TUTORIALS — Static Final

version **#1.0.0**

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

Contents

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

1 Static

The meaning of the `static` keyword provided by *Java* depends on where it is located:

- In a **class**, **method** and **attribute** declaration;
- In an **initializer block** declaration;
- In a **nested class** declaration.

1.1 Class, method and attribute

Each attribute and method you have seen until now were related to an instance of a class. But what if we want some of them to be related to the class itself? In order to have an attribute or a method common to every object of your class, you can use the `static` keyword:

```java
public class Main {
    public static class MyClass {
        public static int myInt = 47;

        public static void myPrint() {
            System.out.println("You mean it's working? For real this time?");
        }
    }

    public static void main(String[] args) {
        MyClass.myPrint();
        System.out.println(MyClass.myInt);
    }
}
```

Output:

```
You mean it's working? For real this time?
47
```

Going further...

Actually, you already have used static methods (`public` **static** `void main`) and attributes (`System.out`: `out` is a static attribute of the `System` class).

There are limitations to this keyword: a static method can neither use non-static attributes nor call non-static methods:

```java
public class Main {
    public static class MyStaticClass {
        private final String myNonStaticString = "Non-static String!";
        private static final String myStaticString = "Static String!";

        public void myNonStaticMethod() {
            System.out.println("Non-static method!");
        }

```

(continues on next page)

(continued from previous page)

```java
        public static void myStaticMethod() {
            System.out.println("Static method!");
        }
        public static void run() {
            System.err.println(myNonStaticString); /* Error: Non-static field
                                                     * myNonStaticString cannot be referenced
                                                     * from a static context
                                                     */
            System.out.println(myStaticString); // Good!

            myNonStaticMethod(); /* Error: Non-static method myNonStaticMethod cannot be
                                  * referenced from a static context
                                  */
            myStaticMethod(); // Good!
        }
    }
}
```

Be careful!

Inheritance does not make sense for static methods. Indeed, the *Java* compiler resolves the static method calls at compilation time. Hence, overriding does not exist for those methods. Instead, redefining a static method leads to *shadowing*.

1.2 Initializer blocks

An initializer block is a code block written in your class file. It can either be static or not. In the latter case, it is named an “instance initializer block”.

An instance initializer block is executed at every object instantiation, while a static initializer block is executed at **Class Loading**.

These blocks are especially useful if a variable initialization requires some logic (error handling or `for` loops for instance), and you do not want to clutter your constructor since it will always be the same algorithm.

```java
public class Main {
    public static class MyStaticClass {
        private static List<Integer> myStaticList = new ArrayList<>();
        private List<Integer> myList = new ArrayList<>();

        static {
            for (int i = 0; i < 5; i++) {
                myStaticList.add(2 * i);
            }

            System.out.println("Static initializer block executed!");
        }

        {
            for (int i = 0; i < 5; i++) {
                myList.add(2 * i + 1);
```

(continues on next page)

(continued from previous page)

```
            }

            System.out.println("Instance initializer block executed!");
        }

        public MyStaticClass() {
            System.out.println("MyStaticClass instantiated!");
        }
    }

    public static void main(String[] args) {
        System.out.println("Program starting!");
        final var myInstance = new MyStaticClass();
    }
}
```

Output:

```
Program starting!
Static initializer block executed!
Instance initializer block executed!
MyStaticClass instantiated!
```

Be careful!

This example might trick you into thinking that an instance initializer block is executed before the constructor. It is rather copied into the constructor, after the call to the `super` constructor (which might be implicit if the `super` constructor does not take any arguments):

```
public class Main {
    public static class ParentClass {
        public ParentClass() {
            System.out.println("Parent class constructor!");
        }
    }

    public static class ChildClass extends ParentClass {

        {
            System.out.println("Child class initializer block!");
        }

        public ChildClass() {
            System.out.println("Child class constructor!");
        }
    }

    public static void main(String[] args) {
        final var childInstance = new ChildClass();
    }
}
```

Output:

```
Parent class constructor!
Child class initializer block!
Child class constructor!
```

1.3 Static nested classes

Java allows you to define nested classes. Nested classes can either be non-static or static (Static nested classes).

Let us see how it works:

```java
public class Main {
    public static class MyClass {
        public static class MyStaticNestedClass {
            public void myPrint() {
                System.out.println("I am the static nested class!");
            }
        }
    }

    public static void main(String[] args) {
        final var myInstance = new MyClass.MyStaticNestedClass();
        myInstance.myPrint();
    }
}
```

Output:

```
I am the static nested class!
```

Just like inner classes, static nested classes can have any modifier: `public`, default, `protected` or `private`.

While inner classes, being non-static, have to exist within an instance of their enclosing class and can therefore access any of its instance members, static nested classes can only access the static fields of their enclosing class.

Therefore, they are mostly used for packaging convenience rather than for the access to these fields. For this very reason, declaring a private static nested class would not be particularly useful.

```java
public class Main {
    public static class MyClass {
        public String myNonStaticString = "Non-static String!";
        public static String myStaticString = "Static String!";

        public void myNonStaticMethod() {
            System.out.println("Non-static method!");
        }

        public static void myStaticMethod() {
            System.out.println("Static method!");
        }
```

(continues on next page)

(continued from previous page)

```java
        public static class MyStaticNestedClass {
            public void run() {
                System.err.println(myNonStaticString); /* Error: non-static field
                                                         * myNonStaticString cannot be
                                                         * referenced from a static context
                                                         */
                System.out.println(myStaticString); // Good!

                myNonStaticMethod(); /* Error: non-static method myNonStaticMethod cannot be
                                      * referenced from a static context
                                      */
                myStaticMethod(); // Good!
            }
        }
    }
}
```

Just like inner classes, static nested classes can declare variables that could shadow variables from their enclosing class. Once again, the solution to this is a small syntax trick:

```java
public class Main {
    public static class EnclosingClass {
        private static final int myShadowedInt = 1;
        private static final int myNonShadowedInt = 2;

        public static class NestedClass {
            private final int myShadowedInt = 47;
            private final int myNonShadowedInt = 3;

            public void run() {
                System.out.println("My shadowed int: " + myShadowedInt);
                System.out.println("My non-shadowed int: " + EnclosingClass.myNonShadowedInt);
            }
        }
    }

    public static void main(String[] args) {
        final var nestedClassInstance = new EnclosingClass.NestedClass();
        nestedClassInstance.run();
    }
}
```

Output:

```
My shadowed int: 47
My non-shadowed int: 2
```

2 Final

The meaning of the `final` keyword provided by *Java* depends on where it is located:

- In a **class** declaration;
- In a **method** declaration;
- In an **attribute** declaration;
- In a **variable** declaration.

2.1 Final class

When a class is declared as `final`, it means that no other class can inherit from it. If you think of inheritance as a tree, then a `final` class is a leaf.

```java
public class Main {
    public static final class FinalClass {
    }

    public static class BrokenClass extends FinalClass {
        // Error: Cannot inherit from the final class FinalClass
    }
}
```

2.2 Final method

When a method is declared as `final`, it means that it cannot be overridden.

```java
public class Main {
    public static class ParentClass {
        public final void finalMethod() {
            System.out.println("Final method");
        }
    }

    public static class ChildClass extends ParentClass {
        @Override
        public void finalMethod() {
            // Error: Cannot override final method finalMethod from ParentClass
            System.err.println("Attempting to override final method");
        }
    }
}
```

2.3 Final attribute

When a class attribute is declared as `final`, it means that its value is evaluated at instantiation, and that from this point on, it cannot be reassigned.

```java
public class Main {
    public static class MyClass {
        public final int myInt = 0;

        public void myMethod() {
            myInt++; // Error: Cannot assign a value to final variable myInt
        }
    }
}
```

This point can be quite misleading: you could assume that it behaves like *C* keyword. Indeed, for *primitive types*, there is no great difference between `final` and `const`.

However, when we start playing with *objects*, it gets completely different: where `const` makes the object **read-only**, `final` just **prevents it from being reassigned**.

```java
public static class MyClass {
    private final List<String> myList = new ArrayList<>();

    public void myMethod() {
        // Even though myList is final, we can still add elements to it...
        myList.add("Hello");
        myList.add("world!");

        // Set its elements...
        myList.set(1, "Hello yaka!");

        // Or even remove
        myList.remove(0);

        // However, you cannot reassign it.
        myList = new ArrayList<>(); // Error: Cannot assign a value to final variable myList
    }
}
```

Going further...

If you want one of your classes to be read-only, you can make it **immutable**! Since *Java 9*, you can create *immutable collections* using List.of.

```java
public class Main {
    public static void main(String[] args) {
        final var myImmutableList = List.of("Can't", "touch", "this");

        // UnsupportedOperationException (at runtime): cannot append to an␣
↪immutable List
        myImmutableList.add("Attempting to add an element");


        // UnsupportedOperationException (at runtime): cannot set an element of␣
↪an immutable List
        myImmutableList.set(0, "Attempting to set an element");
    }
}
```

However, this can only bring us as far as creating a list of `final` *references*. This means that it is possible to do this:

```java
public class Main {
    public static void main(String[] args) {
        final var myList1 = new ArrayList<Integer>();
        final var myList2 = new ArrayList<Integer>();
        for (int i = 0; i < 5; i++) {
            myList1.add(2 * i);
            myList2.add(2 * i + 1);
        }
        final var myNotSoImmutableList = List.of(myList1, myList2);
        System.out.println(myNotSoImmutableList.toString());
        myList1.add(42);
        myList2.add(47);
        System.out.println(myNotSoImmutableList.toString());
    }
}
```

Output:

```
[[0, 2, 4, 6, 8], [1, 3, 5, 7, 9]]
[[0, 2, 4, 6, 8, 42], [1, 3, 5, 7, 9, 47]]
```

2.4 Blank `final`

Sometimes, you cannot know right away the value your `final` variable will have: it will depend on the arguments given to the constructor of your class. You have a solution for this: the blank final.

Evaluating the value of an attribute at instantiation means you can directly inline its value at field declaration (as you have seen above), but also assign it in the constructor. That is what a blank final is: a final attribute which value is assigned in the constructor.

Beware: if you declare a blank final, its value must still be set at instantiation. You cannot assign it in another method.

```
public static class MyClass {
    private final int inlinedFinal = 0; // Value is inlined at instantiation: good.

    // Error: variable blankFinalNeverSet might not have been assigned.
    private final int blankFinalNeverSet;

    private final int blankFinalSetInAMethod;
    private final int blankFinalTriedToBeSmart;

    private final int goodBlankFinal;

    public MyClass() {
        usedOnlyInConstructor();
        goodBlankFinal = 4;
    }

    public void myMethod() {
        // Error: You cannot set a blank final in any other method than the constructor.
        blankFinalSetInAMethod = 2;
    }

    private void usedOnlyInConstructor() {
        blankFinalTriedToBeSmart = 3; // Well tried...
    }
}
```

2.5 Final variable

A `final` variable is pretty much like a `final` attribute, but lives in the scope of a method: it cannot be reassigned. You can also declare it without a value, if for instance its value depends on an `if ... else` or `switch ... case` statement.

```
public class Main {
    public static void main(String[] args) {
        final String assignedAtDeclaration = "Our only hope is ";
        final String blankFinal;

        if (args.length != 1) {
            blankFinal = "to recode everything!";
        } else {
            blankFinal = args[0];
        }

        System.out.println(assignedAtDeclaration + blankFinal);
    }
}
```

Likewise, a `final` argument can be given to a method, in order to check that it is never reassigned. The `final` keyword is not considered part of the method signature: therefore, it does not create issues for overriding.

Tips

Using the `final` keyword can greatly improve the performance and the quality of the code you are writing. Knowing that a value cannot be reassigned makes reading industrial code easier. By having that information, the *Java* compiler can sometimes optimize the access of elements that use the keyword.

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