Lund University
Computer Science
Christoff Bürger, Görel Hedin, and Niklas Fors

Compilers
EDAN65
2017-08-17

Programming Assignment 0
Java, Unix, and Gradle

In this course, you need to be fairly proficient with Java and the Unix command line, and you will also use the build tool Gradle. Depending on your experience, you may already know some of the things in this assignment. You should do this assignment on your own, and you may skip any section that practices things you are already familiar with.

You can do this assignment either on the student lab machines with Unix, or on any other Unix computer, e.g., Mac or Linux.

There are student lab machines with Unix in the following rooms:
`venus, mars, jupiter` (north part of E-house basement)
`hacke, panter, lo, val, falk, varg` (south part of E-house basement)
`alfa, beta, gamma` (2nd floor, E-house)
`backus` (entry level, Math house)

1 The Unix file system

Can you do the following tasks without looking in a manual?

- Open a terminal window.
- Go to your home directory.
- Print the full path name of your home directory.
- Find how many files are in the current directory.
- Get help for a command by reading its manual. For example, can you find out what the `-t` option does for the `ls` command?
- Go to the parent directory. What files are there and which access rights do they have?
- Go to the `/bin` directory. Do you recognize any of the files there?
- Go to your home directory.
- Create a new directory `my-dir`.
- Create a new file `my-file` (you can use the `touch` command to create new empty files).
- Copy `my-file` to `my-dir`.
- Rename `my-file` to `my-file.txt`.
- Delete `my-file.txt`.
- Create a new directory `my-dir-2`.
- Copy `my-dir` to `my-dir-2`.
- Imagine you would like to move `my-dir` instead of copying it. Delete the copy `my-dir-2/my-dir` and move `my-dir` to `my-dir-2`. Use `ls` if you get confused what is where.
- Delete everything you created in the above tasks (should be just the directory `my-dir` and its contents).

If you are sure you can do all these tasks without looking in a manual, you can skip the rest of this section and go to Section 2. If not, read the rest of this section, then go back and do the tasks above.

1.1 Opening a terminal window

If you don't know how to open a terminal window on your operating system, search on Google. For example, if you are using Ubuntu Linux, google the following phrase:

```
open terminal window in ubuntu
```

1.2 Navigation

Play around with the following commands to understand how you can navigate in the Unix file system: *Note!* If you don't know how to type the tilde character (~) on your platform, google this.

pwd (print working directory)

cd (change directory to your home directory)

cd dir (go to the *local* directory `dir`, relative to the current directory)

cd /usr/bin (go to the directory with the *absolute* path `/usr/bin`, not relative to the current directory)

cd .. (go to the parent directory)

cd ~/foo (go to the directory `foo` in your home directory)

ls (list all files in the working directory; files starting with a period like `.gitignore` are excluded)

ls -a (list all files in the working directory, including those starting with a period)

ls -l -h (list files with detailed information like size, change time and access rights)

man ls (Show the manual for the `ls` command. Type space to go to the next page. Type `b` to go to the previous page. Type `q` to quit.)

1.3 Manipulation

Play around with the following commands to understand how you can manipulate files and directories. *Note!* There is no undo in Unix! **Be careful** not to delete or overwrite important files.

mkdir d (make (create) directory `d`)

touch f1.txt (create a new empty file named `f1.txt`. The `touch` command updates the last modified timestamp of a file, but if the file did not exist before, it is created.)

cp dir/f1.txt f2.txt (copy file `dir/f1.mk` and name the copy `f2.txt`; the copy will be created in the working directory, the original is in subdirectory `dir`)

cp f .. (copy file `f` to its parent directory)

cp ../f . (copy file `f` in the parent directory to the working directory.

cp -r d1 d2 (recursively copy directory `d1` into directory `d2`; copies `d1` and all its content)

rm f (remove file `f`)

rm -r d (recursively remove directory `d`; deletes `d` and all its content)

mv x y (assume `x` is a file and `y` a directory: move `x` to `y`)

mv x y (assume `x` is a file and `y` does not exist: rename `x` to `y`)

mv x y (assume `x` and `y` are files: overwrite `y` with `x` and remove `x`)

mv x y (assume `x` is a directory: rename it to `y` if `y` does not exist yet, otherwise move it to `y`; the operation is aborted with an error if `y` is an already existing file)

1.4 Further reading

If you need further help in understanding the commands, you can read, for example:

`http://linuxcommand.org` -> Learning the shell -> Navigation/Looking Around/Manipulating Files

Introduktion till LTH:s Unixdatorer, by Per Foreby, 2012. In Swedish.
`http://www.student.lth.se/fileadmin/ddg/text/unix-x.pdf`

1.5 Check your proficency

Now check your proficiency with Unix commands by going back to the beginning of section 1 and do the tasks there.

2 Editing text files

You should be able to use an ordinary text editor like Emacs, Vim, Gedit, Sublime, or Nano to edit text files. If you are not used to a particular text editor, we recommend Nano. Nano is simple and runs directly in the terminal window. Here is how you get started with it:

nano f (type this in a terminal window to start nano on the file f)

type text (to add text to the file)

use the arrow keys (to navigate in the file)

ctrl-X (to exit the nano editor) (**ctrl-X** means hold down the ctrl-key while you type X.)

ctrl-G (to get a help text. Then type **ctrl-X** to get back to the file you were editing.)

2.1 Create a small Java program by using a text editor

Use the text editor to construct a java program `Hello.java`:

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

Make sure the file contains the correct code by typing

```
less Hello.java
```

in the terminal window. The `less` command lists the contents of a file, and you can move forward and backward using space and `b`, just like in the `man` command. You can also use the arrow keys. Type `q` to quit from the `less` command.

3 Java from the command line

You need to be able to compile and run Java programs from the command line.

Can you do the following tasks without looking in a manual?

- Implement two Java classes, `A` and `B` in a package `p`, where `B` contains a main method and imports `A`. The main method prints `Hello!` on the standard output and the `toString()` value of an `A` object.
- Compile all classes.
- Run the main method of `B`.
- Pack the classes into an executable Java Archive called `hello.jar` (you have to create a manifest).
- Execute `hello.jar`.
- Add a `while (true) { System.out.println("Foo"); }` loop to the main method of `B`; recompile it. If you execute it, how can you abort?
- Add a string argument to `B` so that it prints `Hello` *arg* `!` instead of only `Hello!`, where *arg* is the argument you give on the command line when you run `B`.

Hints: Classes must be in directories named like their package; their first code line should declare the package name. To compile and run them, your working directory has to be the parent directory of the package hierarchy.

If you are sure you can do all these tasks without looking in a manual, you can skip the rest of this section. Otherwise, read through the rest of this section and then go back and do the tasks above.

3.1 Compiling and running Java programs

Here are some commands for compiling and running Java programs from the command line.

javac C1.java C2.java (compile classes `C1` and `C2` to bytecode for the Java Virtual Machine)

javac p/*.java (compile all classes in the package `p`)

javac -d dir p/C.java (compile class `p.C`; generate its class file in directory `dir`)

javac -cp d1:d2 C.java (compile class `C`; `d1` and `d2` are added to the classpath for compilation (the working directory is always included), such that `C` can use classes in `d1` and `d2`)

java p.C (execute the main method of class `C` in package `p`)

java p.C just three args (execute `p.C` with arguments `just`, `three`, and `args`)

java -cp d1:d2 p.C (execute `p.C`; `d1` and `d2` are added to the classpath for execution, such that classes in `d1` and `d2` can be used. (The working directory is always on the classpath by default.)

java -jar prog.jar (execute the main method specified in the manifest of the Java Archive `prog.jar`)

`ctrl-c` **while program runs** (abort execution of running program)

3.2 Jar files and classpaths

Java programs are packaged and distributed as JAR (**J**ava **Ar**chive) files. We will now create a JAR file that will contain the `A` and `B` classes defined before. A JAR file requires a Manifest file that specifies, for example, which class is the main class of the JAR file. We will start by creating the Manifest file.

Add the following code to the file `MANIFEST.MF`.

```
Manifest-Version: 1.0
Main-Class: B
```

We can see that `B` is specified as the main class. The command for creating JAR files is `jar`; the pattern for using it is as follows.

```
$ jar cfm jar-file manifest-file input-file(s)
```

For example, we can create a JAR file `HelloJar.jar` as follows, using the Manifest file that we defined earlier.

```
$ jar cfm HelloJar.jar MANIFEST.MF p/A.class p/B.class
```

Note that we only include the class files, and not the source code (`A.java` and `B.java`). A JAR file can contain source code, but that is not necessary. Having the JAR file, it can be executed as follows.

```
$ java -jar HelloJar.jar
```

The command `java` starts the Java virtual machine (JVM) which runs a program by loading class files and calling the `main` method of the main class.

The *classpath* specifies where the JVM should look for the class files. Specifying the classpath may be required when several different libraries are used that are located in different JAR files. The classpath can be given using the flag `-cp` as the following illustrates.

```
$ java -cp lib1.jar:lib2.jar:. MainClass
```

The classpath is given as a list of colon-separated JAR files and directories. In this case, the JVM will look for class files in `lib1.jar`, in `lib2.jar`, and in the current working directory; the dot (`.`) means the current working directory. The standard classpath is the current working directory, which is equivalent to the following.

```
$ java -cp . MainClass
```

Often, it is required to include the library files when compiling the source code as well. For example, suppose we have a class `MainClass` that uses classes from the JAR files `lib1.jar` and `lib2.jar`. Then when compiling the `MainClass.java` file, we need to include the JAR files in the classpath as follows.

```
$ javac -cp lib1.jar:lib2.jar MainClass.java
```

3.3 Check your proficiency

Now check your proficiency with creating, compiling and running Java programs by going back to the beginning of section 3 and do the tasks there.

4 The build tool Gradle

As a project grows larger, there may be many commands that need to be run in order to compile, test, create jar files, etc. Running these commands manually after each change of the source code would quickly become both tedious and error prone. Instead, software projects use *build tools* that document and automate the commands to run. Examples of common build tools are Make, Ant, and Gradle. In this course, we will use *Gradle*.

The Gradle build tool is implemented in the language Groovy, which compiles to Java bytecode. We will run the Gradle tool via a wrapper script called `gradlew`. When this script is run the first time, the Gradle tool (a jar file) is downloaded from the web. This way, you don't need to manually install the Gradle tool on your computer.

Gradle is a very powerful build tool. Here are some things it can do:

- automate common tasks like compiling, producing jar files, and running tests
- support different languages by the use of *plugins*. There is, for example, a Java plugin.
- incremental build, i.e., avoid building things that are already up to date. For example, if you run a build, but did not change any source files since your last build, then no compilation or testing will be run.
- automatic download of *dependencies*, i.e., appropriate versions of library files that your project depends on. These files are downloaded from global repositories like Maven and Ivy.

4.1 Gradle basics

To understand the basics of how Gradle works, download `A0-JavaGradleProject.zip` from `http://cs.lth.se/edan65`, and unzip it. This is an empty Java project, set up to build with Gradle. In the project you find the following files and directories:

- `build.gradle` – the build script
- `gradle/` – contains the `gradle-wrapper.jar` file, among other things.
- `gradlew` – the wrapper script for running Gradle on Unix
- `gradlew.bat` – the wrapper script for running Gradle on Windows
- `src/main/java` – where your main Java code should be placed
- `src/test/java` – where your JUnit test code should be placed

When you run Gradle (using the script `gradlew` or `gradlew.bat`), it by default reads in the file `build.gradle`. This file is the build script and it contains rules for how to build the project. Take a look at `build.gradle`. The only thing it contains is a line of code that says that the Java plugin should be used:

```
apply plugin: 'java'
```

The Java plugin defines rules for building Java projects. It defines a number of *tasks*, i.e., pieces of work. You can find out which tasks are supported by running the command[1]

```
./gradlew tasks --all
```

As you see, there are quite a few tasks. We will only discuss the following:

[1]The first time you run the `./gradlew` command, it will download the Gradle jar file from the web, which may take a short while. The Gradle jar file will be placed in a directory `.gradle` in your home directory.

- `build` – compiles any source and test files, creates a jar file, and runs any test files
- `compileJava` – only compiles the main Java code (not the test code)
- `compileTestJava` – only compiles the test code (the JUnit tests)
- `clean` – removes the generated files (class files, jar files, etc.)

Try out these tasks. For example, run

```
./gradlew build
```

Note that all generated files are placed in a generated directory `build`. Since there are no source files yet in this project, there will in this case be no generated class files, but a jar file will be generated, although it contains nothing but a manifest. The jar file is named `A0-JavaGradleProject.jar` after the project directory.

Now run the `clean` task, and note that the `build` directory with all the generated files is removed.

```
./gradlew clean
```

Tasks depend on each other in a directed acyclic graph. Note that when you run a task, it lists all dependent tasks it runs (its subtasks). Run the `build` task again. Note that it, for example, depends on the `compileJava` task.

When a task is called, and everything it depends on is unchanged, it will not be run again, but is instead reported to be `UP-TO-DATE`. Run the `clean` task, and then the `build` task twice. Note how several subtasks that were not up to date the first time are now up to date.

Gradle keeps its internal information in a directory `.gradle` in your home directory, and in a directory `.gradle` in each project. Any dependencies (library jar files, etc.) downloaded by a build will be stored in the `.gradle` directory in your home directory, so that they can be shared by all your Gradle projects. The Gradle tool itself is also stored there.

4.2 Experiment with Gradle on some Java code

Add some Java code to the project. For example, you could add a package `p` with a class `A` whose main method prints `Hello`. The code should be placed in the `src/main/java` directory. You will need to create the `src/main/java/p` directory to place your file in. Run a build (`./gradlew build`). Note where the classes and the jar file are placed (`build/classes/main` and `build/libs`, respectively). You should now be able to run your code by either

```
java -cp build/classes/main p.A
```

or

```
java -cp build/libs/A0-JavaGradleProject.jar p.A
```

4.3 Experiment with adding a test case

We will now show how you can run JUnit tests from Gradle.[2]

First you need something to test. Edit your class `A`, and add a static method `m` that simply returns a constant, say 5. Now create a test class `TestA` with a JUnit test method that tests that `m` works correctly:

```
package p;

import org.junit.*;
import static org.junit.Assert.*;

public class TestA {
        @Test
        public void testm() {
                assertEquals(5, A.m());
        }
}
```

Gradle will only recognize tests if they are placed in the `src/test/java` directory, so the above test class should be placed at the location `src/test/java/p/TestA.java`.

The imports used in the test code refer to entities in the JUnit library. To compile the tests, Gradle needs access to this library. To handle this, you declare a *dependency* in the `build.gradle` file. More precisely, when compiling tests, there is a dependency on a particular version of the JUnit library. You also need to state in what global repository Gradle can find the dependencies (i.e., the libraries). You do this by adding the following code to the `build.gradle` file:

```
repositories {
    mavenCentral()
}

dependencies {
    testCompile "junit:junit:4.12"
}
```

Here, `testCompile` is a so called *dependency configuration* defined by the Java Gradle plugin, and which is used by the task that compiles the tests (`compileTestJava`).

Now, when you run `./gradlew build`, Gradle will locate the Maven central repository, and find the JUnit library jar file (version 4.12), and download it to your computer (placing it in the `.gradle` directory in your home directory). It will then compile the test code, using this jar on the class path, and it will also run the tests.

Check that you can get this to work, and that the test case succeeds (the build should be reported as successful).

Now change the code so that the test will fail. For example, let the method in `A` return 4 instead of 5. What happens when you build again? Where can you find a report of the test results?

[2]JUnit is a framework for automated testing of Java code. If you are not familiar with JUnit, read through the introduction to JUnit used in the course EDAF45, see `http://fileadmin.cs.lth.se/cs/Education/EDAF45/2016/labs/L3/junitForEDAF45.pdf`.

5 Further reading

To get quick help, use a search engine like Google. Often you will find useful answers at Stack Overflow (`http://stackoverflow.com`).

Additional resources:

- Java cheat sheet from Princeton:
  `http://introcs.cs.princeton.edu/java/11cheatsheet/`
- Oracle's Java tutorial: `https://docs.oracle.com/javase/tutorial/java/index.html`
- Java standard library documentation: `http://docs.oracle.com/javase/8/docs/api/index.html`
- About running `java` and `javac` from the command line:
  ```
  man java
  man javac
  ```
- JAR tutorial: `https://docs.oracle.com/javase/tutorial/deployment/jar/index.html`
- Gradle documentation: `https://gradle.org/docs/`
- User guide for the Java plugin for Gradle: `https://docs.gradle.org/3.3/userguide/java_plugin.html`