBRO:
    case NOVENBRO: dias = 30; break;
    case FEVEREIRO: dias = 28; break;
    default: dias = 31;
}
```