# Java review

## Abbreviations

In this review, the following abbreviations are used:

* OS: Operating System
* JVM: Java Virtual Machine
* WORA: write once, run everywhere
* OO: object oriented

## History

Java is a general purpose programming language ( embedde languages -> 5. semntics). It was released together with the Java platform in 1996 by a team at Sun MicroSystems (now Oracle), whose leader was James Gosling. The Java language has been defined as a core element of the Java platform, until Java 7. The project started in December 1990, with the name *Stealth Project,* then renamed *Green Project*. Other relevant developers are Mike Sheridan and Patrick Naughton.

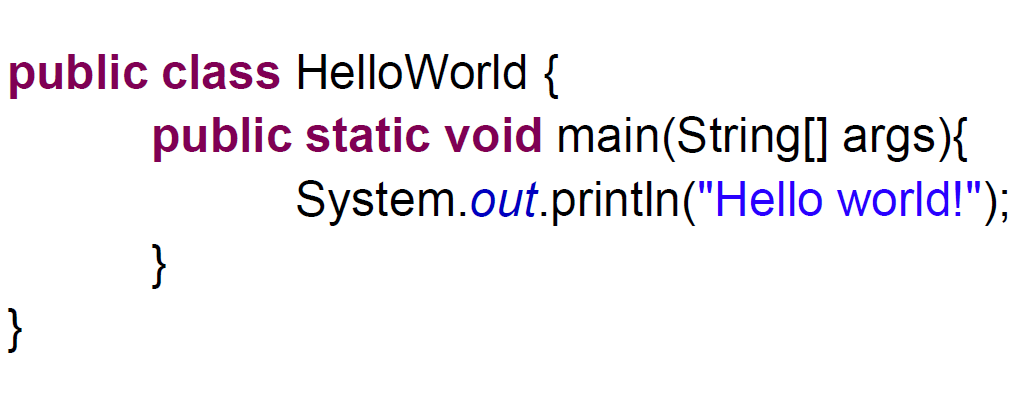
The language was initially called Oak after an oak tree that stood outside Gosling's office. Later the project went by the name Green and was finally renamed Java, from Java coffee, the coffee from Indonesia.

The original target of the project was interactive television, since the team was dreaming of combining digital consumer devices and computers, to exploit the power of the latter in the everyday life. Indeed, the Green Team demonstrated their new language with an interactive, handheld home-entertainment controller that was originally targeted at the digital cable television industry. Considering that it was just the 90’s, it really was a dream, and unfortunately the concept was much too advanced for the team at the time and didn’t succeed.  Luckily, it was successfully employed in the new born Internet. Since there weren’t languages able to provide dynamic content to HTML pages, Java filled this gap allowing, through the so-called *Java applets,* the creation of dynamic, OS independent, and browser independent content for the web. All the original success of Java comes from this.

Moreover, the main goal of the project was to offer to developers a way of achieving the principle WORA (*write once, run anywhere).* Indeed, Java platform is a software product which includes an execution engine (called a Virtual Machine), a compiler and a set of libraries. The deployment of a Java program works as follows: the compiler produces java bytecode from the original source code, then the Virtual Machine, which is different per each OS, executes the bytecode in the context of the specific machine. This allows portability of java programs, since the developers don’t have to take care of the underlying machine, and a compiled program can run everywhere without the need for recompiling it in the new machine. Java platform have been provided for almost every machine, making the Java language so popular even after nowadays.

Between November 2006 and May 2007 Sun has released almost the entire Java technology under the GNU General Public Licence, making it free.

The language evolved during time: as an example, the original applets, which had so much importance, are now deprecated. The current supported versions are Java 13, released in October 2019, and Java 11, released on September 25, 2018.



Snippet 1: A Java program

## Characterisation of the language

Java is (primary) an object-oriented language. A Java program is organized as a set of classes, where each class corresponds to the declaration of a new user-defined type. This is the simple way to create new types in Java: it doesn’t have explicitly type constructors. Each class allows for the declaration of variables (*attributes* in Java) and functions (*methods)* and constant attributes,which compose the class. With attributes and methods, the user can specify characteristics and possible operations on the type itself. The instances of classes are called objects. Attribute values distinguish single objects inside a *class* type and are a representation of the *state* of an object at every execution point of the program. Methods are used to manipulate instances, such as changing their attributes’ values. All classes names should start with uppercase letters, while methods and variable names start with lowercase letters. The main program is represented by a special method of a class, called *main,* which is the portion of code executed when we run a program. Java provides some libraries with useful classes (such as the Collection library, or the Exception library) to allow the developer to exploit already built functionalities.

When defining a new class, the developer may specify how to create objects of that class by adata constructor*:* a constructor is simply a method which has the same name of the class, does not have a return type, and returns an instance of the class. A constructor may have parameters. The user can specify as many constructors as desired, but all with a different set of parameters. If no constructor is provided, Java provides a default constructor, where all the attributes are set at some default values (which is null for user defined types). Qua discuti dei costruttori se hai tempo

An object of a class (i.e., an instance of a user defined type) is created via the *new* operator, which is applied to a constructor. It allocates at runtime the necessary memory and return a reference of the new object. Java allows the user to create new objects and bind a reference of the new object to a *variable*. A variable declaration must have a unique name and a type (either primitive or user defined). Java is then *explicitly typed*: all the variable declaration must be accompanied by the type of the variable. A variable name is associated with the corresponding value in the main memory only if the variable is of a primitive type (see below), while if it’s of a class, it contains only a reference to the starting point in memory where the space for the variables of that class is allocated.

Java is *statically typed*: all the types (of program variables, class attributes, and methods) must be known at compile time and are controlled by the type checker for consistency. If not consistent, an error is raised at compile time.

Java enforces also *static (or lexical) scoping* of variables: the scope of a variable name is the portion of the program in which that name is associated with a particular variable. Static scoping means that the binding between a name and a variable is done at compile time and not at run time (which is dynamically scoping). If the local block of code that uses or assigns a variable doesn’t contain the declaration or initialization of that variable before the use or assign, the compiler looks at the ascendent blocks (*static parents*) as they are lexically written in the code to do the binding name-variable. If no variable declaration has been found, a compile time error is raised: contrary to JavaScript, Java requires that all variables are declared before using or assigning them.

Since Java does not support natively functional programming (see below), methods are not first-class citizens of the language and cannot be passed as arguments to other methods. Moreover, all the arguments of a method are evaluated before the method itself regardless if this is needed in the called method (*eagerly evaluation,* contrary to *lazy* *evaluation*). Then, when calling a method, Java does not pass the references of the arguments, but stores their values into new variables (*call by value*, contrary to *call by reference*). Java supports lazy evaluation only for Boolean && and || operators, which will not evaluate their right operand when the left operand is false (&&) or true (||) (*short circuit evaluation*) and the ?: operator, which evaluates a Boolean expression and subsequently evaluates only one of two alternative expressions (of compatible type) based on the Boolean expression's true/false value.

When we state that Java applies a call by value strategy, we have to keep in mind that a class variable are just a reference to the variables of the class, so if the caller passes a copy of this reference to the callee when calling a method, it’s still passing a reference. The only case when references are not passed is for primitive types ( see below).

Java allows for recursive data types: a class can contain an instance of the same class as attribute. As an example, see snippet 2, where a simple recursive list is defined. To avoid infinite recursion which would cause a segmentation fault, in Java variables can be *null* (since, as already mentioned, variables are references, it means that they not reference any object) to make the list of a limited length.

**class** **List**<E> {

E value;

List<E> next;

}

Snippet 2: a simple definition of a list

The code in snippet 2 is relevant also to spot another Java feature, related to polymorphism: *generics*. Generics were added to the language starting from Java 5 and consist of the possibility of specifying type variables inside the declaration of class, methods, constructors, and interfaces (for interfaces, see the part about polymorphism). Type variables are simply unqualified identifiers. A class (or a method, constructor or interface) is generic if declares one or more type variables. These type variables are called the type parameter for the class (formal type parameter in case of a method or a constructor). A constructor can be generic even if the corresponding class is not generic. In snippet 2, *List* is a generic class which declares the type variable *E*. Type variables are expressed with the <> syntax. Generics are indeed extensively used in the Collection Java library, since they allow to avoid writing different code for every possible type of the members of a collection. A generic class is also called a *parameterized class*. When we want to create an object of the List class, we have to pass the *actual* type parameter, as in snippet 3. The type checker guarantees type safety at compile time.

List<String> v = **new** ArrayList<String>();

v.add("test");

Integer i = v.get(0); *// (type error) compilation-time error*

Snippet 3: how to use a generic class

On the other hand, we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

Qua si potrebbero dire centomila cose sui generics

The use of generics is also known as parametric polymorphism in Java. Moreover, Java allows also other types of *polymorphism,* related to the *subtyping* system Java has. Java indeed provides *subclassing,* which is a special type of subtyping which allows the subtypes to reuse the implementation of the supertype. The subclassing relation we observe in Java is the same as the subset interpretation of the subtyping relation: when a class is a subclass of another class (the superclass), evert object of the subclass is still an object of the superclass. This is also called *inheritance,* and it’s one of the three core principles of OO programming (the other two are encapsulation and polymorphism). Inheritance allows to reusability of code and variables.

When we want to declare a class as subclass of a superclass, we use the *extends* syntax. A class can be a subclass only of one superclass. All the user defined classes, if not already subclass of another class, are actually a subtype of the class *Object*. The subclass relation is transitive: we can have a chain of subclasses, where each class inherits all the methods and variable of the ancestors. We can assign to a variable of a class an instance of a subclass of the declared class. As an example, in Snippet 3, an instance of the ArrayList class is assigned to a List object. In this way we have a variable with both a *static type (*the type defined in the declaration) and a *dynamic type* (the actual type of the variable) which can differ, and still we have the same functions (methods*.* Java guarantees that this does not compromise the type safety by checking whether the object is manipulated correctly based on the static type. This prevents errors at run time. Similarly, we can pass as actual parameter to a method an object of a subclass of the class of he formal parameter of the declaration of the method.

In the context of subclasses, we obtain a polymorphic behaviour of the code through *overriding*. In fact, a subclass may have his own implementation of the inherited methods “overriding” the implementation of the superclass. To do this, it simply should declare a method with the same name, the same parameters and the same return type of a method of a superclass. When a method is called, the JVM looks at run time at the dynamic type of the variable and executes the most specific implementation of the called method. The binding between the called method and the implementation is dynamic, i.e. is done at run time. This means that the compiler doesn’t generated the code to execute the method, but it just generates the code which looks for the right implementation of the called method.

If we want to prevent from extending a class or a method, we have to declare it *final*.

A method can be declared abstract when no implementation is specifying. A class which contains an abstract method is an *abstract class.* It’s forbidden to create objects of an abstract class. Abstract classes are a way to introduce some kinds of high-level abstractions from the actual implementation. We could define an abstract class and use it extensively in the code and then implement it in different ways in different concrete subclasses (and change the implementation if needed). Classes which extend an abstract class should implement all the abstract methods, otherwise they should be declared abstract as well.

Java introduces also the concept of *interface* to allow for multiple inheritance. An interface is much similar to a class, with the exception that it can’t have variables and can only have abstract methods and constant attributes. Interface can inherit from more than one interface, and a class can implement one of more interfaces. When a class implements an interface, it must implement all the methods of that interface, otherwise it must be declared abstract.

At the core of this strategy there is the concept of contract.

Another way Java provides polymorphism with is overloading (also called ad hoc polymorphism) --- lasciamo stare un attimo l’overloading

Principio di sostituzione di Liskov??

An important aspect of Java generics is the ? wildcard, since it guarantees type safety.

Recent versions of Java are not only a simple object-oriented language but have included also some support for functional programming (and procedural programming). Due to that, Java is also called a multi-paradigm programming language.

---- ----

Java provides also a way of handling concurrency safely.

------

Java is a type safe language. Indeed, Java has a sophisticated way of handling exception and errors, via the Throwable library. The Throwable library contains as subclasses the Error class and the Exception class. The Error class (along with the RunTimeException class, a subclass of Exception) consists of errors which can be fixed only by changing the code (such as division per zero

---------

A useful feature of Java is the garbage collector, which is a routine of the JVM called periodically which checks if there are variables stored in memory which are no more referenced in the current environment, deleting them.

### Primitives types

Apart from user-defined types, Java provides the following already build in types (*primitives types*):

* Byte: 8-bit signed two's complement integer. It takes values between -128 and 127 (inclusive);
* Short: 16-bit signed two's complement integer. It takes values between -32,768 and 32,767 (inclusive);
* Int: 32-bit signed two's complement integer. It takes values between -231 and 231-1 (inclusive). It can be used also to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 232-1. The *Integer* class allows indeed to do that;
* Long: 64-bit signed two's complement integer. It takes values between -263 and 263-1 (inclusive). It can be used also to represent an unsigned 64-bit integer, which has a minimum value of 0 and a maximum value of 264-1. The *Integer* class allows indeed to do that;
* Float: a single-precision 32-bit IEEE 754 floating point;
* Double: a double-precision 64-bit IEEE 754 floating point;
* Boolean: only two possible values (*true* and *false*);
* Char: a single 16-bit Unicode character;

In addition to that, Java provides support for character strings, through the *String* class. String objects are immutable, which means that once created, their values cannot be changed.

Java allows for embedded languages

Global and local variables????

## Strengths and weaknesses of the language

The power of Java is its portability, along with the fact that it has been progressively enriched with much more functionalities (e.g., functional programming), without losing its original features. Indeed, the WORA principle, which has been completely achieved by Java, makes it very pervasive in the computer science communities and even in the society, and still nowadays is an impressive feature of the language.

The explicit typing system makes it easy to read and maintain, and to use for teaching purposes.

The fact that it’s statically typed allows to avoid some errors at runtime, since the type checker detects them at compile time. In software engineering, detecting the errors as soon as possible is a core principle, since it has been proved that the cost of recovering from a code bug increases exponentially with the stage in which the bug is detected.

Java provides a lot of libraries with useful functionalities: for example, the Socket/RMI libraries allow quite easily to manage the communication of the program. Other language, such as Haskell, do not provide the same functionalities, and force the developer to implement from scratch these libraries. Already implemented libraries have also the advantage that they are widely used (and tested), so they are less probable to contain bugs, compared to just written code.

The way Java handles the concurrency is smart and intuitive: for example, if a synchronized method tries to acquire a lock on the same class (maybe via recursion or calling another synchronized method, it should cause deadlock ( since it would be waiting for a lock from itself, which is a “silly” deadlock indeed), but the JVM is able to realize that and let the method keep the lock and continue the execution. This is not the case for the C++ language, and some complex alternatives should be implemented by the developer.

The Exception handling system could be found very annoying and boringdy developers, so that even in the Java official Oracle Tutorial (for JDK 8) it is mentioned the case where developers make their exceptions subclasses of RunTimeException to avoid catching it every time. On the other hand, Exceptions in Java make it type safe and allow for an accurate control of the operations and the execution flow.

Static scoping allows to easily detect how a variable is initialized just by looking at the plain code: on the other side, with dynamic scoping, in some cases, the developer is obliged to consider all the possible executions flows if he wants to know the value of a variable, which for big programs is very hard.

The Java Garbage collector is a strength of the language, since it allows to save memory. Comparing to C, this is particularly useful for non-expert programmers, which may forget to call the *free* C function and then cause the program to be incredibly memory consuming at runtime.

The need for a JVM, which is an additional needed software, could be a negative point when we are dealing with microcontrollers or such small devices. Even though some tentative have been done, still C and C++ are preferred. This is the downside of the JVM.

Java played an important role in the diffusion of the Test-Driven Development through the deployment of Junit, which is a simple framework to write repeatable tests.

Encapsulation, which is one the core principles of OO languages, permits to update and change the code very easily, achieving the modularity principle, which is very important as well in software engineering to save money and time when some changes need to be done.

All mentioned pros and cons make Java, from my point of view, particularly suitable for big projects which combine several models, when there is need for readable and clear code and correctness is a very complex issue. Java seems particularly suitable for defensive programming.

Moreover, today Java is not considered a Web language, since it needs for tools and layers to be used in developing web application. Because of that, other languages are sometimes preferred. This is a challenge for the future which, if not solved, could reduce the diffusion of Java.

The absence of pointers makes Java much safer than, for example, C: a common computer security issue is the case when the user tries to access forbidden memory security areas through the input to the program, thanks to pointers. This is simply not possible in Java, which is commonly defined as a high-level programming language due to the absence of pointers.

The frequency of new releases (a 6-month cadence for the last ones) has been pointed out by some developers as causing “release fatigue”: it’s hard to update the code so fast, especially for open souce project. On the other hand, these releases aims at filling the gaps of Java in providing some functionalities ( most of all, functional programming) which have been claimed by the community.

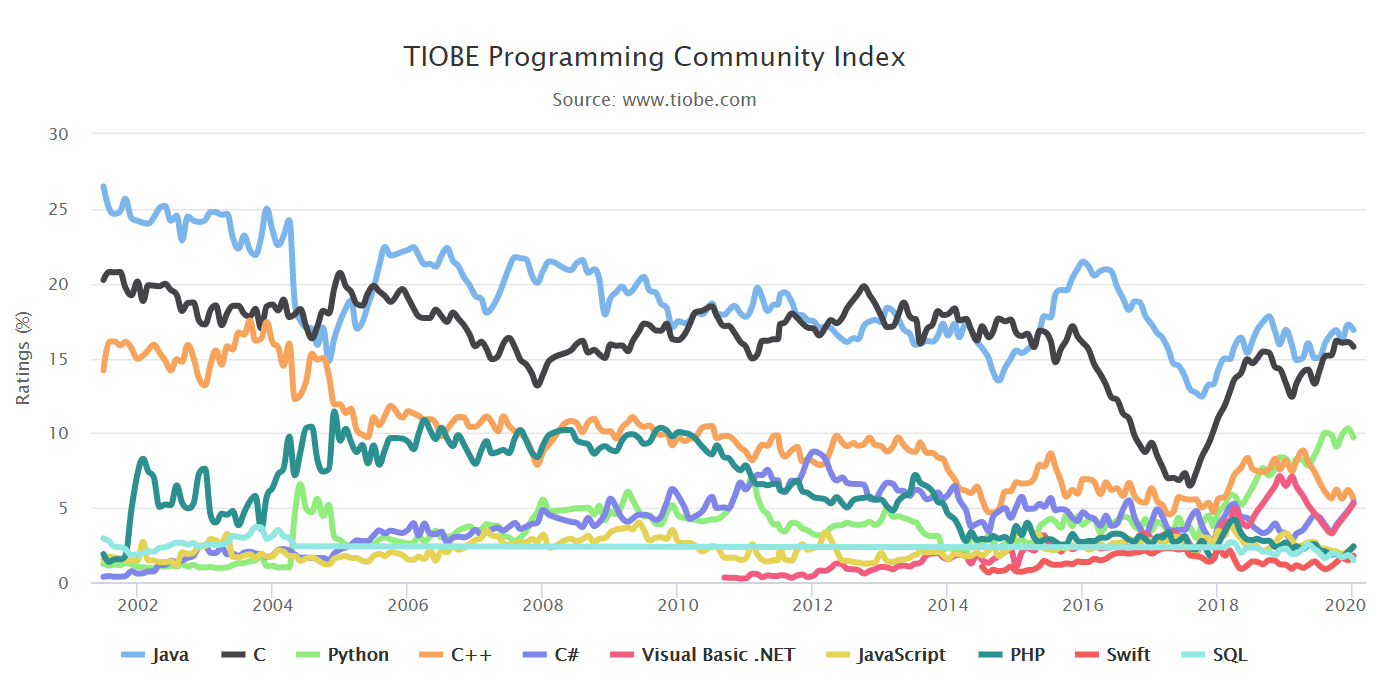


Figure 2: The TIOBE Programming Community index is an indicator of the popularity of programming languages. As the beginning of 2020, Java is the first one, strictly followed by C.

* Manca l’ereditarietà multipla
* Java funzionale (che però è stato introdotto di recente)
* Construttori che non ci sono più
* Release frequency

## References

From the Internet:

* <https://en.wikipedia.org/wiki/Java_(programming_language)>
* <https://en.wikipedia.org/wiki/Java_(software_platform)>
* <https://www.appdynamics.com/blog/engineering/the-history-and-future-of-java-programming-language/>
* <https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>
* <https://stackoverflow.com/questions/20103318/java-based-microcontrollers>
* <https://www.tiobe.com/tiobe-index/>
* <https://junit.org/junit5/>
* <https://dzone.com/articles/jdk-10-release-cadence-release-fatigue-and-support>

From books (in italian)

* Pellegrino Principe, (2018), Java 11: Guida allo sviluppo in ambienti Windows, macOS e GNU/Linux

Notine

* Come funziona l’overloading in Java? Il tipo ritornato deve essere lo stesso)