Object-Oriented Programming: Application to the Java Language

Inheritance

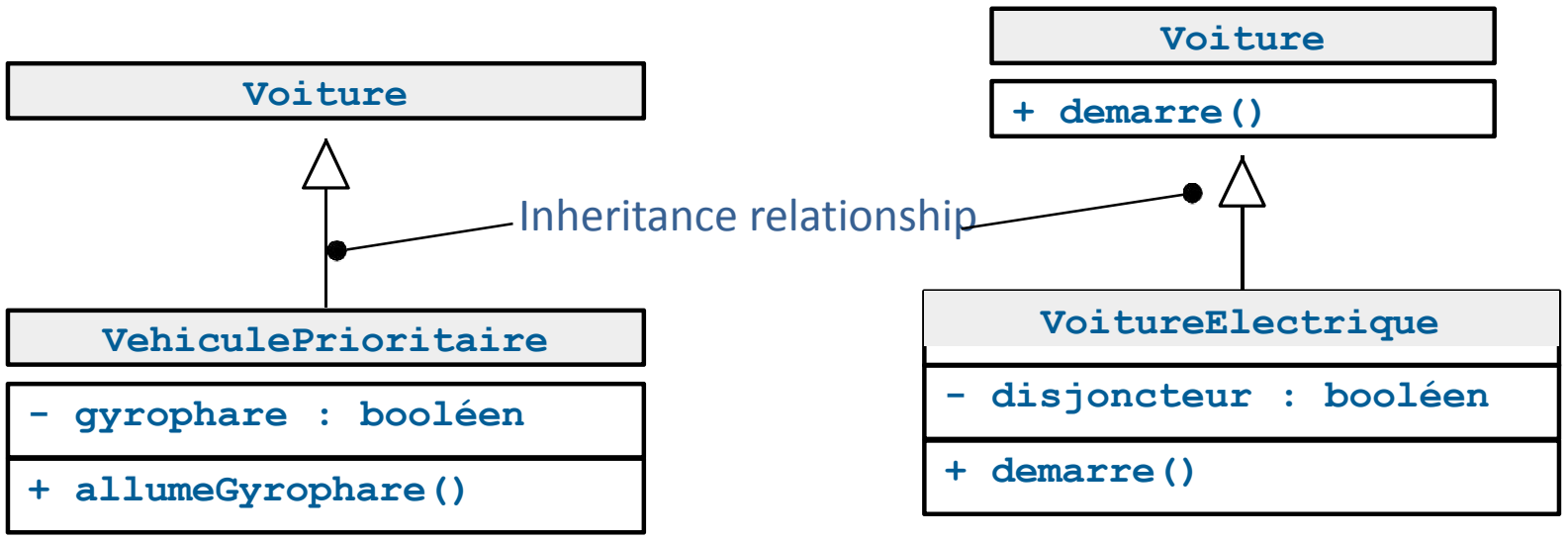
Definition and Benefits

- Inheritance
  - A technique provided by programming languages to build a class from one (or more) other classes by sharing its attributes and operations.

- Benefits
  - **Specialization, enrichment**: A new class reuses the attributes and operations of an existing class while adding and/or modifying specific operations for the new class.
  - **Redefinition**: A new class redefines the attributes and operations of an existing class to change its meaning and/or behavior for the specific case defined by the new class.
  - **Reuse**: Avoids rewriting existing code, especially when the source code of the inherited class is not available.

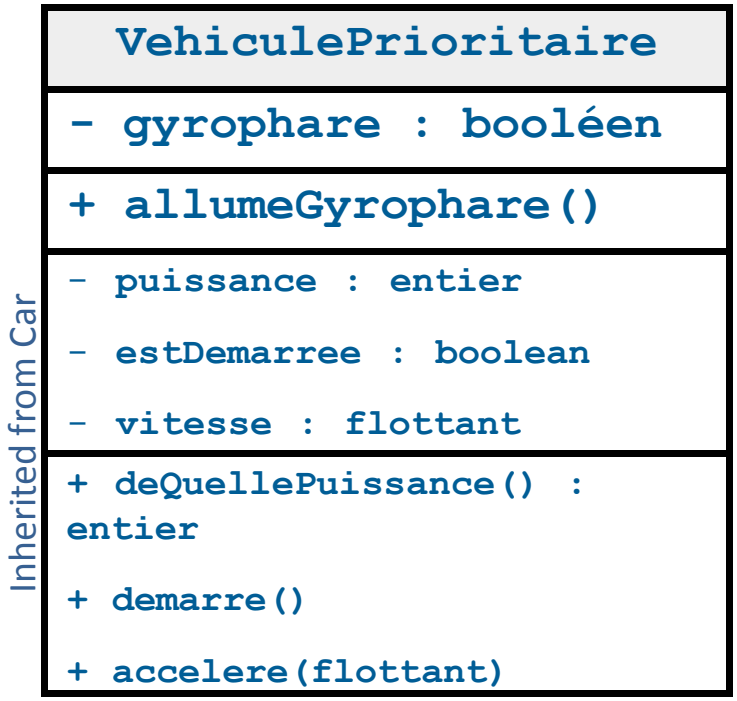
Specialization of the “Voiture" class

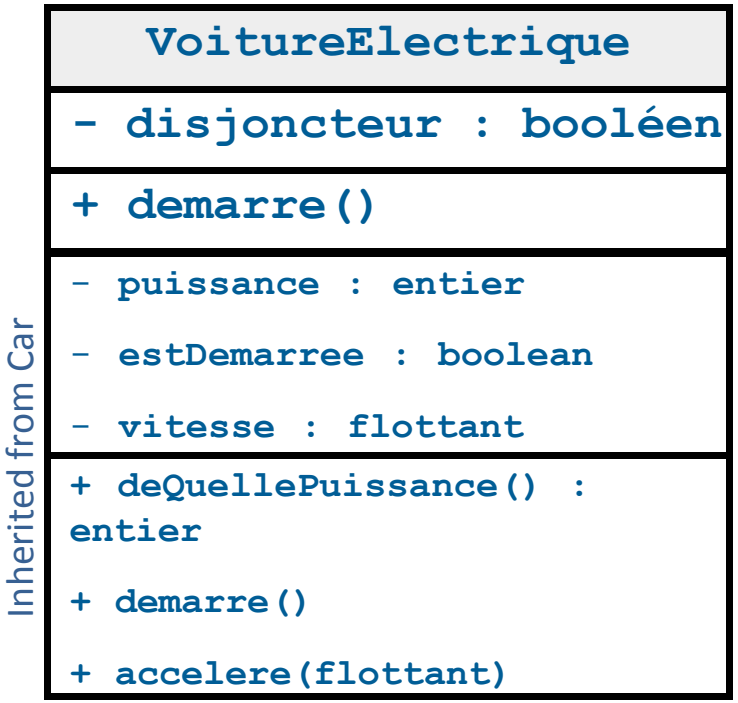
- A priority vehicle is a car with a siren light.
  - A priority vehicle responds to the same messages as the Car.
  - You can turn on the siren light of a priority vehicle.
- An electric car is a car whose starting operation is different.
  - An electric car responds to the same messages as the Car.
  - An electric car is started by activating a circuit breaker.



Classes and subclasses

• An object of the class *VehiculePrioritaire* or *VoitureElectrique* is also an object of the class *Voiture*, so it has all the attributes and operations of the *Voiture* class.





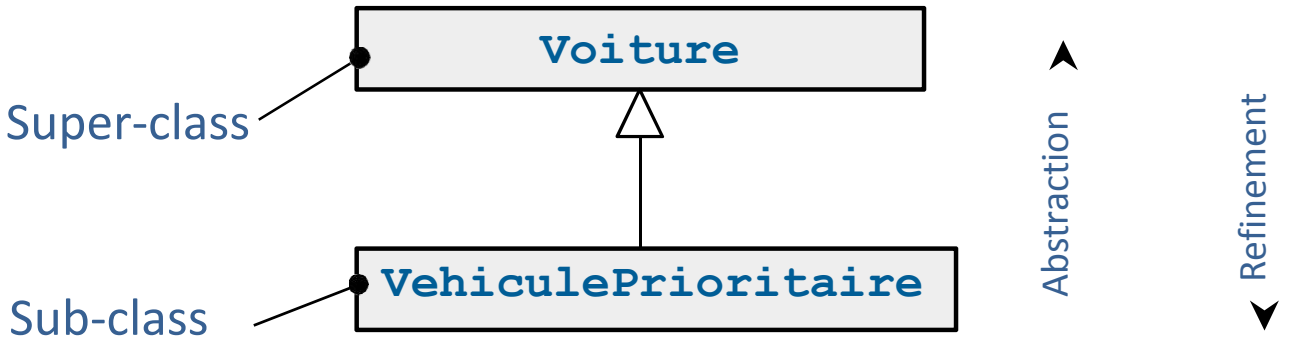
Classes and Subclasses: Terminology

- Definitions
  - The class *VehiculePrioritaire* **inherits** from the class *Voiture*
  - *Voiture* is the **parent class** and *VehiculePrioritaire* is the **child class**
  - *Voiture* is the **superclass** of the class *VehiculePrioritaire*
  - *VehiculePrioritaire* is a **subclass** of *Voiture*

- Attention 
  - An object of the *VehiculePrioritaire* class or *VoitureElectrique* class is necessarily an object of the *Voiture* class
  - An object of the *Voiture* class is not necessarily an object of the *VehiculePrioritaire* or *VoitureElectrique* class

Generalization and Specialization

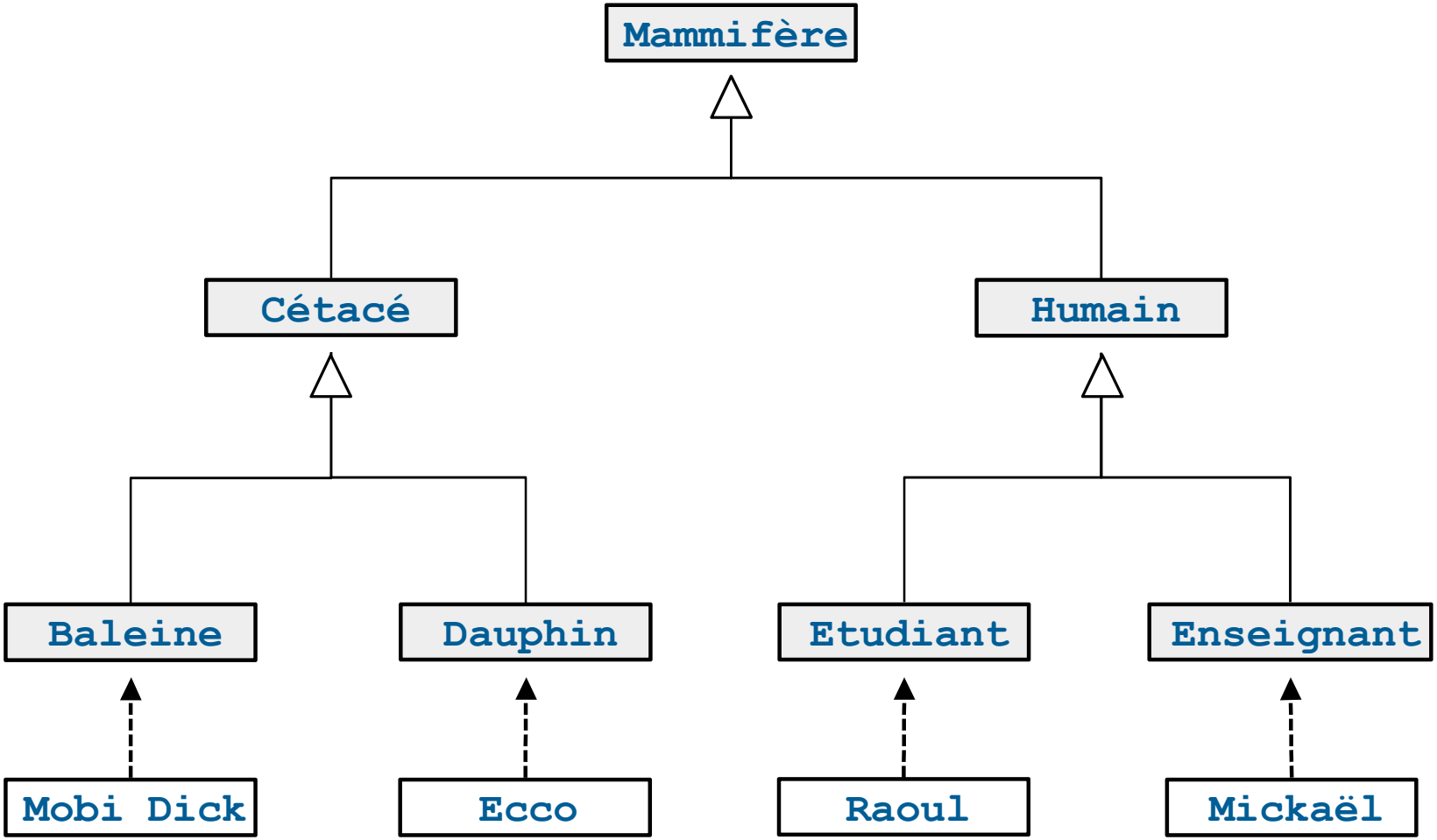
- Generalization expresses an **"is-a"** relationship between a class and its superclass.



- Inheritance allows
  - to **generalize** in the sense of abstraction
  - to **specialize** in the sense of refinement

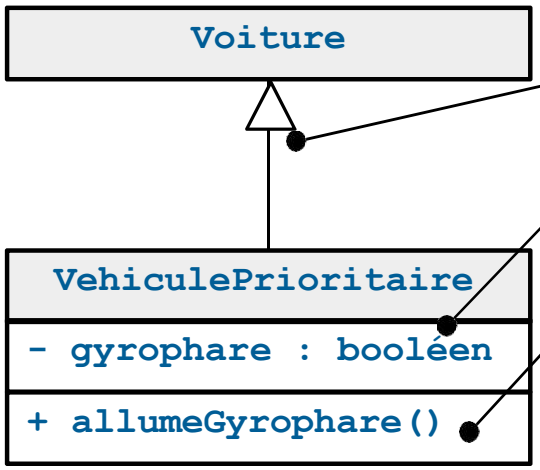
Inheritance Example

- Example: Species



Inheritance and Java

- Simple Inheritance
  - A class can inherit from only one other class
  - In some other languages (e.g., C++), multiple inheritance is possible
  - Use of the keyword **extends** after the class name



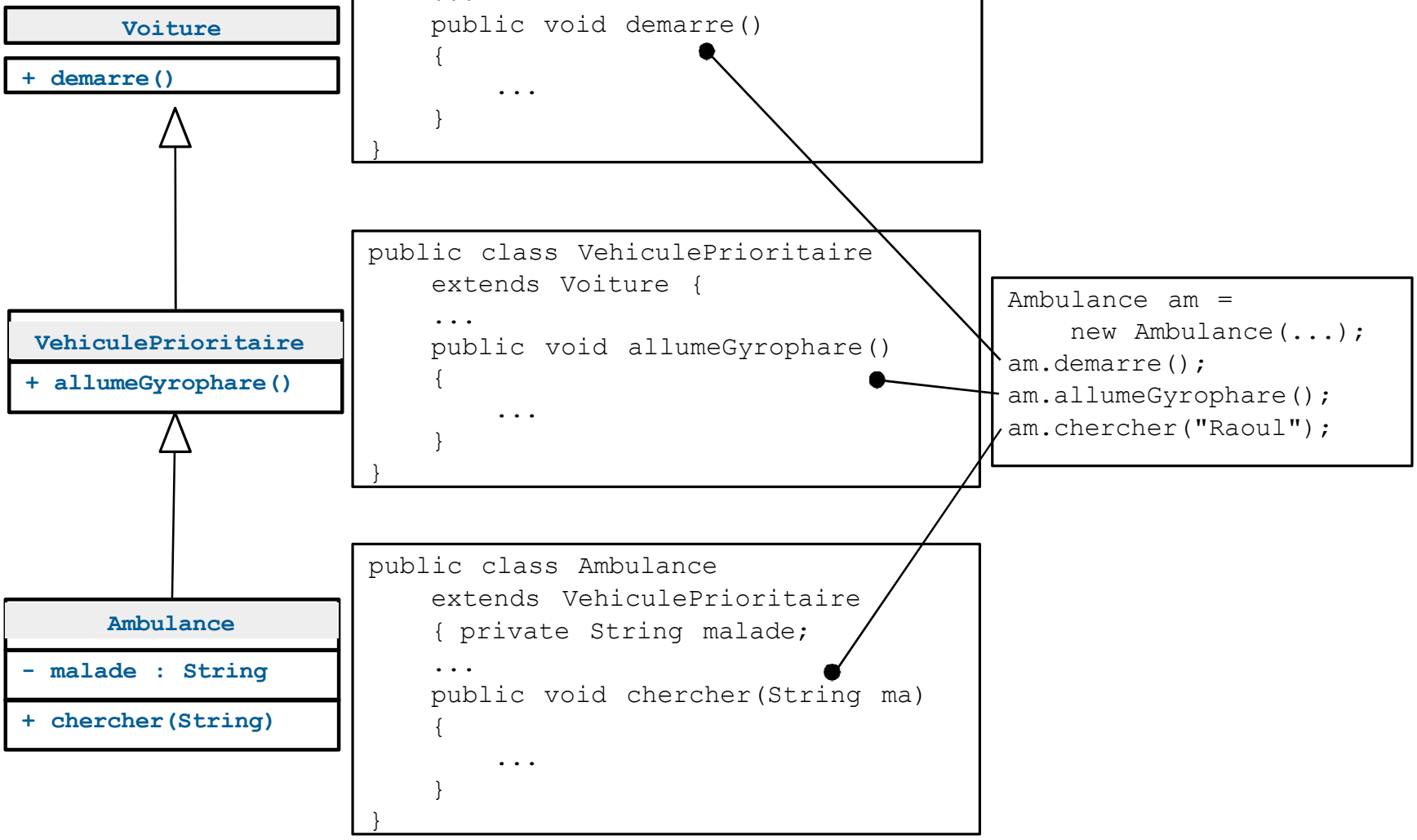

```
public class VehiculePrioritaire extends Voiture {

   private boolean gyrophare;
   ...

   public void allumeGyrophare() {
      gyrophare = true;
   }
   ...
}
```

Do not try to inherit from multiple classes (extends Voiture, Sante, ...) as it does not work

Multilevel inheritance



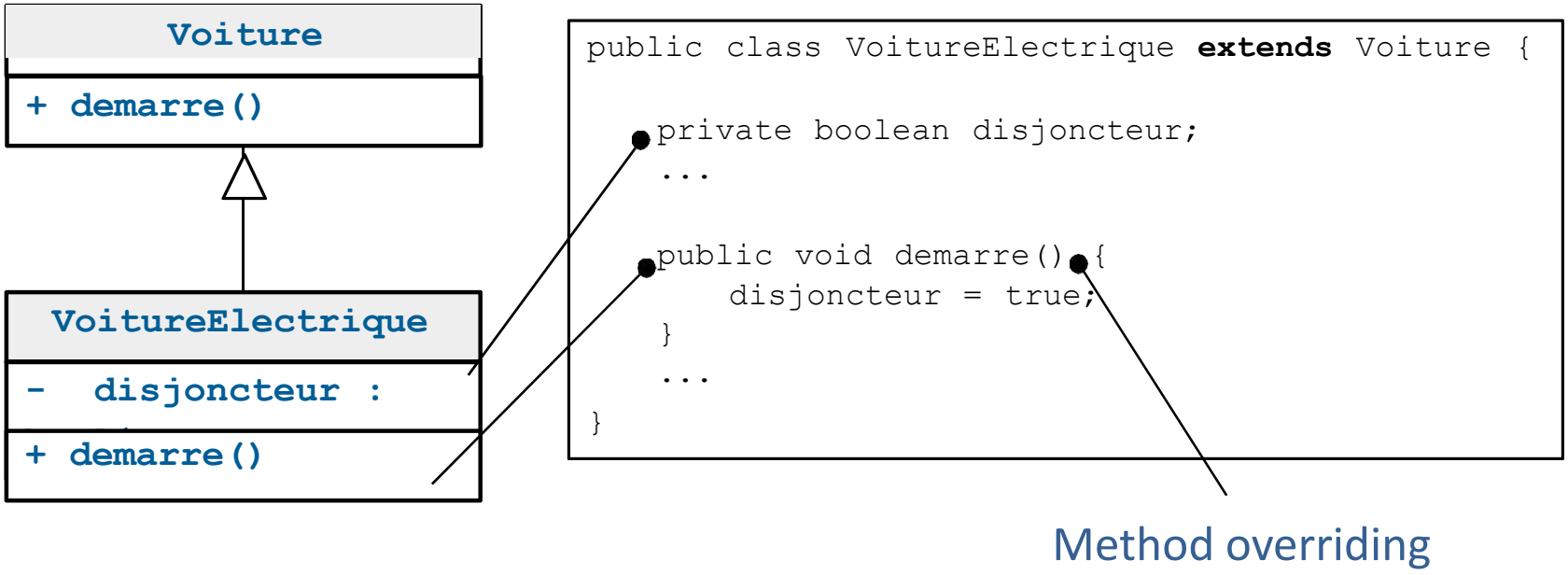
Overloading and overriding

- Inheritance
  - A subclass can add new attributes and/or methods to those it inherits (overloading is part of it).
  - A subclass can override (overriding) the methods it inherits and provide specific implementations for them.
- Reminder of *overloading*: the possibility to define methods with the same name but different arguments (parameters and return value).
- *Overriding*: when the subclass defines a method whose name, parameters, and return type are identical.

Overloaded methods can have different return types as long as they have different arguments

Overloading and overriding

- An electric car is a car whose startup operation is different
  - An electric car responds to the same messages as the Car.
  - We start an electric car by activating a circuit breaker.



Overloading and overriding

```
public class Voiture {
    ...
    public void demarre() {
        ...
    }
}
```

Do not confuse overloading and overriding. In the case of overloading, the subclass adds methods, while overriding “specializes” existing methods

Overriding

Overloading

```
public class VoitureElectrique
    extends Voiture {
    ...
    public void demarre() {
        ...
    }
}
```

```
public class VehiculePrioritaire
    extends Voiture {
    ...
    public void demarre(int code) {
        ...
    }
}
```

VoitureElectrique has “at most" one method less than *VehiculePrioritaire*

VehiculePrioritaire has "at most" one more method than *VoitureElectrique*

Overriding with reuse

- Interest
  - Overriding a method hides the code of the inherited method.
  - Reuse the code of the inherited method using the **super** keyword.
  - **super** allows explicit designation of an instance of a class whose type is that of the parent class.
  - Access to attributes and methods overridden by the current class but that one wishes to use.

```
super.nomSuperClasseMethodeAppelee(...);
```

- Example of *Voiture*: limitations to resolve
  - The call to the *demarre* method of *VoitureElectrique* only modifies the *disjoncteur* attribute.

Overriding with reuse

- Example: method reuse

```
public class Voiture {

    private boolean estDemarree;
    ...

    public void demarre() {
        estDemarree = true;
    }
}
```

Updating the attribute *estDemarree*

```
public class VoitureElectrique extends Voiture {

    private boolean disjoncteur;
    ...

    public void demarre() {
        disjoncteur = true;
        super.demarre();
    }
    ...
}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création
        VoitureElectrique laRochelle =
            new VoitureElectrique(...);
        laRochelle.demarre();
    }
}
```

Sending a message by calling *demarre*

The position of super does not matter here.

Usage of constructors: continuation

• Possibility, like methods, to reuse the code of the superclass constructors.

• Explicit call to a constructor of the parent class inside a constructor of the child class.

• Uses the keyword **super**

The call to the superclass constructor must absolutely be made as the first statement

```
super(paramètres du constructeur);
```

• The implicit call to a constructor of the superclass is made when there is no explicit call. Java implicitly inserts the **super()** call.

Usage of constructors: continuation

- Exemple : constructors *voiture*

```
public class Voiture {
    ...

    public Voiture() {
        this(7, new Galerie());
    }

    public Voiture(int p) {
        this(p, new Galerie());
    }

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

The call to the constructor of the super-class must absolutely be made as the first statement

Implementation of the constructor of *VoiturePrioritaire* from *Voiture*.

```
public class VoiturePrioritaire
    extends Voiture {

    private boolean gyrophare;

    public VoiturePrioritaire(int p, Galerie g) {
        super(p, null);
        this.gyrophare = false;
}}
```

Usage of constructors: continuation

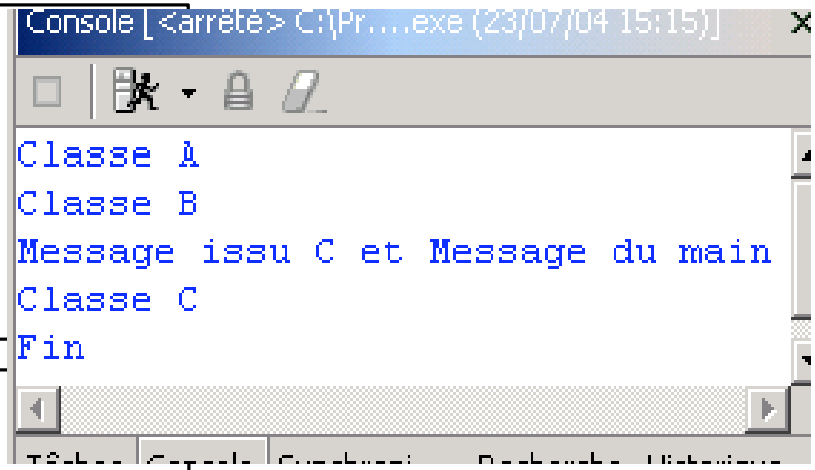
- Example: constructor chaining

```
public class A
    { public A()
      {
          System.out.println("Classe  A");
      }
}
```

```
public class B extends A {
     public B(String message)
     {
          super(); // Appel implicite
          System.out.println("Classe  B");
          System.out.println(message);
     }
}
```

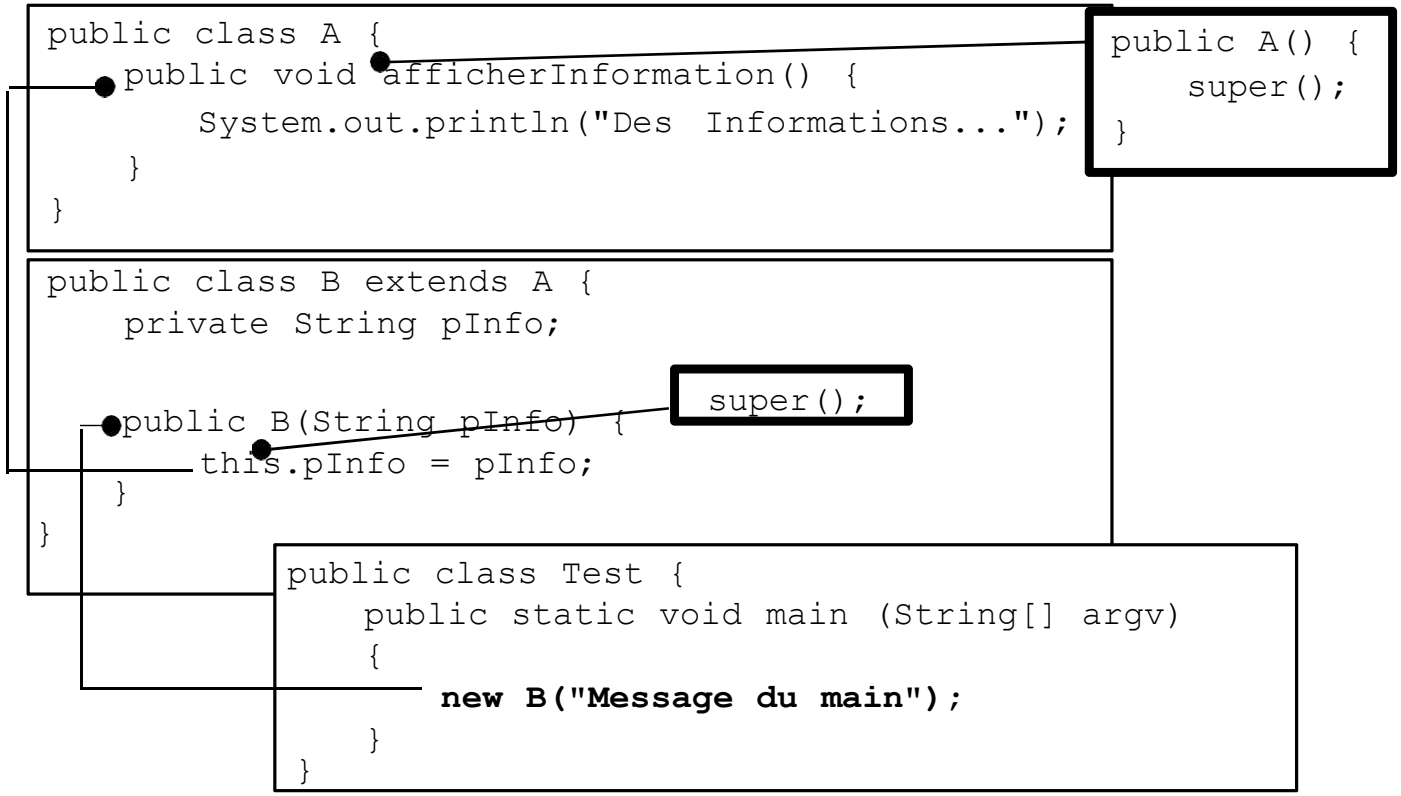
```
public class C extends B {
     public C(String debut)
     {
          super("Message issu C" + debut);
     }  Sys
}       Syst
```

```
public class Test {
    public static void main (String[] argv)  {
        new C(" et Message du main");
    }
}
```



Usage of constructors: continuation

- Reminder: if a class does not explicitly define a constructor, it has a default constructor
  - Without parameters
  - Which does nothing
  - Useless if another constructor is explicitly defined



Usage of constructors: continuation

- Example: explicit constructor

```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```

Explicit constructors disable the default constructor

Error: there is no constructor without parameters in *Voiture*

```
public class VoiturePrioritaire
    extends Voiture {

    private boolean gyrophare;

    public VoiturePrioritaire(int p, Galerie g) {
        this.gyrophare = false;
    }
}
```

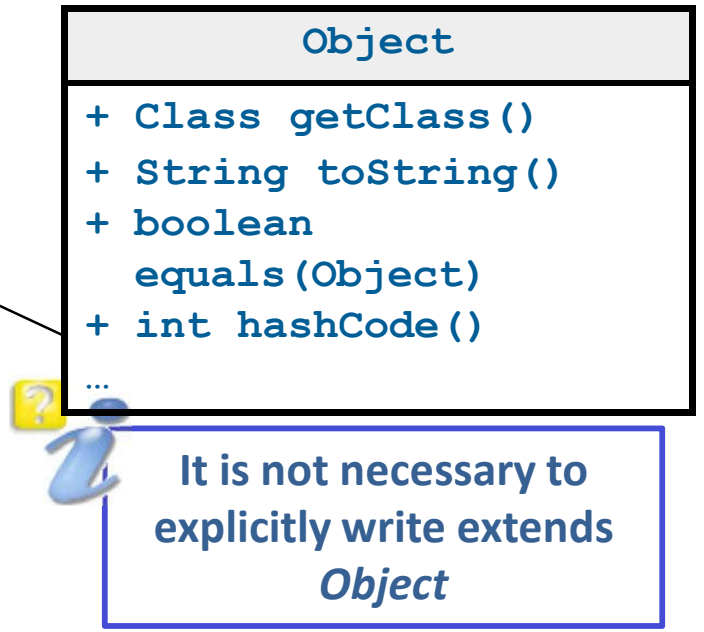
```
super();
```

The Object class

- The **Object** class is the highest-level class in the inheritance hierarchy.
  - Any class other than **Object** has a superclass.
  - Every class inherits directly or indirectly from the **Object** class.
  - A class that does not define an **extends** clause inherits from the **Object** class.

```
public class Voiture extends Object {
    ...

    public Voiture(int p, Galerie g) {
        puissance = p;
        moteur = new Moteur(puissance);
        galerie = g;
        ...
    }
    ...
}
```



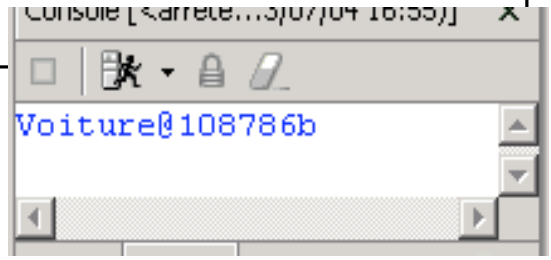
The Object class

Before Overriding

```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }
}
```

```
public class Test {
    public static void main (String[] argv) {
        Voiture maVoiture = new Voiture(5);
        System.out.println(maVoiture);
    }
```

```
public Sring toString() {
    return (this.getClass().getName() +
        "@" + this.hashCode());
}
```

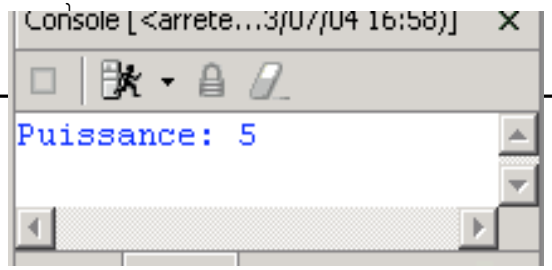


After Overriding

```
public class Voiture {
    ...
    public Voiture(int p) {
        this(p, new Galerie());
    }

    public String toString() {
        return("Puissance:" + p)
    }
}
```

```
.ln(maVoiture.toString());
public class Test {
    public static void main (String[] argv) {
        Voiture maVoiture = new Voiture(5);
        System.out.println(maVoiture);
}
```

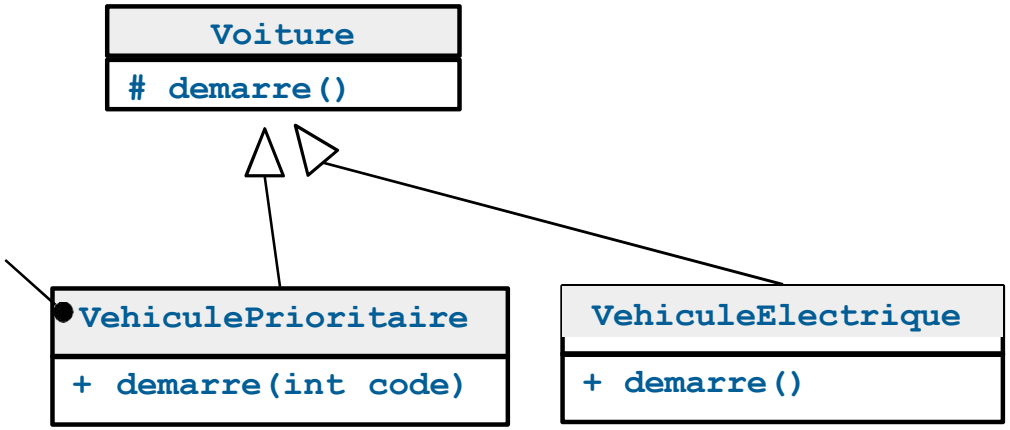


Overriding the *toString()* method

Access rights to attributes and methods

- Example of Voiture: the limitations to resolve
  - *demarre()* is available in the *VehiculePrioritaire* class That means it can start without entering the code!!!
  - Solution: protect the *demarre()* method in the *Voiture* class
- Implementation
  - Use the **protected** keyword before the definition of methods and/or attributes
  - Members are accessible within the class where they are defined, in all its subclasses

The method *demarre()* is not accessible publicly in an object of class *VehiculePrioritaire*



Access rights to attributes and methods

- Example: access to methods

```
public class Voiture {

    private boolean estDemarree;
    ...

    protected void demarre() {
        estDemarree = true;
    }
}
```

```
public class VoiturePrioritaire
    extends Voiture {

    private int codeVoiture;

    public void demarre(int code) {
        if (codeVoiture == code) {
            super.demarre();
        };
    }
}
```

```
public class TestMaVoiture {

    public static void main (String[] argv) {
        // Déclaration puis création de maVoiture
        VehiculeElectrique laRochelle = new VehiculeElectrique(...);
        larochelle.demarre(); // Appel le demarre de VehiculeElectrique

        VehiculePrioritaire pompier = new VehiculePrioritaire(...);
        pompier.demarre(1234); // Appel le demarre VoiturePrioritaire
        pompier.demarre(); // Erreur puisque demarre n'est pas public
    }}
```

Final methods and classes

- Definition
  - Usage of the **final** keyword
  - Method: prevent potential overriding of a method

```
public final void demarre();
```

  - Class: prevent any specialization or inheritance of the concerned class

```
public final class VoitureElectrique extends
Voiture {
   ...
}
```

The String class, for example, is final