6.005 Elements of Software Construction
Fall 2008

6.005 Elements of Software Construction
Fall 2008
Lab 0.1: Introductory Java

This lab introduces you to some of the basic language features of Java, focusing on procedural programming: using variables, control structures, and procedures, and working with primitive data like numbers, strings, and arrays. We assume that you have already seen these concepts in a previous course (possibly using Python or Scheme), so we can focus on how each concept is implemented in Java instead.

This lab also introduces various tools you will use throughout the semester, including the Eclipse integrated development environment and Subversion source code control system.

The exercises in this lab involve answering some questions and writing some code. The questions require only very brief answers, so there's no need to create additional files. Just write the answers as comments in the code.

Before the Lab Session

Before coming to the Friday lab session, you must do the following:

1. **Read the lab thoroughly.** Read this lab handout, and read all the links marked **Required Reading**. (Other links are provided as supplementary material for further reference or understanding; you don't need to read them in advance.) You will not have enough time during the lab session to read *and* do the exercises, so do the reading first.
2. **Install the software you'll need on your laptop.** You need to install three things for this lab:
   - **JDK 6** (for Windows or Linux; Mac users probably already Java, if not, get it here for 10.4). From this web page, download "JDK 6 Update 7"; you don't need NetBeans or Java EE.
   - **Eclipse 3.4**. Choose Eclipse IDE for Java, which will download a ZIP file. Unpack the ZIP file, go inside the resulting folder, and run Eclipse.
   - **Subclipse**. To install Subclipse, use Eclipse's Software Updates mechanism. Follow the installation instructions on the Subclipse website. Be sure to use the appropriate "update site" URL for Eclipse 3.4.

   Athena already has this software installed, so if you'll be using an Athena workstation in the lab, you don't need to install anything.

Eclipse

The Eclipse integrated development environment (IDE) is a powerful, flexible, complicated, and occasionally frustrating set of tools for writing, modifying, and debugging programs. It is especially useful for working in Java.

On Athena, Eclipse is in the `eclipse-sdk` locker; the command name is `eclipse`.

When you run Eclipse, you will be prompted for a "workspace," which is where Eclipse will store all the different projects you work on. On Athena, for example, the default location is a directory called `workspace` in your home directory. In addition to code, Eclipse stores its own "metadata" in hidden folders in the workspace. **You should not run more than one copy of Eclipse at the same time** with the same workspace, or the metadata will become corrupted.

The first time you run Eclipse, it will show you a welcome screen. Click the button to go directly to the "workbench" and you're ready to begin. For some useful tips and troubleshooting information, you can look at the 6.170 Eclipse Reference, but disregard the 6.170-specific notes there.

Subversion

In this exercise, you will learn about the Subversion version control system, and how to retrieve code from a Subversion repository using Eclipse.

O'Reilly publishes *Version Control with Subversion*, conveniently available online. This book describes what Subversion (or SVN) is and how to use it:

> Subversion is a free/open-source version control system. That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of your data, or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of "time machine".
>
> Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incorrect change is made to the data, just undo that change. [What is Subversion?]

If you have used other version control software, such as CVS, many of the concepts and procedures of SVN will be familiar to you. If not, there are two important ideas to learn: *repositories* and *working copies*.

Repositories

> Subversion is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a filesystem tree—a typical hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.
>
> [...] What makes the Subversion repository special is that it remembers every change ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.
>
> When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view previous states of the filesystem. For example, a client can ask historical questions like, "What did this directory contain last Wednesday?" or "Who was the last person to change this file, and what changes did he make?" [The Repository]

Every SVN repository has a URL at which it can be accessed.

Working Copies

In order to make changes to files in the repository, you must obtain (or "check out") a copy of the current version of those files:

> [... E]ach user's client contacts the project repository and creates a personal working copy—a local reflection of the repository's files and directories. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. [Versioning Models]

Any number of people can have any number of working copies (or "checkouts") of different parts of a single SVN repository. Those working copies might be on different machines and have different versions of files.

SVN in 6.005

In 6.005 you will be using a few different Subversion repositories...

- The "**published**" repository will contain example code presented in lecture. This repository can be accessed by anyone and does not require authentication. The URL for this repository is:
  `https://svn.csail.mit.edu/6.005/published`
  Don't forget the 's' in https, or the server will deny knowledge of this URL. Also note that you can visit this URL in your web browser, and see the latest version of any file in the repository—great for quickly bringing up an example you're looking for.
- You will have your own **personal repository**, accessible only to you, for use in the labs. The URL for this repository is:
  `svn+ssh://athena.dialup.mit.edu/mit/6.005/svn/users/<username>`
  Before you can check out this repository, you'll need to do some extra setup in your Athena account, described below.
- You will have a **group repository** for each project you work on during the course. The URLs for these will look something like:
  `svn+ssh://athena.dialup.mit.edu/mit/6.005/svn/groups/<project>/<section>/<group>`

A more common structure for SVN would be to have all these various activities—different users, different groups, staff machinations—sharing the same repository. However, in order to have different access privileges and also authenticate users with Athena (and CSAIL) logins, we're using this multitude-of-repositories structure.

Setup

The user and group repository URLs use the "svn+ssh" protocol, in which access to the SVN repository is tunneled through SSH (compare to the "published" repository, where access is via HTTP). In order for this to work, you need to set up your Athena account with access to the `6.005` and `svn` lockers.

Log in to Athena at a workstation or remotely. Create or edit the file `~/.environment` (that's the file in your home directory called "environment" with a dot in front of it) and have it add the necessary lockers with the command:

```
add svn eclipse-sdk 6.005
```

If you use the bash shell on Athena, you should edit `~/.bash_environment` instead.

Save this file, log out of Athena, log in again, and check that this has worked by first running the Subversion command line client:

```
svn
```

Which should reply with:

```
Type 'svn help' for usage.
```

Now check that you can see your own repository by running:

```
svn info file:///mit/6.005/svn/users/your-username-here
```

You should see a reply that contains various information about your 6.005 repository, and not any sort of error messages.

SVN in Eclipse

Eclipse has built-in support for working with version control systems, but does not include specific support for SVN. Subclipse, a plug-in for Eclipse, provides this support, allowing you to check out files from a repository, check them in, and use all the other features SVN provides. You can read the Subclipse documentation, also available inside the Eclipse help viewer, to learn more about the features of Subclipse.

On Windows and OS X, there is one extra step you need to take to configure Subclipse. Go to the SVN preferences page ("Window → Preferences... → Team → SVN") and under "SVN interface," choose "SVNKit (Pure Java)." The "JavaHL (JNI)" option will only work if you have a command line Subversion client installed in your $PATH.

Connecting to Repositories

Working with Subclipse begins in the "SVN Repository Exploring" perspective (a perspective in Eclipse is a particular set and arrangement of interface panels). After you install Subclipse and restart Eclipse, switch to this perspective by following their instructions.

We'll begin with your personal Subversion repository, where you will work on the labs in 6.005, including the current lab. Follow the Subclipse documentation for Creating a New Repository Location, with the URL:

```
svn+ssh://athena.dialup.mit.edu/mit/6.005/svn/users/your-username-here
```

Note that to enter the "SVN Repository View" you should click the "Open Perspective" button in the upper right corner of the Eclipse window (it is a small window icon with a plus) and select the "Other..." option. You'll be presented with a dialog listing a number of perspectives, including "SVN Repository Exploring", the view you want.

Checking Out Projects

Checking out a project from a repository once you've added it is straightforward, if that project was originally created in Eclipse. Right now, we'll check out the code for today's lab. You may be asking: where did this code come from, if it's in my personal repository!? And even if you weren't asking, you should still know that for each of the labs and projects, the staff will start you off by providing a skeleton Eclipse project, which you can check out, complete, and check in. These skeleton projects will be added right into your personal or group repository by the

staff.

Navigate to the `procedural_java` folder in your repository, right click it, and select "Checkout..." (the documentation on this is under Checking out a Project). You should accept the name of the project unchanged, for this and all labs and all group projects.

If Eclipse does not prompt you, switch to the Java perspective with the menu in the top right of the workbench window.

Basic Computation

Required Reading (from the Java Tutorial):

- Hello World!
- Primitive Data Types
- Assignment/Arithmetic Ops
- Equality/Relational/Conditional Ops
- Math class
- Defining methods
- Calling methods
- Returning a value from a method

Look at the code below which calculates the value of investing an initial sum of money at a specified interest rate and for a specified number of years. Open the source file `FinancialCalc.java` (which you'll find inside the folder "(default package)" in Eclipse).

```
class FinancialCalc {

  public static void main(String[] args) {
    double principal = 1000.00;    // $1000 initial investment

    double interestRate = 0.035;   // 3.5% interest rate
    int numOfYears = 7;            // investment length is 7 years

    double finalValue = 0.0;
    finalValue = principal * Math.pow((1 + interestRate), numOfYears);

    System.out.println("Investing $" + principal +
                       " at an interest rate of " + (interestRate*100) + "%" +
                       " for " + numOfYears + " years" +
                       " will have a final worth of $" + finalValue);
  }

}
```

Run the code in Eclipse to see what it does. Use Run → Run As... → Java Application, and look for the output in the Console pane at the bottom of the screen.

Change the program so that it calculates the final value in a separate method `computeFinalValue`, which should be a function of three arguments (the principal, interest rate, and number of years).

```
static double computeFinalValue (double principal, double interestRate, int numOfYears) {
  /* YOUR CODE HERE */
}
```

Your `main` method should still print the same result on the output.

In general, the final value (***V***) of an investment (principal ***P***) compounded yearly for ***Y*** years at interest rate ***I*** is given by the formula:

$$V = P\ (1 + I)^Y$$

Given any three of these variables, you can write code that computes the fourth. **Write methods that compute each of the variables** (*P*, *I*, and *Y*) when given values for the other three, and demonstrate by having your `main` method print the following:

1. The amount of money you should invest today (at an interest rate of 5%) to have a total amount of $1000.00 at the end of 4 years.
2. The interest rate you need to turn an initial investment of $500.00 into $525.00 at the end of 3 years (*hint: be careful when carrying out division on integers! Cast the integer to a double when necessary*).
3. The number of years you need to invest an initial sum of $100.00 at an interest rate of 4.4% to have a final value of $150.00 (*hint: the number of years is not necessarily an integer*).

Checking In Changes to Subversion

Now that you've changed some code and tested your changes, let's save the changes back to your personal Subversion repository. This process is also known as "committing" or "checking in".

You should always remember to commit your changes to the repository at the end of lab. In fact, it's not a bad idea to commit after every exercise, as soon as you're happy with the code you've written. When you're working on a team, you'll need to commit your code regularly—until you commit, no one else sees the changes you've made. It is, however, not a good idea to commit code that prevents the program from compiling or that makes it less stable. This is called "breaking the build" as it is a major developer *faux pas*.

You should also remember to commit your projects by the deadline. Where other courses ask you to "hand in" your assignments, in 6.005 you will submit all your assignments by checking them into Subversion.

Before checking in changes, it is always good practice to review when you've done. Right-click on the `procedural_java` folder that holds your work for this lab, and select "Team → Synchronize with Repository." Eclipse will prompt you to open yet a new perspective that shows how your local working copy compares with the repository (see Team Synchronizing for more explanation). Grey right-facing arrows indicate files you've modified that need checking in.

Right-click the file you changed and select "Open in Compare Editor" to see a detailed differences, or "diff," view that shows exactly what has been changed. Supposing you're satisfied that this is a change you want to check in, right click the project and select "Commit..." if you're in the Team Synchronizing perspective, or "Team →

Commit..." from any other. Read the documentation for the Commit dialog box, where you should:

- Write a message that describes your commit, and
- Make sure the set of files to be checked in is what you want—for example, don't commit Java class files, just source files.

If all goes well, your changes should be committed to the repository. If you have any doubt about anything you've done up until now, or don't feel confident that your updated code was committed correctly, ask the staff. Otherwise—on with Java!

Control Flow

Required Reading (from the Java Tutorial):

- Expressions, Statements, Blocks
- If-then statements
- Switch statements
- While, do-while statements
- For statements
- Break/continue

Look at the method below which finds prime numbers, which is found in Eclipse in the source file `Primes.java`.

```
    static int findPrimes(int n, boolean printPrimes) {
        boolean isPrime = true;
        int numPrimes = 0;

        for (int i = 2; i <= n; i++) {
            isPrime = true;

            for (int j = 2; j < i; j++) {
                if (i % j == 0) {
                    isPrime = false;
                    break;
                }
            }

            if (isPrime) {
                ++numPrimes;
                if (printPrimes)
                    System.out.println(i);
            }
        }
        return numPrimes;
    }
```

Run the code and observe its behavior. What does the variable `numPrimes` compute? What does the argument `printPrimes` mean?

Implement the new method `findPrimesFaster` by copying code from the `findPrimes` method and modifying it to have the following features:

- Uses labeled `continue` instead of `break`.
- Does not require the `isPrime` variable.
- Only tries to divide by integers up to the square root of the number being tested.

Check the output of `findPrimesFaster` carefully. It should produce the same results as `findPrimes`. How would you do this check mechanically?

Arrays

Required Reading (from the Java Tutorial):

- Arrays

In this problem you will make an even faster prime finder based on the following observation: when testing whether an integer is prime, it is sufficient to try and divide by the list of prime numbers up to the square root of the number being tested. For example, if the method has already discovered the prime numbers 2, 3, 5, 7, 11, and 13, and we are testing whether the integer 17 is prime, it is sufficient to try and divide the integer 17 by the prime numbers 2 and 3.

Implement the method `findPrimesEvenFaster` using this observation. Use an array to store the prime numbers that it has already found.

Check your method's behavior carefully against the simpler versions of the prime finder. How much faster does it run?

Characters & Strings

Required Reading (from the Java Tutorial):

- Characters
- Strings
- Converting between numbers & strings
- Manipulating strings

In this exercise, you will write several methods in the class `Scrambler`. The documentation of the String class in the Java API will be very useful.

Write the method `reverse`, which takes a string and returns its characters in reverse order. For example, `reverse("abc")` should return `"cba"`. One way to do this is by getting each character out of the string with its `charAt()` method and concatenating it onto the result string.

Write the method `reverseWords`, which takes a string and reverses the order of the words in the string without reversing the words themselves. For the purpose of this problem, assume that the words in the string are separated

by single spaces. For example, `reverseWords("go west!")` should return `"west! go"`. The `split()` method of String is an easy way to pull the string apart into words.

Object Equality

Java actually has two ways to test whether two values are equal:

- the == operator, which you have already seen and used with primitive data types. For example, `2+3 == 5` is true.
- the `equals()` method, which can be used on objects like Strings. For example, `"2"+"3".equals("23")` is true.

Confusion can arise because these two ways of testing for equality don't do the same thing. For example, you'll find that even though it's true that `"2"+"3".equals("23")`, it may *not* be true that `"2"+"3" == "23"`. This is because `equals` compares the contents of the objects, i.e., the actual sequences of characters, whereas == compares the identity of the objects, i.e., the locations where they are stored in memory. Since the concatenation `"2"+"3"` may create a new string in memory, the == operator may consider it different from `"23"`.

(In fact, it's even trickier than this, because sometimes the Java compiler will figure out how to make "2"+"3" and "23" share the same storage location, so "2"+"3"=="23" will be true. But other times it won't be; for example, if `String x="2"` and `String y="3"`, then `x+y == "23"` is not true, at least not for the JDK 5.0 compiler we're using in this class. If you can explain to the staff why the first case is == but the second case isn't, you'll get a cookie.)

If you know Scheme, the distinction between == and `equals` is similar to `eq` vs. `equal`. Python makes a similar distinction between the operators == and `is`. But be careful -- == in Python compares object contents, whereas == in Java compares object identity.

We'll learn more about `==` vs. `equals` in future lectures. For now, it's enough to follow this simple rule:

> **For primitive data (numbers and characters), always use the == operator.** This is easy, because you actually *can't* use the `equals()` method on primitive types; you'll get an error message from Eclipse.
>
> **For objects (like Strings, URLs, etc.), use the `equals` method.** That ensures that you're comparing the contents of the objects, and ignoring where they happen to be stored.

Using this rule, **write the method `isWordPalindrome`**, which takes a string and tests whether it is a word-level palindrome (consisting of the same sequence of words backwards and forwards). For example, `"fall leaves when leaves fall"`) is a word palindrome, but not an ordinary palindrome.

Exceptions

You may have noticed after you implemented `isWordPalindrome()` that your program output now ends with an error message, which may look something like this:

```
Exception in thread "main" java.lang.NullPointerException
        at Scrambler.reverseWords(Scrambler.java:10)
```

```
        at Scrambler.isWordPalindrome(Scrambler.java:20)
        at Scrambler.main(Scrambler.java:69)
```

This message is an *exception*. Java uses *exceptions* to handle any abnormal condition, including errors, generated during the program's execution. The error message you see in the output includes a *stack trace* which shows the methods that called the code that generated the error. **Click on each of the links in the stack trace** to understand how the program reached the line that eventually caused the error.

We will learn more about exceptions in a later lecture, but here are some exceptions that you are likely to encounter when you're debugging your code.

- `NullPointerException` — Thrown when an application attempts to use `null` in a case where an object is required. Typically, you are trying to call a method on a variable which you have forgotten to initialize.
- `IndexOutOfBoundsException` — Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range (see also ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException). This happens when you try to access an element that is not there (for example, if you have an array with 10 elements, and you try to access the 12^{th}, you will get an ArrayIndexOutOfBoundException at runtime). Bear in mind that Java lists are numbered from 0, so a 10-element list has elements at indices 0 through 9 but not 10.
- `ClassNotFoundException` — Thrown when an application tries to load in a class but no definition for the class with the specified name could be found. Any number of things can cause this, but first check for spelling mistakes and mistakes on the input signatures and output types of your code.

Fix your code by making each function you've written return a reasonable result when it is passed `null` as an argument. You will find that you can't call `equals()` on a `null` value, which requires us (already) to amend our rule about equality testing:

> **For comparing primitive data types, always use the == operator.**
>
> **For comparing two objects, use the `equals` method.**
>
> **For comparing an object reference against `null`, use the == operator.**

Commit Your Solutions

This is the end of the lab. Commit your solutions to your personal Subversion repository, as described above under Checking In Changes to Subversion.