

Laboratory: Java from the Command Line

Summary: We explore a variety of issues in compiling and running Java from the command line.

Prerequisite Knowledge: Objects, interfaces, generics.

Preparation

Fork and clone the repository at <https://github.com/Grinnell-CSC207/command-line>.

Scan through the files to make sure that you understand the purpose of each file.

Exercises

Exercise 1: Compiling from the Command Line

The command `javac` is used to compile Java files.

- Use `javac` to compile `Sorter.java`. What kind of file is produced?
- Remove `Sorter.class`. Use `javac` to compile `ISort.java`. Do you get anything in addition to `ISort.class`? What are the implications?
- You can run a Java `.class` file with the `java` command and the name of a class that contains a `main` method. Type the following command (without the dollar sign prompt) to run our sample sorting program.

```
$ java ISort 6 1 2 9 6 0
```

Exercise 2: Packages and Your CLASSPATH

Right now, the `.java` files are in the default package. As a good Java programmer, you know that every class should really be in an appropriate package. So let's put these files in the package `username.sorting`.

- Using your favorite text editor (Emacs, vi, vi-mangled, GEdit), add the line `package username.sorting;` to the top of each file.
- What happens when you try to compile `ISort.java`?
- As you've discovered, the Java compiler can no longer find the other `.java` files. What do we do? Two steps, believe it or not. First, we need to use the appropriate directory structure. In particular, if the package is `username.sorting`, the code is supposed to be in the directory `username/sorting`. Make that directory. Then, move your `.java` files to that directory. But that's not all!

d. The other important thing to do is to set up your `CLASSPATH` variable, which governs where Java looks for code files. Type the following

```
$ export CLASSPATH=/path/to/parent/directory
```

For example, if the Java code is in `/home/username/Java/username/sorting`, use `/home/username/Java` as the directory in the command above.

e. What do you expect to have happen if you try to run `ISort`? Check your answer experimentally.

f. The full name of `ISort` is now `username.sorting.ISort`. See if you can run it with that name.

Exercise 3: Library Files

As you may recall from your work in C, we often use library files to store utility code. In Java, library files are called “jar files” or “jars”. You build a “jar” with

```
$ jar cf File.jar classfiles
```

a. Create a jar file named `Sorting.jar` with the utility files `Sorter.class`, `StandardIntegerComparator.class`, and `BuiltinSorter.class`.

b. Remove those three `.class` files.

c. What do you expect to have happen when you try to run `ISort` with the following command?

```
$ java username.sorting.ISort 10 5 1 2 4 9 1
```

d. Check your answer experimentally.

e. As you probably expected, Java doesn’t automatically recognize the files in the `.jar`. However, we can solve that problem: You can add the `.jar` to your classpath. Try the following command:

```
$ export CLASSPATH=Sorting.jar:$CLASSPATH
```

f. Now try to run `ISort` again.

Exercise 4: Specifying a Main

What do we do when we want to ship a Java program? Usually, we put all of the files, including the class with the `main` method, in the same `.jar` file. We then have to specify where to find `main`. We do that with a “manifest file” that looks something like the following.

```
Main-Class: ISort
```

a. Create a file, `manifest.txt` with that line.

b. Create a jar file, `ISort.jar`, with the following command:

```
$ jar cvfm ISort.jar ISort.class manifest.txt Sorter.class BuiltinSorter.class StandardIntegerComparator.class
```

c. You can run the jar file with the following command. Try it with your own set of values.

```
$ java -jar ISort.jar 5 4 1 2 3
```

Exercise 5: Unit Testing

a. What do you expect to have happen if you compile the unit test, which you can find in `BISTest.java`?

b. Check your answer experimentally.

c. As you probably discovered, the Java compiler is upset that it doesn't know where the JUnit code is. But you can tell the compiler where to find JUnit. The JUnit code is in `/usr/share/java/junit.jar`. Add that directory to your `CLASSPATH` and see if you can get the test to compile.

d. Of course, you also want to run the unit tests. But where's the `main` method? Somewhere in JUnit. There are a variety of ways to run unit tests. Here's one that should work.

```
$ java org.junit.runner.JUnitCore username.sorting.BISTest
```

Give it a try.

Exercise 6: Javadoc

We've seen how to compile Java files, run the compiled files, put them into libraries, and test them. What's left? Generating documentation!

You can create Javadoc documentation with the `javadoc` command. Try creating the documentation with

```
$ javadoc *.java
```

What files are created? Try to open the natural files in your browser.

For Those With Extra Time: An Improved Structure

Right now, we're doing everything in the same directory. But that clogs the directory. Really, we should have separate directories for our main source code, our tests, our compiled files, our Javadoc, and perhaps other things.

Ant provides a way to organize all of those files. Read the Ant documentation. Set up an appropriate directory structure and write an Ant build file to put things in the appropriate place.

Copyright (c) 2013-14 Samuel A. Rebelsky.



This work is licensed under a Creative Commons Attribution 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.