ICS111 Home

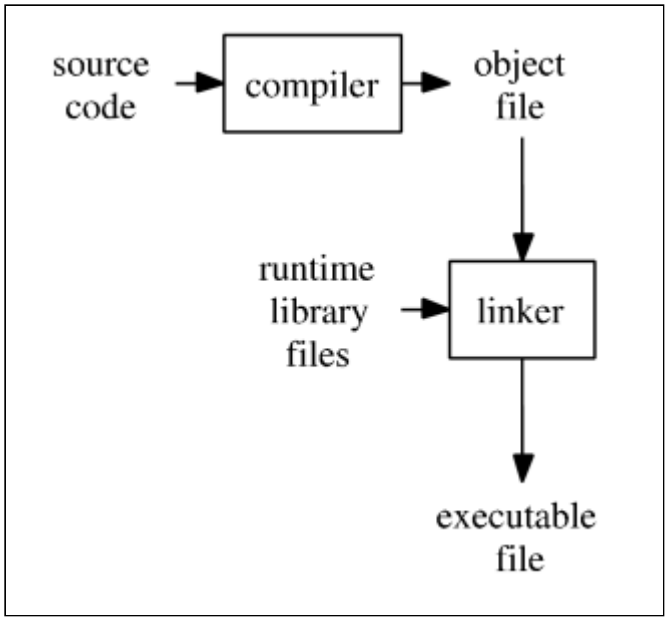
The Programming Process

Computers do not understand human languages. In fact, at the lowest level, computers only understand sequences of numbers that represent operational codes (op codes for short). On the other hand, it would be very difficult for humans to write programs in terms of op codes. Therefore, programming languages were invented to make it easier for humans to write computer programs.

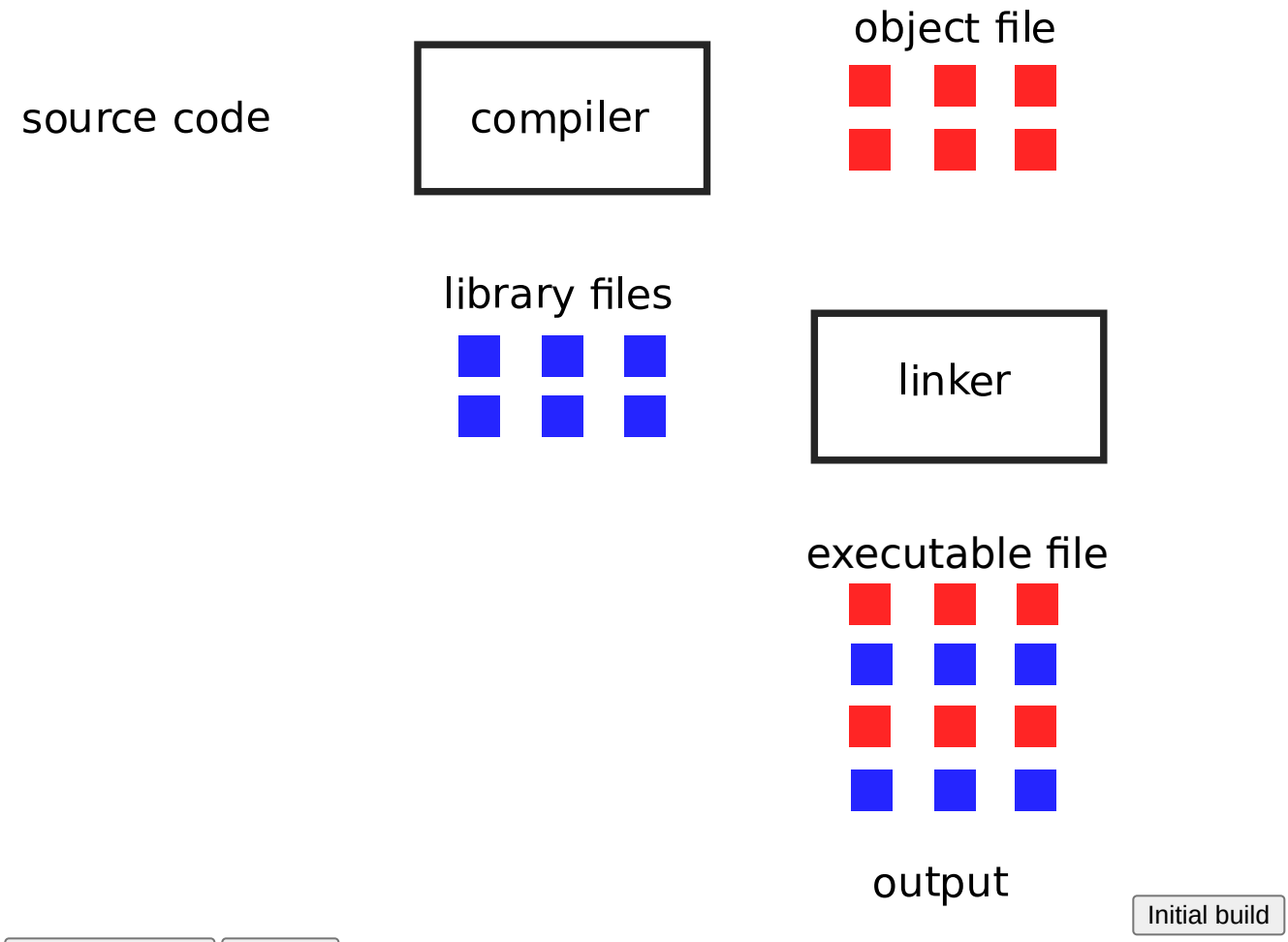
Programming languages are for humans to read and understand. The program (source code) must be translated into machine language so that the computer can execute the program (as the computer only understands machine language). The way that this translation occurs depends on whether the programming language is a compiled language or an interpreted language.

Compiled languages (e.g. C, C++)

The following illustrates the programming process for a compiled programming language.



A compiler takes the program code (source code) and converts the source code to a machine language module (called an object file). Another specialized program, called a linker, combines this object file with other previously compiled object files (in particular run-time modules) to create an executable file. This process is diagrammed below. Click **Initial build** to see an animation of how the executable is created. Click **Run executable** to simulate the running of an already created executable file. Click **Rebuild** to simulate rebuilding of the executable file.

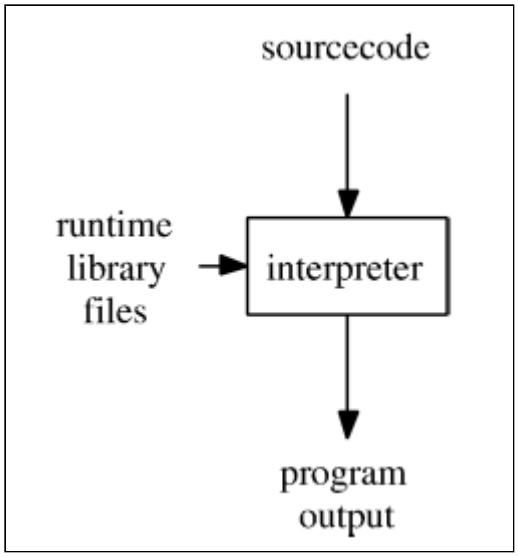


So, for a compiled language the conversion from source code to machine executable code takes place before the program is run. This is a very different process from what takes place for an interpreted programming language.

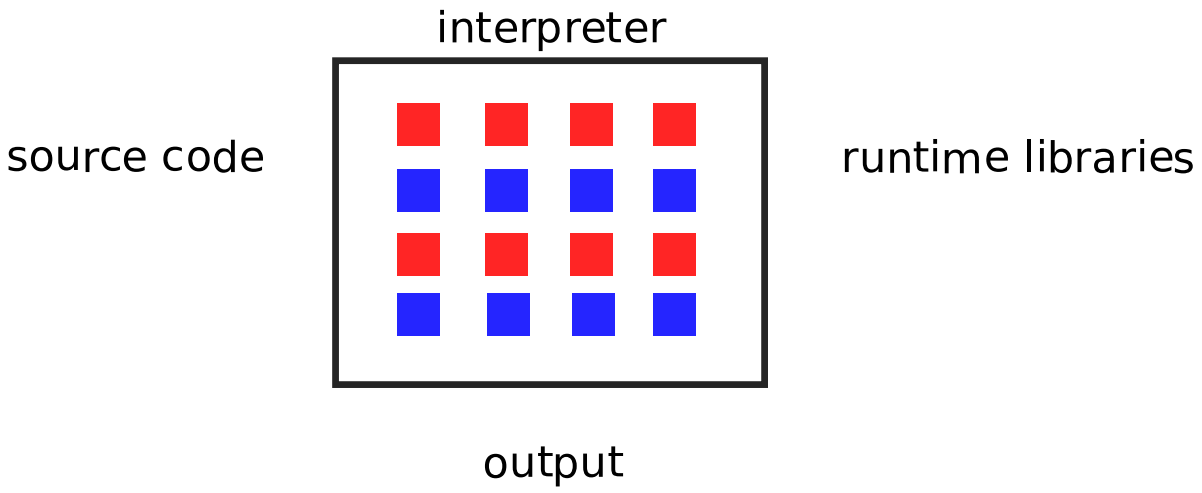
This is somewhat simplified as many modern programs that are created using compiled languages makes use of dynamic linked libraries or shared libraries. Therefore, the executable file may require these dynamic linked libraries (Windows) or shared libraries (Linux, Unix) to run.

Interpreted programming languages (e.g. Python, Perl)

The process is different for an interpreted language. Instead of translating the source code into machine language before the executable file is created, an interpreter converts the source code into machine language at the same time the program runs. This is illustrated below:



Interpreted languages use a special program called an *interpreter* that converts the source code, combines with runtime libraries, and executes the resulting machine instructions all during runtime. Unlike a compiled language, there is no precompiled program to run. The conversion process and combination with runtime libraries takes place every time an interpreted language program is run. This is why programs written in compiled languages tend to run faster than comparable programs written in interpreted languages. Click **Start** to run the simulation of an interpreted program. Click **Restart** if you want to run the simulation again.

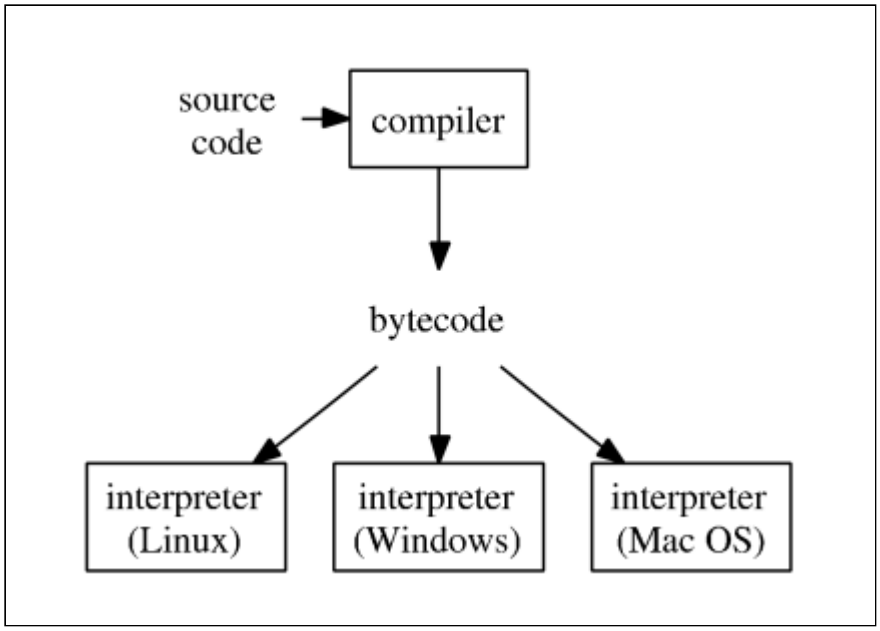


Each time an interpreted program is run, the interpreter must convert source code into machine code and also pull in the runtime libraries. This conversion process makes the program run slower than a comparable program written in a compiled language.

Because an interpreter performs the conversion from source to machine language during the running of the program, interpreted languages usually result in programs that execute more slowly than compiled programs. But what is often gained in return is that interpreted languages are often platform independent because a different interpreter can be used for each different operating system.

And now for something different ... Java

The Java programming language does not fit into either the compiled language or interpreted language models. This is illustrated in the figure below.

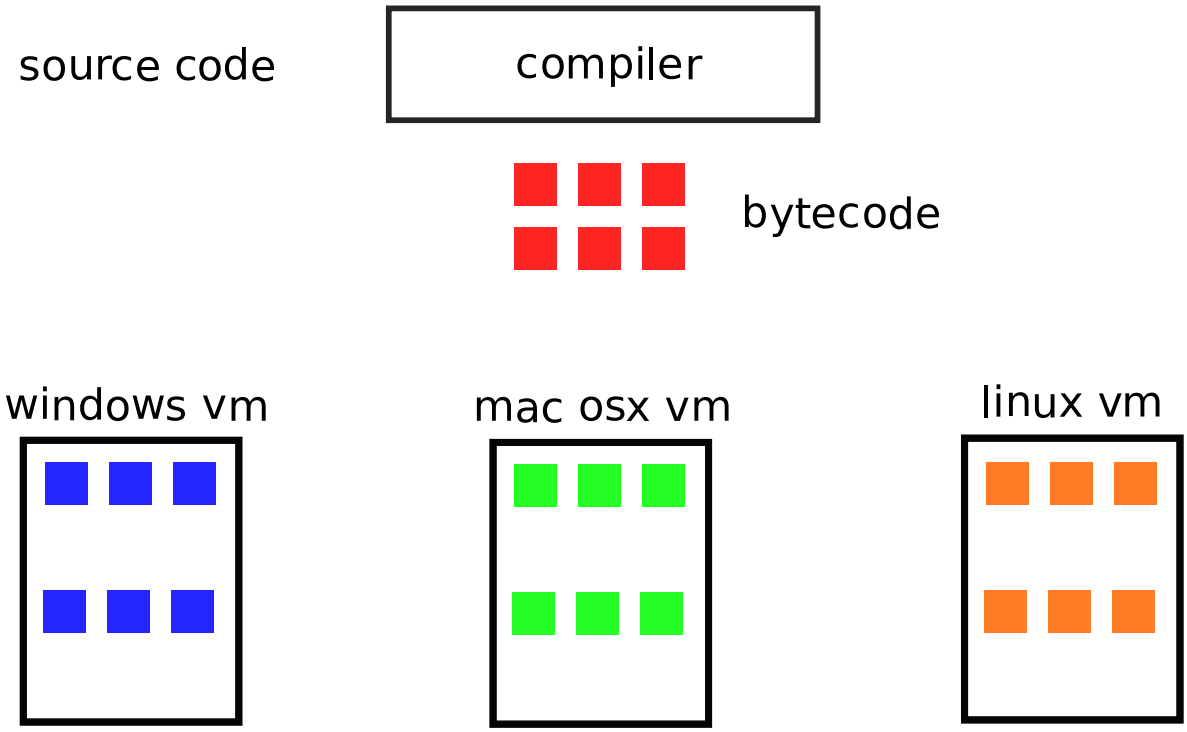


The Java compiler (javac) converts the source code into bytecode. Bytecode is a kind of average machine language. This bytecode file (.class file) can be run on any operating system by using the Java interpreter (java) for that platform. The interpreter is referred to as a Virtual Machine. Thus, Java is an example of a Virtual Machine programming language.

Virtual machine languages were created to be a compromise between compiled and interpreted languages. Under ideal conditions, virtual machine language programs run closer in speed to compiled language programs but have the platform indepency of interpreted language programs.

Virtual machine languages makes use of *both* a compiler and an interpreter. The compiler converts the source code into a kind of average machine language. In Java, this average machine language is called **bytecode**. In Visual Studio.NET languages, this average machine language is called MSIL (Microsoft Intermediate Language). (To keep the discussion on this page simpler, this compiled code will be referred to generically as bytecode from this point on.) The interpreter for virtual machine languages is a special program that provides the runtime libraries for the given operating system. That means that there is a different virtual machine interpreter for all of the supported operating systems.

The way that virtual machine programming languages get some of the speed of compiled languages is that the source code is run through the compiler to create the bytecode. That conversion takes place before the program is ever run. The way that virtual machine languages gain their portability (platform independence) is by having a different interpreter for each supported operating system. This interpreter ties in the correct runtime libraries for each different operating system. The compiled bytecode is an average machine language that will work without changes with any of the virtual machine interpreters for that language. This process is illustrated next. We have a compiler that converts the source code into bytecode. This can be simulated by clicking on the **Compile** button. Once the bytecode has been created, that same bytecode can be used without any changes on any operating system that has a virtual machine interpreter for the programming language. Note that each of the virtual machine interpreters have different runtime library code, because each operating system has different runtime libraries. This is how the virtual machine language gets around platform dependency problems. Click **Run Windows**, **Run Mac OSX** or **Run Linux** to simulate running the program on any of those operating systems.



Compile | Run Windows | Run Mac OSX | Run Linux

Once again, note that the bytecode does not need to be recompiled to run on any of the different operating systems. The only reason to recompile a program is if you changed the source code.

Hopefully, you can see how virtual machine language programs will have better performance than interpreted language programs. The virtual machine languages convert the source code to an average machine code before the program is ever run. Virtual machine languages don't quite match the performance of compiled languages because the bytecode still has to be loaded by the virtual machine before running.

Details of the Java programming process

The source code for a Java program is a text file that ends in **".java"**. Suppose you typed out the following file, **"Hello.java"**.

```
1 class Hello {
2    public static void main(String[] args) {
3       System.out.println("Hello");
4    }
5 }
```

To compile this program, you would type the following at the command line:

```
javac Hello.java
```

The Java compiler is named **javac**. The **javac** program is unique in that it does not produce actual machine code. Instead it produces something called bytecode. Unlike machine code, bytecode is not platform specific. The bytecode produced on a Windows machine is the same bytecode that is produced on a Linux machine. This means that the bytecode can be run (without recompiling) on any platform that has a Java interpreter.

If the compilation into bytecode is successful, the bytecode will be contained in a file called **"Hello.class"** is created. To run this bytecode, the Java interpreter is invoked in the following way.

```
java Hello
```

Note the name of the Java interpreter is **java**. Also note that you do not include the **.class** at the end of the filename when invoking the interpreter. By default, the .class file is created in the same directory as the directory you are running the compiler from.

Programming tip

At this point, one of the best ways to make progress in Java programming is to take a program that works and purposely introduce errors in the source code. This will help you to start recognizing how the compiler reports the various kinds of errors. For example, try the following:

- Remove the semicolon at the end of a statement.
- Remove the right curly brace at the end of a block.
- Add an extra left curly brace just before the beginning of a block.
- Misspell the word main. The main method marks the starting point of the program.

When the error is reported, take note of the location of the error that the compiler reports. As you will see, the line that the compiler points to as having the error may not be the actual line the error occurs on.

ICS111 Home