CIS 201 – Computer Science I
Laboratory Assignment 1b

- Learn how to create and navigate directories
- Learn how to start (and stop) the `emacs` text editor
- Learn how to create a `.java` file
- Learn how to compile a `.java` file
- Learn how to run a `.java` (or rather, `.class`) file
- Learn how to correct *compiler errors*

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. In this class we will be using 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:

```
uuuu@admin-156-xx:~$ _
```

The bit before the `@` is your login name (here displayed as `uuuu`). 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 has a concept of 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 folder, a container for files and folders. For historic reasons, Linux tends to refer to these as directories; these two terms will be used interchangeably in this class.

The directory path displayed as `~` is special: it is *your* home directory – the directory of the currently logged in user. So, the above prompt is saying that you (user `uuuu`) are currently "in" your own home directory. This is where all of your terminal sessions will start. It is the folder where you will save all of your work (or rather, where you will make subdirectories to keep your work).

Note that the _ after the prompt represents where you can type commands. We will not show this in the remaining examples. Indeed, we will normally just show the `$` prompt and omit the stuff the precedes it.

Create a CS1 and a Lab01 folder

Don't type any commands yet until you are asked to do so! Just read the following description.

To create a folder, you can use the `mkdir` (make directory) command followed by the name you want to give your new directory. For example, if you wanted to create a new directory with the name `foo`, your command would look like this:

```
$ mkdir foo
```

There must be at least one space between the command `mkdir` and the name of the directory you want to create, in this case `foo`.

The new directory will be created *as a subdirectory of your current directory*.

Once you have created this subdirectory, you can use the `cd` (change directory) command followed by the name of the subdirectory you just created to make the subdirectory your new current directory. For example, if you wanted to enter the newly created directory `foo`, your command would look like this:

```
$ cd foo
```

Your shell prompt (the stuff preceding the $ sign) will display the fact that your current directory is now `~/foo`.

You can use the `cd` command to change to *any* directory in the file structure, provided that you have permission to do so.

You haven't typed anything yet, have you? Good! Now you are ready to type in your first shell commands.

To create a CS1 directory (where you will keep all your CS1 sample programs, labs, and other notes), change to that directory and then create a Lab01 directory (and change to that directory), type in the following commands exactly as shown here:

```
$ mkdir CS1
$ cd CS1
$ mkdir Lab01
$ cd Lab01
$ ls -a
 . .. <- don't type this!!!
```

Notice how the current directory changes in your shell prompt as you change directories.

The `ls -a` command displays two entries, called "dot" and "dot-dot". These are actually directories: "dot" refers to the *current directory* (`Lab01` in this case) and "dot-dot" refers to the *parent directory* (`CS1`) of the current directory. You can treat these directory names just like anything else. For example, if you enter the command

```
$ cd .
```

(be sure to type the dot after the `cd`), you will "change" directory to the current directory – in other words, you will stay at the same directory. This isn't a particularly useful command, but it does show that "dot" is a real directory name.

Checkpoint 3

Show us your progress at this point.

If you are in the `Lab01` directory, how would you change the current directory back to `CS1`?

Starting and stopping `emacs`

In CS1 we will be using the `emacs` text editor. A *text editor* is a lot like a word processor. The primary difference is that while the displayed text in a word processor may have different colors and differentfonts and the like, a text editor saves just the characters and not their formatting. This is important because the Java compiler does not understand bold face or italics or anything like that. It understands just plain characters.

Don't type anything yet until you are directed to do so!

To run `emacs`, you can either type the command `emacs` into the shell *or* you can select `emacs` from the applications menu on the desktop. When you use the `emacs` command in the shell window, you can also specify the name of a file you wish to edit.

Let us assume that your current directory is `Lab01` and you want to edit a file called `NewtonsApple.java` (notice the `.java` at the end: the Java compiler requires that if we are going to compile our Java code). In your shell window, you would type the command

```
$ emacs NewtonsApple.java &
```

What is that `&` doing there? It is a special symbol that tells the shell to run the command (`emacs` in this case) but not to wait until it is finished. Before, when we listed the files (for example, with

`ls`), the next shell prompt didn't appear until the command was finished. When you use the `&` at the end of the command, the next shell prompt appears *and* `emacs` starts to run. *Remember to use the & whenever you run `emacs`.*

It is time to write some Java code. Start editing the file `NewtonsApple.java` in emacs by typing the following command in your shell window:

```
$ emacs NewtonsApple.java &
```

After your `emacs` window finishes opening, you will see a fairly standard menu line at the top of the window. If you open the **File** drop-down menu by clicking on it, you will see that the last item is **Exit Emacs**. Also in that drop-down menu are the **Save File** and **Save Buffer 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 character on the keyboard. (There is a diskette icon in the icon bar below the main menu bar; clicking on this is the same as choosing `File->Save`.)

Notice that there is a menu bar entry for **JDE**. This is short for Java Development Environment, an `emacs` extension we use that makes it a little easier to write and compile Java.

Entering your first program in `emacs`

Now type the following into the text entry part of your `emacs` window `emacs` window and `Save` it. **Type this in exactly as it appears here**.

```
import fang.core.Game;

public class NewtonsApple
    extends Game {
}
```

Having saved the file, you can now compile it. You can select `JDE->Compile` to do this. The blank line at the bottom of the `emacs` window (known as the *minibuffer* in `emacs` parlance) will say something about starting the "Beanshell", then the window will split and a sub-window labeled **JDEE Compile Server** will appear with the output of the compilation. If all went well, the output will look something (but probably not exactly) like this:

```
CompileServer output:

-classpath <stuff delted ... >

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

To get rid of the split in the window so your program text occupies the entire window, position the cursor in the top window and select `File->Unsplit Window` (or use the keyboard shortcut of `C-x 1`).

Position your cursor in your shell window and run the directory lister `ls` again:

```
$ ls
NewtonsApple.class NewtonsApple.java semantic.cache semantic.cache~
```

(You may or may not see the `semantic.cache` entries. Not to worry.) When we compiled the program in `emacs`, the Java compiler (we'll get to the Java compiler command later on) created a file called `NewtonsApple.class`. You can ignore the `semantic.cache` files – they are created by `emacs`.

Run your program

Now, run your program. To do this in `emacs`(assuming that your compile step completed successfully and without error), select `JDE->Run App`. The window will split again and a new game window will appear labeled `NewtonsApple` with four big buttons across the bottom labeled `Start`, `Sound`, `Help`, and `Quit`. Nothing else should happen at this point, so just click on the `Quit` button in this game window. The game window should then disappear

Checkpoint 4

Show us your work at this point. In particular, show us the game window that appears when you run your program within `emacs`.

Add objects to your game window

Now, modify your program by updating the contents of `NewtonsApple.java` to be as shown here. You can simply go back to your `emacs` window and make these changes and additions.

```
import fang.core.Game;
import fang.sprites.OvalSprite;
import fang.sprites.RectangleSprite;

public class NewtonsApple extends Game {
  private OvalSprite apple;
```

```
  private RectangleSprite newton;

  public void setup() {
    apple = new OvalSprite(0.10, 0.10);
    apple.setColor(getColor("red"));
    dropApple();

    newton = new RectangleSprite(0.10, 0.10);
    newton.setColor(getColor("green"));
    newton.setLocation(0.5, 0.9);

    addSprite(apple);
    addSprite(newton);
  }

  public void dropApple() {
    apple.setLocation(random.nextDouble(), 0.0);
  }
}
```

Now when you save, compile, and run the program, you will see a red “apple” (a circle, partially off the top of the screen) and a green “Newton” (a square box, at the center bottom of the screen). Pressing the `Start` button on the game box will not do anything (yet). Be sure to `Quit` the game before you proceed.

Correcting typos

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

```
CompileServer output:

... snip ...
/home/student/uuuu/CS1/Lab01/NewtonsApple.java:11: cannot find symbol
symbol  : method setColour(java.awt.Color)
location: class fang.sprites.OvalSprite
    apple.setColour(getColor("red"));
          ^
/home/student/uuuu/CS1/Lab01/NewtonsApple.java:15: cannot find symbol
symbol  : method setColour(java.awt.Color)
location: class fang.sprites.RectangleSprite
```

```
    newton.setColour(getColor("green"));
          ^
2 errors

Compilation exited abnormally with code 1 at ...
```

emacs will move its block cursor down to the line in your program text (the top window of emacs) where your first error was found: in this case, to line 11 (the line with the error). In this example, the error is `cannot find symbol`. The next line of the error display 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 ctrl-x followed by a back-tick, which is key to the left of the `1` key on most keyboards). The error message window and the source code window will both move to the next error.

Checkpoint 5

Show us your work at this point. In particular, show us your game window when you run your program.

Finish `NewtonsApple`

Here's the complete code for Newton's Apple. Make the appropriate changes in your emacs buffer.

```
import fang.attributes.Location2D;
import fang.core.Game;
import fang.sprites.OvalSprite;
import fang.sprites.RectangleSprite;
import fang.sprites.StringSprite;

public class NewtonsApple extends Game {
  private OvalSprite apple;
  private RectangleSprite newton;
  private int applesCaught;
  private int applesDropped;
  private StringSprite displayScore;
```

```
public void setup() {
  applesCaught = 0;
  applesDropped = 0;

  displayScore = new StringSprite(); // no text to display yet
  displayScore.scale(0.10);
  displayScore.setColor(getColor("white"));
  updateScore(); // updated the content of the displayed score

  apple = new OvalSprite(0.10, 0.10);
  apple.setColor(getColor("red"));
  dropApple();

  newton = new RectangleSprite(0.10, 0.10);
  newton.setColor(getColor("green"));
  newton.setLocation(0.5, 0.9);

  addSprite(apple);
  addSprite(newton);
  addSprite(displayScore);
}

public void advance(double secondsSinceLastCall) {
  Location2D position = getPlayer().getMouse().getLocation();
  if (position != null) {
    newton.setX(position.x);
  }

  apple.translateY(0.33 * secondsSinceLastCall);

  if (apple.intersects(newton)) {
    applesCaught = applesCaught + 1; // another apple caught
    updateScore();
    dropApple();
  }

  if (apple.getY() >= 1.0) {
    updateScore();
    dropApple();
  }
}

public void dropApple() {
  apple.setLocation(random.nextDouble(), 0.0);
```

```
    applesDropped = applesDropped + 1; // another apple dropped
  }

  public void updateScore() {
    displayScore.setText("Score: " + applesCaught + "/" + applesDropped);
    displayScore.setLocation(displayScore.getWidth() / 2,
                             displayScore.getHeight() / 2);
  }
}
```

Save your work, compile it, correct any compilation errors, and run it. You should have a working game running.

Checkpoint 6

Show us your working program.