Tutorials — Enum Interface Sealed

version **#1.0.0**

Yet Another Kind of Assistants 2026 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

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

1 Enumerations

In *Java*, an **enum** is a class containing underlying objects. It acts like enumerations in other languages, but you can add constructors, attributes and methods. Enums are created like classes, but with the keyword `enum` instead of `class`. The method `value()` returns the list of declarations of the enumeration.

Here is an example of how it works:

```
public enum Animals {
    DOG (4),
    CAT (4),
    SNAKE (0),
    SPIDER (8);

    private int legs;

    Animals(int legs) {
        this.legs = legs;
    }

    public int numberOfLegs() {
        return legs;
    }
}

public class Main {
    public static void main(String args[]) {
        for (Animals animal : Animals.values()) {
            if (Animals.SNAKE.equals(animal)) {
                System.out.println(animal + ": It has no legs");
            }
            else {
                System.out.println(animal + ": It has " + animal.numberOfLegs() + " legs");
            }
        }
    }
}
```

```
DOG: It has 4 legs
CAT: It has 4 legs
SNAKE: It has no legs
SPIDER: It has 8 legs
```

Note that, obviously, constructors cannot be called outside of the enum initialization block.

This will not work:

```
Animals animal = new Animals(1);
```

2 Interfaces

Interfaces are kinds of abstract classes that do not implement any code: they just define *prototypes*. You can see an interface as a contract that a class has to respect.

To express that a class must fulfill an interface's "contract", we use the `implements` keyword:

```
public interface Drawable {
    void draw();
}
```

```
public abstract class Shape {
    public abstract double area();
}
```

```
public class Square extends Shape implements Drawable {
    public void draw() {
        // Drawable code
    }

    public double area() {
        // Shape code
    }
}
```

Be careful!

A class can inherit from **only one other class**, but it can implement **several** interfaces:

```
class MyClass implements Interface1, Interface2 {
    // ...
}
```

Tips

By default, a basic method declared in an interface is an `abstract` method. Therefore, in the previous example the method `draw`, in the `Drawable` interface, is an abstract method.

3 Sealed classes and interfaces

The possibility to seal classes and interfaces was introduced as a preview in the JDK 15 and improved in the JDK 16, with the intent to give a finer grained control over class inheritance and interface implementation mechanisms.

Through class hierarchy, *Java* allows you to factorize and reuse code in numerous subclasses. Yet, reusing code is not always a goal when developing. Sometimes, it might be useful, for a clarity purpose, to restrict the classes that will be able to inherit from a certain superclass or implement a certain interface.

To be able to do so, some keywords were introduced in the *Java* language: `sealed`, `permits` and `non-sealed`.

Let us say that we want to represent the majors that exist at EPITA by creating an abstract class `Major`. We know that only some specific classes will be valid subclasses of `Major`: Finance is not a valid major name at EPITA for instance. Thus, we would like to restrict the use of our superclass `Major` and specify the classes that are allowed to extend it, by making `Major` a sealed class:

```
public abstract sealed class Major
    permits Image, GISTRE, MTI, TCOM, Sante, GITM, SRS, SIGL, SCIA, /* ... */
{
    /* ... */
}
```

The `permits` keyword must be placed after the `extends` or `implements` clauses of a superclass or interface declaration.

Then you must declare your subclasses either in the same module as your superclass or in the same package.

The permitted subclasses must also declare a modifier:

- `final`: will be detailed in the *Static Final* tutorial. In this case, it prevents any further extension of the subclass
- `sealed`: restricts the extension of the subclass to specific classes
- `non-sealed`: opens anew the possibility to freely extend the subclass.

The purpose of sealed classes is to limit the use of a superclass and at the same time, let it be accessible. Declaring a class with the default visibility mode restrains the classes that are able to extend it to only those present in the same package. At the same time, it conceals the existence of this superclass to all the classes outside the package.

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