Mise à niveau en Java

Introduction and Basic Concepts
M1

Amin Farvardin
m.Farvardin@outlook.com

Course Presentation

- 19 hours module
- Classes
- Tutorials
- Practical work

Course objectives

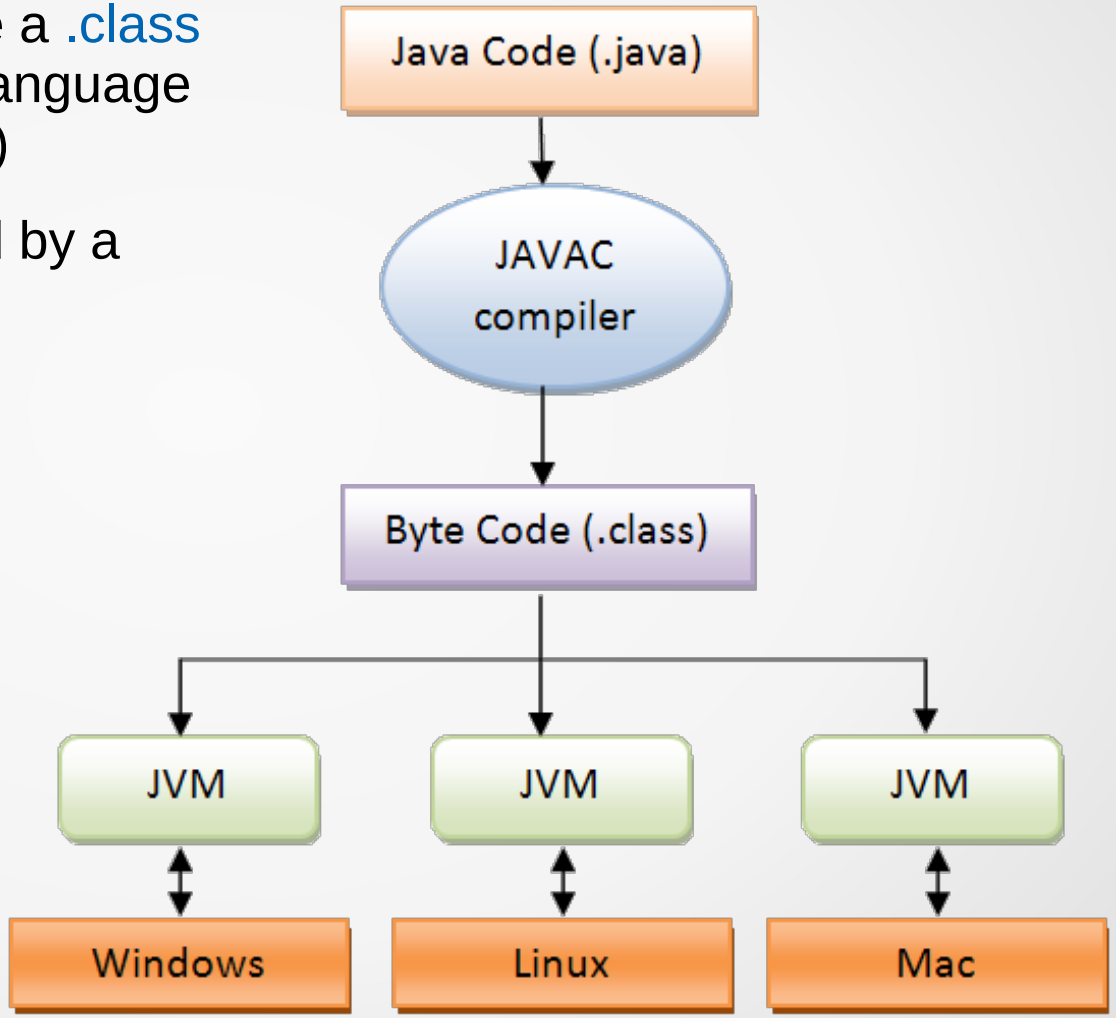
- Introduce the object-oriented paradigm
- Consolidate the basics in object-oriented programming
- Become familiar with the Java language

Generalities

- Little more than a programming language:
  - Object oriented language
  - Platform independent (via VM)
  - Interpreted language and byte code
  - Numerous libraries
  - Strongly typed
    - Any variable must be declared with a type
    - The compiler checks that the uses of variables are compatible with their type
    - The types are on the one hand provided by the language, but also by the definition of the classes

Java Platform

- The Java compiler generate a .class in byte code (intermediate language independent of the platform)
- The byte code is interpreted by a Java JVM interpreter.



General architecture of a Java program

- Java source program = set of ".java" files
- Each “.java” file may contains:
  - One or more class definitions (e.g., **class** A, **class** B, **class** HelloWorld, etc.)
  - Packages (e.g., java.lang, java.util, java.io, etc.)
  - Definition of functions
  - Definition of the **main** program

HelloWorld.java

```
1 public class HelloWorld {
2
3     public static void main(String args[]) {
4         System.out.println("Hello World!");
5     }
6 }
```

Types in Java

- Separation between **primitive types** and **object types**
- Primitive types handled by their **value**
  - Boolean value: boolean (true or false)
  - Signed integer numeric value: byte (8 bits), short (16 bits), int (32 bits), long (64 bits)
  - Floating numeric value: float (32) bits), double (64 bits)
  - Unicode character: char (16 bits)

- Object types handled by reference
  - Present in the JDK API (Date, String, …)
  - User Defined Classes

- When we declare an object variable, we are actually reserving memory space for the reference. (**Class** is like a building blueprint and an **Object** is the building itself)

Objects and Classes

- A **class** is an abstract type characterized by properties common to a set of objects and allowing the creation of objects having these properties.
- A class is made up of:
  - Attributes: data representing the state of the object
  - Methods: operations applicable to objects

- An **object** or a **class instance** has a behavior and state that can only be changed by the actions of the behavior.

Naming

- By convention:
  - Class starts with a capital letter.
  - A method, a field, a local variable start with a lowercase
  - Variables with more than one part, from the second part, start with a capital letter. (e.g., cityName)

MyClass.java

```
1 public class MyClass {
2     public static void main(String args[]) {
3         int temp = 20;
4         String cityName = "Paris";
5
6         System.out.println(cityName + " " + temp);
7     }
8 }
```

Already existing classes

- Java has a large class library. The bookstore consists of different packages and sub-packages:
  - Java.lang: basic types and functions
  - Java.io.File: file management
  - Java.io: input/output management
  - Java.awt: elements of graphical interfaces
  - Java.math: types and mathematical functions
  - And many other packages
- How to use a package in a **class**:

```
import java.lang.Integer;
```

- Full use of a package:

```
import java.lang.*;
```

Useful link: http://docs.oracle.com/javase/7/docs/api/overview-summary.html

Constructor

- To create an **object** from a class, we use the new operator

```
MyClass c1 = new MyClass();
```

- The **new** operator calls the class constructor.
- A constructor has the same name as the class in which it is defined.
- A constructor has no return type (not even void)

Greeting.java

```
1  public class Greeting {
2
3      public Greeting() {
4          System.out.println("Hi, Welcome!");
5      }
6
7      // declaration of attributes
8      // ...
9      // definition of methods
10     // ...
11
12     public static void main(String args[]) {
13         // your code
14     }
15 }
```

Constructor Cont.

- **Default constructor**: constructor without arguments, initializes the variables of the class to the default values

```
Person() {}
OR
Person() {
        age=25;
        nationality= ‘french’;
}
```

- **Overloaded constructor**: constructor with different signatures

```
Person(int age, char nationality) {
        this.age=age;
        this.nationality: nationality;
}
```

Constructor Cont.

- If no constructor is created in the class, the java compiler automatically creates a default constructor.
- If an overloaded constructor is created in the class, the default constructor will no longer be created by the compiler.
- The java platform differentiates between the different constructors declared within the same class based on the number of parameters and their types.
- You cannot create two constructors with the same number and types of parameters.

```
Person(int age) {
   this.age=age;
}
Person(int age) {
   his.age=age*2;
}
```

Compilation error

Constructor Cont.

Person.java

```
public class Person{
        public String firstName;
        public String lastName;
        public int age;

        // Definition of a constructor
        public Person(String fn, String ln, int a)
        {
                this.firstName = fn;
                this.lastName = ln;
                this.age=a;
        }
}
```

Application.java

```
public class Application
{
        public static void main (String args[])
        {
                Person jean = new Person();
                jean.setName('Jean');
        }
}
```

!! Compilation Error
Default constructor no longer exists

Constructor Cont.

Shirt.java

```
public class Shirt{
        int id;
        char color;
        float price;
        int quantity;
        String description;

        Shirt(){}

        Shirt (int id){
                this.id = id;
        }

        Shirt (int id, char color){
                this.id = id;
                this.color = color;
        }
}
```

Which constructor will choose Java when from the compilation?

1) Shirt sh1 = new Shirt();
2) Shirt sh1 = new Shirt(122);
3) Shirt sh1 = new Shirt(122, 'B');

Visibility (4 visibility modifiers for members of a class)

- **public**: Visible to everyone
- **private**: Visible only in the classroom
- **protected**: Visible by inherited classes and that of the same package
- Without modifier: Visible by inherited classes and that of the same package

	public	private	protected	No modifier
In the same class	Yes	Yes	Yes	Yes
In a class of the same package	Yes	No	Yes	Yes
In a subclass of another package	Yes	No	Yes	No
In any class of another package	Yes	No	No	No

Instance variables and class variables

Instance variables:

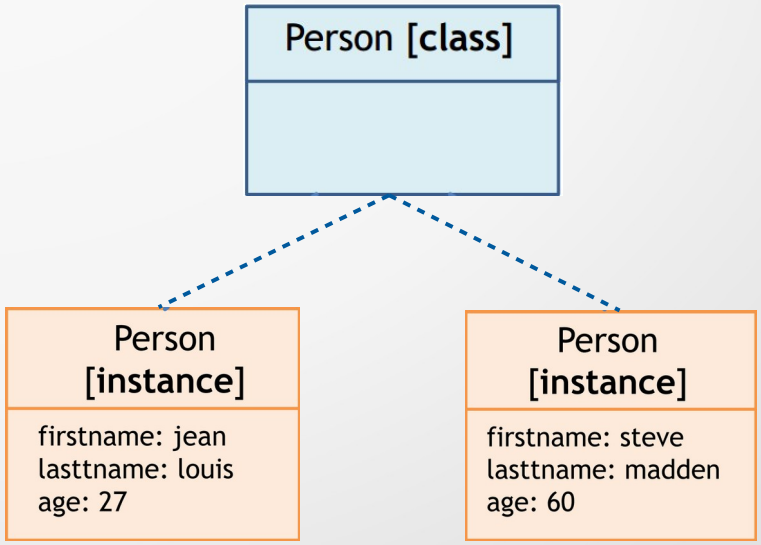
- Each instance of a class has its own variable values. These variables define the characteristics of the object.

Access: <Object name>.<Attribute name> => An attribute value of the corresponding object
Example: *System.out.println*(p1.firstname) => jean

```
public class Person{
	public String firstname;
	public String lastname;
	public int age;

	public Person(String fn, String ln, int a)
	{
		this.firstname=fn;
		this.lastname=ln;
		this.age=a;
	}
}
```

- Person p1 = new Person('jean', 'louis',27);
- Person p2 = new Person('steve', 'madden',60);



Instance variables and class variables

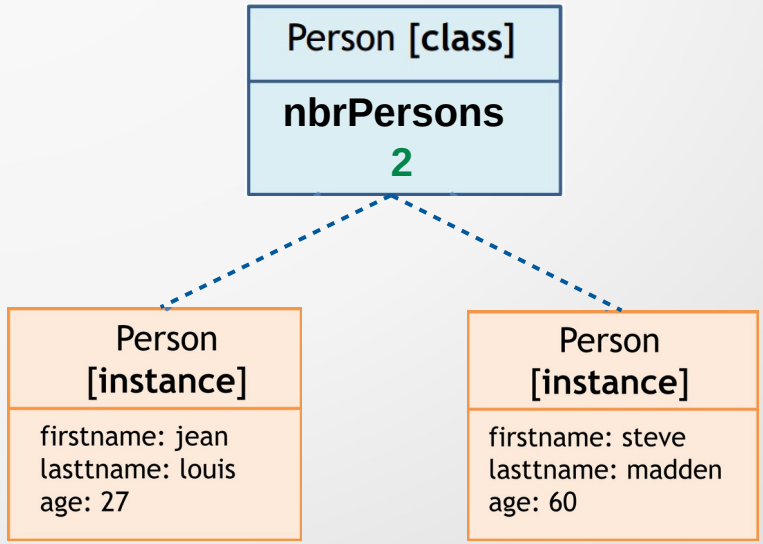
Class variables

- The value of a class variable is **common** to all instances of the class, that is, the class variable does not belong to a particular instance.
  - Statement with the keyword **static**
  - Mandatory initialization

access : <class name>.<number of the class variables>
Example: *System.out.println*(**Person.nbrPersons**) => 2

```
public class Person{
        public String firstname;
        …
        static int nbrPersons;
        public Person(String fn, String ln, int a)
        {
                this.firstname=fn;
                …
                nbrPersons++;
        }
}
```

- Person p1 = new Person(‘jean’, ‘louis’,27);
- Person p2 = new Person(‘steve’, ‘madden’,60);



Instance methods and class methods

- instance method: these methods allow you to modify or access the state of the object.
- class method: these methods do not change the internal state of an object.

Example: the Float class

- Instance method of class String: *toString()*
  - returns a string representation of the current object
- class method of static String: *toString(Float f)*
  - returns a string representation of the float passed as a parameter

```
Float f = 3F;
System.out.println(f.toString());
System.out.println(Float.toString(3.14F));
```

Destruction of an object

- The destruction of objects is handled by Java using a garbage collector (GC).
- The GC destroys objects (i.e. clears memory) that are not referenced by any other object.
- The destruction is asynchronous and there is no guarantee that the objects will be destroyed.
- An optional method called *finalize()* is called when the object is destroyed.
  - For example, it can ensure that files or connections are closed before the destruction of the object.