# [**Scripting Language vs Programming Language**](https://stackoverflow.com/questions/17253545/scripting-language-vs-programming-language)

Scripting languages are programming languages that don't require an explicit compilation step.

For example,

in the normal case, you have to compile a C program before you can run it. But in the normal case, you don't have to compile a JavaScript program before you run it. So JavaScript is sometimes called a "scripting" language.

This line is getting more and more blurry since compilation can be so fast with modern hardware and modern compilation techniques.

For instance, V8, the JavaScript engine in Google Chrome and used a lot outside of the browser as well, actually compiles the JavaScript code on the fly into machine code, rather than interpreting it. (In fact, V8's an optimizing two-phase compiler.)

Also note that whether a language is a "scripting" language or not can be more about the environment than the language. There's no reason you can't write a C interpreter and use it as a scripting language (and people have). There's also no reason you can't compile JavaScript to machine code and store that in an executable file (and people have). The language Ruby is a good example of this: The original implementation was entirely interpreted (a "scripting" language), but there are now multiple compilers for it.

Some examples of "scripting" languages (e.g., languages that are *traditionally* used without an explicit compilation step):

* Lua
* JavaScript
* VBScript and VBA
* Perl

And a small smattering of ones *traditionally* used with an explicit compilation step:

* C
* C++
* D
* Java *(but note that Java is compiled to bytecode, which is then interpreted and/or recompiled at runtime)*
* Pascal

...and then you have things like Python that sit in both camps: Python is widely used without a compilation step, but the main implementation (CPython) does that by compiling to bytecode on-the-fly and then running the bytecode in a VM, and it *can* write that bytecode out to files (.pyc, .pyo) for use without recompiling.

That's just a *very* few, if you do some research you can find a lot more.

# [**What is the difference between statically typed and dynamically typed languages**](https://stackoverflow.com/questions/1517582/what-is-the-difference-between-statically-typed-and-dynamically-typed-languages)

### Statically typed languages

A language is statically typed if the type of a variable is known at compile time. For some languages this means that you as the programmer must specify what type each variable is; other languages (e.g.: Java, C, C++) offer some form of *type inference*, the capability of the type system to deduce the type of a variable (e.g.: OCaml, Haskell, Scala, Kotlin).

The main advantage here is that all kinds of checking can be done by the compiler, and therefore a lot of trivial bugs are caught at a very early stage.

Examples: C, C++, Java, Rust, Go, Scala

### Dynamically typed languages

A language is dynamically typed if the type is associated with run-time values, and not named variables/fields/etc. This means that you as a programmer can write a little quicker because you do not have to specify types every time (unless using a statically-typed language with *type inference*).

Examples: Perl, Ruby, Python, PHP, JavaScript, Erlang

Most scripting languages have this feature as there is no compiler to do static type-checking anyway, but you may find yourself searching for a bug that is due to the interpreter misinterpreting the type of a variable. Luckily, scripts tend to be small so bugs have not so many places to hide.

Most dynamically typed languages do allow you to provide type information, but do not require it. One language that is currently being developed, [Rascal](http://www.rascal-mpl.org/), takes a hybrid approach allowing dynamic typing within functions but enforcing static typing for the function signature.

**Programming Paradigm**

A **programming paradigm** is a style, or “way,” of programming.

Some languages make it easy to write in some paradigms but not others.

Some common paradigm:

* **Imperative**: Programming with an explicit sequence of commands that update state.
* **Declarative**: Programming by specifying the result you want, not how to get it.
* **Structured**: Programming with clean, goto-free, nested control structures.
* **Procedural**: Imperative programming with procedure calls.
* **Functional** (Applicative): Programming with function calls that avoid any global state.
* **Function-Level** (Combinator): Programming with no variables at all.
* **Object-Oriented**: Programming by defining objects that send messages to each other. Objects have their own internal (encapsulated) state and public interfaces. Object orientation can be:
  + **Class-based**: Objects get state and behavior based on membership in a class.
  + **Prototype-based**: Objects get behavior from a prototype object.
* **Event-Driven**: Programming with emitters and listeners of asynchronous actions.
* **Flow-Driven**: Programming processes communicating with each other over predefined channels.
* **Logic** (Rule-based): Programming by specifying a set of facts and rules. An engine infers the answers to questions.
* **Constraint**: Programming by specifying a set of constraints. An engine finds the values that meet the constraints.
* **Aspect-Oriented**: Programming cross-cutting concerns applied transparently.
* **Reflective**: Programming by manipulating the program elements themselves.
* **Array**: Programming with powerful array operators that usually make loops unnecessary.

Paradigms are **not meant to be mutually exclusive**; a single program can feature multiple paradigms!