Java 101 For Capital One

Lab Exercises

Eclipse Neon Version

Version 1.0.5 December 1, 2016

DESCRIPTOR SYSTEMS

"We Bring You Up To Speed"

Java 101 For Capital One Labs

Lab 1: Java Hello, World

In this lab, you will write the *Hello, world* program in Java.

Objectives:

- To write and run a Java program
- To use the Java perspective in Eclipse

Part 1: Java Hello World

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: ____

Eclipse Installation directory: ____

If your computer is running Windows Vista, Windows 7 or later and has the standard lab setup, the directories will be:

```
Lab Installation directory:      c:\Users\username\javaclass
Eclipse Installation directory:  c:\Users\username\java\eclipse
```

_2. Start Eclipse running:

a. Use Windows Explorer or My Computer to navigate to the **{Eclipse Installation Directory}**.

If your computer is running Windows Vista, Windows 7 or later, the directory is:

```
c:\Users\username\java\eclipse
```

Note: Be sure to substitute your computer's actual user name for the "username" above.

b. Start Eclipse running by double-clicking on *eclipse.exe*.

c. When prompted for the *Workspace*, browse to **{Lab Installation Directory}/workspace**, then press OK.

If your computer is running Windows Vista, Windows 7 or later and has the standard lab setup, the directory is:

```
c:\Users\username\javaclass\workspace
```

Note: Be sure to substitute your computer's actual user name for the "username" above.

d. Close the Welcome window by clicking on the 'X' in its titlebar.

We also recommend running Eclipse as a maximized window.

_3. Eclipse supports the notion of *perspectives*, which are sets of window panes that let you work in a particular role. For this lab, you will work in the *Java* perspective, which contains window panes that let you create and edit Java programs.

Open the *Java perspective* by choosing **Window - Open Perspective - Java**.

_4. Your first job is to create a new *project* for your lab. Follow these steps:

a. Choose *File - New - Java Project*, Then press Next.

b. On the *Create a Java Project* page, enter **Lab01** for the *Project name*, then press Finish.

You should see the new project in the Package Explorer pane. Expand it and note that it's empty except for a reference the Java Runtime Environment (JRE) and a *src* folder.

_5. The basic building block of a Java program is the *class*. Now you will use a wizard to create a class for your program. Follow these steps:

a. To start the wizard, right-click on the *Lab01* entry in the Package Explorer and choose **New - Class**.

b. On the first wizard page, enter **lab01** for the *Package*. A package is an organizational unit for Java classes. By convention, Java package names start with a lower-case first character.

For the *Name*, enter **HelloWorld**. By convention, Java class names start with an upper-case first character.

Near the bottom of the page, put a checkmark in the box labeled *public static void main(String[] args)*. This will cause the wizard to create a *main* method for you which will act as the starting point of your program.

Then press Finish. Note that the new package name and class appear in the Package Explorer and that Eclipse opens the new class's source code into the Java editor.

_6. Using the course notes as a guide, complete the class so it prints "Hello, world" to the console.

_7. Press Ctrl+S or choose **File - Save** to save the class -- Eclipse compiles the Java code whenever you save it.

_8. To run your program, right-click on the class in the Package Explorer, then choose **Run As - Java Application**.

You should see the "Hello, world" text in the Console pane at the bottom of the Eclipse window.

Part 2: Compile Errors

In this part, you will introduce some errors so you will be able to fix "real" errors in later labs. When you get each error, note that Eclipse's *Problems* view shows you a (hopefully) informative diagnostic message.

Steps:

_1. Now let's introduce a compilation error. Follow these steps:

a. In the HelloWorld.java editing window, change the class name from *HelloWorld* to **Test**, then save the file. This will cause an error, since Java requires class names to match file names.

b. Note the many ways Eclipse lets you know there's an error:

- The editing window's title bar shows a red 'X' next to the file name
- In the Package Explorer, the project, package and class entries all display a red 'X'
- In the editing window, there's a red 'X' in the left gutter on the line that has the error
- In the editing window, there's a red squiggly underline beneath the offending code
- The *Problems* view at the bottom of the window shows an error message (you may need to click on the Problems tab to surface the Problems view instead of the Console). Note that you can hover

the mouse over the error message if it's cut off in the *Description* column. You can also resize the columns by dragging the column heading

c. In this small program, it's easy to see exactly where the error is, but it's not so easy in a larger program. The best way to navigate errors in a large program is to double-click the error message in the *Problems* view -- try it and note how Eclipse positions the editing cursor on the line containing the error.

d. To fix the error, change the class name back to **HelloWorld**, then save the file -- all of the red 'X' indicators should disappear.

Important: You can never run a program until you have fixed *all* of the errors!

e. Run the program. Right-click on the class in the Package Explorer, then choose **Run As - Java Application**.

You should see the "Hello, world" text in the Console as before. (Note how the Console pane automatically surfaces whenever a running program creates console output.)

_2. Introduce another error by changing *System.out.println* to **System.out.Println** (note the upper-case 'P'). Save the file. What error does this cause? Fix the error before continuing and run the program again to test it.

_3. Remove the last closing bracket ('}') in the file and then save. What error does this cause? Fix before continuing and run the program again to test it.

_4. Change the *main* method's name to **Main** (note the uppercase 'M') and save. Does that cause a compile error? Try running the program. Any problems? Fix, save and re-test.

_5. Remove the square brackets ('[ ]') from main(), save and retest. What does all of this tell you about the main() method?

Part 3: Reading JavaDoc

In this part, you will learn how to use the documentation for the Java class library, such as System.out.println. This documentation is HTML-based, and is known as *JavaDoc*.

Steps:

_1. Open the JavaDoc and set its home page as a favorite by following these steps:

a. Open a Web browser and navigate to:

```
http://docs.oracle.com/javase/8/docs/api
```

b. Use your browser's facilities to set this page as a favorite or a bookmark.

_2. Use the following steps to find the documentation for the *System.out.println* method you've already used:

a. The JavaDoc documentation is divided into panes; the left-most panes contain a list of *packages* and classes (Java classes are normally organized into groups referred to as packages).

The upper-left pane contains the package list, while the lower-left pane shows the classes in the selected package. When you first open the documentation, the *All Classes* link has been pre-selected in the package pane -- that means that the class pane shows all classes, regardless of package.

b. In the package pane, find the *java.lang* package and click on it -- the classes pane should update with a list of the interfaces and classes from that package. Scroll down to find the *System* class. The main pane then contains the documentation for the fields (variables) and methods in the System class.

c. In the main pane, scroll down to find the *out* field, then click on it. The documentation should scroll to a description of that field. What is the type of this field? Note that you can click on all underlined phrases to hyperlink to additional documentation.

d. Click on the *PrintStream* link (if you haven't already done so). In what package does the PrintStream class reside? Find the *println* method(s). Why are there so many? What do they do? Which one did you use in the previous part of lab?

e. Now notice that at the top of the main pane, there's an *Index* link. Try using it to find the *System* class. Then try using it to find the *println* method. Do you see how the index could be useful if you know the name of a method or field, but don't remember in which package and class it resides?

_3. To test your documentation-reading skills, answer the following questions:

1. In which package does the *JButton* class reside?
2. What are the arguments to the *elementAt* method in the *Vector* class?

Part 4: Working with Eclipse

In this part, you will learn more about the Eclipse development environment.

Steps:

_1. Eclipse stores programs in a *workspace* directory -- you specified the workspace when you started Eclipse. If you want to work with Java code that exists outside of the workspace directory, you can *import* it into a project within the workspace.

Let's start by importing a slightly more complicated program than your "Hello, World" application. Follow these steps:

a. Open Windows Explorer or My Computer and browse to **{Lab Installation Directory}**/starters/lab01.

If your computer has the standard lab setup on Windows Vista or later, the directory is:

```
c:\Users\username\javaclass\starters\lab01
```

b. Right-click on the *com* folder and choose *Copy* to copy the folder to the Windows clipboard.

c. Back in Eclipse, in the Package Explorer, right-click on the *Lab01/src* folder and choose *Paste*.

d. Expand the *Lab01/src/com.descriptor.javautils* folder -- you should see the imported *Java2UML* class. There should be no errors at this point. If you see errors, right-click on the imported file and delete it, then try importing again, following the instructions as closely as possible.

_2. Open Java2UML.java into the editor by double-clicking on it. This program accepts the name of a Java class, examines it, then prints the names of all of the methods in the class.

At this point in the course, we don't expect you to understand all of this Java syntax, but go ahead and examine it and see if makes any sense.

_3. Now let's experiment with Eclipse's font settings. Follow these steps:

a. From the Eclipse menu, choose **Window - Preferences** and then expand the *General - Appearance* category and highlight the *Colors and Fonts* entry. In the right-hand pane, expand the *Basic* entry.

b. Choose the *Text Font* entry and press the *Edit* button -- try *Lucida Console*, *Regular*, *12 point*, then press OK twice. Note how the editor's font changed.

c. If you don't like that font, go back and change it to something you like!

_4. Now let's experiment with the *Outline* view -- you should see it on the right side of the Eclipse window. If you don't see it, choose **Window - Show View - Outline**.

The Outline View shows your class's features, especially its methods. Click on a method name in the Outline and note how Eclipse scrolls the editing window so the method is visible. In this small program, that's not very useful, but as your programs grow, the Outline View becomes very handy.

_5. Now let's rearrange the views, just so you can see how it's done. Click on the title bar for the Outline View and drag the view over the *Package Explorer* pane on the left side of the window. When you drop it, note that Eclipse now shows the Outline View in that pane but that the Package Explorer is available by clicking on a tab at the bottom of the pane.

Click on the *Outline View* tab to surface it and then click and drag its titlebar downward until the mouse cursor changes into a down-arrow icon, then let go of the mouse. Note how Eclipse now draws the Outline View as a separate pane so you can see it and the Package Explorer at the same time.

Experiment dragging and dropping views to customize Eclipse the way you might like it. If you accidentally "lose" a view, you can get it back by choosing **Window - Show View**.

_6. Now let's try running the program. Right-click on the class in the Package Explorer and then choose **Run As - Java Application**.

Note the message in the Console. This program requires a *command-line argument* that specifies the name of a class. To supply the argument, follow these steps:

a. In the Package Explorer, right-click on the class and choose **Run As - Run Configurations** to display the *Run configurations* wizard.

b. Click the *Arguments* tab, and in the *Program arguments* field, enter **java.util.ArrayList** -- this is the class whose methods we will display.

c. Press **Apply** and **Run** -- note how the *Console* view shows the methods in the ArrayList class. You should verify this list by looking in the JavaDoc.

d. You can maximize any window pane by double-clicking on its title bar -- try it with the Console.

To restore the pane to its normal size, double-click on the title bar again.

Part 5: Debugging

In this part, you will experiment with Eclipse's Java debugger.

Steps:

_1. Eclipse defines the notion of *perspectives*, which are groups of views geared toward a particular task. So far, you have been working in the Java perspective, which provides views for editing and navigating code. Now you will use the *Debug perspective*, which provides facilities to diagnose problems and understand how your program executes.

To switch to the Debug perspective, choose **Window - Open Perspective - Debug**.

Note that you can quickly switch between perspectives by clicking on the icons arranged horizontally across the top right side of the Eclipse window.

_2. The key window panes in the Debug perspective are the *Debug view*, which is in the upper left, and the set of views in the upper-right pane organized as tabs: *Variables* and *Breakpoints*. Try clicking on the tabs to surface each view.

Note that the Debug view pane's titlebar contains icons for debugging commands, such as Resume, Step Over and so forth. Hover the mouse over the icons to see the command name (some may be grayed out).

_3. The Debug view keeps a record of the programs that you've previously executed. To clear the list, right-click on the Debug view pane and to display a popup menu and choose **Remove All Terminated**.

Note that there are grayed-out commands on the popup menu to let you terminate running programs -- this is handy if you ever have a program get into an infinite loop!

_4. Now let's set a breakpoint. Follow these steps:

a. In the Java2UML source window, find the line containing the first *if* statement.

b. In the left gutter for that line, double-click -- Eclipse draws a circle to indicate that this line contains a breakpoint.

Also note that the new breakpoint shows up in the Breakpoints view (this view is a tab in the upper-right window pane).

c. From the Eclipse main menu, choose **Run - Debug** and note that Eclipse selects the Run configuration you configured previously.

d. Eclipse executes the program until the breakpoint is reached. Note how the Java2UML source window has the breakpoint line highlighted and that there's an arrow in the gutter.

Also note that the Debug view has an entry for the application being debugged.

_5. Now that execution has stopped before executing the highlighted line, you can examine variables and perform other debugging tasks as necessary. To examine the contents of the command-line argument variable, follow these steps:

a. Click on the *Variables* tab in the upper-right window pane.

b. At this location in the program, there's only one variable defined -- the *args* command-line argument array (we will cover arrays in more detail later). Keep expanding the "+" symbols until you see the characters that make up the command-line argument string: 'j', 'a', and so forth.

_6. Next, let's try single stepping. Find the icon on the Debug view's titlebar that says *Step Over* and click it. Note how the highlighted line moves in the Java2UML source pane to show which line will be executed next. Also, note how the first single-step skipped the "if" statement body since the "if" expression is false.

Keep stepping over until the highlight is on the line:

```
Method method = methods[i];
```

Look in the Variables view -- how many variables are available at this point in the program? Note that you can collapse the "+" symbols on the *args* variable.

_7. The current line is inside of a *for* loop. Debugging loops by single stepping can be tedious, so let's try a different technique:

a. Set a breakpoint on the line:

```
sb.append("(");
```

b. On the Debug View titlebar, press the icon for *Resume* -- this causes the program to execute until the breakpoint.

c. Find the *sb* entry in the Variables tab and click on it to display its contents. What's the name of the first method that this program processed?

d. Press the Resume icon again. What's the name of the second method? (Note that it might the same as the first - Java allows a class to define multiple methods with the same name as long as the method's arguments are different.)

e. Repeat this process four or five times and write down the method names.

_8. Remove the second breakpoint either by double-clicking on the circle in the gutter or by using the Breakpoints view.

_9. Press the Resume button -- since there are no more breakpoints in the flow, the program will run to completion. Note how the entry in the Debug view indicates termination.

Examine the Console view, scrolling up as necessary to see the names of the methods that this program processed. Note that they will not be in the same order since the program sorted them alphabetically.

_10. Remove any remaining breakpoints.

_11. Set a breakpoint on the line:

```
Collections.sort(methodList);
```

_12. Restart the debugger by choosing **Run - Debug**.

_13. Use the Variables view and single-stepping to see the methodList *elementData* before and after sorting. Note how Eclipse displays changed variables in a different color.

_14. When you are finished, run the program to completion, remove all breakpoints, and switch back to the Java perspective.

Part 6: Reinforcement

In this part, you will repeat the procedures learned in earlier parts, this time with minimal instruction. By re-reading and repeating the steps, you will better remember how to do them in future labs.

Steps:

_1. Create a new Java project named **Lab01Reloaded**.

_2. In the new project, create a class named **PrintPI** in the **extra** package. The new class should have a *main* method.

_3. Use the JavaDoc to find information about the *Math* class.

_4. In the new class's main(), write Java code to display your name on one line, your title on the next, and the value of pi on the next.

Hint: To reference a *static final* field, you use syntax like classname.fieldname, for example, **Math.PI**.

_5. Compile and test your program.

Lab 2: Data Types and Assignment

In this lab, you will work with Java data types and the assignment operator.

Objectives:

- To learn Java primitive data types
- To understand the structure of a Java class

Part 1: Class Structure

Steps:

_1. Ask your instructor for the following information:

Lab Installation directory: ____

Eclipse Installation directory: ____

If your computer is running Windows Vista, Windows 7 or later and the standard lab setup, the directories will be:

```
Lab Installation directory:      c:\Users\username\javaclass
Eclipse Installation directory:  c:\Users\username\java\eclipse
```

_2. Start Eclipse running, if it's not already running.

_3. Use **File - Close All** to close all existing source windows, then create a new Java Project named **Lab02**.

_4. In the Package Explorer, right-click on the new project and choose **New - Package**. Name the package **part1**.

_5. Right-click on the *part1* package and create a new class named **Order** -- the class should have a *main* method.

_6. A class can define state (referred to as *fields* in Java) and behavior (methods). The wizard created a *main()* method -- in this step, you will define fields.

In the new class, above the *main* method, but within the block that defines the class, write the following fields:

```
private int orderNumber = 5;
private LocalDate orderDate;
private double listPrice = 13.05;
private double sellingPrice = 10.44;
private String notes = "X32 Flange Support";
```

We recommend that you indent the field definitions using the tab key to better show the class structure.

Save the file to compile it -- did you get an error?

_7. The error is due to the fact that the *LocalDate* class is defined in a different package and needs to be *imported*. The good news is that Eclipse can write the import statement for you.

From the Eclipse menu, choose **Source - Organize Imports**.

Note that Eclipse added an *import* statement near the top of the file. You should now be able to save with no errors.

_8. Now let's add a new method to the class. Above the *main* method, but after the field definitions, enter:

```
public String toString()
{
    return null;
}
```

This defines a method named *toString* that accepts no arguments, but returns a String. Currently, it returns the special *null* reference, but you will fix that in a moment.

_9. Fields are considered part of the class, and are global to all methods. Let's prove that. Change the *toString* method so it looks like:

```
public String toString()
{
    String s = "Order number: "   + orderNumber +
               " Order date: "    + orderDate +
               " List price: "    + listPrice +
               " Notes: "         + notes +
               " Selling price: " + sellingPrice;
    return s;
}
```

This code uses the string concatenation operator '+' to build a String object that contains labels and values for most of the class's fields. The *toString* method then returns that String.

_10. Update the *main* method so that it creates an Order object and calls *toString*:

```
public static void main(String[] args)
{
    Order o = new Order();
    String s = o.toString();
    System.out.println(s);
}
```

Note that the variable *s* is a local variable, which means that it's defined in a method, not as part of the class. And note that it's perfectly OK to use the same local variable names in more than one method.

_11. Run the program using the same technique you learned in the previous lab. What output do you see? Do the values for the fields make sense?

To see a better value for the date, change the field to:

```
private LocalDate orderDate = LocalDate.of(1994, 8, 27);
```

Save, then run again. Does it look better?

_12. Modify the lines in *main* so they look like:

```
Order o = new Order();
System.out.println(o);
```

Run the program again and note that it works the same. System.out.println() automatically calls toString() on objects like your Order object.

_13. In the *toString* method, above the String definition, add the line:

```
int orderNumber = 12;
```

This new local variable has the same name as a field. Try running the program. Which value displays? Note how the local variable takes precedence over the field.

Also note how Eclipse puts a yellow underline on the *private int orderNumber* field to indicate that it's unused.

_14. In *toString*, modify the String initialization for orderNumber to look like (leave the rest the same):

```
String s = "Order number: "   + this.orderNumber +
```

Run the program again. What order number displays? Note that now Eclipse draws the yellow underline on the local variable to indicate it's unused.

The *this* keyword explicitly tells Java to use a field. It's actually implicitly there on all field references, but you can omit it when there's no conflict with a local variable name. Try putting *this* in front of all of the field names:

```
String s = "Order number: "   + this.orderNumber +
           " Order date: "    + this.orderDate +
           " List price: "    + this.listPrice +
           " Notes: "         + this.notes +
           " Selling price: " + this.sellingPrice;
```

Part 2: Literal Values

In this part, you will learn about assigning literal values to variables of different types.

Steps:

_1. In the Lab02 project, create a new package named **part2** and within it, a class named **TestLiterals**. The class should have a *main* method.

_2. In the *main* method, define local variables according to the following table:

```
Variable name      Variable type
-------------      -------------

isManagement        boolean
middleInitial       char
teenYear            byte
lifeSpan            short
population          int
grainsOfSand        long
salary              float
nationalDebt        double
name                String
birthDate           LocalDate
```

_3. Write assignment statements to initialize each of the variables to a literal value of your choice.

_4. Use *System.out.println* to display each of the variables.

_5. Run your program and ensure that the results make sense.

Part 3: Type Conversions

In this part, you will experiment with type conversions.

Steps:

_1. Create a new class named **TestConversions** in the **part3** package. Make sure that the class has a *main* method.

_2. In *main*, enter the following:

```
byte firstByte;

firstByte = 20;
System.out.println(firstByte);
```

Run the program. Do you get the output you expected?

_3. Add the following lines:

```
firstByte = 255;
System.out.println(firstByte);
```

What happens when you save? The compiler realizes that 255 is too large to fit in a byte, so it gives you an error.

"Fix" the error using a cast:

```
firstByte = (byte)255;
System.out.println(firstByte);
```

Then run the program. Do you get the results you expected? Java treats *byte*, *short*, *int* and *long* types as *signed* -- the value 255 stored in 8-bit, twos-complement notation is -1.

_4. Change the above assignment to be *0xff* instead of *255* -- 0xff is hexadecimal notation for the same value.

_5. Add the following to *main*:

```
float f = 123;
System.out.println(f);
```

Save to compile. What is going on here? The "123" is actually a literal for an integer, and the compiler automatically promotes integers to floats.

Run the program and note that System.out.println formats floats to always have at least one digit following the decimal point.

Change the code so it looks like:

```
float f = 123f;
System.out.println(f);
```

Any difference when you run it? Now there's no type promotion since the types on both side of the assignment operator are the same.

_6. Change the above code so it looks like:

```
float f = 123.0;
System.out.println(f);
```

Did you get a compilation error? The issue is that Java interprets literal constants containing a decimal point as *double*, not *float* -- and since in general, doubles are larger than floats, the compiler generates an error.

You can fix the error by putting an 'f' character after the ".0" so the literal constant looks like *123.0f*. That forces the compiler to interpret the literal as a float instead of a double.

_7. Add the following to *main*:

```
char firstChar = "C";
System.out.println(firstChar);
```

When you save, you'll get an error -- "C" is literal for a String object, not a *char* primitive. To fix the error, change the double quotes to single quotes -- that defines a *char* typed literal.

_8. Add the following to *main*:

```
boolean firstBool = 'true';
System.out.println(firstBool);
```

When you save, you'll get an error because *'true'* is not a legal literal constant for a boolean. The only possible boolean literals are *true* and *false*: no quotes. Fix the error before continuing.

_9. Add the following to *main*:

```
char secondChar = firstChar + 2;
System.out.println(secondChar);
```

What's wrong here? The problem is that Java interprets the *2* as a literal *int* and promotes *firstChar* to *int*. Then the assignment operator complains because the right side is "larger" than the left side (int cannot in general fit in char).

How do we fix this? But we must ask: is this a good idea in the first place? It looks like we are trying to perform arithmetic on characters, which is generally a bad idea. You may have done such arithmetic in other languages that use ASCII characters, but Java uses Unicode characters, Unicode doesn't (in general) support character manipulation via addition and subtraction.

But if you really want to do this, you could write something like:

```
char secondChar = (char)(firstChar + 2);
```

This code uses parenthesis to guarantee that the addition happens first (using int) and then we cast the sum back to *char*.

Part 4: Primitive Wrapper Classes

In this part, you will experiment with converting primitives to Strings and vice versa.

Steps:

_1. Create a new class named **TestWrappers** in a package named **part4**. The class should have a *main* method.

_2. Java provides primitive types such as *int* and *double* so that you can perform arithmetic on values. However, in many common scenarios, such as receiving user input, you will obtain data as Strings. Before you can perform math, you will need to convert Strings to the appropriate primitive.

To convert from String to a primitive, Java provides several object classes in the *java.lang* package that have names that are similar to a primitive (remember that Java is case-sensitive and that class names usually start with an upper-case first character):

primitive type	"Wrapper" class
boolean	Boolean
byte	Byte
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

_3. Open the JavaDoc and look up the class named *Integer*. This class defines several useful methods -- we will examine a couple here. Please examine the documentation for the following:

```
static int parseInt(String s)
static String toString(int i)
```

The first method accepts a String and returns an *int*, while the second does the opposite. Both are defined as *static*, which means that we don't need to create an *Integer* object to call the method -- we will cover *static* in more detail later in the course.

_4. First, let's try converting an *int* to a String. Add the following code to *main*:

```
int firstInt = 42;
String firstString = firstInt;
System.out.println(firstString.length());
```

When you save, you will get a compile error because a String object reference is not compatible with an *int*.

Fix the code:

```
int firstInt = 42;
String firstString = Integer.toString(firstInt);
System.out.println(firstString);
```

As you can tell from the JavaDoc, the *Integer.toString* method accepts an *int* and returns a String object reference that contains the characters that correspond to the *int's* digits.

_5. Now let's go the other direction. Add the following to *main*:

```
String secondString = "987";
int secondInt = secondString;
System.out.println(secondInt * 2);
```

Again, you will get a compile error due to incompatible types. Fix the error:

```
String secondString = "987";
int secondInt = Integer.parseInt(secondString);
System.out.println(secondInt * 2);
```

The *Integer.parseInt* method accepts a String, hopefully formatted as an integer and returns the corresponding *int* value, which can participate in math operations.

_6. Add the following to *main*:

```
String thirdString = "hello";
int thirdInt = Integer.parseInt(thirdString);
System.out.println(thirdInt * 2);
```

What happens when you run the program? The issue is that "hello" is not formatted as a proper *int*, so *Integer.parseInt* throws a NumberFormatException. We will cover exceptions in more detail later in the class.

_7. Using the concepts you learned in this part, define a String with "43.76". Then use the *Double* class's *parseDouble* method to convert the String into a double. Then multiply the double times two and display the result.

Part 5: Working with References

Java supports two basic types of variables: variables that contain primitive values (e.g. *int*) and variables that contain references to objects. We saw in the last part how to convert String objects to/from primitives. In this part, you will learn more about how references work.

Steps:

_1. Create a new class named **TestReferences** in the **part5** package with a *main* method.

_2. Add the following to *main*:

```
int firstInt = 12;
int secondInt = 12;
boolean theSameInt = (firstInt == secondInt);
System.out.println ("Same integer: " + theSameInt);
```

This code creates two *int* variables and then uses the *equality operator* '==' to compare them. Run the program and note the output.

_3. Add the following to *main*:

```
Date firstDate = new Date();
```

Click on *Date*, then right-click and choose **Source - Add Import** to import the java.util.Date type.

This code creates a Date object that represents the current time and date.

_4. Add the following:

```
long time = firstDate.getTime();
Date secondDate = new Date(time);
```

This code retrieves a *long* that contains the number of milliseconds since January 1, 1970, 00:00:00 GMT that the first Date represents.

We then create another Date object using that *long* so that the two Data objects represent the exact same instant in time.

_5. Now let's compare the two Dates. Add this code:

```
boolean theSameDate = (firstDate == secondDate);
System.out.println ("Same date: " + theSameDate);
```

The *equality operator* returns a boolean that compares the references. Run the program and note the output. Does it make sense?

_6. The issue is that the equality operator works a bit differently than you might expect for references. References, as their name implies, *refer*, or point to objects. When you you compare references, you're comparing the reference values, not the objects to which they reference. And since we have two distinct objects on the heap, their reference values are different, even though the objects contain the same data.

_7. Modify the code so it looks like:

```
boolean theSameDate = firstDate.equals(secondDate);
```

Then run the program again. Now you are asking the first Date object to compare its value against the second object. Do the results make sense?

Part 6: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts. We will supply you with a Java class that contains many syntax and conversion errors. Your job is to get a clean compile and run.

Steps:

_1. To import the error-ridden class, follow these steps:

a. Open Windows Explorer or My Computer and browse to **{Lab Installation Directory}**/starters/lab02.

If your computer has the standard lab setup on Windows Vista or later, the directory is:

```
c:\Users\username\javaclass\starters\lab02
```

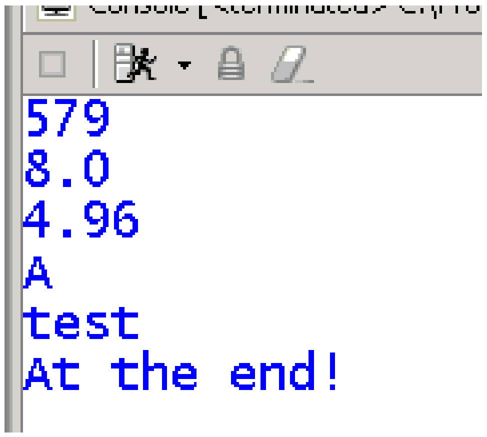
b. Right-click on the *part6* folder and choose *Copy* to copy the folder to the Windows clipboard.

c. Back in Eclipse, in the Package Explorer, right-click on the *Lab02/src* folder and choose *Paste*.

Press Finish to import the class. You will note that it has several compilation errors.

_2. After fixing all of the compilation errors, try to run the program and fix and issues that come up when you attempt to run.

When you are successful, the output should look something like:



```
579
8.0
4.96
A
test
At the end!
```

Figure 2

Lab 3: Operators

In this lab, you will work with Java operators.

Objectives:

- To learn Java operators

Part 1: Getting Started

Steps:

_1. Start Eclipse if it's not running.

_2. Use **File - Close All** to close all existing source windows, then create a new Java Project named **Lab03**.

_3. In the Package Explorer, right-click on the new project and choose **New - Package**. Name the package **part1**.

_4. Right-click on the *part1* package and create a new class named **CalcCircleArea** -- the class should have a *main* method.

_5. The CalcCircleArea program should calculate the area of a circle using the formula:

```
area = Math.PI * radius * radius
```

To get started, in *main*, define a variable to hold the radius and initialize it:

```
double radius = 32.5;
```

_6. Using the above equation, initialize a new *double* variable, **area** to calculate the area given the *radius* variable:

```
double area = . . .
```

_7. Print the area to the console:

```
System.out.println("Area: " + area);
```

_8. To run the program, in the Package Explorer, right-click on the *CalcCircleArea.java* entry and choose *Run As - Java Application*. The result should be **3318.307240354219**.

Part 2: Math Operators

Steps:

_1. In the Package Explorer, right-click on the *Lab03* project and choose **New - Package**. Name the package **part2**.

_2. Right-click on the *part2* package and create a new class named **TestQuadratic** -- the class should have a *main* method.

_3. In this part, you will write two methods that solve the *quadratic equation* (a.k.a. the quadratic formula). As you may recall from school, the quadratic equation solves equations of the form:

$$ax^2 + bx + c = 0$$

Figure 1

Using this formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Figure 2

_4. The quadratic formula returns two results (note the +-), so you will write two methods. Add the following two methods above *main*, but inside of the class:

```
public static double solveQuadraticPlus (double a, double b, double c)
{
    return 0;
}

public static double solveQuadraticMinus (double a, double b, double c)
{
    return 0;
}
```

Note how these methods accept *parameters* for 'a', 'b' and 'c' - you will call the methods from main(), supplying the values for the parameters. In other words, each of these methods can assume the parameter values will be supplied by the caller.

_5. Using math operators and methods from the *Math* class, complete the *solveQuadraticPlus* method so it returns the result using the '+' case.

Note: Java defines the - operator that flips the sign of a numeric variable (e.g. a *double*).

Hint: You can implement the equation using multiple lines of code. For example, you can calculate the value of the expression inside of the square-root radical using code like (put this code in *solveQuadraticPlus*):

```
double inner = b * b - 4 * a * c;
```

Then you could calculate the square root:

```
double radical = Math.sqrt(inner);
```

Use similar steps to complete the equation.

Another hint: A method sends a result back to its caller using the *return* keyword. Each of your methods should return a *double* value back to the caller:

```
return xxxx;      // xxxx is the name of your 'result' variable
```

_6. Complete the *solveQuadraticMinus* method so it returns the result using the '-' case.

_7. In *main*, call the two methods and print the results. for example:

```
double answerPlus = solveQuadraticPlus(1, 3, -4);
System.out.println (answerPlus);
double answerMinus = solveQuadraticMinus(1, 3, -4);
System.out.println (answerMinus);
```

For testing, if you use *a=1*, *b=3*, *c=-4* (note the minus sign), the answers should be 1.0 and -4.0.

Part 3: More on Math Operators

In this part, you will learn more about math operators and related topics.

Steps:

_1. Create a new class named **MoreMath** in the **part3** package. The new class should have a *main* method.

_2. In *main*, write the following code that loops 100 times (we will cover loops in detail later):

```
for (int i=0; i<100; i = i + 1)
{
}
```

This loop uses the variable named *i* to count from zero to 99.

_3. Inside of the loop, use an *if* statement so that the program prints only values of *i* that are multiples of seven. **Hint:** use the *remainder* '%' and *equality* '==' operators.

Run and test.

_4. Above *main*, but within the class, write a new method that calculates Ohm's Law:

```
public static int calcCurrent (int volts, int resistance)
{
    return volts/resistance;
}
```

Note that for now, we use *int* for the types.

_5. In *main*, after the loop, call the *calcCurrent* method, passing *volts=5* and *resistance=0*.

Run the program. What happens and why?

_6. Convert the *calcCurrent* method to use *doubles* instead of *int*.

Run again. What does this tell you about how Java treats floating-point and integer arithmetic?

Part 4: More Operators

In this part, you will practice with compound assignment operators.

Steps:

_1. Create a new class named **ScorePrediction** in the **part4** package. The new class should have a *main* method.

_2. In the new class, above *main*, define a method:

```
public static int calcBonus ( int predictedTeam1,
                              int predictedTeam2,
                              int actualTeam1,
                              int actualTeam2 )
{
    return 0;
}
```

Your program will implement part of a scoring system for a sports "pool" in which participants predict the output of a game between two teams. The *calcBonus* method accepts a participant's predicted scores for each team and the actual results of the game.

_3. Your job is to complete this method so it returns a "bonus" value based on how accurate the prediction was. Here are the rules for calculating the bonus:

- If the participant predicted the actual score for either team, award 3 points for each correctly predicted score (6 points maximum)
- If the participant correctly predicted the absolute value of the margin between the winning and losing teams, award 5 points
- If the participant predicted the correct winning team, award 10 points

Here is a snippet of code to help you get started (it implements part of the first bullet point):

```
public static int calcBonus ( int predictedTeam1,
                              int predictedTeam2,
                              int actualTeam1,
                              int actualTeam2 )
{
    int score = 0;

    if (predictedTeam1 == actualTeam1)
        score += 3;

    return score;
}
```

Here are some hints:

- There can be no tie games
- The *Math* class contains a method that calculates absolute value
- You can construct *boolean* expressions to help you work with determining the winning team:

```
boolean isTeam1PredictedWinner =
      predictedTeam1 > predictedTeam2;
```

_4. In *main*, call the method and print the result to the console. Here are four test cases you can try to verify correct operation:

Team1 Prediction	Team2 Prediction	Team1 Actual	Team2 Actual	Bonus
28	21	28	21	21
28	21	21	28	5
28	21	1	0	10
28	21	0	1	0

Your code to call the method for the first test case might look like:

```
double bonus = calcBonus(28, 21, 28, 21);
if (bonus == 21)
{
    System.out.println("Passed Case 1");
}
else
{
    System.out.println("Case 1 failed, calculated bonus: " + bonus);
}
```

Part 5: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts.

Steps:

_1. Create a new class named **CompoundInterest** in the **part5** package. The class should have a *main* method.

_2. In this class, you will write a method that calculates the future value of an investment that accumulates compounded interest. The formula is:

$$P = C\,(1 + r/n)^{nt}$$

Figure 3

```
P = Future value
C = Initial deposit
r = Interest rate expressed as a fraction (e.g. 0.05)
n = Number of times interest is compounded each year
t = Number of years
```

_3. Write a *static* method named *calcCompoundInterest* that accepts the four input arguments as *doubles* and returns a *double* result containing the future value.

_4. Call the method from *main* and print the result to the console.

Here is an example you can use for testing:

Paul invested $10,000 at an interest rate of 7.5% compounded quarterly. How much money would he have after five years?

The correct answer is $14,499.48 (rounded to two decimal places).

Lab 4: Flow Control

In this lab, you will work with Java flow-control constructs such as *if* statements and *while* loops

Objectives:

- To learn Java decision-making statements
- To learn Java flow-control statements

Part 1: Getting Started with If

Steps:

_1. Start Eclipse if it's not running.

_2. Use **File - Close All** to close all existing source windows, then create a new Java Project named **Lab04**.

_3. Right-click on the new project and choose *New - Class* to create a class named **SimpleIf** in the **part1** package. The new class should have a *main* method.

_4. The goal of this program is to retrieve a random number that we will interpret as an employee's salary. Then, based on the salary, we will classify the employee. Follow these steps:

   a. Inside *main*, create a random-number generator object:

   ```
   Random rand = new Random();
   ```

   Use *Source - Organize Imports* to import the Random type from the *java.util* package.

   b. Retrieve a random number that's less than 200,000 for the employee's salary, then print it out:

   ```
   int salary = rand.nextInt(200000);
   System.out.println("Salary: " + salary);
   ```

   c. Write an *if-else if-else* to classify the employee. If the salary is greater than or equal to 100,000, the employee is "A highly paid executive". If the salary is greater than or equal to 50,000 (but less than 100,000), the employee is "A middle manager". If the salary is less than 50,000, the employee is "An entry level employee".

_5. Run the program by right-clicking on SimpleIf.java in the Package Explorer and choosing *Run As - Java Application*. What kind of employee did you get? Run it a few times to see if the logic works for different random salaries.

Part 2: Making Decisions

In this part, you will write a program that makes decisions based on user input. You can use Java's *if-else if -else* statement.

Steps:

_1. Create a new class named **DecisionsDecisions** in a new package named **part2**. The new class should have a *main* method.

_2. The goal of this program is to:

1. Prompt the user to enter a line of text
2. Retrieve the input string
3. If the string is "exit", print "Goodbye"
4. If the string is your name, print "Hello, " and your name
5. If the string is anything else, print "Who are you?"

The next few steps give you some hints.

_3. To prompt the user, you can use *System.out.print*, which is similar to *System.out.println*, but doesn't print a new line:

```
System.out.print("Enter your name or 'exit': ");
```

_4. To retrieve the input line, you can use code like:

```
Scanner scanner = new Scanner(System.in);
String line = scanner.nextLine();
```

You can use **Source - Organize Imports** so that Eclipse writes an *import* statement for the java.util.Scanner class.

_5. To compare the characters in a String with a literal, you can use the *equals* method. Here's an example:

```
boolean exit = line.equals("exit");
boolean name = line.equals("Sam");
```

You can use the return from the *equals* method in an *if - else if - else* statement:

```
if (exit == true)
{
. . .
}
else if (name == true)
{
. . .
}
else
{

}
```

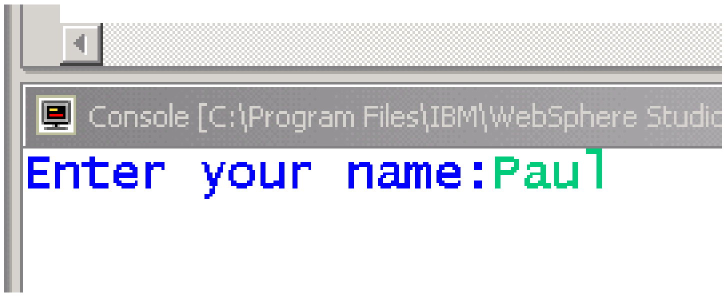
Note that the comparison is case sensitive.

_6. After the *else* closing brace, close the scanner:

```
scanner.close();
```

_7. Run your program as a Java application.

To test it, click the mouse in the *Console* view immediately following your input prompt. You then should be able to enter the input string and press Enter:



```
Enter your name:Paul
```

Figure 1

Part 3: Looping

In this part, you will work with Java loops.

Steps:

_1. Create a new class named **MeasureDistribution** in a new **part3** package. The new class should have a *main* method.

_2. The goal of this program is to:

1. Create a pseudo random-number generator object
2. Initialize three *int* variables : one to hold a count of positive numbers, one to hold a count of negative numbers and one to hold a count of zeros
3. Define a block that loops 100,000 times
4. Each time through the loop, generate a random integer. Based on the integer's value, increment one of the counter variables
5. After exiting the loop's block, print the counter variables

The next few steps give some hints.

_3. Look in the course notes and the Javadoc for how to create and use the *java.util.Random* class

_4. You can use a *for* loop to perform the looping. Each time through the loop, retrieve a random integer. Use the *nextInt()* method with no arguments.

_5. You can use an *if - else if - else* statement to determine whether the random integer is positive, negative or zero.

_6. Run your program and examine the results. Note that you may not get exactly the same number of negative and positive numbers -- that's normal for a pseudo random-number generator. Also note that the chances of getting any zeros are slim, so don't be surprised if you don't see any, even if you run the program several times.

Part 4: The Switch-Case Statement

In this part, you will get a chance to work with Java's *switch-case* statement.

Steps:

_1. Create a new class named **SysInfo** in the **part4** package. The new class should have a *main* method.

_2. The goal of this program is to:

1. Set up an infinite loop
2. Inside the loop, display a '>' prompt character and then retrieve a single keystroke from the user
3. If the key is 'q' or 'Q', break out of the loop and exit the program
4. If the key is 'u' or 'U', display the current user's name
5. If the key is 'o' or 'O', display the current operating system's name
6. If the key is 'h' or 'H', display the Java installation directory

The next few steps give you some hints.

_3. Create a java.util.Scanner connected to System.in as you did in an earlier part of lab.

See the course notes for an example of an infinite *while* loop.

_4. To display the prompt character, you can use *System.out.print*, which is similar to *System.out.println*, but doesn't print a new line.

To retrieve a single keystroke from the user, you can use the Scanner:

```
char c = scanner.nextLine().charAt(0);
```

_5. Use a *switch-case* statement to make a decision based on the input character.

_6. To retrieve the Java home directory and the OS and User names, you can call the *System.getProperty* method:

```
String s = System.getProperty("java.home");
```

Use the JavaDoc to read about the *System* class for details. Hint: search the System class's JavaDoc Web page for "java.home".

_7. To exit the program, you can use the *System.exit(0)* method. See the course notes for an example.

_8. Run and test your program.

Part 5: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts.

Steps:

_1. Create a new class named **CountPunctuation** in the **part5** package. The class should have a *main* method.

_2. The goal of this lab is to write a program that analyzes lines of text entered by the user. Here are the specifications:

- The program should run in an infinite loop, prompting the user, then reading a line of text each time through the loop
- If the user enters "exit" or "EXIT", the program should terminate

- The program should loop through the characters in the input line, counting space characters, punctuation and "normal" characters. Punctuation is defined as a comma, colon, semicolon, or a period.
- After analyzing the string, the program should display the count of spaces, punctuation characters and normal characters

The next few steps have some hints.

_3. You will need to nest a loop within a loop -- the outer loop will be infinite and the inner will loop through the characters in the inputted String.

_4. You can retrieve individual characters from a String by calling the *charAt* method:

```
int i = 0;
char c = line.charAt(i);
```

If you do this in a loop, varying the index each time, you can retrieve all of the characters in the string.

_5. You can retrieve the number of characters in a String by calling the *length* method -- you can use this length to terminate a loop.

_6. You can use a *switch-case* statement to determine which category a given character falls into.

Lab 5: Java and Object Orientation

In this lab, you will work with object-oriented techniques in Java.

Objectives:

- To write Java classes and create objects
- To get started with inheritance

Part 1: Writing a Class with State and Behavior

Steps:

_1. Use **File - Close All** to close all existing source windows, then create a new Java Project named **Lab05**.

_2. Create a new class named **SavingsAccount** in the **bank** package. The class SHOULD NOT have a *main* method -- you will write a separate testing class in a later step.

_3. This class represents a savings account for a bank's IT system.

Define the following fields as private members of the class:

```
Type            Name                Description
-----           ----                -----------

int             accountNumber       Unique ID for this account
LocalDate       createDate          The date the account was created
double          balance             Current account balance
```

_4. Now let's add some behaviors to the class. Write the following methods within the class's block:

```
public void withdraw(double amount)
{
}

public void deposit(double amount)
{
}

public String toString()
{
   return "Account number: " + accountNumber +
          " Creation date: " + createDate +
          " Account balance: " + balance;
}
```

Add code to the *withdraw* and *deposit* methods so that they correctly update the balance using the amount parameter.

_5. Create a new class named **TestSavingsAccount** in a new package named **test**. This new class SHOULD have a *main* method.

_6. In *main*, create a *SavingsAccount* object named **acct**. Call *deposit* and *withdraw* on the object, passing values of your choice. Then display the object to the console:

```
System.out.println(acct);
```

Note: You will need to import the SavingsAccount class since it resides in a different package.

_7. Run your program -- are you satisfied with the balance? Don't worry if you don't see reasonable values for the creation date and account number. You will fix those later.

Add the following line to *main*:

```
acct.accountNumber = 12;
```

Save to compile. Why doesn't this work? You will get around this issue in the next part, but for now, comment-out the offending line.

Part 2: Get/Set Methods

In the last part, there was no way to change the account number and creation date from outside of the class since the fields were *private*.

In this part, you will provide controlled access so the outside world can examine or modify a SavingsAccount object's state.

Steps:

_1. The standard Java technique for exposing state is to write a pair of methods variously referred to as *get/set* methods or *accessor/mutator* methods. Writing such methods is trivial but tedious -- the good news is that Eclipse will write them for you! Follow this procedure:

a. Open the source editor for *SavingsAccount.java*

b. Right-click on the source window and choose **Source - Generate Getter and Setter**.

c. In the wizard, select the *accountNumber* and *createDate* entries so that getters and setters are generated for both, then press OK.

Note: We do not provide get/set methods for *balance* since that field is accessed via the *deposit* and *withdraw* methods.

d. Examine the generated methods and make sure that you understand them. Feel free to move them to a different location in the class if you don't like where the wizard inserted them.

_2. Edit *TestSavingsAccount*, and in *main*, immediately after creating the SavingsAccount object, call the *set* methods to initialize it to values of your choice. For the date, you can use *LocalDate.of(year, month, dayOfMonth)*, for example:

```
LocalDate.of(2012, 2, 18)
```

Then try running your program as before. Do you see correct output?

Part 3: Introduction to Inheritance

In this part, you will learn how any Java class can override behavior from the superclass of all Java classes: *Object*.

Steps:

_1. In the first part of this lab, you created a class named *SavingsAccount*, and you did not specify its superclass. In Java, whenever you don't explicitly specify a superclass, Java uses the predefined *java.lang.Object* class as the superclass. Thus, the *Object* class provides the basic behavior for all Java objects.

Use the JavaDoc to scan the documentation of the Object class. There's much in there that will be difficult to understand until you learn more Java, but you can get the general idea of what Object is all about.

In particular, note that it defines a method named *toString* -- we will work with that method later in this part of lab.

_2. Edit the *SavingsAccount* class, and comment out the existing *toString* method.

_3. Try running the TestSavingsAccount class -- what output did this produce?

When you pass an object reference to System.out.println, it calls the *toString* method to obtain a String to send to the console. At this point, there is no *toString* method in SavingsAccount, so Java uses the implementation inherited from *Object*, which returns the object's class name and its *hashcode*. While this is interesting, the output doesn't look very pretty or informative.

_4. The trick is to *override* the toString method in the SavingsAccount class so that Java uses your implementation rather than the implementation from Object.

Remove the comments from the *toString* method and run TestSavingsAccount again. Do the results make sense?

Part 4: Optional: Using BigDecimal

In the previous part, you used *double* to store the account balance. While this is simple, using floating-point for monetary transactions can lead to round-off errors. It's better to use the java.math.BigDecimal class instead. For a good discussion, see:

```
www.opentaps.org/docs/index.php/
   How_to_Use_Java_BigDecimal:_A_Tutorial
```

Note: Since this part is an advanced optional exercise, we do not give you step-by-step instructions. You will need to read about BigDecimal and then use your creativity and programming skills to complete the exercise.

Steps:

_1. Look in the JavaDocs for the BigDecimal type, then convert the SavingsAccount class to use it instead of *double* as the type for the *balance* field. We recommend that you keep the signatures for the get/set and withdraw/deposit methods the same as before (using *double*) - modify the code inside them to reflect the change from *double* to BigDecimal.

Hints:

- You can initialize a BigDecimal object by providing a value:

```
BigDecimal myvar = new BigDecimal(123.45 + "");
```

We recommend you pass a String rather than a double or float when creating the BigDecimal object to get an exact value.

- BigDecimal objects are *immutable* - that is, you cannot modify their value after creation. But the *add* and *subtract* methods return a modified BigDecimal:

```
myvar = myvar.subtract(new BigDecimal(100.20 + ""));
```

Part 5: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts.

Steps:

_1. Create a new class named **AccountList** in the **bank** package. This class should not have a *main* method.

_2. This class will keep track of SavingsAccount objects. The proper way to do this would be to use an array or a collection, but we haven't covered these yet, so this class will just maintain a couple of SavingsAccount references as fields.

Define the following fields:

```
Type                Name              Description
----                ----              -----------

SavingsAccount      acct1           First account
SavingsAccount      acct2           Second account
```

_3. Write two methods, one named *setAccount1* and the other named *setAccount2* that accept a SavingsAccount reference and assign it to one of the SavingsAccount fields. Both methods should return *void*.

_4. Write a method named *getAverageBalance* that accepts no arguments and returns a *BigDecimal*.

Implement the method by returning the average balance score of the two SavingsAccount objects.

_5. Override *toString* and return a String containing a labeled value for the average balance.

_6. Create a new class named **TestAccountList** in the *test* package. This class should define a *main* method.

_7. Add code to the *main* method so that it:

- Creates an AccountList object
- Creates two SavingsAccount objects and initializes their fields. You can copy some of this from the *TestSavingsAccount* class you write earlier
- Calls the *addAccount1* and *addAccount2* methods on the AccountList to set the SavingsAccount objects
- Uses System.out.println to print the SavingsAccount object to the console (this will invoke the EntryList toString method)

_8. Run and test your program.

Lab 6: Methods

In this lab, you will work with Java methods.

Objectives:

- To work with constructor methods
- To work with static methods and static fields

Part 1: Constructor Methods

Steps:

_1. In this lab, you will work with a *PoolEntry* class that represents an entry in an office "sports pool". (You may recall writing a *calcBonus* method in an earlier lab -- you will make use of that method in this lab, too.) To save time, we will provide you with some partially completed code that you will import into a project. Follow these steps:

  a. Create a new project named **Lab06**.

  b. Open Windows Explorer or My Computer and browse to **{Lab Installation Directory}**/starters/lab06.

  If your computer has the standard lab setup on Windows Vista or later, the directory is:

```
c:\Users\username\javaclass\starters\lab06
```

  c. Right-click on the *pool* and *test* folders and choose *Copy* to copy the folders to the Windows clipboard.

  d. Back in Eclipse, in the Package Explorer, right-click on the *Lab06/src* folder and choose *Paste*.

_2. Examine the imported pool.PoolEntry class and note how it represents a an entry in a sports pool. Then examine the imported test.TestPoolEntry class and note how it creates a few PoolEntry objects and initializes its fields using "set" methods.

_3. Edit the PoolEntry class and write a public constructor method that accepts parameters that match all of the fields in the class. Remember that constructor methods have the same name as the class and have no return value.

Implement the method by assigning each parameter to the corresponding field. Remember that you can use the *this* reference, for example:

```
public PoolEntry ( int entrantID,
                   LocalDate entryDate,
                   int predictedTeam1,
                   int predictedTeam2,
                   int actualTeam1,
                   int actualTeam2)
{
    this.entrantID = entrantID;
    . . .
}
```

_4. Save PoolEntry.java and note that you now have errors in TestPoolEntry and TestPoolEntryList!

That's because TestPoolEntry is using PoolEntry's built-in, no-argument constructor -- remember that Java gives you that constructor if you don't define ANY constructors. But once you define any constructor, Java "takes" the built-in constructor away.

_5. There are a couple of ways we could fix this. One way would be to recode the "test" classes to use the new, 6-argument constructor. However, let's take a different tack.

Edit PoolEntry and write a new constructor that accepts zero-arguments. This constructor should assign a random number for the entrantID:

```
public PoolEntry()
{
    java.util.Random r = new java.util.Random();
    this.entrantID = r.nextInt();
}
```

After you save, the compilation errors should disappear.

_6. Modify the TestPoolEntry class so that the first object instantiation (p1) uses the 6-argument constructor. You can use any *entrantID* you want, and use *LocalDate.now()* for the *entrantDate*.

Note that since you supplied all of that object's state in the constructor, the *set* methods are redundant for the *p1* object. You can comment them out.

Save, run and test that the program works. Note that you might need to scroll up in the Console to see the output from the first object.

_7. You may have noticed that the objects created with the zero-argument constructor have a *null* entry date. That's because we didn't initialize the field, and Java assigns *null* to object references by default.

Modify the zero-argument constructor to assign a default date, then run and test.

_8. Write a new constructor that accepts 4 arguments -- this constructor should omit the *entryDate* and *entrantID* fields.

To implement the new constructor, call the zero-argument constructor and then initialize the remaining fields:

```
public PoolEntry (
                   int predictedTeam1,
                   int predictedTeam2,
                   int actualTeam1,
                   int actualTeam2)
{
    this();
    this.predictedTeam1 = predictedTeam1;
    this.predictedTeam2 = predictedTeam2;
    this.actualTeam1 = actualTeam1;
    this.actualTeam2 = actualTeam2;
}
```

Then modify TestPoolEntry so that the second object (p2) instantiation uses the 4-argument constructor. Again, you can comment out redundant *set* methods. Save, run and test.

_9. Write another 4-argument constructor that omits the *entrantDate* and the *actualTeam2* fields, but has the other four fields (you can copy and paste and modify the 6-argument constructor). Don't implement it yet -- save the file after you write the constructor method signature.

The reason this cannot compile is that you would have two constructors that have the exact same number and types of parameters, and that's illegal because it is ambiguous. (Remember that the compiler doesn't really care what NAMES you assign to parameters -- it's their type and order that determines the method signature.)

Comment out this 4-argument constructor to get rid of the compile errors.

Part 2: Static Fields

Static fields are considered to be part of the runtime existence of a class, not as part of individual objects. Even though you might create many objects, there's only one "class object" at runtime -- this enables us to use static fields to share data among all of the objects of a class.

In the last part, since you defined the random number generator object reference as a local variable within the constructor, each object created its own generator object, used it to retrieve a random number, and then allowed the generator object to be garbage collected. The program would be more efficient if the program created a single generator object and shared it amongst the instances.

To share the object, we will define the generator object reference as a *static* field.

Steps:

_1. Modify the PoolEntry class so that the Random object is defined as a static field instead of being a local variable in the zero-argument constructor:

```
private static java.util.Random r = new java.util.Random();
```

This initialization runs before the first PoolEntry object is created.

The zero-argument constructor should still call the *nextInt* method and assign it to the *entrantID*, but be sure to delete the *new* within the constructor or comment it out.

_2. Save, run and test -- the program should work as before. The new version is more efficient, but the change is probably not noticeable to the human eye.

Part 3: Static Methods

Like static fields, static methods are considered part of the class, not part of any individual object. Often, we write static methods to manipulate static fields.

One side-effect of a method being static is that you don't need to first create an object before calling it -- in an earlier lab, you used the Integer.parseInt() method, which is a good demonstration of a static method.

In this part, you will experiment with static methods.

Steps:

_1. Create a new class named **MyStatic** in the **mystatic** package. This class should not have *main* method.

_2. Define the following fields:

```
private int myInt = 42;
private static int myStaticInt = 43;
```

_3. Write the following methods:

```
public int getMyInt()
{
    return myInt;
}

public static int getMyStaticInt()
{
    return myStaticInt;
}
```

_4. Create a new class named **TestStatic** in the *test* package. The new class should have a *main* method.

_5. In *main*, write code to call the static method and display its result to the console:

```
System.out.println(MyStatic.getMyStaticInt());
```

Note that you don't need to create an object! (You will need to import the MyStatic class.)

_6. Write code in *main*, in the normal fashion, to create a MyStatic object, call the non-static method and print the result:

```
MyStatic mys = new MyStatic();
System.out.println(mys.getMyInt());
```

Note the difference -- to call a static method, you don't need to first create an object. Instead, you use the class name before the dot.

_7. Edit the MyStatic class. In the static method, add this line:

```
System.out.println(myInt);
```

Save the file to compile it.

This results in a compilation error because static methods are NOT part of any individual object, and thus cannot access instance fields. This becomes very clear if you think about the code you wrote in TestMyStatic -- you were able to call the static method *before* ever creating an object! In other words, there were no instance fields in memory at that point in time.

_8. Comment out the offending line before continuing.

_9. Add the following line to the non-static method:

```
 System.out.println(myStaticInt);
```

Save the file to compile it. This is OK since static fields are considered to be global to all instances. In other words, any instance can access the static fields, since the "class object" is in memory before any instances are created.

Part 4: Implementing Singleton

One useful application of a static methods and fields is to implement a class of which there can be but a single instance. This *design pattern*, known as *Singleton* is what you will implement in this part. You will convert the *EntryList* class to implement the Singleton pattern using a static field and method.

Steps:

_1. Edit the EntryList class and write a **private** zero-argument constructor. You don't need to put any code in the constructor body for now.

Save the file and note that there is now an error in the TestPoolEntry class, which tries to use the zero-argument constructor.

In fact, there is now no way to create an EntryList object from outside of the class, since the only constructor is private! We will address this issue in a moment.

_2. Add a private, static field to the EntryList class:

```
private static EntryList theInstance;
```

_3. Add a public, static method that manipulates this field:

```
public static EntryList getInstance()
{
    if (theInstance == null)
        theInstance = new EntryList();

    return theInstance;
}
```

This publicly available static method returns the single instance of the class and is the *only* way for the outside world to obtain that instance. Note that it uses a technique referred to as *lazy initialization* -- it only instantiates that object the first time it's asked for.

_4. Modify the TestEntryList class to use the static *getInstance* method.

Save, run and test. Even if we had multiple testing programs running, they would all use the same EntryList object.

Part 5: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts.

Steps:

_1. Create a new class named **AutoPolicy** in the **policy** package. This class should not have a *main* method.

_2. Add the following fields to the class to represent information about an automobile insurance policy. Use types as you feel are appropriate:

```
policyNumber
policyDate
amount
customerID
```

_3. Override *toString* and return an appropriately labeled String.

_4. Write three public constructors in the class:

- A zero-argument constructor that initializes the date to the current date and time
- A three argument constructor that accepts all of the fields except for the date. This constructor should call the zero-argument constructor
- A four-argument constructor that accepts values for all fields

_5. Create a new class named **TestPolicy** in the **test** package with a *main* method.

Add code in *main* to create three AutoPolicy objects, each with a different constructor. Print the contents of each object to the console.

_6. Save, run and test.

Lab 7: Exceptions

In this lab, you will write a Java application that lets the user enter checks into a personal finance system. Your application will store the data in a simple database and will verify that the account has sufficient balance before posting the transaction.

You will handle two types of exceptions: SQL exceptions from the database and application-defined exceptions.

Objectives:

- To write code that catches exceptions
- To write code that throws exceptions

Part 1: Getting Started with Exceptions

Steps:

_1. Create a new Java project named **Lab07**.

_2. Right-click on the Lab07 project and choose *New - Class* to create a class named **ConvertTemperature** in the **part1** package. The new class should have a *main* method.

_3. This program's job is to accept data from the user that contain the current temperature. When the user enters this data, it will come into your program as a String; your program will convert to the appropriate Java type, *double*. During conversion, Java will throw an exception if the input data is not correctly formatted, for example if the user enters "abc" for the temperature rather than a numeric string such as "33". Your program will handle these exceptions.

Complete the *main* method as follows:

a. Create a java.util.Scanner object connected to System.in:

```
Scanner scanner = new Scanner(System.in);
```

b. Prompt the user for the temperature, and retrieve that String:

```
System.out.print("Enter the current temperature (F): ");
String sTemp = scanner.nextLine();
```

c. Use the *java.lang.Double* class to attempt to convert the "temperature" String into a *double*:

```
double fahrenheit = Double.parseDouble(sTemp);
```

d. Do a simple calculation and display the results:

```
double celsius = (fahrenheit - 32) * 5 / 9;
System.out.printf("The temperature in Celsius is %2.2f\n",
        celsius);
```

e. Close the scanner.

_4. Run the program and enter a legal integer, e.g. 42 - the program should display the equivalent Celsius temperature.

Run the program again, but this time, enter "abc" - look in the Console for the uncaught exception trace. Note that the "at part1.ConvertTemperature.main" line in the trace has underlined text with the line number - click on the underlined text and note how Eclipse highlights the offending line in the ConvertTemperature.java editing view. This makes it easy to find and fix the problem.

_5. Highlight all of the lines in *main* and choose *Source - Surround With - Try/catch* block so that Eclipse writes simple exception handling for you.

Then modify the Eclipse-generated code in the *catch* block so that it looks like:

```
System.out.println("You entered an illegal number!");
```

_6. Run the program, entering a legal integer - it should work as before.

Try running the program again, this time entering **abc** for the Fahrenheit temperature. What happens? Why?

Part 2: Database Administration and Project Setup

Steps:

_1. You must configure the database for your program. We will use the free, open-source HypersonicSQL database to hold the check information. Follow these steps to configure the database:

a. Open a command-prompt window (choose Start - Run and type **cmd**), then change to the **{Lab Installation Directory}/lib** directory. If your computer has the standard lab setup:

```
c:
cd \Users\username\javaclass\lib
```

Substitute your actual user name for the "username" above.

b. Start the HypersonicSQL server running:

```
java -cp hsqldb.jar org.hsqldb.Server -database mydb
```

The server will start and display a sign-on message. If prompted by Windows, allow access for the server.

Note: Be sure to leave this command-prompt running until you are finished with the lab.

c. Open another command-prompt window and change to same directory as you did in the last step.

d. Start the HypersonicSQL administrator's GUI:

```
java -cp hsqldb.jar org.hsqldb.util.DatabaseManager
```

e. In the GUI, change the *Type* to **HSQL Database Engine Server**, then press OK. **Warning:** Be sure to choose the Server option!

f. In the GUI, in the command area between the Clear and Execute buttons, create the database table, by entering the following command all on one line:

```
CREATE TABLE checks (checknum INTEGER PRIMARY KEY,
accountnum INTEGER,amount DOUBLE)
```

Then press the Execute button.

To test the new table, first choose *View - Refresh Tree*, then press the CLEAR button, then enter:

```
SELECT * from checks
```

Press the Execute button -- you should see an empty table with the proper column names. You can expand the CHECKS table entry in the left-hand navigation tree to see the table schema.

You can close the GUI before continuing, but be sure to leave the server running.

_2. Next, you will import a class that contains the business logic for the program and communicates with the database. As you have done in previous labs, copy and paste the **{Lab Installation Directory}/starters/lab07/checks** folder into the Lab07 project's *src* folder.

_3. Edit the **checks/CheckingAccount.java** file and examine it and see if you can make any sense of the JDBC code that talks to the database.

Find the *writeCheck* method and note what exceptions it throws.

_4. The CheckingAccount class uses the JDBC API to communicate with the database. JDBC requires that you use a *JDBC driver* library -- for this program, we need to ensure that the JDBC driver is accessible to our program. Follow these steps:

a. In the Project Explorer, right-click on the *Lab07* project and choose *Build Path - Add External Archives*.

Navigate to the **${Lab Installation Directory}lib** directory and select **hsqldb.jar**. This archive contains the JDBC driver for HSQLDB. Press Open followed by OK.

Part 3: Handling Exceptions

In this part, you will write a program that writes a check and catch exceptions.

Steps:

_1. Create a new class named **TestChecking** in the **test** package. The new class should have a *main* method. You will add code to *main* so that it retrieves a check number and amount from the user and calls the *writeCheck* method. Here are the details:

a. Create a java.util.Scanner object connected to System.in.

b. Use the CheckingAccount class's two-argument constructor to create a CheckingAccount object with an account number of 12 and an initial balance of 100:

```
CheckingAccount acct = ........
```

c. Write an infinite *while* loop.

d. Inside the loop, ask the user if they want to continue and retrieve a keystroke. If the user enters 'Y', break out of the loop:

```
System.out.print("Quit? ('Y' or 'N'):");
char theChar = scanner.nextLine().charAt(0);
if (theChar == 'Y' || theChar == 'y')
{
    scanner.close();
    break;
}
```

e. In the loop, ABOVE the quit keystroke-checking code, prompt the user, then call the scanner's *nextLine* method to retrieve two Strings -- one for the check number, one for the amount. For example, to prompt the user and retrieve a String for the check number, use code like:

```
System.out.print("Enter check number: ");
String checkNumberStr = scanner.nextLine();
```

f. Use *Integer.parseInt* to convert the check-number String to an *int* and *Double.parseDouble* to convert the amount String to a *double*. For example, to convert the checkNumberStr to an integer, use code like:

```
int checkNumber = Integer.parseInt(checkNumberStr);
```

Hint: Read the JavaDocs for the equivalent method for the *java.lang.Double* class to convert the amount string into a *double*.

g. Call the *writeCheck* method on the *acct* reference, passing the checkNumber and amount integer and double.

h. Save the file to compile. You should have two compilation errors about *Unhandled exceptions* for *ClassNotFoundException* and *SQLException* You will fix those in a moment, but if you have any other compilation errors, fix them before continuing.

_2. Examine the *writeCheck* method and note which exceptions it *throws* -- do the compilation errors make sense?

There are two basic ways to handle exceptions -- you can write a *try-catch* block or pass the exceptions back to whomever called you. We'll start with the latter approach.

Modify the *main* method, adding the clause *throws Exception*. Since Exception is the superclass of all catchable exceptions, this should fix both compilation errors.

_3. Try running the program: pass *123* for the check number and *5.5* for the amount. The program should work fine. Don't quit yet.

When prompted, enter the same check number and amount. What happens? Quit the program.

The problem is that the database is enforcing the primary-key constraint so you cannot insert two checks with the same check number.

Since your *main* method passed the exception back to its caller, the exception was handled by the Java Virtual Machine, which prints the lovely stack trace. While the stack trace is useful to help a developer diagnose the problem, it's probably not a good idea to display that type of information to the user -- programs that react like this to errors do not inspire user confidence.

_4. Let's make this a bit more user friendly by handling the exception in our program. The good news is that Eclipse can help you write *try-catch* blocks. Follow this procedure:

a. Remove the *throws Exception* clause from *main*.

b. Highlight the line that calls *writeCheck*, then right-click on the highlighted area and choose **Source - Surround with try/catch block**.

c. Note that Eclipse determined which exceptions the highlighted line might throw and wrote a *catch* block for each. The Eclipse-generated code is not that user-friendly, so delete it in both *catch* blocks and replace it with:

```
System.out.println(e.getMessage());
```

This code displays the exception object's *message string*, which should be a human-readable description of the exception.

_5. Try running the program again, using the same check number and amount. You should see a message indicating a "violation of unique index...". This message is a little better than the stack trace, but you will improve it in a later part of lab. Don't quit yet.

_6. When prompted, enter "abc" for the check number and any amount. What happened and why? Quit the program.

The problem is that the conversion from String to *int* and *double* can throw the NumberFormatException. Follow this procedure to improve the program:

a. Highlight all of the lines from the Integer.parseInt call down through the end of the last *catch* block for SQLException.

b. Right-click on the highlighted lines and choose **Source - Surround with try/catch block**.

c. In the catch block for NumberFormatException, replace the generated code with:

```
System.out.println("Invalid numeric data entered.");
```

d. Note that you now have a try-catch nested within another try-catch -- that's perfectly OK. The idea here is that we only want to attempt to write the check if both input Strings contain properly formatted numbers.

Try running the program again with "abc" for the check number and any amount. Does that look better? Be sure to quit the program.

_7. Stop the database server by pressing Ctrl+C on its command window, then try running your program with 156 for the check number and 5.5 for the amount. What happens? Does your exception handling make sense?

Quit the program and restart the database server before continuing.

Part 4: Writing Exception Classes

In this part, you will write exception class that encapsulate information about exceptional conditions that this program may encounter.

Steps:

_1. Create a class named **DuplicateCheckNumberException** in the *checks* package by following these steps:

a. Right-click on the *checks* package and choose **New - Class**.

b. In the New Class wizard's first page, enter **DuplicateCheckNumberException** for the *name*. For the *superclass*, press the *Browse...* button and start typing *Exce* into the entry field -- note how the wizard narrows down the superclass name based on your typing. Select *Exception* from the list and press OK.

This new class doesn't need a *main* method. Press Finish to create the class.

c. Edit the new class and write a zero-argument constructor that invokes the superclass:

```
super ("Check numbers must be unique");
```

This defines the *message string* for the new exception class.

_2. Repeat the above step to create an exception class named **InsufficientFundsException**. Use an appropriate message string.

_3. Edit **CheckingAccount.java** and update it to use the new exception classes. Perform the following steps:

a. Add the **DuplicateCheckNumberException** and **InsufficientFundsException** to the list of exceptions thrown by the *writeCheck* method.

b. Find the closing brace of the *if* statement that checks for a non-zero balance in the *writeCheck* method and add an *else* clause:

```
else
{
    throw new InsufficientFundsException();
}
```

c. Find the *catch* block in the *writeCheck* method that catches the *SQLException* and modify it so it looks like:

```
catch ( SQLException e )
{
    if ( e.getErrorCode() == -104 )
        throw new DuplicateCheckNumberException();

    throw e;
}
```

_4. Save CheckingAccount.java and note that there are now errors in TestChecking.java due to the new exception types.

Edit TestChecking.java and write additional *catch* blocks for the new exceptions. In the catch blocks, you can display the message string in a similar fashion as before.

_5. Save, then run the program.

To test, write a check using *123* for the check number and *5.5* for the amount -- you should see the "unique check number" message. Don't quit yet.

Try writing another check with *444* for the check number and *500* for the amount -- you should see your "insufficient funds" message. (The initial balance of the account is $100 -- see the CheckingAccount class to see how this works.)

You can quit the program when you are finished.

Part 5: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts.

Steps:

_1. Create a new class named **EchoFile** in the *io* package. The class should have a *main* method.

_2. Read the JavaDoc about the *java.io.FileReader* class. Note that the class defines a constructor that accepts a file name. Note also that it defines a *read* method that returns a character from the file. Please make special note of the exceptions thrown by the constructor and the *read* method.

_3. In *main*, write code to open the *CheckingAccount.java* file using a *FileReader*. The pathname should be **src/checks/CheckingAccount.java**.

Use a *try-catch* block to handle any exceptions.

_4. Within the *try* block from the previous step, write a loop that reads characters from the file until the *read* method returns -1. Use *System.out.print* to print each character to the console.

After the loop finishes, call the *close* method to close the FileReader.

Use a *try-catch* block to handle any exceptions.

Note: The *read* method returns an *int* so you can tell end-of-file by looking for -1. However, before displaying, you will need to cast the *int* to *char*.

_5. Save, run and test. You should see the text of the file displayed in the console.

_6. Try changing the pathname to *xxxx* and running your program. Does your exception handling kick in?

Lab 8: Arrays and Collections

In this lab, you will work with arrays and collections.

Objectives:

- To work with arrays
- To investigate collection behavior

Part 1: Using Arrays

Steps:

_1. Create a new Java project named **Lab08**.

_2. In the project, create a new class named **Game** in the **pool** package. This class doesn't need a *main* method.

_3. Complete the class to the following specifications:

  - The class should define fields for *gameDate*, *team1Name* and *team2Name* using String as the type for each field
  - The class should define get/set methods for each of the fields. (Remember that Eclipse will generate get/set methods for you!)
  - The class should define a three-argument constructor that accepts a parameter for each of the fields and assigns the parameter values to the fields. Review Lab06 if you cannot remember how to write constructors
  - The class should override *toString* and return a String with labeled field values. Review Lab02 if you cannot remember how to write a toString method

_4. Right-click on the project and choose New - File to create a file named **games.txt**.

Enter the following into the file:

```
2014-10-23
Iowa
Nebraska
2014-10-30
Michigan
Ohio State
2014-11-7
Penn State
Maryland
```

This file cotains data for three games, each game taking up three lines of text.

Note: Ensure that there are no blank lines at the beginning or end of the file.

_5. Create a class named **TestArrays** in the **part1** package. This class should have a *main* method.

_6. Complete *main* according to the following specifications:

  - Define and allocate an array that will hold three Game objects

- Define an integer that will hold an array index, initializing it to zero
- Create a Scanner connected to a FileReader that references your data file:

  ```
  Scanner scanner = new Scanner(new FileReader("games.txt"));
  ```

- Define a loop that will loop three times
- Inside of the loop, read three strings from the file:

  ```
  String date = scanner.nextLine();
  String team1Name = scanner.nextLine();
  String team2name = scanner.nextLine();
  ```

- After retrieving the Strings, create a *Game* object and store the reference in the array using the loop's index variable
- After the loop, write another loop that traverses all of the elements in the array that have objects, passing each object reference to System.out.println
- Close the scanner.
- Highlight all of the lines in *main* and use the menu *Source - Surround with - try/catch block* to generate simple exception handling code.

_7. Run your program. Do you see the correct console output for the games data you entered into games.txt?

Modify games.txt, adding three lines for data for another game. Then change the program's loop(s) so they go through four times. Run the program again. What happens? How could you fix this? Remember that you cannot change the size of an array once it's allocated!

Part 2: Using an ArrayList Collection

In this part, you will convert your program from the previous part to use an *ArrayList* collection rather than a simple array. The primary benefit of this is that you don't need to know in advance how many entries will go into the collection.

Steps:

_1. Create a new package named **part2** and copy the *TestArrays* class into the new package. Then right-click on the copied class and choose **Refactor - Rename** and change the name to **TestArrayList**.

_2. Modify the class so that it uses an ArrayList collection instead of the array. Here are some hints:

- You should use the generic Java 5 ArrayList collection, typed to **Game**.

_3. Test your program -- it should work! Add data for another game or two to games.txt and test again.

Part 3: Using a HashMap Collection

In this part, you will use a *HashMap* collection to store unique words parsed from a String.

Steps:

_1. Create a new class named **WordCount** in the **part3** package. The class should have a *main* method.

_2. In *main*, create a HashMap instance named **words**.

_3. Prompt the user and retrieve a String using a Scanner connected to System.in.

_4. To parse the String, you can use the *java.util.StringTokenizer* class. Please read the JavaDoc and ensure that you understand the basics of how it works.

Use a StringTokenizer to parse the String and loop through the words (assuming you named your String *inputString*):

```
StringTokenizer st = new StringTokenizer(inputString);
while (st.hasMoreTokens())
{
    String word = st.nextToken();

}
```

_5. The goal of this part of lab is keep track of each unique word and how many times the word occurs in the input string. To do this, you will use each word as the *key* in the HashMap -- HashMaps guarantee key uniqueness.

For the *value* associated with the key, we would like to store an integer count of how often the word appears. However, HashMaps require that the *value* be an Object, not a primitive, so we can't use *int*!

This is a perfect example of when the so-called "wrapper" classes come in handy. You will use objects of the java.lang.Integer *class* as the value -- remember that Integer objects internally store an *int* and provides a method to retrieve the *int*.

So inside of the tokenizer loop (after the call to *nextToken*), write the following to retrieve the Integer associated with the current word, creating a new Integer object if it's a new word (i.e. not already in the HashMap):

```
Integer countObj = (Integer)words.get(word);
if ( countObj == null )
    countObj = new Integer(1);
```

Note that if the word isn't in the HashMap, we create a new Integer object with a count of one.

_6. Add an *else* to the *if* statement for the case that the word's already in the collection:

```
else
{
    int count = countObj.intValue();
    count++;
    countObj = new Integer (count);
}
```

If the word already exists in the HashMap, we retrieve its Integer object, increment the count and create a new Integer with the updated count.

_7. We can now store either the new or updated Integer as the *value*, using the word as the *key*:

```
words.put(word, countObj);
```

_8. The entire loop should look like:

```
StringTokenizer st = new StringTokenizer(inputString);
while (st.hasMoreTokens())
{
    String word = st.nextToken();

    Integer countObj = (Integer)words.get(word);
    if ( countObj == null )
        countObj = new Integer(1);
    else
    {
        int count = countObj.intValue();
        count++;
        countObj = new Integer (count);
    }

    words.put(word, countObj);
}
```

_9. Next, you can retrieve the *Set* of keys from the HashMap and iterate the Set, each time displaying the *key* (i.e. the word) and its associated *value* (i.e. the Integer).

Use the HashMap example in the course notes as inspiration.

_10. Run your program, entering the following String (on a single line):

```
Now is the time for all good men to come to the aid of their
country for the cause of freedom
```

Did your program display the correct word counts?

Part 4: Arrays of Arrays

In many scenarios, especially in math-heavy applications, it's useful to work with multi-dimensional arrays. While Java doesn't directly support multi-dimensional arrays, we can get a close approximation using arrays that contain other arrays.

One nice use of a two-dimensional array is to represent the mileage distance matrix between cities you often see on maps. Figure 1 shows such a matrix for the state of Iowa:

	Cedar Rapids	Davenport	Des Moines	Dubuque	Iowa City	Keokuk	Mason City	Omaha	Sioux City	Waterloo
Davenport	80	N/A	167	70	56	141	226	301	364	136
Des Moines	126	167	N/A	200	113	202	119	131	198	125
Sioux City	324	364	198	398	310	399	213	97	N/A	236
Waterloo	55	136	125	91	85	170	91	263	236	N/A

Figure 1

In this lab, you will write a method that returns the distance between two cities, given their names.

Steps:

_1. Create a new class named **IowaDistanceMatrix** in the **part4** package. The class should have a *main* method.

_2. Next, you need to define the mileage matrix as an array containing an array and two String arrays that contain the names of the cities. Typing these in is tedious, so you can copy and paste the definitions from **{Lab Installation Directory}/starters/lab08/arrays.txt** and we've repeated them here:

```
static private int[][] matrix =

{
    { 80,  -1, 167,  70,  56, 141, 226, 301, 364, 136},
    {126, 167,  -1, 200, 113, 202, 119, 131, 198, 125},
    {324, 364, 198, 398, 310, 399, 213,  97,  -1, 236},
    { 55, 136, 125,  91,  85, 170,  91, 263, 236,  -1},
};

static private String[] rows =
               {"Davenport",
                 "Des Moines",
                 "Sioux City",
                 "Waterloo"
                };

static private String[] cols =
                {"Cedar Rapids",
                 "Davenport",
                 "Des Moines",
                 "Dubuque",
                 "Iowa City",
                 "Keokuk",
                 "Mason City",
                 "Omaha",
                 "Sioux City",
                 "Waterloo"
                };
```

Note that the *matrix* array defines an array of *int[]*, and each subarray is of type *int* (a numeric literal with no decimal point is interpreted as *int*). Also note that we use -1 to represent the "N/A" value (distance from a city to itself).

Note that the *rows* and *cols* arrays are one-dimensional String arrays and contain the city names of the rows and columns in Figure 1.

_3. Next, define a static method that will return the distance between two cities:

```
public static int distance(String city1, String city2)
{
    return 0;
}
```

Complete the method so that it uses the *city1* String for a "row" city and "city2" as a "column" city and returns the distance between them. Here are some more specifications and hints:

- One way to implement this is to do two loops: one that looks up *city1* in the *rows* array and determines its index, and another loop that looks up *city2* in the *cols* array to find its index. You can then use the two indexes to retrieve the mileage from the *matrix* array
- Remember to compare the characters in a String, you use the *equals* method!

- If the specified city is not in the row or column array, the method should throw the *IllegalArgumentException*. This is not a *checked* exception, so you don't need to declare it as a *throws* clause on the method definition

_4. In the *main* method, call the *distance* method, passing the names of two cities and print the result to the console.

Part 5: Measuring Traversal Performance

In this part, you will compare how long it takes to traverse a List collection using an Iterator versus using the collection's *get* method.

Steps:

_1. Create a new class named **MeasureTraversal** in the **part5** package. The new class should have a *main* method.

_2. In *main*, begin by writing code to create a LinkedList collection and store 30000 String objects in it:

```
List<String> list = new LinkedList<String>();

for (int i=0; i<30000; i++)
    list.add("String" + i);
```

_3. Next, mark the current system time:

```
long startTimeGet = System.currentTimeMillis();
```

_4. Then write code to retrieve all of the Strings using each String's index to retrieve it:

```
for (int i=0; i<list.size(); i++)
{
    String s = (String)list.get(i);
}
```

_5. Then retrieve the current time after the loop (i.e. the endTimeGet). Subtract the start time from end time and print the result to the console.

_6. Repeat the timing and traversal code, but this time use an Iterator instead using a *for* loop and the *get* method:

```
Iterator iter = list.iterator();
while (iter.hasNext())
{
    String s = (String)iter.next();
}
```

Again, print the time difference to the console.

_7. Run your program and write down the two times (they are in milliseconds). Which traversal technique is faster for a LinkedList?

_8. Change the collection type from *LinkedList* to *ArrayList* and repeat the experiment. How do the numbers stack up? Why is the performance for *get* so slow on a LinkedList?

Part 6: Reinforcement

In this part, you will practice the techniques covered in this lab to reinforce the concepts.

Steps:

_1. Create a new class named **PrintCommandLineArgs** in the **part6** package. The new class should have a *main* method.

_2. Write code in *main* so that the program simply prints each of its command-line arguments to the console in the form:

```
arg[0]=xxxx
arg[1]=yyyy
```

_3. Review Lab01 on how to configure command-line arguments in Eclipse, then run your program.

Lab 9: Inheritance and Polymorphism

In this lab, you will work with inheritance and polymorphism.

You will write three classes that implement a class hierarchy using inheritance. The application is for a business that purchases automobiles from domestic and foreign manufacturers. The business must pay a duty tax on each foreign car that is imported -- the duty calculation depends on the car's origin. Here is the UML class diagram for the system:

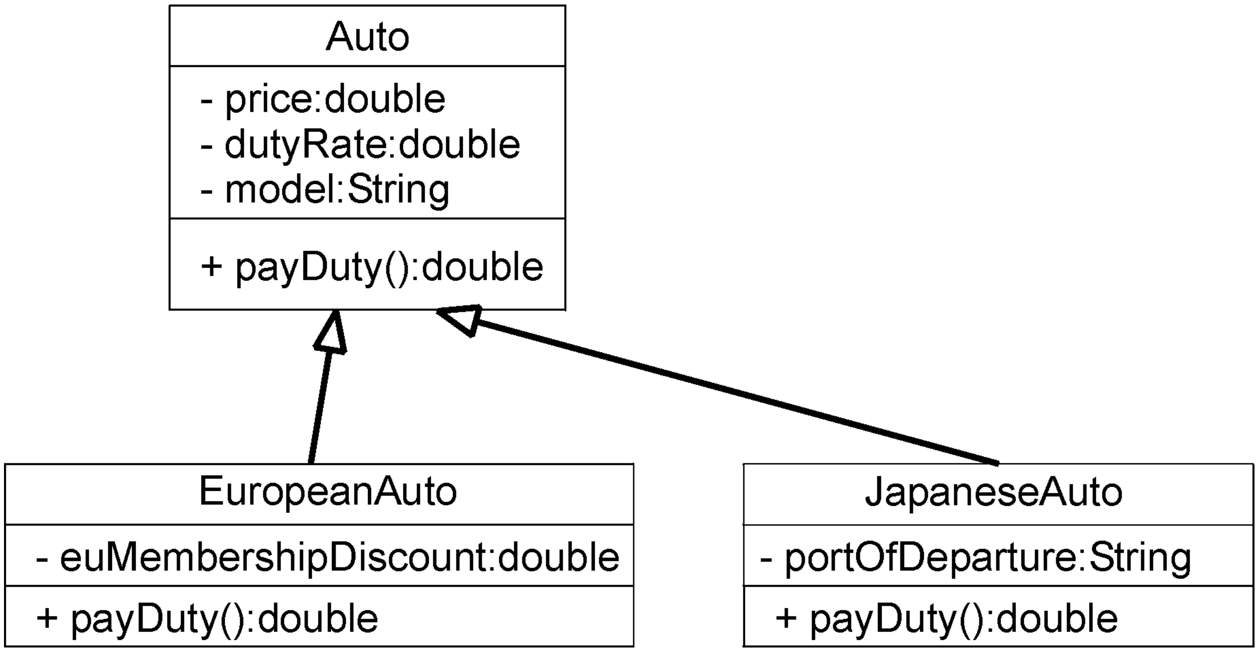


Figure 1

Here are the business rules:

- Unless specified otherwise, the duty is zero
- For European cars, the duty is the (price * the dutyRate) less the (price * euMembershipDiscountRate)
- For Japanese cars, the duty is the (price * the dutyRate), but there's an extra 5% departure tax for vehicles shipped from Yokohama -- the departure tax is based on the vehicle price

Objectives:

- To write subclasses that override methods
- To investigate constructors in subclasses

Part 1: Inheritance

Steps:

_1. Create a new Java project named **Lab09**.

_2. Create a new class named **Auto** in the *inventory* package. The class does not need a *main* method.

Complete the class according to this specification:

a. Add the fields to the class as documented in the UML diagram. All fields should be private.

b. Use Eclipse to generate get/set methods for all of the fields.

c. Write a *payDuty* method as described in the UML diagram. The method should return the value zero.

_3. Create a new class named **EuropeanAuto** in the *inventory* package that extends **Auto**. The class doesn't need a *main* method.

Hint: In the New Class wizard, you can choose the superclass by pressing the Browse button next to the superclass entry field and then start typing "Aut" into the search field.

Complete the class according to the following specifications:

a. Add the fields as specified in the UML diagram. All fields should be private, but your methods should be public.

b. Use Eclipse to generate get/set methods for the fields.

c. Write a public *payDuty* method as described in the UML diagram.

Implement this method using the business rule defined above. Note that to perform the duty calculation, you will need to call the *getDutyRate* and *getPrice* methods from the superclass:

```
double dutyRate = getDutyRate();
double price = getPrice();
```

Why can't the subclass access the fields directly?

_4. Create a new class named **JapaneseAuto** in the *inventory* package. The class doesn't need a *main* method.

Complete the class according to the following specifications:

a. Add the fields as specified in the UML diagram. All fields should be private.

b. Use Eclipse to generate get/set methods for the fields.

c. Write a *payDuty* method as described in the UML diagram.

Implement this method using the business rule defined above. Note that you will need to call the *getDutyRate* and *getPrice* methods from the superclass.

For the *portOfDeparture* portion of the calculation, you can use the String class's *equals* method on the *portOfDeparture* field to determine if the JapaneseAuto was shipped from Yokohama or not:

```
boolean f = portOfDeparture.equals("Yokohama");
```

_5. Create a new class named **TestInventory** in the *test* package. This class should have a *main* method.

_6. Add code to *main* as described below.

a. Create objects for an *Auto*, an *EuropeanAuto* and a *JapaneseAuto*. Call the *set* methods to initialize their fields using the following data:

```
Object               Model  Price  Duty Rate
------               -----  -----  ---------

Auto object          Ford   20000    0%
EuropeanAuto object  BMW    35000    5%
JapaneseAuto object  Honda  30000    7%
```

The EuropeanAuto object should have an *euMemberShipDiscount* value of 2%.

The JapaneseAuto object was shipped from Yokohama.

b. Call the *payDuty* method on each object and print the result to the console.

_7. Run and test your program to see if the duty calculations are correct.

Part 2: Polymorphism and Collections

In this part, you will create a collection of Autos and use polymorphism to simplify the task of calculating duty.

Steps:

_1. Create a new class named **Inventory** in the *inventory* package. This class should not have a *main* method.

_2. Define a private field in the class of type *ArrayList* named **inventory**. Be sure to call *new* to initialize this field.

_3. Define a method named **addAuto** that accepts an *Auto* reference and returns *void*. This method should simply add the reference to the *inventory* collection.

_4. Define a public method named **calcTotalDuty()** that accepts no parameters and returns a *double*. Here are some hints on implementing this method:

a. Define a local variable named **totalDuty** of type *double* and initialize it to zero.

b. Set up a loop using the enhanced *for* loop to traverse all of the objects in the collection.

c. Inside of the loop, for each object retrieved, call the *payDuty* method, adding the the returned duty to the *totalDuty* variable.

d. Return the *totalDuty*.

_5. Edit the *TestInventory* class and do the following:

a. At the top of *main*, create an *Inventory* object.

b. After creating and initializing the Auto objects, call the *Inventory* object's *addAuto* method to add each of the Auto objects to the collection.

c. Call the *Inventory* object's *calcTotalDuty* method and print the result to the console.

_6. Run and test your program. Is the total duty calculation correct?

Part 3: Adding a New Class

In this part, you will add a new type of Auto to the application and demonstrate how polymorphism makes it easy to add function to a program without affecting existing code.

You will write a new class named *AustralianCar* that is a subclass of Auto. The duty calculation for these cars is simply the *dutyRate* times the *price*.

Steps:

_1. Define a new class named **AustralianAuto** that extends **Auto** in the *inventory* package. This class does not need a *main* method.

_2. In the new class, write a *payDuty* method that implements the business rule described above.

_3. Edit TestInventory, and in *main*, create an *AustralianAuto* and initialize its fields:

```
Object                 Model  Price  Duty Rate
------                 -----  -----  ---------

AustralianAuto object Holden 33000    8.5%
```

Add the AustralianAuto object to the Inventory collection.

Add code to print out the AustralianAuto's duty to the console.

_4. Run and test your program. The key point here: What changes did you need to make to the total-duty calculation? The answer is: NONE! This is polymorphism in action -- polymorphism lets you add new subclasses without affecting existing algorithms. Look back at the *calcTotalDuty* method in the *Inventory* class and make sure you understand how this works.

Part 4: Constructors and Inheritance

In this part, you will investigate how constructors work in subclasses and superclasses.

Steps:

_1. Edit the *Auto* class and write a three-argument constructor that accepts parameters for all of the fields and assigns the parameter values to the fields.

Save the file and note the Problems View -- you should now have errors in the *TestInventory* class and in each of the subclasses.

The error in the TestInventory class should be expected -- that code explicitly used the built-in zero-argument constructor that no longer exists -- but what about the subclasses? (Remember that if you define no constructors, Java provides a built-in zero-argument constructor. If you write any other constructor, Java "takes back" the built-in one.)

The reason the subclasses have errors is that their built-in zero-argument constructors implicitly call the zero-argument constructor of the superclass -- since that zero-argument constructor no longer exists, the subclasses are in error.

_2. Fix the *TestInventory* class's error by changing the Auto object's constructor to use the three-argument constructor. You can then comment-out the *set* methods for this object, as they are redundant.

_3. To fix the subclasses, you will need to write an explicit constructor in each that explicitly calls the superclass's three-argument constructor. Follow these steps:

a. In the *EuropeanAuto* class, write a four-argument constructor that accepts parameters for all of the Auto fields as well as the *euMembershipDiscount* field. In the constructor, first call the superclass three-argument constructor, then assign the euMembershipDiscount field.

Then edit *TestInventory* and for the EuropeanAuto object, change it to use the new four-argument constructor and then comment-out the *set* methods.

b. Repeat the above step for the *JapaneseAuto* class, but the "extra" parameter is for the *portOfDeparture* rather than *euMembershipDiscount*.

c. Repeat the above step for the *AustralianAuto* class -- since this class contributes no new state, it needs only a three-argument constructor.

_4. Run and test your program. You should see the same results as before.

Part 5: Overriding Methods to Augment Function

In this part, you will investigate how a subclass method can invoke the implementation in a superclass.

Steps:

_1. Edit the *Auto* class and write a *toString* method that returns a String containing labeled values of all the object's fields.

In this method, do NOT invoke the superclass's (Object) implementation of toString. That means that the Auto class is *replacing* the implementation provided by the superclass.

_2. Edit the *EuropeanAuto* class and write a *toString* method that returns a String containing the labeled values for not only fields contributed by EuropeanAuto, but also the fields from the Auto superclass. The code should look something like:

```
public String toString()
{
    String s = super.toString();
    s += "EU discount: " + euMembershipDiscount;
    return s;
}
```

Note that this method invokes the superclass behavior and then *augments* it. Compare this to the implementation in *Auto* that completely *replaces* the behavior from its superclass.

_3. Repeat the above step for the *JapaneseAuto* class, augmenting the String with state contributed by the JapaneseAuto class.

_4. Examine the *AustralianAuto* class -- since it contributes no additional state, you don't need to write a new toString method -- the implementation from the Auto superclass should be sufficient.

_5. Edit TestInventory and add code that calls System.out.println on the Auto object reference and on the objects of each of the subclasses (you don't need to print the Inventory object).

_6. Run and test your program -- do you see the augmented output?

If you're not happy with the way the output is formatted, edit the toString methods to fix it.

Part 6: Optional: Reinforcement

In this optional part, you will do some more work with inheritance.

Note: This part is quite time-consuming and challenging. Don't feel bad if you don't have time to completely finish it!

Your job is to implement a two-function calculator that supports addition, subtraction and, most importantly, an *undo* facility. The standard way to code an *undo* facility is to implement the *Command* design pattern. The following figure shows a UML diagram for how you will implement the system:

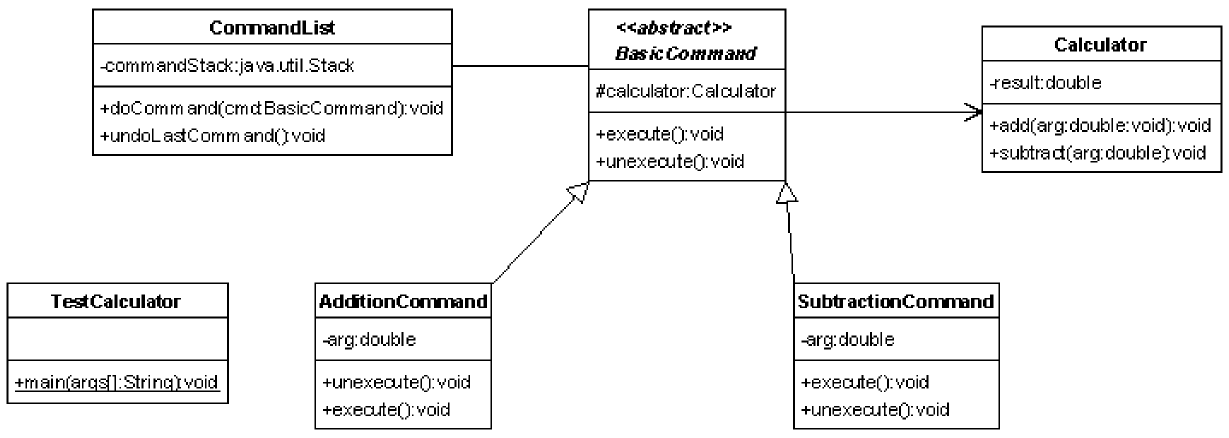


Figure 2

Here are a list of specifications for the program:

- The *BasicCommand* class should be an *abstract* class and the *execute* and *unexecute* methods should be abstract. This class should maintain a field that references a *Calculator* object and define a one-argument constructor that accepts the Calculator object reference. The Calculator field should use *protected* access
- The *AdditionCommand* and *SubtractionCommand* classes subclass *BasicCommand* and should each define a field for an argument and define a two-argument constructor that accepts a Calculator object and the argument. Each class should implement *execute* and *unexecute* in an appropriate manner by calling a method in the Calculator
- The *Calculator* class should define a field that contains the current result and a *get* method for the result. It should also define *add* and *subtract* methods that update the result
- The *CommandList* class should define a field of type *java.util.Stack*. Stack is a collection that provides methods to add an object to end of the stack (push) and to pop off the last object pushed. The *doCommand* method should execute the command and then push it onto the stack. The *undoLastCommand* method should pop the command and then call *unexecute* on it
- The *TestCalculator* class should define a *main* method that implements the following activity diagram (flow chart):

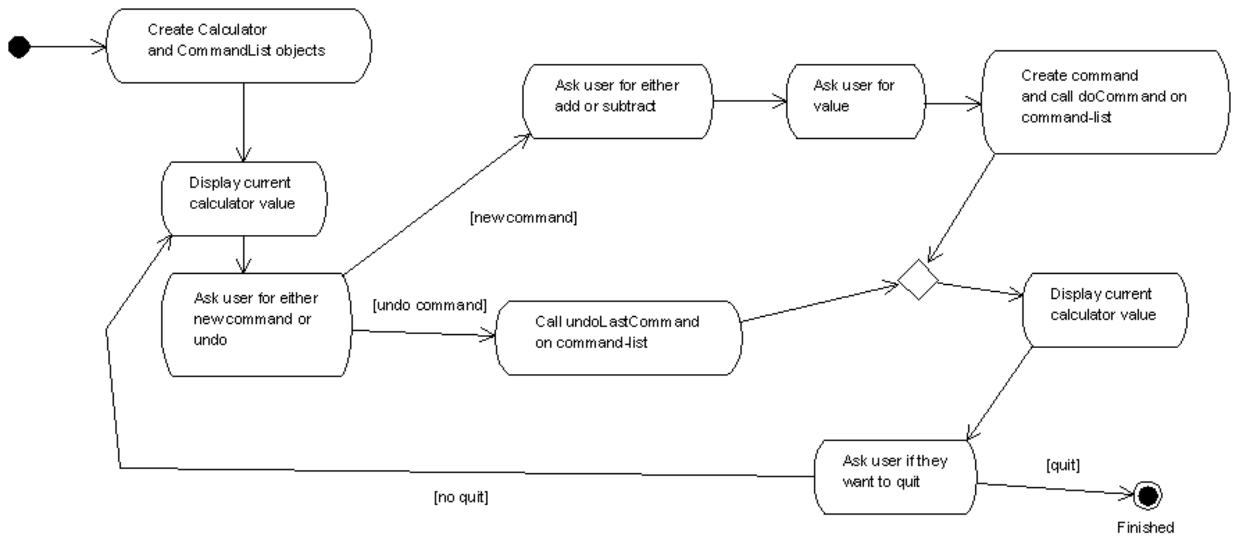

Create Calculator and CommandList objects
Display current calculator value
Ask user for either new command or undo
[new command]
Ask user for either add or subtract
Ask user for value
Create command and call doCommand on command-list
[undo command]
Call undoLastCommand on command-list
Display current calculator value
Ask user if they want to quit
[quit]
Finished
[no quit]

Figure 3

Steps:

_1. Create a package named **part6**, and in it, write the classes that implement the requirements described above.

_2. Run and test. To test, try executing a couple of addition and subtraction commands, then un-doing them. Do the results make sense?

Lab 10: Interfaces

In this lab, you will work with interfaces.

Objectives:

- To write an interface
- To write a class that implements an interface

Part 1: Implementing an Interface

In this part, you will write a class that implements an interface -- the interface specifies the required behavior for filtering lists of files.

Steps:

_1. Use Windows Explorer to examine the contents of the **{Lab Installation Directory}/stuff** directory -- note that it contains a mixture of *.txt* and other files.

If your computer has the standard lab setup, that directory is:

```
c:\Users\username\javaclass\stuff
```

_2. In Eclipse, create a new Java project named **Lab10**.

_3. Look in the JavaDoc for the *java.io.File* class. The File class represents an file or directory on a disk and provides methods to let you work with the file or directory.

Note that the class has a constructor that accepts a pathname String (which could be for a directory) and that it defines overloaded *listFiles* methods that return an array containing entries representing files in the directory.

One of the *listFiles* methods accepts an argument of type *FileFilter*, which is an interface -- classes that implement this interface can filter the returned list of files.

Create a class named **MyFileFilter** in the *part1* package that implements the *FileFilter* interface. This class does not need a *main* method, but ensure that the *Inherited abstract methods* checkbox is selected.

Hint: In the New Class wizard, you can add interfaces by pressing the Add button next to the Interfaces entry field and then starting to type "FileFil".

_4. To implement *FileFilter*, your class must provide a method named *accept* that accepts a File object and returns *true* if the file should be listed, false if not.

Complete this method so that it returns true for files whose *path* ends in *.txt* and false otherwise. Hint: look at the File class's *getPath* method and the String class's *endsWith* method. Here's some code to help you get started:

```
String path = arg0.getPath();
boolean ok = path.endsWith(".txt");
```

_5. Create another class named **ListFiles** in the *part1* package. This class should have a *main* method.

_6. Complete *main* so that it:

a. Creates a File object using the File class's one-argument constructor that accepts a String pathname. Set the pathname to **{Lab Installation Directory}/stuff**, which is the directory containing a mixture of text and other files.

b. Creates an instance of the MyFileFilter class using the no-argument constructor.

c. Calls the File object's *listFiles* method, passing the MyFileFilter object reference. Note that this method returns an array of File objects -- each entry in the array represents a file in the directory. If your filter is working properly, the array should only contain File objects for files whose name ends in .txt.

d. Has a *for* loop that traverses the returned array of File objects, printing each File object to the console.

_7. Run and test your program -- do you see a listing of only file names that end in *.txt*?

Part 2: Writing an Interface

In this part, you will write an interface. The company that imports automobiles has decided to expand to import other goods that require a duty payment. You will thus write an interface that describes the behavior required by all dutiable items.

Steps:

_1. Use the Eclipse wizard to create a new interface named **Dutiable** in the *part2* package.

Add two methods to the interface:

```
public double payDuty();
public double getDutyRate();
```

_2. Create a new class named **SpeedBoat** in the *part2* package. The new class should implement the *Dutiable* interface. It does not need a *main* method.

Complete the class to the following specifications:

a. Define a field named **price** of type *double*.

b. Implement the two interface methods. The duty rate for *SpeedBoats* is 7.5% of the *price*.

c. Write a constructor that accepts the price and assigns it to the field.

_3. Create a class named **ExoticReptile** in the *part2* package that implements the *Dutiable* interface. Complete the class:

a. Define a field named **price** of type *double*.

b. Implement the two interface methods. The duty rate for *Exotic Reptiles* is 11% of the *price*.

c. Write a constructor that accepts the price and assigns it to the field.

_4. Create a new class named **TestDutiable** in the *part2* package. This class should have a *main* method. Complete *main*:

a. Create an ArrayList collection typed to *Dutiable*. Note how we type the collection to the interface type, which acts as the base type for everything we will put in the collection.

b. Create a few SpeedBoat and ExoticReptile objects and store their references in the collection.

c. Use an loop to calculate the total duty, then display the total and average to the console.

_5. Run and test your program.

Part 3: Discriminating Based on an Interface

In this part, you will write a program that takes different action based on whether an object implements an interface or not.

Steps:

_1. Create a new package named **part3** and copy all of the classes from *part2* to the new package.

_2. Create a new class named **GMOFruit** in the *part3* package. This new class should NOT implement the *Dutiable* interface.

_3. Edit TestDutiable and trye to add a *GMOFruit* object to the collection in addition to the *SpeedBoat* and *ExoticReptile* objects.

What happens when you compile and save? Why?

_4. Modify the ArrayList so its type is *java.lang.Object* instead of *Dutiable*. This essentially allows the collection to hold anything since Object is the root of all Java objects. The compiler should be happy.

Modify the iteration loop so it's typed to java.lang.Object. Note that each time through the loop, you will need to cast from Object to Dutiable so you can call the payDuty() method.

_5. Run the program - what happens?

_6. Modify the loop so it uses the *instanceof* operator to determine if each object is of type *Dutiable* before casting and calling the *payDuty* method.

_7. Run and test.

Part 4: Reinforcement

In this part, you will work with a program that uses interfaces to reinforce the topics you learned in this lab.

Steps:

_1. Import the *part4* directory from **{Lab Installation Directory/starters/lab10** into the *Lab10* project's **src** folder.

_2. Examine the *MyWindow* class. This class represents a Java Swing GUI window with a frame. Run the program to see what it does currently (it's not very exciting).

_3. Your job is to modify the program so that it responds to mouse clicks on the window by drawing the text "Hello" at the point where the mouse was clicked.

Start by reading the JavaDoc about the *java.awt.event.MouseListener* interface -- note that it defines a method named *mousePressed* that the windowing system will call when the user clicks the mouse.

Also read about the *MouseEvent* object that's passed on *mousePressed* -- especially note that it makes the mouse-click coordinates available.

_4. Modify the *MyWindow* class so that it implements the *java.awt.event.MouseListener* interface. Then save the file -- note that you have errors regarding the interface methods.

_5. Right-click on the editing view and use Eclipse's **Source - Override/Implement Methods** menu to generate empty methods for the interface.

_6. Complete the *mousePressed* method so that it draws "Hello" at the mouse-click coordinates. To draw to the window, you can use the *java.awt.Graphics* class's *drawString* method. You can retrieve a *Graphics* object, with this code:

```
Graphics g = getGraphics();
```

_7. Find the *MyWindow* constructor, and add this line above the call to *setSize*:

```
addMouseListener(this);
```

This call registers the MyWindow object with the windowing system for mouse events.

_8. Run the program and test it by clicking the mouse on the window -- do you see the "Hello" text?