TUTORIALS — Object Oriented - Part 2

version **#1.0.0**

YET ANOTHER KIND OF ASSISTANTS 2026 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.forge.epita.fr

1 Inheritance

A fundamental feature of object-oriented programming is inheritance. Inheritance allows a new class to inherit attributes and methods from an existing class.

In object-oriented programming, inheritance is the process by which an `A` class inherits from what is defined in a `B` class. The `A` class, the one who receive information, can be called the **child class**, and `B` class, the one who provide information, the **parent class** or **superclass**. Shared functionality can be implemented in the superclass, which reduces code duplication. Existing classes can be extended to add new features without modifying the original class.

Encapsulation rules apply, so be aware to the scope wanted for your attributes and methods. For example: if you declare a **private** attribute `length` in a parent class, it will not be available in child class.

An `extends` keyword put in a class declaration means that the class on the left of the keyword inherits from the class on its right.

```java
public class ParentClass {
    protected String inheritedString = "Inheritance is cool!";

    public void printPublic() {
        System.out.println("Hello world!");
    }

    protected void printProtected() {
        System.out.println("Hello subclasses!");
    }

    void printDefault() {
        System.out.println("Hello package!");
    }

    private void printPrivate() {
        System.err.println("Hello myself...");
    }
}
```

```java
public class ChildClass extends ParentClass {

    public void printChild() {
        System.out.println("Hello from Child class!");
    }

    public void run() {
        printPublic(); // Hello world!
        printProtected(); // Hello subclasses!
        printDefault(); // Hello package!
        System.out.println(inheritedString); // Inheritance is cool!
        printChild(); // Hello from Child class!

        printPrivate(); // Error: printPrivate has private access in ParentClass
    }
}
```

Going further...

All *Java* classes inherit from the Object class.

1.1 Super

If the superclass of a class does not define a *no-arg constructor*, the child class' constructor will not compile: that is because no constructor can be implicitly called for the superclass. Worry not! You can call the parent class' constructor with the `super` keyword.

The `super` keyword is a bit special, as it takes arguments. Used in a constructor, it calls the constructor of the superclass with the given arguments. If you do not explicitly call `super` in the child class' constructor, it will try to implicitly call the superclass' no-arg constructor: therefore, if the superclass does not define one, the compiler will issue an error.

```java
public class ParentClass {
    // Implicitly defines a default no-arg constructor
}
```

```java
public class ChildClass extends ParentClass {
    protected String inheritedString;

    public ChildClass(String inheritedString) {
        // Implicit call here to the parent class' no-arg constructor (super())
        this.inheritedString = inheritedString;
    }
    // No need to define a default no-arg constructor, since a constructor is provided
}
```

```java
public class BadGrandChildClass extends ChildClass {
    /* Error: Cannot find any no-arg constructor for ChildClass: cannot define a default
     * no-arg constructor
     */
}
```

```java
public class GoodGrandChildClass extends ChildClass {
    private String childString;
    public GoodGrandChildClass(String inheritedString, String childString) {
        super(inheritedString); // Explicit call to the parent constructor
        this.childString = childString;
    }
}
```

Going further...

It is a good practice to always explicitly call the parent constructor, even when it could be done implicitly through a no-arg constructor.

You can also use the `super` keyword in a method: in this case, it will try to call the overridden superclass' method.

```
public class ParentClass {
    public void myPrint() {
        System.out.println("Hello");
    }
}
```

```
public class ChildClass extends ParentClass {
    public void myPrint() {
        super.myPrint();
        System.out.println("you!");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        var child = new ChildClass();
        child.myPrint();
    }
}
```

Output:

```
Hello
you!
```

2 Polymorphism

In *Java*, methods are chosen at runtime, which means that if you write in your child class a method with the same prototype as one of its superclass, the superclass' method will automatically be overridden. The *JVM* handles this process, known as dynamic method dispatch, by looking at the actual type of the object at runtime to determine which method should be called. Moreover, *Java* is **polymorphic**, which means that an instance of a class can be cast into any of its superclasses.

```
public class ParentClass {
    public void inheritedMethod() {
        System.out.println("Parent method");
    }
}
```

```
public class ChildClass extends ParentClass {
    @Override
    public void inheritedMethod() {
        System.out.println("Child method");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ParentClass parent = new ParentClass();
```

(continues on next page)

(continued from previous page)

```java
        ChildClass child = new ChildClass();

        /* Polymorphism: ParentClass is actually polymorphicChild's static type
         * (compile-time).
         * Its dynamic type (runtime) is ChildClass.
         */
        ParentClass polymorphicChild = new ChildClass();

        parent.inheritedMethod();
        child.inheritedMethod();

        /* Dynamic dispatch: call the method from the dynamic type
         */
        polymorphicChild.inheritedMethod();
    }
}
```

Output:

```
Parent method
Child method
Child method
```

Yet, through the annotation mechanism, the *Java* language provides a way to ensure that the superclass method is overridden correctly by indicating this method as overridden to the compiler, using the `@Override` annotation. This annotation will cause a compilation error if no method is a match for the override.

Tips

This annotation is not mandatory to override a function. However, most IDEs put this annotation automatically. You should keep it and add it to any overridden function for clarity purposes.

3 Runtime type-checking

The `instanceof` keyword checks if an object reference is an instance of a type and returns `true` if the object is an instance of the given type or any of its supertypes. Notice that `instanceof Object` always returns `true` since all *Java* objects are inherited from `Object`, and `instanceof NullType` always returns `false`.

```java
public class ParentClass {
}
```

```java
public class ChildClass extends ParentClass {
}
```

```java
public class Main {
    public static void main(String[] args) {
        ParentClass parent = new ParentClass();
```

(continues on next page)

(continued from previous page)

```
        ChildClass child = new ChildClass();
        ParentClass polymorphicChild = new ChildClass();

        System.out.println(parent instanceof Object); // Always true
        System.out.println(parent instanceof NullType); // Always false

        System.out.println(parent instanceof ParentClass); // true
        System.out.println(parent instanceof ChildClass); // false

        System.out.println(child instanceof ParentClass); // true (inheritance)
        System.out.println(child instanceof ChildClass); // true

        System.out.println(polymorphicChild instanceof ParentClass); // true
        System.out.println(polymorphicChild instanceof ChildClass); /* true (these checks are
                                                                     * performed at runtime,
                                                                     * keep polymorphism in
                                                                     * mind!)
                                                                     */
    }
}
```

In *Java*, classes are also objects. Indeed, there is a class in *Java* called Class, which any class is an instance of. You can retrieve a `Class` instance from an object or from a type name:

- If you want to retrieve it from a type name (called *type literal*), append `.class` to it.
- If you want to retrieve it from an object, you can call the `getClass` method on it (inherited from `Object`).

```
public class Main {
    public static void main(String[] args) {
        System.out.println(String.class); // From type literal

        var myString = new String();
        System.out.println(myString.getClass()); // From instance
    }
}
```

Output:

```
class java.lang.String
class java.lang.String
```

The `Class` class defines methods which allow you to perform runtime type-checks similar to those you did with `instanceof`:

- `isAssignableFrom`: returns `true` if the calling `Class` is the same as (or a superclass of) the one given as argument;
- `isInstance`: returns `true` if the `Object` given as argument is an instance of the calling `Class` or one of its child classes.

```
public class ParentClass {
}
```

```java
public class ChildClass extends ParentClass {
}
```

```java
public class GrandChildClass extends ChildClass {
}
```

```java
public class Main {
    public static void main(String[] args) {
        var myClass = ChildClass.class;
        System.out.println(myClass.isAssignableFrom(GrandChildClass.class)); // true
        System.out.println(myClass.isAssignableFrom(ChildClass.class)); // true
        System.out.println(myClass.isAssignableFrom(ParentClass.class)); // false

        var child = new ChildClass();
        System.out.println(ParentClass.class.isInstance(child)); // true (inheritance)
        System.out.println(ChildClass.class.isInstance(child)); // true
        System.out.println(GrandChildClass.class.isInstance(child)); // false
    }
}
```

> **Going further...**
>
> Notice that `Class#isAssignableFrom` checks for an inheritance relationship opposite to the one checked by `instanceof` or `Class#isInstance`.

4 Casting objects

Java features implicit and explicit casts to reinterpret objects.

The implicit casts are always successful and do not require a check at runtime because they are the result of an upcast (from a subclass to a superclass).

```java
public class Inherited {
}
```

```java
public class Inheritor extends Inherited {
}

public class Main {
    public static void main(String[] args) {
        Inheritor inheritor = new Inheritor();

        Inherited inherited = inheritor; // implicit cast
    }
}
```

At some points, you may need to perform explicit casts. This kind of cast exists mainly to perform downcasting (from a superclass to a subclass). Performing an explicit cast is costly and can generate errors. Indeed, casting to an incompatible type with the object's type will generate a ClassCastException at runtime. Instead, you should always use the *instanceof* operator to check that the operation can be performed.

```java
public class Parent {
    public void hello() {
        System.out.println("Hello from Parent class");
    }
}
```

```java
public class Child extends Parent {
    @Override
    public void hello() {
        System.out.println("Hello from Child class");
    }
}
```

```java
public class OtherChild extends Parent {
    @Override
    public void hello() {
        System.out.println("Hello from OtherChild class");
    }
}
```

```java
public class Main {

    private static Parent randomParentGenerator() {
        int random = new Random().nextInt(2); // A random number
        return random % 2 == 0 ? new Child() : new OtherChild();
    }

    public static void main(String[] args) {
        Parent parent = randomParentGenerator();

        // Fails half of the time in this example
        // Child child = (Child) parent;

        // Instead prefer
        if (parent instanceof Child child) {
            // child is a valid instance of Child that can be used
            child.hello();
        }

        else if (parent instanceof OtherChild otherChild) {
            // otherChild is a valid instance of OtherChild that can be used
            otherChild.hello();
        }

        // On older versions of Java
        if (parent instanceof Child) {
            Child newChild = (Child) parent; // Always valid
            newChild.hello();
        }
    }
}
```

5 Abstract classes

An **abstract class** is a class that is not made to be instantiated, but rather to define methods and attributes common to several other classes, that will inherit from it.

Java provides a keyword to make your abstract class apparent right away: you can declare a class as `abstract`.

```
public abstract class ParentClass {
   void hello() {
      System.out.println("Hello!");
   }
}
```

```
public class ChildClass extends ParentClass {
}
```

```
public class Main {
    public static void main(String[] args) {
        // Error: Cannot be instantiated
        ParentClass parent = new ParentClass();

        // Good: Can be instantiated
        ChildClass child = new ChildClass();

        // Good: polymorphicChild is actually an instance of ChildClass
        // stored in a ParentClass
        ParentClass polymorphicChild = new ChildClass();

        child.hello();
        polymorphicChild.hello();
    }
}
```

```
Hello!
Hello!
```

You can also define **abstract methods** in a class: a method that is not yet implemented, but will be in the child classes. You can do so by adding the `abstract` keyword to a method declaration. Beware: if your class declares any abstract method, then the class itself **must** be declared as `abstract`.

One last thing: if your abstract class declares abstract methods, any non-abstract class inheriting from it **must** implement all of them.

```
public abstract class ParentClass {
    public abstract void abstractPrint();
}
```

```
public class BadParentClass {
    public abstract void abstractPrint(); // Error: Abstract method in non-abstract class
}
```

```java
public class IncompleteChildClass extends ParentClass {
    /* Error: Since IncompleteChildClass does not implement abstractPrint, it must also
     * be marked as abstract.
     */
}
```

```java
public class ChildClass extends ParentClass {
    @Override
    public void abstractPrint() {
        System.out.println("Knowledge only grows through challenge.");
    }
}
```

Being a hero means fighting back even when it seems impossible.