

# Programação Avançada

Introdução à linguagem Java

# Linguagem Java

- 1<sup>a</sup> versão disponibilizada em 1995 pela *Sun Microsystems*
  - Estudo iniciado em 1991
- Linguagem totalmente orientada aos objetos
- *Cross-platform*
- A sintaxe segue o estilo C/C++
- Objetivos (<https://www.oracle.com/java/technologies/introduction-to-java.html>)
  - Simples, orientada ao objeto e familiar
  - Robusta e segura
  - Não dependente da arquitetura e portável
  - Alto desempenho
  - Interpretada, multitarefa e dinâmica

# Compilação de um programa Java

- A compilação, que inclui a verificação sintática e geração de código compilado, é realizada usando o ***Java Development Kit (JDK)***
  - Alguns ambientes de desenvolvimento possuem versões integradas
  - Inclui ferramentas e bibliotecas de classes
  - Inclui uma ferramenta para compilação: `javac`
  - É gerado código intermédio, designado *Java bytecode*
    - Código independente da arquitetura

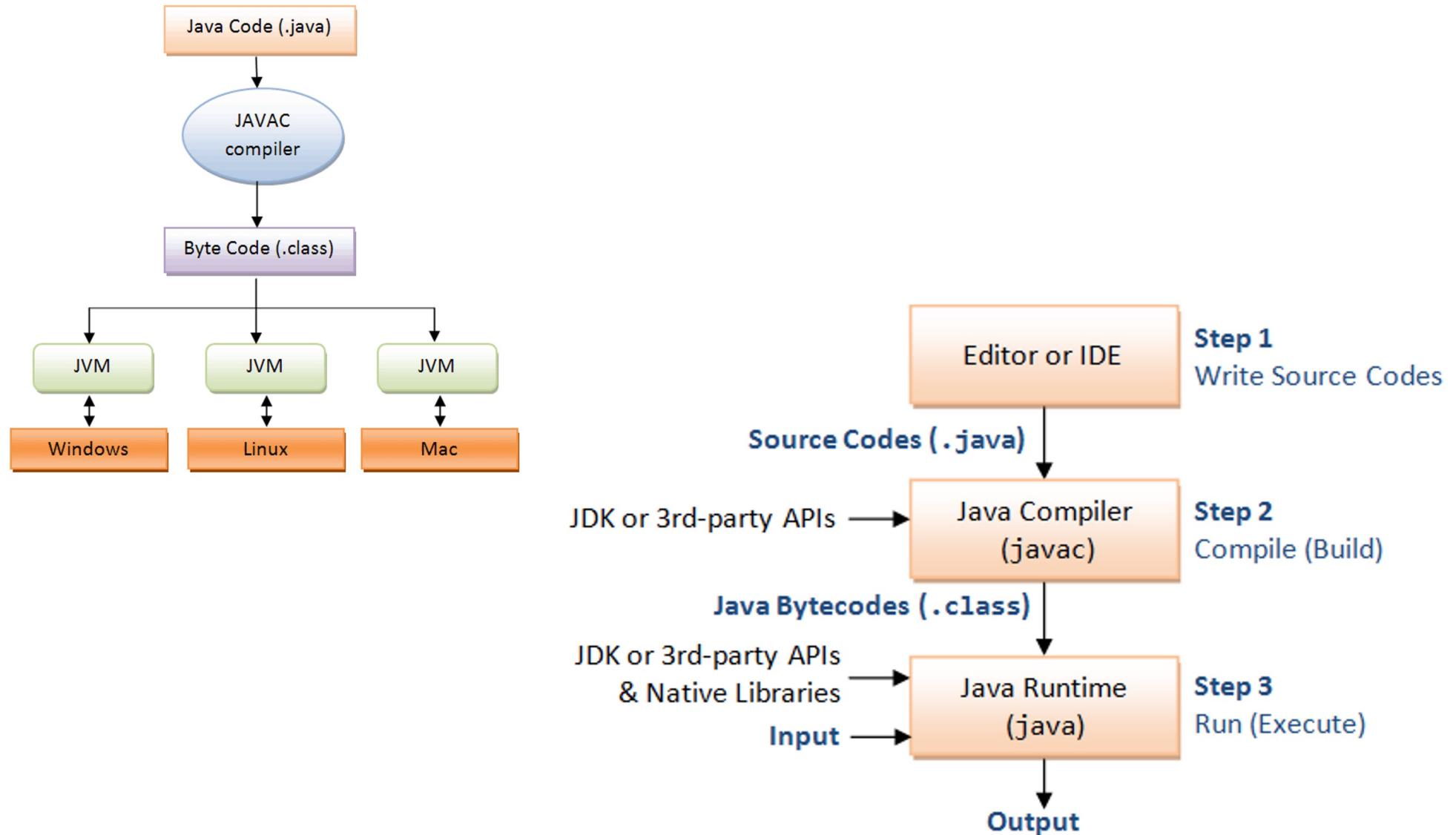
# Instalação do JDK

- *OpenJDK vs OracleJDK*
  - *OracleJDK*
    - Versão comercial, exigindo licenciamento
    - Suporte dado pela *Oracle*
    - Supostamente, mais estável e com melhor desempenho
  - *OpenJDK*
    - *Open source*, podemos contribuir no seu desenvolvimento
    - Embora o suporte seja essencialmente feito pelas comunidades ativas no seu desenvolvimento, existem distribuições específicas com suporte próprio

# Java Virtual Machine

- Como o resultado da compilação não é código nativo, é necessário possuir um interpretador de *Java bytecode*
- O intérprete de *Java bytecode* designa-se **Java Virtual Machine (JVM)**
  - Para que os programas Java possam ser executados numa determinada plataforma é necessário que exista uma JVM para cada
  - A JVM também oferece um ambiente seguro e isolado em que os programas são executados sem afectarem ou serem afetados por outros programas
  - Nota: não é necessário instalar o JDK nas máquinas onde se pretende apenas executar os programas, bastando instalar uma versão simplificada que suporta apenas a execução, designada por *Java Runtime Environment (JRE)*
    - Inclui comando "**java**" para executar os programas previamente compiladas em *Java bytecode*

# Criação e execução



# Exemplo

```
// Exemplo de um programa em Java
// Nome do ficheiro: Exemplo1.java

public class Exemplo1 {
    public static void main(String args[]) {
        System.out.println("Java@DEIS-ISEC");
    }
}
```

- Todo o código é encapsulado em classes ou similares
- Uma classe pública (no exemplo: `class Exemplo1`) é definida num ficheiro com o mesmo nome dessa classe e extensão `.java`
  - Podem existir várias classes no mesmo ficheiro, mas apenas uma pode ser pública
- A primeira função a ser executada num programa em *Java* é `public static void main(String args[])`

# Exemplo (compilação e execução)

- Depois de escrito o código deve ser gravado com o nome da classe pública presente no ficheiro de código e com extensão .java
  - Neste caso Exemplo1.java
- A compilação é realizada fazendo  
javac Exemplo1.java
  - Em caso de sucesso é criado o ficheiro Exemplo1.class
  - Caso existam erros estes são devidamente indicados através do número de linha no ficheiro em causa
- A execução é realizada fazendo  
java Exemplo1

```
PA@deis$ cat > Exemplo1.java
public class Exemplo1 {
    public static void main(String args[]) {
        System.out.println("Java@DEIS-ISEC");
    }
}
PA@deis$ javac Exemplo1.java
PA@deis$ java Exemplo1
Java@DEIS-ISEC
PA@deis$
```

# Linguagem Java – Conceitos básicos

Variáveis e tipos primitivos

*Arrays*

Operadores

Controlo de fluxo

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts>

# Tipos de dados primitivos

- Embora a linguagem Java ser totalmente orientada a objetos, são suportados os seguintes tipos de "dados primitivos":
  - **byte (8 bits)**: -128..127
  - **short (16 bits)**: -32768..32767
  - **int (32 bits)**:  $-2^{31}..2^{31}-1$
  - **long (64 bits)**:  $-2^{63}..2^{63}-1$
  - **float (IEEE754 32 bits)**
  - **double (IEEE754 64 bits)**
  - **boolean (true ou false => 1 bit)**
  - **char (16 bits)**: *Unicode*

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	\u0000'
String (or any object)	null
boolean	false

# Variáveis

- Permitem o armazenamento de informação tipificada ou referências para objetos
- A declaração de uma variável simples é realizada da seguinte forma:

***<tipo> <nome\_da\_variável> [ = <valor\_por\_omissão> ]***

- O nome da variável pode ser um conjunto de caracteres maiúsculos ou minúsculos, números ou os caracteres ‘\_’ ou ‘\$’ (este não deve ser usado!)
  - Não deve ser iniciada por um número

# Exemplo de variáveis tipificadas

```
// The number 26, in decimal
int decVal = 26;
// The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;

double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1 = 123.4f;

char c = 'A';
```

- Podem ser usados ‘\_’ para melhorar a legibilidade de números:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

... mas não são permitidos os seguintes casos:

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// OK (decimal literal)
int x1 = 5_2;
// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;
// OK (decimal literal)
int x3 = 5_____2;

// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// OK (hexadecimal literal)
int x6 = 0x5_2;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

# Conversões entre tipos (cast)

- As conversões entre tipos numéricos é realizada
  - Implicitamente
    - conversões para tipos que não implicam eventual perda de precisão. Ex.:
      - `int => long`
      - `int => float`
    - Explicitamente
      - conversões para tipos onde poderá ocorrer perda de informação. Ex.:
        - `long => int`
        - `float => int` (arredondamento "truncagem")

# Classes wrapper

- Sendo a linguagem Java totalmente orientada a objetos, para compatibilização no tratamento de todos os dados primitivos existe um equivalente definido através de um tipo de classe de objetos
  - Estas classes fornecem algumas funcionalidades que poderão ser úteis, por exemplo, para realizar conversões
    - Ex: `int i = Integer.parseInt("1234");`
- Classes equivalentes:
  - `byte`  $\Leftrightarrow$  `Byte`
  - `short`  $\Leftrightarrow$  `Short`, `int`  $\Leftrightarrow$  `Integer`, `long`  $\Leftrightarrow$  `Long`
  - `float`  $\Leftrightarrow$  `Float`, `double`  $\Leftrightarrow$  `Double`
  - `boolean`  $\Leftrightarrow$  `Boolean`
  - `char`  $\Leftrightarrow$  `Character`

# *Boxing e Unboxing*

- *Boxing* – conversão de dados primitivos para o objeto correspondente
  - `Integer i = 123;`
  - `Integer i = new Integer(123);`
    - `Integer i = Integer.valueOf(123);`
- *Unboxing* – conversão de instâncias das classes *wrapper* para os tipos primitivos correspondentes
  - `Double d1 = 123.45;`
  - `double d2 = d1;`
  - `double d3 = d1.doubleValue();`

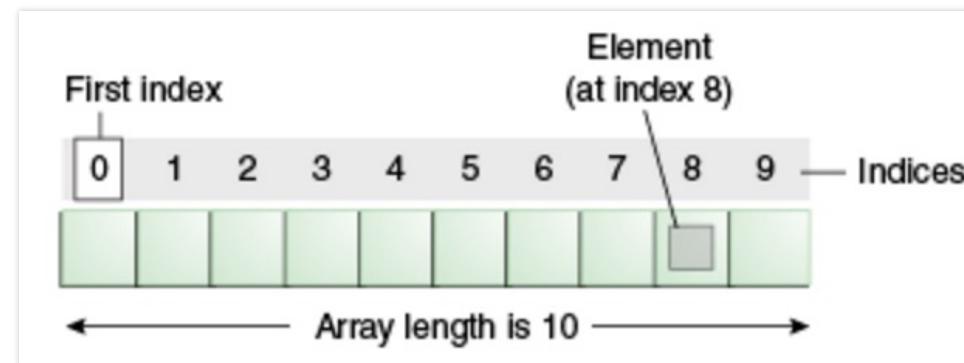
# Tratamento de números "grandes"

- BigInteger
- BigDecimal

```
BigDecimal bd1 = BigDecimal.valueOf(2e-323);
BigDecimal bd2 = BigDecimal.valueOf(3e-323);
BigDecimal bd3 = bd1.multiply(bd2);
System.out.println(bd3); // 6.00E-646
```

# Arrays

- Permitem o armazenamento de múltiplos valores
  - Suporte para dados primitivos ou referências para objetos
  - O número máximo de elementos é definido aquando da sua criação
  - O *array* possui um nome, correspondente ao nome da variável que o suporta
  - Cada valor é acedido através da variável do *array* e do respetivo índice (0..n-1)



# Declaração de Arrays

- A declaração pode ser realizada usando os seguinte formatos
  - $\langle tipo \rangle [ ] \langle nome \rangle$
  - $\langle tipo \rangle \langle nome \rangle [ ]$
- Exemplo:

```
int [ ] idades;
float meses [ ];
```

# Criação de *arrays*

- Quando são apenas declarados, os *arrays* possuem o valor `null`, correspondendo à não definição do *array*
- Para criar (definir) o *array* é necessário criar o número de células/valores pretendidos para o mesmo usando a palavra chave `new`

```
int [ ] idades;  
idades = new int[10];
```

```
float [ ] meses = new float[12];
```

# Exemplos de utilização de *arrays*

```
idades[0] = 10;
```

```
idades[1] = 15;
```

```
idades[2] = 20;
```

```
int i1 = idades[0] + 1;
```

```
float m = (idades[0]+idades[1])/2.0;
```

# Iniciação de arrays

- Quando um *array* é criado é possível iniciá-lo através da enumeração dos valores por omissão
  - A dimensão do *array* é deduzida a partir da quantidade de elementos enumerada

```
int [ ] tab1 = { 10, 20, 30, 40, 50 };  
char [ ] tab2 = { 'a', 'b', 'c' };
```

- A quantidade de elementos de um *array* pode ser facilmente obtida através da propriedade `length`  
`int n_items = tab1.length;`

# Arrays multidimensionais

- Em Java são permitidos *arrays* de múltiplas dimensões, os quais são declarados com a inserção de pares de [ ] por cada dimensão

```
int [ ][ ] array1 = new int [5][2];
```

```
double [ ][ ][ ] array2 = new double [5][4][3];
```

- O acesso é realizado da seguinte forma

```
array2[0][1][2] = array2[2][1][0] * 3.1415;
```

# Arrays multidimensionais

- Os *arrays* poderão ter um número de elementos variável por linha, nesse caso a criação de cada linha deve ser realizada de forma independente

```
int [][] b = new int[3][];
```

```
b[0] = new int[3];
```

```
b[1] = new int[4];
```

```
b[2] = new int[2];
```

# Iteração sobre arrays

- Obtenção de informação sobre o número de elementos de um *array*
  - Assumindo `int [][] b = ...`
    - `b.length` – número de linhas do *array* b
    - `b[i].length` – número de elementos da linha *i*
- Exemplo:

```
for ( int i=0 ;  i < b.length ; i++)
    for ( int j=0 ;  j< b[i].length ; j++){
        // ... b[i][j] ...
    }
```

# Variáveis do tipo *array*

- As variáveis do tipo *array* são referências para os objetos correspondentes
- Assim:

```
int [] t1 = {1,2,3};  
int [] t2;  
t2 = t1;  
t2[1] = 123;  
System.out.println(t1[1]); //123
```

# Métodos utilitários

- Existem várias classes e bibliotecas em Java que disponibilizam métodos úteis em situações diversas
- No caso de operações sobre *arrays* podem-se salientar:
  - `System.arraycopy(src, src_pos, dst, dst_pos, length)`
    - Permite copiar elementos de um *array* para outro, criado previamente
  - A classe `java.util.Arrays`
    - Disponibiliza um conjunto de métodos para trabalhar com *arrays*
      - `copyOf`, `copyOfRange`
      - `fill`
      - `compare`, `equals`, `mismatch`
      - `binarySearch`, `sort`, `stream`
      - `toString`

# Operadores

- Os operadores são similares aos existentes noutras linguagens (C/C++, C#, ...)

Operator Precedence	
Operators	Precedence
postfix	<i>expr++ expr--</i>
unary	<i>++expr --expr +expr -expr ~ !</i>
multiplicative	<i>* / %</i>
additive	<i>+ -</i>
shift	<i>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</i>
relational	<i>&lt; &gt; &lt;= &gt;= instanceof</i>
equality	<i>== !=</i>
bitwise AND	<i>&amp;</i>
bitwise exclusive OR	<i>^</i>
bitwise inclusive OR	<i> </i>
logical AND	<i>&amp;&amp;</i>
logical OR	<i>  </i>
ternary	<i>? :</i>
assignment	<i>= += -= *= /= %= &amp;= ^=  = &lt;=&gt;= &gt;&gt;&gt;=</i>

# Operador de atribuição =

- Diferentes comportamentos consoante o tipo de dados
  - Dados primitivos
    - é feita a cópia do valor
  - Objetos
    - As variáveis armazenam apenas a referência para o objeto
    - é feita a cópia da referência
      - O objeto referenciado por ambas as variáveis é o mesmo
- A passagem de parâmetros segue a mesma lógica do operador de atribuição

# *Garbage Collector*

- Através das operações de atribuição, passagem de parâmetros e outras similares, relativas a referências para objetos, o número de referências a essas instâncias vai sendo gerido centralmente, de forma automática
- Quando o número de referências é zero isso significa que não existe forma de aceder ao objeto e, como tal, o objeto fica marcado para eliminação
  - Não é responsabilidade do programador libertar a memória associada

# *Garbage Collector*

- A gestão da memória que vai ser reservada e a posterior libertação é feita pelo *Garbage Collector*
  - Quando os objetos são marcados para eliminação o método `finalize()` é invocado
    - Não existe garantia de quando este método é chamado
    - Marcado como *deprecated* desde a versão 9
  - Depois do método `finalize` ser chamado a memória associada ao objeto será libertada, mas sem garantia de quando o será.

# Operadores == e !=

- Quando os operadores == e != são usados com operandos do tipo referência, é verificado se os dois operandos referem o mesmo objeto

```
Integer a1 = new Integer(123);
Integer a2 = new Integer(123);
Integer a3 = a1;
System.out.println(a1==a2);          // false
System.out.println(a1==a3);          // true
System.out.println(a2==a3);          // false
System.out.println(a1.equals(a2));    // true
```

# instanceof

- O operador `instanceof` permite verificar se uma determinada referência corresponde a um objeto de determinado tipo

```
if (value1 instanceof Integer) {  
    ...  
}
```

# Controlo de fluxo – if

- *if-then*

```
if (condition) {  
    ...  
}
```

```
if (condition1) {  
    ...  
} else if (condition2) {  
    ...  
} else if (condition3) {  
    ...  
} else {  
    ...  
}
```

- *if-then-else*

```
if (condition) {  
    ...  
} else {  
    ...  
}
```

# Controlo de fluxo – switch, break

- *switch-case*

```
switch (source) {  
    case op1:  
    [case op2:]  
        ...  
        break;  
    ...  
    default:  
        ...  
        break;  
}
```

- *switch-arrow case*

```
switch (source) {  
    case op1 -> ... ;  
    case op2 -> ... ;  
    ...  
    default -> ... ;  
}
```

- Não são necessários *breaks*

- Podem ser usadas *strings* nos *cases*

# Controlo de fluxo – ciclos

- *for*

```
for(<init>; <condition>; <update>) {  
    ...  
}
```

- *while*

```
while (condition) {  
    ...  
}
```

- *for-each*

```
for(<type> <var>:<collection/array>) {  
    ...  
}
```

- *do-while*

```
do {  
    ...  
} while (condition);
```

# Controlo de fluxo

- Outras instruções de controlo de fluxo na execução de um programa Java
  - `continue`
  - `break`
    - As instruções *continue* e o *break* podem ser seguidas por uma *label* associada ao ciclo em que devem atuar
- `return`

# Interação com o utilizador

- O Java disponibiliza um conjunto de objetos que permitem representar as entradas e saídas típicas de um programa/processo
  - `System.in` – representa a entrada de dados por omissão, normalmente associada à entrada de dados da consola
  - `System.out` – representa a saída de dados por omissão, normalmente associada à saída de dados para a consola
  - `System.err` – representa a saída para erros, normalmente associada à saída de dados para a consola

# Saída de dados

- `System.out.print(...)`
- `System.out.println(...)`
- `System.out.printf(<format>, param1, param2, ...)`
- `System.err.print(...)`
- `System.err.println(...)`
- `System.err.printf(<format>, param1, param2, ...)`

# Entrada de dados

- Embora o objeto `System.in` permita obter dados introduzidos pelo utilizador na consola, esses dados são interpretados como *bytes*

- Para facilitar o acesso a essa informação, de forma mais facilitada e tipificada, pode-se usar um objeto `Scanner`

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

- Um objeto `Scanner` permite ler sequências de caracteres separados por delimitadores.

```
int n = sc.nextInt();           // Lê um inteiro  
double x = sc.nextDouble();    // Lê um double
```

- Por omissão, os delimitadores são os espaços em branco, *tabs* e mudanças de linha. No entanto pode ser definido outro delimitador:

```
sc.useDelimiter("-");
```

- É possível testar o tipo do próximo valor a ser lido:

```
if( sc.hasNextDouble() ) // Verifica se próximo valor é um double  
x = sc.nextDouble();
```

# Classe Math

- A classe Math, pertencente ao package `java.lang`, disponibiliza um conjunto de funções matemáticas
  - `sin`, `cos`, `tan`
  - `abs`, `round`, `floor`, `ceil`, `rint`
  - `max`, `min`
  - `pow`, `exp`
  - `hypot`, `sqrt`
  - `random`
  - `toDegrees`, `toRadians`
  - `E`, `PI`

# Math.random

- Método que gera números pseudo-aleatórios
  - `double r = Math.random();`
    - $0.0 \leq r < 1.0$
  - Exemplo para gerar números entre 1 e 100, inclusive
    - `int i = (int) (Math.random() * 100) + 1;`

# Classe Random

- No package `java.util` é disponibilizada a classe `Random` a qual possui um conjunto de métodos que permitem um acesso mais flexível a sequências de números pseudoaleatórios

```
Random rnd = new Random();
int i = rnd.nextInt();
int j = rnd.nextInt(100) + 1;
double d = rnd.nextDouble();

...
rnd.setSeed(1234);

...
```

# String

- A classe `String` representa uma cadeia de caracteres imutável
  - A alteração de uma `String` (por exemplo, concatenando através do operador '+') origina sempre a criação de uma nova `String`
- Disponibiliza métodos utilitários para trabalhar com os caracteres presentes na `String`
  - `equals`, `equalsIgnoreCase`, `matches`, `compareTo`, `compareToIgnoreCase`, `startsWith`, `contains`, `endsWith`, `indexOf`
  - `isBlank`, `isEmpty`
  - `concat`, `replace`, `replaceAll`, `repeat`, `trim`
  - `split`
  - `toUpperCase`, `toLowerCase`
  - ...

# String

- Existem alguns pontos sobre a classe *String* a salientar, que se podem verificar pelo exemplo seguinte

```
String s1 = "DEIS-ISEC";
String s2 = new String("DEIS-ISEC");
String s3 = s2;
String s4 = "Deis-Isec";

System.out.println(s1 == s2); // false
System.out.println(s2 == s3); // true
s3 = s1;
System.out.println(s2 == s3); // false
System.out.println(s1.equals(s2)); // true
System.out.println(s2.equals(s4)); // false
System.out.println(s2.equalsIgnoreCase(s4)); // true
```

# StringBuffer e StringBuilder

- As classes `StringBuffer` e `StringBuilder` permitem gerir *strings* mutáveis
- Ambas as classes disponibilizam métodos diversos para trabalhar com *strings*, incluindo métodos para adicionar novos caracteres ou *strings*, modificar caracteres pontuais ou *substrings*, ...
- Diferenças principais
  - `StringBuffer` – *thread-safe*
  - `StringBuilder` – mais rápida

# Linguagem Java

Programação orientada a objetos

# Programação orientada a objetos

- Sendo a linguagem Java totalmente orientada a objetos, inclui todos os princípios associados a este tipo de programação, nomeadamente:
  - Abstração e Encapsulamento
    - Definir novos tipos de objetos

```
class <nova_classe> { ... }
```
    - Esconder detalhes de implementação
    - Controlar acesso à informação
    - Generalizar utilização
  - Herança
    - Definição de um novo tipo de objeto como especialização de outro tipo de objeto

```
class <nova_classe> extends <classe_base> { ... }
```
    - São herdadas as características do objeto base
    - Podem ser definidas novas características e comportamentos adequados à especialização em causa
  - Polimorfismo
    - Redefinição (@Override) dos comportamentos declarados e/ou definidos na classe base, permitindo a sua execução de forma genérica

# Classes

- Para definir uma nova classe de objetos usa-se a seguinte sintaxe:

```
class <nome> {  
    <variáveis>  
    <métodos/funções>  
}
```

- Não é obrigatório que as variáveis sejam definidas todas no início ou os métodos no fim, mas é uma boa política para a sua organização

# Classes

- Exemplo:

```
class Ponto {  
    int x,y;  
  
    void move(int dx,int dy) {  
        x += dx;  
        y += dy;  
    }  
}
```

- Para criar um objeto de um classe usa-se a instrução **new**

```
Ponto p = new Ponto();
```

# Métodos *overloaded*

- Podem existir vários métodos com o mesmo nome desde que possam ser distinguidos pelos seus parâmetros
- A distinção pode ser realizada em
  - Número de parâmetros
  - Tipo dos parâmetros
- O tipo definido para o retorno do métodos não é usado para distinguir dois métodos *overloaded*

# Construtores

- Sempre que um objeto é criado, é chamado automaticamente um construtor para fazer a iniciação desse objeto
  - Os construtores são métodos com o mesmo nome da classe e sem tipo de retorno declarado
  - Podem existir vários construtores, com diferentes parâmetros
    - Um construtor sem parâmetros é designado "construtor por omissão"
  - Em Java as variáveis não iniciadas explicitamente são iniciadas com valores por omissão (0 ou null)

```
class Ponto {  
    int x,y;  
  
    Ponto(int xi, int yi) {  
        x = xi;  
        y = yi;  
    }  
}
```

# this

- No contexto de qualquer instância de um determinado tipo de objeto, poder-se-á ter acesso à sua referência (autorreferência) através da palavra-chave `this`

```
class Point {  
    int x, y;  
  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    // ...  
}
```

# Delegação de construtores

- Um construtor pode delegar a construção de um objeto outro construtor do mesmo objeto (quando existem construtores *overloaded*)
  - A delegação faz-se através da utilização do *this* como se de uma função se tratasse e cujos parâmetros são os parâmetros do construtor no qual se vai delegar a construção do objeto

```
class Point {  
    int x, y;  
  
    Point() {  
        this(0,0);  
    }  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    //...  
}
```

# Membros estáticos

- Na definição de uma classe os diversos membros, variáveis ou funções, podem ser definidos como sendo estáticos
  - Utilização da etiqueta `static`
- Os membros estáticos...
  - Podem ser acedidos/usados sem que existam instâncias desse tipo de objeto
    - A sua utilização faz-se através da indicação do nome da própria classe e não de uma referência para um objeto desse tipo
  - Os valores das variáveis são acedidos e passivos de alteração, sendo partilhados pelas eventuais instâncias existentes desse tipo de objeto
    - As alterações realizadas a variáveis estáticas no contexto de uma instância são visíveis para todas as outras instâncias

# Membros estáticos

- Os métodos estáticos apenas podem chamar outros métodos estáticos
  - Podem chamar métodos de instâncias desde que devidamente enquadrados nas respectivas instâncias
- No contexto de membros estáticos não se usa a referência *this*, uma vez que os membros estáticos não estão associados a qualquer instância
- As variáveis estáticas apenas são iniciadas na primeira utilização da classe
  - Os valores podem ser iniciados por atribuições diretas ou através de um construtor estático
    - Definido através de um bloco `static { ... }`

# Membros estáticos

```
class Person {  
    static int count;  
    static {  
        count = 101;  
    }  
    static void resetCounter() {  
        count = 1;  
    }  
  
    int id;  
    String name;  
  
    Person(String name) {  
        id = (count++);  
        this.name = name;  
    }  
    void show() {  
        System.out.printf("%5d - %s\n", id, name);  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Person p1 = new Person("João Felix");  
        Person p2 = new Person(  
            "Cristiano Ronaldo");  
        p1.show();  
        p2.show();  
        Person.resetCounter();  
        var p3 = new Person("Eusébio Ferreira");  
        p3.show();  
    }  
}
```

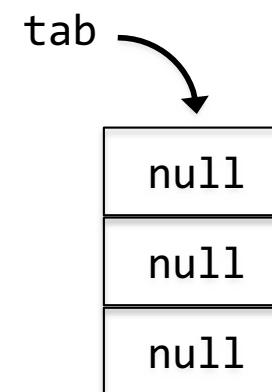
Output:

```
101 - João Felix  
102 - Cristiano Ronaldo  
1 - Eusébio Ferreira
```

# Arrays de objetos

- Um *array* de objetos quando é criado, sem serem especificados os elementos de forma explícita, contém apenas referências nulas (`null`)

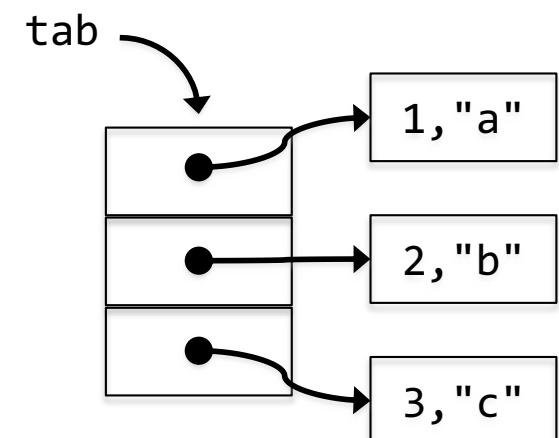
```
Person [] tab = new Person[3];
```



# Arrays de objetos

- Um *array* de objetos quando é criado, sem serem especificados os elementos de forma explícita, contém apenas referências nulas (`null`)

```
Person [] tab = new Person[3];
tab[0] = new Person("a");
tab[1] = new Person("b");
tab[2] = new Person("c");
```



# Arrays de objetos

- Os *arrays* de objetos podem ser criados por enumeração.

```
Person [] tab = {new Person("a"),new Person("b")};  
for(var p : tab)  
    p.show();  
Person [][] groups = {  
    { new Person("a1"),new Person("a2")},  
    { new Person("a3"),new Person("a4")}  
};  
for(var group:groups) {  
    for (var student : group)  
        System.out.print("\t" + student.name);  
    System.out.println();  
}
```

- Quando é necessário passar por parâmetro um *array* de objetos criado por enumeração, pode-se usar o seguinte formato:  
`proc(new Person[] { new Person("a"), new Person("b")});`

# Packages

- As classes são agrupadas em *packages*
  - Organização de código
    - Constituição de *namespaces* para nomear as classes
  - Permite evitar conflito de nomes entre diversas classes
    - Quando existe conflito de nomes entre classes de diferentes *packages*, dever-se-á indicar o seu nome completo, ou seja, incluído o nome do package: <package>.<classe>
  - Facilitar a distribuição de aplicações
    - Forma mis simples de disponibilizar os programas em Java para utilização pelos utilizadores finais
  - Facilitar reutilização
    - Muitas funcionalidades poderão ser reutilizadas noutros projetos através da constituição e disponibilização de *packages* com classes de objetos
      - Segue a mesma lógica da utilização de todas as classes que constituem a *Java API*

# Packages - criação

- Definição do *package*
  - O *package* possui um nome definido através da instrução "package <nome>;" indicada, normalmente, como primeira linha efetiva num ficheiro de código
    - O nome base é habitualmente constituído por duas informações distintas, em letra minúscula
      - Nome do domínio da empresa por ordem inversa (deis.isec.pt => pt.isec.deis)
      - Nome do projeto concreto
    - Exemplo: package pt.isec.a200212345.proj\_aula1;
  - Os ficheiros que incluem as classes pertencentes a um *package* devem ser colocados numa hierarquia de diretórios correspondente ao nome do *package*

```
Proj_Aula1
  pt
    isec
      a200212345
        proj_aula1
          Exemplo1.java
```

# Packages - utilização

- Para utilizar as classes ou outros elementos definidos noutras *packages* deve-se usar a instrução **import**
  - Podem existir várias instruções de importação (usual!)
  - Podem ser incluídos todos os elementos de um package: `import pt.isec.pa.utils.*;`
  - Pode ser incluído apenas o elemento pretendido: `import pt.isec.pa.utils.FileUtils;`

# Packages - exemplo

```
PA@deis$ cat Exemplo1.java
package pt.isec.pa.exemplo1;

public class Exemplo1 {
    public static void main(String args[]) {
        System.out.println("Java@DEIS-ISEC");
    }
}
PA@deis$ javac Exemplo1.java
PA@deis$ java Exemplo1
Error: Could not find or load main class Exemplo1
Caused by: java.lang.NoClassDefFoundError: pt/isec/pa/exemplo1/Exemplo1 (wrong name: Exemplo1)
PA@deis$ mkdir -p pt/isec/pa/exemplo1
PA@deis$ mv Exemplo1.class pt/isec/pa/exemplo1
PA@deis$ java pt.isec.pa.exemplo1.Exemplo1
Java@DEIS-ISEC
PA@deis$
```

# Packages - exemplo

```
PA@deis$ cat Exemplo1.java
package pt.isec.pa.exemplo1;

public class Exemplo1 {
    public static void main(String args[]) {
        System.out.println("Java@DEIS-ISEC");
    }
}
PA@deis$ javac -d . Exemplo1.java
PA@deis$ java pt.isec.pa.exemplo1.Exemplo1
Java@DEIS-ISEC
PA@deis$
```

# Packages - acesso

- Para além das vantagens de organização e reutilização que os *packages* oferecem, também permitem controlar o acesso a particularidades de implementação dos elementos que definem
- O controlo de acesso dos diversos membros é realizado através das palavras-chave `public`, `protected`, `private` ou pela sua não especificação, tendo em conta a seguinte tabela

Etiqueta	classe	<i>package</i>	subclasse	outros
<code>public</code>	Sim	Sim	Sim	Sim
<code>protected</code>	Sim	Sim	Sim	Não
<i>(sem etiq.)</i>	Sim	Sim	Não	Não
<code>private</code>	Sim	Não	Não	Não

- Nota: as classes apenas permitem `public` ou sem etiqueta

# Ficheiros jar

- Para facilitar as tarefas de *deployment* de programas Java, as classes que constituem esse programa, bem como outros recursos (imagens, sons, ...), deverão ser incluídos(as) num ficheiro jar
  - O ficheiro jar é na realidade um ficheiro zip
  - Permite manter de forma simples a hierarquia de diretórios que representam os vários *packages* que poderão constituir um programa
  - É incluído um ficheiro de manifesto que permite especificar atributos especiais
- Criar ficheiro jar  
`jar cf exemplo1.jar pt/*`
- Executar jar  
`java -cp exemplo1.jar pt.isec.pa.exemplo1.Exemplo1`

# Ficheiros jar executáveis

- A quando da criação de um ficheiro jar pode-se especificar uma classe que inclui uma função main, a qual irá ser executada quando o ficheiro jar for executado com: `java -jar <ficheiro.jar>`

```
PA@deis$ javac -d . Exemplo1.java
PA@deis$ jar cfe exemplo1.jar pt.isec.pa.exemplo1.Exemplo1 pt/*
PA@deis$ java -jar exemplo1.jar
Java@DEIS-ISEC
PA@deis$ mkdir temp
PA@deis$ cp exemplo1.jar temp/exemplo1.zip
PA@deis$ cd temp
PA@deis$ unzip exemplo1.zip
Archive: exemplo1.zip
  creating: META-INF/
  inflating: META-INF/MANIFEST.MF
  creating: pt/isec/
  creating: pt/isec/pa/
  creating: pt/isec/pa/exemplo1/
  inflating: pt/isec/pa/exemplo1/Exemplo1.class
PA@deis$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: 17.0.2 (Oracle Corporation)
Main-Class: pt.isec.pa.exemplo1.Exemplo1

PA@deis$
```

# Ficheiro de manifesto

- O ficheiro de manifesto, criado automaticamente no exemplo anterior, pode ser criado e configurado de forma manual

```
PA@deis$ cat > MANIFEST.TXT
Main-Class: pt.isec.pa.exemplo1.Exemplo1
PA@deis$ jar -cfm exemplo1.jar MANIFEST.TXT pt/*
PA@deis$ java -jar exemplo1.jar
Java@DEIS-ISEC
PA@deis$ mkdir temp
PA@deis$ cp exemplo1.jar temp/exemplo1.zip
PA@deis$ cd temp
PA@deis$ unzip exemplo1.zip
Archive: exemplo1.zip
      creating: META-INF/
      inflating: META-INF/MANIFEST.MF
      creating: pt/isec/
      creating: pt/isec/pa/
      creating: pt/isec/pa/exemplo1/
      inflating: pt/isec/pa/exemplo1/Exemplo1.class
PA@deis$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Main-Class: pt.isec.pa.exemplo1.Exemplo1
Created-By: 17.0.2 (Oracle Corporation)

PA@deis$
```

# Composição vs Herança

- Uma classe, por si só, permite representar um determinado conceito
- Na maior parte das situações, o conceito será definido com o auxílio da especificação de características ou propriedades, representadas com o auxílio de variáveis
- Dependendo da complexidade e da situação concreta, a definição de um objeto poderá recorrer a diferentes modelos...
  - Composição
    - Um objeto é definido através da composição de vários outros objetos
      - Um computador é constituído pela *motherboard*, o processador, a placa gráfica, ...
  - Herança
    - Um objeto é definido como uma especialização de um outro objeto mais genérico
    - São herdadas todas as características que definem um tipo de objeto base e são apenas programadas as diferenças para o objeto base
      - Um carro com mudanças automáticas é um carro, mas com um forma diferente de gerir a mudança ativa
  - Ambas
- Estas técnicas permitem usufruir das características e comportamentos de outros objetos, sem ter que repetir a sua implementação

# Composição

```
class Student {  
    long nr;  
    String name;  
  
    public Student(long nr, String name) {  
        this.nr = nr;  
        this.name = name;  
    }  
    // ...  
}  
  
class Group {  
    Student s1;  
    Student s2;  
  
    public Group(Student s1, Student s2) {  
        this.s1 = s1;  
        this.s2 = s2;  
    }  
    // ...  
}
```

# Herança

- Para criar uma nova classe de objetos a partir de uma outra, herdando todas as características dessa, utiliza-se a seguinte sintaxe:

```
class <nova_classe> extends <classe_base> {  
    ...  
}
```

- Não sendo definida qualquer informação ou comportamento adicional, as instâncias da classe derivada são funcionalmente idênticas às da classe base
- Em Java apenas é possível derivar uma classe a partir de uma única classe
  - Não existe herança múltipla

# Herança

- No contexto da classe derivada podem-se referir características da classe base usando a palavra-chave `super`
  - `super.<variavel_da_classe_base>`
  - `super.<método_da_classe_base>([<params, ...>]);`
- Numa classe derivada tem-se acesso aos métodos e variáveis `public` e `protected` da classe base
  - Não se tem acesso aos membros marcados como `private`
  - Continua-se a ter acesso aos membros sem etiqueta (*package private*)

# Herança

- Se existir o construtor por omissão na classe base, ele será automaticamente chamado aquando da criação de um objeto da classe derivada, mesmo que não exista uma chamada explícita
- Caso a classe base não possua o construtor por omissão então a classe derivada terá que, obrigatoriamente, definir um construtor que permita chamar um construtor da classe base
  - O redireccionamento de parâmetros para o construtor da classe base é realizado colocando como primeira linha do construtor da classe derivada a chamada a uma função de nome `super` e com os parâmetros respetivos
    - De forma similar à delegação entre construtores de uma mesma classe, mas trocando o `this` pelo `super`

# Herança

- Os comportamentos (métodos) definidos nas classes base podem ser redefinidos nas classes derivadas
  - O nível de acesso não pode ser mais restrito do que o definido na classe base. Exs.:
    - Se a classe base define um método como `public`, na classe derivada não pode ser redefinido como `protected`
    - Se a classe base define um método como `protected`, na classe derivada pode ser redefinido como `protected` ou `public`
  - Os métodos redefinidos devem ser marcados com a anotação `@Override`

```
@Override  
void procData() { ... }
```

# Herança – Exemplo

```
class Student {  
    protected long nr;  
    protected String name;  
  
    public Student(long nr, String name) {  
        this.nr = nr;  
        this.name = name;  
    }  
  
    public long getNr() {  
        return nr;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getEmail() {  
        return String.format("a%d@isec.pt",nr);  
    }  
    // ...  
}
```

# Herança – Exemplo (cont.)

```
class ErasmusStudent extends Student {  
    protected String country;  
  
    public ErasmusStudent(long nr, String name, String country) {  
        super(nr, name);  
        this.country = country;  
    }  
  
    public String getCountry() {  
        return country;  
    }  
  
    @Override  
    public String getEmail() {  
        return String.format("a%d.%s@isec.pt", nr, country);  
    }  
  
    // ...  
}
```

# Herança – Exemplo (cont.)

```
public class Main {  
    public static void main(String[] args) {  
        ErasmusStudent e1 = new ErasmusStudent(20220202021,"John Smith","uk");  
        ErasmusStudent e2 = new ErasmusStudent(20220303031,"Pierre Durand","fr");  
        ErasmusStudent [] tab = {e1, e2};  
        for(ErasmusStudent s: tab)  
            System.out.printf("%s: %s\n",s.getName(),s.getEmail());  
    }  
}
```

# final

- A palavra-chave **final** permite indicar que o elemento onde se aplica não é passível de alteração
- Quando aplicado a...
  - variável
    - Ex: `final int i = 123;`
    - Quando uma variável **final** assume um valor, esse valor não poderá ser alterado
      - Se esse valor for uma referência para um objeto, as características internas desse objeto podem ser alteradas, desde que não sejam **final**
    - Uma variável **final** terá que obter valor por atribuição direta ou, no caso das variáveis membro, obter valor no contexto de todos os construtores que existam
  - método
    - Ex: `public final String getEmail() { ... }`
    - Quando o método é marcado como **final** então não poderá ser redefinido nas classes derivadas
  - classe
    - Ex: `final class ErasmusStudent extends Student { ... }`
    - Uma classe marcada como **final** não pode ser a classe base para uma outra classe
      - Por outras palavras, "corta o processo de herança"

# *sealed* (Java 17)

- Uma *sealed class* apenas permite como subclasses as que forem indicadas com a sua definição

```
sealed class <base_class> permits <other1>, <other2>, ... {  
    // ...  
}
```

- Uma classe derivada de uma classe *sealed* tem que ser **final**, *sealed* ou **non-sealed**
- Exemplo:

```
sealed class A permits B,D,E { /* ... */ }  
sealed class B extends A permits C { /* ... */ }  
final class C extends B { /* ... */ }  
non-sealed class D extends A { /* ... */ }  
final class E extends A { /* ... */ }  
class F extends D { /* ... */ }
```

# Classes e métodos abstratos

- Uma classe poderá não fornecer implementação para todos os seus métodos ou, simplesmente, não permitir que sejam criadas instâncias diretas
  - Por exemplo, classes que representam conceitos abstratos
- Os métodos que não possui implementação devem ser etiquetados com a palavra-chave `abstract`
- As classes que possuem métodos abstratos têm obrigatoriamente que ser também classificadas como abstratas, devendo incluir a etiqueta `abstract`
- Uma classe pode ser declarada como abstrata mesmo que não tenha qualquer método abstrato

# Classes e métodos abstratos

- As classes que derivam de classes abstratas devem fornecer implementações para os métodos abstratos ou serem também classificadas como abstratas
- Não se podem criar instâncias de uma classe abstrata, mas as variáveis do tipo da classe abstrata podem referenciar instâncias de classes que sejam suas derivadas

# Classes e métodos abstratos

```
abstract class Fruit {  
    // ...  
    abstract int getColor();  
    // ...  
}  
  
class Orange extends Fruit {  
    // ...  
    @Override  
    int getColor() {  
        return 0xFFA500;  
    }  
    // ...  
}
```

# Interfaces

- Uma classe abstrata pode, no limite, fornecer um conjunto de métodos todos eles abstratos, sem qualquer funcionalidade associada
- A linguagem *Java* disponibiliza uma forma mais adequada para este tipo de declaração a que se chama *interface*

# Interface

- Uma *interface* em Java é uma descrição de um protocolo que outras *interfaces* ou classes têm que respeitar
- Na sua utilização mais tradicional, as *interfaces* permitem "definir" um conjunto de métodos que uma ou mais classes terão que respeitar
- Exemplo de definição de uma *interface*

```
interface MyInterface {  
    void my_func();  
}
```

# Interface

- Implementação de uma *interface*

```
class MyClass implements MyInterface {  
    @Override  
    public void my_func() {  
        // ...  
    }  
}
```

- Todos métodos implementados são **public**
  - Não é possível restringir o acesso para passarem a ser **protected** ou **private**
- Uma classe pode implementar diversas *interfaces*
  - Recordar: uma classe apenas pode derivar de uma classe
- Notas:
  - Se for utilizada uma classe abstrata em vez de uma interface, podem-se definir métodos **protected** para serem redefinidos nas classes derivadas
  - Apenas se pode estender uma classe, mas podem-se implementar várias interfaces

# Interface

- Uma classe que implementa uma interface, mas que não fornece implementações para os métodos abstratos, deve ser etiquetada como abstrata
- Uma variável do tipo da *interface* pode referenciar instâncias de classes que implementam essa *interface*
  - Com uma variável do tipo da *interface* apenas se tem acesso às funcionalidades declaradas nessa *interface*
- Nota: uma instância de MyClass é também instância (*instanceof*) de MyInterface

# Interface

- Para além da declaração de métodos que as classes deverão implementar, uma *interface* pode também conter:
  - constantes
    - Variáveis que são definidas no interface
    - Estas variáveis recebem automaticamente os "modificadores" `public`, `static` e `final`, ou seja, as variáveis são constantes
  - métodos *default*
    - Métodos etiquetados com a palavra `default`
    - Fornecem implementações por omissão, mas que podem ser redefinidas nas classes que implementam a *interface*
  - métodos *static*
    - Métodos etiquetados com a palavra `static`
    - Fornecem implementações por omissão que não podem ser redefinidas nas classes que implementam a *interface*

# *Superinterface e subinterface*

- Uma *interface (subinterface)* pode derivar de outra *interface (superinterface)*, estendendo o protocolo
  - Deve-se usar a palavra chave `extends`, como nas classes.

```
interface A { ... }
interface B { ... }

interface C extends A, B {
    ...
}
```

```
interface A { int n = 1; }
interface B { int p = 2; }
interface C extends A, B {
    int n = 5;
    int x = A.n + p;
}
```

```
interface A { int n = 1; }
interface B { int n = 2; }
interface C extends A, B {
    // int x = n + 4; // ERRO
    int x = A.n + 4;
}
```

# Hierarquia única – a classe Object

- Em Java todas as classes derivam direta ou indiretamente da classe Object
  - Mesmo quando não é explicitada essa dependência, ela é assumida de forma implícita
- class MyClass { ... } <=> class MyClass extends Object { ... }
- A classe Object disponibiliza um conjunto de funcionalidades (métodos) onde se incluem
  - getClass
  - clone
  - equals
  - hashCode
  - toString

# Object.getClass

- O método `Class<?> getClass()` retorna um objeto `Class` representivo do tipo de objecto a que corresponde a instância
- Um objeto `Class` permite usufruir de funcionalidades de "introspecção"
  - Obter informações sobre o próprio código (construtores, métodos, variáveis, anotações, ...)
    - Por exemplo, tendo a referência para um objeto `Class`, correspondente a um determinado tipo de objecto, poder-se-ão criar novas instâncias desse tipo, enumerando os respetivos construtores executando o pretendido
- A comparação de instâncias dos objetos `Class` permite verificar se dois objetos são exatamente do mesmo tipo
  - `obj1.getClass() == obj2.getClass()`

# Object.clone

- O método `protected Object clone()` deverá retornar uma cópia exata da instância no contexto da qual é chamado
- Os tipos de objetos que pretendam usufruir desta funcionalidade deverão implementar a interface `Cloneable`
  - Como a classe `Object` não implementa a interface `Cloneable` origina a geração de uma exceção que originará a interrupção do programa (`CloneNotSupportedException`)
- A redefinição deste método numa classe derivada deve chamar o método `clone` na classe base: `super.clone()`
  - Copia a mais baixo nível o objeto
  - O construtor do tipo de objeto não é chamado
  - Poderão ser necessárias alterações adicionais depois da chamada do `clone` da classe base, uma vez que o objeto pode conter referências para objetos, os quais devem também ser *clonados*.

# Object.equals

- Um objeto que vá ser alvo de comparações com outros objetos similares deverá disponibilizar uma implementação do método `equals`, herdado a partir da classe `Object`
- O método `public boolean equals(Object other)` recebe como parâmetro a referência do outro objeto com o qual se pretende realizar a comparação, retornando
  - `true`, se for considerado igual
  - `false`, se for considerado diferente

```
@Override  
public boolean equals(Object o) {  
    // ...  
    return <resultado_comparação>;  
}
```

- Ter em atenção que o parâmetro do método `equals` é do tipo `Object`, ou seja, pode referenciar qualquer tipo de objeto Java

# Object.equals

- O método `equals` deverá verificar se a comparação está a ser realizada com o objeto adequado
  - Obrigar a que seja exatamente um objeto do mesmo tipo

```
this.getClass() == other.getClass()
```
  - Permitir que o objeto possa ser uma instância de um tipo de objeto de determina hierarquia (em princípio da mesma à qual o próprio objeto pertence)

```
other instanceof <classe_base_da_hierarquia>
```
- Para além da verificação do tipo deverá depois ser verificado o conteúdo que se pretende realmente comparar
  - Necessário fazer a conversão (*cast*) para o tipo pretendido para aceder aos membros adequados

# Object.hashCode

- Sempre que se redefine o método `equals` deve também ser redefinido o método `hashCode`
- O método `public int hashCode()` deverá retornar um inteiro que represente a instância em causa
  - Pode corresponder a um identificador único atribuído ao objeto ou pode ser calculado tendo em conta uma ou mais características do objeto
    - Normalmente são consideradas as características usadas para a comparação no `equals`
    - Por omissão (implementação da classe `Object`) retorna um valor inteiro correspondente a endereço interno

```
@Override  
public int hashCode() {  
    return <valor_único_para_a_instância>;  
}
```

# Object.hashCode

- Ter em consideração:
  - Dois objetos considerados iguais pelo método equals (true) devem possuir o mesmo hash code
  - Não é necessário que dois objetos considerados diferentes pelo equals (false) tenham hash codes diferentes
- O *hash code* é usado para otimizar a indexação da informação (similar a técnicas de *clustering*), por exemplo, no contexto das coleções de dados
  - A pesquisa de elementos similares é apenas realizada nos agrupamentos (*clusters*) a que correspondem itens com o mesmo *hash code*
  - Se para dois objetos iguais forem gerados *hash codes* diferentes (o que originará a sua colocação em *clusters/grupos* diferentes) os objetos pretendidos não serão encontrados

# Object.toString

- O método `public String toString()` deve retornar um objeto `String` representativo na instância do objeto que representa
- Por omissão (caso não seja redefinido por uma classe derivada) é retornado o nome da classe seguida de um valor hexadecimal representativo da instância (correspondente ao `hashCode` do objeto)
  - `MyObject@1FA4D5`

# Classes interiores

- Em Java é possível definir classes dentro de classes
  - Definidas da mesma forma que as classes, mas colocadas dentro de outras classes ou funções

```
class ClassExterior {  
    // ...  
    class ClassInterior {  
        // ...  
    }  
}
```

- Fora da classe onde se insere, a classe interior é referida pelo seu nome completo:

`ClassExterior.ClassInterior`

# Classes interiores

- As classes interiores podem possuir os modos de acesso adequados (`public`, `protected`, `private` ou *package-private*) e/ou serem definidas como `static`

```
public class ClassExterior {  
    public ClassExterior() {}  
  
    public class ClassInterior1 {...}  
  
    public static class ClassInterior2 {...}  
}
```

# Classes interiores – *Inner classes*

- Só podem ser acedidas a partir de uma instância da classe onde se inserem
  - As instâncias destas classes ficam associadas a uma instância da classe onde se inserem
- Têm acesso aos membros da classe onde se inserem através da referência para essa classe:  
`<class_exterior_name>.this`
- Externamente não se podem criar instâncias destas classes, mesmo que sejam públicas
  - Poderá existir um método de instância da classe exterior que retorne instâncias da classe interior

# Classes interiores – *Inner classes*

```
public class ClassExterior {  
    int value;  
  
    public ClassExterior(int value) { this.value = value; }  
  
    @Override  
    public String toString() { return "ClassExterior{" + value + '}'; }  
  
    public ClassInterior1 getClassInterior1(int value) { return new ClassInterior1(value); }  
  
    // Inner class  
    public class ClassInterior1 {  
        int value;  
  
        public ClassInterior1(int value) { this.value = value; }  
  
        @Override  
        public String toString() {  
            return String.format("ClassInterior1{%d,%d}",  
                value, ClassExterior.this.value);  
        }  
    }  
}
```

# Classes interiores – *Nested classes*

- Definidas com o auxílio da etiqueta adicional `static`  
`[public] static class <nested_class> { ... }`
- Podem ser criadas instâncias destas classes, de forma direta, sem que nessa necessário existir uma instância da classe exterior que lhe dê suporte.

# Classes interiores – *Nested classes*

```
public class ClassExterior {  
    int value;  
  
    public ClassExterior(int value) { this.value = value; }  
  
    @Override  
    public String toString() { return "ClassExterior{" + value + '}'; }  
  
    // Nested class  
    public static class ClassInterior2 {  
        int value;  
  
        public ClassInterior2(int value) {  
            this.value = value;  
        }  
  
        @Override  
        public String toString() {  
            return "ClassInterior2{" + value + '}'; // no direct access to ClassExterior.value  
        }  
    }  
}
```

# Classes Interiores – criação de instâncias

- Exemplo de utilização das classes dos exemplos anteriores:

```
public class Main {  
    public static void main(String[] args) {  
        ClassExterior a1 = new ClassExterior( value: 100);  
        //ClassExterior.ClassInterior1 a2 = new ClassExterior.ClassInterior1(200); /* ERROR!!! */  
        ClassExterior.ClassInterior1 a2 = a1.getClassInterior1( value: 200);  
        ClassExterior.ClassInterior2 a3 = new ClassExterior.ClassInterior2( value: 300);  
        System.out.println(a1);  
        System.out.println(a2);  
        System.out.println(a3);  
    }  
}
```

# *enum*

- Tal como em C ou C++, o enum permite definir um conjunto de constantes que permitem melhorar a legibilidade do código
- Por convenção os enum, tal como as constantes, escrevem-se em maiúsculas

```
enum Planet {  
    MERCURY, VENUS, EARTH, MARS,  
    JUPITER, SATURN, URANUS, NEPTUNE  
}  
...  
Planet p = Planet.EARTH;  
...  
if (p != Planet.URANUS) ...  
...
```

# enum

- Como em Java tudo são objetos, o enum também permite usufruir de membros que dão mais informação sobre o enum em causa

```
enum Planet { MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE }

class TestPlanets {
    public static void main(String[] args) {
        System.out.println(Arrays.toString(Planet.values()));
        for(var planet : Planet.values()) {
            System.out.printf("%-10s [%d] ", planet, planet.ordinal());
            switch (planet) {
                case EARTH -> System.out.println("Beatiful");
                case JUPITER -> System.out.println("Big");
                case MERCURY, VENUS -> System.out.println("Hot");
                default -> System.out.println();
            }
        }
    }
}
```

*Output:*

```
[MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE]
MERCURY      [0] Hot
VENUS        [1] Hot
EARTH        [2] Beatiful
MARS         [3]
JUPITER      [4] Big
SATURN       [5]
URANUS       [6]
NEPTUNE      [7]
```

# *enum "personalizados"*

- As enumerações podem ter
  - membros variáveis
  - métodos que podem ser ou não ser abstratos

```
enum Planet {  
    MERCURY(2440), VENUS(6052), EARTH(6371), MARS(3390),  
    JUPITER(69911), SATURN(58232), URANUS(25362), NEPTUNE(24622);  
  
    private int radius; // Km  
  
    Planet(int radius) { this.radius = radius; }  
  
    public int getRadius() { return radius; }  
  
    @Override  
    public String toString() {  
        String str=super.toString().toLowerCase();  
        str = Character.toUpperCase(str.charAt(0))+str.substring(1);  
        return str;  
    }  
}
```

# *enum "personalizados"*

- Para o enum Planet anterior...

```
class TestPlanets {  
    public static void main(String[] args) {  
        System.out.println(Arrays.toString(Planet.values()));  
        for(var planet : Planet.values())  
            System.out.printf("%-10s [%d] (%5d km)\n",  
                planet, planet.ordinal(), planet.getRadius());  
    }  
}
```

*Output:*

```
[Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune]  
Mercury      [0] ( 2440 km)  
Venus       [1] ( 6052 km)  
Earth       [2] ( 6371 km)  
Mars        [3] ( 3390 km)  
Jupiter     [4] (69911 km)  
Saturn      [5] (58232 km)  
Uranus      [6] (25362 km)  
Neptune     [7] (24622 km)
```

# enum – métodos abstratos

```
enum Operation {  
    add { double calculate(double x, double y) { return x + y; } },  
    sub { double calculate(double x, double y) { return x - y; } },  
    mul { double calculate(double x, double y) { return x * y; } },  
    div { double calculate(double x, double y) { return x / y; } };  
  
    abstract double calculate(double x, double y);  
}  
  
class TestOperation {  
    static void f(double x, double y) {  
        for (Operation op : Operation.values()) {  
            System.out.println(x+ " "+op+" "+y+ " = "+op.calculate(x, y));  
        }  
    }  
    public static void main(String[] args) {  
        f(12,4);  
    }  
}
```

*Output:*

```
12.0 add 4.0 = 16.0  
12.0 sub 4.0 = 8.0  
12.0 mul 4.0 = 48.0  
12.0 div 4.0 = 3.0
```

# enum – redefinição de métodos

```
enum Operation {  
    add {  
        double calculate(double x, double y) { return x + y; }  
  
        @Override  
        public String toString() { return "plus"; }  
    },  
    sub { double calculate(double x, double y) { return x - y; } },  
    mul { double calculate(double x, double y) { return x * y; } },  
    div { double calculate(double x, double y) { return x / y; } };  
  
    abstract double calculate(double x, double y);  
}  
  
class TestOperation {  
    static void f(double x, double y) {  
        for (Operation op : Operation.values()) {  
            System.out.println(x+" "+op+" "+y+" = "+op.calculate(x, y));  
        }  
    }  
    public static void main(String[] args) {  
        f(8,5);  
    }  
}
```

*Output:*  
8.0 plus 5.0 = 13.0  
8.0 sub 5.0 = 3.0  
8.0 mul 5.0 = 40.0  
8.0 div 5.0 = 1.6