Tutorials — Syntax

version #1.0.0

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

Copyright

Contents

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

1 Introduction to *Java*

1.1 General presentation

Java is a class-based and object-oriented computer programming language. It is intended to let application developers *write once, run anywhere*, meaning that compiled code can run on all platforms that support *Java* without the need for recompilation. *Java* applications are typically compiled to **bytecode** that can run on any *Java virtual machine (JVM)* regardless of the computer's architecture. *Java* was originally developed by James Gosling at Sun Microsystems (merged into Oracle Corporation).

You will learn this language using **Java 21**.

1.1.1 Overview

Let's have a quick overview of what you can and cannot do in *Java*.

- In *Java*, we use **packages** to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, and enumerations easier, etc. A package can be defined as a group of related types providing access protection and namespace management. Programmers can define their own packages to bundle some classes together.

  Some existing packages in *Java*:

  - `java.lang` - the fundamental classes
  - `java.io` - classes for input and output methods

  To use a package, we use the `import package.class` statement. To assign a class to a specific package, we use the `package` statement at the top of a class file.

- The file extension of a *Java* file is `.java`.
- There are **no stand-alone functions** in *Java*: each method must be member of a class. The same applies to **variables.** As a consequence, there are neither global variables nor global methods in *Java*.
- Except `Object`, which has no superclass, every class has **one and only one direct superclass (single inheritance).** In the absence of any other explicit superclass, every class is implicitly a subclass of `Object`.
- **Forget pointers!** The language does not allow you to modify a symbol's address (and even less to iterate upon it).

- **No manual memory management!** *Java* uses a garbage collector. We don't have to free nor delete our objects. *Java* does it for us when we don't use them anymore.
- All objects in *Java* are passed **by reference**[1] except for primitive types which are passed by value. We will explain this point later.
- *Java* is a **multi-platform** language. Therefore, you should not use "`\n`" (Linux/UNIX/*BSD's line break) as a line separator; instead, you **must** use the *Java*'s system-independent line break:

  `System.lineSeparator()`

  which works as well on your own Linux as on Windows or MacOS.

1.2 Basics

The main control structures in *Java* are (almost) the same as in *C*:

- `if (...)` **...** `else`
- `while (...)`
- `for (...)`
- `switch (...) case`

They won't be explained in detail here, as you already know them and will pick up their *Java* syntax easily.

1.3 Methods

Methods are the equivalent of functions for the *Java* language.

We will not go into the details of their usage now. You only need to be able to declare and implement one for now.

The syntax to declare a basic function you saw in *C* was the following:

```c
// exported by a header file if needed
int myFunction(int c)
{
    return c + 1;
}
```

This declaration would have the following equivalent in *Java*:

[1] If necessary, feel free to read explanations and examples at http://stackoverflow.com/questions/40480/is-java-pass-by-reference.

```java
// file: MyClass.java
public class MyClass {
    public static int myFunction(int c) {
        return c + 1;
    }
}
```

`myFunction` is a *static* function, meaning that you can use it in another file or function with this syntax:

```java
MyClass.myFunction(2);
```

1.4 The Main class

This is how to create a class with a main method in a `Main.java` file. To display something on the standard output in *Java*, call the `println` method, located in the class `System.out`.

```java
package mypackage;

class Main {
    public static void main(String[] args) {
        System.out.println("My first class!");
    }
}
```

2 Boxing and unboxing

Java is a strongly typed object-oriented language. Every *Java* class inherits from the `Object` class. Yet, for performance reasons, a part of the *Java* language does not inherit from the `Object` class and is not object-oriented: the primitive types.

Those primitive types are `int`, `short`, `long`, `byte`, `char`, `float`, `double` and `boolean`. They represent single values, not complex objects.

However, sometimes you will need to use an object representation of such types. For example, data structures in *Java* only work with objects. This is where wrapper types are useful: `Integer`, `Short`, `Long`, `Double`... They are classes that encapsulate a primitive type within an object.

2.1 What are boxing and unboxing?

Java supports the conversion of *primitive types* into their object equivalents, *wrapper types*, in assignments or methods and constructors invocations. The encapsulation of a primitive value within its object equivalent is known as *boxing*.

The reverse operation is called *unboxing*: *Java* supports the conversion of *wrapper types* into their *primitive* equivalents (if needed) for assignments or methods and constructors invocations.

```
public int extract(Integer iObj)
{
    int j = iObj.intValue(); // unboxing

    return j;
}

public Integer encapsulate(int i)
{
    Integer jObj = new Integer(i); // boxing

    return jObj;
}
```

These conversions can also be performed automatically by the *Java* compiler. This is called **autoboxing** and **auto-unboxing**. For autoboxing, the primitive type is automatically converted to its object equivalent when needed without calling the constructor of this class.

```
public Integer autoConvert(int i)
{
    Integer jObj = i; // autoboxing

    return jObj;
}
```

2.2 When should you use boxing/unboxing?

It is not appropriate to use boxing and unboxing without a specific need (i.e. when using data structures) as a repeated conversion will have a performance impact on your program. Moreover, an `Integer` is not a substitute for an `int` — boxing and unboxing blur the distinction between primitive types and reference types, but they don't eliminate it.

3 List

3.1 ArrayList

For the next parts of this tutorial, you will need to use lists. *Java* provides an implementation of this data structure, named ArrayList

```
public class Main {
    public static void main(String[] args) {
        var myList = new ArrayList<Integer>();
        // Appends the given list at the end of myList.
        myList.addAll(Arrays.asList(1, 3, 5, 7, 9));

        myList.add(2); // Appends 2 at the end of myList
        myList.remove(3); // Remove the element at index 3 (7)
        myList.set(0, 47); // Sets the value at index 0 to 47

        final var myListSize = myList.size(); // Number of elements in the list
        // Print the content of myList
        for (var index = 0; index < myListSize; index++) {
            var valAt = myList.get(index); // Gets the element at the given index
            System.out.println(valAt);
        }
    }
}
```

Output:

```
47
3
5
9
2
```

The type between angle brackets ('< >') in the syntax of `ArrayList` is called a generic type. You will see more about them in the tutorial on generics. As we said in the part about *boxing*, data structures in Java only work with objects: keep in mind that you cannot pass a primitive type (`int`, `float`, `char`...) as a generic type, you can only use their object counterpart (respectively `Integer`, `Float`, `Character`).

3.2 Iteration

Iterating over an `ArrayList` using an `int` index can be quite tiresome. Thankfully, *Java* provides a convenient way to smoothly iterate over data structures:

```java
public class Main {
    public static void main(String[] args) {
        var myList = new ArrayList<Integer>(Arrays.asList(1, 3, 5, 47, 9));
        for (var value : myList) {
            System.out.println(value);
        }
    }
}
```

Output:

```
1
3
5
47
9
```

4 String

4.1 Operations

There are several ways to instantiate a new `String` object from the String class. Here is a short example:

```java
public class Main {
    public static void main(String[] args) {
        char[] hello = { 'H', 'e', 'l', 'l', 'o' };
        var helloStr = new String(hello);
        var world = "World!";
        var empty = new String(); // Builds an empty String ("")
        var goodBye = new String("Goodbye");
        var worldCpy = new String(world);

        System.out.println(helloStr);
        System.out.println(world);
        System.out.println(empty);
        System.err.println(goodBye);
        System.err.println(worldCpy);
    }
}
```

Output:

```
Hello
World!

Goodbye
World!
```

Going further...

The `var` keyword allows you to use *local type inference*: the compiler can deduce the variable's type from the given value.

Java's `String` does not behave in the same way as *C*'s `char *`. There is no `operator[]` here: if you want to access a character at a certain index, you need to use the `charAt` method.

```
public class Main {
    public static void main(String[] args) {
        var myString = "Java";

        System.err.println(myString[0]); // Error: array required, String found
        System.out.println(myString.charAt(0)); // Output: J
    }
}
```

Java's `String` also implements several very useful methods:

- `strip`: returns the `String` stripped from all leading and trailing whitespaces;
- `substring`: returns a substring of the `String`;
- `toUpperCase`/`toLowerCase`: returns the given `String` in upper or lower case respectively.

```
public class Main {
    public static void main(String[] args) {
        var tooManyWhitespaces = "      I just  ";
        var bigString = "I wanna feel sunlight on my face.";
        var lowerCase = "help you";

        System.out.print(tooManyWhitespaces.strip());
        System.out.print(bigString.substring(1, 8));
        System.out.println(lowerCase.toUpperCase());
    }
}
```

You can find the documentation about these methods (and a lot more) in the Oracle documentation.

Going further...

More generally, you should take a look at the Oracle documentation for every class from the *Java* API that you have to use.

If you want to compare the content of two String, the == operator will not do what you want. Remember that in *Java*, every time you create a new variable, it then holds a reference to an allocated object (except for primitive types, which String is not). Therefore, comparing two String with the == operator will compare their addresses (the same way as comparing two char* in *C*). Comparing two String's contents shall be done through the equals method.

```java
public class Main {
    public static void main(String[] args) {
        var myString = "Don't be afraid.";
        var myStringCpy = new String(myString);

        System.err.println(myString == myStringCpy); // false
        System.out.println(myString.equals(myStringCpy)); // true

        // However...
        var myOtherString = "Don't be afraid.";
        // myString and myOtherString are string literals, they refer to the same memory area.
        System.out.println(myString == myOtherString); // true
    }
}
```

Going further...

If you want to know more about string literals, read the part 3.10.5 of the Java specification.

You can also append a String to another with the operator +. However, **you should not use this.** *Java* Strings are **immutable**: once they have been instantiated, they cannot be modified. Therefore, using the operator + allocates a brand new String and shoves the contents of both String into it.

4.2 StringBuilder

The StringBuilder class represents a mutable sequence of characters. This is what you should use in order to build a String through appending, using the append method, and then toString, or String's constructor which takes a StringBuilder as argument.

```java
public class Main {
    public static void main(String[] args) {
        var stringBuilder = new StringBuilder("Hello ");

        stringBuilder.append('w');
        stringBuilder.append(0); // Appends '0', not (char)0!
        stringBuilder.append((char)114); // 114 is the ASCII value of 'r'
        stringBuilder.append("ld!");
        System.out.println(stringBuilder.toString());
```

(continues on next page)

(continued from previous page)

```
        var myString = new String(stringBuilder);
        System.out.println(myString);
    }
}
```

Output:

```
Hello w0rld!
Hello w0rld!
```

There is another class, named StringBuffer, which does a similar job to `StringBuilder`, the main difference between the two of them being that `StringBuffer` is thread-safe. However, since it has to lock, it is way slower than a `StringBuilder`. Therefore, use `StringBuffer` only when necessary.

Be careful!

During this workshop, you might come across some `String` appended with the operator `+`, in tutorials for example. This is done only to avoid cluttering. You should use `StringBuilder` if you want to append anything to a `String`. Note that, in simple cases, the + operator can sometimes be desugared to `StringBuilder` by the compiler.

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