POLYTECH MONTPELLIER

SESSION 6

PROJECT DEVELOPMENT

packaging

C.L.I.

archive

doc°

POLYTECH MONTPELLIER

OBJECTIVES

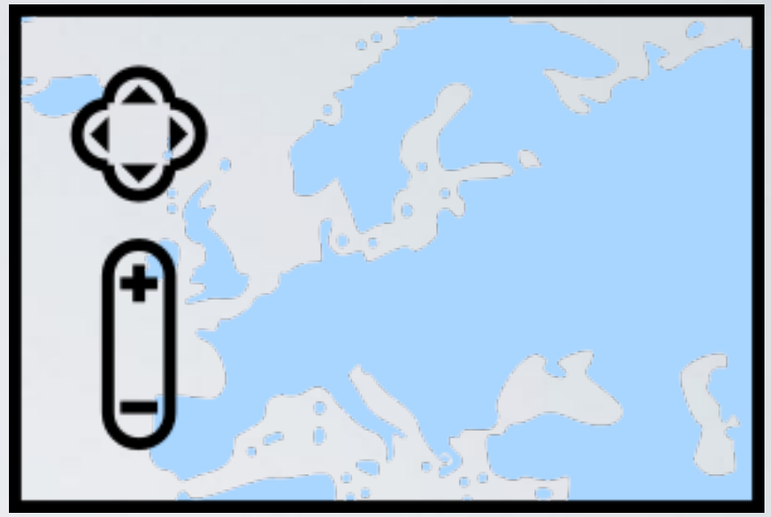
1) Understanding of Java code and project organization

2) Ability to edit, compile and run code on a distant machine hosting a Java application (e.g. in the cloud or on a client's side)

3) Understanding of how good it is to have parts of the process automated in an IDE while still being able to go ahead in case of pbm with the IDE's config files or quick remote edits

POLYTECH MONTPELLIER

HOW DO WE GO THERE?



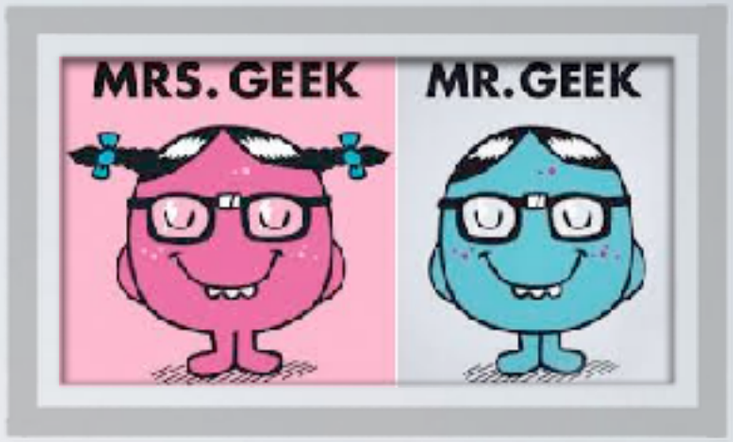
I. Packages

II. Java in the CLI

III. Java archives

IV. Annotation & Documentation

Note on Lab parts



- during this lesson, you're not allowed to use any IDE (no, not even Eclipse, Visual Code, IntelliJ), any Java code must be edited/compiled and run from the terminal (yes you can !):
  - editors: **`nano or emacs -nw`** (Linux and OSX)
  - edit command (Windows)

- Anyone bypassing this will regret it

- Hint: pbm #1 is usually the **`CLASSPATH`** variable handling the access to classes

INTRODUCTION

- Project life cycle:
  - Much more time is spent in debugging and maintaining a software than in developing it
  - developers are numerous and often replaced
- Project sources must then:

1. follow uniform syntax & presentation
   naming and organization rules
2. be structured in an understandable way
   package grouping, design patterns
3. come with enough documentation
   source annotation
4. be easily compiled and distributed
   archives

source : J. Sopena

I - PACKAGES

Images : https://openclipart.org

PACKAGES : DEFINITION

- A « **package** » is a **set of classes** with a common goal (functionality)
- Example of packages available in standard Java :

`java.math` : mathematical operators
`java.awt` : Toolkit for designing graphical interfaces
`java.awt.event` : handling user events
`java.beans` : developing re-usable components
`java.sql` : JDBC API for accessing DBMS

- **Packages** are used to
  - Structure projects
  - Remove ambiguity in names (avoiding conflicts) -> each resource belongs to a specific package, allowing several resources with identical names to be differentiated
  - Ease the access for « friend » classes

INDICATING CLASSES OF A PACKAGE

- Inside a class **A** of a package **p**, when you refer to a class **B**, the compiler checks for the following places to locate **B**:
  - inside file **A.java** (case of an *inner class*) ← See example
  - inside the package **p**
  - in Java's base classes (packages **java.lang**)
- If class **B** is not defined in the above places, the developer has to indicate the compiler where to find it:
  - whenever class **B** is used
  - or better: <u>once and for all at the top of class **A**</u>.
    - ➡ this is done with the **import** keyword

Source : J. Sopena

Example

A refers to class **B**

Helping the compiler to find the `Poum` class of the `pim.pam`, package

1) whenever **class `B`** is used , we indicate the full path to find it: **`pim.pam.Poum`** :

```
public class Canon {
  pim.pam.Poum poum ;

  public Canon () {
    this.poum = new pim.pam.Poum();
  }
  public pim.pam.Poum getPoum() {
    return this.poum ;
  }
}
```


Source : J. Sopena

Example

Helping the compiler to find the `Poum` class of the `pim.pam`, package

2) once and for all at the top of class: indicate that it is in the **`pim.pam`** package:

```
import pim.pam.Poum ;

public class Canon {
  Poum poum ;

  public Canon () {
    this.poum = new Poum();
  }
  public Poum getPoum() {
    return this.poum ;
  }
}
```

Worth noting:

- syntax is
  `import monpackage.*`
  to import all classes of the package
- but « * » does not import classes of the sub-packages. This should be done as follows:

```
import java.awt.*;
import java.awt.event.*;
```

- Importing static resources:

```
import static java.lang.Math.PI
```


Source : J. Sopena

SOURCES BELONGING TO A PACKAGE

- Developers define their own packages: on the **first line** of the `.java` file of a class belonging to your package, include:

```
package bigpack.subpack.mypackage ;
```

- Convention : package names are lowercased (and without numbers nor **-**)

- Packages are also a way to organize Java sources in a **coherent hierarchy** of directories and files:

  - classes of a same package are all stored in a **same** (sub-)**directory**, named named after the (sub-)package
  - A package can have sub-packages that must be sub-directories of the package directory; however, types in these sub packages must be imported to be used by other of different sub-packages and super-packages
  - Classes of these sub-packages are stored in these sub-directories

w.r.t. classes and packages

	private	(default)	protected	public
The same class	Yes	Yes	Yes	Yes
Subclass in the package	No	Yes	Yes	Yes
Non-subclass in the package	No	Yes	Yes	Yes
Subclass in another package	No	No	Yes	Yes
Non-subclass in another package	No	No	No	Yes

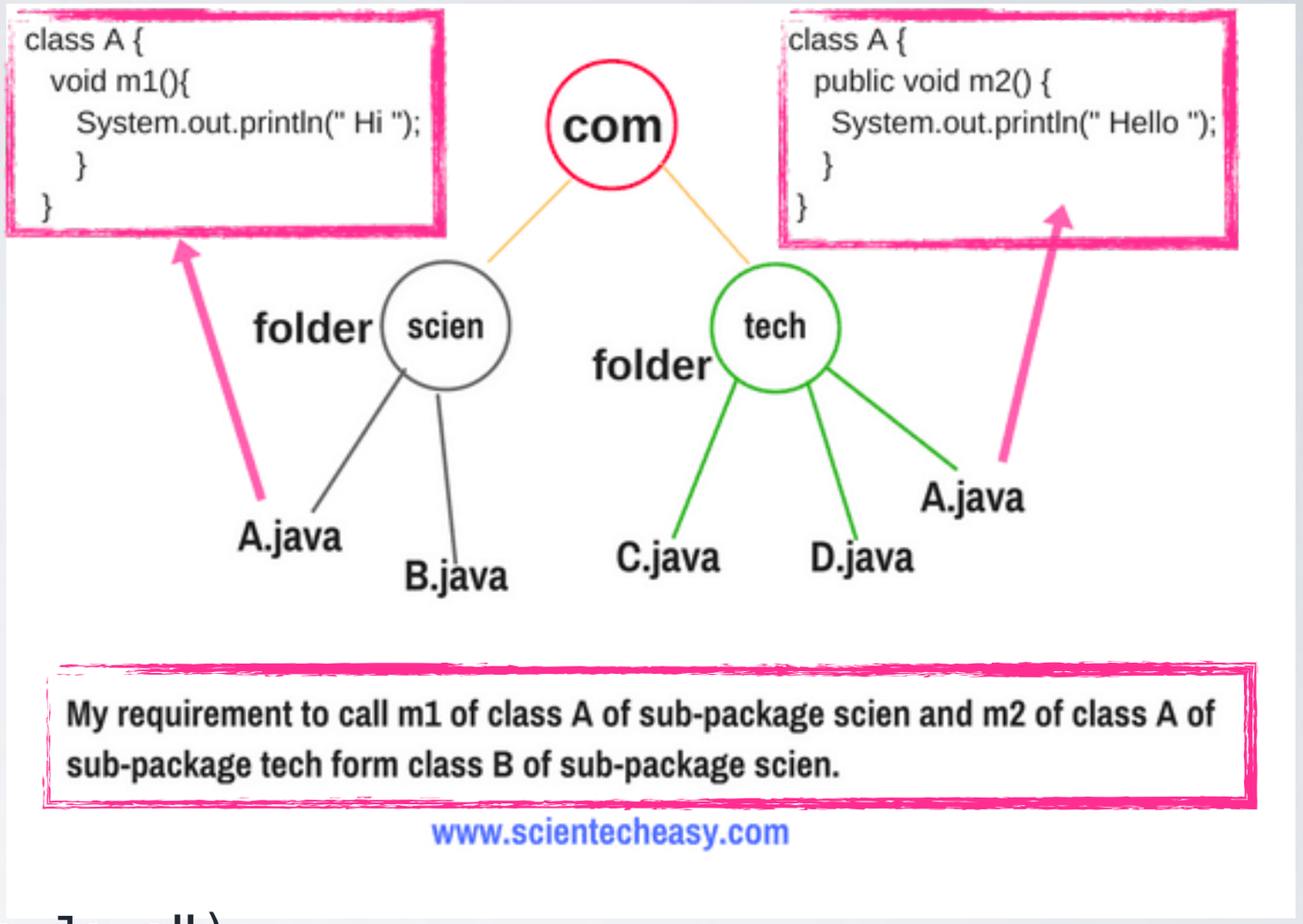
If a type has no visibility modifier then it can be seen only from within its own package (« package-private »)

Developing java classes in packages

1) What is missing in first line of these **`A.java`** files?

2) Solve problems in this code:

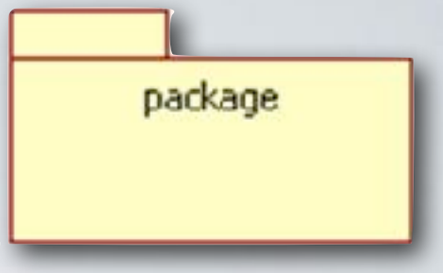
```
package com.scien;    // 1
import com.tech.A;    // 2
class B {
   void m3() {
      System.out.println("Hello Java");
 }
 public static void main(String[ ] args){
    A a=new A();      // 3
    a.m1();           // 4
    A a1=new A();     // 5
    a1.m2();          // 6
    B b=new B();      // 7
    b.m3();           // 8
  }
}
```



SolvePbms.zip

package hierarchy
=
directory hierarchy

DEFAULT PACKAGE



- Classes without a `package` clause are assigned to a default package. Thus these classes all belong the same package

- This is handy to develop a few classes, …

- But in a real **project** use **named packages** and not the default package: the compiler cannot find classes in the default package from a class in a named package

More details here: http://stackoverflow.com/questions/283816/how-to-access-java-classes-in-the-default-package

Source : J. Sopena

NAMING PACKAGES

- Convention : to avoid name conflicts, a company / organization can use its internet domain (in the reverse sense) to distribute its packages

- Example of the *IGcorporation* company:

```
package com.igcorporation.packagezazoo
```

LAB #4 – PART I

1. Suppose the **DO company** needs the following packages: **`student`**, **`room`** (containing a **`plug`** package) and **`stuff`** (containing a **`computer`** package). Create the corresponding folder hierarchy
2. Implement just one class, called **`RJ45`**, in the correct folder with the proper heading and just one attribute.

---

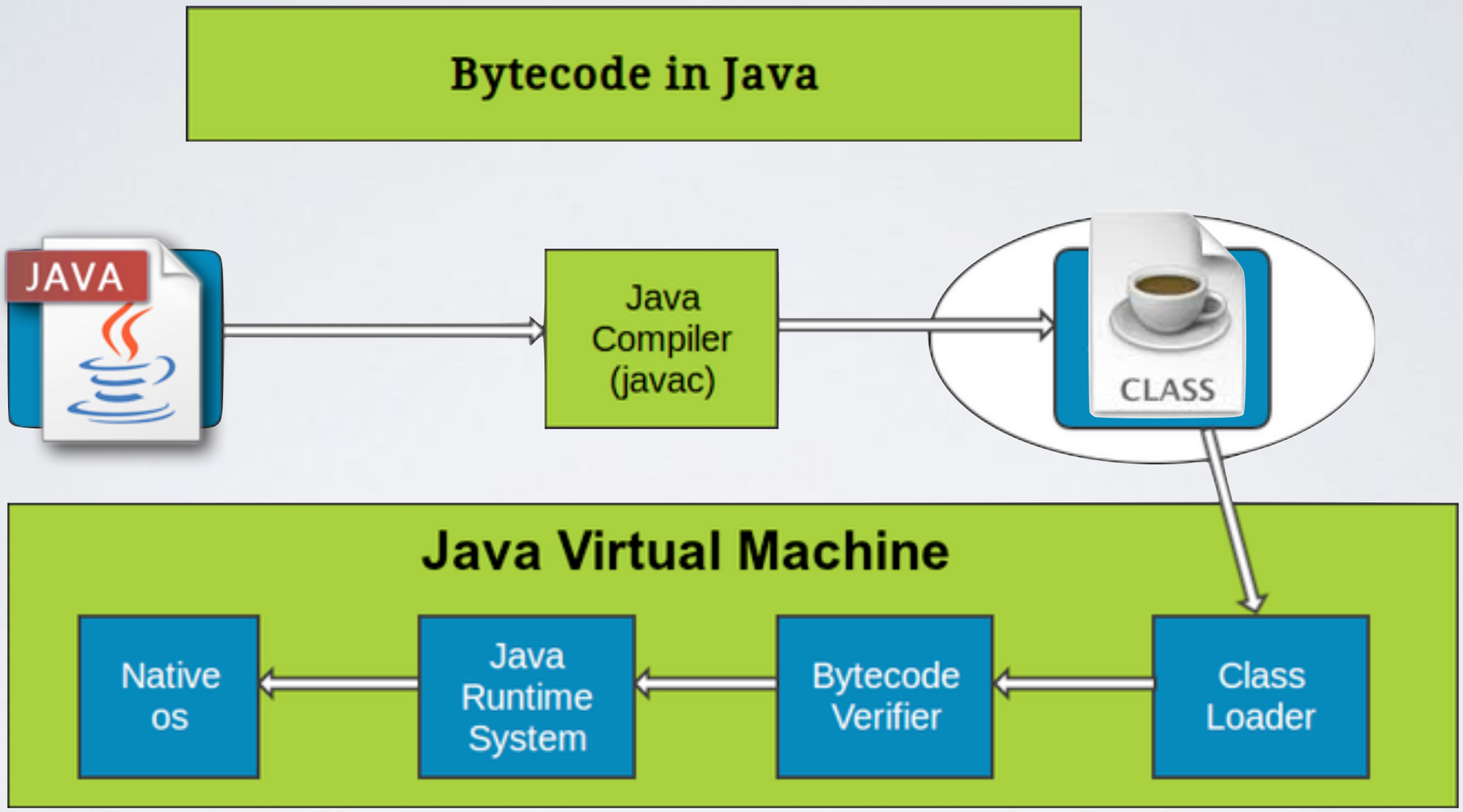
3. Get the java source files for Lab #4, then make a directory called **`part1`** and and put and **`ExampleClassPackage.java`** and **`TestMyClass.java`** files inside. Then correct source code
4. Now put the **`ExampleClassPackage`** class in a subpackage **`mypackage.subPack`** and organize your directories accordingly.

Bonus: can you guess how to compile from the terminal?

II - JAVA IN THE COMMAND LINE INTERFACE

Images : https://openclipart.org

REMINDER



Source : http://java.meritcampus.com

- a source code cannot be run directly by the JRE, it must first be compiled in java «bytecode» (a code that can be run by the JVM)

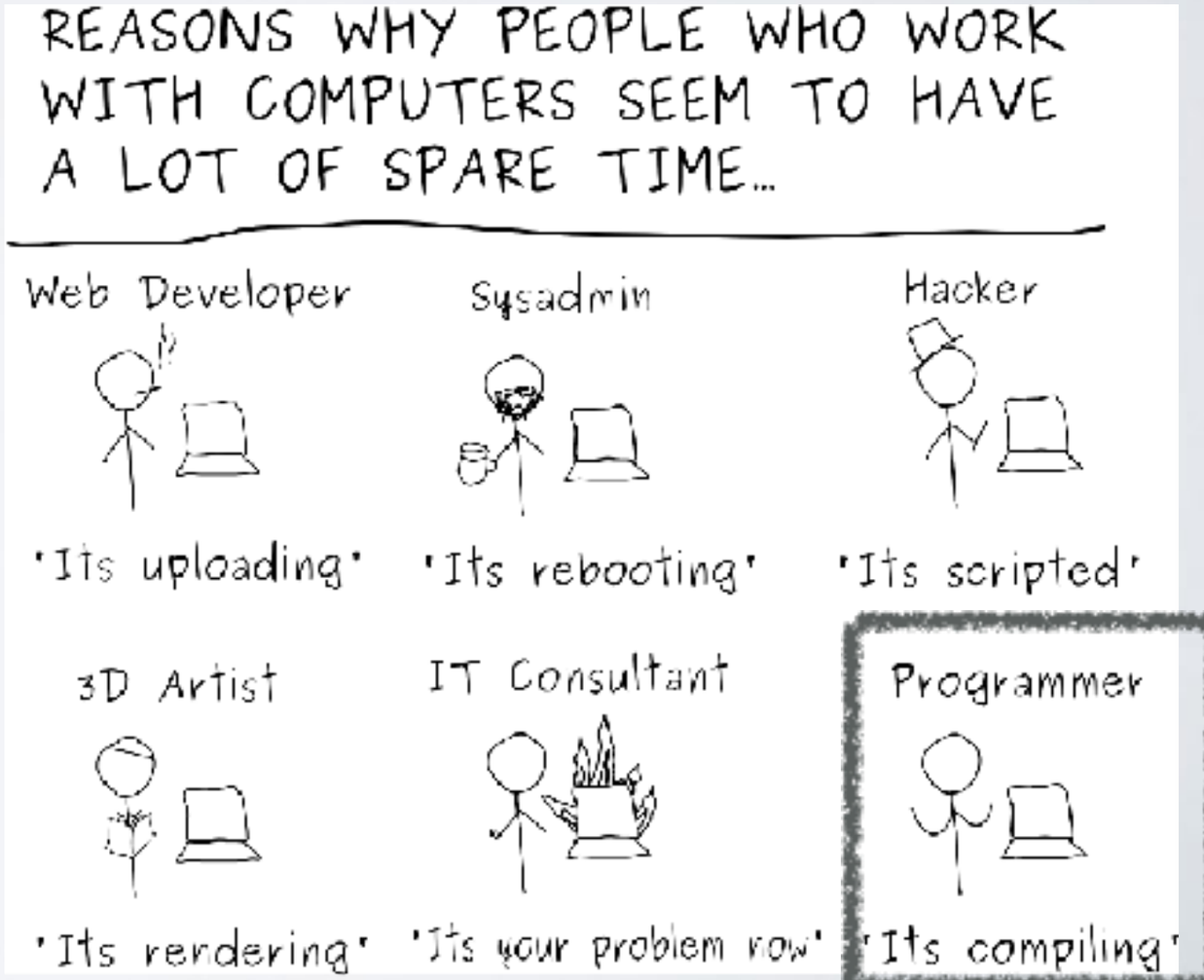
COMPILING

- The `javac` compiler is included in the JDK

  How to use: `javac HelloWorld.java`

- Result: creates a **`HelloWorld.class`** file (bytecode)
- The source code can be prefixed by a path:

```
javac ../MyApp.java
```



Source : R. Grin

COMPILING (CONT'D)

- By default the compiler puts generated bytecodes **in the same directory** as the source code.
- Guess why developers usually prefer separating bytecodes from source files?

- In order to keep them apart, we use specific directories:
  - `src` for all sources, following the package hierarchy
  - `bin` for bytecode files, **still** following the hierarchy
  - then compile indicating a destination directory:

```
javac -d bin MyFantasticApp.java
```


Source :J. Sopena

COMPILING (CONT'D)

- `src` for all sources, following the package hierarchy
- `bin` for bytecode files, **still** following the hierarchy
- then compile indicating a destination directory:

```
javac -d bin src/MyFantasticApp.java
```

Example :

ExampleClassPackage.java

```
1  package mypackage ;
2  public class ExampleClassPackage {
3    public static void main(String args[]) {
4      System.out.println("Hello from ExampleClassPackage");
5    }
6  }
```

When issuing the following command:

```
$ javac -d bin src/mypackage/ExampleClassPackage.java
```

Where goes the created `.class` file?

Source :J. Sopena

COMPILING (CONT'D)

- **Remarks**:
  - the **destination folder** (`bin`) has to pre-exist
  - **dependencies:** other classes referred to by a compiled class are also compiled if their bytecode does not yet exist or if it older than the corresponding source code.
  - to indicate that their ***bytecode*** is in `bin`:

```
javac -d bin -classpath bin src/MyClass.java
```

- **`-classpath <dir>`** : adds **`<dir>`** (could be a list of dirs) to the list of directories where the compiler looks for classes (both compiled and source):

  Excerpt from **`man javac`** :

  - If you use the **`-sourcepath`** option, the compiler searches the indicated path for source files;
  - **`-classpath`** or **`-cp`**
  - otherwise the compiler searches the user **class path** (**`CLASSPATH`** os variable) **both** for **class files and source files**.

LAB #4 – PART II

1) Come back to code in **`ExampleClassPackage`** **`and TestMyClass`** classes and `rm -r *.class` files

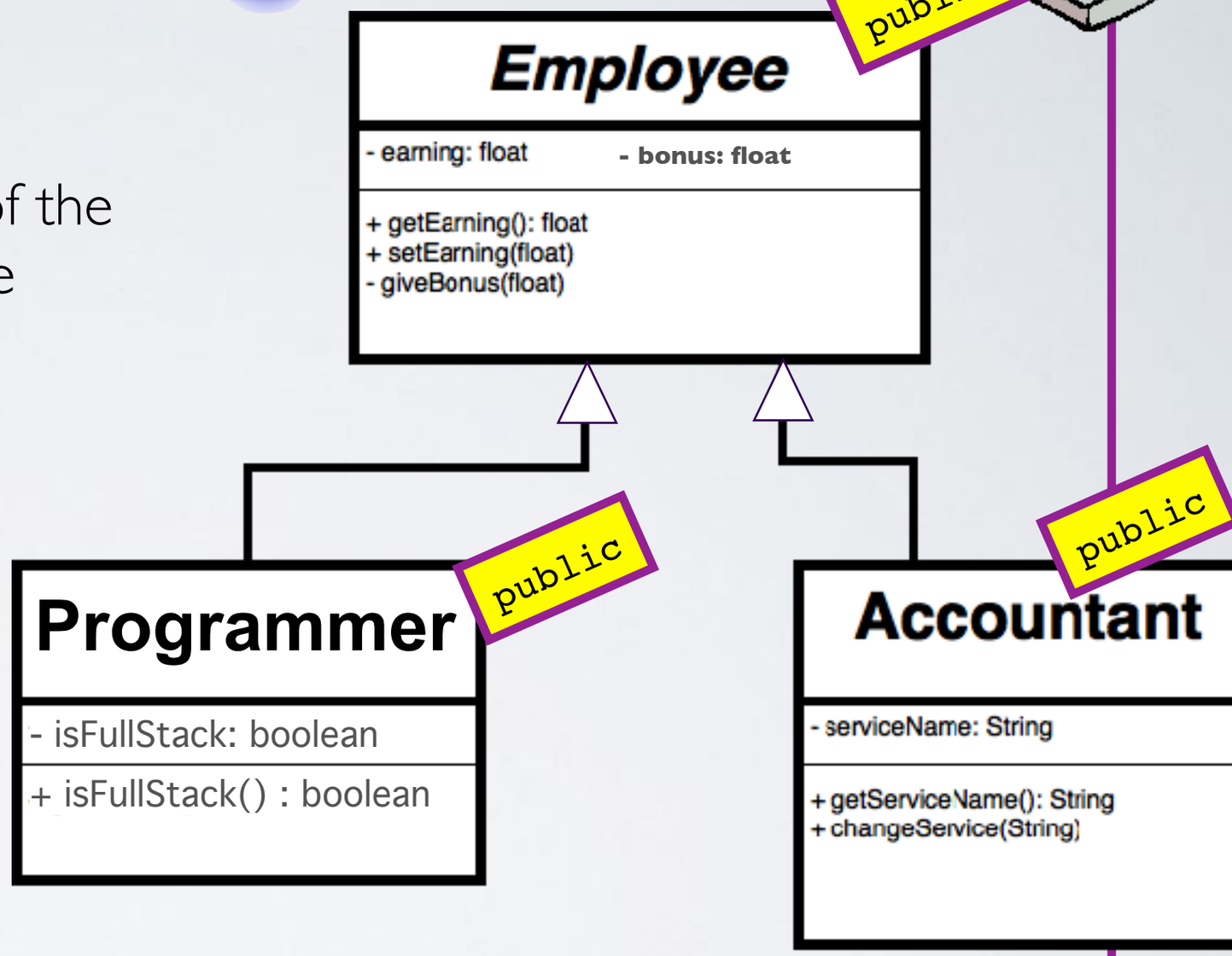
2) Organize **`src`** and **`bin`** directories

3) How to compile these files? (from which folder?)

TRAINING LAB

1. Implement the following 3 classes of the **`company`** package, in the right place with the proper headings (and using **`src`** and **`bin`** folders)

2. Aside from the **`company`** package, implement a **`test`** package and create a **`TestCompany`** class in it, whose **`main(...)`** creates objects from the previous classes and prints simple messages



RUNNING

- The `java` command starts a JVM which loads the java bytecode of the asked program

How to use: `java HelloWorld`

don't mention the `.class` suffix

Where is the `.class` file to be found?

`HelloWorld.class` must be in the **CLASSPATH**

The `classpath` can be given a value:

- by the **`-classpath`** option of the java command:

```
java –cp rep1/rep2 HelloWorld
```

- by the **`CLASSPATH`** environnement variable
- **(by default the classpath is the current directory ?)**

Source : R. Grin

RUNNING

- ***Remark*** : in the **`java`** command, **`HelloWorld`** is a class name, not a file!
- so you can simply not concatenate it with a path for the class:

  WRONG is:

  ```
  java superheroes/marvel/VilainClass
  ```

  ➡ this goes directly into an **error!**

- But if you use the **`classpath`**, java will guess the file name and location from it
- How to indicate that a class belongs to (sub)packages is done as follows:

  GOOD example:

  syntaxe!

  ```
  java mypackage.subpackage.HelloWorld
  ```

From which directory should this command be issued?

Source : R. Grin

LAB #4 – PART III

1) Go to the `bin` directory.
   From there run the `ExClassePackage` class

2) can you run the `TestMyClass` class?

3) Now how to run the same classes while being in the parent directory of `src` and `bin`?
   Try it!

WHAT? SEVERAL CLASSES IN A SAME FILE?!

To avoid! …but this can be handy to have that temporarily for test purposes:

Car.java

```
1  package movingthings;
2  /** Modelizing a car */
3  public class Car {
4    private String driver;
5    private int odometer;
6    public Car () {odometer=0;driver="";}
7    public Car (int odometer) {this();this.odometer=odometer;}
8    public Car(String driver) {this();this.driver = driver;}
9    public Car(String driver, int odometer) {
10     this(driver);this.odometer=odometer;}
11   public String toString() {
12     return "Car of "+driver+" ("+odometer+" km)";}
13   public void drive(int dist) {odometer+=dist;}
14 }
15 /** Testing the Car class */
16 class TestCar {
17   public static void main(String[] args) {
18     if (args.length<1) {System.exit(-1);}
19     Car yourCar = new Car(args[0]);
20     Car myCar = new Car("Arthur",100);
21     System.out.println(myCar+"\n"+yourCar);
22   }
23 }
```

- Compiling generates two `.class` files

Why could the 4th constructor not be written as:
this(driver);
this(odometer);

WHAT? SEVERAL CLASSES IN A SAME FILE?!

Other examples : inner classes and nested classes, check on the internet what's the difference and try to find a cool example and when this is useful:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

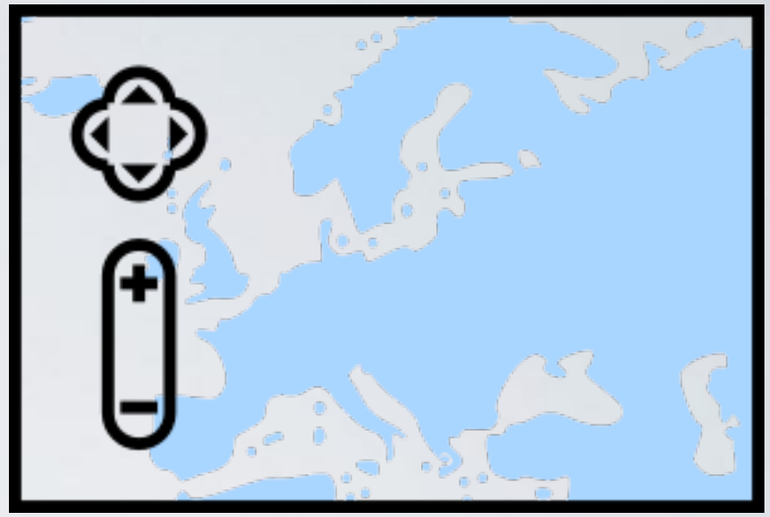
TAKE THE CHALLENGE

Can you in **5mn** (yes, that much for just 4 to 5 commands!):

1) download **`Car.java`**
2) put the file in a correct place, creating along the way other necessary directories
3) and in only 2 commands:
   a. compile this source file, obtaining two bytecodes that are directly placed in the correct directory
   b. run the bytecode, obtaining two lines of result on your screen

....I bet you don't have the shoulders

ON OUR WAY



I. Packages

II. Java in the CLI

III. Java archives

IV. Annotation & Documentation

III - JAVA ARCHIVES

Images : https://openclipart.org

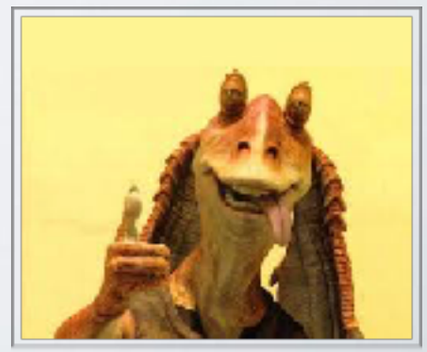
WHAT'S THE POINT?

- Java packages can be stored **compressed**, inside a `.jar` file (**j**ava **ar**chive, which is a special ZIP file).

- Is that file extension remembering you another one?

- Allows to release / download much more easily a java application/library (group of packages and classes).

- Use the **`jar`** command to create such archives



SYNTAX OF THE **JAR** COMMAND

- It takes operations similar to the tar command: **`c`**, **`t`** and **`x`**, as well as the **`v`** option (verbose details) and **`f`** to indicate the archive file name:

Creating an archive recursively including all classes

```
jar cvf SuperHeroes.jar .
```

What's in an archive?

```
jar tf SuperHeroes.jar
```

Extract files from an archive:

```
jar xf SuperHeroes.jar
```

Extract a specific file from an archive:

```
jar xf SuperHeroes.jar Vilains.class
```

SYNTAX OF THE **JAR** COMMAND

- The **`MANIFEST.MF`** file contains all metadata for a jar archive, written as a unique text file stored in the **`META-INF`** root directory
- Among others, it can contain:
  `Manifest-Version`: version number
  `Created-By`: author name
  `Class-Path`: other archives containing dependences (jar libraries)
  `Main-Class`: name of the class containing the main method to be run

- To indicate that the **`main`** is in the **`MyClass`** class, inside **`MyPack`**, we can use the **`m`** option of the **`jar`** command:
  - create a **`Manifest.txt`** file containing:

```
Main-Class: mypackage.MyClass
```

  - **`jar cfm MyJar.jar src/Manifest.txt MyPack/*.class`**

Beware :this file must be UTF-8 encoded **+** ends with a carriage return (\n) **+** have no space (" ") at any end of line

Source : J. Sopena

RUNNING A JAR ARCHIVE

- It's completely possible to run a java app incorporated in a jar file:

```
java -jar [path/]jar-file.jar
```

- Notes:
  - all the app code must be in the archive
  - the **`MANIFEST.MF`** must indicate in which class the **`main`** method is to be found,
    … otherwise:

```
→  sources java java -jar becker.jar
no main manifest attribute, in becker.jar
```

SYNTAX OF THE **JAR** COMMAND

- Other option: ask java to create the manifest file when creating the archive. But we need to indicate the entry point (main class):

```
jar cvfe arch.jar mypack.MyClass
```

- **Warning: by default jar stores the path of files put inside the archive!** (like the tar command). This has implications when you work in a structured way:

```
jar cvf arch.jar bin/graph
added manifest
adding: arch.jar
adding: bin/graph/Edge.class …
java -jar arch.jar
```

Error: Could not find or load main class graph.TestGraph

Two solutions:

1. `cd bin ; jar cvf arch.jar graph`
2. `jar cvf arch.jar -C bin graph`

Source : J. Sopena

A JAR ARCHIVE AS A LIBRARY

- Decompress the archive to get/modify some of its classes is possible

how 

- ...but in practice, jar files are often used as libraries to get some functionality for our app
- In that case, our code uses classes of the jar file, so the jar file must be included in the **`classpath`** when compiling our code:

```
javac -cp libs/mail.jar:../tools/json.jar -d bin src/MyProg.java
```

- and running it:

```
java -cp libs/mail.jar:../tools/json.jar:./bin MyProg
```

indicate the name of the jar, not just its path!

necessary to find our main app class

  - Notes :
    - **`-cp`** or **`-classpath`**
    - arg separator is OS dependent: «**`:`**» (Unix) or «**`;`**» (Windows)

LAB #4 - PART IV

1) Get the `becker.jar` archive and put it in a `myapp/lib` directory

2) Get `SquareMover` source file that make use of this jar archive and put them (once de-zipped) in a `myapp/src` directory

3) How to compile sources of `SquareMover` without uncompressing `becker.jar` file? Find at least two solutions

4) Now, how to run the application?

5) Why do we get failure with this run command when issued from `myapp` folder?

6) `java -cp lib/becker.jar:bin/ squaremover.Main`

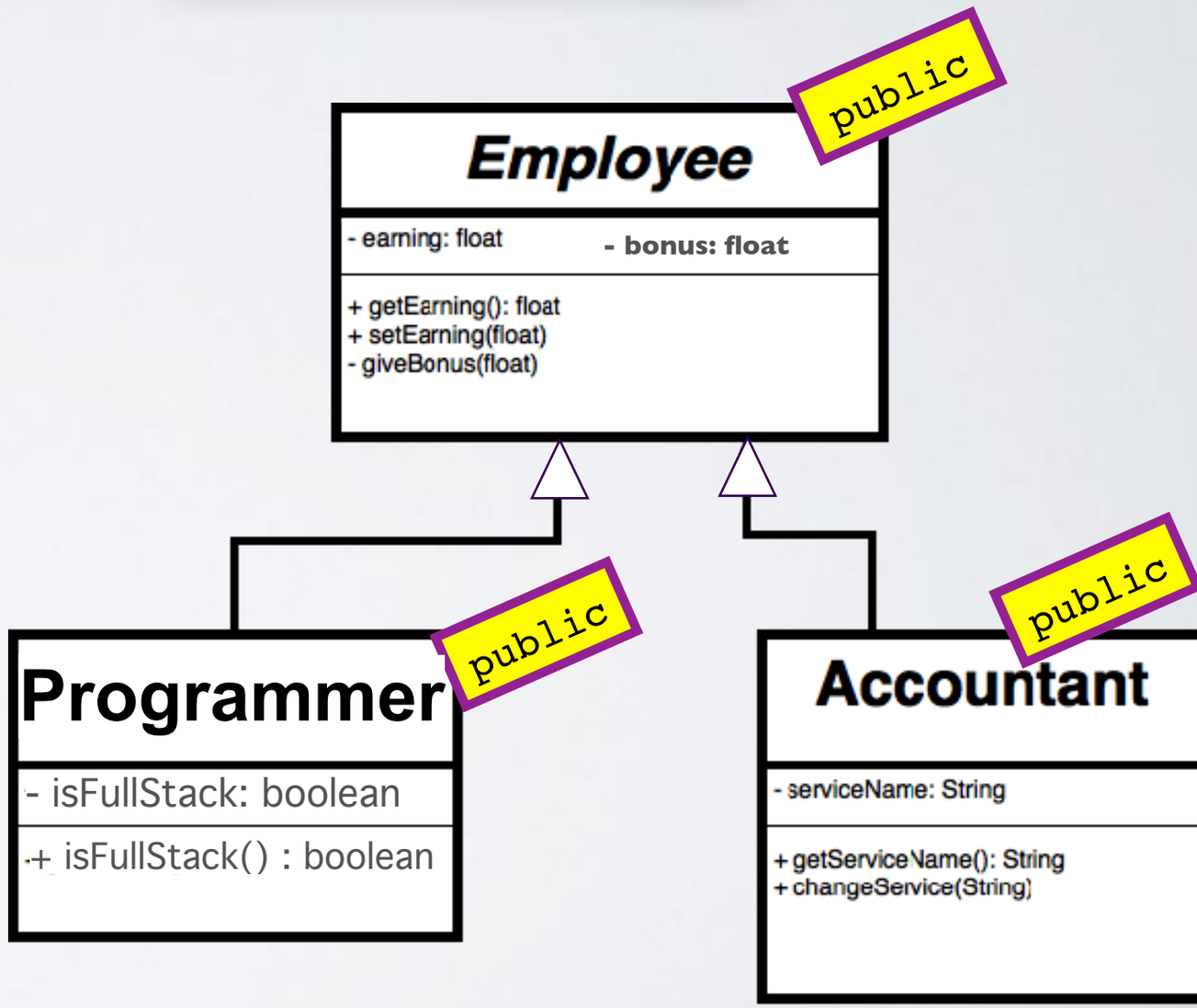
LAB #4 – PART IV

7) Come back to the `Company` application now and package the example we've developed previously inside a `.jar` archive. It should contain everything needed for someone to run it (including the manifest)

8) Use network / email / usb-key to send it to the guy working next to you and in exchange get its `jar` file. Now position the Terminal in the directory where the archive you get is located. Are you able to run the application of the other guy? Is the other guy able to run yours?

9) Now go to a different directory and find a way to run it from this other place.

10) Add a `StudentIntern` class to the application (but outside of the `jar`) you got from the other guy (without uncompressing it!), find how to compile and run the modified application



IV - DOCUMENTATION

Le temps perdu à mettre en place une documentation est du temps gagné pour la suite

Images : https://openclipart.org

COMMENTS OF THE CODE

- **Comments** : parts of the source file ignored by the compiler, but meaning a great deal to debug/maintain/extend the code: it specifies the role of parameters, prerequisites of a method, ...

A good comment:

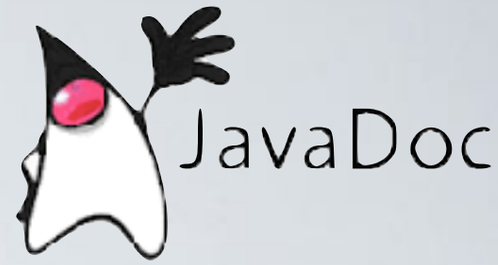
- Does not paraphrase the code
- briefly describes the "why" of the code

Before adding a comment:

1) Question whether the code should not be clarified (rewritten)
2) After refactoring the code: is a comment still necessary?

Source : J. Sopena

DOCUMENTATION



- When delivering an application, you must provide a documentation: what's in it, how it works, how to run it, …

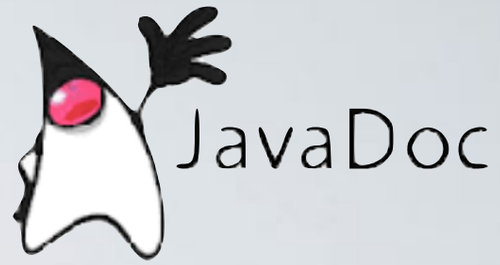
Example : http://www.greenfoot.org/files/javadoc/

- **JavaDoc** is a tool, included in the JDK, allowing to create documentations in HTML (or other) format from the **documentation comments** («*doc* comments») included in the Java source code.

```java
/**
 * Get the sum of two integers
 * @param a The first integer
 * @param b The second integer
 * @return The result of summing the two specified integers
 */
public int sum(int a, int b) {
    return a + b;
}
```


Source : J. Sopena

DOC COMMENTS (CONT'D)

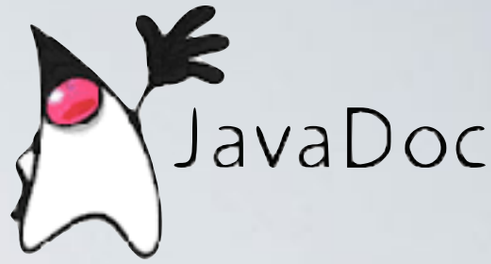


This kind of comment **are like** declarations **they specify** objects used in the code (for explaining code, use traditional comments : `//` or `/*` and `*/`)

Doc comments:

- come **before** what they declare
- begin with `/**` and ends by `*/`
- each line following the first one begins with «`*`»
- use **tags** defined in javadoc, beginning with `@`

DOC COMMENTS (CONT'D)



Acccents are a problem:

- write in english to avoid them
- otherwise write them like &Agrave
- Use specific options:

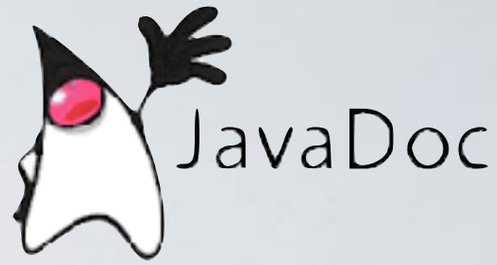
`-encoding` flag to specify the encoding used in source files **`*.java`**

`-docencoding` : to specify the encoding used to generate the HTML documentation

`-charset` : allows to add the encoding in the generated HTML header

```
javadoc -d doc -encoding ISO-8859-1 -docencoding UTF-8  \
  -charset UTF-8 Etudiant.java
```

DOC COMMENTS (CONT'D)



A non-exhaustive list of tags used in JavaDoc comments

TAG	USED WHERE	PURPOSE
`@author` *name*	Interface and Classes	Indicates the author of the code.
`@since` *version*	Interfaces and Classes	Indicates the version item was introduced.
`@version` description	Interfaces and Classes	Indicates the version of the source code.
`@deprecated`	Interfaces, Classes and Methods	Indicates a deprecated API item.
`@param` *name* description	Methods	Indicates the method's parameters.
`@return` description	Methods	Indicates the method's return value.
`@throws` *name* descripion	Methods	Indicates exceptions the method throws.
`@see` Classname	All	Indicates additional class to see.
`@see` Classname#member	All	Indicates additional member to see.

GENERATING THE DOC

In the command line:

```
javadoc <options> file<s>.java
```

-d <dir> :	Specifies the destination directory where javadoc saves the generated HTML files…
-author :	Includes the @author text in the generated docs.
-version :	Includes the @version text in the generated docs. This text is omitted by default.
-public :	Shows only public classes and members.
-protected :	Shows only protected and public classes and members. This is the default.
-package :	Shows only package, protected, and public classes and members.
-private :	Shows all classes and members.

Source : J. Sopena

LAB #4 - PART V

1) in the `company` use case seen earlier on, add doc comments indicating:

   - what packages, classes are for
   - what's the purpose of the fields
   - for some methods: the meaning of their parameters and return values

2) create the documentation of the whole application in a **`doc`** directory, next to **`src`** and **`bin`**

3) Now go consult the generated doc in a browser

GOING FARTHER

- `man javadoc` : very good, many examples
- Oracle website on Javadoc:
  https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

- Now that you know what you're doing, you're encouraged to use Java IDE allows you to code faster, but also to compile / run / debug java files composing an app

➡ Lab part 5 on build tools (Ant, Maven and Gradle) allow to automate the life cycle of an application (building, testing, checking code style, documentation, deploying, …)

LAB #4 - PART 5

Back to our IDE! Paired-reviewed activity on OPs side

& learn from the others

1) Choose two topics: 1 you'd like to read from + 1you'd like to inform others on) among any combination of techno x EDI (Maven / ANT / Gradle / Git integration in IDE for IntelliJ, VScode, web IDE, …)

2) Describe what you expect (sections) in a tutorial you'd like to read

3) Collect material outside of your own knowledge to encompass the topic you'll write on

4) Write a 2 pages tutorial on the techno x IDE you selected (use at most two small yet readable screenshots on each page

5) Review the tutorial that you commanded to others: put annotations and a grade + global comment

6) Correct the tutorial you wrote according to reviews of others