

# Mise à niveau en Java

Legacy and consequences  
M1

*Amin Farvardin*  
[m.Farvardin@hotmail.com](mailto:m.Farvardin@hotmail.com)

# Goal: not to code the same thing

```
Person.java
1 public class Person {
2     public String name;
3
4     // default constructor
5     public Person() {
6         this.name = "unknown";
7     }
8
9     public Person(String name) {
10        this.name = name;
11    }
12 }
```

- we now want to make "*Gaulois*" and "*Romains*" classes to have more specific behaviors.
- How to do it?
  - Copy what was done in the "Person" class and add specific methods.

# Goal: not to code the same thing

```
Person.java
1 public class Person {
2     public String name;
3
4     // default constructor
5     public Person() {
6         this.name = "unknown";
7     }
8
9     public Person(String name) {
10        this.name = name;
11    }
12 }
```

- we now want to make "*Gaulois*" and "*Romains*" classes to have more specific behaviors.
- How to do it?
  - ~~Copy what was done in the "Preson" class and add specific methods.~~

# Goal: not to code the same thing

```
Person.java
1 public class Person {
2     public String name;
3
4     // default constructor
5     public Person() {
6         this.name = "unknown";
7     }
8
9     public Person(String name) {
10        this.name = name;
11    }
12 }
```

- we now want to make "*Gaulois*" and "*Romains*" classes to have more specific behaviors.
- How to do it?
  - ~~Copy what was done in the "Person" class and add specific methods.~~
  - Actually, we don't want to duplicate code! But how?

# Goal: not to code the same thing

```
Person.java
1 public class Person {
2     public String name;
3
4     // default constructor
5     public Person() {
6         this.name = "unknown";
7     }
8
9     public Person(String name) {
10        this.name = name;
11    }
12 }
```

we now want to make "*Gaulois*" and "*Roman*" classes to have more specific behaviors.

- How to do it?
  - ~~Copy what was done in the "Person" class and add specific methods.~~
  - Actually, we don't want to duplicate code! But how?

Java offers **inheritance** as a solution.

# Heritage

- **Inheritance** allows an object to acquire the properties of another object factorization:
- The parent class (or base class) is more general.
  - It contains the properties common to all the child classes (or derived or inherited class)
- The daughter classes have more specific properties:
  - We obtain a hierarchy of classes.
- To express that a class is a **child** class, we use the **extends** keyword in the declaration of a class:

```
class <child_class_name> extends <parent_class_name>
```

- In Java, we are able to inherit from only a unique class.
- (A class that does not inherit from any class that actually inherits implicitly from the Object class)

# Consequences

- What happens to the instance variables.
- What happens to the methods in the parent class.
- Constructors

# public and private members

- Methods or attributes
  - **public** members are always accessible by a child class.
  - **private** members remain inaccessible, even for a child class!
    - It may seem a little surprising.
    - Obviously, the private attributes are still inherited: even if we do not have direct access to the attributes, they are present!
    - Leave the freedom to change the attribute in the parent class without having to change anything in the child classes.

```
1 public class Person {  
2     private String name;  
3  
4     public String getName() {  
5         return this.name;  
6     }  
7 }
```

```
1 public class Gaulois extends Person {  
2  
3     public String presentation() {  
4         return name + "I am Gaulois.";  
5     }  
6 }
```



# Protected members - **protected**

- new **protected** scope:
  - only the class and the derived classes have access to members declared **protected**.

# Parent class methods

- For the **public** or **protected** methods, we have the choice:
  - Either the behavior is the same: we can/must omit the rewriting of the method.
  - The behavior is different: we can rewrite the method.
- there are two references to browse the hierarchy:
  - **this**: is a reference on the instance of the class.
  - **super**: is a reference on the parent instance
- Obviously, we can add specific methods to the child class!

# Child class constructor

- 1) Call the constructor of the parent class: the method is called **super** quite simply.
  - If the call is not explicit, Java will try to automatically call the default constructor (without an argument).
  - If you have defined a constructor instead of the default constructor in the parent class, therefore, you will need to call explicitly the constructor of the parent class.
- 2) Carry out specific treatments for the child class.

# Example

```
1 public class Person {
2     private String name;
3
4     public Person(String name) {
5         this.name = name;
6     }
7
8     public String presentation() {
9         return "My name is " + this.name + ".";
10    }
11 }
```

```
1 public class Gaulois extends Person {
2
3     public Gaulois (String name) {
4         super(name);
5     }
6
7     public String presentation() {
8         return super.presentation() + " I am a Gaulois.";
9     }
10
11     public static void main(String[] args) {
12         Gaulois asterix = new Gaulois("Asterix");
13         System.out.println(asterix.presentation());
14     }
15 }
```

# Example

```
1 public class Person {
2     private String name;
3
4     public Person(String name) {
5         this.name = name;
6     }
7
8     public String presentation() {
9         return "My name is " + this.name + ".";
10    }
11 }
```

Output:

My name is Asterix. I am a Gaulois.

```
1 public class Gaulois extends Person {
2
3     public Gaulois (String name) {
4         super(name);
5     }
6
7     public String presentation() {
8         return super.presentation() + " I am a Gaulois.";
9     }
10
11    public static void main(String[] args) {
12        Gaulois asterix = new Gaulois("Asterix");
13        System.out.println(asterix.presentation());
14    }
15 }
```

# instanceof operator

- Operator to check if a class is indeed an instance of a class.

```
1 public class Person {
```

```
1 public class Gaulois extends Person {
```

```
1 public class IrreducibleGaulois extends Gaulois {
```

```
1 public class Romain extends Person {
```

```
2
```

```
3 public static void main(String[] args) {
```

```
4     IrreducibleGaulois asterix = new IrreducibleGaulois();
```

```
5     System.out.println(asterix instanceof Person);
```

```
6     System.out.println(asterix instanceof Gaulois);
```

```
7
```

```
8 }
```

```
9 // ...
```

# instanceof operator

- Operator to check if a class is indeed an instance of a class.

```
1 public class Person {
```

```
1 public class Gaulois extends Person {
```

```
1 public class IrreducibleGaulois extends Gaulois {
```

```
1 public class Romain extends Person {  
2  
3     public static void main(String[] args) {  
4         IrreducibleGaulois asterix = new IrreducibleGaulois();  
5         System.out.println(asterix instanceof Person);  
6         System.out.println(asterix instanceof Gaulois);  
7  
8     }  
9 // ...
```

true

true

Asterix is indeed a Gaulois, he is even an irreducible Gaulois, and especially not a Romans.



# Access Levels (recap)

Modifier	Class	Package	Subclass	World
-----				
public	Y	Y	Y	Y
protected	Y	Y	Y	N
(Default)	Y	Y	N	N
private	Y	N	N	N



# Polymorphism

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

```
class Animals {
    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}

class Pig extends Animals {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}

class Dog extends Animals {
    public void animalSound() {
        System.out.println("The dog says: bow wow");
    }
}

class Bird extends Animals {
```

# Polymorphism

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

```
class Animals {
    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}

class Pig extends Animals {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}

class Dog extends Animals {
    public void animalSound() {
        System.out.println("The dog says: bow wow");
    }
}

class Bird extends Animals {
}
```

```
class MyClass {

    public static void main(String[] args) {
        Animals a = new Animals();
        Pig p = new Pig();
        Dog d = new Dog();
        Bird b = new Bird();

        a.animalSound();
        p.animalSound();
        d.animalSound();
        b.animalSound();
    }
}
```

# Polymorphism

- Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.
- Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

```
class Animals {
    public void animalSound() {
        System.out.println("The animal makes a sound");
    }
}

class Pig extends Animals {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }
}

class Dog extends Animals {
    public void animalSound() {
        System.out.println("The dog says: bow wow");
    }
}

class Bird extends Animals {
```

```
class MyClass {

    public static void main(String[] args) {
        Animals a = new Animals();
        Pig p = new Pig();
        Dog d = new Dog();
        Bird b = new Bird();

        a.animalSound();
        p.animalSound();
        d.animalSound();
        b.animalSound();
    }
}
```

## Output:

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
The animal makes a sound
```

# final keyword (a non-access modifier)

- **For a variable:** the variable cannot be modified.
- **For a method:** this method cannot be re-defined in a derived class.
- **For a class:** a final class will not have a child class
  - security reason to avoid "hijackings".

**Final Variable** → **To create constant variables**

**Final Methods** → **Prevent Method Overriding**

**Final Classes** → **Prevent Inheritance**

# final variables

- When a variable is declared with final keyword, its value can't be modified.
- This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed
  - i.e. you can add or remove elements from final array or final collection.
- It is good practice to represent final variables in all uppercase, using underscore to separate words.

# final methods

- When a method is declared with final keyword, it is called a final method.
- A final method cannot be overridden.
- We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes.



# final classes

- When a class is declared with final keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class:
  - **To prevent inheritance**, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float etc. are final classes. We can not extend them.
  - **To create an immutable class** like the predefined String class. You can not make a class immutable without making it final.

# Java **abstract** Classes and Methods

- Data abstraction is the process of **hiding certain details** and **showing only essential** information to the user.
  - Goal of use: To achieve security.
- Abstraction can be achieved with either abstract classes or interfaces (later we talk about it)
- The **abstract** keyword is a non-access modifier, used for classes and methods:
  - Abstract class is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
  - Abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
- An abstract class can have both abstract and regular methods:



# Example 1

```
abstract class Animal {  
    public abstract void animalSound();  
  
    public void sleep() {  
        System.out.println("Zzz");  
    }  
}
```

From the example above, it is not possible to create an object from the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

# Example 2

```
// Abstract class
abstract class Animal {
    // Abstract method (does not have a body)
    public abstract void animalSound();
    // Regular method
    public void sleep() {
        System.out.println("Zzz");
    }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}

class MyClass {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

## Output:

The pig says: wee wee  
Zzz

# abstract Vs. final

- In java, you will never see a class or method declared with both final and abstract keywords.
- For classes, final is used to prevent inheritance whereas abstract classes depends upon their child classes for complete implementation.
- In cases of methods, final is used to prevent overriding whereas abstract methods needs to be overridden in sub-classes.

# Interface (Another way to achieve abstraction)

- An interface is a completely "abstract class" that is used to group related methods with empty bodies:

```
// interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void run(); // interface method (does not have a body)
}
```

- To access the interface methods by another class, we use **implements** keyword (instead of extends).
- The body of the interface method is provided by the "implement" class.
- A class can have more than one interface.

# Example

```
// Interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        // The body of sleep() is provided here
        System.out.println("Zzz");
    }
}
```

```
class MyMainClass {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

## Output:

The pig says: wee wee  
Zzz

# Interface - recap

- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default **abstract** and **public**
- Interface attributes are by default **public**, **static** and **final**
- An interface cannot contain a constructor (as it cannot be used to create objects)

## Why And When To Use Interfaces?

- 1) To **achieve security** - hide certain details and only show the important details of an object (interface).
- 2) **Java does not support "multiple inheritance" but support "multiple implements"**
  - A class can only inherit from one superclass. However, it can be achieved with interfaces, because the class can implement multiple interfaces.

# Multiple Interfaces

**Note:** To implement multiple interfaces, separate them with a comma.

```
interface FirstInterface {  
    public void myMethod(); // interface method  
}  
  
interface SecondInterface {  
    public void myOtherMethod(); // interface method  
}  
  
class DemoClass implements FirstInterface, SecondInterface {  
    public void myMethod() {  
        System.out.println("Some text..");  
    }  
    public void myOtherMethod() {  
        System.out.println("Some other text...");  
    }  
}
```

