Compiler

Your Java program, in a bunch of `.java` files like `Animal.java` and `A2.java`, cannot be executed or run in that form. Another program is needed to put it in a form that can be executed. That program is called a *compiler*.

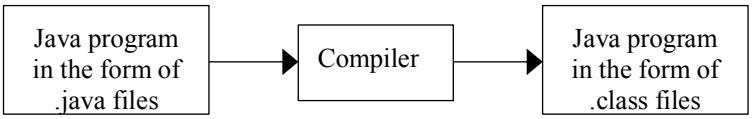
The Java compiler has two tasks:

1. Check that the program is syntactically correct —it follows the grammar rules of the Java Language Specification. For example, the assignment statement to the right is incorrect because a **`float`** value may not be stored in an **`int`** variable, and the expression under it is incorrect because it is missing a right parenthesis.[1]

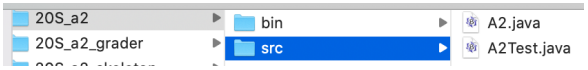
```
int k= 2.3;
3* (2 + 4
```

2. If the program is syntactically correct, translate into a form that can be executed: Translate each `.java` file, say a file `C.java`, into the machine language and store it in a corresponding file, `C.class`. These `.class` files are what can be executed, or run.

The diagram to the right depicts the process. Of course, the compiler produces a `.class` file only if the `.java` file is syntactically correct.



Where are the `.class` files placed? Below to the left, we show the contents of the hard drive on our computer for an Eclipse project named `20S_a2`. Directory `20S_a2` contains two subdirectories. Subdirectory `src` has been selected, and it contains two `.java`. files. To the right, we show the same directory except that subdirectory `bin` has been selected; in it are the corresponding `.class` files. Spend some time looking at the hard drive on your own computer to see the `.java` and `.class` files.



The process of compiling the program happens so fast that you don't notice it. In fact, as you edit lines in a `.java` file in Eclipse, the compiler is continually checking for correctness. Eclipse tells you if there is a syntax error. For example, to the right, we show part of a `.java` file that is being edited in the Eclipse window. The two red X's —actually white X's in a red circle— tell you that those lines have syntax errors in them. Hover your mouse over a red X and a small window opens, explaining the error.

```
94        int k= 2.3;
95        return 3* (2 + 4;
96    }
```

Thus, if you see a red X, you know the `.java` file that is being edited can't be compiled, or translated, into the machine language. But if there are no red X's, then the compiler that Eclipse uses has already compiled the class into a `.class` file. Oooh! It's fast.

JVM: The Java Virtual Machine

If your main goal in reading this document is to understand the word *compiler*, you can stop reading. If you want to learn more about executing the .class files, read on.

When Java was first developed, the designers wanted a language (and compiler, etc.) that would be *platform independent* —a Java program compiled on one computer could be executed on any other computer, even one with a different architecture. To achieve this, they designed a *virtual* machine —a physical, hardware version of it does not

---

[1] Here is one definition of syntax of Java: The syntax of the Java programming language is the set of rules defining how a Java program is written. The two examples of an illegal assignment statement and expression certainly don't conform to those rules.

However, the field sometimes considers as syntax only properties that are expressed in a "formal grammar", and everything else is "static semantics". For example, some would say that **int** k= 2.0; is illegal according to the static semantics rules.

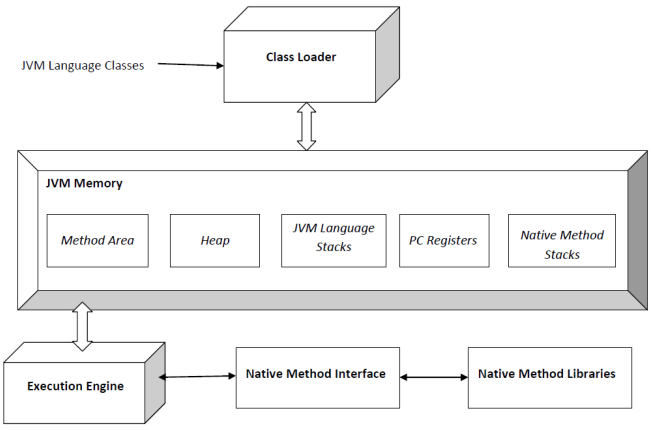
We will continue to use the words *syntax* and *semantics* as they are intended to be used: (1) syntax deals with how a program is written, (2) semantics deals with how a program is executed.

exist— called *JVM*. The instruction set of this virtual machine is called *Java bytecode*. A `.class` file consists of Java bytecode instructions. A program called an *interpreter* then reads the Java bytecode and executes it.

Other languages at the time were not platform independent. For example, a program written in the programming language C could give different results when executed on computers with different architectures.

Here are more details on the process of executing a Java bytecode program. A program called the *Class Loader* reads the .class files and prepares the computer memory. As you can see in the diagram below, memory is organized into

1. Method area: contains the actual instructions to be executed for the methods in the classes
2. Heap: Space where all created objects and arrays are stored.
3. JVM Language Stacks: For example, each thread of execution has its own call stack —the stack of frames for method calls that have not completed.
4. PC Registers: These are places where, for example, the operands of an addition and the result of an addition are stored.
5. Native Method Stacks: Call stacks for *native* methods —methods that are written in the language of the computer on which the program is being run.



From https://en.wikipedia.org/wiki/Java_virtual_machine

A second task of the Class Loader is to perform various checks on the .class files to make sure they can be trusted.

After the Class Loader loads the .class files into memory, the *Execution Engine* than actually executes the program, using the methods in the Method Area.

JVM Languages

Some compilers for languages other than Java now translate into Java bytecode, so they also can be run on the JVM. Among these are

- Jython, an implementation of Python,
- JRuby, an implementation of Ruby,
- Kotlin, which was developed in 2011 and is now Google's preferred language for Android developers,
- Scala, which was designed to be more concise than Java and to address other criticisms of Java, was first released in 2004.

Over forty other programming languages now have compilers that translate into Java bytecode; look here: https://en.wikipedia.org/wiki/List_of_JVM_languages.

One reason for translating a language into the JVM is to allow interoperability. For example, libraries of code written in Java can be used in a Scala program, and vice versa.

Just-in-time compiling

One of the criticisms of this system is that an interpreter running a program is less efficient than running a program written in the native machine language of the computer itself. To mitigate this problem, just-in-time (JIT) compilers are used. At some point during interpreting, pieces of the Java bytecode will be compiled into the native machine language and then executed directly.