

# Programming Languages

---

Programming languages define the vocabulary used to encode our ideas about what a computer should do in a format that a computer can understand. Programming languages represent the second half of the programming problem; that is, they define a format that the computer can understand:

The process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer. [Jean-Michel Hoc and Anh Nguyen-Xuan]

Programming languages have two main goals:

- 1) To provide a format that is understandable to humans as they encode their intent for the desired behaviour of a program; and
- 2) To provide a format that can be unambiguously understood by automated tools enabling the language to be transformed into more compact executable representations.

These two constraints are often in opposition to one another. There is a tension between making a programming language easy to read and understand by developers and making it imprecise for machine analysis and transformation.

## Learning new languages

While languages come in all shapes and sizes (e.g., strongly typed, weakly typed, dynamically typed, statically typed, functional, object-oriented) at their core they each try to satisfy the two main goals listed above. Although learning a new programming language can seem like a

daunting task, languages all come from common pedigrees that make it easier to try to understand at a high level how the new language corresponds to your existing one. This [great image](#) shows the conceptual lineages of many popular programming languages and can be helpful to recognize how similar many languages are in practice.

Being able to quickly self-learn new languages is a fundamental skill for all software engineers. Language choices are infrequently made, which means the vast majority of teams you join will already be using a set language that you will not have control over choosing (and may even be an internal language that has not been released).

Consider the following three snippets:

## Java

```
public boolean isSplit(String input) {
    this.lastString = input;

    String[] parts = input.split(",");
    for (String s : parts) {
        System.out.println(s);
    }
    return parts.length > 0;
}
```

## TypeScript

```
public isSplit(input: string): boolean {
    this.lastString = input;

    let parts = input.split(",");
    for (var s of parts) {
        console.log(s);
    }
}
```

```
    }  
    return parts.length > 1;  
}
```

## JavaScript

```
isSplit(input) {  
    this.lastString = input;  
  
    let parts = input.split(",");  
    for (var s of parts) {  
        console.log(s);  
    }  
    return parts.length > 1;  
}
```

In these examples we can see that there are far more commonalities between these methods than differences. JavaScript is an untyped language, Java is a strongly typed language, and TypeScript augments JavaScript with type checking; even with these large underlying differences their syntax (and in this case even semantics) are nearly identical.

## Language properties

Syntax and semantics define the form and meaning of programming languages. The syntax of the language defines the grammar (tokens) and the legal order that these tokens can be used in valid programs. [EBNF](#) is often used to define the legal syntax of a language precisely. In contrast to syntax, the semantics of the language capture the actual meaning of the ordered tokens. Some statements can be syntactically valid while being semantically meaningless (e.g., `if (foo === foo)` will always be

true and has no semantic value, while still being syntactically correct).

Beyond syntax, developers frequently use small patterns, or idioms, that are common to a language. Examples for [JavaScript](#), [TypeScript](#), or [Java](#) are provided for reference. Following common idioms, along with code style (like those enforced by linters for any programming language), and code conventions make it easier for people to understand and read your code. This is particularly important for large code bases as it helps make the code more consistent, easing navigation and program understanding tasks.

There is also an important distinction to be made between the static representation of a system (aka its source code) and the dynamic representation of a system (aka its runtime state and behaviour). This distinction is important because it is often not obvious from the source code how the code will execute at runtime (for example due to different code paths being invoked due to polymorphism). Another way to think of this is during debugging: while you can pause execution on a single breakpoint and get a view of the program at that instant in time, the true dynamic behaviour of a program is an aggregation of *all* program states across all executions. This is one of the main challenges making understanding bug reports difficult.

## Language features

There are many programming languages, each of which have their own syntax (see this [language tree](#) for a visual representation). But syntax is not the most important differentiator between languages. Every language designer makes higher-level design decisions that provide more coarse-grained differences between them than their syntax. While there are many of these design decisions, here we will just discuss two of them.

# Interpreted vs. compiled

One of these design decisions is whether the language is interpreted or compiled. An interpreted language executes the code by starting at the top of the code listing and executing successive lines (much like how a person would read a book). For example, in an interpreted language the first listing below would work while the second would not (because the symbol `m1` has not been defined when it is first referenced).

```
// this will work because m1 is  
// declared before it is called  
  
declare m1() {  
    print 'in m1'  
}  
  
m1();
```

```
// this will not work because m1 is  
// not declared before it is called  
  
m1();  
  
declare m1() {  
    print 'in m1'  
}
```

In contrast, compiled languages are first analyzed and transformed by a compiler before they can be executed. This transformation can change the source code directly to machine code, or may emit some form of bytecode. This analysis step helps in several ways, it enables (this is not a complete list):

- The code to be analyzed for syntax problems before runtime.
- Type checking (if the language is statically typed).
- The emitted code to be optimized for runtime performance improvements.
- The code to reference elements that have not yet been defined (as the whole code can be analyzed in multiple passes by the compiler).

We already know that Java is not an interpreted language because we explicitly compile it (with `javac`). We can also see this because we can reference fields and methods before they are declared (e.g., `m1` before it is defined):

```
class MyClass {  
  
    public void action() {  
        m1();  
    }  
  
    private void m1() {  
        System.out.println('in m1');  
    }  
}
```

Similarly, JavaScript is also not an interpreted language (by definition above) as the following snippet also works:

```
m1();  
  
function m1() {  
    console.log('in m1');  
}
```

While JavaScript started as an interpreted language, all modern

JavaScript implementations use Just-in-time (JIT) compilation to transform the JavaScript code into bytecode which can then be evaluated. An interesting discussion about this can be found [here](#).

## Static vs. dynamic types

Another important language design question is whether the language uses static or dynamic types. In languages that are statically typed, *variables* are given types and a compiler checks that assignments to and usages of these variables are do not violate the type declaration. For example, in Java:

```
int age; // age is declared to be an int
age = 10; // ok, 10 is an int
age = 100; // ok, 100 is an int
age = false; // !ok, false is not an int
age = '10'; // !ok, '10' is a string, not an int
```

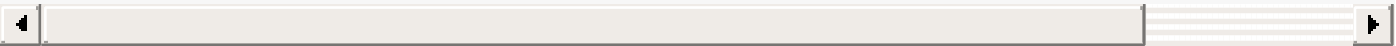
In contrast, languages that use dynamic types, *values* rather than variables have types. For example, in JavaScript:

```
var age; // age does not have a type
age = 10; // ok, 10 is a number, so age is now a number
age = 100; // ok, age is still a number
age = false; // ok, false is a boolean so age is now a boolean
age = '10'; // ok, '10' is a string so age is now a string
```

Types are enforced by a compiler that checks source code to ensure that variables are only used in type-compatible ways. Dynamically typed languages do not perform type checking, this can cause problems at runtime. For example in the JavaScript below the first call to `toFixed` works because `toFixed` is a method valid for all numbers. But the

second time it is called it fails because `v` is now a string and strings do not have a `toFixed` method. It is important to remember that the failure below will only be detected when the code is run, which may be fine for a 10 line program but will be more problematic for a million line program.

```
var v = 1.11111; // v is a number
v = v.toFixed(1); // v now the string '1.1'
v = v.toFixed(1); // will fail with 'TypeError: v.toFixed is not a
```



TypeScript extends JavaScript by adding static type declarations that are checked by a compiler. For example,

```
var age: number; // age is a number
age = 10; // ok, 10 is a number
age = 100; // ok, 100 is a number
age = false; // !ok, false is a boolean, not a number
age = '10'; // !ok, '10' is a string, not a number
```

TypeScript uses type inference to figure out the types in your code from how they are used, rather than forcing you to explicitly define them.

```
var age; // age does not have a type
age = 10; // ok, 10 is a number so age is now a number
// age has now been given the type number, this cannot be changed
age = 100; // ok, 100 is a number
age = false; // !ok, false is a boolean, not a number
age = '10'; // !ok, '10' is a string, not a number
```

TypeScript also gives you the ability to turn off type checking for specific variables with the `any` type. This tells the compiler not to emit type errors for that variable so you can change the type of a variable



dynamically.

```
var age: any; // age can take any type
age = 10; // ok
age = 100; // ok
age = false; // ok
age = '10'; // ok
```

## References

- Gerald Sussman's [role of programming](#) talk.
- A practical [article](#) on some TypeScript benefits in practice.
- Why you should learn [a new language](#) every year.
- Languages are all related. [This article](#) reenforces how some ideas propagate (without justification) through programming languages.
- TypeScript vs JavaScript from an [angular developer](#).
- The [functional features](#) in JavaScript are the same as TypeScript.
- Article about how TypeScript is [gradually typed](#).
- Discussion of the [functional features](#) of JavaScript.



[Reid Holmes](#)