**[PVM vs. JVM - What the difference?](https://morioh.com/p/01d84b0bab10)**

Evidently, Humans can understand anything through natural languages, but a computer doesn’t. The computer needs a translator to convert the languages written in the human-readable form to the computer-readable form.

Compiler and interpreter are the types of a language translator. What is Language translator? This question might be arising in your mind.

A language translator is a software which translates the programs from a source language that are in human-readable form into an equivalent program in an object language. The source language is generally a high-level programming language, and the object language is typically the machine language of an actual computer.

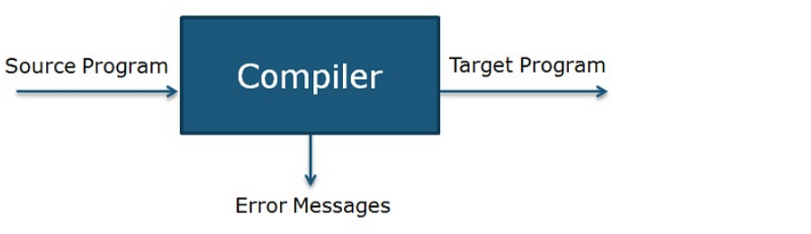
there are two types of Translators

1. Compiler (invented by Grace Hopper)

2. Interpreter (invented by Steve Russel)

**Compiler**

* A ***compiler*** is a program that reads a program written in the high-level language and converts it into the machine or low-level language and reports the errors present in the program. It converts the entire source code in one go or could take multiple passes to do so, but at last, the user gets the compiled code which is ready to execute. The returned **target code** file can be run with many different inputs, over and over. the compiler doesn’t need to be around for any subsequent reruns.



Usually, the source form is a higher-level language than the destination form, such as when converting from C to machine code. But converting from JavaScript 8 to JavaScript 5 is also a kind of compiling.

Compiler operates on phases; various stages can be grouped into two parts that are:

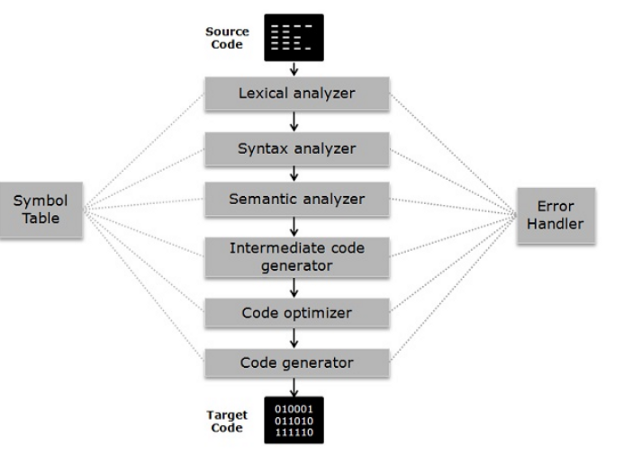
* **Analysis Phase** of the compiler is also referred to as the front end in which program is divided into fundamental constituent parts and checks grammar, semantic and syntax of the code after which intermediate code is generated. Analysis phase includes lexical analyzer, semantic analyzer, and syntax analyzer.
* **Synthesis phase** of the compiler is also known as the back end in which intermediate code is optimized, and the target code is generated. Synthesis phase includes code optimizer and code generator.

Grace Hopper’s work on the first compiler laid the groundwork for another translator that came into existence a few years later: the interpreter.

**PHASES OF COMPILER**

Now let’s understand the working of each stage in detail.

1. **Lexical Analyzer**: It scans the code as a stream of characters, groups the sequence of characters into lexemes and outputs a sequence of tokens with reference to the programming language.
2. **Syntax Analyzer**: In this phase, the tokens that are generated in the previous stage are checked against the grammar of programming language, whether the expressions are syntactically correct or not. It makes parse trees for doing so.
3. **Semantic Analyzer**: It verifies whether the expressions and statements generated in the previous phase follow the rule of programming language or not and it creates annotated parse trees.
4. **Intermediate code generator**: It generates an equivalent intermediate code of the source code. There are many representations of intermediate code, but TAC (Three Address Code) is the used most widely.
5. **Code Optimizer**: It improves time and space requirement of the program. For doing so, it eliminates the redundant code present in the program.
6. **Code generator**: This is the final phase of the compiler in which target code for a particular machine is generated. It performs operations like memory management, Register assignment, and machine-specific optimization.



The **symbol table** is somewhat a data structure which manages the identifiers along with the relevant type of data it is storing. **Error Handler** detect, report, correct the errors encountering in between the different phases of a compiler.

**Definition of Interpreter**

The interpreter is an alternative for implementing a programming language and does the same work as a compiler. Interpreter performs **lexing**, **parsing** and **type checking** similar to a compiler. But interpreter processes syntax tree directly to access expressions and execute statement rather than generating code from the syntax tree.

An interpreter may require processing the same syntax tree more than once that is the reason why interpretation is comparatively slower than executing the compiled program.

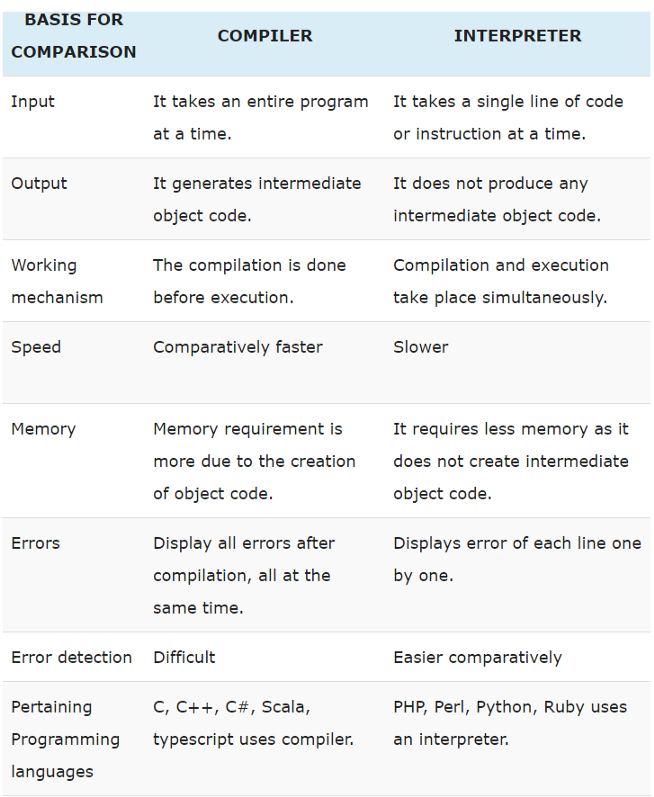
A compiler is comparatively faster than Interpreter as the compiler take the whole program at one go whereas interpreters compile each line of code after the other.

**What is the Need of Interpreter if we already had Compiler?**

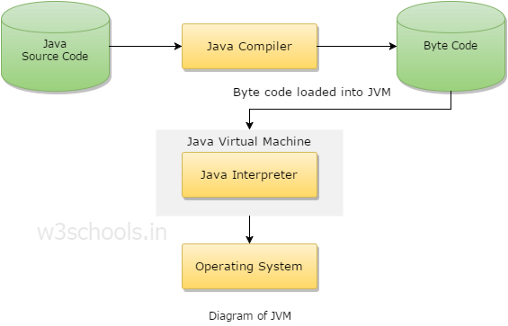
The real need of interpreter comes where compiler fails to satisfy the software development needs. The compiler is a very powerful tool for developing programs in a high-level language. However, there are several demerits associated with the compiler. If the source code is huge in size, then it might take hours to compile the source code. Which will significantly **increase the compilation duration**. Here interpreter comes handy and can cut this huge compilation duration. Interpreters are designed to translate single instruction at a time and execute them immediately.

An interpreter also comes handy when you frequently update the source code. As if it takes 5 minutes to compile the entire source code. If you have updated it 5 times. Then the total compile time would be 25 minutes which is significantly big. However, if we use an interpreter we can reduce this compile time. we can conclude the following points from above :

1. The compiler requires more memory than interpreter because of the generation of object code of the whole program at once.
2. Compiler presents all errors concurrently, and it’s difficult to detect the errors in contrast interpreter display errors of each statement one by one, and it’s easier to detect errors.
3. In compiler when an error occurs in the program, it stops its translation and after removing error whole program is translated again. On the contrary, when an error takes place in the interpreter, it prevents its translation and after removing the error, translation resumes.
4. Intermediate code or target code is generated in case of a compiler. As against interpreter doesn’t create intermediate code.
5. when we use interpretation, we can distribute our source code directly, rather than worrying about platform-specific executables, or thinking about how we’ll go about compiling binaries for everyone to use. However, in this case, we’ll need the consumer of our program to download an interpreter — which often actually just comes along with a language — and be sure that the interpreter exists on their machine(s). Once they have the interpreter, they can see our original source code, take that code, and then rely on their own version of the interpreter to run it locally. In this scenario, we rely on the interpreter to make the compatible on all platforms, and we, as programmers and consumers, don’t need to think about it. Also, if something goes wrong (either something in the source that we wrote, or something relating to their own platform), the consumer of our code can figure out what the issue is much more easily than with a compiled file. The interpreter will make it easy to debug any problems, regardless of who is running our code.



**Java uses the compiler as well as Interpreter. How?**



The Compiler of java called as **javac** converts source code into an Intermediate file known as **Bytecode** file. The Bytecode file is unique for all types of OS means bytecode is platform-independent. The Interpreter of java (**java**) converts Bytecode into the specific OS-compatible machine code. This code will vary according to OS.

**What is Bytecode file? why it is called a Bytecode file?**

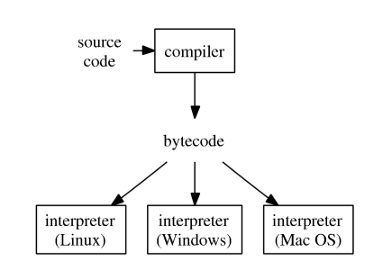
So, the answer is Bytecode is just like Shorthand language that stores each keyword of java as a sign. and each sign takes 1 byte of memory in RAM. Hence the name called Bytecode file.

**Why we use both — Compiler as well as Interpreter in Java(Need of Separate compiler and Interpreter)**

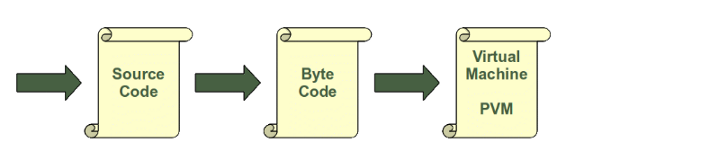
At the time of C and C++, only Compiler was there that converts source code into specific OS machine code. The machine code was OS dependent that varied from OS to OS.

The problem the programmer was facing that They had to design a different compiler for different Operating System that is too difficult, Too time-consuming and much Costly.

Then Sun Microsystem take the whole Initiative by making a **Unique compiler** that produces Platform Independent Bytecodes and the **Specific JVM**(Java Virtual Machine)or **Interpreter** that converts bytecodes into machine code that will vary from OS to OS.



**Similarly, Python uses the interpreter as well as the compiler. How?**

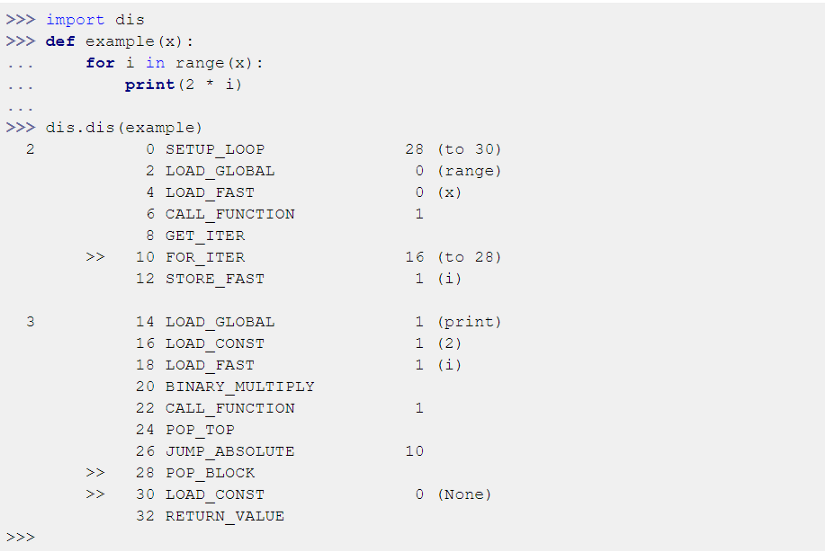


In Python, the source is compiled into a much simpler form called [bytecode](https://en.wikipedia.org/wiki/Bytecode). These are instructions similar in spirit to CPU instructions, but instead of being executed by the CPU, they are executed by a software called a virtual machine. (These are not VM’s that emulate entire operating systems, just a simplified CPU execution environment.) .This is a similar approach to the one taken by Java. There is even a way of translating Python programs into Java byte code for the Java Virtual Machine (JVM). This can be achieved with Jython.

but in python we don’t have to compile **program** manually it is done by python for us automatically.

Here’s an example of a short Python function, and its bytecode:

The [**dis**](https://docs.python.org/3/library/dis.html) module in the Python standard library is the disassembler that can show you Python bytecode. It’s also the best (but not great) documentation for the bytecode itself.



another way to compile this is as follow:

This is image title

it generates compiled file named **python\_source\_file.cpython-36.pyc** inside the **pycache** folder in the current directory.

You can also automatically compile all Python files using the **compileall** module.

This is image title

The compilation is hidden from the user for a good reason. Some newbies to Python wonder sometimes where these ominous files with the .pyc suffix might come from. ***If Python has write-access for the directory where the Python program resides, it will store the compiled byte code in a file that ends with a .pyc suffix. If Python has no write access, the program will work anyway. The byte code will be produced but discarded when the program exits.***  
Whenever a Python program is called, Python will check, if a compiled version with the .pyc suffix exists. This file has to be newer than the file with the ***.py*** suffix. If such a file exists, Python will load the byte code, which will speed up the start-up time of the script. If there exists no byte code version, Python will create the byte code before it starts the execution of the program. Execution of a Python program means execution of the byte code on the Python Virtual Machine (PVM).

An important aspect of Python’s compilation to bytecode is that it’s entirely implicit. You never invoke a compiler, you simply run **a .py** file. The Python implementation compiles the files as needed. This is different than Java, for example, where you have to run the Java compiler to turn Java source into compiled class files. For this reason, Java is often called a compiled language, while Python is called an interpreted language. But both compile to bytecode, and then both execute the bytecode with a software implementation of a virtual machine.

Another important Python feature is its **interactive prompt**. You can type Python statements and have them immediately executed. This interactivity is usually missing in “compiled” languages, but even at the Python interactive prompt, your Python is compiled to bytecode, and then the bytecode is executed. This immediate execution and Python’s lack of an explicit compile step are why people call the Python executable “the Python interpreter.”

This shows just how flimsy the words “interpreted” and “compiled” can be. Like most adjectives applied to programming languages, they are thrown around as if they were black-and-white distinctions, but the reality is much subtler and complex.

Finally, how your program gets executed isn’t a characteristic of the language at all: it’s about language implementation. I’ve been talking here about Python, but this has really been a description of CPython, the usual implementation of Python, so-named because it is written in C. [PyPy](https://pypy.org/) is another implementation, using a JIT compiler to run code much faster than CPython can.

**But why Python needs both a compiler and an interpreter?**

***Speed***. Strict interpretation is slow. Virtually every “interpreted” language actually compiles the source code into some sort of internal representation so that it doesn’t have to repeatedly parse the code. In python’s case it saves this internal representation to disk so that it can skip the parsing/compiling process next time it needs the code.

**Is .pyc file (compiled bytecode) is platform independent?**

Compiled Python bytecode files are architecture-independent, but VM-dependent. A .pyc file will only work on a specific set of Python versions determined by the magic number stored in the file.

**Conclusion**

is Python compiled? Yes. Is Python interpreted? Yes. Sorry, the world is complicated… Python as a programming language has no saying about if it’s an **compiled** or **interpreted** programming language, only the implementation of it. The terms interpreted or compiled is not a property of the language but a property of the implementation. Python program runs directly from the **source code** . so, Python will fall under byte code interpreted. The .py source code is first compiled to byte code as .pyc. This byte code can be interpreted (official CPython), or JIT compiled (PyPy).