Prepared by Linan Qiu <lq2137@columbia.edu>

Stuff You Probably Know

Java Language

Let's start with the basics.

Java is a programming language. You (mainly) write source code in files with the extension `.java`. Let's say you write the following code in `Hello.java`:

```
// Hello.java

public class Hello {
  public static void main(String[] args) {
    System.out.println("Han Solo dies.");
  }
}
```

The Java compiler compiles the source code into a class (`.class`) file, `Hello.class`. These are also called Java bytecode. If you're used to being coddled by IDEs like Eclipse or NetBeans, do take a second or so to remind yourself that the `javac` compiler exists on your computer to actually compile Java source code into Java bytecode. To run the compiler, you do

```
$ javac Hello.java
```

By the way, when you see the `$` sign in the rest of this set of notes (or anywhere in most manuals, readmes, etc) it means that the stuff following the `$` is what you type into your console.

You'll now find `Hello.class` in your directory in addition to `Hello.java`.

Java bytecode is then executed by the Java Virtual Machine. You do that when you type the `java` command.

```
$ java Hello
Han Solo dies.
```

Note that we leave out the `.class` when we run the program.

Why all this trouble? This is where the beauty of Java comes in:

- Bytecodes are the same no matter what machine you compile them on (provided you use the same version of `javac`).

- These bytecodes can then be expected to produce the same result on any Java Virtual Machine.

Essentially, it is like forcing publishers to write documents in English, then giving all readers a translator so that they can all understand the documents in the same way.

In this way, an user won't have to care about where the document was published, and a publisher won't have to worry about what reader reads it. Similarly, Java writers won't have to worry about the end user not being able to run the code, and Java users won't have to worry about how the code was compiled.

Here's the important bit (and students get it wrong all the time): when you install Java, you can either choose to install the **JRE (Java Runtime Environment)** or the **Java SDK (Java Software Developer Kit)**. You'd want to choose the second, not just because it has your job title in it. The JRE only contains the Java Virtual Machine and not `javac`, so you'd be able to run bytecode but not compile source code. Java SDK contains both the Java Virtual Machine (the JRE) and the compiler. If you're looking to compile Java code, as you'd be doing every other day in this class, choose the Java **SDK**. Preferably, choose version 1.8 (also known as Java 8).

Primitives

Primitives are Java's built-in data types. There are 8 of them: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.

Convince yourself that the following is true:

1. Downcasting Can Be Done Directly

```
// in a main method far far away

double x = 1.11;
int y = x;
System.out.println(y); // prints 1
```

2. Upcasting Has To Be Explicit

```
// in a main method far far away

int x = 1;
double y = x; // the compiler will say:
// Type mismatch: cannot convert from double to int
```

```java
// instead you'll have to do
double y = (double) x;
```

3. chars are ints in Disguise

Review ASCII.

Classes

Class are data types that you define. Here's a quick reminder. Let's say you have a class like this:

```java
// Jedi.java

public class Jedi {
  public String name;
  private int lightsaberCount;

  public Jedi(String aName, int aLightsaberCount) {
    name = aName;
    lightsaberCount = aLightsaberCount;
  }

  public void setLightsaberCount(int lightsaberCount) {
    this.lightsaberCount = lightsaberCount;
  }

  public int getLightsaberCount() {
    return lightsaberCount;
  }

  public void buildLightsaber() {
    lightsaberCount++;
  }
}
```

To avoid making this set of notes too long, **convince yourself that the following is true**:

1. Function Scope and this Keyword

You can change the constructor to:

```
public Jedi(String name, int lightsaberCount) {
  this.name = name;
  this.lightsaberCount = lightsaberCount;
}
```

And the code will still be exactly the same. In fact, this is recommended. Way neater and readable.

2. Instanting Classes into Objects

In another class (say `Test.java`), you can do the following:

```
// Test.java

public class Test {
  public static void main(String[] args) {
    Jedi obiwan = new Jedi("Obiwan Kenobi", 1);
    System.out.println(obiwan.name);
  }
}
```

Then, when you `javac Test.java`, you'll compile `Jedi.java` as well. You can then run this code by `java Test`, and it will output `Obiwan Kenobi`.

In fact, you can put this main method into `Jedi.java` to avoid having another `Test.java` file:

```
// Jedi.java

public class Jedi {
  public String name;
  private int lightsaberCount;

  public Jedi(String name, int lightsaberCount) {
    this.name = name;
    this.lightsaberCount = lightsaberCount;
  }

  public void setLightsaberCount(int lightsaberCount) {
    this.lightsaberCount = lightsaberCount;
  }

  public int getLightsaberCount() {
    return lightsaberCount;
  }
```

```
  public void buildLightsaber() {
    System.out.println("Going to the crystal cave");
    lightsaberCount++;
  }

  public static void main(String[] args) {
    Jedi obiwan = new Jedi("Obiwan Kenobi", 1);
    System.out.println(obiwan.name);
    Jedi luke = new Jedi("Luke Skywalker", 1);
    System.out.println(luke.name);
  }
}
```

3. `public` and `private` Variables

In the main method, you can do this:

```
Jedi obiwan = new Jedi("Obiwan Kenobi", 1);
System.out.println(obiwan.name);
```

But you cannot do this:

```
Jedi obiwan = new Jedi("Obiwan Kenobi", 1);
System.out.println(obiwan.lightsaberCount);
```

Instead, you need to do this:

```
Jedi obiwan = new Jedi("Obiwan Kenobi", 1);
System.out.println(obiwan.getLightsaberCount());
```