CIS 201 Computer Science I
Fall 2009 Lab 01b

August 28, 2009

- Learn how to log into your Linux account.
- Learn how to log into your Moodle account.

- Learn where class resources reside on the Linux boxes.
- Learn how to create and navigate directories.
- Learn how to start (and stop) the Emacs text editor.
- Learn how to start a `.java` file.
- Learn how to compile a `.java` file to produce a `.class` file.
- Learn how to run a `.java` (or rather, `.class`) file.
- Learn how to use a Java class file as a *template*.
- Learn how to correct *compiler errors*.

Checkpoint 4 Launch a terminal.

Linux provides a desktop environment similar to that found in other operating systems (Windows XP/Vista or Mac OS X). It also offers a *command-line interface*, or a *shell* where the user can type commands into a command interpreter which runs the commands and then prompts the user for a new command. This class uses the command-line interface a great deal.

A shell runs inside of a terminal; go to the menu at the upper-left of the desktop and find **Accessories | Terminal**. Launching it should give you a window that has a prompt something like this:

```
laddbc@haystack:~$ _
```

The bit before the is your login name (Dr. Ladd's is `laddbc`). Between the and the `:` is the name of the machine you are logged into. Between the `:` and the `$` is the *current directory path*. Like many computer file systems, the Linux file system uses hierarchical *directories* or *folders*. Entries in the file system are either files, containing information (*e.g.*, a Java program, a term paper, your favorite MP3) or

a directory, a container for files and folders. For historic reasons, Linux refers to these as directories; folder and directory are used interchangeably in this class.

Course files: The directory `/home/students/Classes/201` is the home directory for this class *on the lab computers*. The same materials and sample files you can find through the Web (Moodle links go into this space) reside here. Lecture notes (sample programs), lab assignments, all are here. Another way to have gotten *this* document would have been to copy `/home/students/Classes/201/Labs/Lab01/lab01b.pdf` to your home directory (or just open it with the a PDF reader such as `evince`).

The directory path `~` is special: it is the home directory of the currently logged in user. So, the above prompt is saying that Dr. Ladd is in his own home directory. This is where the terminal program starts. It is the folder where you save all of your work (or rather, where you put subdirectories to save your work).

Your home directory follows you when you log onto any lab machine. On `haystack` or `algonquin` or any other lab machine *your* home directory is automatically mounted when you sign on.

Note that the _ after the prompt represents where you can type. It is not shown in other examples.

Show your work on Checkpoint 4 to the lab monitor; have them sign off on it before continuing.

Checkpoint 5 Create a CS1 and a Lab01 folder.

To create a folder, you can use the `mkdir` (make directory) command with the name of the new directory. To change the current directory, you can use the `cd` (change directory) command. To create a `CS1` directory and then, in that directory create a `Lab01` directory, the commands are:

```
laddbc@haystack:~$mkdir CS1
laddbc@haystack:~$cd CS1
laddbc@haystack:~/CS1$mkdir Lab01
laddbc@haystack:~/CS1$cd Lab01
laddbc@haystack:~/CS1/Lab01$ls -a
. ..
laddbc@haystack:~/CS1/Lab01$
```

Notice how the current directory changes as you navigate into directories below the home directory. Also, notice that in the newly created directory `Lab01`, when all of the files are listed (`ls` for listing; `-a` to tell the listing program to list all), there are already two items: `.` and `..`. These are the current directory (any directory refers to itself with a single dot) and the parent of the current directory (any directory refers to the directory it is contained in with two dots).

How would you change the current directory back to `/CS1`?

Show your work on Checkpoint 5 to the lab monitor and answer the question above for them. Have them sign off on it before continuing.

Checkpoint 6 Starting and stopping `emacs`

CIS 201 uses the `emacs` text editor. A *text editor* is a lot like a word processor. The primary difference is that while the displayed text may have different colors and different fonts, the text editor saves *just the characters* and not their formatting. This is important because the Java compiler doe not understand bold face or italics.

To run `emacs`, you can either type the name of the program into the shell *or* you can select `emacs` from the applications menu on the desktop. When you type the name in, you can also specify the name of a file you wish to edit. Let us assume that we want to edit a file in `Lab01` called `NewtonsApple.java` (notice the `.java` at the end; the Java compiler requires that if we are going to compile our Java code). We can just type:

```
laddbc@haystack:~/CS1/Lab01$emacs NewtonsApple.java &
 [1] 1234
laddbc@haystack:~/CS1/Lab01$
```

What is that `&` doing there? It is a special character telling the shell to run the command given but not to wait until it is finished. Before, when we listed the files, the next shell prompt didn't appear until the command was finished. Here we have the next shell prompt *and* `emacs` is running. The `[1] 1234` is an indication that we are running one command in the background and what *process id* the operating system gave `emacs`; ignore that number for the moment.

If you look at the top of the `emacs` window, there is a standard menu bar. If you look in the **File** menu, the last item is **Exit Emacs**. Also in that directory are the **Save File** and **Save File As** commands. Those commands have their keyboard equivalents written next to them in the menu; **C-** in front of a character means press the **Ctrl** key while pressing the key.

Notice in the top line menu that there is also an entry for **JDE**. This is short for Java Development Environment, an `emacs` extension that makes it a little easier to write and compile Java.

It is time to write some Java code. Type the following into your `emacs` window and save it:

```
1 import fang2.core.Game;
2
3 public class NewtonsApple
4     extends Game {
5 }
```

Having saved the file, you can now compile it. You can select **JDE | Compile**. The blank line at the bottom of the `emacs` window (known as the *minibuffer* in `emacs` parlance) says something about starting the "Beanshell", then the main window splits and a window labeled **JDEE Compile Server** appears with the output of the compilation. Upon success the output looks like this (`-classpath` line broken across 3 line for readability):

```
CompileServer output:

-classpath /home/faculty/laddbc/CS1/Lab01:
           /usr/local/share/java/jdk1.6.0/lib/fang2.jar
           /home/faculty/laddbc/CS1/Lab01/NewtonsApple.java

Compilation finished at Sat Aug 23 16:37:57
```

To get rid of the split in the window you can, with the cursor in the top window, select **File | Unsplit Window** (or use the keyboard shortcut of **C-x 1** to get just one (1) window).

Go over to your shell window and run `ls -a` again:

```
laddbc@haystack:~/CS1/Lab01$ls -a
. .. NewtonsApple.class NewtonsApple.java semantic.cache semantic.cache~
```

The two `NewtonsApple` files are the compiled file (`.class`) and the Java source file (`.java`, the one we edited and saved from `emacs`). Ignore the `semantic.cache` file (created by `emacs` when it edits source code).

Show your work on Checkpoint 6 to the lab monitor; have them sign off on it before continuing.

Checkpoint 7 Run the program

Now, run the program. That is, in `emacs`, after the compile step succeeds, run the program by selecting **JDE | Run App**. The window splits again and a new window appears labeled *NewtonsApple* with several buttons across the bottom. Nothing else happens at this point, so just press the **Quit** button.
Show your work on Checkpoint 7 to the lab monitor; have them sign off on it before continuing.

Checkpoint 8 Continue the program.

Now, modify your program by updating the contents of `NewtonsApple.java` to be:

```
1  import fang2.core.Game;
2  import fang2.sprites.OvalSprite;
3  import fang2.sprites.RectangleSprite;
4
5  public class NewtonsApple extends Game {
6    private OvalSprite apple;
7    private RectangleSprite newton;
8
9    public void setup() {
10     apple = new OvalSprite(0.10, 0.10);
11     apple.setColor(getColor("red"));
12     dropApple();
13
14     newton = new RectangleSprite(0.10, 0.10);
15     newton.setColor(getColor("green"));
16     newton.setLocation(0.5, 0.9);
17
18     addSprite(apple);
19     addSprite(newton);
20   }
21
22   public void dropApple() {
23     apple.setLocation(random.nextDouble(), 0.0);
24   }
25 }
```

Now when you save, compile, and run the program, an apple appears, just off the top of the screen, and a green Newton is visible below. **Start** starts the game but the game does nothing; there is no method to advance the game.

Correcting Typos: If you have any compiler errors, `emacs` shows them to you. Consider, for example, misspelling `Color` as `Colour`. Then the CompileServer output (in the bottom window) would look like this:

```
CompileServer output:

-classpath /home/faculty/laddbc/CS1/Lab01:
           /usr/local/share/java/jdk1.6.0/lib/fang2.jar
           /home/faculty/laddbc/CS1/Lab01/NewtonsApple.java

/home/faculty/laddbc/CS1/Lab01/NewtonsApple.java:11: cannot find symbol
symbol  : method setColour(java.awt.Color)
location: class fang2.sprites.OvalSprite
    apple.setColour(getColor("red"));
          ^
/home/faculty/laddbc/CS1/Lab01/NewtonsApple.java:15: cannot find symbol
symbol  : method setColour(java.awt.Color)
location: class fang2.sprites.RectangleSprite
    newton.setColour(getColor("green"));
           ^
```

```
2 errors

Compilation exited abnormally with code 1 at Mon Aug 25 15:48:00
```

`emacs` moves the cursor down to the first `/home/faculty/...` line, and also move the cursor in the `NewtonsApple.java` buffer to line 11 (the line with the error). The first line in the error window identifies the file and line with the error and the error the Java compiler encountered. In this case the error is `cannot find symbol`. The next line gives the symbol that could not be found (`setColour`) and the line after that gives the class where the Java compiler expected to find the symbol (`OvalSprite`).

The next two lines are the offending line from the `.java` file and a line with a `^` pointing to the spot where the Java compiler realized there was an error. In this case, we see that the method name in our listing was `setColor` though we typed `setColour`. Removing the "u" fixes the problem.

After correcting one error you can move to the next error by pressing `C-x ‘` (that is a back-tick, next to the `1` on most keyboards). The error message window and the source code window both move to the next error.

Show your work on Checkpoint 8 to the lab monitor; have them sign off on it before continuing.

Checkpoint 9 Finish `NewtonsApple`.

Here's the finished listing for Newton's Apple.

```
1  import fang2.attributes.Location2D;
2  import fang2.core.Game;
3  import fang2.sprites.OvalSprite;
4  import fang2.sprites.RectangleSprite;
5  import fang2.sprites.StringSprite;
6
7  public class NewtonsApple extends Game {
8    private OvalSprite apple;
9    private RectangleSprite newton;
10   private int applesCaught;
11   private int applesDropped;
12   private StringSprite displayScore;
13
14   public void setup() {
15     applesCaught = 0;
16     applesDropped = 0;
17
18     displayScore = new StringSprite(); // no text to display yet
19     displayScore.scale(0.10);
20     displayScore.setColor(getColor("white"));
21     updateScore(); // updated the content of the displayed score
22
23     apple = new OvalSprite(0.10, 0.10);
24     apple.setColor(getColor("red"));
25     dropApple();
26
27     newton = new RectangleSprite(0.10, 0.10);
28     newton.setColor(getColor("green"));
29     newton.setLocation(0.5, 0.9);
30
```

```
31      addSprite(apple);
32      addSprite(newton);
33      addSprite(displayScore);
34    }
35
36    public void advance(double secondsSinceLastCall) {
37      Location2D position = getPlayer().getMouse().getLocation();
38      if (position != null) {
39        newton.setX(position.x);
40      }
41
42      apple.translateY(0.33 * secondsSinceLastCall);
43
44      if (apple.intersects(newton)) {
45        applesCaught = applesCaught + 1; // another apple caught
46        updateScore();
47        dropApple();
48      }
49
50      if (apple.getY() >= 1.0) {
51        updateScore();
52        dropApple();
53      }
54    }
55
56    public void dropApple() {
57      apple.setLocation(random.nextDouble(), 0.0);
58      applesDropped = applesDropped + 1; // another apple dropped
59    }
60
61    public void updateScore() {
62      displayScore.setText("Score: " + applesCaught + "/" + applesDropped);
63      displayScore.setLocation(displayScore.getWidth() / 2,
64                               displayScore.getHeight() / 2);
65    }
66  }
```

Modify your program, save, compile, and run `NewtonsApple`. Save your work, compile it, correct any compilation errors, and run it. You should have a working game running.
Show your work on Checkpoint 9 to the lab monitor; have them sign off on it before continuing.

Log off of the lab computer you are using before leaving they lab. Anyone entering the lab has unlimited access to your files if you remain logged on. **DO NOT** turn off lab computers! They are a shared resource and there might be someone else logged in to "your" machine.