Java review

Abbreviations

In this review, the following abbreviations are used:

- OS: Operating System
- JVM: Java Virtual Machine
- WORA: write once, run anywhere
- OO: object oriented
- STM: software transactional memory

History

Java is a general-purpose programming language. It was released together with the Java platform in 1996 by a team at Sun MicroSystems (now Oracle), whose leader was James Gosling. The binding between these two artefacts was intended to be strong, even though it has been reduced recently: from Java 8, the Java language hasn't been defined anymore as a core element of the Java platform. 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 the field of 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 presented the language to the public by demonstrating its power 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 at the time and didn't succeed. Luckily, it was successfully employed in the newborn 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.

The main goal of the project was to offer to developers a way of achieving the so called WORA principle (*write once, run anywhere)* which means that a piece of code can be executed on every machine, or, just the same, that developers don't have to take care of the machine where their code will be executed. This is obtained with the already mentioned Java platform, 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 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 nowadays.

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

The language evolved impressively during time: as an example, the original applets, which had so much importance at early stages, are now deprecated. The current (January, 2020) supported versions are Java 13, released in October 2019, and Java 11, released on September 25, 2018.

```
public class HelloWorld {
        public static void main(String[] args){
                System.out.println("Hello world!");
        }
}
```

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), functions (*methods)* and constant attributes, which all together 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 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 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 where the execution flow starts from when running a program. The developer can specify the *main* method everywhere, but only one main per each 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. These classes can't be changed by the developer. Java provides also some primitive types: see below.

When defining a new class, the developer may specify how to create objects of that class by a data 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 for example is null for user defined types).

An object of a class (i.e., an instance of a user defined type) is then 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, 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 is 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 recursive list

The code in snippet 2 is relevant also to spot another Java feature: *generics*. The use of generics is also known as *parametric polymorphism* in Java. 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.

A main feature of Java is *subclassing.* When a class is a subclass of another class (the superclass), every object of the subclass inherits all the variables and methods of the superclass. This is also called in the Java context *inheritance,* and it's one of the three core principles of OO programming (the other two are encapsulation and polymorphism). Inheritance is extremely powerful because it allows the reusability of implementations, thus saving time and reducing the code dimension.

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.

Inheritance gives rise to a subtype relation between classes, thus allowing for polymorphism. 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. We can even reassign the variable to an object of another subclass. Every time we need an object of class List, we can provide an object of class ArrayList without changing the behaviour of the program: this is a subtype relation, and in particular it's the subset interpretation of the subtype relation.

In this way we have a variable with both a *static type (*the type defined in the declaration) and a (possibly varying) *dynamic type* (the actual type of the variable) which can differ, and still type safety is guaranteed. Indeed, the Java compiler checks whether the object is manipulated correctly based on the static type. Since all the methods and variables of the static type are always inherited by the subclass which is the actual class, there is guarantee that there will be no errors at runtime. Unfortunately, if we call a method which is declared by the dynamic type, but not by the static type, a compile error is raised, since the compiler cannot know in advance whose dynamic type the variable be during the execution.

We obtain then 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. This is done through the so called vtable. 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.

Inheritance and subtyping are two orthogonal concepts, and their relation is under discussion. When overriding, some care must be taken to make sure that an object of a subclass can safely substitute an object of a superclass: while it's true that the substitution will always be syntactically valid, it may affect the program behaviour and semantic correctness. To avoid this, a stronger notion of subtyping(*behavioural subtyping*) has been introduced.

At the core of the problem there is the concept of contract. When we define a method, its semantics is defined by a "contract" signed between the method itself and the client code which uses the method. This contract defines the preconditions (what the method asks for), the postconditions (what the method guarantees if the preconditions are met) of the method and the invariants (properties that always hold). When a class reimplements a method of a superclass, it must respect not only the method signature (i.e., the method name and parameters type), but also this contract, so that the user of the superclass can safely use an object of the subclass without knowing it. The same requirements apply to variables. This principle is called *The Liskov Substitution Principle*, and states that:

- preconditions cannot be strengthened in a subtype;
- postconditions cannot be weakened in a subtype;
- invariants of the supertype must be preserved in a subtype.

If the LSP applies, then the subclass relation is also a behavioural subtype relation. It's a good habit to always apply it in all the subclasses of the program.

In general, deciding whether the LSP applies for a subclass relation it's an undecidable problem. This paragraph isn't meant to be an extensive discussion of he problem of semantic type safety in subtyping, but merely an introduction. For more details, see the references.

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 specified. 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 pure subtyping. 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. When a class *implements* an interface, it must implement all the methods of that interface, otherwise it must be declared abstract. A class or an interface can implement more than one interface. Since an interface does not implement any method, it does not specify any behaviour, so all the classes implementing the interface are actually subtypes of the interface, since all the methods overridings fulfil the LSP requirements: they just preserve the method signature.

We can use this kind of polymorphism almost everywhere in our coee: for example, we can pass as actual parameter to a method an object of a subclass of the superclass of the formal parameter of the declaration of the method.

Another way Java provides polymorphism is overloading (also called ad hoc polymorphism): overloading means specifying a method with the same name as another method of the same class (or of a superclass, which is pretty the same due to inheritance), but with parameters of different types. In this way, we allow the function (method) to work on set of types, with a different implementation per each type. Overloading should not be confused with overriding: when doing method overloading, the binding is solved at compile time looking at the static types of the actual parameters, which are matched against the different definition of a method.

Regarding algebraic data types, the subclass relation can be used in Java to define the so-called sum types. To destruct the sum type, we can use the *instanceof* syntax, as in Snippet 4.

Java doesn't provide an explicit way of building product types: they must be wrapped as attributes inside another class.

```
Fruit fruit = ...;
if (fruit instanceof Apple )
          {
             System.out.println("it is an Apple");
          }
```

Snippet 4: instanceof in Java

Java allows to force explicitly the conversion of the static type of an object from a class to a subclass, provided that the dynamic type of the object belongs to the subclass (or to a subclass of

the subclass), with the *cast* syntax (also called downcasting). Snippet 5 provides an example of casting.

```
public class Fruit{}  // parent class
public class Apple extends Fruit{}  // child class

public static void main(String args[]) {
    // The following is an implicit upcast:
    Fruit parent = new Apple();
// Fruit is the static type
// Apple is the dynamic type
    // The following is a downcast. Here, it works since the variable
`parent` is
    // holding an instance of Apple:
    Apple child = (Apple)parent;

}
```

Snippet 5: casting in Java

The third Java principle is encapsulation: hiding the implementation ensures modularity of the code. As already seen, Java provides abstract classes and interfaces, which are a useful way of abstract from the implementation and ensure encapsulation. Another way is by specifying a method or an attribute as *private.* When an attribute is private, it can be accessed only by the methods of the class itself. More options about how a method or a variable can be accessed are provided.

Java allows the user to create separate execution threads. Every thread shares the address space of all the other threads. A thread is in practice just an implementation of the already provided Runnable interface. To handle concurrency, and allow for atomic operations, Java provides the *synchronized* modifier, which can be added to the signature of a method. Java associates an intrinsic lock to every object. This lock is also called *monitor*. When a synchronized method of an object is called, if another synchronized method of the same object is in execution, the task is suspended until the other synchronized method finishes its execution. A constructor can't be synchronized. Java allows also to specify some synchronized statements with the following syntax.

```
synchronized (Object whose lock I want to acquire) {}
```

`Wait()` and `notify()` are a more complex way to handle concurrency in Java. These two methods can be called only inside a synchronized block of code. Wait() suspends the current thread and releases the lock, while notify awakes a suspended task, if it exists. The synchronized keyword is not inherited by the subclasses.

With Java 5, more advanced concepts have been introduced, in the Java.util.concurrent library, which allows to create ad hoc "lock" objects and atomic objects. Other ways of handling concurrency have been introduced in the language (like STM), but these are the most common ones. Refer to Snippet 7 to see how a thread is launched with its own separated code but sharing the same memory of the main process. As a remainder, note that a thread can still have its own variables.

Recent versions of Java (Java 8, in particular) are not only a simple object-oriented language but have included also some support for functional programming. Due to that, Java is also called a multi-paradigm programming language. To allow for functional programming without interfering with other features of the language, Java has included a library (*java.util.function*) which contains functional interfaces. These functional interfaces provide target types for lambda expressions and methods references. Only the classes that implement these interfaces can have functions as first-class citizens and have function closures and partial application. As an example, see Snippet 6,

where the lambda expression *length* is introduces, and Snippet 7, where the example of a closure can be found. Before Java 8, local classes were a way to fake lambda expressions.

```
public static void main(String[] args) {
    Function<String, Integer> length = s -> s.length();

    System.out.println( length.apply("Hello, world!") ); // Will print
13.
}
```

Snippet 6: lambda expressions in Java

```
class CalculationWindow extends JFrame {
    private volatile int result;
    ...
    public void calculateInSeparateThread(final URI uri) {
        // The code () -> { /* code */ } is a closure.
        new Thread(() -> {
                calculate(uri);
                result = result + 10;
        }).start();
    }
}
```

Snippet 7: a simple closure executed by a Thread.

Java is a type safe language. Indeed, Java has a sophisticated way of handling exception and errors, via the Throwable library. The Throwable class contains as subclasses the Error class and the Exception class. All the errors and exceptions in Java are then of the same type. The Java mechanism of handling the errors is based on the *try catch* syntax, along with the *throws* syntax. 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, violation of memory limits...), while the Exception class consists of expected but irregular occurrences, which can be handled by the program (like for example the impossibility to open a file). The developer may extend the Exception class to introduce its own exceptions to describe undesired situation. Exceptions may contain information to solve the issue and continue the execution. The procedure is the following: a method can return a normal object of its return type or a special object of class Exception (or one of its subclasses). The method signature (i.e., declaration) must declare the types of Exceptions the method can throw via the throws syntax. When some code calls a method which throws some exceptions, it must handle then all the exceptions in a try catch block, otherwise it must add these to its own throws declaration. A developer may want to handle also RunTimeExceptions and Errors, but it's not forced to do so. If an exception is raised, if there exists a catch block which handles it, the code of this block is executed (if multiple ones are found, the first one in the stack), otherwise the program terminates. To raise explicitly an exception, the throw syntax is used. Since the exceptions extend the Throwable class, all the discussion about polymorphism applies here.

The *finally* syntax allows to specify some code which is executed both if some exception is raised (in this case, it's executed after the catch block) or no exception is raised.

Snippet 6 provides an example of how exceptions are handled in Java (with the use of pseudocode).

```
try {
    line = console.readLine();

    if (line.length() == 0) {
        throw new EmptyLineException("The line read from console was
empty!");
    }

    console.printLine("Hello %s!" % line);
    console.printLine("The program ran successfully.");
}
catch (EmptyLineException e) {
    console.printLine("Hello!");
}
catch (Exception e) {
    console.printLine("Error: " + e.message());
}
finally {
    console.printLine("The program is now terminating.");
}
```

Snippet 6: exceptions in Java

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 -2^{31} and $2^{31}-1$ (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 $2^{32}-1$. The *Integer* class allows indeed to do that;
- Long: 64-bit signed two's complement integer. It takes values between -2^{63} and $2^{63}-1$ (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 $2^{64}-1$. 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.

I's remarkable that the Java subtyping system for primitive types works in a completely different way, since it enforces the coercion interpretation of the subtyping relation: a type a is subtype of type b if every value of a can be coerced to a value of b in a unique way. The following coercions are valid in Java.

- byte -> short, int, long, float, double,
- short -> int, long, float, double
- int -> long, float, double
- long -> float, double
- float -> double
- char -> int, long, float, double

The following explicit casting operations are allowed, when possible and with loss of information:

- short -> byte, char
- char -> byte, short
- int -> byte, short, char
- long -> byte, short, char, int
- float -> byte, short, char, int, long
- double -> byte, short, char, int, float
- byte -> char

Strengths and weaknesses of the language

The biggest 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 newly written code. Maybe paradoxically, the fact that this language is so popular contributes to its popularity

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 boring by developers, so that even in the Java official Oracle Tutorial (for JDK 8) some cases are mentioned where developers make their exceptions subclasses of RunTimeException to avoid catching them 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, especially to point out the exact point in the code when the exception was raised.

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 strategy 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. The complex access level modifiers of private, public, protected, package- protected could result hard to understand, but provides a lot of possibilities to the programmer.

The Object Oriented paradigm Java enforces makes it simple to reason about the program, but it's remarkable that functional programming languages which allow only for pure functions are much safer than Java: from great power (Java objects and methods with side effects) comes great responsibility.

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.

Today Java is not primarly 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.

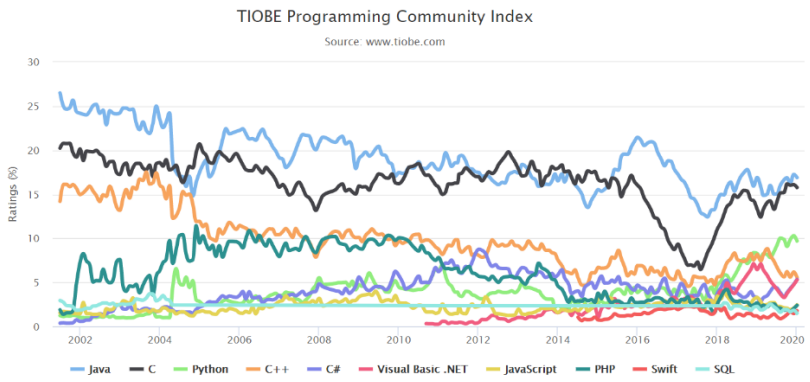

Figure 2: The TIOBE Programming Community index is an indicator of the popularity of programming languages. As the beginning of 2020, Java is the first one, strictly followed by C.

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