6.005 Elements of Software Construction
Fall 2008
Lab 0.2: Object-Oriented Java, and More Tools

In this lab, you will learn about basic object oriented programming in Java, and get some experience with some new tools and techniques, including unit testing with JUnit and using Subversion with multiple developers.

Before Lab

Before coming to lab, please do the following:

- **Read the lab handout and required readings.** You won't have time in lab to read, so do it first.
- **Check out the `oop_java` module from your SVN repository.** Please review Lab 0.1 if you do not know how to do this.

Automated Unit Testing with JUnit

Readings:

- **(required)** Annotations
- **(required)** JUnit cookbook

JUnit is a widely-adopted Java unit testing library, and we will use it heavily in 6.005. A major component of the 6.005 design philosophy is to decompose problems into minimal, orthogonal units, which can be assembled into the larger modules that form the finished program. One benefit of this approach is that each unit can be tested thoroughly, independently of others, so that faults can be quickly isolated and corrected, as code is rewritten and modules are configured. Unit testing is the technique of writing tests for the smallest testable pieces of functionality, to allow for the flexible and organic evolution of complex, correct systems.

By writing thoughtful unit tests, it is possible to verify the correctness of one's code, and to be confident that the resulting programs behave as expected. In this lab, you will learn the basic vocabulary of JUnit, how to run existing tests, and how to write new ones. In 6.005, we will use JUnit version 4.

The Anatomy of JUnit

JUnit unit tests are written method by method. There is nothing special a class has to do to be used by JUnit; it only need contain methods that JUnit knows to call, which will be referred to as test methods for the remainder of the lab. Test methods are specified entirely through *annotations*, which may be thought of as keywords (more specifically, they are a type of metadata), that can be attached to individual methods and classes. Though they do not themselves change the meaning of a Java program, at run-time other Java code can detect the annotations of methods and classes, and make decisions accordingly. The Java annotation system, judiciously used, can create dynamic and powerful code. Though we will not deeply explore annotations in 6.005, you will see how other libraries, such as JUnit, make effective use of them.

In the `test` package, you will find a pair of files, `FibonacciGenerator.java`, and `FibonacciTest.java`. `FibonacciGenerator.java` contains the implementation for a class that is a bidirectional generator of Fibonacci numbers. The `FibonacciGenerator` and the `FibonacciTest` classes are written to be used in an object oriented style. That is, a `FibonacciGenerator` is an object that maintains state in its fields (`first` and `second`), that can only be operated on by the methods exposed through the class (`next` and `previous`). `FibonacciTest` is written similarly, though it is intended to be used in a specific way, by the JUnit library.

Look closely at `FibonacciTest.java`, and note the `@Before`, `@Test`, and `@After` symbols, that precede method definitions. These are examples of annotations. The JUnit library uses these particular annotations to determine which methods to call when running unit tests. The `@Test` annotation denotes a test method; there can be any number in a single class. Even if one test method fails, the others will be run. The two test methods are very different. The `generateAndCheck` method contains calls to `assertEquals`, which is an assertion that compares two objects against each other and fails if they are not equal. Here is a list of the other assertions supported by JUnit. If an assertion in a test method fails, that test method returns immediately, and JUnit records a failure for that test.

The other test method, `callIllegalPrevious`, operates very differently, as it contains no assertions itself. Instead, the assertion is contained within the annotation itself! What the annotation expresses is that running the `callIllegalPrevious` method should result in an uncaught `IllegalStateException` being thrown. If you look at the `previous()` method in `FibonacciGenerator.java`, you can see how the exception can be thrown. For the `callIllegalPrevious` test method to succeed, that line must be executed.

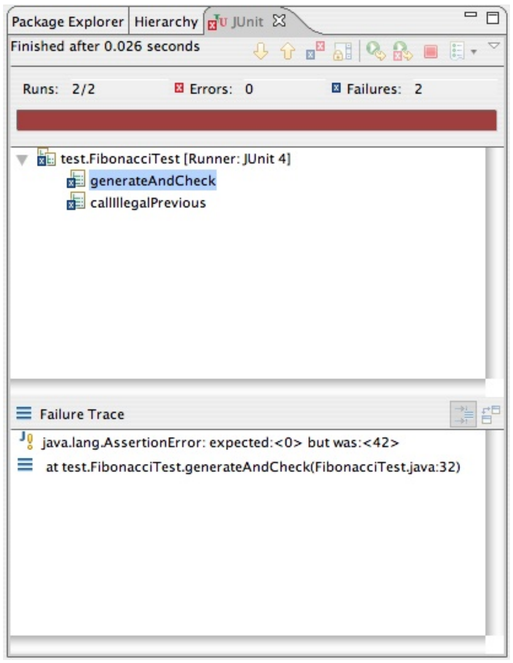
The final two annotations, `@Before` and `@After`, are easier to explain. Each denotes a method that is called either before or after, respectively, each test method is called. `@Before` methods are a good way to share common setup code betweeen multiple tests. `@After` methods can ensure that cleanup code runs, even if a test fails. For example, if you are testing code that writes to a temporary files on disk, you may want to ensure that the temporary files are deleted whether or not the tests fail.

Running Existing Tests

To run the tests in `FibonacciTest`, simply right click on the `FibonacciTest.java` file in either your

Package Explorer or Navigator view, and mouse-over the 'Run As' option. Click on the 'JUnit Test' option, and you should see the JUnit view appear, with a green bar indicating that all test methods ran successfully.

To see what a test failure looks like, try changing the initial value of `first` in `FibonacciGenerator` to something outrageous (like 42), then rerun the tests. You should now see a red bar in the JUnit view, and if you click on `generateAndCheck`, you will see a stack trace in the bottom box, which provides a brief explanation of what went wrong. In this case, it expected the first value of the generator to be 0, but it was actually 42 (or whatever value you chose). Double clicking on lines in the Failure Trace will bring up the code for the test that failed.



```
java.lang.AssertionError: expected:<0> but was:<42>
at test.FibonacciTest.generateAndCheck(FibonacciTest.java:32)
```

Courtesy of The Eclipse Foundation. Used with permission.

For a more thorough introduction, O'Reilly has a JUnit and Eclipse tutorial, with screen-shots to help you get acquainted with using JUnit from within Eclipse. The guide was written for JUnit 3, so the code samples use the older (but still supported) JUnit API.

Later in the lab, you'll write some unit tests of your own.

Object Oriented Programming in Java

Create a file named `oop_answers.txt` under the oop_java folder (in the same directory as `assignment.html`). Add it to the SVN repository (right-click on the file → Team → Add to Version Control) and put your answers to the questions in this section there.

1. **Warm-up: Light it up**

   Readings:

   - **(required)** Classes (through "Passing Information to a Method or a Constructor")
   - **(required)** More on Classes (through "Summary of Creating and Using Classes and Objects")
   - (optional) Lesson: Classes and Objects
   - (optional) The static keyword
   - (optional) The final keyword

   Java is an object oriented language. As such, Java programs are made up of classes. Each class has its own `.java` file with the same name.

Take a look at `Light.java`. Note that it contains a single class named Light, which represents a simple light that can be on or off.

Its structure is:

```
public class Light {
    // Some fields

    // Some Constructors

    // Some Methods
}
```

The first line is the class declaration and gives the visibility of the class, the keyword `class`, and the name of the class.

Question: What changes would you have to make in order to rename the Light class to ShinyLight?

Just inside the class declaration are the field declarations. Fields are the variables that the class keeps track of.

They tend to look like this:

```
private typeName variableName;
```

You can put certain keywords between the visibility and the typeName that change the field's behavior. For example, the keyword `final` indicates that the field's value cannot be changed (with certain caveats if the value is a mutable Object). The keyword `static` turns the field into a class variable.

Since static fields are not tied to a particular instance, you can access them through *`Classname.fieldName`*. An example of this is the `Math.PI` and `Math.E` fields. Creating static final fields are a useful way to declare constants.

Question: What would happen if we changed Light's isOn field to be static and final?

A class's constructors are methods that are used to make a new instance of a class. `Light` has two different constructors. A constructor can access other constructors in the current class with the method `this()`.

Question: What happens when the no-argument constructor, `Light()` is called?

Methods are the operations provided by a class. They tend to look like:

```
public returnType methodName ( arg1Type arg1Name, arg2Type arg2Name, ... ) {
     // body of method
}
```

Methods can also have keywords between their visibility and returnType. Adding the keyword `static` to a method indicates that the method is not associated with a particular instance of the class. An example of this is `Math.max(a, b)`.

Question: Why can't static methods directly access non-static fields?

Exercise: Fix the problems in `Light.java`. We have included a JUnit test called `LightTest.java` to help with this. You can run this unit test the same way you ran the Fibonacci test, by right-clicking on `LightTest.java` in Packge Explorer or Navigator, then mousing-over 'Run As', then clicking on 'JUnit Test'. If you are using Eclipse and are paying attention to its warnings, you should be able to find at least one of the bugs without referring to the JUnit results.

2. **A light of a different color**

   Readings:

   - **(required)** Inheritance (through "Using the Keyword super")
   - (optional) Using Package Members

   Now take a look at `ColoredLight.java`. A `ColoredLight` is simply a `Light` that also has a `Color`. We use the `extends` keyword followed by `Light` in the method declaration to indicate that ColoredLight inherits from (is a subclass of) Light. We also import `java.awt.Color` for our color property.

   Exercise: Implement the constructor and methods for `ColoredLight`.

   Note that the method `ColoredLight.randomChange()` overrides the parent's method `Light.randomChange()`. However, a `ColoredLight` can still access its parent's version of the method by calling `super.randomChange()`. A subclass's constructor can access a parent's constructor the same way. For example, in `ColoredLight`'s constructors, one could call `super()` or `super(`*`boolean`*`)` in order to access `Light`'s constructors.

   Question: How might `ColoredLight`'s constructor have to change if the no-argument `Light()` constructor was not defined? If you are unsure, try deleting that first constructor in `Light.java` and see what errors come up in `ColoredLight.java`.

Exercise: Create a JUnit test for `ColoredLight` named `ColoredLightTest`. Use `LightTest.java` as a model. (To start the file, you can right-click on "lights" and select New → Class.)

3. **All in a row**

   Readings:

   - **(required)** What is an interface?
   - (optional) Interfaces (whole subsection through "Summary of Interfaces")

   We are now going to string some lights together to make light patterns. To do this we are going to define an interface, `HolidayLights`. This interface specifies that anything that implements it will have a method telling us how long this string of lights is (`getLength()`) and a method that will return a sequence of lights representing how the lights will look at the next timeslice (`next()`).

   Question: What's the difference between an interface and a class?

   Question: When might you use an interface instead of a class (that can then be subclassed)?

   The return type for `next()` is `List<Light>`. This is an example of using generics, a very useful Java language feature. In our case, it specifies that `next()` must return a `List` that only contains `Light`s.

4. **Running along now**

   Readings:

   - **(required)** Implementing an Interface
   - **(required)** Creating Objects
   - (optional) ArrayList javadocs
   - (optional) The for Statement (section about enhanced `for` loops)

   Now we're going to create a class that implements the `HolidayLights` interface. `RunningHolidayLights` is a fixed-length string of lights with exactly one light on at a time. The index of the light that is on will increase until it hits the end of the string and then start over from the front again.

   As a result, `RunningHolidayLights.next()` should generate a series of light sequences, all of fixed-length `n`. The first sequence it generates should have the first light on and all the rest off. The second sequence would have the second light on, and the rest off, and so on.

   Exercise: Implement the methods in `RunningHolidayLights`. Also create

`RunningHolidayLightsTest.java` to test your implementation. Make sure to test the constructor as well as all the methods.

You should take advantage of Java's `ArrayList` class, which implements the `List` interface. The line:

```
ArrayList<Light> lightList = new ArrayList<Light>();
```

creates a new `ArrayList<Light>`. Note that you must import the `ArrayList` class in order to use this. To do so, add

```
import java.util.ArrayList;
```

to the top of your file, after the package `lights;` declaration. In Eclipse, you can also hit **CTRL-SHIFT-O** to automatically **O**rganize your imports. Eclipse will then attempt to figure out what class(es) you mean to import and add them for you. If the name of the class that needs to be imported is ambiguous — for example, there is a `java.util.List` and a `java.awt.List` — then Eclipse will prompt you to choose one to import.

`HolidayLightsWindow` is a class we're providing that will give a visual representation of anything that implements `HolidayLights`. You should not have to edit any of the code. `Main` creates some `RunningHolidayLights` of length 12, puts them in a `HolidayLightsWindow`, and makes the window visible.

After you have finished implementing and testing `RunningHolidayLights`, run `Main.java` as a Java application (right-click on the file, go to "Run As" and then "Java Application") to see the lights!

5. (Optional) **Lights of your very own**

   Create a new class `MyHolidayLights`, which implements `HolidayLights` but displays a different pattern than `RunningHolidayLights`.

   To start you off, here are some suggestions:

   - Use `randomChange`.
   - Have multiple running lights.
   - Have blinking lights.
   - Have lights run from both ends.
   - Use colored lights.
   - Periodically change how the lights behave.

   Remember to think of what internal variables it needs to store and any convenience methods it might want to have.

You should also create corresponding MyHolidayLightsTests.

Afterwards, change the first line in Main.java's main method to instantiate a MyHolidayLights instead of a RunningHolidayLights. Since HolidayLightsWindow only depends on the HolidayLights interface, it should be able to display your new class.

✓**Checkpoint.** Find a TA or another member of the course staff and review your code and oop_answers.txt.

Exceptions

Readings

- **(required)** Exceptions
- (optional) HashMap javadocs
- (optional) Enum Types
- (optional) Lesson: Basic I/O (especially the I/O from the Command Line section)
- (optional) java.io javadocs

Create a file named exceptions_answers.txt under the oop_java folder. Add it to the SVN repository and put your answers to the questions in this section there.

Now take a look at the grades package.

LetterGrade is an enumeration of all the letter grades one can get.

InvalidGradeException is a *checked* exception that should be thrown if something that cannot become a valid grade is passed in where one is expected.

The heart of the program lies in GradeManager. Uncomment the main method. In Eclipse, you can do this by selecting the relevant lines and then typing **CTRL /** (or clicking **Source->Toggle Comment** in the menubar) to comment/uncomment the selected lines.

GradeManager's main method starts a loop that reads in input from System.in and does some action corresponding to that input. You can access System.in through the Console tab in Eclipse. (In the Java perspective, this tab is usually grouped with the Problems, Javadoc, and Declaration tabs below the code window.)

Question: What is the difference between checked and unchecked exceptions?

Exercise: Fix and finish implementing `GradeManager.main`. You should use a try/catch block.

Exercise: Implement `GradeManager.addGrade`. Do this *without* using a try/catch block.

Question: Currently `GradeManager.printHistogram` throws an exception but does not need to declare it in its method signature. Why not?

Exercise: Finish implementing `GradeManager`. Run it as a Java Application to see it at work.

Optional Exercise: Write some JUnit tests for `GradeManager`.

Optional Exercise: Add the capability for `GradeManager` to load and save grades.

Collaborative Development with Subversion

The following exercises should familiarize you with several important aspects of collaborative, multi-person development using a version control system such as Subversion.

Who, What, Where, When, and Why

In addition to storing your source code files and the changes made to them, Subversion stores information about who changed what, when, in which files—and if the person doing the changing was playing nice and writing good commit log messages, it can even tell you why the changes were made.

In the Package Explorer of Eclipse, browse to one of the files you modified and committed during today's lab, or during the previous lab. Right-click the file, and select "**Team → Show History**." A new view will appear (at the bottom of the Eclipse window, if you don't drag it somewhere else). After a few moments of discussion with the Subversion repository, that view will list all the revisions committed to the repository that affected the selected file.

Clicking on a particular revision will show you both the files that were added ("A"), modified ("M"), or deleted ("D") by that commit, as well as the commit comment written by the committer. Whenever you use version control to collaborate, write commit comments!

Shared Repositories

Speaking of using version control to collaborate... let's check out a new Subversion repository that, unlike your personal repository, is accessible to the entire class. Refer to the previous lab and the SVN documentation and add the following repository:

```
svn+ssh://athena.dialup.mit.edu/mit/6.005/svn/groups/oop_java/everyone
```

In that repository is a project named `collab_svn`—**check it out**.

Once you've checked out the project, return to the Java perspective and **run the `Main` class** in package `collab` (for example, by right-clicking and selecting "Run As... → Java Application").

The `Main` class `main` method looks for all the Java classes in the `collab` package, creates a new instance of each one, and calls the method `toString()` on each instance.

Some of your classmates may already have added their own classes to the `collab` package, but there should be at least one other class from one of the teaching assistants. Using the teaching assistant class as a template, **create a new class** in the `collab` package that is named with your Athena username (but follows the Java convention of CamelCaseCapitals for class names). One way to do this is to right-click "collab" and choose "Add → Class." Have the `toString()` method of this class `return` a `String` of your choosing. You should be able to run the `Main` class again and see your message in the list.

Updating

Right-click on the "collab_svn" project and choose "Team → Update." Alternatively, select the "Team → Synchronize with Repository" option to use the synchronization screen seen in the last lab, where you can see updates before they happen. Right-click and choose "Update" to update files or directories.

Updating gives you the latest version of something (a file, or a directory or directory tree full of files) from the repository. Relevant reading from the SVN documentation includes the sections on How Working Copies Track the Repository & Mixed Revision Working Copies.

Committing

Before continuing, **make sure you can run the `Main` class successfully** and see everyone's messages, because you are about to check in. As mentioned in the previous lab, checking in broken code will surely earn you the ire of other students who, should they update and receive your nonfunctional changes, will be unable to continue working.

Check in your work. Since you have created a new file, you need to add it to version control. **Eclipse does not automatically put new files into Subversion**; you have to tell it that you want to commit each new file. (Once you do that, all future changes you make to that file will be committed.) One way to add a new file to subversion is to right-click on it and select "Team → Add to Version Control". You also have a chance to add files using the list of checkboxes in the commit dialog itself.

When you try to check in, your attempt may fail with an error from Subclipse because your working copy is out of date. That means somebody else in the class committed since the last time you updated, so Subversion doesn't know what to do with the differences. Update your working copy, and try the commit again.

All is Well

Find a partner, and make sure both you and your partner have received each other's code by committing and updating; you should both be able to run the `Main` class and see your partner's message in the console.

Merging

With the help of your partner, you will now simulate a situation where the use of Subversion becomes more complex. Make sure both you and your partner have updated versions of the shared project. **Pick one of the Java files added by either you or your parter**. You will both edit this same file, but you will do it on different machines. One of you (say, Alice) should add a comment line near the top of Alice's copy of the file, right after the line "`package collab;`" The other person (Bob) should add a comment line at the end of Bob's copy of the file file. Be sure each person makes no other changes.

Pick one of you to commit first (say, Alice), and **commit**. Once Alice commits, Bob will **update**. Even though Bob has modified the same file Alice committed, because the changes were in different parts of the file, Subversion will merge the changes automatically. Now Bob can **commit**, and Alice should **update**. Once this process is done, both you and your partner should have identical source files with both of the extra comments.

Conflicts

While SVN does a good job of merging unrelated changes to the same source file, it is not magic.

In the same file you've been editing with your partner, both you and your partner should **modify the `String` returned by `toString()` to have *different* values**.

Pick one person to commit first. Once the change is committed, the other person should update, and you should both take a look at the result, which should contain a **conflict**.

First, notice that a blue-ish square-ish icon has been added to the conflicted file in the Package Explorer. This indicates that the file contains a conflict that Subversion should not merge automatically. Until you inform Subversion that the conflict has been manually resolved, removing that conflict icon, it will refuse to allow you to commit the file.

Second, several extra files have appeared, with the different bits of code Subversion couldn't fit together. (The SVN documentation on Resolve Conflicts (Merging Others' Changes) explains what these files are.)

To merge the conflict manually, right-click on the file and pick "**Team → Edit Conflicts**," which brings up a two-paned editor. On the left hand is your (local) code, and on the right is your partner's (committed) code.

Edit the left side to your mutual, conflict-resolving satisfaction, and save. Then do "**Team → Mark Resolved**." The extra files will be deleted, and the conflict icon will disappear. Check that the resolved version runs fine—don't break the build!—and commit it.

Fall Back!

One final note: the Replace With... menu may come in handy if you realize you need to fall back to a previous version of your work. Just remember to commit **working code** with **good commit comments**, and it should always be easy to restore order to any mess you find yourself in.

✓**Checkpoint.** Find a TA or another member of the course staff and review your work on exceptions and collaborative development.

Commit Your Solutions

This is the end of the lab. Be sure to commit your solutions to your personal Subversion repository, adding any files that you created.