3TC36: Object-Oriented Programming in Java

Class inheritance (Part 1)

Dominique Blouin
`dominique.blouin@telecom-paris.fr`
Le February 24, 2026

Learning objectives

- Class inheritance
- Polymorphism
- Dynamic binding
- `Final` keyword

Recap

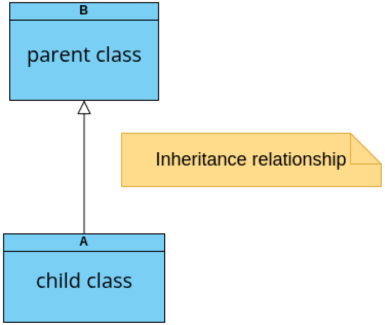
- **Objects** are program entities that communicate by sending **messages**.
- Objects contain values called **attributes**. Among these attributes, there may be **references** to other objects.
  - A reference to an object allows sending it a message.
- For each type of message that an object can receive, the object knows a **method** associated with that type of message.
- This method is a procedure **executed by the object** when it receives the associated type of message.

Recap

- An object type is described by a **class**.
  - This class describes the **attributes**: `name` and value `type`.
  - It also describes the **methods** used to respond to messages.
- The programmer can create objects from the class. This is the **instantiation** process.
- We say that objects are **instances** of the **class** or that objects **belong** to the **class**.

Class inheritance

- A class `A` can declare that it **inherits** from a class `B`.
  - Class `A` is called a **child** class or **subclass** of class `B`.
  - Class `B` is called a **parent** class or **superclass** of class `A`.
- Meaning: The child class **inherits** the **declarations** made in the parent class.



Class inheritance (continued)

- Enrichment:
    - When declaring that a child class inherits from a parent class, it is possible to **enrich** the child class with **additional attributes and methods**.
    - This is called **enrichment**, or **modular extension**.
- Redefinition / Overriding:
    - It is also possible to **redefine inherited methods** by providing a **new implementation** for these methods.
    - This is called **redefinition** or **substitution**.
- Enrichment and redefinition are not mutually exclusive (i.e., you can do both at the same time in the same child class).

Example of inheritance and method redefinition (1/2)

```java
public class Item { // An item in a store

    private double netPrice;

    public double getNetPrice() {
        return netPrice;
    }

    public double getVAT() { // VAT = Value Added Tax
        return 0.185 * netPrice; // 18.5%
    }

    public double getATIPrice() { // ATI = All Taxes Included
        return netPrice + getVAT();
    }
    // ...
}
```

Example of inheritance and method redefinition (2/2)

- The `extends` keyword declares the inheritance relationship:

```
public class LuxuryItem extends Item {

    @Override
    public double getVAT() { // Method overriding
        return 0.33 * getNetPrice(); // 33% tax rate
    }
    // ...
}
```

- `@Override` is an annotation. It serves as an indication to the compiler that the **method is being overridden.**
  - The compiler will check that this is the case.
- The `@Override` annotation is not mandatory, but it is **strongly recommended**

Enriching a class: Example

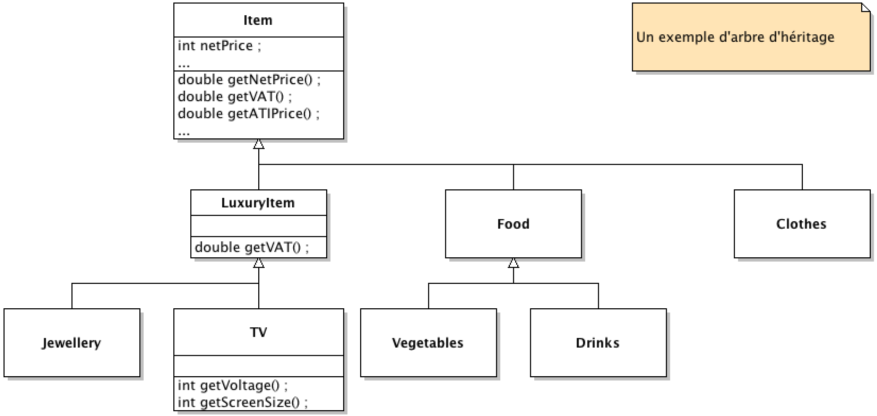
- A particular luxury item:

```
public class Television extends LuxuryItem {

    private int voltage;
    private int screenSize;

    public int getVoltage() {
        return voltage;
    }

    public int getScreenSize() {
        return screenSize;
    }    // ...
}
```

- The `voltage` and `screenSize` attributes enrich the class with additional data.

Inheritance tree

- The inheritance relationship between classes is represented by a tree called the **inheritance tree.**



Root of the inheritance tree

- If a class does not specify a parent class, it inherits by default from the `Object` class in the `java.lang` package.
- Search: JAVA SE Object.
- In Java, each class can inherit **from only one** class.
- This is called **single inheritance**.

Visibility of elements in the parent class

- **Four types of visibility (reminder):**
  - `public`, `private`, `package`, and `protected`.
- Anything declared `public` in the parent class is accessible from **all the classes**.
- Anything declared `private` in the parent class is only accessible **from the class itself**, not from the child classes or other classes in the same package as the parent class.
- Anything declared `package` (default value) in the parent class is only accessible from **classes in the same package** as the parent class, including child classes in the same package, but **not from classes in other packages**.
- Anything declared `protected` in the parent class is accessible **from child classes**, regardless of their package, and also from classes in the same package, regardless of inheritance.

Data encapsulation and use of the `protected` visibility

- An attribute declared `protected` does not respect the principle of encapsulation.
- In fact, an object of a child class or a class in the same package can directly **modify the attribute** without any **control**.
- Therefore, it is preferable to use `private` visibility and use accessors from the child class, just like in other classes.
- This is why the `protected` keyword will only be used for methods for which we want to restrict usage to child classes or classes in the same package.

Example of visibility of elements in the parent class

- Accessors allow access to elements with `private` visibility:

```java
public class Item {

   // Do not use protected
    private double netPrice;

    public double getNetPrice() {
        return netPrice;
    }
    // ...
}
```

```java
public class LuxuryItem extends Item {

    @Override
    public double getVAT() {

        // netPrice cannot be used directly
        return 0.33 * getNetPrice(); // 33%
    }
    // ...
}
```

Inheritance of constructors

- If the parent class has **constructors**, the constructors of the child class must **call** one of the constructors of the parent class.
- The call to the parent class constructor must be the **first instruction** in the child class constructor.
- This call is made using the `super` keyword followed by the parameters in parentheses.
- The compiler is strict about this.

Example with the `Point` class

```java
public class Point {

    private int xCoord;
    private int yCoord;

    public Point(int xCoord,
                 int yCoord) {

        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
}
```

```java
public class ColouredPoint extends Point {

    private Color color;

    public ColouredPoint(int xCoord,
                         int yCoord,
                         Color color) {

        // It must be the first instruction
        super(xCoord, yCoord);
        this.color = color;
    }
}
```

Inheritance of methods and polymorphism

- Consider a `Shape` class that models shapes located in a plane.
  - A shape is characterized by its coordinates in the plane:

```java
public class Shape {

    private int xCoord;
    private int yCoord;

    public Shape(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
    // // -> continues on the right
```

```java
    public int getXCoord() {
        return xCoord;
    }

    public int getYCoord() {
        return yCoord;
    }
}
```

A rectangular shape

- A rectangle is a particular shape with height and width. It can be modelled by a `Rectangle` class:

```java
public class Rectangle extends Shape {

    private int width;
    private int height;

    public Rectangle(int xCoord, int yCoord,
        int width, int height) {

        super(xCoord, yCoord);
        this.width  = width;
        this.height = height;

    }    // -> continues on the right
```

```java
    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }
}
```

Other geometric shapes... (square)

- A **square** is a type of rectangle with equal height and width.

```
public class Square extends Rectangle {

    public Square(int xCoord, int yCoord, int width) {

        super(xCoord, yCoord, width, width);

    }
}
```

Other geometric shapes... (circle)

- A **circle** is a shape with a **radius** attribute:

```
public class Circle extends Shape {

   private int radius;

   public Circle(int xCoord, int yCoord, int radius) {

      super(xCoord, yCoord);
      this.radius = radius;

   }

   public int getRadius() {
      return radius;
   }
}
```

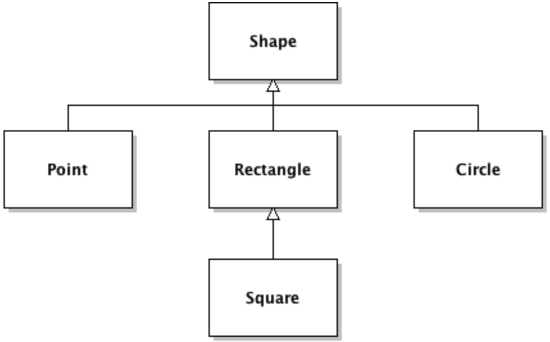
Other geometric shapes... (point)

- A **point** is a shape that only requires coordinates:

```
public class Point extends Shape {

   public Point(int xCoord, int yCoord) {

      super(xCoord, yCoord);

   }
}
```

Inheritance tree

- Technically, an object of the `Square` class possesses the three types: `Square`, `Rectangle`, and `Shape`.
  - A square object is also a *rectangle* and is also a *shape*.
- Similarly, an object of the `Rectangle` class possesses the two types: `Rectangle` and `Shape`, but does not possess the type `Square`.
- Objects of the `Point`, `Square`, `Rectangle`, and `Circle` classes all have the type `Shape`.



Definition of polymorphism

- An object is always an instance of a class. Thus, if we write:

```java
Square square = new Square(10, 10, 100);
```

- The referenced object is an instance of the `Square` class.
- But this object has **three types**: `Square`, `Rectangle`, and `Shape`.
- When objects can **have multiple types**, we speak of **polymorphism**.

An example of polymorphism: Printing shapes to the console

```java
public class Shape {

    private int xCoord;
    private int yCoord;

    public Shape(int xCoord,
                 int yCoord) {

        this.xCoord = xCoord;
        this.yCoord = yCoord;

    } // -> continues on the right
```

```java
    public int getXCoord() {
        return xCoord;
    }

    public int getYCoord() {
        return yCoord;
    }

    public void print() {
        System.out.print("x = " +
            getXCoord() +
            " y = " + getYCoord());
    }
}
```

Redefining display based on shape type

- In the `Rectangle` class, we use the `super` keyword to call the `print()` method from the parent class:

```java
@Override
public void print() {
    System.out.print("Rectangle: ");
    super.print();
    System.out.print(" width = " + getWidth() + " height = " + getHeight())
          ;
}
```

- This avoids **duplicating the code** for displaying the coordinates of the shape.

Redefining display based on shape type (continued)

- In the `Point` class:

```
@Override
public void print() {
    System.out.print("Point: ");
    super.print();
}
```

- In the `Circle` class:

```
@Override
public void print() {
    System.out.print("Circle: ");
    super.print();
    System.out.print(" radius = " + getRadius());
}
```

Redefining display based on shape type (continued)

- In the `Square` class:

```java
@Override
public void print() {
    System.out.print("Square: ");
    super.print();
    System.out.print(" width = " + getWidth());
}
```

- Does this last implementation work?

A possible solution...

- In the `Shape` class:

```
public void print() {
    printCoordinates();
}

protected void printCoordinates() {
    System.out.print("x = " + getXCoord() + " y = " + getYCoord());
}
```

- In the `Rectangle` class:

```
@Override
public void print() {
    System.out.print("Rectangle: ");
    super.print();
    System.out.print(" width = " + getWidth() + " height = " + getHeight())
        ;
}
```

A possible solution... (continued)

- In the `Square` class:

```
@Override
public void print() {
    System.out.print("Square: ");
    printCoordinates();
    System.out.print(" width = " + getWidth());
}
```

Display a list of elements of various shape types on the console: Polymorphism with `ArrayList<Shape>`

- Here is where polymorphism really shines:

```java
ArrayList<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Square(0, 0, 10));
shapes.add(new Rectangle(0,0,10,20));
shapes.add(new Circle(0,0,10));
shapes.add(new Point(0,0));

// Printing the shapes
for (Shape shape : shapes) {
    shape.print();
}
```

Question: For each shape, which `print()` method gets called?
Answer: The correct method is called for each shape.

Method binding

Two possible strategies:

- **Static** method lookup: The compiler assumes the declared type of the variable `shape` is `Shape` and decides that the `print()` method from the `Shape` class should be executed.
    - The term "**static**" means that the method to be executed is determined **at compile time**.
    - This is called **static binding**.
- **Dynamic** method lookup: The compiler schedules a method lookup based on **the actual class** of the referenced object, which is checked **at runtime**.
    - The term "**dynamic**" means that the method to be executed is only determined during **runtime**.
    - This is called **dynamic binding**.

Java uses **dynamic** method lookup.

- It is a **dynamically-bound** language.

Dynamic method binding

- The method executed when calling `shape.print()` is determined at runtime.
- Java looks at the **actual class** of the object referenced by the `shape` variable and chooses the `print()` method from that class.
- This mechanism is well implemented and has minimal runtime overhead.

```
ArrayList<Shape> shapes = ...;

for (Shape shape : shapes) {
    shape.print();
}
```

- How is this achieved? Search: JAVA SE Object
- Refer to the method in the `Object` class:

```
public final Class<?> getClass()
```

The `toString()` method

- In previous labs, you **overrode** the `toString()` method in your `Robot` class.
- What happens when you call:

```
System.out.print(myRobot);
```

- In the `PrintStream` class, which is the class of the static `System.out` attribute, the print method simply calls the `toString()` method of the `Object` class.
- If you overrode `toString()` in your `Robot` class, dynamic binding then determines that it is the `toString()` method of your `Robot` class that should be called.

The `final` keyword

- Even though dynamic method binding is very efficient, it still has some overhead.
- However, there are cases where it is possible to avoid dynamic method lookup: when the method to execute can be determined **statically** at **compile time**.
- To achieve this, we declare a method as `final`, which means that a subclass is not allowed to **override** this method.
- Since the method will never be overridden, its call can be determined **statically** at compile time without any ambiguity about which method to call.

Example

- Let's add a method to the `Shape` class:

```
public void println() {
    print();
    System.out.println();
}
```

- This method does not need to be overridden in subclasses; it is correct for **all types** of shapes. Why?
- Therefore, for a call `shape.println()`, there is no need for dynamic method lookup because the method that will be executed is known—it will always be the (unique) method in the `Shape` class.
- The programmer knows this, but the compiler has no way of knowing.
  - When compiling the `Shape` class, it cannot know if the `println()` method will b overridden or not because it does not know all the subclasses.

Example

- It is up to the **programmer** to indicate with the `final` keyword that this method will not be overridden:

```
public final void println() {
    print();
    System.out.println();
}
```

- Thanks to this declaration, the compiler **knows** that the method will never be overridden by **any** subclass, as this would generate a compilation **error**.
- A good compiler will avoid performing dynamic method lookup in this case.
- Using the `final` keyword for methods that will never be overridden allows the compiler to perform **optimizations** that improve the program's execution speec

Other uses of the `final` keyword: `final class`

- The `final` keyword can also qualify a class. This is the case with the `String` class in the JDK:

```java
public final class String { /* ... */ }
```

- It is **not possible** to inherit from a class declared as `final`.
  - The methods of a class declared as `final` are implicitly all declared `final` as well.
- Different uses of the `final` keyword allow the compiler to perform optimizations:
  - final class, final method, final attribute, and final class variable.
- Making a class `final` can also **strengthen security**.
  - For example, the `String` class is vital because it is used by the compiler and other important parts of Java.
- It is therefore important **to prevent altering the behavior of methods** in the class, which could be done by subclassing the class.

Other uses of the `final` keyword: Immutability of objects

- The `final` keyword can also qualify the **attributes** of a class.
- Values of these attributes can **only be assigned once**.
- These are **non-mutable** attributes.
  - The position of the shape can only be changed by **instantiating a new object**.
- What happens if the shape is a robot from the simulator?

```
public class Shape {

    private final int xCoord;
    private final int yCoord;

    public Shape(int xCoord, int yCoord) {
        this.xCoord = xCoord;
        this.yCoord = yCoord;
    }
```

```
    public int getXCoord() {
        return xCoord;
    }

    public int getYCoord() {
        return yCoord;
    }
}
```

Other uses of the `final` keyword: Parameters and final variables

- The `final` keyword can also qualify the **parameters** of a method.
- It can also qualify **local variables** within a method.

```
public void doSomething(final int xPar) {
    // ...
    final int number = ... ;
    // ...
}
```

- The compiler can easily determine if the program modifies a variable or parameter.
- The `final` qualification is used to indicate that the programmer wants to ensure the variable or parameter is not modified.
- In both cases, the compiler will check that no statement inadvertently modifies th value of the parameter or variable.

Simple and multiple inheritance

- Inheritance is called **simple** if each class inherits from **at most** one class.
  - Examples: **Java**, SmallTalk, and Ada.
  - In this case, inheritance is represented as a **tree** or a **forest of trees.**
- Inheritance is called **multiple** if a class can inherit from **more** than one class.
  - Examples: C++ and Eiffel.
  - In this case, inheritance is represented as one or more **directed graphs**.