1 Interpreters vs. compilers

There are two strategies for obtaining runnable code from a program written in some programming language that we will call the source language. The first is compilation, or translation, in which the program is translated into a second, target language.

![Diagram of compiling a program]

Figure 1: Compiling a program

Compilation can be slow because it is not simple to translate from a high-level source language to low-level languages. But once this translation is done, it can result in fast code. Traditionally, the target language of compilers is machine code, which the computer’s processor knows how to execute. This is an instance of the second method for running code: interpretation, in which an interpreter reads the program and does whatever computation it describes.

![Diagram of interpreting a program]

Figure 2: Interpreting a program

Ultimately all programs have to be interpreted by some hardware or software, since compilers only translated them. One advantage of a software interpreter is that, given a program, it can quickly start running it without the time spent to compile it. A second advantage is that the code is more portable to different hardware architectures; it can run on any hardware architecture to which the interpreter itself has been ported.

The disadvantage of software interpretation is that it is orders of magnitude slower than hardware execution of the same computation. This is because for each machine operation (say, adding two numbers), a software interpreter has to do many operations to figure out what it is supposed to be doing. Adding two numbers can be done in machine code in a single machine-code instruction.
2 Interpreter designs

There are multiple kinds of software interpreters. The simplest interpreters are AST interpreters of the sort seen in HW6. These are implemented as recursive traversals of the AST. However, traversing the AST makes them typically hundreds of times slower than the equivalent machine code.

A faster and very common interpreter design is a bytecode interpreter, in which the program is compiled to bytecode instructions somewhat similar to machine code, but these instructions are interpreted. Language implementations based on a bytecode interpreter include Java, Smalltalk, OCaml, Python, and C#. The bytecode language of Java is the Java Virtual Machine.

A third interpreter design often used is a threaded interpreter. Here the word “threaded” has nothing to do with the threads related to concurrency. The code is represented as a data structure in which the leaf nodes are machine code and the non-leaf nodes are arrays of pointers to other nodes. Execution proceeds largely as a recursive tree traversal, which can be implemented as a short, efficient loop written in machine code. Threaded interpreters are usually a little faster than bytecode interpreters but the interpreted code takes up more space (a space–time tradeoff). The language FORTH uses this approach; these days this language is commonly used in device firmware.

3 Java compilation

Java combines the two strategies of compilation and interpretation, as depicted in Figure 3.

Source code is compiled to JVM bytecode. This bytecode can immediately be interpreted by the JVM interpreter. The interpreter also monitors how much each piece of bytecode is executed (run-time profiling) and hands off frequently executed code (the hot spots) to the just-in-time (JIT) compiler. The JIT compiler converts the bytecode into corresponding machine code to be used instead. Because the JIT knows what code is frequently executed, and which classes are actually loaded into the current JVM, it can optimize code in ways that an ordinary offline compiler could not. It generates reasonably good code even though it does not include some (expensive) optimizations and even though the Java compiler generates bytecode in a straightforward way.

The Java architecture thus allows code to run on any machine to which the JVM interpreter has been ported and to run fast on any machine for which the JIT interpreter has also been designed to target. Serious code optimization effort is reserved for the hot spots in the code.
The Java architecture also makes it relatively easy to generate new code at run time. This can help increase performance for some tasks, such as implementing a domain-specific language for a particular application. The HW6 project is an example where this strategy would help.

The class `javax.tools.ToolProvider` provides access to the Java compiler at run time. The compiler appears simply as an object implementing the interface `javax.tools.JavaCompiler`. It can be be used to dynamically generate bytecode. Using a classloader (also obtainable from `ToolProvider`), this bytecode can be loaded into the running program as a new class or classes. The new classes are represented as objects of class `java.lang.Class`. To run their code, the reflection capabilities of Java are used. In particular, from a `Class` object one can obtain a `Method` object for each of its methods. The `Method` object can be used to run the code of the method, using the method `Method.invoke`. At this point the code of the newly generated class will be run. If it runs many times, the JIT compiler will be invoked to convert it into machine code.

Another strategy for generating code is to generate bytecode directly, possibly using one of the several JVM bytecode assemblers that are available. This bytecode can be loaded using a classloader as above. Unfortunately JVM bytecode does not expose many capabilities that aren’t available in Java already, so it is usually easier just to target Java code.

The Java Virtual Machine

JVM\(^1\) bytecode is stored in class files (.class) containing the bytecode for each of the methods of the class, along with a constant pool defining the values of string constants and other constants used in the code. Other information is found in a .class file as well, such as attributes.

It is not difficult to inspect the bytecode generated by the Java compiler, using the program `javap`, a standard part of the Java software release. Run `javap -c <fully-qualified-classname>` to see the bytecode for each of the methods of the named class.

The JVM is a stack-based language with support for objects. When a method executes, its stack frame includes an array of local variables and an operand stack.

Local variables have indices ranging from 0 up some maximum. The first few local variables are the arguments to the method, including `this` at index 0; the remainder represent local variables and perhaps temporary values computed during the method. A given local variable may be reused to represent different variables in the Java code.

For example, consider the following Java code:

```java
if (b) x = y+1;
else x = z;
```

1 Consult the [Java Virtual Machine Specification](https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html#jvms-4.4) for a thorough and detailed description of the JVM.
The corresponding bytecode instructions might look as shown in Figure 4. Each bytecode instruction is located at some offset (in bytes) from the beginning of the method code, which is shown in the first column in the figure. In this case the bytecode happens to start at offset 13, and variables b, x, y, and z are found in local variables 3, 4, 5, and 6 respectively.

All computation is done using the operand stack. The purpose of the first instruction is to load the (integer) variable at location 3 and push its value onto the stack. This is the variable b, showing that Java booleans are represented as integers at the JVM level. The second instruction pops the value on the top of the stack and sees if it is equal to zero (i.e., it represents false). If so, the code branches to offset 26. If non-zero (true), execution continues to the next instruction, which pushes y onto the stack. The instruction iconst_1 pushes a constant 1 onto the stack, and then iadd pops the top two values (which must be integers), adds them, and pushes the result onto the stack. The result is stored into x by the instruction istore 4. Notice that for small integer constants and small-integer indexed local variables, there are special bytecode instructions. These are used to make the code more compact than it otherwise would be: by looking at the offsets for each of the instructions, we can see that iload_3 takes just one byte whereas iload_5 takes two.

5.1 Method dispatch

When methods are called, the arguments to the method are popped from the operand stack and used to construct the first few entries of the local variable array in the called method’s stack frame. The result from the method, if any, is pushed onto the operand stack of the caller.

There are multiple bytecode instructions for calling methods. Given a call x.foo(5), where foo is an ordinary non-final, non-private method and x has static type Bar, the corresponding invocation bytecode is something like this:

```
invokevirtual   #23
```

Here the #23 is an index into the constant pool. The corresponding entry contains the string f:(I)V, showing that the invoked method is named f, that it has a single integer argument (I), and that it returns void (V). We can see from the fact that the name of the invoked method includes the arguments of the types that all overloading has been fully resolved by the Java compiler. Unlike the Java compiler, the JVM doesn’t have to make any decisions about what method to call based on the arguments to the call.

At run time, method dispatch must be done to find the right bytecode to run. For example, suppose that the actual class of the object that x refers to is Baz, a subclass of its static type Bar, but that Baz inherits its implementation of f from Bar. The situation inside the JVM is shown in Figure 5.

Each object contains a pointer to its class object, an object representing its class. The class object in turn points to the dispatch table, an array of pointers to its method bytecode. In the depicted example, the JVM has decided to put method f:(I)V in position 2 within this array. To find the bytecode for this method, the appropriate pointer is loaded from the array. The figure shows that the classes Bar and Baz share inherited bytecode. If Baz had overridden the method f, the two dispatch tables would have pointed to different bytecode.

The JIT compiler converts bytecode into machine code. In doing so, it may create specialized versions of inherited methods such as f, so that different code ends up being executed for Bar.f and Baz.f. Specialization of code allows the compiler to generate more efficient code, at the cost of greater space usage.

There are other ways to invoke methods, and the JVM has bytecode instructions for them:

- `invokestatic` invokes static methods, using a table in the specified class object. No receiver (this) object is passed as a an argument.
- `invokespecial` invokes object methods that do not involve dispatch, such as constructors, final methods, and private methods.

---

2In C++ implementations, this array is known as the vtable, and objects point directly to their vtables rather than to an intervening class object.
Figure 5: Method dispatch to an inherited method

- **invokedynamic**: invokes object methods without requiring that static type of the receiver object support the method. Run-time checking is used to ensure that the method can be called. This is a recent addition to the JVM, intended to support dynamically typed languages. It should not ordinarily be needed for Java code.

6 Bytecode verification

One of the most important properties of the JVM is that JVM bytecode can be type-checked. This process is known as bytecode verification. It makes sure that bytecode instructions can be run safely with little run-time checking, making Java both faster and more secure. For example, when a method is invoked on an object, verification ensures that the receiver object is of the right type (and is not, for example, actually an integer).

7 Type parameterization via erasure

The JVM knows nothing about type parameters. All type parameters are erased by the Java compiler and replaced with the type `Object`. An array of parameter type T then becomes an array of `Object` in the context of the JVM, which is why you can’t write expressions like `new T[n]`. 