Fall 2008 CS1 Family Weekend Lab

September 20, 2008

- 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 simple *compiler errors*

Checkpoint 1. 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:

```
laddbc@admin-156-97:~$ _
```

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*. The ~ is Linux shorthand for your home directory.
Note that the _ after the prompt represents where you can type. It will not be shown in other examples which will show commands typed in after the prompts.
The Linux file system uses hierarchical *directories* (sometimes called *folders*), containers for files and other directories. You will now change to the `Lab1` directory using the `cd` (change directory) command:

```
laddbc@admin-156-97:~$ cd Lab1
laddbc@admin-156-97:~/Lab1$ ls -a
. .. NewtonsApple.java
laddbc@admin-156-97:~/Lab1$
```

The current directory, displayed in the command prompt, changes as you navigate into directories below your home directory. The `ls -a` command tells the shell to list all files and directories in the current directory. `.` and `..` are special names for the current and parent directories. `NewtonsApple.java` is the Java program template you will extend in this lab.
How would you change the current directory back to ~?

Checkpoint 2. Starting and stopping `emacs`
In CS1 we use a *text editor* to enter our Java code; a text editor is like a word processor except that the text editor does not save formatting information. Instead it saves just the characters. This is important because the Java compiler does not understand formatting.
To run the `emacs` text editor, you can either type the name of the program into the shell. When you type the name in, you can also specify the name of a file you wish to edit. We can type:

```
laddbc@admin−156−97:~/CS1/Lab1$ emacs NewtonsApple.java &
[1] 1234
laddbc@admin−156−97:~/CS1/Lab1$
```

What is that `&` doing there? It runs the command in the background so we can keep using the shell. The following line is information about the background process.
`emacs` has a fairly standard menu bar. Under **File** there are commands to open files, save files, and exit `emacs`. The keyboard shortcuts are also listed (**C**- in front of a character means press the **Ctrl** key while pressing the key).
The **JDE** menu is the *Java Development Environment*. That is where the commands to compile and run Java programs can be found.
Look at the code. The colors and indentation are provided to make it easier for you to read. In Java, many things are held in *blocks* or containers between curly braces (`{ }`). To emphasise this relationship, lines end with opening curly braces and lines contained within that block are indented.
The lines that appear purple and have `//` or `/*` and `*/` around them are comments. They are written by programmers for other programmers. Java treats them as empty lines when compiling.
Enough preliminaries! It is time to compile a program.
Select **JDE** | **Compile**. The blank line at the bottom of the `emacs` window (known as the *minibuffer* in `emacs` parlance) reports something about starting the "Beanshell", then the window splits horizontally and a window labeled **JDEE Compile Server** appears with the output of the compilation. If all went well, the output will look something like this:

```
CompileServer output:

-classpath /home/faculty/laddbc/CS1/Lab1:/home/faculty/laddbc/jar/fang.jar
/home/faculty/laddbc/CS1/Lab1/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).

Checkpoint 3. Run the program
If your program compiled without errors, select **JDE** | **Run App**. The window splits again with Java output in the second pane and a new window appears labeled *NewtonsApple*. This is the game. The buttons along the bottom row are provided by FANG for all games.

Checkpoint 4. Continue the program.
Update your program file by finding the comments given in each of the examples below and entering the code shown just below them. Most of the comments (and code) in this section belong in the `setup` method though two new methods are defined at the bottom of the class.

•**dropApple** We must create a new command for our game. The command, called `dropApple`:

```
    /**
     * New command to drop a new apple. Positions apple at top of
     * screen and increments the number of apples that have been
     * dropped.
     */
    public void dropApple() {
        apple.setLocation(randomDouble(), 0.0);
        applesDropped = applesDropped + 1;
    }
```

Go ahead and try compiling; the compiler is like a spell checker that you should run early and often! You should probably save your program and compile after each of these items.

•**apple** The apple is an `OvalSprite` with equal diameters, making it a circle. It is colored red and then dropped (using our new command from above). All sprites must be added to the game.

```
        // initialize the apple: OvalSprite constructor takes an
        // x-diameter and a y-diameter (to make a circle, make them
        // the same). They are expressed in screens. getColor can find
        // named colors (here ”red”, ”green”, and ”white” are used)
        apple = new OvalSprite(0.10, 0.10);
        apple.setColor(getColor(”red”));
        dropApple();
        addSprite(apple);
```

•**newton** A square (same dimension both ways) the same size as the apple. Colored green, positioned at the bottom of the screen, and added to the game.

```
        // newton is small, green, and begins at the middle bottom of the
        // screen.
        newton = new RectangleSprite(0.1, 0.1);
        newton.setColor(getColor(”green”));
        newton.setLocation(0.5, 0.9);
        addSprite(newton);
```

•**updateScore** A new command that puts the number of apples dropped and the number of apples caught into the sprite on the screen and adjusts its position to keep it in the upper-left corner.

```
    /**
     * Update displayScore sprite with current score (apples caught /
     * apples dropped).  Adjust position to make sure it is all
     * visible.
     */
    public void updateScore() {
        displayScore.setText(”Score: ” + applesCaught + ”/” + applesDropped);
        displayScore.setLocation(displayScore.getWidth() / 2,
                                               displayScore.getHeight() / 2);
    }
```

•**score counters** Initialize the number of apples dropped and caught to `0`; note that this is before calling either of the new commands (both of which use the score variables).

```
        // initialize the score
        applesCaught = 0;
        applesDropped = 0;
```

•**displayScore** Create the on-screen score display, set its color, and use `updateScore` to update its contents.

```
        // The score is in the upper left corner of the screen in white
        displayScore = new StringSprite(0.10);
        displayScore.setColor(getColor("white"));
        updateScore();
        addSprite(displayScore);
```

Now, after you compile the last bit, you can again run the program (and get something reasonable). The apple appears partially off the top of the screen, the score in the upper-left corner, and the Newton at the bottom center of the screen.

The game is not yet playable; if you press **Start** you will find that nothing happens. This is because FANG calls the `advance` method as part of the main game loop and we have nothing but comments in that method.

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 like this:

```
CompileServer output:

-classpath /home/faculty/laddbc/CS1/Lab1:/home/faculty/laddbc/jar/fang.jar
/home/faculty/laddbc/CS1/Lab1/NewtonsApple.java
/home/faculty/laddbc/CS1/Lab1/NewtonsApple.java:11: cannot find symbol
symbol  : method setColour(java.awt.Color)
location: class fang.sprites.OvalSprite
    apple.setColour(getColor("red"));
         ^
/home/faculty/laddbc/CS1/Lab1/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 Mon Aug 25 15:48:00
```

`emacs` will move 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 will both move to the next error.

Checkpoint 5. Finish `NewtonsApple`
To finish `NewtonsApple`, we just find the appropriate comments in `advance` and add the code below to move newton, move the apple, and check if the apple has been caught or hit the ground.

- **Move newton** Set `newton` horizontal location to match the player's mouse's location.

```
        // (1) Move newton so his x-coordinate matches x-coordinate of mouse
        Location2D position = getPlayer().getMouse().getLocation();
        if (position != null) {
            newton.setX(position.x);
        }
```

- **Move apple** Move `apple` down according to its falling velocity (in screens).

```
        // (2) Translate apple down screen; velocity * time = distance
        apple.translateY(0.33 * secondsSinceLastCall);
```

- **Catch?** If `newton` gets the `apple`, update score and drop a new apple.

```
        // (3) Check if newton has "caught" the apple
        if (apple.intersects(newton)) {
            applesCaught = applesCaught + 1;
            updateScore();
            dropApple();
        }
```

- **Splat?** If `apple` hits the bottom of the screen, drop a new apple.

```
        // (4) Check if the apple has hit the ground (y-coordinate 1.0)
        if (apple.getY() >= 1.0) {
            updateScore();
            dropApple();
        }
```

Checkpoint 6. Save, compile, and run `NewtonsApple`.
Save your work, compile it, correct any compilation errors, and run it. You should have a working game running. Normally we would now upload the program to the on-line course management system.

Checkpoint 7. Make `NewtonsPear` (If you have time)
Open a new file (in the same directory) called `NewtonsPear.java`. Newton's pear is just like Newton's apple but the pear is yellow rather than red.

- Use **File** | **Insert File...** in the new buffer. Select `NewtonsApple.java` to insert.
- Save the file.
- The file must have the name of the class updated. The name of the class *must* match the name of the file (less the `.java`). You can go to **Edit** | **Search** | **Replace**and replace `NewtonsApple` with `NewtonsPear`.
- Similarly, the variable `apple` should be renamed `pear`. Use the same command to replace one with the other.
- Finally, lets fix up line 16 so that instead of setting the color to "red", the color of the oval is set to "yellow".

Now save and compile your second program. It should look familiar but with a small difference.