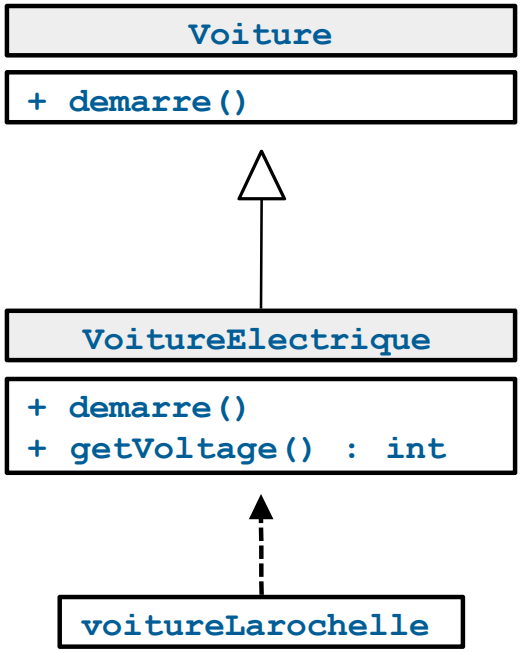
Object-Oriented Programming: Application to the Java Language

Polymorphism

Definition of polymorphism

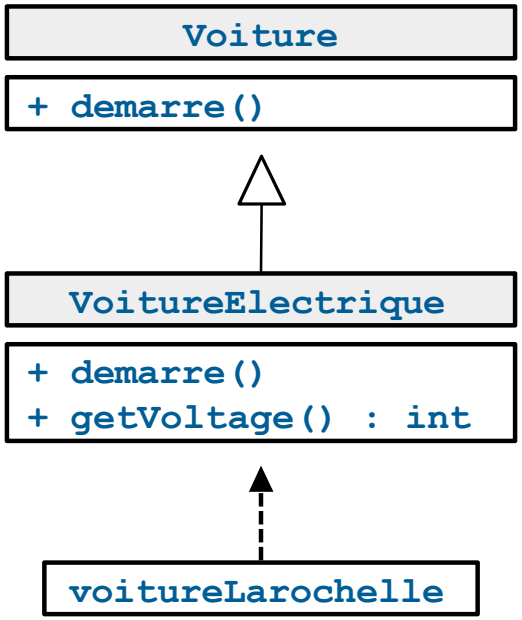
- Definition
  - An object-oriented language is said to be polymorphic if it allows an object to be perceived as an instance of various classes, depending on the needs.
  - **A class B that inherits from class A can be seen as a subtype of the type defined by class A.**



- Reminder
  - *voitureLarochelle* is an instance of the class *VoitureElectrique*
- But also
  - *voitureLarochelle* is an instance of the class *Voiture*

Polymorphism and Java: Upcasting

- Java is polymorphic
  - A reference of the class *Voiture* can be assigned a value that is a reference to an object of the class *VoitureElectrique*
  - This is called upcasting
  - A reference of a given type, say A, can be assigned a value that corresponds to a reference to an object whose actual type is any direct or indirect subclass of A



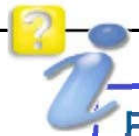
Object of a type that is a direct subclass of *Voiture*

```
public class Test {
    public static void main (String[] argv)  {

        Voiture voitureLarochelle =
            new VoitureElectrique(...);
    }
}
```

Polymorphism and Java: Upcasting

- At compilation
  - When an object is "upcast", it is seen by the compiler as an object of the type of the reference used to designate it.
  - Its functionalities are then restricted to those offered by the class of the reference type.

```
public class Test {
    public static void main (String[] argv)  {

        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

        // Utilisation d'une méthode de la classe Voiture
        voitureLarochelle.demarre();

        // Utilisation d'une méthode de la classe VoitureElectrique
        System.out.println(voitureLarochelle.getVoltage()); // Erreur
    }
}
```



Examine the type of the reference

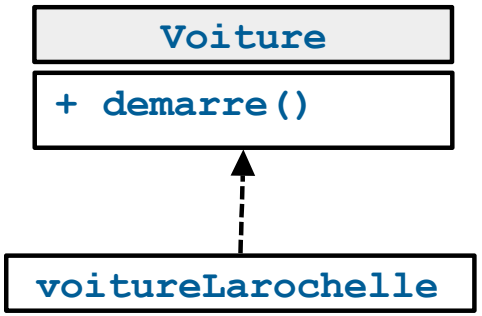
The method *getVoltage*() is not available in the *Voiture* class!!!

Polymorphism and Java: Upcasting

- Example: upcasting

```
public class Test {
    public static void main (String[] argv)  {
        // Déclaration et création d'un objet Voiture
        Voiture voitureLarochelle = new VoitureElectrique(...);

        // Utilisation d'une méthode de la classe Voiture
        voitureLarochelle.demarre();

        // Utilisation d'une méthode de la classe VoitureElectrique
        System.out.println(voitureLarochelle.getVoltage());
    }
}
```

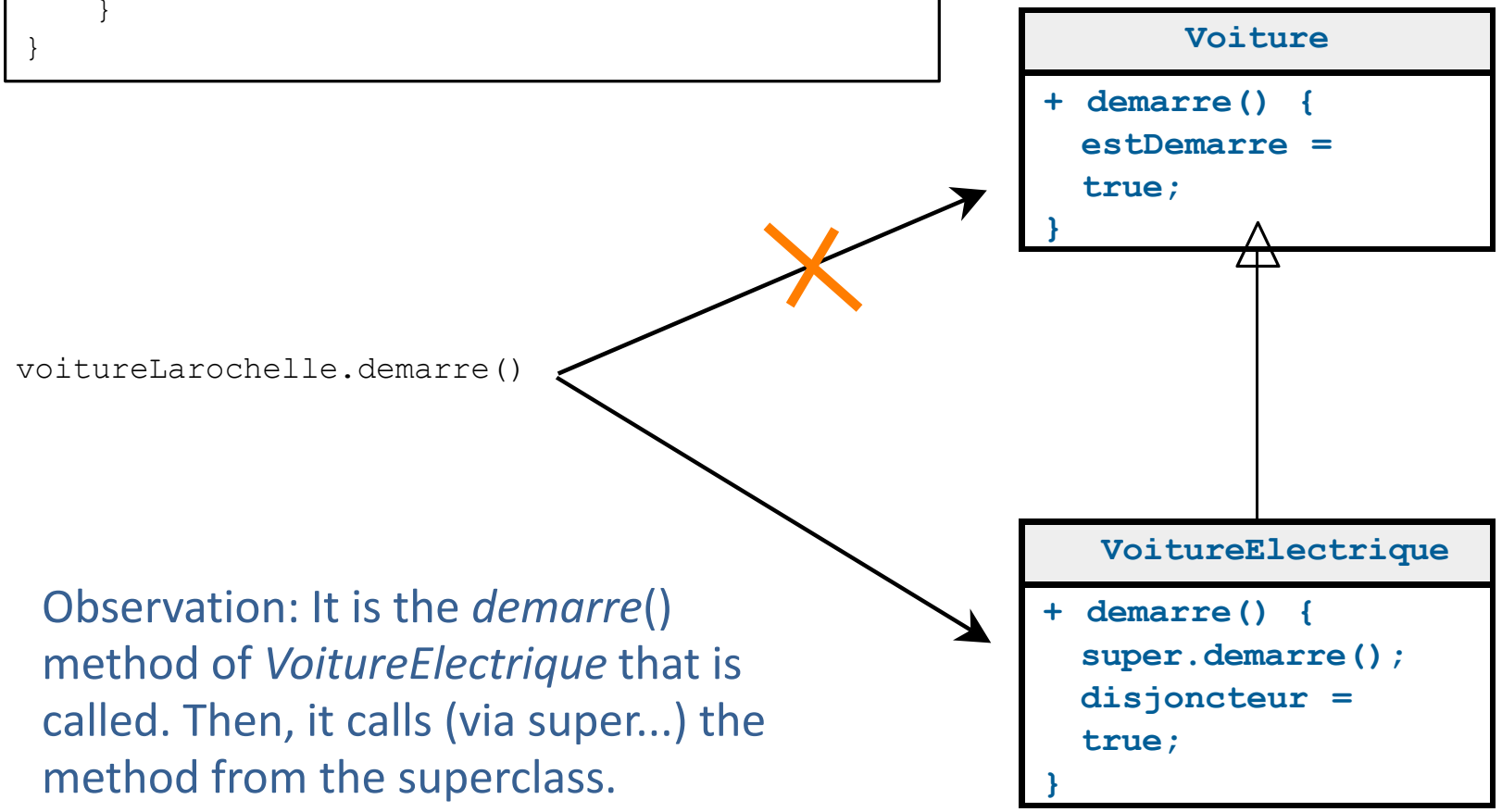
Note: Which code will actually be executed when the *demarre*() message is sent to *voitureLarochelle*??



Polymorphism and Java: dynamic binding

```
public class Test {
    public static void main (String[] argv)
        { Voiture voitureLarochelle =
            new VoitureElectrique(...);

        voitureLarochelle.demarre();
    }
}
```

The object *voitureLarochelle* initializes the attributes of the *VoitureElectrique* class.



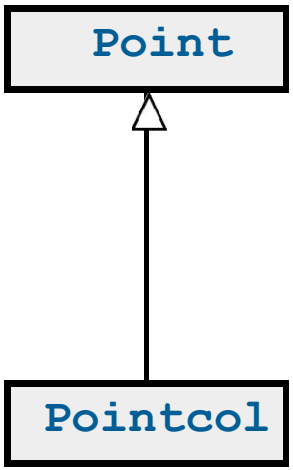
Observation: It is the *demarre*() method of *VoitureElectrique* that is called. Then, it calls (via super...) the method from the superclass.

Polymorphism and Java: dynamic binding

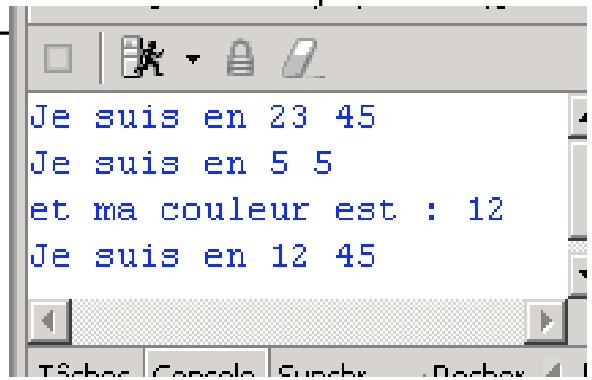
- Example: dynamic binding

```
public class Point {
     private int x,y;
     public Point(int x, int y) { this.x = x; this.y = y; }
     public void deplace(int dx, int dy) { x += dx; y+=dy; }
     public void affiche() { System.out.println("Je suis en "+ x + " " + y);}
}
```

```
public class Pointcol extends Point {
     private byte couleur;
     public Pointcol(int x, int y, byte couleur) {
          super(x,y);
          this.couleur = couleur;
     }
     public void affiche() {
          super.affiche();
          System.out.println("et ma couleur est : " + couleur);
     }
}
```



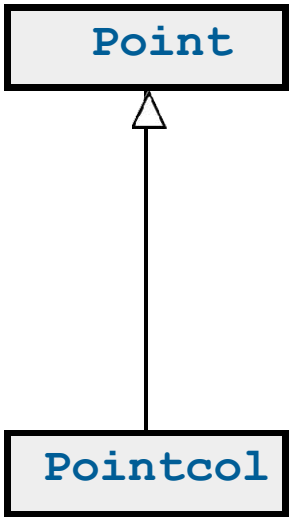

```
public class Test {
     public static void main (String[] argv)        {
          Point p = new Point(23,45);
          p.affiche();
          Pointcol pc = new Pointcol(5,5,(byte)12);
          p = pc;
          p.affiche();
          p = new Point(12,45);
          p.affiche();
     }

}
```



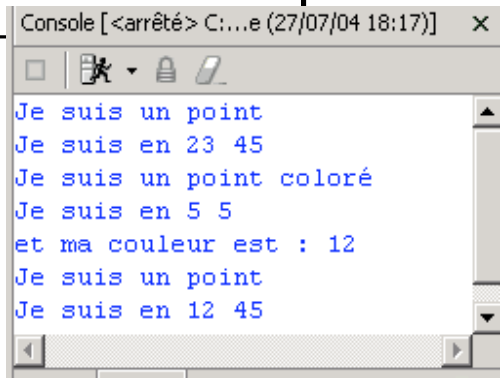
Polymorphism and Java: dynamic binding

```
public class Point
     { private int
     x,y;
     public Point(int x, int y) { this.x = x; this.y = y; }
     public void deplace(int dx, int dy) { x += dx; y+=dy;
     } public void affiche() {
          this.identifie();
          System.out.println("Je suis en "+ x + " " + y);
     }
     public void identifie() {System.out.println("Je suis un point");}
}
```

```
public class Pointcol extends Point
     { private byte couleur;
     public Pointcol(int x, int y, byte couleur)
     {...} public void affiche() {
          super.affiche();
          System.out.println("et ma couleur est : " + couleur);
     }
     public void identifie() {System.out.println("Je suis un point coloré");}
}
```




```
public class Test {
     public static void main (String[]
          argv)
          { Point p = new Point(23,45);
          p.affiche();
          Pointcol pc = new Pointcol(5,5,(byte)12);
          p = pc;
          p.affiche();
          p = new
          Point(12,45);
          p.affiche();
}    }
```

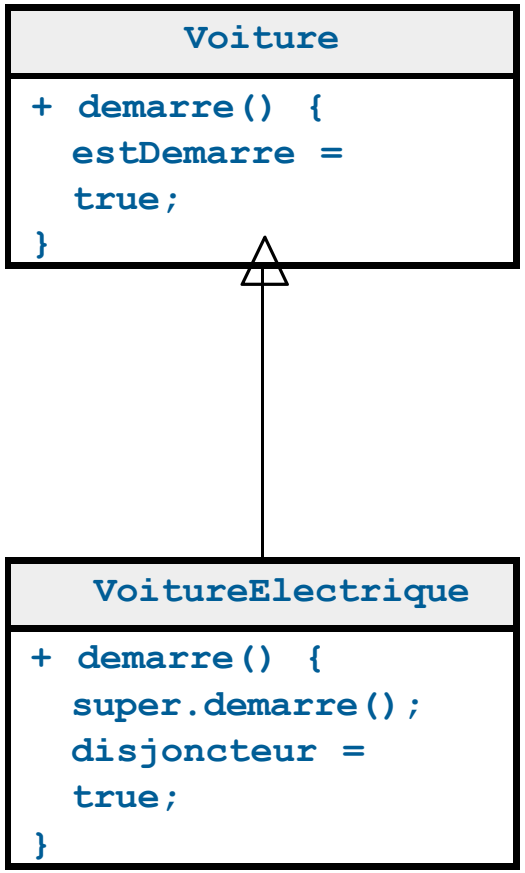


Polymorphism and Java: dynamic binding

- At runtime
  - When a method of an object is accessed through an "upcasted" reference, it is the method as defined in the actual class of the object that is invoked and executed.
  - The method to be **executed** is determined at runtime, not at **compilation**.
  - This is referred to as **late binding**, **dynamic binding**, or **run-time binding.**

Polymorphism and Java: summary

```
public class Test {
    public static void main (String[] argv) {
        Voiture maVoit = new VoitureElectrique(...);
        maVoit.demarre();
    }
}
```



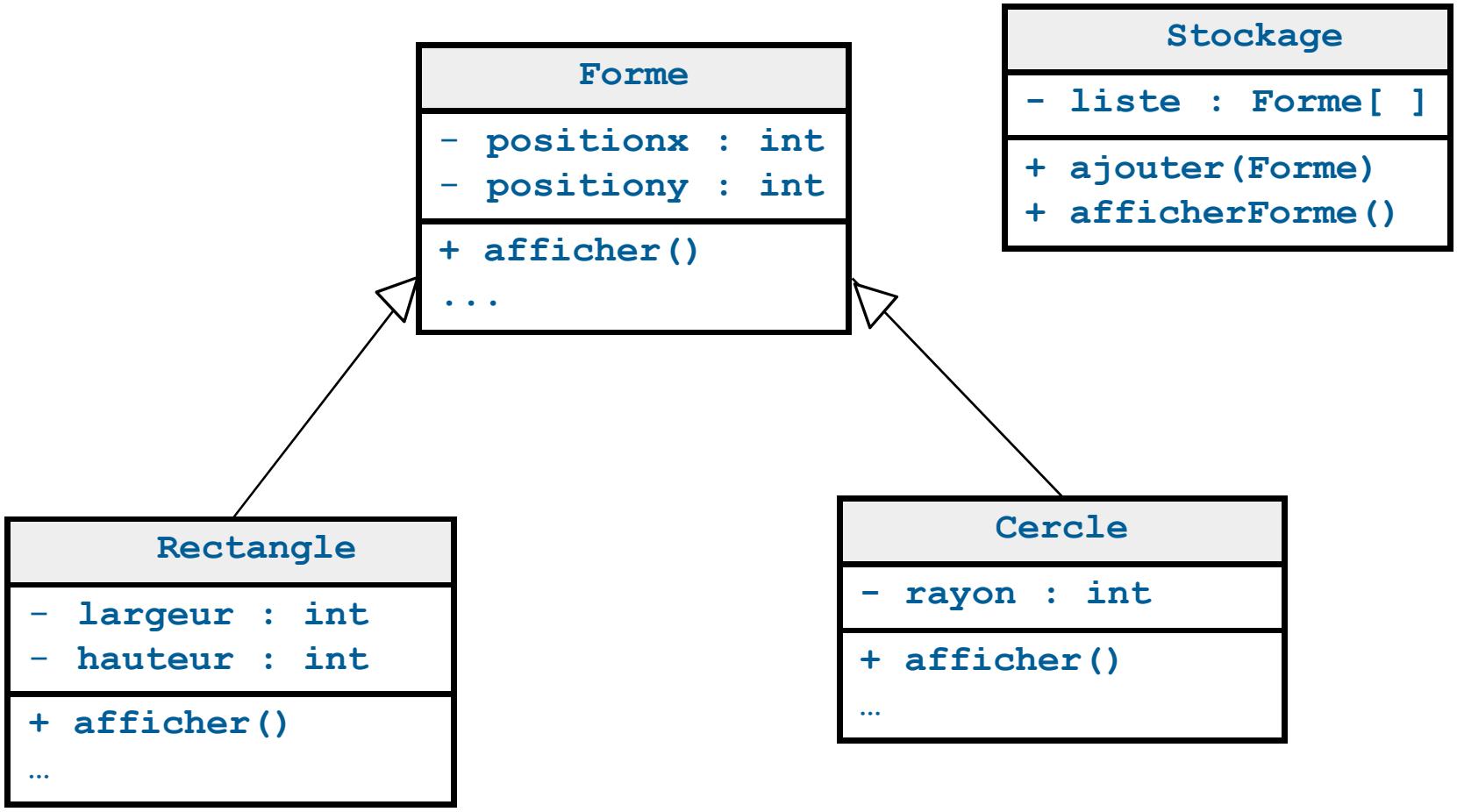
- **Upcasting** (compilation)
  - A variable *maVoit* is declared as a reference to an object of the *Voiture* class.
  - An object of the *VoitureElectrique* class is created.
  - For the compiler, *maVoit* remains a reference to an object of the *Voiture* class, and it prevents access to methods specific to *VoitureElectrique*.
- **Dynamic binding** (execution)
  - A variable *maVoit* is indeed a reference to an object of the *VoitureElectrique* class.

Polymorphism: okay, but why use it?

- All advantages...
  - No need to distinguish between different cases based on the class of objects.
  - Polymorphism is the third essential feature of an object-oriented language, after data abstraction (encapsulation) and inheritance.
  - Greater ease of code evolution. It is possible to define new functionalities by inheriting new data types from a common base class without needing to modify the code that manipulates the base class.
  - **Faster** development.
  - Greater **simplicity** and **better organization** of code.
  - More easily **extensible** programs.
  - **Easier** maintenance of code.

Polymorphism: a typical example

- Example: geometry
  - Store *Forme* objects of any type (*Rectangle* or *Cercle*) and then display them



Polymorphism: a typical example

- Example (continued): geometry

```
public class Stockage {
    private Forme[] liste;
    private int taille;
    private int i;

    public Stockage(int taille) {
        this.taille = taille;
        liste = new Forme[this.taille];
        i = 0;
    }

    public void ajouter(Forme f) {
        if (i < taille) {
            liste[i] = f;
            i++;
        }
    }

    public void afficherForme() {
        for (int i = 0; i < taille; i
            liste[i].afficher();
        }
    }
}
```

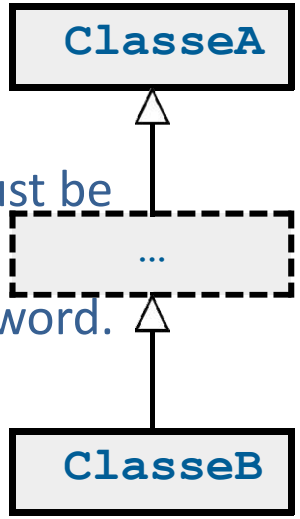
If a new type of *Forme* is defined, the code of the *Stockage* class does not need to be modified

```
public class Test {
    public static void main (String[] argv) {
        Stockage monStock = new Stockage(10);
        monStock.ajouter(new Cercle(...));
        monStock.ajouter(new Rectangle(...));

        Rectangle monRect = new Rectangle(...);
        Forme tonRect = new Rectangle(...);
        monStock.ajouter(monRect);
        monStock.ajouter(tonRect);
    }
}
```

Polymorphism: downcasting

- Purpose
  - Forces an object to “reveal” functionalities hidden by upcasting
  - Explicit type conversion (cast). Already seen with primitive types

```
ClasseA monObj = ...
ClasseB b = (ClasseB)monObj
```



  - For the “cast” to work, the actual type of *monObj* at runtime must be “compatible” with the type *ClasseB*.
  - Compatible: you can test compatibility using the **instanceof** keyword.

```
obj instanceof ClasseB
```

Returns true or false

Polymorphism: downcasting

- Example: downcasting

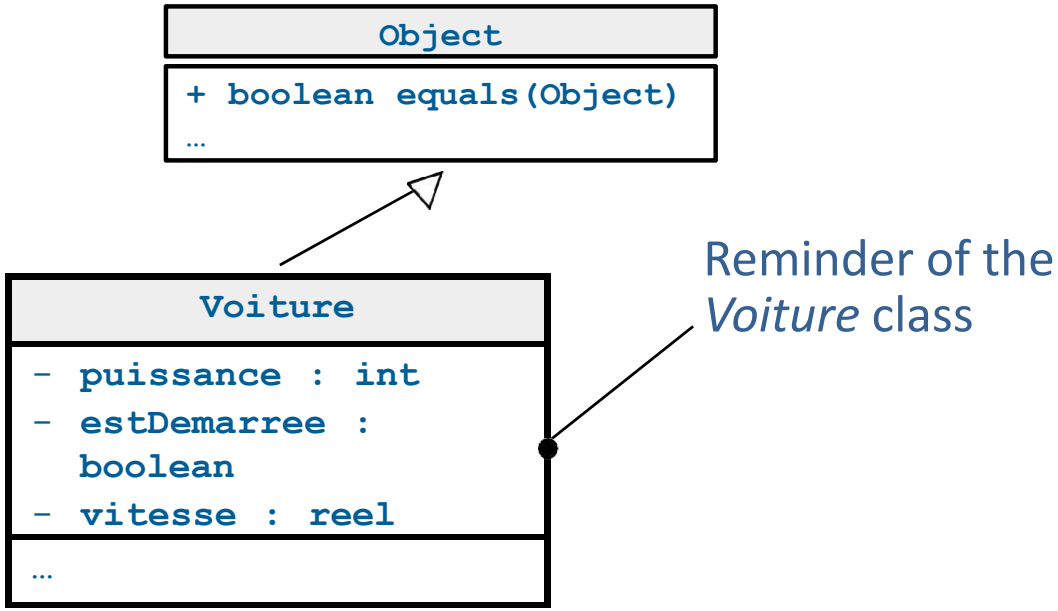
```
public class Test {
    public static void main (String[] argv)
        { Forme maForme = new Rectangle();
        // Je ne peux pas utiliser les méthodes de la classe Rectangle

        // Déclaration d'un objet de type Rectangle
        Rectangle monRectangle;
        if (maForme instanceof Rectangle) {
            monRectangle = (Rectangle)maForme;
            // Utilisation possible des méthodes spécifiques de Rectangle
        }
    }
}
```

Performing the conversion of the object from type *Forme* to type *Rectangle*.

Be careful: if the compatibility is false and the cast is performed, a *ClassCastException* is thrown.

The equals() method

- Two possibilities to compare objects of a class
  - Create an ad-hoc method "*boolean comparer(MaClasse c) {…}*" that compares the attributes
  - Redefine the *"boolean equals(Object o)"* method to maintain compatibility with other Java classes
    - Re-implement the *"boolean equals(Object o)"* method by comparing the attributes (using explicit type conversion)



The equals() method

- Example: overriding the equals method

```
public class Voiture extends Object {
    public boolean equals(Object o) {
        if (!o instanceof Voiture) {
            return false;
        }

        Voiture maVoit = (Voiture)o;
        return this.puissance == maVoit.puissance && this.estDemarree ==
            maVoit.estDemarree && this.vitesse == maVoit.vitesse;
    }
    ...
}
```

Overriding the *equals* method of the *Object* class

Same argument values

```
public class Test {
    public static void main (String[] argv)
        { Voiture maVoit = new Voiture(...);
        VoitureElectrique maVoitele = new VoitureElectrique(...);

        maVoit.equals(maVoitele); --> TRUE
    }
}
```

Attention: the reference equality == checks if the references are the same, it does not compare the attributes.