Advanced Programming

Introduction to Java language

Java language

- The 1st version was made available in 1995 by *Sun Microsystems*
  - Its study started in 1991

- Fully object-oriented language
- *Cross-platform*
- The syntax follows the C/C++ style

- Objectives (https://www.oracle.com/java/technologies/introduction-to-java.html)
  - Simple, object-oriented and familiar
  - Robust and safe
  - Architecture-independent and portable
  - High performance
  - Interpreted, multitasking and dynamic

Compiling a Java program

- Compilation, which includes syntax checking and generation of compiled code, is performed using the ***Java Development Kit*** **(JDK)**
  - Some development environments have built-in versions
  - Includes tools and class libraries
  - Includes a compilation tool: `javac`
  - Intermediate code, *called Java bytecode*, is generated
    - Architecture independent code

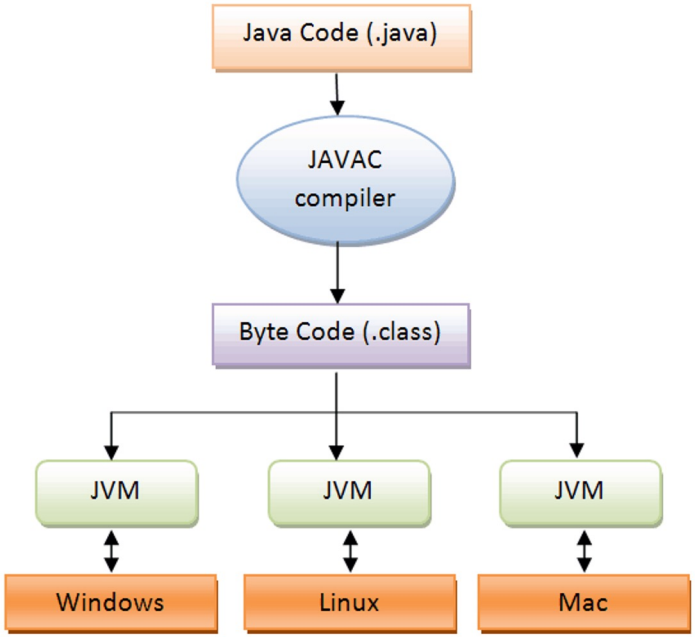
JDK installation

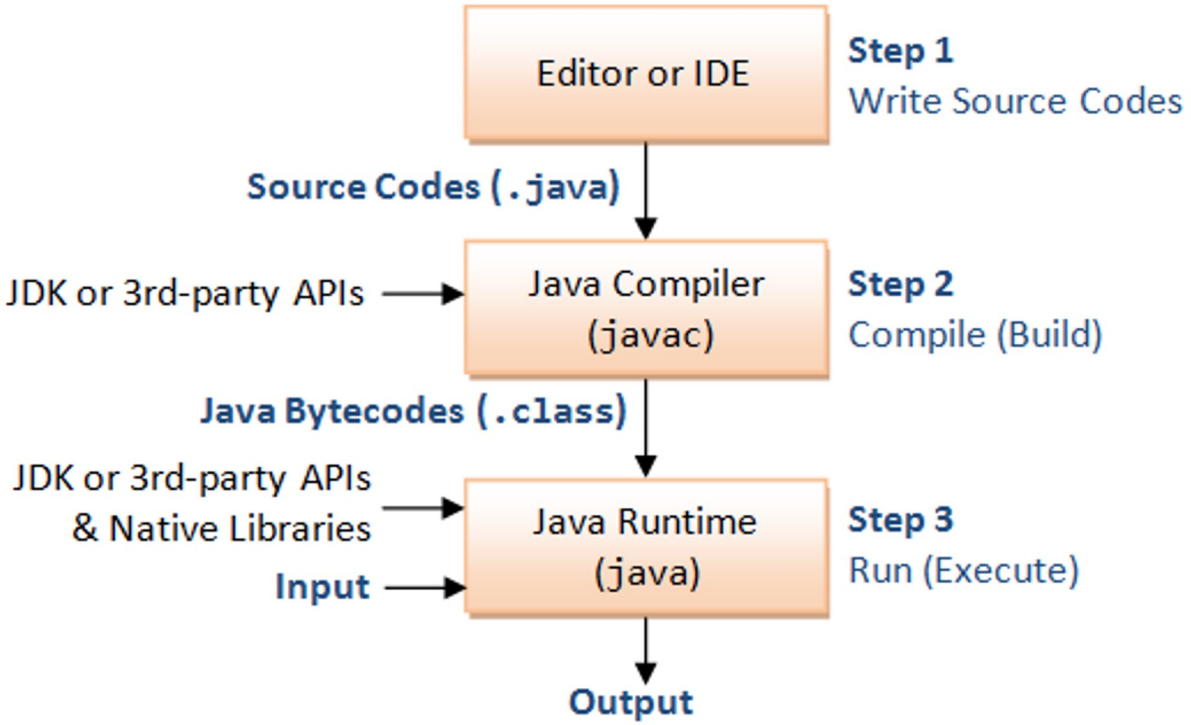
- *OpenJDK* vs. *OracleJDK*
  - *OracleJDK*
    - Commercial version, requiring licensing
    - Support provided by *Oracle*
    - Supposedly more stable and with a better performance
  - *OpenJDK*
    - *Open source* (we can contribute to its development)
    - Although its support is essentially done by communities active in its development, there are specific distributions with their own support.

Java Virtual Machine

- As the compilation result is not native code, it is necessary to have a *Java bytecode* interpreter
- The *Java bytecode* interpreter is called ***Java Virtual Machine*** **(JVM)**
  - In order for Java programs to run on a certain platform, a JVM must exist for that platform
  - The JVM also provides a secure, isolated environment in which programs run without affecting or being affected by other programs (*sandbox*)
  - Note : it is not necessary to install the JDK on machines where you just want to run the programs, just install a simplified version that only supports execution, called *Java Runtime Environment* (JRE)
    - Includes "**`java`**" command to run programs previously compiled to *Java bytecode* (`.class` files)

Creation and execution





Example

```
// Example of a Java program
// Filename: Example1.java

public class Example1 {
   public static void main(String args[]) {
      System.out.println("Java@DEIS-ISEC");
   }
}
```

- All code is encapsulated in classes or similar
- A public class (in the example: `class Example1`) is defined in a file with the same name as that class and the extension `.java`
  - There can be several classes in the same file, but only one can be public
- The first function to be executed in a *Java program* is
  `public static void main ( String args [])`

Example (compile and run)

- After writing the code, it must be saved with the name of the public class present in the code file and with a `.java` extension
  - In this case `Example1.java`

- Compilation is performed by doing

  `javac Example1.java`
  - In case of success, the file `Example1.class` is created
  - If there are errors, these are indicated through the line number in the file in question.

- The execution is carried out by doing:

  `java Example1`

```
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$
```

Note:
Exemplo (pt) ⇔ *Example* (en)

Java Language - Basic Concepts

Variables and primitive types

Arrays

Operators

Flow control

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

Primitive data types

- Although the Java language is fully object-oriented, the following types of "primitive data" are supported:
  - `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* or *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

Variables

- Allow the storage of typed information or references to objects
- The declaration of a simple variable is performed as follows:

<type> <variable_name> [= <default_value>]

  - The variable name can be a set of uppercase or lowercase characters, numbers, or the characters '_' or '$' (this one should be avoided!)
    - Must not start with a number

Example of typed variables

```
// 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';
```

- The '_' can be used to improve the readability of numbers:

```
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;
```

... but the following cases are not allowed:

```
// 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 (literal decimal)
int x1 = 5_2;
// Invalid : cannot put underscores
// at the end of a literal
int x2 = 52_;
// OK (literal decimal)
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 (literal hexadecimal)
int x6 = 0x5_2;
// Invalid : cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

Conversions between types (cast)

- The conversions between numeric types are performed…
  - Implicitly
    - conversions to types that do not imply any loss of precision. Ex.:
      - `int` => `long`
      - `int` => `float`
  - Explicitly
    - conversions to types where some information loss may occur. Ex.:
      - `long` => `int`
      - `float` => `int` ("*truncation*")

Wrapper classes

- Since the Java language is totally object-oriented, to maintain the compatibility when using primitive data there is an equivalent defined through an object class type.
  - These classes provide some functionality that may be useful, for example, to perform conversions
    - Ex: `int i = Integer.parseInt("1234");`

- Equivalent classes:
  - `byte` ⇔ `Byte`
  - `short` ⇔ `Short`, `int` ⇔ `Integer`, `long` ⇔ `Long`
  - `float` ⇔ `Float`, `double` ⇔ `Double`
  - `boolean` ⇔ `Boolean`
  - `char` ⇔ `Character`

Boxing and *Unboxing*

- *Boxing* – conversion of primitive data to the corresponding object
  - `Integer i = 123;`
  - `Integer i = new ~~Integer~~(123);`
    - `Integer i = Integer.valueOf(123);`

- *Unboxing* – converting instances of *wrapper* classes to the corresponding primitive types
  - Unboxing example (assuming… `Double d1 = 123.45;`):
    - `double d2 = d1;`
    - `double d3 = d1.doubleValue();`

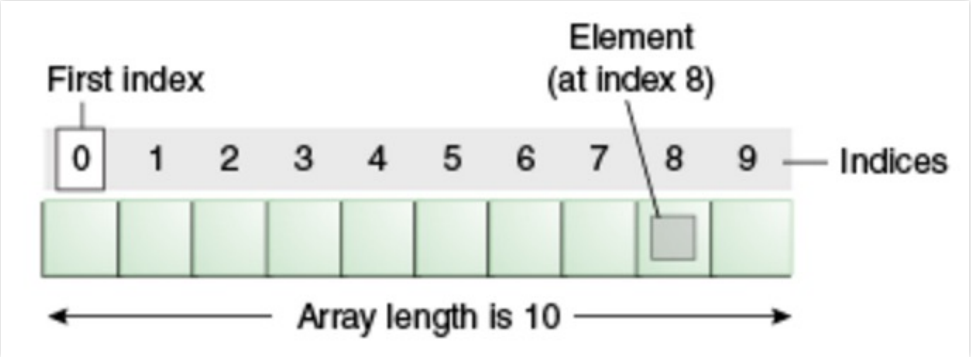
Handling "Big" Numbers

- BigInteger
- BigDecimal

```
BigDecimal bd1 = BigDecimal.valueOf(2e-323);
BigDecimal bd2 = BigDecimal.valueOf(3e-323);
BigDecimal bd3 = bd1.multiply(bd2);
System.out.println(db3); // 6.00E-646
```

Arrays

- Allow the storage of multiple values
  - Support for primitive data or references to objects
  - The maximum number of elements is defined when the *array* is created
  - The *array* has a name, corresponding to the name of the variable that supports it
  - Each value is accessed through the *array variable* and its index (0..*n-1*)



Array declaration

- The declaration can be performed using the following formats
  - *<type> [ ] <name>*
  - *<type> <name> [ ]*

- Example:

```
int [ ] ages;
float months [ ];
```

Array creation

- When *arrays* are just declared, the related variable has the value `null`, corresponding to the non-definition of the *array*
- To create (define) the *array* it is necessary to create the number of cells/values intended for it, using the keyword `new`

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

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

Examples of *array* usage

```
ages[0] = 10;

ages[1] = 15;

ages[2] = 20;

int i1 = ages[0] + 1;

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

Initialization of Arrays

- When an *array* is created it is possible to initialize it by enumerating the default values
  - *array* dimension is deducted from the number of elements enumerated

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

- The number of elements in an *array* can be easily obtained through the `length` property

```
int n_items = tab1.length;
```

Multidimensional *arrays*

- In Java, *arrays* of multiple dimensions are allowed, which are declared by inserting pairs of [ ] for each dimension.

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

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

- Items access is carried out as follows

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

Multidimensional *arrays*

- *Arrays* can have a variable number of elements per line/dimension
  - In this situation, the creation of each line/dimension must be performed in an independent way

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

b[0] = new int[3];

b[1] = new int[4];

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

Iteration over *arrays*

- Getting the number of elements in an *array*
  - Assuming `int [][] b = ...`
    - `b.length` – number of rows in *array* b
    - `b[i].length` – number of elements of line *i*

- Example:

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

Array variables

- Variables of *array type* are references to the corresponding objects

- Taking this into consideration:

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

Useful methods

- There are several classes and libraries in Java that provide useful methods for different situations.

- For operations on *arrays, the* following can be highlighted:
  - `System.arraycopy(src,src_pos,dst,dst_pos,length)`
    - Allows to copy elements from one *array* to another, previously created
  - The `java.utils.Arrays class`
    - Provides a set of methods for working with *arrays*
      - `copyOf, copyOfRange`
      - `fill`
      - `compare, equals, mismatch`
      - `binarySearch, sort, stream`
      - `toString`

Operators

- The operators are similar to those existing in other languages (C/C++, C#,...)

Operator Precedence

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

assignment operator =

- Different behaviors depending on the type of data
  - primitive data
    - value is copied
  - objects
    - Variables only store the reference to the object
    - a copy of the reference is made
      - The object referenced by both variables is the same
- Passing parameters when calling a function follows the same logic as the assignment operator

Garbage Collector

- When operations like assignment, parameter passing or similar are executed with references to objects, the number of references to the related instances is automatically managed centrally.
- When the number of references is zero it means that there is no way to access the object and, as such, the object is marked for deletion
  - It is not the programmer's responsibility to free the associated memory

Garbage Collector

- The management of the memory that will be reserved and its subsequent release is done by the *Garbage Collector*
  - When objects are marked for deletion the `finalize()` method is invoked
    - There is no guarantee when this method is called
    - Marked as *deprecated* since version 9
  - After the `finalize` method is called, the memory associated with the object will be freed, but there is no guarantee of when.

== and != operators

- When the operators == and != are used with reference-type operands, what is checked is whether the two operands refer to the same object.

```
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

- `instanceof` operator allows checking if a given reference corresponds to an object of a given type

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

Flow control – if

- *if-then*

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

- *if-then-else*

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

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

Flow control – switch, break

- *switch-case* + *break*

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

- *switch-arrow case*

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

- Similar to classic switch-case, but the instruction `break` is not needed
- This format also allows its use as an expression

- *Strings* can be used in *cases*

Flow control - cycles

- *for*

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

- *while*

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

- *for each*

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

- *do-while*

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

flow control

- Other flow control statements in the execution of a Java program:
  - `continue`
  - `break`
    - The *continue* and the *break* instructions can be followed by a *label* associated with the loop in which they are related to

```
loop1: for(int i=0; i<10 ; i++)
           for(int j=10; j>0 ; j--)
               if (i == j) break loop1;
```

  - `return`

User interaction

- Java provides a set of objects that allow representing the typical inputs and outputs of a program/process.
  - `System.in` – represents the default input, typically associated with console input
  - `System.out` – represents the default output, typically associated with output to the console
  - `System.err` – represents output for errors, typically associated with output to the console

data output

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

Data input

- Although the `System.in` object allows obtaining data entered by the user in the console, this data is interpreted as *bytes*

- To facilitate access to this information, in an easier and more typified way, a `Scanner` object can be used.

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

- `Scanner` object allows to read character sequences separated by delimiters.

```
int n = sc.nextInt();          // Read an integer
double x = sc.nextDouble();  // Read a double
```

- By default, the delimiters are whitespace, tabs and newlines. However, other delimiters can be defined:

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

- The type of the next value to be read can be tested using:

```
if ( sc.hasNextDouble() ) // Check if the next value is a double
   x = sc.nextDouble () ;
```

Math

- `Math` class, belonging to the `java.lang` package, provides a set of mathematical functions
  - `sin, cos, tan`
  - `abs, round, floor, ceil, rint`
  - `max, min`
  - `pow, exp`
  - `hypot, sqrt`
  - `random`
  - `toDegrees, toRadians`
  - `E, IP`

Math.random

- Method that generates *pseudo-random numbers*
  - `double r = Math.random();`
    - $0.0 \leq r < 1.0$
  - Example for generating numbers between 1 and 100, inclusive
    - `int i = (int) (Math.random() * 100) + 1;`

Class Random

- In the package `java.util`, the `Random` class is provided, which has a set of methods that allows a more flexible access to sequences of *pseudo-random numbers*

```
Random rnd = new Random();
int i = rnd.nextInt();
int j = rnd.nextInt(100) + 1;
double d = rnd.nextDouble();
...
rnd.setSeed(1234);
...
```

String

- The `String` class represents an immutable *charsequence*
  - Changing a `String` (for example, concatenating using the '+' operator) always creates a new `String`

- This class provides utility methods to work with the characters present in a `String`
  - `equals, equalsIgnoreCase, matches, compareTo, compareToIgnoreCase, startsWith, contains, endsWith, indexOf`
  - `isBlank, isEmpty`
  - `concat, replace, replaceAll, repeat, trim`
  - `split`
  - `toUpperCase, toLowerCase`
  - `...`

String

- There are some cautions about the *String class* to point out, which can be verified by the following example

```
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 and StringBuilder

- `StringBuffer` and `StringBuilder` classes allow you to manage mutable *strings*
- Both classes provide several methods for working with *strings*, including methods for adding new characters or *strings*, modifying individual characters or *substrings* , ...
- Main differences
  - `StringBuffer` – *thread-safe*
  - `StringBuilder` – faster

Java language

Object oriented programming

Object-oriented programming

- As the Java language is fully object-oriented, it respects all the principles associated with this programming paradigm, namely:
  - Abstraction and Encapsulation
    - Define new object types
      ```
      class <new_class_name> { ... }
      ```
    - Hide implementation details
    - Control access to information
    - Generalize the way it is used
  - Inheritance
    - Defining a new object type as a specialization of another object type
      ```
      class < new_class > extends < base_class > { ... }
      ```
    - The characteristics of the base object are inherited
    - New characteristics and behaviors of the specialization can be defined
  - Polymorphism
    - Redefining (`@Override`) the behaviors declared and/or defined in the base class

Classes

- To define a new class of objects, the following syntax is used:

```
class <name> {
    <variables>
    <methods/functions>
}
```

- It is not mandatory that all variables are defined at the beginning or the methods at the end, but it is a good policy for their organization

Classes

- Example:

```
class Point {
    int x,y;

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

- To create an object of a class, the new statement is used

```
Point p = new Point();
```

Method overload

- There can be several methods with the same name defined, as long as they can be distinguished by their parameters

- The distinction can be made in
  - number of parameters
  - Parameter type

- The type defined for the method return is not used to distinguish two *overloaded* methods.

Constructors

- Whenever an object is created, a constructor is automatically called to initialize the object.
  - Constructors are methods with the same name as the class and no return type declared.
  - Multiple constructors can be defined, with different parameters
    - A parameterless constructor is called "default constructor"
  - In Java, variables not explicitly started (within the constructor or by direct assignment when declaring them) are started with default values (`0` or `null`)

```
class Point {
    int x,y;

    Point( int xi, int yi ) {
        x = xi;
        y = yi;
    }

}
```

this

- In the context of any instance of a certain type of object, its reference can be accessed (self-reference) through the keyword `this`

```
class Point {
   int x, y;

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

Constructors delegation

- A constructor can delegate the construction of an object to another constructor of the same object (when there are *overloaded constructors*)
  - Delegation is done by using `this` as if it were a function and whose parameters are those of the constructor to which the construction of the object will be delegated

```
class Point {
    int x, y;

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

Static members

- In the definition of a class, the various members, variables or functions, can be defined as being static
  - The `static` tag is used
- The static members...
  - Can be accessed/used without instances of this type of object
    - Its use is made by indicating the name of the class itself and not a reference to an object of that type.
  - The values of the static variables are accessed and subject to change, being shared by any existing instances of this type of object
    - Changes made to static variables in the context of one instance are visible to all other instances

Static members

- Static methods can only call other static methods
  - They can call instance methods as long as they are properly framed in the respective instances
- In the context of static members, the `this` reference is not used, since static members are not associated with any instance
- Static variables are only initialized on the first usage of the class
  - Values can be initiated by direct assignments or via a static constructor
    - Defined through: `static { ... }`

static members

```
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();
   }
}
```

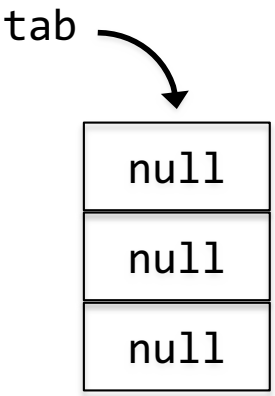
Output:

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

Object *arrays*

- An *array* of objects is created, without explicitly specifying the elements, it contains only null references (`null`)

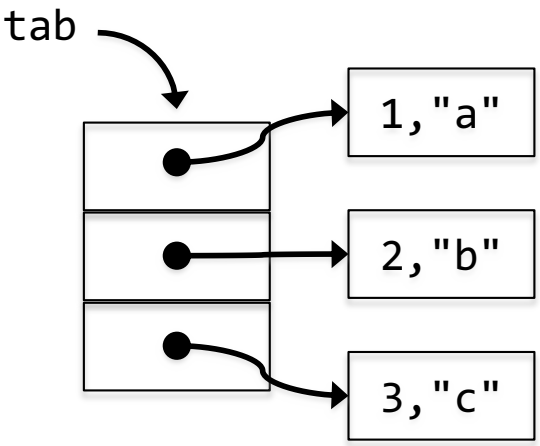
```
Person [] tab = new Person[3];
```



Object *arrays*

- An *array* of objects is created, without explicitly specifying the elements, it contains only null references (`null`)

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



Object *arrays*

- Object *arrays* can be created by enumeration.

```
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();
}
```

- When it is necessary to pass an *array* of objects created by enumeration to a function, the following format can be used:

```
proc( new Person [] { new Person("a"), new Person("b")} );
```

Packages

- Classes are grouped into *packages*
  - For code organization
    - Creation of *namespaces* to name the classes
  - Solve name conflicts between different classes
    - When there is conflict of names between classes of different *packages*, its full name must be indicated, ie, including the name of the package: `<package>.<class>`
  - Simplify application distribution
    - Easier way to make Java programs available for use by end users
  - Facilitate code reusability
    - Many features can be reused in other projects through the constitution and availability of *packages* with object classes
      - It follows the same logic of using all the classes that are available in *Java API*

package - creation

- *Package* definition
  - The *package* has a name defined through the statement "`package <name>;`" indicated, normally, as the first effective line in a code file
    - The base name is usually made up of two separate pieces of information, in lowercase
      - Company domain name in reverse order (`deis.isec.pt => pt.isec.deis`)
      - Project name
    - Example: `package pt.isec.a200212345.proj_lesson1;`
  - Files that include classes belonging to a *package* must be placed in a directory hierarchy corresponding to the *package name*

```
Project_Lesson1
  pt
    isec
      a200212345
        project_lesson1
           Example1.java
```

package - usage

- To use classes or other elements defined in other *packages*, the `import` statement must be used
  - There may be multiple import statements (usual!)
  - All elements of a package will be included:

```
import pt.isec.pa.utils.*;
```

  - Only the desired element will be included:

```
import pt.isec.pa.utils.FileUtils;
```

packages - example

```
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 n
ame: 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$
```

Note:
Exemplo (pt) ⇔ *Example* (en)

packages - example

```
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$
```

Note:
Exemplo (pt) ⇔ *Example* (en)

package - access

- In addition to the organization and reusable advantages that *packages* offer, they also allow you to control access to implementation particularities of their elements
- The access control of the different members is carried out through the keywords `public`, `protected`, `private` or by not specifying them (no label), taking into account the following table

Tag	class	*package*	subclass	others
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	**No**
(no label)	Yes	Yes	**No**	**No**
`private`	Yes	**No**	**No**	**No**

- Note: class definition statement only allows `public` tag or untagged (however, *nested* classes can have other modifiers/tags)

jar files

- To facilitate the tasks of *deploying Java* programs, the classes that make up this program, as well as other resources (images, sounds, ...), must be included in a `jar` file.
  - The `jar` file is actually a `zip` file
  - It allows a simpler maintenance of the hierarchy of directories that represent the several *packages* that can constitute a program
  - A manifest file is included which allows special attributes specification

- create `jar` file

```
jar cf example1.jar pt/*
```

- run `jar`

```
java -cp example1.jar pt.isec.pa.example1.Example1
```

Executable jar files

- When creating a `jar file` you can specify a class that includes a `main` function, which will be executed when the `jar file` is executed with: `java –jar <file.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$
```

Note:
Exemplo (pt) ⇔ *Example* (en)

Manifest file

- The manifest file, created automatically in the previous example, can be created and configured manually

```
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$
```

Note:
Exemplo (pt) ⇔ *Example* (en)

Composition vs Inheritance

- A class, by itself, allows to represent a certain concept
- In most situations, the concept will be defined with the help of the specification of characteristics or properties, represented with the help of variables
- Depending on the complexity and the specific situation, the definition of an object may use different models...
  - Composition
    - An object is defined by an aggregation of several other objects.
      - A computer consists of the *motherboard* , the processor, the graphics card, ...
  - Inheritance
    - An object is defined as a specialization of another more generic object.
    - All characteristics that define a base object type are inherited and only the differences for the base object are/must be programmed
      - A car with automatic gears is a car (represented by another class), but with a different way of changing gears (so, eventually, only this behavior must be reprogrammed)
  - or both
- These techniques allow a new object to take advantage of the characteristics and behaviors of other objects, without having to repeat their implementation.

Composition

```
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;
    }
    // ...
}
```

Inheritance

- To create a new class of objects from another one, inheriting all its characteristics, the following syntax is used:

```
class < new_class > extends < base_class > {
    ...
}
```

- With no additional information or behavior defined, the derived class instances are functionally identical to the base class
  - The derived class is also called a *subclass*
- In Java a class can only inherit from one other class (single inheritance)
  - Multiple inheritance is not allowed

Inheritance

- In the context of a subclass the features of the base class can be referred using the keyword `super`
  - `super.<base_class_variable>`
  - `super.<base_class_method>([<params, ...>]);`

- In a derived class you have access to the `public` and `protected` methods and variables of the base class
  - There is no access to members marked as `private` in the base class
  - Untagged members (*package-private* members) are still accessible

Inheritance

- If the *default constructor* exists in the base class, it will be called automatically when creating an object of the derived class, even if there is no explicit call

- If the base class does not have the *default constructor*, then the subclass will have to define a constructor that allows calling a constructor of the base class.
  - The redirection of parameters to the constructor of the base class is done by placing the call to a function named `super` and with the respective parameters in the first line of the subclass constructor
    - Similar to delegating between constructors of the same class, but replacing `this` with `super`

Inheritance

- Behaviors (methods) defined in base classes can be redefined in derived classes
  - The access level cannot be more restricted than defined in the base class. Eg.:
    - If the base class defines a method as `public`, in the subclass it cannot be redefined as `protected`
    - If the base class defines a method as `protected`, in the subclass it can be redefined as `protected` or public
  - The redefined methods must have the annotation `@Override`

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

Inheritance – Example

```
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 );
    }
    // ...
}
```

Inheritance – Example (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 );
    }

    // ...
}
```

Inheritance – Example (cont .)

```
public class Main {
  public static void main( String [] args ) {
    ErasmusStudent e1 = new ErasmusStudent(20220202l,"John Smith","uk");
    ErasmusStudent e2 = new ErasmusStudent(20220303l,"Pierre Durand","fr");
    ErasmusStudent [] tab = {e1, e2};
    for( ErasmusStudent s: tab )
      System.out.printf ("%s: %s\n", s.getName(), s.getEmail());
  }
}
```

Final

- The `final` keyword indicates that the element where it is applied is not subject to change
- When applied to a...
  - variable
    - Eg.: `final int i = 123;`
    - When a `final` variable is assigned with a value, that value cannot be changed.
      - If this value is a reference to an object, the internal characteristics of that object can be changed as long as they are not `final`
    - A `final` variable will have to get a value, by direct assignment or get its value in the context of all the class constructors (in the case of member variables)
  - method
    - Eg.: `public final String getEmail() { ... }`
    - When the method is marked `final` then it cannot be overridden in derived classes
  - class
    - Eg.: `final class ErasmusStudent extends Student { ... }`
    - A class marked `final` cannot be the base class for another class.
      - In other words, "*cuts the inheritance process*"

sealed (Java 17)

- A *sealed class* only allows as subclasses those that are indicated with their definition

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

- A class derived from a class *sealed* must be `final`, `sealed` or `non-sealed`
- Example:

```
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 { /* ... */ }
```

Abstract classes and methods

- A class may not provide implementation for all of its methods or simply not allow direct instances to be created.
  - For example, classes that represent abstract concepts
- The methods that are only declared, ie, do not have a *scope*, must be tagged with the keyword `abstract`
- Classes that have abstract methods must also be classified as abstract, and must include the `abstract` tag
- A class can be declared abstract even if it doesn't have any abstract methods

Abstract classes and methods

- Classes that derive from abstract classes must provide implementations for the abstract methods or be classified as abstract as well.

- An abstract class cannot be instantiated, but variables of the abstract class type can reference instances of classes that are derived from it.

Abstract classes and methods

```
abstract class Fruit {
    // ...
    abstract int getColor();
    // ...
}

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

Interfaces

- At its limit, an abstract class can be constituted by a set of methods that are all abstract, without any associated functionality.

- The *Java language* provides a more adequate form for this type of declaration, which is called an *interface.*

Interface

- An *interface* in Java is a description of a protocol that other *interfaces* or classes have to respect .

- In their most traditional use, an *interface* allows the declaration of a set of methods that one or more classes will have to respect.

- Example of defining an *interface*

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

Interface

- Implementing an *interface*

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

- All implemented methods are `public`
  - Cannot restrict access to be `protected` or `private`
- A class can implement several *interfaces*
  - Remember: a class can only derive from one class

- Notes:
  - If an abstract class is used instead of an interface, `protected` methods can be defined to be overridden in derived classes.
  - You can only extend one class, but you can implement multiple interfaces.

Interface

- A class that implements an interface, but does not provide implementations for abstract methods, must be labeled abstract

- A variable of the *interface's type* can reference instances of classes that implement that *interface*
  - With a variable of the *interface type,* you only have access to the functionalities (methods) declared in that *interface*

- Note: an instance of `MyClass` is also an instance (`instanceof`) of `MyInterface`

Interface

- In addition to declaring methods that classes must implement, an *interface* can also contain:
  - *constants*
    - Variables that are defined in the interface
    - These variables automatically receive the "modifiers" `public`, `static` and `final`, that is, the variables are constants
  - *default methods*
    - Methods tagged with the word `default`
    - Provide implementations by default, but which can be overridden in the classes that implement the *interface*
  - *static methods*
    - Methods tagged with the word `static`
    - Provide default implementations that cannot be overridden in classes that implement the *interface*

Superinterface and *Subinterface*

- An *interface* (*subinterface*) can derive from another *interface* (*superinterface*), extending the protocol
  - For this, the `extends` keyword must be used, as in inheritance among 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; // ERROR
   int x = A.n + 4;
}
```

Single hierarchy – the `Object` class

- In Java all classes derive directly or indirectly from the `Object` class.
  - Even when this dependence is not made explicit, it is assumed implicitly.

```
class MyClass { ... } <=> class MyClass extends Object { ... }
```

- The `Object` class provides a set of methods, which include:
  - `getClass`
  - `clone`
  - `equals`
  - `hashCode`
  - `toString`

Object.getClass

- The `Class<?> getClass()` method returns a `Class` object representative of the type of object to which the instance corresponds
- A `Class` object allows a program to take advantage of "introspection" functionalities
  - Get information about the code itself (constructors, methods, variables, annotations, ...)
    - For example, having the reference to a `Class` object, corresponding to a certain type of object, it is possible to create new instances of that type, enumerating the respective constructors and executing the desired one
- Comparing instances of `Class` objects permits checking whether two objects are of exactly the same type
  - `obj1.getClass() == obj2.getClass()`

object.clone

- The protected method `Object clone()` should return an exact copy of the instance in the context it is called from

- The types of objects that want to use this functionality must implement the `Cloneable` interface
  - As the `Object` class does not implement the `Cloneable` interface, it generates an exception that will cause the program to be interrupted (`CloneNotSupportedException`)

- The redefinition of this method in a derived class should call the `clone` method of the base class: `super.clone()`
  - A low level copy is made
  - Object type constructor is not called
  - Additional changes may be required after calling the base class `clone`, as the object may contain references to objects, which must also be *cloned* .

object.equals

- An object that is going to be compared with other similar objects must provide an implementation of the `equals` method , inherited from the `Object` class

- The `public boolean equals(Object other)` method receives the reference to the other object as a parameter with which the comparison is intended, returning
  - `true`, if considered equal
  - `false`, if considered different

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

- Keep in mind that the `equals` method parameter is of type `Object`, that is, it can reference any type of Java object

object.equals

- The `equals` method should check if the comparison is being performed with the appropriate object, alternatively
  - Force it to be exactly one object of the same type

```
this.getClass () == other.getClass ()
```

  - Allow the object to be an instance of an object type of a certain (sub)hierarchy, usually, the same to which the object itself belongs

```
other instanceof <base_class>
```

- In addition to verifying the type, the content to be compared must be verified.
  - Need to cast to desired type to access proper members

Object.hashCode

- Whenever the `equals` method is redefined, the `hashCode` method must also be redefined
- The `public int hashCode()` method should return an integer representing the instance in question
  - It can correspond to a unique identifier assigned to the object or it can be calculated taking into account one or more characteristics of the object
    - Usually, the characteristics used for the comparison in `equals` are considered.
    - By default (implementation of the `Object class`) returns an integer value corresponding to its internal address

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

Object.hashCode

- Important:
  - Two objects considered equal by the `equals` method must have the same *hash code*
  - It is not necessary for two objects considered different by `equals` to have different hash codes

- the *hash code* is used to optimize information indexing (similar to *clustering techniques*), for example in the context of data collections
  - The search for similar elements is only performed in groups of items (*clusters*) that have the same *hash code*
  - If two identical objects have *different hash codes* (causing them to be placed in different *clusters*/groups) the desired objects will not be found

Object.toString

- The `public String toString()` method should return a `String` object that represents the object instance to which it applies

- By default (if not overridden by a derived class) the fullname of the class is returned followed by a hexadecimal value representing the instance (corresponding to the object's `hashCode`)
  - `MyObject@1FA4D5`