Lecture 1: Introduction to CS18

Jan 21, 2015

Contents

1 Introduction to CS 18 **1**

2 Getting Started **2**

2.1 Key Features of Java and Scala 2

2.2 The Java Virtual Machine 3

3 Getting Started with Java **4**

4 Hello, World! **6**

Objectives

By the end of this lecture, you will know:

- our goals for the semester
- key features of Java and Scala
- what the Java virtual machine (JVM) is

By the end of this lecture, you will be able to:

- write a program that prints `"Hello world!"` in Java

1 Introduction to CS 18

Welcome to CS 18! This course is a follow on to CS 17. The mechanics of CS 18 are very similar to those of CS 17. That is, lectures, homeworks, labs, projects, and exams all work in pretty much the same way; but new to CS 18 are occasional in-class informal "workshops" – a chance for students to work together in small groups (read, "learn from one another").

Content-wise, here's a rundown of some of what we will cover this semester.

- **Imperative programming**: computations are expressed in terms of state. Statements can change that state. The key word here is *mutation*.
- **Object-oriented programming (OOP)**: computations are expressed in terms of objects. Statements allow objects to interact with one another.

- Data Structures:
  - **Arrays**: collections of data, indexed by integers, with constant time access to individual elements
  - **Lists**: implemented as **linked lists** and **dynamic arrays**
  - **Hash Tables**: collections of data, indexed by non-integers, with constant time access to individual elements
  - **Heaps**: tree-based data structures such that the greatest (or least) element is always stored at the root
  - **KD Trees**: Binary trees which support efficient nearest neighbor search for multidimensional data
- Algorithms:
  - **Dynamic programming**: faster algorithms for some problems we solved last semester using enumeration
  - More sorting algorithms
  - Graph algorithms
  - Parsing
- Two programming languages: Java and Scala
- Important programming tools:
  - Eclipse, an integrated development environment (IDE)
  - JavaFX, to develop GUIs (Graphical User Interfaces) for Java or Scala programs

2 Getting Started

We will start off this semester with Java, an object-oriented programming language that is widely used in industry today. About halfway through the semester, we will transition to Scala, an object-oriented programming language that supports both functional and imperative programming.

Hence, in CS 18, you'll be learning how to program in both Java and Scala. But, more generally, you'll learn how to structure programs using the object-oriented and imperative paradigms, the focus of CS 18, together with the functional programming paradigm of CS 17.

Like Racket, which is an offspring of Lisp, Java derives from and extends earlier languages. Java syntax is closest to that of C++ (which in turn, is based on C), and its execution model (the Java Virtual Machine—**JVM**) is based on Smalltalk. Scala runs on the JVM, and can leverage all code written in Java, including Java's libraries (Java's Application Programming Interface—**API**). Hence, Scala **interoperates** with Java.

2.1 Key Features of Java and Scala

- **Object-oriented**: Scala and Java include fundamental object-oriented design capabilities such as inheritance, encapsulation, and subtype polymorphism, which facilitate code re-use and extensibility. We will cover these principles in detail over the course of the semester.

- **Imperative**: Imperative programming is "natural," because we often think in an imperative way. In imperative languages, programs are sequences of steps, much like a recipes. As a sequence of steps executes, a program's state changes. This is called **mutation**.
- **Platform Independence**: Java's motto is "Write Once, Read Anywhere," meaning: write a program once, compile it once, and run it everywhere.[1] Java achieves this ideal through the Java Virtual Machine (JVM) described below, as does Scala because it runs on the JVM. (And so do a number of other languages, like Groovy, Clojure, JRuby, and Jython.)
- **Extensive libraries**: Both Java and Scala have extensive libraries, meaning they provide support for pretty much everything (e.g., networking, graphics, parsing, security, etc.). However, in CS 18, we work from the ground up, so we won't allow you to use libraries for a while. Instead, you'll start by developing the data structures you learn about: e.g., you'll learn to implement mutable lists! So, in short, hands off the libraries until we say otherwise.

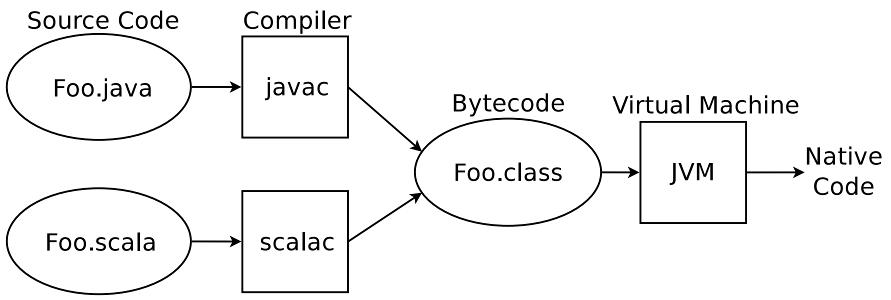
2.2 The Java Virtual Machine

People write **source code**, but machines read **machine code**, or, more specifically, platform-dependent **native code**. There are two ways (and combinations thereof) to run source code: it can be **compiled** into machine code, so that it can be directly executed, or it can be **interpreted**, meaning that it is indirectly executed by another program, called an **interpreter**.

A **compiler** is a program that translates source code into machine code, which is then directly excuted on the intended platform. Hence, a compiler is specific to a given machine architecture (e.g., there is a C compiler for Macs, and another one for IBMs). Using a compiler (only), running a program is a two-step process:

1. **Compilation**: Translate the source code into machine code.
2. **Execution**: Run the machine code.

An **interpreter** is a program that executes code. Historically, the code executed by an interpreter was source code, written by humans (e.g., the first Lisp interpreter). Today it is more common for there to be an additional layer of indirection, so that the code executed by an interpreter is not source code, but **virtual machine code**, output by a compiler. Indeed, Java and Scala,[2] are compiled into virtual machine code (also called **bytecode**), which is then interpreted by the **Java Virtual Machine (JVM)**, as illustrated below:



[1]In reality, "Write Once, Run Anywhere" is often more like "Write Once, Debug Everywhere" due to minor incompatibilities and inconsistencies among JVM implementations.
[2]and Racket, and optionally OCaml

The concept of a virtual machine dates back to at least Smalltalk, but the JVM popularized the idea. Compilers are difficult to write, but they are well-optimized so that compiled code executes very fast. Interpreters are easy to write,[3] but they execute code very slowly. On the JVM, byte code, not source code, is interpreted. Byte code interpreters are relatively easy to write and they execute code relatively quickly. So this additional layer of indirection helps achieve a sweet spot between writing correct code, and generating code that executes quickly.

Furthermore, the JVM facilitates platform-independence. A developer can code on a MAC, and then compile to byte code, which he can then release, and that byte code should run reliably on a Linux box. This approach is preferable to either releasing his source code for users to attempt to compile and run on their own (this would both give away intellectual property, and be a burden for users), and to compiling his source code for every imaginable platform. According to Sun Microsystems, the creators of Java, there are over 5.5 billion JVM-enabled devices! Imagine a developer attempting to release his code for even a small fraction of that many devices.

3 Getting Started with Java

Lexical Structure

- Java is a case-sensitive language (e.g., "if" is different than "If")
- Comments take two forms:

```
// this is a single line comment

/*
 * this is a
 * multiple line comment
 */
```

- Keywords with predefined meanings include: **if**, **else**, **true**, **false**, **class**, **interface**, **abstract**, **public**, **private**, **protected**, **static**, **final**, **void**, **this**, etc.
- Literals (i.e., constants) are numbers, characters in single quotes, strings in double quotes, true, false, etc..
- Identifiers are names that are given to a part of a program (e.g., a variable or a method). Their meaning is known to Java through their declarations (e.g., **int** $i = 18$ declares an integer named i with value 18). Identifiers are comprised of arbitrary numbers of alphanumeric characters, and the underscore character, beginning with a lowercase letter (by convention) (e.g., *i*, *x18*, *foo*, *myObject*, *your_Object*).
- Punctuation Marks: {, }, `(`, `)`, `;`, `.`, etc.
- Arithmetic Operators: `+`, `-`, `*=`, `/=`, `%`, etc.
- Comparison Operators: `<`, `<=`, `>`, `>=`, `==`, `!=`, etc.
- Logical Operators: &&, ||, `!`

[3] Indeed you wrote one in CS 17—your first semester of CS.

Primitive Data Types The following are primitive data types in Java:

- a **boolean** type: **boolean**
- a **character** type: **char**
- four **integer** types: **byte**, **short**, **int**, and **long**. These vary in size, from 8 bits to 64.
- two **floating-point** types: **float** and **double**. These also vary in size: 32 bit and 64.

Collectively, the integer types are called **integral** types, and the integer and floating-point types together are called **numeric** types.

Conditionals Flow of control in imperative programs is more complicated than in functional programs. Among other things, there are more control flow constructs. For today, however, we stick with two which are already familiar from CS 17: conditionals and functions (*a.k.a* **methods**).

We use `if/else` statements to incorporate conditional logic into our OOPs. We test a condition to see if it is true relative to our data and choose to invoke different behaviors based on the result. For example:

```
if (temp > 100) {
  // Execute this code
} else if (temp < 0) {
  // Execute this code
} else {
  // Execute this code
}
```

Methods Functions are called **methods** in Java and other object-oriented programming languages. Here is the Java syntax for a one-line method.

```
<ReturnType> <methodName>(<arg1Type> <arg1Name>, <arg2Type> <arg2Name>, ...) {
  return (<expression of type <ReturnType>>);
}
```

Before the name of the method, the method's return type is given. Then the formal parameters are listed, again with each type preceding each name. The code appears between left and right curly braces, delimiting the start and end of the method. The only line of a one-line method is a **return** statement. This statement "returns," meaning it passes flow of control back to the method that invoked this method, and along with that flow of control it also passes an expression whose type is a value of type *ReturnType*.

For example, the following *addition* method adds two integers and returns their sum.

```
int add(int x, int y) {
    return (x + y);
}
```

4 Hello, World!

When you set out to learn a new programming language, 90% of the time the first program you write is "Hello, World!" This program does one thing, and one thing only. It outputs "Hello, World" to standard output (i.e., on the console).

Here it is, folks; the "Hello, World!" program, in Java.

```java
/*
 * the HelloWorld class prints "Hello, world!" to standard output
 */
public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello, World!"); // print "Hello, World!"
  }

}
```

Object-oriented code is structured in terms of **classes** and **objects**. Objects are instances of classes; likewise, classes are grouping of objects. Each object stores and manipulates data. The latter is achieved via **methods**, which specify behavior relevant to an object. What we called functions or procedures in CS 17 are called methods in CS 18.

Our "Hello, World" program consists of one class definition, the *HelloWorld* class. That class is declared to be **public**. The fact that classes can be declared public, private, etc., is called **access control**. We will undertake a detailed discussion of access control next week (or the week after). For now, we will simply declare all our classes to be **public**.

Within the *HelloWorld* class, there is one method definition, *main*. Every Java program has exactly one main method, which, as in C and C++, is the program's entry point: i.e., the point where the flow of execution begins. The main method is always declared to be both **public** and **static**. When a method is static, it means that the method is accessible even if an instance of its container class is not created (which it is not in this program). The *main* method is required to be static so that the JVM can access the entry point of a Java program without having to instantiate any objects. It leaves the instantiation of objects to the programmer (and we leave a discussion of instantiating objects until next week).

The signature of the main method in a Java program is **void** *main*(*String*[] *args*). That is, the input to the main method is *String*[] *args*, and its output is **void**. The **void** keyword means "nothing." So, *main* returns nothing. Why invoke a method that returns nothing? Such a method is executed for its **side effects** only. An example of a side effect is printing to standard output.

The syntax *String*[] means an array of strings. We will learn about arrays in two weeks. For now, just think of them as lists. This means that the *main* method takes as input a list of strings, which, by convention, is always named *args* in Java programs. The strings in the list *args* are called **command-line arguments**. They are useful when you want your program's performance to vary at run time. For example, the unix program *ls* takes in a variety of command-line arguments, such as -*a* and -*l*. Try them. This single program can function very differently (without recompiling) depending on its input.

Within the main method, the line *System.out.println*("Hello, World!"); is a call to the *System* class in the Java library to print the "Hello, World!" message to standard output. When we start programming in Scala, you will be relieved to learn that it is sufficient to type *println*("Hello World!");. But Java requires that you type the entire path to the method in the Java library.

Compiling and running the "Hello, World!" program is fairly straightforward. To compile source code into bytecode, use `javac`, the Java compiler. To run bytecode, use `java`, the JVM.

```
cslab1a ~ $ javac HelloWorld.java
cslab1a ~ $ java HelloWorld
Hello, World!
```

By way of comparison, here is the "Hello, World" program in Basic.

```
10 PRINT "Hello, World!"
```

It's pretty easy to figure out what this program does, even if you don't know any Basic. It has one line, and that one line does one action, PRINT, which means print the string that follows the PRINT command to the console.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs018/feedback`.