

**WA2892 Booz Allen Hamilton Tech
Excellence Modern Software
Development Program - Phase 1**

Student Labs

**LabGuide 1
(Java Labs)**

Web Age Solutions Inc.

Table of Contents

Lab 1 - The HelloWorld Class.....	3
Lab 2 - Refining The HelloWorld Class.....	19
Lab 3 - The Arithmetic Class.....	24
Lab 4 - Project - Prompt and Store StockAccount Information (Optional).....	36
Lab 5 - Creating A Simple Object.....	41
Lab 6 - Project - Create a StockAccount Class (Optional).....	52
Lab 7 - Getters and Setters.....	54
Lab 8 - Project - Improve Encapsulation (Optional).....	63
Lab 9 - Using Constructors.....	68
Lab 10 - Project - Add Constructors (Optional).....	78
Lab 11 - Project - Create a Stock Class (Optional).....	81
Lab 12 - Project - Buy Stock (Optional).....	84
Lab 13 - Project - Sell Stock (Optional).....	88
Lab 14 - Looping.....	93
Lab 15 - Project - Loop Until Quit (Optional).....	100
Lab 16 - Subclasses.....	106
Lab 17 - Project - Dividend Stocks (Optional).....	117
Lab 18 - Arrays.....	121
Lab 19 - Method Overriding.....	129
Lab 20 - Project - Improved Stock Output (Optional).....	140
Lab 21 - Exception Handling.....	144
Lab 22 - Project - Better Error Handling (Optional).....	153
Lab 23 - Interfaces.....	156
Lab 24 - Collections.....	169
Lab 25 - Project - Multiple Stocks (Optional).....	179
Lab 26 - Introduction to Lambda Expressions.....	189
Lab 27 - Writing To A File.....	205
Lab 28 - Project - Saving to a File (Optional).....	212
Lab 29 - Build Script Basics.....	216
Lab 30 - Build Java Project, Management Package Dependencies, and Run Unit Tests with Gradle.....	227

Lab 1 - The HelloWorld Class

Time for lab: 40 minutes

In this lab, you will use Eclipse to write a very simple “Hello World” Java class.

Part 1 - Start Eclipse and Explore the IDE

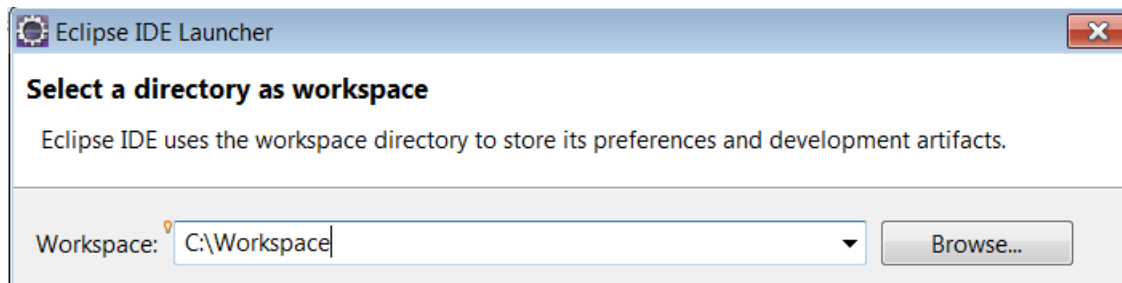
Eclipse is a **Java Integrated Development Environment (IDE)**. It is a GUI-driven and tool-assisted program that we can use to develop Java code in an effective and efficient manner. While developing Java code does not necessarily require an IDE (all the exercises in this lab guide could be achieved by using a simple text editor and the command line compiler), using an IDE will greatly simplify development and decrease coding time. As such, we will use Eclipse to develop all our code in this class.

We will begin by launching Eclipse and exploring some of its facets.

- __ 1. Open a file browser and navigate to **C:\Software\eclipse**.
- __ 2. Run **eclipse.exe** by double clicking on the file.

The *Workspace Launcher* dialog will appear.

- __ 3. Change the **Workspace** to **C:\workspace** as shown below and click **Launch**.

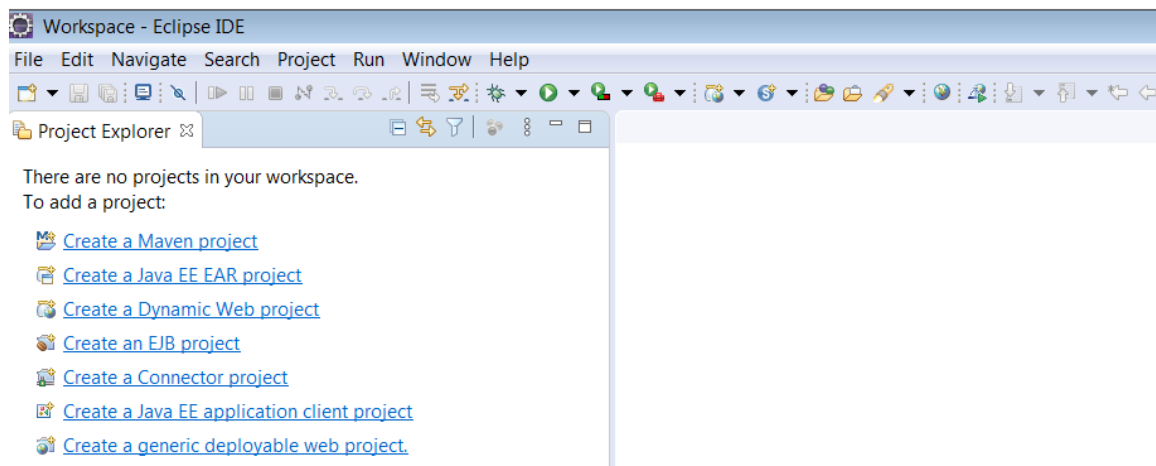


Eclipse will start, and the *Welcome* screen may be shown.



___ 4. Close the Welcome screen.

___ 5. From the menu bar, select **Window | Perspective | Open Perspective | Java**.

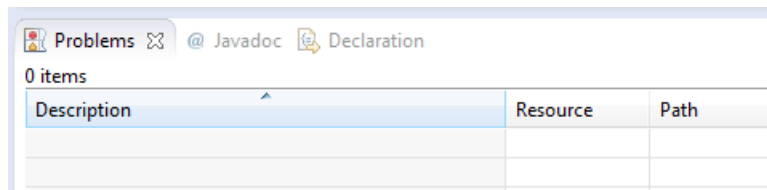


What has happened? Simple: you have changed *perspective*. A perspective in Eclipse is a collection of views chosen to achieve a specific task. We are currently looking at the **Java** perspective, for which Eclipse opens the appropriate *views*.

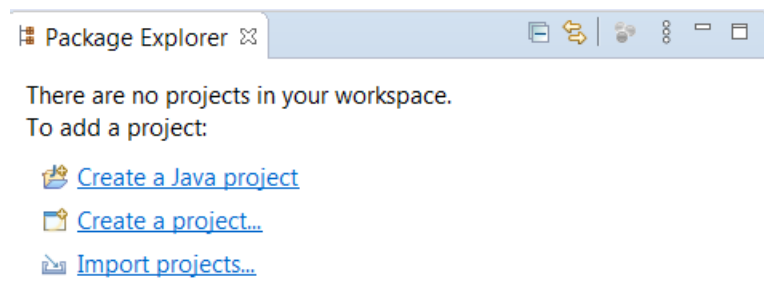
A view is simply a tiled window in the Eclipse environment. For example, in the screen shot above, you can see the *Package Explorer* view, the *Task List* view, and part of the *Outline* view.

Look at your Eclipse environment and locate the following *views*:

The *Problems* view. This should be at the bottom of the Eclipse environment. Any coding/compile errors you make will be displayed here and (in a fit of unbridled pessimism) you will most likely be referring to this view quite a bit.

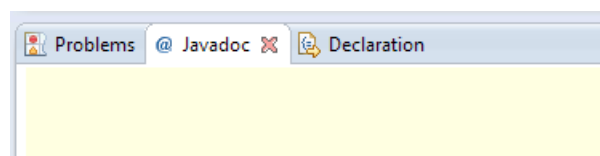


The *Package Explorer*. This is on the left of the IDE.



This will allow us to navigate through the Java classes we will be creating shortly.

__6. Open the *Javadoc* view. To find it, click the *Javadoc* tab next to the *Problems* tab.



__7. You can switch between the two views by clicking the appropriate tabs.

__8. Similarly, locate the *Declaration* view.

The big empty space in the middle of the IDE does not look very interesting right now; that is because it is the *Code* view – and we currently do not have any code to display. Do not worry about it for now.

__9. Each view can be moved and resized. Click on the *Package Explorer* tab and drag it over the area where the *Problems* view is. Note that the view “docks”. Experiment with resizing and moving. Don't worry about “messing up” the perspective, because we will *reset* it in a moment.

__10. From the menu bar, select **Window | Perspective | Reset Perspective** and click **OK** in the box that appears. All the views should “reset” to their original layouts. Note that it is possible to *Save* a perspective that you have changed. This allows you to customize exactly which views you want open and how you want them laid out. This is something you will probably do as you become more experienced with Eclipse.

Part 2 - Create A New Java Project

Before we can create any Java code, we first need to create a *Java Project*. A project in Eclipse is essentially a folder for related files. We will create a new project and then create our classes in that project. Later on, if we work on a completely unrelated Java class, we could then create a separate, new project and create those classes in there. This helps keep our code environment organized.

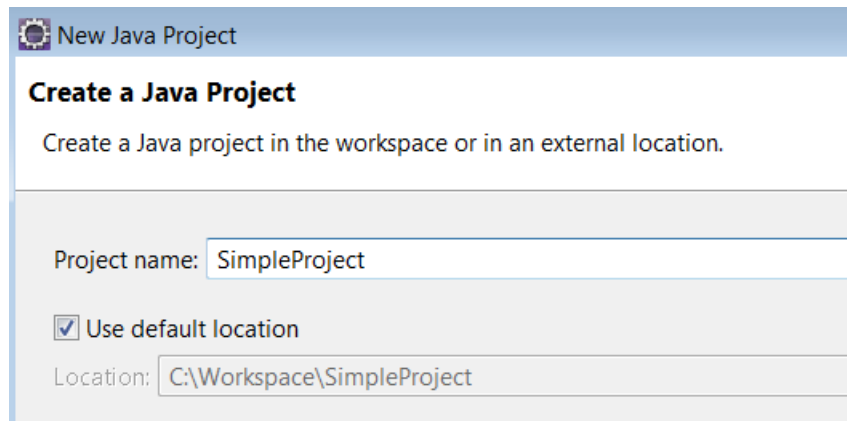
__1. From the menu bar select **File | New | Java Project**

Eclipse can actually handle many different types of projects, depending on the code that needs to be developed. In our case, we will create a simple Java project.

The *New Java Project* wizard will begin.

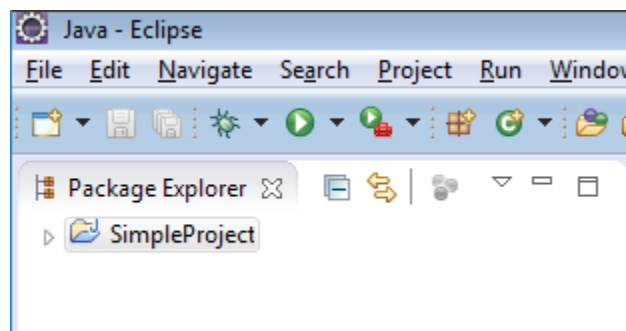
Here, we will describe the details of the Java project that will contain our simple HelloWorld Java.

___2. Set the *Project name:* to be **SimpleProject** and click **Finish**.



A *New module-info.java* window will open, click **Don't create**.

The project will be created. How do you know the project was created? Simple. Look in the *Package Explorer* view.



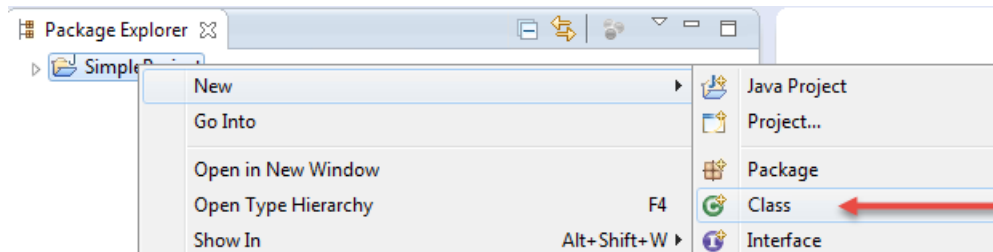
You should see the newly created project listed there. Our Java code will go in this project.

Part 3 - Create a Java Class

Recall that in Java, all code must be contained within a Java class. Within a class, code is typically placed within *methods*, and any class can have a **main** method which will serve as its “executable” method.

We will now create a Java class called “HelloWorld” that will have a single **main** method, and this method will simply print out “Hello World” to the *console*.

___ 1. In the *Package Explorer* view, right click on **SimpleProject** and select **New | Class**



The *New Java Class* window will appear.

This wizard will create a new Java class for us, and allows us to specify some options on the class. When we click the *Finish* button on this wizard, Eclipse will generate a new Java class according to the options we specify here. Wizards like this are one of the reasons that Eclipse is so useful; while it would have been possible to generate the class ourselves, Eclipse does it much faster.

Most importantly, we need to give the class a **name**.

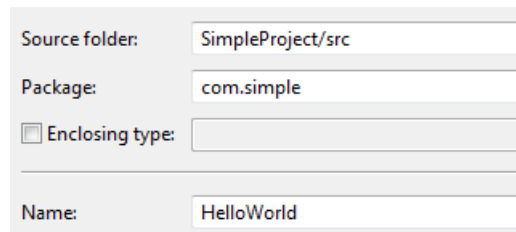
Java classes should also belong to a **package**, for naming convention reasons. Think of a package as a name prefix. Imagine you are working at an insurance company in a Claims department, and you create a class called **Account**. However, imagine that another developer that you are working with (same company, but in the Billing division) also creates a class called **Account**. These two classes are completely different, but have the same name. How would we know which one to use?

The answer is that each class can be placed in a *package*. A package is a string of characters separated by periods (e.g. com.mycompany.www). The characters can be anything (although typical Java naming convention usually has your company name or URL as a part of the package name), and it is pre-pended to the class name. So a “fully qualified” class name is the name of the class preceded by its package.

In the example above, the Billing department could be assigned a different package name from the Claims department; so there could be **com.insuranceco.billing.Account** and **com.insuranceco.claims.Account** as two separate entities. Even though the classes are both called **Account**, they belong to unique packages, thus allowing for differentiation.

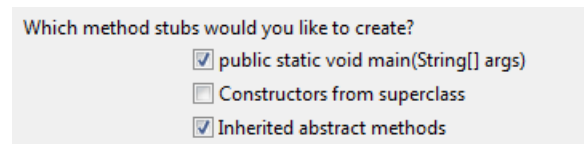
In any case, we will use the *New Java Class* dialog to specify a class name and package name for our new class.

__2. For the *Package* enter **com.simple** and for the name, enter **HelloWorld**



Source folder:	SimpleProject/src
Package:	com.simple
<input type="checkbox"/> Enclosing type:	
Name:	HelloWorld

__3. In the section *What method stubs would you like to create?* make sure that the box for **public static void main(String[] args)** is checked.



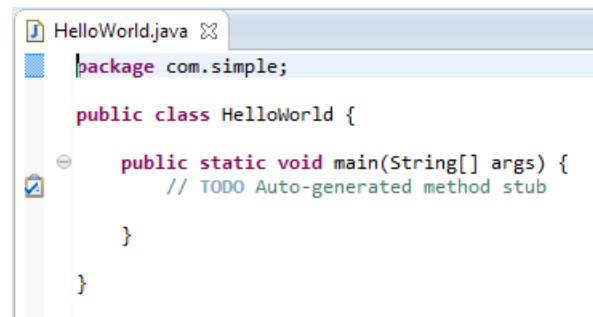
Which method stubs would you like to create?	
<input checked="" type="checkbox"/>	public static void main(String[] args)
<input type="checkbox"/>	Constructors from superclass
<input checked="" type="checkbox"/>	Inherited abstract methods

This means that when the class is created, Eclipse will automatically generate the **main** method for us.

You can ignore the rest of the options for now.

__4. Click **Finish**

Eclipse will now generate the Java class. A lot has now changed in the various views. Firstly, the empty space that was in the center of the environment is now showing Java code. This is the Code Editor view, and it is showing the results of the *New Java Class* wizard's generation. Examine it.



```

HelloWorld.java
package com.simple;

public class HelloWorld {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }

}
```

Firstly, notice that the class this view is currently showing is called **HelloWorld.java**. You can tell this by looking at the name of the tab.

This Code Editor view is *editable*. You can place the cursor inside this view and type away; this is where you will do your coding. However, before we examine the code, let us examine some of the other views.

__5. Examine the *Package Explorer* view.

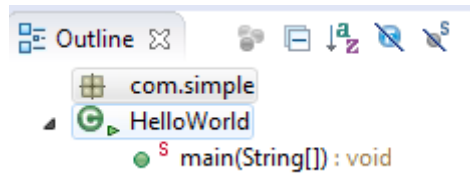
__6. The *Package Explorer* provides us with a high-level navigable view of each project and their contents. At the moment, the only project is our **SimpleProject**

Note the tree structure. The top level is our project (**SimpleProject**) and immediately underneath that is the package **com.simple**

Beneath the package is our class, **HelloWorld.java**. If you double-clicked on it (or any other class that might be listed here), Eclipse would open it in the code editor window.

If we created new packages/classes in the project, they would similarly appear here in the tree structure.

__7. Examine the *Outline* view.



This view presents a summary of the class that is currently open in the code editor view. At the moment, it is examining the HelloWorld class (as represented by the green C) that belongs to the **com.simple** package. The outline is also showing that the class currently has one method: a *static public* method called **main(String[])**. We know it is a *public* method because of the green circle next to it, and we know that the method is *static* because of the S. (We will discuss the concept of static and public modifiers later on).

Remember that this view is just a summary of the Java class. As we add/remove code from the actual Java class, this view will update accordingly.

Turn your attention back to the code editor window. Let us examine the actual Java code now. All this code has been generated for us by Eclipse as a convenience.

Notice that the code is color coded. *Keywords* are in purple, *comments* are in blue and the remaining text is in black. Also notice the use of curly braces '{' and '}' to denote where classes and methods end and begin.

__8. Look at the first line:

```
package com.simple;
```

This is the package declaration. This should always be the first line in a Java class. This line simply states that the following class belongs to the **com.simple** package.

__9. Examine the next line:

```
public class HelloWorld {
```

Here, we *declare* the class. We are saying that this is a new class that is called **HelloWorld** and that it is **public** (we will discuss **public** modifiers later).

This is followed by an opening curly brace '{'. Later, at the end of the file, this should be matched with a corresponding closing curly brace '}'. Locate this closing brace now.

Any code in between these braces will be considered belonging to this class. This shows the use of braces to indicate *scope*.

An unmatched curly brace pair will be flagged as an error and the code will refuse to compile. An important part of Java programming is learning how to properly open/close scope.

__ 10. Examine the next few lines:

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
  
}
```

This is a **method**. The first line declares the method. It is a **public** method that is **static**, and its *return type* is **void**. It also takes one argument: an array of String objects. This argument is called *args*.

Remember that **main** is a special method. Since this HelloWorld class has such a main method, it can be “run”. When the class is run, any code inside this method will be executed. Not all classes will necessarily have main methods, however.

Again, note the use of curly braces to denote where the method begins and ends. Just like the class declaration, any code that is in between these braces will be a part of this method.

Currently, the body of the method only has one line and that line is a comment that was placed there by Eclipse. A comment is a piece of **non-code text** that is placed within code as a note to the person reading the code. It can form the basis of documentation, or simply serve as helpful reminders as to what the code is doing for anyone reading the code. Comments in Java can either be preceded with a *//* sequence, or surrounded by a */** and **/* sequence. Remember that this is **non-code** which means the compiler will not attempt to process it.

Keeping this in mind, we can see that the method currently does nothing. Let us change this now.

__ 11. Using the editor, delete the line

```
// TODO Auto-generated method stub
```

__12. Replace it with the following line:

```
System.out.println("Hello World");
```

What is this line doing? This line is invoking the **println** method of **System.out** and passing the string “**Hello World**” to it. **System.out** is an object that is provided by the Java language itself and represents the “console” (the default location where Java outputs to; typically, the screen).

println is a method that is provided for **System.out** and it takes one argument: a **String object** representing what needs to be printed. (Objects will be discussed in more detail in class)

Pay special close attention to the brackets '(' and ')' as well as the trailing semi-colon. Every Java statement should end with a semi-colon.

__13. Your code is complete. Save your code by going to **File** → **Save** or by typing **Ctrl-S**.

__14. Check the *Problems* view. You may see a JRE warning, make sure there are no errors.

If you had made any code mistakes (e.g. typos, syntax errors, etc), an item would have appeared in the list.

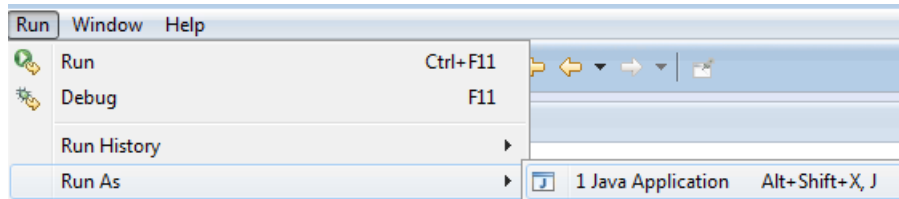
If any errors are there right now, go back and double check the code you typed in. Correct any mistakes you find. Do not proceed until there are no problems listed here.

__15. Your code is now complete. The next step is to run it.

Part 4 - Run the Code

Now that we have completed writing the code, we can run it to see the results. Running Java code implies launching an instance of the Java Virtual Machine (JVM), and then loading the appropriate Java class and executing its **main** method. Fortunately for us, Eclipse makes that simple.

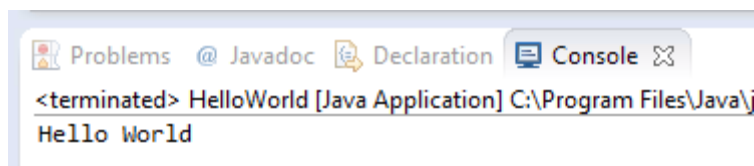
1. Click anywhere in the code editor and from the menu, select **Run | Run As | Java Application** from the menu bar.



Troubleshooting

If you don't see the option **Java Application** under **Run | Run As**, click in the code editor and try again. Alternatively, you can select HelloWorld.java in the *Package Explorer* view. As you can see, Eclipse is context sensitive.

A new view (the *Console* view) should appear in the same area where the *Problems* view is. Any text sent to the console (e.g. by a call to **System.out.println()**) will appear in this view.



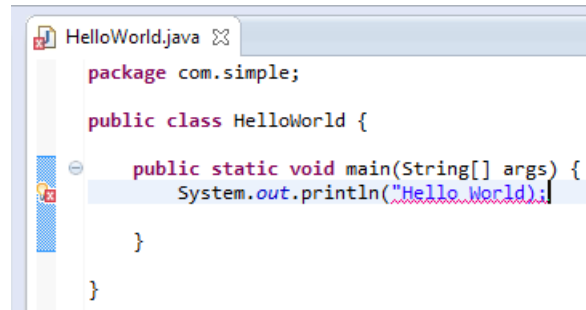
Voila! Our Java code has executed and Hello World has been printed. Congratulations. You have written and run your first Java class.

Part 5 - Change the Code

Just as an exercise, let us change the code a little.

___1. Let us now introduce an error to see what sort of error handling features that Eclipse offers. Delete the closing " mark from the **println** statement and save your code.

What happens?



```
package com.simple;

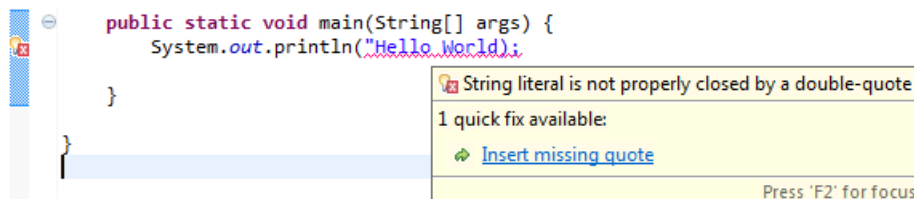
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println('Hello World');
    }
}
```

Note that a red X has appeared in the tab title (next to “HelloWorld.java”). This indicates there is an error in the class.

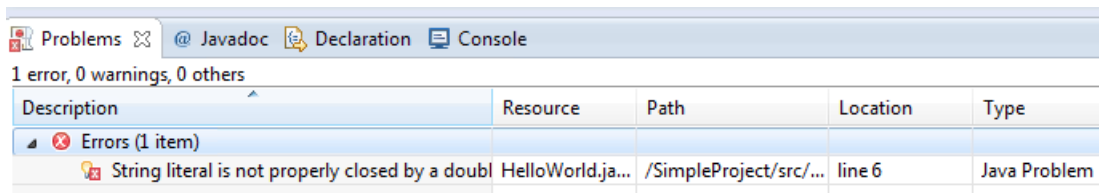
___2. Also note that in the left margin a similar red X has appeared on the same line that the error occurs on. Finally, note that the un-closed string (“Hello World); is itself underlined in red.

___3. Float the mouse cursor over the red underline.



A little pop up appears with a description of the problem. This is a hint to tell you what the corrective action should be. Do not fix it yet though!

__4. Locate the *Problems* view. Expand **Errors**.



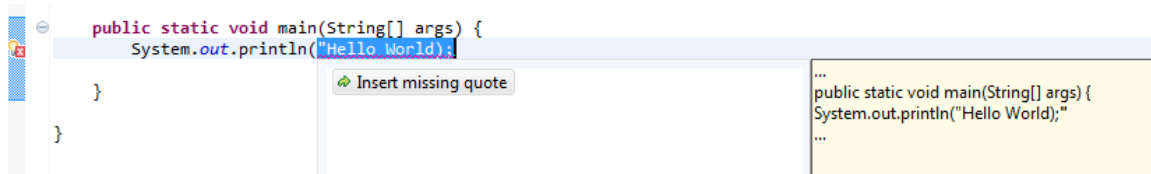
The screenshot shows the Eclipse IDE's 'Problems' view. At the top, there are tabs for 'Problems', '@ Javadoc', 'Declaration', and 'Console'. Below the tabs, it says '1 error, 0 warnings, 0 others'. A table lists the error details:

Description	Resource	Path	Location	Type
Errors (1 item)				
String literal is not properly closed by a double quote	HelloWorld.java	/SimpleProject/src/...	line 6	Java Problem

Note that the error in our code has been detected. You may have to resize the *Description* field to see the whole text. This view shows what resource (Java file) the error occurs in, and even shows the line number. This becomes invaluable if you are working on multiple Java files. Finally, if you double click on the error listed doing so will open the editor on the exact spot where the error is. Again, this is invaluable if you are working with multiple Java files.

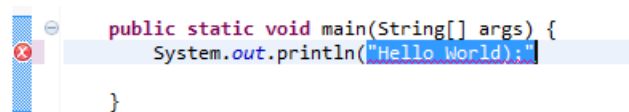
__5. Finally, click on the red X in the left margin.

__6. Two boxes pop-up. Ignore the box in yellow; it is the box in white that is interesting. Eclipse has located the error and is actually suggesting a fix!



__7. Double click where it says *Insert missing quote*

What happens?



Eclipse has indeed added a closing quote mark, but unfortunately in the *wrong* location (after the semi-colon, instead of before the closing bracket). Let us see what this “fix” does.

__ 8. Save the file.

__ 9. Look at the *Problems* view. There are now two errors! So while the fix suggested by Eclipse was on the right track, it ended up doing more harm than good.

__ 10. Fix the errors properly (by putting the quote in the right place, immediately after **World** and save the file. Make sure all the errors are gone.

From this, you should see that the error handling features of Eclipse are quite thorough; however, you should also see that it is not perfect. Additionally, Eclipse can only catch *compile time* errors (i.e. syntax errors and errors in the actual code) and not *run time* errors. The difference should become clear to you as work through the rest of these lab exercises.

__ 11. Immediately after the existing **System.out** line, add another line of code as follows.

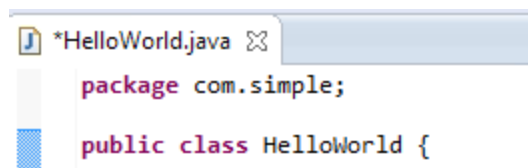
```
System.out.println("Goodbye, cruel world!");
```

If you are not particularly conscious about using tabs, spacing, and line breaks properly in your code (i.e. code style), it can end up looking a little disorganized. Fortunately, Eclipse can format code for us so it looks neatly arranged.

__ 12. Right click anywhere in the code editor and select **Source | Format**.

If your code was not “neatly” written, it will be miraculously re-formatted to look perfectly neat and organized.

Note that in the tab for the code editor (where the file name **HelloWorld.java** is displayed) an asterisk has appeared. This is to indicate that the file was changed, but has not been saved.



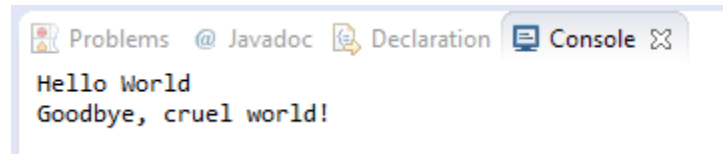
```
*HelloWorld.java
package com.simple;

public class HelloWorld {
```

__ 13. Save the file. Notice that the asterisk in the code editor tab disappears, indicating the file has been saved. This means the file is “current”. There should be no errors. It should look like the following.

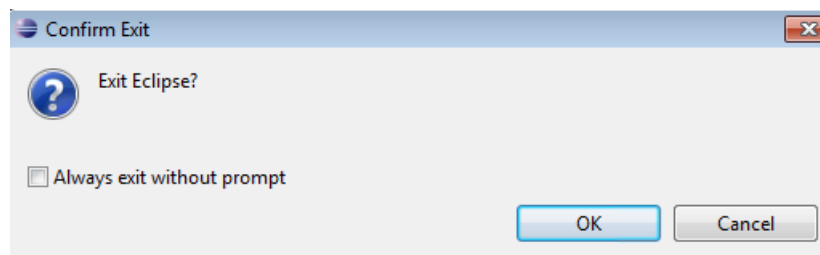
__14. Run the code again. (To do this, go to the menu bar and select **Run | Run As | Java Application.**)

The console window should show your new updated output.



__15. Close Eclipse by select **File | Exit.**

__16. Confirm by clicking OK.



Don't worry about losing the state of your project; the next time you open Eclipse, it will remember what your last state was.

Part 6 - Review

In this lab exercise, you explored Eclipse and coded your first Java class.

You saw that a Java class has a name and (usually) a package. It has methods and its method/class scope is determined by braces. You learned how to print something to the console by invoking the **System.out.println** command.

Using Eclipse , you created a Java project to contain the HelloWorld.java class. You learned about perspectives and views, and used wizards to generate some code for you. Finally, you also saw some of the error-handling features of Eclipse.

Lab 2 - Refining The HelloWorld Class

Time for Lab: 30 minutes

In this lab exercise, you will make a few changes to the HelloWorld class that you created in the last lab exercise. Specifically, you will make it a little more interactive. By using the **Scanner** class, you will actually prompt the user for a name which will then be printed back out.

By doing this, you will gain more exposure to both Eclipse and the Java language. You should start to build a better understanding of the syntax of Java.

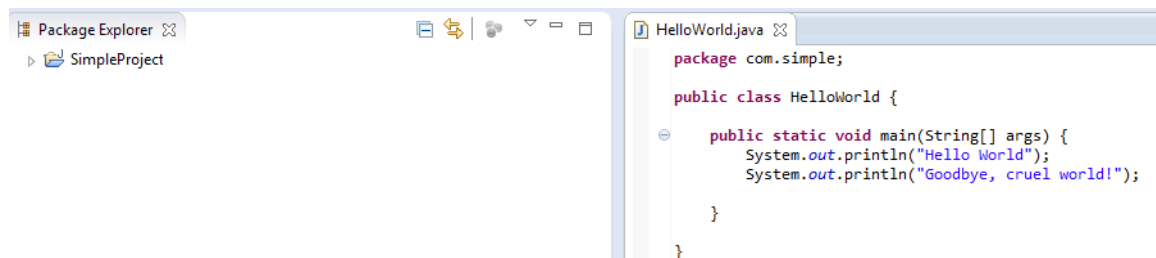
Part 1 - Edit the Class

Right now, our code gives out a generic “Hello World” greeting, which is a tad on the mundane side. We will edit the class to print out a request for a name, and then use the **Scanner** class to extract input from the user. A more personalized greeting will then be delivered to the user.

___1. You should have shut down Eclipse at the end of the last lab exercise, so now we will restart it. Launch Eclipse. (**C:\Software\eclipse\eclipse.exe**)

___2. When prompted to *Select a workspace* just click **Launch**.

When Eclipse starts up, it should look exactly like it did when you shut it down. Eclipse “remembers” the last state of the various views so it is easy to keep working where you left off.



___3. In the **HelloWorld.java** class, **remove** the lines:

```
System.out.println("Hello World");
System.out.println("Goodbye, cruel world!");
```

__4. Then **replace** it with the following code:

```
System.out.println("Hello.  Please enter your name:");
```

Nothing interesting here; we are merely changing the greeting text.

__5. Immediately after that code, enter the following:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

You will notice that as you were typing this, Eclipse will pop up a box underneath the cursor listing various class and package names; this is a convenience that Eclipse is trying to offer; specifically, it is trying to give you a *code assist* (i.e. guess what class/method you are trying to invoke, allowing you to click it instead of typing the entire thing). This can be handy, but is an advanced feature; ignore those pop up boxes for now.

What is this line doing? Well, we need to use the **Scanner** object, and doing so will require us to declare and initialize an *instance* of the **Scanner**. This is precisely what we are doing here. **Scanner** is a class provided by Java that is in the **java.util** package.

The first part of the statement (**java.util.Scanner scan**) is the *declaration* and the second part (**new java.util.Scanner(System.in)**) is the *initialization* of the object.

Note that the initialization uses the **new** keyword, which implies that we are using a *constructor* on the **Scanner** class. This creates a *new instance* of the class.

Also note that the reference to **java.util.Scanner** is a *fully qualified* reference, meaning the entire package/class name has been specified. This can be clunky, and we will see how to shorten this later in this lab.

Finally, notice that the *constructor* of the **Scanner** takes an argument which in this case is **System.in**. **System.in** represents an 'input stream' that Java can obtain input from. By default, this is the keyboard.

So, put together, this line creates a new instance of a Scanner class, and sets it up to read data from the keyboard.

__6. Immediately after that code, enter the following:

```
String name = scan.nextLine();
```

This line declares a new String variable (called `name`) and assigns it to the result of a call to `scan.nextLine()`. In effect, when the user types something into the prompt and hits Enter, the text that is entered will be placed inside the `name` variable.

__7. Enter the following line of code:

```
System.out.println("Hello, " + name);
```

This is similar to the usual `System.out.println` code we have used before, but with a twist; we are now using a variable inside the print argument.

The code

```
"Hello, " + name
```

Uses the `+` symbol as a *concatenation* tool. Basically, it combines the string `"Hello, "` with `name`. A string can be concatenated with another string, and the result is a larger string. It is this larger string that is actually sent to **`println`**.

__8. Finally, enter the following line of code which will close the input scanner:

```
scan.close();
```

__9. Our code is done! Save your code. There should be no errors. If there are, fix them before proceeding.

Part 2 - Run the Code

We can now execute our new, more interactive, HelloWorld class.

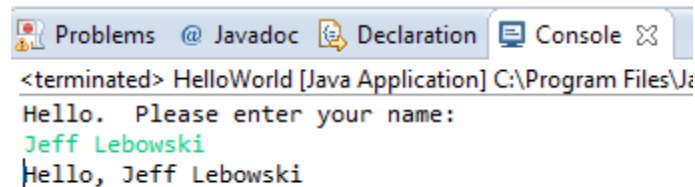
__1. Run the code as you did before. (Click anywhere on the code editor and then go to **Run | Run As | Java Application**)

Examine the console view.

```
HelloWorld [Java Application] C:\Program Files
Hello. Please enter your name:
```

This looks the same as before; our standard **println** commands are in use. However, what is different is that you can now enter text into the console.

__2. Click anywhere on the console view and type your name, followed by the Enter key. The Java class will then respond with an appropriate message.



```
<terminated> HelloWorld [Java Application] C:\Program Files\J...
Hello. Please enter your name:
Jeff Lebowski
Hello, Jeff Lebowski
```

Note that the code you type is in a different color.

__3. Run the code multiple times to make sure it works.

Congratulations! You have managed to add some interactivity to your simple HelloWorld program.

Part 3 - Import Packages

Earlier, when you entered the code to declare the **Scanner** object, you saw how the fully qualified class name was used. This is because, by default, Java does not “know” about the **Scanner** class. In order to locate and use it, the fully qualified package name had to be provided.

If you know that the **Scanner** class will be used multiple times, it may be a good idea to **import** the class. Importing a class will allow the code to use the class without having to fully qualify it. Typically, a Java program will import all the appropriate classes to simplify coding. We will do this now.

__1. In the HelloWorld.java class, add the following statement immediately after the **package** statement:

```
import java.util.Scanner;
```

__2. Now, change the line where the scanner is initialized to the following:

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

Note that we no longer need the fully qualified **Scanner** package name because it has been imported.

__3. Save the code. There should be no problems.

__4. Run the code. It should work the same as before.

Part 4 - Review

In this lab, you edited your HelloWorld class to add some interactivity. You saw the use of a constructor to initialize a new class instance, and you saw how a string can be assigned to the result of a method call.

You also saw how importing of packages can simplify coding.

Lab 3 - The Arithmetic Class

Time for Lab: 45 minutes

In this lab, you will build a slightly more complex class than the HelloWorld class; it will perform some basic arithmetic operations, and print out the results of the operations.

This lab is designed to give you yet more experience with the Java syntax. Additionally, you will see an **if** statement to handle *branching logic* for the first time.

Part 1 - Create The Arithmetic Class

Instead of using the same HelloWorld.java class, we will create an all new class file to contain our code. We will create this class in the same project (**SimpleProject**) and package (**com.simple**) as our existing class.

___ 1. In the *Package Explorer*, expand the **SimpleProject** > **src** > **com.simple** package.

___ 2. Right click on the **com.simple** package and select **New | Class**

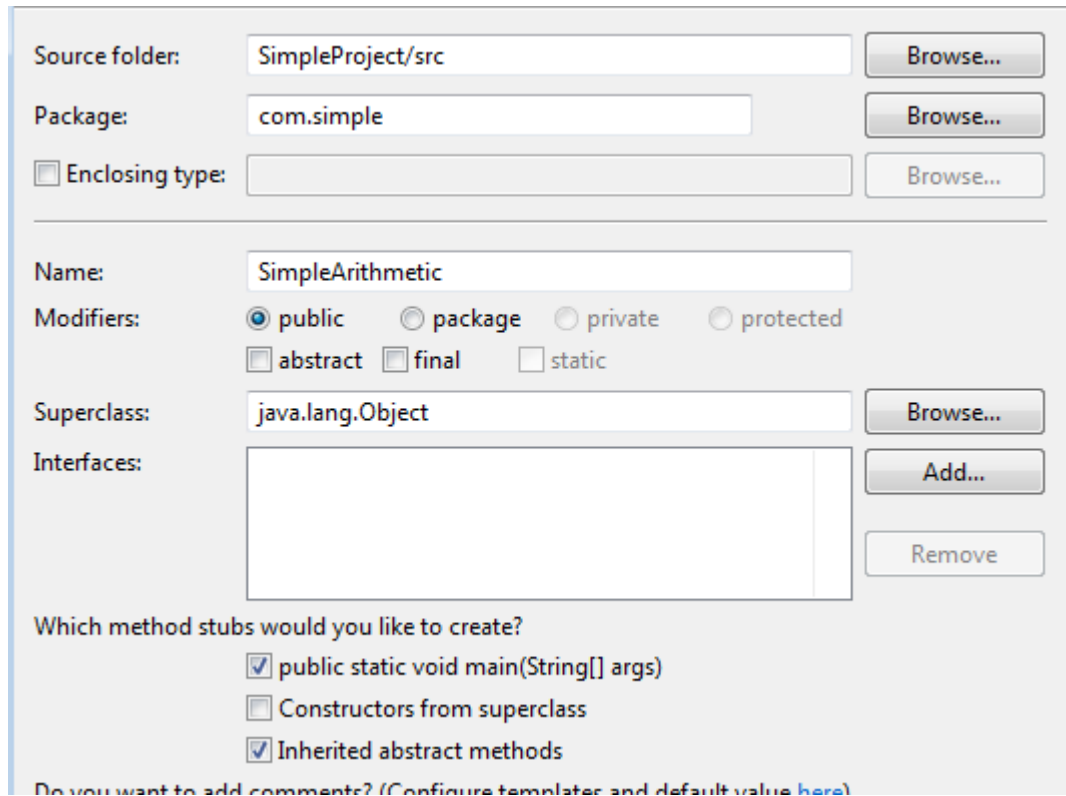
The *New Java Class* wizard will appear.

This should look familiar; we used the same wizard to create the HelloWorld class. As before, we should specify a name for our new class using this wizard.

___ 3. Enter **SimpleArithmetic** as Name.

Notice that the *Package* field is already populated with **com.simple** saving us the trouble of typing it in. This is because we right-clicked on the **com.simple** package before creating this new class.

4. Check the box marked **public static void main(String[] args)**. We will want Eclipse to generate this method for us.



The image shows the 'New Class' wizard in Eclipse. The 'Source folder' is 'SimpleProject/src', 'Package' is 'com.simple', and 'Enclosing type' is empty. The 'Name' is 'SimpleArithmetic'. Under 'Modifiers', 'public' is selected, and 'static' is checked. The 'Superclass' is 'java.lang.Object'. Under 'Which method stubs would you like to create?', 'public static void main(String[] args)' and 'Inherited abstract methods' are checked. At the bottom, there is a link to configure templates and default values.

Source folder: SimpleProject/src Browse...

Package: com.simple Browse...

☐ Enclosing type: Browse...

Name: SimpleArithmetic

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☒ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

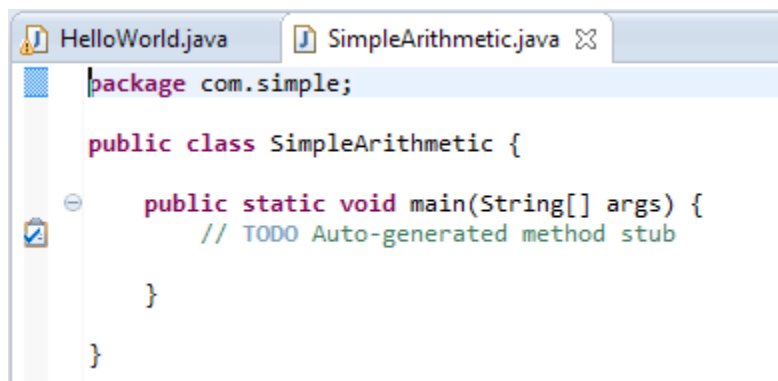
Which method stubs would you like to create?

☒ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default values [here](#))

5. Click **Finish**.

Eclipse will generate the class and open an editor on it.



The image shows the Eclipse IDE with two tabs: 'HelloWorld.java' and 'SimpleArithmetic.java'. The 'SimpleArithmetic.java' tab is active, showing the following code:

```
package com.simple;

public class SimpleArithmetic {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }

}
```

Part 2 - Edit the Class

Now we can begin coding the class. It will be very simple in design; it will first prompt the user to enter a number, followed by a second number. It will then display the results of the two numbers being added, subtracted, multiplied and divided by each other.

__1. Locate the **main** method and delete the `// TODO` comment line inside it.

__2. Let us start by prompting the user to enter two numbers. Enter the following code inside the **main** method.

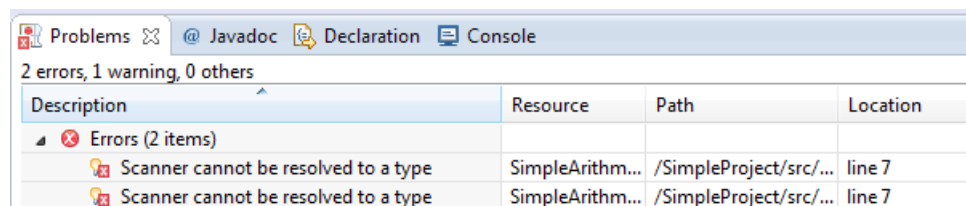
```
System.out.println("Please enter two numbers:");
```

__3. Now, set up the Scanner class (as we did in the previous lab) with the following code:

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

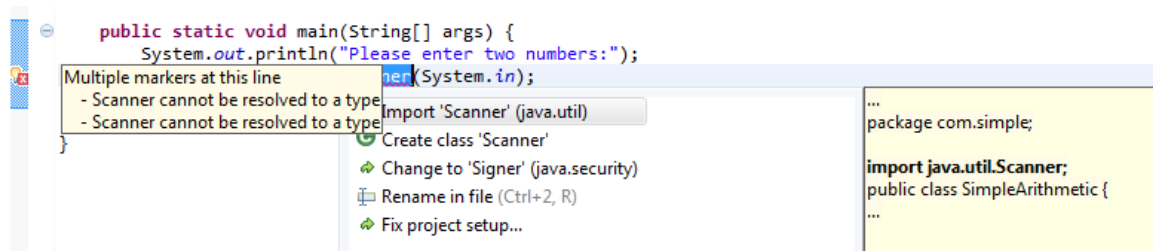
__4. Save the file.

Note that Eclipse will complain that there are errors on this line. What is the error? Look at the *Problems* view.



Eclipse does not understand what the **Scanner** class is. This is because we did not use the fully qualified name of the class (**java.util.Scanner**), nor did we import the package that the class belongs to. Fortunately, we can get Eclipse to fix this for us automatically.

__5. Click on the red X on the left margin of the line where the error is. Some pop up boxes will appear.



__6. In the white box, double-click the line **Import 'Scanner' (java.util)**.

The boxes, and errors should disappear. What happened? Take a look at the top of the source file and notice that the **import** statement has been automatically added for us by Eclipse, thus solving our problem! Eclipse is fairly good at solving these kind of **import** problems.

__7. Save the file and all the errors should go away.

__8. We can continue coding the file. Immediately following the scan initialization line, enter the following bold code:

```
Scanner scan = new Scanner(System.in);  
int x = scan.nextInt();  
int y = scan.nextInt();
```

These two lines are quite straightforward. They declare **int** variables (**x** and **y**) and assign them to the result of **scan.nextInt()** which will retrieve input from the user. After these lines are executed, **x** will be the first number that was entered, and **y** will be the second number that was executed.

__9. We can now print the results of the arithmetic operations. Continue by adding the following bold code:

```
int y = scan.nextInt();  
System.out.println("You entered " + x + " for x");  
System.out.println("You entered " + y + " for y");
```

Here we simply print out what the values entered were. Note the use of concatenation to append the **ints** to the output string.

__10. Continue by adding the following bold code.

```
System.out.println("You entered " + y + " for y");  
System.out.println("The sum of x and y is " + (x+y));
```

This seems like a simple concatenation, but notice the difference; we are adding an operation to the concatenation! The expression **(x+y)** will be evaluated first since it is in parentheses; that evaluated sum will then be concatenated to the string and **println**-ed

Note that since x and y are both **ints**, Java understands that **(x+y)** should invoke the *addition* operation.

__11. Continue with the subtraction and multiplication results.

```
System.out.println("The sum of x and y is " + (x+y));  
System.out.println("The difference of x and y is " + (x-y));  
System.out.println("The product of x and y is " + (x*y));
```

These work in the same way as before. (Note that clever use of the copy-and-paste mechanism of Eclipse can greatly speed up typing those lines...)

__12. Now, things become a little more complex. We have yet to handle division but there is a problem. What happens if the user has entered 0 for the value of y? Java will complain with a “division by 0” error. We should prevent this from ever happening and we can do that by entering the following code:

```
System.out.println("The product of x and y is " + (x*y));  
if(y == 0) {  
    System.out.println("Cannot divide by 0.");  
} else {  
    System.out.println("The division of x and y is " + (x / y));  
}
```

Here, we see a traditional **if-else** statement. The syntax can seem confusing at first (those pesky curly braces are everywhere), but is actually quite simple.

Immediately following the **if** keyword is the *test condition* (which is in parenthesis). In our case, we test to see if y is equal to 0. (Notice that the condition uses a **double equal** sign; **==** is a test for equivalence; putting a single equal sign in there would imply that we are assigning the value of y to 0!)

Immediately after the condition, we open a curly brace and put our *true* clause code in. This is the code that will be executed if the condition evaluates to true. In our case, we simply print out a message saying that division by 0 is not possible. This clause is open and terminated using the traditional curly braces { }

After the true clause, we use the keyword **else** to begin the *false* clause. The code contained within this set of curly braces will be executed in the event that the condition does **not** evaluate to true. In this case, if y is not equal to zero, we simply print out the corresponding division result.

___13. Add the following bold line to close the input scanner.

```
        System.out.println("The division of x and y is " + (x / y));
    }
    scan.close();
```

___14. We are done. Save the code and make sure there are no errors.

Part 3 - Run the Code

___1. Run the code as you have done so in previous labs. Use the console and enter two positive non-zero numbers. Make sure you get expected results! (You may need to resize the console view to see all the text at once)

```
Please enter two numbers:
45
5
You entered 45 for x
You entered 5 for y
The sum of x and y is 50
The difference of x and y is 40
The product of x and y is 225
The division of x and y is 9
```

___2. Run it again and then try entering a zero for the second number. Make sure the appropriate "Cannot divide by 0" message is displayed.

```
Please enter two numbers:
4
0
You entered 4 for x
You entered 0 for y
The sum of x and y is 4
The difference of x and y is 4
The product of x and y is 0
Cannot divide by 0.
```

Part 4 - Break the Application

1. Now, run it again and try entering non-numbers (like characters and punctuation marks). What happens?

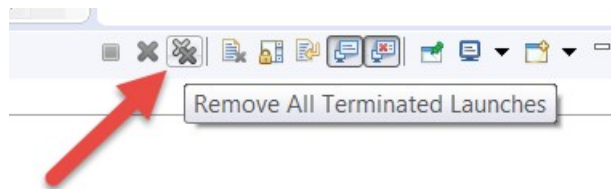
```
Please enter two numbers:  
one  
Exception in thread "main" java.util.InputMismatchException  
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)  
    at java.base/java.util.Scanner.next(Scanner.java:1594)  
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)  
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)  
    at com.simple.SimpleArithmetic.main(SimpleArithmetic.java:10)
```

A little on the ugly side. Java has encountered a problem and is throwing an *exception*. Because the **scan** was expecting an **int**, entering character data instead has caused the **scan** object to complain. The text in red is an exception – which is simply Java's way of telling you that something has gone wrong. Exceptions will be discussed in more detail later on.

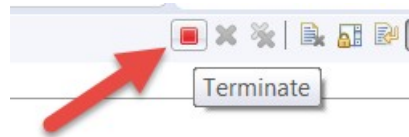
This has happened because our code does not check for all error conditions – such as entering character data instead of numeric data. Currently, our code **does** check for one error condition: division by zero.

Knowing this, you could go back to the code and write **if** statements to check to see if **x** and **y** are indeed numeric values; doing so would be good error handling practice. Roughly sketch out (but do not code) what the **if** statements might look like if you were, in fact handling these types of errors.

2. On the Console tab, click **Remove All Terminated Launches**.



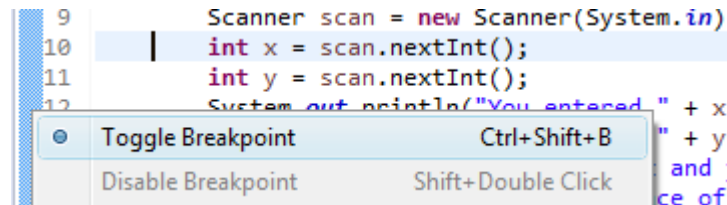
___3. Click **Terminate**.



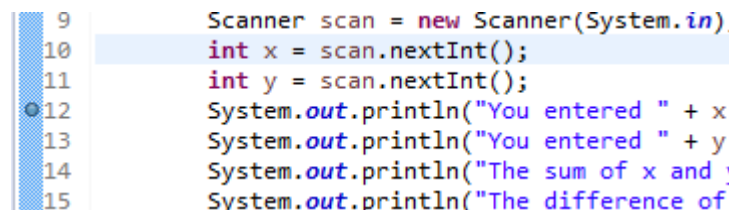
Part 5 - Debug

One useful tool of Eclipse is to run code in "debug" mode to control the execution and even change variable values. There is a special Eclipse perspective used for this which you will see in this section.

___1. In the editor margin to the left of the first 'System.out.println' statement, right click in the margin and select **Toggle Breakpoint**. You can also double click in the margin if this is easier.



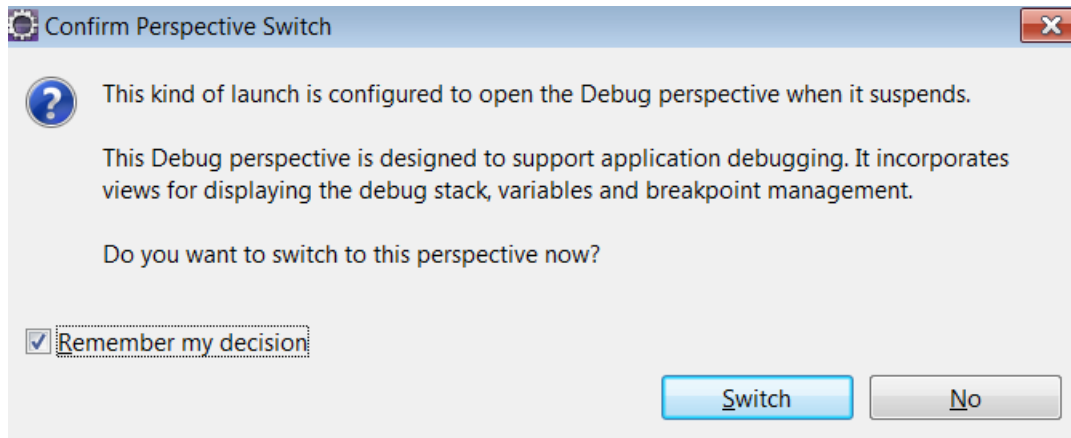
___2. Check that a new breakpoint is added which should show up as a light blue dot in the margin. Make sure it is next to the first 'System.out.println' statement and not the line above or below. If you have line numbers showing the exact line number may be slightly different.



___3. With the editor for the code still active, select '**Run → Debug As → Java Application**'. If you get a prompt about Windows Firewall, unblock the process.

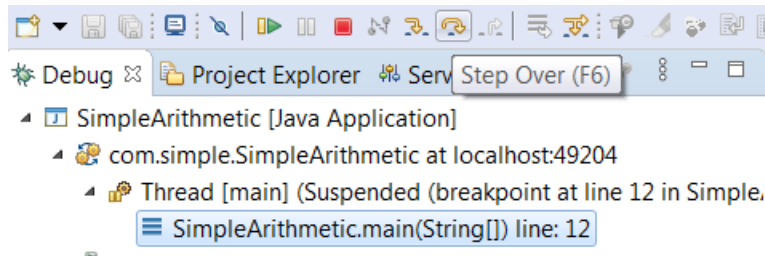
___4. You should initially see the normal output in the Console view so input two numbers with neither being zero.

__5. As soon as you enter the second number, you should get a prompt about switching perspective. Since Eclipse has reached the breakpoint it wants to switch to the *Debug* perspective. Check the box for '**Remember my decision**' and click the **Switch** button.

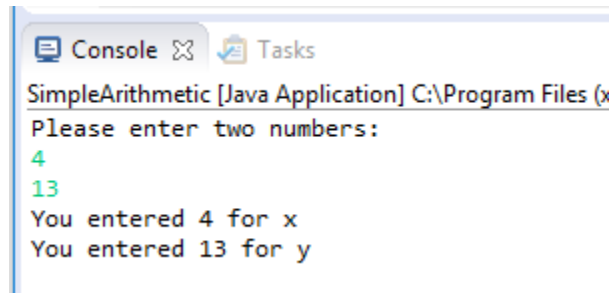


__6. In the *Debug* perspective there are a lot of things going on. You should still see the Console view on the bottom with the editor and outline view in the middle. Along the top are a few views used in debugging the code.

__7. In the *Debug* view in the upper left, make sure the 'SimpleArithmetic.main' line that is currently paused is selected and then click the '**Step Over**' button a few (2-3) times.



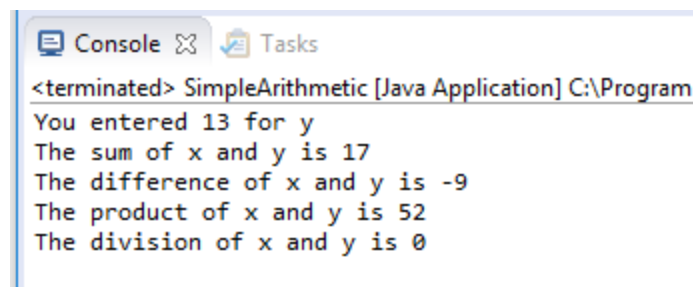
__8. Check that in the *Console* view you see a few lines printed out. This should match the number of times you pressed the '**Step Over**' button.



__9. In the top menu buttons, press the '**Resume**' button once.



__10. In the *Console* view you should see all of the output and that the program has "terminated". The '**Resume**' button goes until the next breakpoint or the program finishes.



__11. Select the code editor in the middle so it is active.

__12. Again select '**Run → Debug As → Java Application**'.

__13. Enter two non-zero numbers in the *Console* view so you will get to the breakpoint.

__14. This time, before moving past the breakpoint, notice in the *Variables* view in the upper right that the values you entered are assigned to the variables. You might need to expand the view a little to see them easily.

Name	Value
args	String[0] (id=16)
scan	Scanner (id=19)
x	3
y	17

__15. Click on the 'y' value, change it to zero, and then hit ENTER.

Name	Value
args	String[0] (id=16)
scan	Scanner (id=19)
x	3
y	0

__16. Use the 'Step Over' and 'Resume' buttons to finish the execution of the program.

__17. This time you should see in the *Console* view that even though you had entered a non-zero number for 'y', the output uses 'y' as 0. This is because the *Variables* view can change the actual variable values in the running program.

```
<terminated> SimpleArithmetic [Java Application] C:\Pro...
You entered 0 for y
The sum of x and y is 3
The difference of x and y is 3
The product of x and y is 0
Cannot divide by 0.
```

__18. Close the *Debug* perspective by selecting '**Window** → **Perspective** → **Close Perspective**'. You should be back in the *Java* perspective.

Note: The *Debug* perspective is good for all the tools you need for debugging but is tough to use for editing code. The editor is shrunk mainly just so you can see what line of code is currently being executed.

__19. Close all open files

Part 6 - Review

In this exercise, you wrote another Java class and saw more detail of how to use Java statements, including arithmetic operations. You also saw how Eclipse can assist with importing of appropriate packages. You saw how to use an **if** statement to control logical flow in Java code. Finally, you saw how to use the debugging tools of Eclipse.

Lab 4 - Project - Prompt and Store StockAccount Information (Optional)

Time for lab: 30 minutes

This lab will begin a “project” that will be improved and modified throughout the rest of the class. Each time you work on the project you will be adding a little more functionality based on the topics covered in the class. This project will represent an application that might be used by a stock broker to track investment information.

The main goal of the “project” is to allow you to become more comfortable with writing code on your own. To promote this, the instructions will not provide the actual code to write but simply requirements that you should try to implement. This will be more like a “real world” project where you must take a description of what the program should do and turn it into working code.

If the requirements do not mention a specific aspect of how the program is supposed to behave make some reasonable assumptions for how you would expect the program to behave. Variations in interpretation of the requirements can lead to very interesting solutions and discussions with the rest of the class.

One tip that will make you more successful is to save, compile, and run your code fairly often. This way you will get immediate feedback on if there is a compilation problem or if it doesn't behave as you might expect.

One thing to remember about the project is there is no single solution. There may be many ways to write code that will fulfill the requirements. You are encouraged to discuss with other students and your instructor various ways to solve the problem at hand. When reviewing the project labs, your instructor can provide some information on various approaches and ways to avoid inefficient or incorrect ways to code a solution.

You will also be provided with a “solution” that you can import. This is just one example of a way to solve the problem and is provided mainly so you have something to compare your code with and can copy code from the solution if you need it to stay current with the project. Do not assume that the way things are done in the provided “solution” is the only way to do things. Instructions will be provided at the end of this lab on how to import these “solutions”.

It is possible that you may not have time to complete all of the project labs. Your instructor must balance between giving everyone the time they need to finish the projects and being able to cover all of the intended topics for the class. You can skip the project labs without any impact on the “regular” labs as the project is completely separate. If your class does begin skipping some of the later project labs you can still work on them after the class as a good exercise to begin applying your Java knowledge.

Part 1 - Start Eclipse and Create Project

- ___ 1. If it is not already running, start Eclipse. Use the same workspace as you are using for other labs to simplify things.
- ___ 2. Be sure you are in the **Java** perspective. If needed, you can select **Window -> Open Perspective -> Other....** Then, select **Java (default)** and click **OK**.
- ___ 3. From the menu bar select **File -> New -> Project...**
- ___ 4. Expand **Java**, select **Java Project** and click **Next**.
- ___ 5. Fill in a project name. You can use any project name you want as long as it is different than existing projects. You can call the project “**StockProject**” if you want but this is only a suggested project name.
- ___ 6. Press the **Finish** button to create the project.

Part 2 - Create the StockTracker Class

- ___ 1. In the *Package Explorer* view, right click on your project and select **New -> Class**.
- ___ 2. Enter **com.stock** for the package.
- ___ 3. Enter **StockTracker** for the name of the class.
- ___ 4. Check off the box for creating a “main” method as this will be required to run the program.
- ___ 5. Press the **Finish** button to create the class.

Part 3 - General Requirements

This section will provide a general description of what the program should do at this point in the project. Other labs that have you work with the project will generally not provide the details in the last few sections but will provide only this section with the requirements and the next section which provides some sample output of a working version of the program.

It may be best to implement only a subset of the requirements at a time and test your program before attempting to do more. This will provide more rapid feedback for any problems that may be occurring before the program is too complex to track them down.

Throughout the project you can assume that the user will supply reasonable input unless you are told specifically to handle a certain type of incorrect input.

Add code to the “main” method of the StockTracker class to do the following:

- Print a few lines of output to the user to introduce the program.
- Prompt the user to enter their name for the account. Instruct them to enter their complete name as we will not track first and last name separately.

- Read in the name and store it in a String variable. Your program should be able to store the entire name and be able to handle spaces in the name.
- Prompt the user to enter the initial balance of their stock account. For now assume they are entering a valid amount for the balance.
- Read in the initial balance and store it in a variable of type double.
- Print the information the user entered about the account
- Modify the code to check that the initial balance of the account is positive. If it isn't, initialize the balance to 1000 and inform the user.

Note: If using the 'Scanner' class with 'System.in' to read in input, you may get a warning about a "resource leak" with the Scanner. The way to avoid this warning would normally be to close the Scanner but don't do that in this case. If you do it will also close system input and this may cause errors in later labs when you may read in multiple times from the input.

Part 4 - Sample Output

The following section provides one example of output you might expect from the program at this point. This is just an example as you may have implemented your program slightly differently to meet the requirements.

Remember that you will need to select the **Console** view to enter the input for the program.

(User input in bold)

```
Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.
```

```
Please enter your name and hit <ENTER>
```

```
Bob Broker
```

```
Please enter your initial account balance and hit <ENTER>
```

```
358.87
```

```
Your account details:
```

```
Name: Bob Broker
```

```
Account Balance: 358.87
```

This is some sample output showing various errors:

```
Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.
```

```
Please enter your name and hit <ENTER>
```

```
Bob Broker
```

```
Please enter your initial account balance and hit <ENTER>
```

```
-245
```

```
Negative balances are not allowed.
```

```
An account has been opened with $1000
```

```
Your account details:
```

```
Name: Bob Broker
```

```
Account Balance: 1000.0
```

Part 5 - Importing Solutions

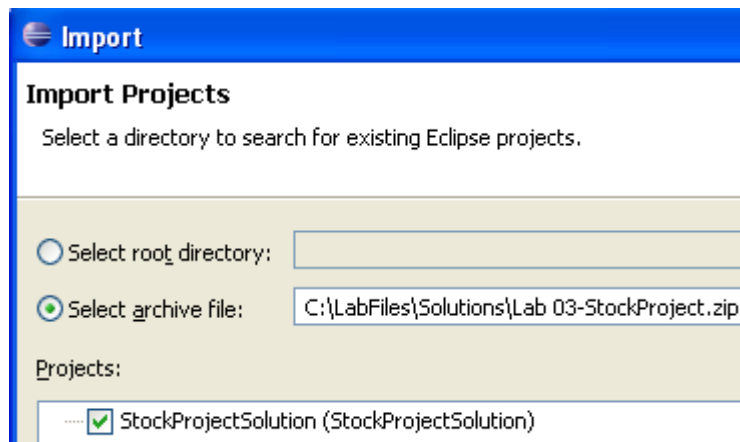
This section will provide instructions on how you can import the sample “solution”. You can follow these steps any time you want to import a solution.

Remember that the “solution” is only one way to solve the problem and it is generally best not to import the solution so it overwrites your own project. This will let you compare different ways to solve the problem.

WARNING: Importing a project in this way will delete any existing project in your workspace with the same name. It is suggested that you copy any code you want to use from the “solution” project into a separate project so importing a new solution project does not delete work you were doing.

- __ 1. Delete any project called '**StockProjectSolution**'.
- __ 2. From the **File** menu, select **File -> Import**
- __ 3. From the list of import sources, expand the **General** folder and select **Existing Projects into Workspace**
- __ 4. Press the **Next** button.
- __ 5. Select the **Select archive file** option.
- __ 6. Press the **Browse** button and find the Zip file for the solution you want to import. This should be 'C:\LabFiles\Solutions\Lab 04-StockProject.zip' and click **Open**. Ask your instructor about the location if you can't find it.

__7. Check off the checkbox next to the project you want to import.

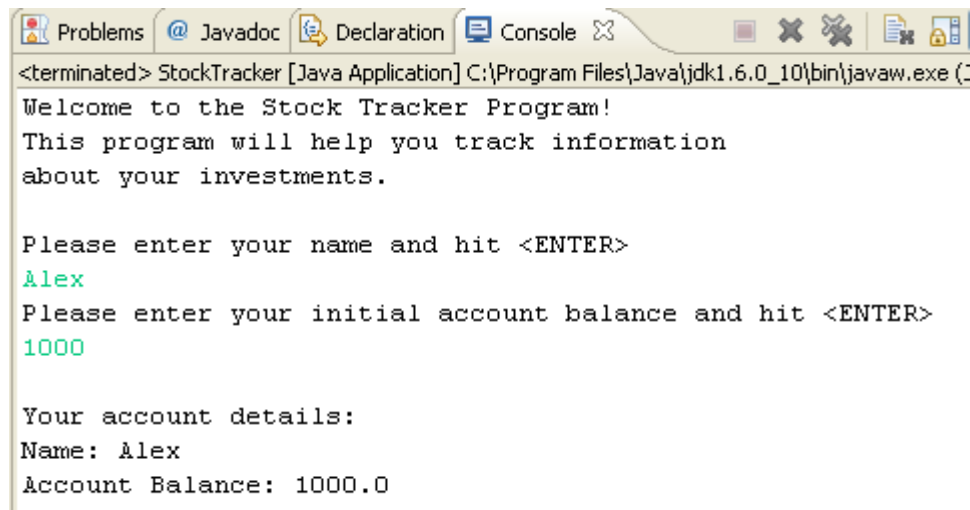


__8. Press the **Finish** button to import the project.

__9. Expand **StockProjectSolution > src > com.stock**

__10. Right click on **StockTracker.java** and select **Run As > Java Application**.

__11. The Console will show the program running. Enter your name and a balance.



Lab 5 - Creating A Simple Object

Time for Lab: 45 minutes

In this lab exercise, you will create a Java object.

Thus far, we have created several Java classes – but we have been using them in a strictly procedural manner. The classes we have created have only served to run via a single main method; in this context we have not really built a real object per se.

Recall that an object is meant to model a real life entity, such as a Person or a BankAccount. Objects have a state (which is maintained by *fields*) and operations (which are maintained by *methods*).

For example, let us say we are modeling a Person object. A Person object may have a field called **name** (of type String) and a field called **age** (of type int). Additionally, this Person object may have methods like **purchaseItem()** and **getHairCut()**.

Obviously, the fields and methods for this model would be pertinent to the overall object model we are building. If our model revolves around a Person operating a car, the **Person** would probably have fields like **driversLicenseNumber** and methods like **driveCar()**.

We will build a simple BankAccount class in this lab exercise.

Part 1 - Create A New Project

We will create an entirely new project to keep our code separate from the other code we have written thus far.

- ___ 1. From the menu bar, go to **File | New | Java Project**.
- ___ 2. The *New Java Project* wizard will appear. For the *Project name*, enter **BankAccountProject**.

__3. Select **Use default JRE 'jdk-14' and workspace compiler preferences**.

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: BankAccountProject

☒ Use default location

Location: C:\Workspace\BankAccountProject

JRE

☐ Use an execution environment JRE: JavaSE-13

☐ Use a project specific JRE: jdk-14

☒ Use default JRE 'jdk-14' and workspace compiler preferences

__4. Click **Finish**.

__5. Click **Create**.

New module-info.java

Create module-info.java

⚠ Discouraged module name. By convention, module names usually start with a lowercase letter

Module name: BankAccountProject

☐ Generate comments (configure templates and default value [here](#))

Create Don't Create

The new project should appear in the *Package Explorer*.

Our project is now created, and we can go ahead and create our class inside it.

Part 2 - Create the BankAccount Class

We will now create a very simple BankAccount class, and add some fields to it. This BankAccount object will be a rough approximation of what a real BankAccount looks like. It will be a very simple model, but will be a good example of the concepts you need to understand.

You will create a new class in the **BankAccountProject**.

__1. Right click on **BankAccountProject** in the *Package Explorer* and select **New | Class**.

The *New Java Class* wizard will appear. We have seen this a few times already, so it should start to look familiar.

__2. For the *Name*, enter **BankAccount**

__3. We will create a new package. For the *Package*, enter **com.simple.account**

__4. This class will not have a **main** method as it is strictly an domain object class, so make sure the box for *public static void main(String [] args)* is left unchecked.

Java Class
Create a new Java class.

Source folder: BankAccountProject/src Browse...

Package: com.simple.account Browse...

☐ Enclosing type: Browse...

Name: BankAccount

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

__5. Click **Finish**.

As usual, Eclipse will open an editor on the new class.

```
package com.simple.account;  
  
public class BankAccount {  
  
}
```

Note that in this editor window, there may be tabs for the other classes you have written. It does not harm anything to leave them open, but you may want to close them to avoid clutter.

__6. To close them, click on the appropriate tab and click the **X** button on the tab. If you need to open them later on, navigate to them through the *Package Explorer* and double click on them to re-open them in the editor.

The first thing we should do is add some fields to the class. This can be done by adding field declarations to the class. Remember that an object should model some real world entity, and fields are used to model an entity's state.

Fields declarations are at the *class scope*, which means they are within the curly braces that form a class, but not within any method.

__7. Add the following field declarations, after the opening '{', following the class declaration.

```
public class BankAccount {  
    public int accountID;  
    public String ownerName;  
    public float balance;  
}
```

What has happened here? We have basically specified that this class, when instantiated, will have three fields; one called `accountID` (which is an *int*), another called `ownerName` (which is a *String*) and a *float* called `balance`.

`accountID` will represent the bank account's ID number.

`ownerName` will represent the name of the owner of the bank account.

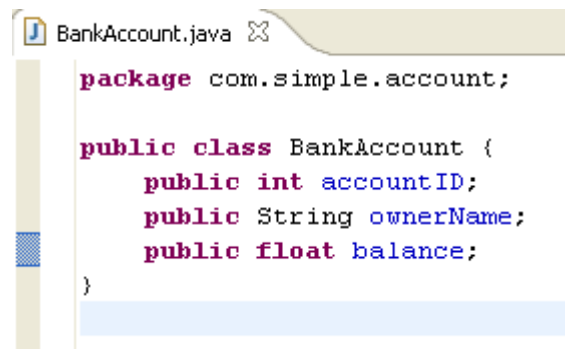
`balance` will represent how much money the bank account has left in it.

This is a fairly standard representation of a real world bank account. Note that the types (e.g. int, String, float) are sensibly chosen based on the type of data they are representing.

Note that field names should always be in lower case. (e.g. **accountID** not **AccountID**). Also notice that we have made all these fields public by using the keyword **public**. This is also known as the *access modifier*. We will discuss access modifiers in more detail later.

Note: Strictly speaking, float is not the correct type to represent a currency value, because its internal representation is binary, which leads to round-off errors when representing decimal numbers. The 'java.lang.BigDecimal' class would be correct, but its usage is more complicated than the primitive 'float'. We are choosing to keep the code simple at this stage.

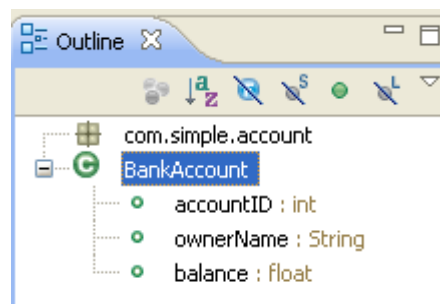
__8. Save your code. There should be no errors. It should look like this:



```
package com.simple.account;

public class BankAccount {
    public int accountID;
    public String ownerName;
    public float balance;
}
```

__9. We have now created a simple Java class that has fields. Expand the **BankAccount** class at the *Outline* view.



Notice that the outline lists the three fields and their type. We know they are fields because they are represented by hollow icons.

__10. Try clicking on one of the fields in the *Outline* view. What happens in the code editor view?

Our next step is to add an operation to the class. We will do this by adding a method. The method we will create will be a simple one; we will write a **deposit** method. This method takes a float as an argument; it should add the float to the existing balance.

__11. Enter the following code after the field declarations.

```
public void deposit(float amount) {  
    balance = balance + amount;  
}
```

The method is called **deposit**. It takes the **amount** passed in as a parameter, and then the body of the method adds it to the **balance** field. Note that we refer to the **balance** field as if it were a local variable.

__12. Save the code. There should be no errors. Your code should look like the following:

```
package com.simple.account;  
  
public class BankAccount {  
    public int accountID;  
    public String ownerName;  
    public float balance;  
  
    public void deposit(float amount) {  
        balance = balance + amount;  
    }  
}
```

__13. Examine the *Outline* view. Notice that the method has been added to it. What happens when you click on it?

Part 3 - Create the Tester Class

Since our **BankAccount** class is complete, we now want to test it. Specifically, we want to write some Java code to create an instance of the **BankAccount** and test its fields and method.

But where do we do this? Recall that all Java execution starts from a **main** method somewhere. Our **BankAccount** class, however, does not have a **main** method. We could add one – but doing so would sort of break the idea of a **BankAccount** modeling a real life entity. After all, when was the last time you had a bank account with a **main** operation? A part of object oriented programming is devising good models.

A better approach would be to build a separate class that is designed specifically to test our **BankAccount** object. We could put a **main** method in this tester class and in that main method we could write code to test the **BankAccount** object. This is exactly the approach we will take.

- ___1. Create a new class by right clicking on **BankAccountProject** and select **New | Class**.
- ___2. Set the *Package* to be **com.simple.test** and set the *Name* to be **BankAccountTester**
- ___3. Check the box for **public static void main(String [] args)**.

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?
☒ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

- ___4. Click **Finish**.

The code editor should open on the new class.

__5. Remove the usual comment from the **main** method.

__6. Now, let us begin coding. In the **main** method, add the following code:

```
BankAccount account = new BankAccount();
```

We declare a new local variable called **account** and call the constructor to initialize it, much as we have done so before.

You have probably noticed that Eclipse is saying there is an error on the line of code you just entered. Assuming you have not made any typos, this is because of the packaging issue. The `BankAccount` class is in the **com.simple.account** package, so it must either be referenced by fully qualified name, or imported. We'll take the easy route and import the package.

__7. An easy way to do this is to click anywhere on the code editor and type

Ctrl-Shift-O. Do this now.

Notice that by doing this, the correct **import** statement is automatically added to the source. The command **Ctrl-Shift-O** is used to *organize imports* and is a very handy feature of Eclipse. Use it when you are unsure of what packages are needed.

__8. Save the file. There should be no errors.

We have now created code to create an instance of the `BankAccount` class, giving us a `BankAccount` instance. Let us now perform some interesting code on it.

__9. Immediately following the instantiation code, add the following statement:

```
account.accountID = 1;
```

This line takes that instance of the `BankAccount` and sets its field `accountID` to the value 1. This means we are explicitly making the value of this data field to be 1. This instance will now remember that state.

Note that we use the dot (.) operator to access a field on the class. We use that access and the = sign to set the value of the field. This is known as *direct method access*.

__10. Set the state of the other two fields by adding the following two lines.

```
account.ownerName = "Jeff Lebowski";  
account.balance = 100f;
```

This is fairly straightforward, except for the 'f' at the end of the 100. This is done because the **balance** field is a **float** and must be assigned to a **float**. Adding the 'f' at the end of 100 tells Java to use the **float** equivalent of 100.

We have now created an instance of a BankAccount and set its fields to some sample data. Let us now try an operation on it. We should now test to see if it worked. This can be done by using the standard println methods. Specifically, we will **println** out the values of the fields to see they were set properly.

__11. Add the following code immediately after the setting of the balance.

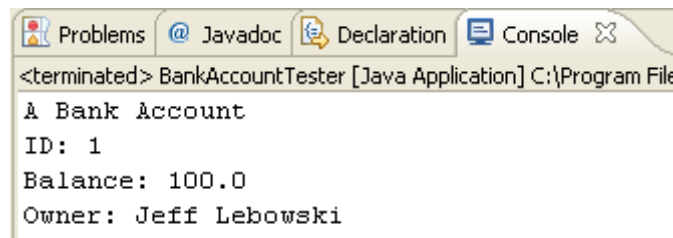
```
System.out.println("A Bank Account");  
System.out.println("ID: " + account.accountID);  
System.out.println("Balance: " + account.balance);  
System.out.println("Owner: " + account.ownerName);
```

This code is quite simple. It simply uses the dot (.) operator to access the fields of the **account** instance so they can be sent to **println**.

__12. Save the code.

__13. Run this code as a Java Application by selecting from the menu, **Run > Run As > Java Application**.

What do you see?

The screenshot shows the Eclipse IDE's console window. At the top, there are tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active. The text in the console reads: '<terminated> BankAccountTester [Java Application] C:\Program File'. Below this, the output of the program is displayed: 'A Bank Account', 'ID: 1', 'Balance: 100.0', and 'Owner: Jeff Lebowski'.

As expected, the BankAccount's fields contain the correct data and were printed out successfully. It looks like our object works!

__14. Now, let us try invoking an operation on the class. Recall that we have written one method: the **deposit** method. We will use it now. Immediately before the first `println` statements, insert the following:

```
account.deposit(50f);
```

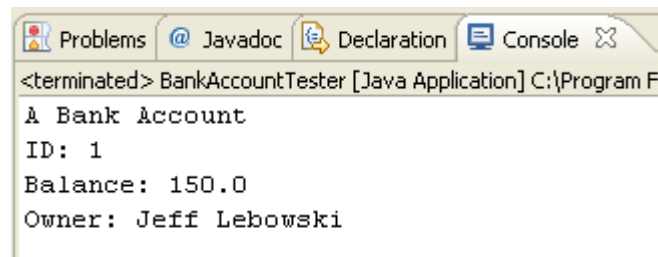
__15. Save the file.

We invoke the method on our `account` by using the dot (`.`) operator, much like we accessed the fields. However, when invoking a method, the brackets (`(` and `)`) are present and they often contain parameters. In this case, we invoked the method with the parameter `50f`. *By invoking the method, we tell Java to go to the class that we invoking the method on, and execute the method with the same name.* In our case, Java looks up the `BankAccount` class, locates the `deposit` method and executes the code there.

What should this code do? Well, if you look back at the source code for the `BankAccount` class, you will see that the code takes the passed in parameter and adds it to the current balance.

So, following this logic, if we run the code again, we should now see an updated balance of `150.0`.

__16. Test this by running the class.



```
<terminated> BankAccountTester [Java Application] C:\Program Fi
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
```

As expected, the balance has been correctly updated.

Congratulations! You have successfully written and tested your first Java object!

Part 4 - Review

In this exercise, you create a Java object and gave it a state (via fields) and operations (via methods). You saw how to define a field and a method. You then wrote a tester class with a main method that created an instance of and then operated on the Java object.

Although this all seems quite straightforward, we actually made quite a few *bad* coding decisions. We will address this in the next exercise.

Lab 6 - Project - Create a StockAccount Class (Optional)

Time for lab: 45 minutes

This lab will continue the project. Currently the StockTracker class uses many different variables to store the stock account information. In this lab you will define a separate class definition to define what properties a StockAccount object would have. The StockTracker will then work with this StockAccount class instead of tracking every variable separately.

Part 1 - General Requirements

- Create a new class in the com.stock package. Call the new class 'StockAccount'. This class will not have a 'main' method.
- Add instance variables for the various properties a StockAccount will have. This includes the name on the account and the current balance. This data is currently tracked separately in variables from the StockTracker class.
- Modify the 'main' method of the StockTracker class to create a new StockAccount object before prompting the user for information about the stock account.
- Modify the rest of the StockTracker code to use the instance variables of the StockAccount object instead of using separate variables for all of the data.
- At the end print the information about the account again, this time using the instance variables of the StockAccount object instead of the local variables.

Part 2 - Sample Output

The output for this part of the project is basically the same as you are simply storing the stock account details in an object instead of individual variables.

Part 3 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- How easy is it to rewrite the StockTracker class once the StockAccount class is introduced?
- How does introducing the StockAccount class change how the StockTracker class prints details about the account?
- Are there other properties (besides buying and selling stocks which will be done later) that would make sense in a StockAccount class?

Lab 7 - Getters and Setters

Time for Lab: 30 minutes

In the previous exercise, you created the BankAccount class which consisted of 3 fields, and a method. You then tested the class by creating an instance of it and settings its three fields to some values by *direct method access*.

In this exercise, we will remove direct method access, and instead use proper *getter and setter* methods.

Part 1 - Problems of Direct Method Access

Direct method access is the use of an object instance's fields by using the dot (.) operator. While there is technically nothing wrong with this (Java fully allows this), it is not good objected oriented design.

Remember that an object is supposed to hide its implementation details from any other class that uses it. A public face (or *application programming interface - API*) to the object should be made available for other classes to use, and it is that public API that should operate on the implementation. By allowing another piece of code to access the fields directly, we are allowing access to the internal representation of the class. This is not good encapsulation.

Additionally, think of potential error situation. Currently, what mechanism is in place to prevent the balance from being set to a negative number? By using direct access, the code:

```
account.balance = -140f;
```

Could be executed. This is a violation of business rules. Ideally, when attempting to set this value, the BankAccount class should raise some sort of error and prevent the actual set from happening. This sort of error handling cannot be done with direct field access.

So, the idea here is that we want to prevent the direct access of fields. Instead, we would like to have some sort of way of accessing the fields that could be made public, and those public ways could have error checking code inside them.

This can be done with the use of *accessors*, also known as *get/set* methods, or simply *getters and setters*.

A *get* method is a public method that retrieves a field value and a *set* method is a public method that sets it. These methods are typically public, while the fields they touch are made private.

In order to change the value of a field on a class, the public *accessors* should be used. The fields themselves would be made **private**, thus preventing direct method access.

Let us make these changes now.

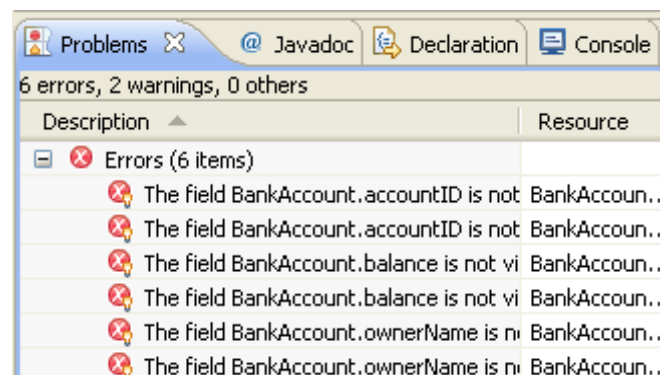
__1. Open the **BankAccount.java** class.

The first thing we need to do is disable direct method access for the fields. We can do this by changing the *access modifier* for the fields. The access modifier specifies what other classes can touch these fields. Right now, the modifier for the fields is **public** which means any other class can – including our tester class. We should change this.

__2. Change the modifiers to **private** as shown below.

```
private int accountID;  
private String ownerName;  
private float balance;
```

__3. Save the file. There should be no errors in the BankAccount class. However, if you look at the *Problems* view, something has appeared.



What is going on? Problems have appeared in the BankAccountTester class. But is that what we expect? Examine the error message.

The field BankAccount.accountID is not visible

Actually, it is. Our BankAccountTester class is still trying to use direct method access – but is not able to do so, since we made the fields **private**. So, the errors are expected – and even wanted. Direct method access has now been disallowed!

__4. Our next step is to begin coding the *accessor* methods. Add the following method, after the closing curly brace } of the **deposit** method.

```
public void setAccountID(int newID) {  
    this.accountID = newID;  
}
```

__5. Save the file.

Examine this method. It is called **setAccountID** and it takes a parameter of type **int** called **newID**. What does it do? Simple. It assigns the field called **accountID** (of **this** object instance) to the passed in parameter. It has a **void** return type – because it does not return anything.

This more or less has the same effect of the direct method access that we were using before, except now it is being done from within the class, and not an external class. It is now an implementation detail.

Since the method is **public**, however, other classes can now use this method. Let's try this.

__6. Switch back to BankAccountTester.java. There should be six errors on the class. Locate the line:

```
account.accountID = 1;
```

__7. Then change it to:

```
account.setAccountID(1);
```

Instead of using direct method access, we use the newly created **set** method. The error should go away in this line!

__8. Now, let us write the corresponding *get* method. Switch back to BankAccount.java and add the following method, after the previous **set** method:

```
public int getAccountID() {  
    return this.accountID;  
}
```

This method is quite simple. It is public, but it now has a *return type*. This means the method will return an **int** when it is executed. What is it that gets returned? Simple. The **accountID** field. This is done in the method body. (Notice, again, the use of the **this** keyword).

__9. Save the file.

__10. We have now written a *get* method. We should change our BankAccountTester class to use it. Switch back to the BankAccountTester class and locate the line:

```
System.out.println("ID: " + account.accountID);
```

__11. Change it to the following:

```
System.out.println("ID: " + account.getAccountID());
```

__12. Save the file.

One error should go away. Previously, the line was using direct access method to *get* the **accountID**. Now, it is using the public **getAccountID()** method.

We have now written the *get* and *set* methods for the **accountID** field. The next step is to write *get* and *set* methods for the other fields. We will do the **ownerName** next.

__13. Switch back to BankAccount.java and add the following two methods to the class:

```
public void setOwnerName(String newName) {  
    this.ownerName = newName;  
}  
  
public String getOwnerName() {  
    return this.ownerName;  
}
```

These work almost identically to the *get* and *set* methods for the **accountID**, except they operate on the **String** field **ownerName**.

__14. Save the file.

There should be no errors in this class.

__15. Switch back to BankAccountTester. We will change it to use the new get and set methods.

__16. Locate the line:

```
account.ownerName = "Jeff Lebowski";
```

__17. Change it to:

```
account.setOwnerName("Jeff Lebowski");
```

__18. Now, locate the line:

```
System.out.println("Owner: " + account.ownerName);
```

__19. Change it to:

```
System.out.println("Owner: " + account.getOwnerName());
```

__20. Save the file. Two more errors should go away.

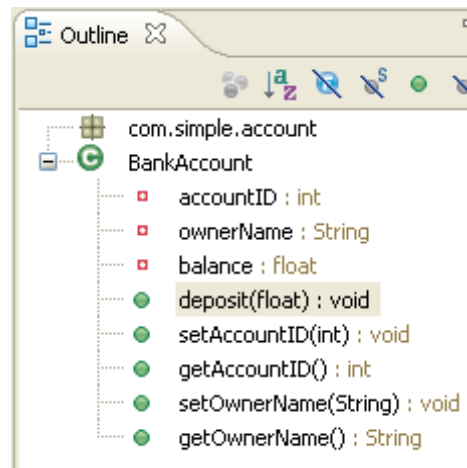
You have now written and used two get/set methods!

Part 2 - Generating Accessors

We have now seen that we create get/set methods for the fields on our class. In general, it is a good idea to create these accessors for every field that needs to be publicly accessed. However, creating them can be monotonous; we've created two, and have to create one more. Imagine a complex class with 10 fields, all of which need accessors!

Fortunately, since the code behind an accessor is quite straightforward, we can actually have Eclipse generate the code for us, saving us the trouble of writing them. We will do this now.

__1. Switch back to BankAccount.java, and examine the *Outline* view.

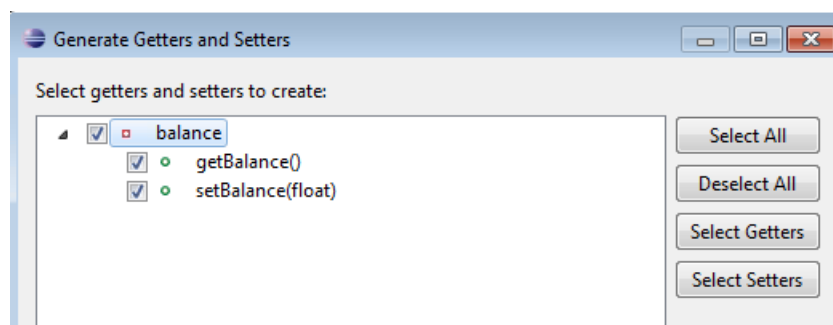


If you have not been paying attention to this view, you would not have noticed the changes. The fields on the class are now represented with a red square, denoting a **private** field. The remaining methods are all public methods, as denoted by the green circle.

We have written get/set methods for the **accountID** and **ownerName**. We just have to handle the balance. We will get Eclipse to do this for us.

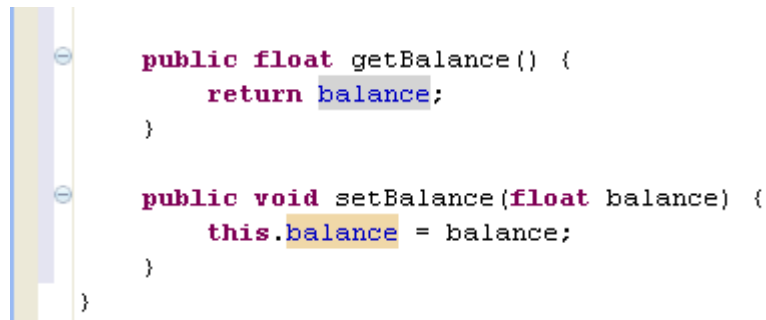
__2. Right click on **balance: float** here in the *Outline* view. Then select **Source | Generate Getters and Setters**.

The *Generate Getters and Setters* wizard will appear.



__3. Notice that it has located our **balance** field and is preparing to generate the get and set methods for it. Click **Generate**.

If you examine your source code, you should now see the new methods. That was easy! (Note that we could have generated all the accessors this way)



```
public float getBalance() {  
    return balance;  
}  
  
public void setBalance(float balance) {  
    this.balance = balance;  
}  
}
```

Notice that it named the methods much in the way we named the other ones: a **get** or a **set** with followed by the field name, with the lettering in “camel casing” style (e.g. **getOwnerName()**, **setBalance()**, etc) This is a general naming convention that all Java programmers should follow.

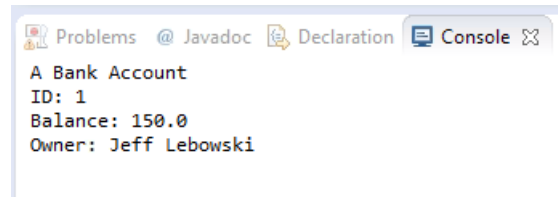
__4. Save the file. There should be no errors in this file.

__5. Switch back to the **BankAccountTester.java** and change the code to use the new getter and setter methods for the balance.

When you are finished, the code should look like this:

```
package com.simple.test;  
  
import com.simple.account.BankAccount;  
  
public class BankAccountTester {  
  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount();  
        account.setAccountID(1);  
        account.setOwnerName("Jeff Lebowski");  
        account.setBalance(100f);  
        account.deposit(50f);  
  
        System.out.println("A Bank Account");  
        System.out.println("ID: " + account.getAccountID());  
        System.out.println("Balance: " + account.getBalance());  
        System.out.println("Owner: " + account.getOwnerName());  
    }  
}
```

- __6. Save the file. All the errors should be gone.
- __7. Run BankAccountTester. It should look the same as before.



While we have not actually changed any of the logic of the code, we have changed it to use better object oriented programming techniques.

Part 3 - Accessor Error Handling

Earlier, we discussed how a potential use for accessors was that of error handling. Specifically, we stated that by using direct method access, there was no way of ensuring that a negative value would not be passed into the balance field. Now that we are using accessors, however, such a mechanism is available.

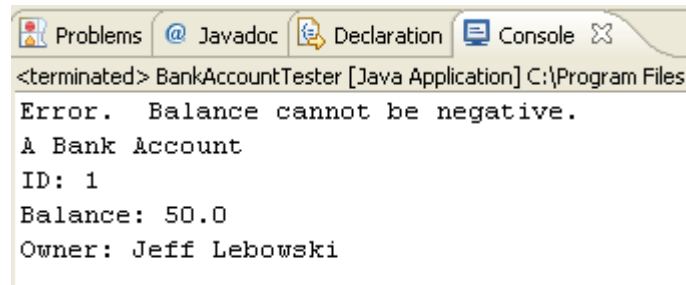
- __1. Open **BankAccount.java**.
- __2. Change the code for the **setBalance(float balance)** method to the following:

```
public void setBalance(float balance) {  
    if(balance > 0f) {  
        this.balance = balance;  
    } else {  
        System.out.println("Error. Balance cannot be  
negative.");  
    }  
}
```

What is happening here? Simple; the method first checks to see if the balance is larger than 0; if it is, then it is valid and the field is set. Otherwise, an error message is printed out. (Note that using `System.out.println` is a bad way of error handling; a better way would be to use the **exception** framework, which we will discuss later in the course)

- __3. Save the file. There should be no error.
- __4. Switch back to the **BankAccountTester.java**.
- __5. Let's test this. Change the **setBalance()** amount to be **-100f** and save.

__6. Now run the code. What do you see?



```
<terminated> BankAccountTester [Java Application] C:\Program Files
Error.  Balance cannot be negative.
A Bank Account
ID: 1
Balance: 50.0
Owner: Jeff Lebowski
```

We see an appropriate error message displayed. Our error handling code has worked!

However, below in the console, we see the balance is still 50.0. Can you explain why that is? (Hint: floats default to a value of 0)

__7. Change the value of the **setBalance** back to the original amount (100f) and save the file.

```
account.setBalance(100f);
```

__8. Run the code again. It should show again 150 as Balance.

Part 4 - Review

In this lab exercise, you modified your BankAccount class to use better OO techniques. Specifically, you made the fields private, and then wrote public accessors for them. You then updated the BankAccountTester to use the new accessors – again, better OO programming style.

In writing the accessors, you saw they provide a public interface to what should otherwise be a private implementation, and you saw that one advantage over direct field access is the ability to put in error handling code on set time.

Finally, you saw that writing accessors can be very tedious; fortunately, Eclipse can generate them for you.

Lab 8 - Project - Improve Encapsulation (Optional)

Time for lab: 15 minutes

This lab will continue the project. Currently the StockTracker class accesses the instance variables of the StockAccount class directly. This is not good practice and the StockAccount class will be rewritten to prevent this direct access and require that any classes use the “getter” and “setter” methods for these properties.

Part 1 - General Requirements

- Make the instance variables of the StockAccount class 'private' and create 'get/set' methods to access and modify the values. Note that when you do this you will get compilation errors in the StockTracker class until it is fixed.
- Fix the StockTracker class to use the 'get/set' methods to access or modify the values of the StockAccount object rather than directly accessing the instance variables. You won't be able to run the code until all the compilation errors are fixed.

Part 2 - Sample Output

The output should be the same as previous versions of the project. The changes in this part of the project are all internal.

Part 3 - Extract Methods

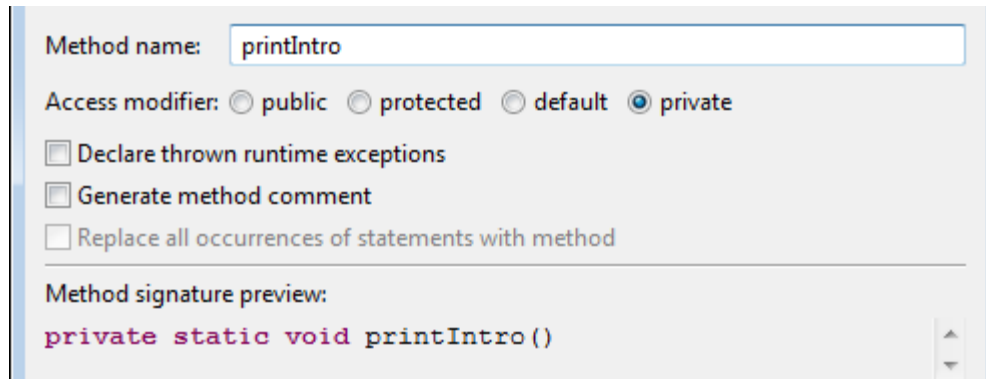
Right now the code in the 'main' method of the StockTracker class continues to get more complex. Before adding other features it would be good to find ways to simplify it before continuing with other parts of the project. One of the ways to do this is to use tools in Eclipse to "refactor" code which can include replacing lines of code with calls to a newly defined method.

__1. In your StockTracker code select the lines of code that print out the introduction to the program. Make sure you select the entire block of statements and not any partial statements or you will not see the options shown next.

```
public static void main(String[] args) {  
    System.out.println("This program will help you track information");  
    System.out.println("about your investments.");  
    System.out.println();  
  
    Scanner input = new Scanner(System.in);
```

__2. With the code above selected choose the **Refactor** → **Extract Method...** menu option.

___3. In the dialog that appears, change the name of the new method to 'printIntro' as shown below and click the **OK** button.



___4. You should see a new private method added that encapsulates the selected code that had been in the 'main' method. The code in the 'main' method was replaced with a call to this method.

```
private static void printIntro() {  
    System.out.println("This program will help you track  
    System.out.println("about your investments.");  
    System.out.println();  
}
```

___5. In your StockTracker code select the lines of code that print out the details of the account. Again make sure to select complete statements.

```
System.out.println();  
System.out.println("Your account details:");  
System.out.println("Name: " + account.getName());  
System.out.println("Account Balance: " + account.getBalance());  
  
}
```

___6. With the code above selected choose the **Refactor** → **Extract Method...** menu option.

7. Change the name of the new method to 'printAccountSummary'. Notice there will also be a method parameter because the method will need to have access to the account to print the summary.

Method name:

Access modifier: ☐ public ☐ protected ☐ default ☒ private

Parameters:

Type	Name
StockAccount	account

☐ Declare thrown runtime exceptions
☐ Generate method comment
☐ Replace all occurrences of statements with method

Method signature preview:
`private static void printAccountSummary (StockAccount account)`

8. Press the **Preview>** button.

9. Look at the summary of changes to be made and press the **OK** button.

Changes to be performed

- StockTracker.java - StockProjectSolution/src/com/stock
 - StockTracker
 - main(String[])
 - Substitute statement(s) with call to printAccountSummary
 - Create new method 'printAccountSummary' from selected statement(s)

StockTracker.java

Original Source	Refactored Source
<pre>} System.out.println(); System.out.println("Your ac System.out.println("Name: " System.out.println("Account</pre>	<pre>} printAccountSummary(acc } private static void printAc System.out.println();</pre>

Note: Even though you could uncheck some of the boxes to prevent the changes from being made you generally do not want to do that. If you disable some of the changes you may be introducing compilation errors. If you do not like the changes that will be made it is better to go back to the previous screen and see if you can change some of the options or cancel and select a different block of code to extract as a method.

__10. Notice the 'main' method has been rewritten to call the new method for printing the account summary and pass in the account object as the parameter to the method. This method is now something we can use anytime you want to print out the basic info for the account.

```
        printAccountSummary(account);  
  
    }  
  
    private static void printAccountSummary(StockAccount account) {  
        System.out.println();  
        System.out.println("Your account details:");  
        System.out.println("Name: " + account.getName());  
        System.out.println("Account Balance: " + account.getBalance());  
    }
```

__11. In the 'main' method, select ALL of the code between the calls to the 'printIntro' and 'printAccountInfo' methods. The code you have to prompt for the account info may be slightly different but you should still select all of the code between the calls to the other two methods.

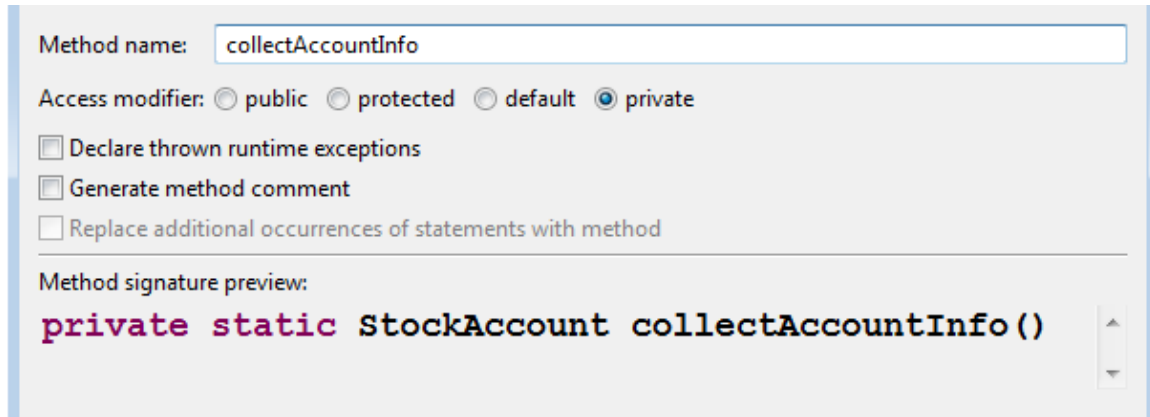
```
        printIntro();
```

```
        Scanner input = new Scanner(System.in);  
        StockAccount account = new StockAccount();  
  
        System.out.println("Please enter your name:");  
        account.setName(input.nextLine());  
  
        System.out.println("Please enter your balance:");  
        account.setBalance(input.nextDouble());  
  
        if (account.getBalance() < 0) {  
            account.setBalance(1000);  
            System.out.println("Negative balance reset to 1000");  
            System.out.println("An account has been created");  
        }
```

```
        printAccountSummary(account);
```

__12. With the code above selected choose the **Refactor** → **Extract Method...** menu option.

__13. Change the name of the new method to 'collectAccountInfo'. Notice that there should not be any parameters but the method should return a StockAccount object. If your method to be created doesn't have this signature contact your instructor as you may need to move code around in the 'main' method before extracting the new method.



Method name:

Access modifier: ☐ public ☐ protected ☐ default ☒ private

☐ Declare thrown runtime exceptions

☐ Generate method comment

☐ Replace additional occurrences of statements with method

Method signature preview:

```
private static StockAccount collectAccountInfo()
```

__14. Click the **OK** button to create the new method and replace the code in the 'main' method with a call to it. Notice how much simpler the code in the 'main' method is.

```
public static void main(String[] args) {  
    printIntro();  
  
    StockAccount account = collectAccountInfo();  
  
    printAccountSummary(account);  
}
```

__15. Save all files and make sure there are no compilation errors. The goal of refactoring is to modify the structure of the code without introducing errors.

Note: In the future you might decide to use this tool again to extract additional code into separate methods but always think carefully about what statements would make sense to extract into a method. The "Preview" option will let you decide before proceeding because the changes introduced might be fairly complex and difficult to "undo".

Part 4 - Discussion Questions

- Why did the new method to prompt for account info return a StockAccount object?

Lab 9 - Using Constructors

Time for Lab: 30 minutes

In the last lab, you took the `BankAccount` class and refined it to be more in line with good Object Oriented programming conventions by making the fields private and adding accessors.

In this exercise, we will continue with our `BankAccount` class, and examine other Java features such as constructors and class members.

Part 1 - Creating Another Instance Of The `BankAccount`

___1. Open `BankAccountTester.java`.

Recall that in this code, we are only using one instance a `BankAccount`; we know this because we only declare a single instance (called **account**) and *initialize* it once.

Let us add some code to create another instance of a `BankAccount`, this one called **account2**.

___2. Add the following code inside the **main** method, after the **println** statements.

```
BankAccount account2 = new BankAccount();
```

___3. Now, set some data on that instance.

```
account2.setAccountID(2);  
account2.setOwnerName("Bunny Lebowski");  
account2.setBalance(5000f);
```

___4. Immediately following this, create yet another instance of the account, and set its data as follows:

```
BankAccount account3 = new BankAccount();  
account3.setAccountID(3);  
account3.setOwnerName("Walter Sobcheck");  
account3.setBalance(1000000f);
```

___5. Save the code. There should be no errors.

The pattern here is all the same. We declare and initialize an new instance, and then set its fields using the set methods.

The initialization step (via the **new**) keyword is implicitly calling a constructor on the class. The constructor is essentially a special method that is invoked when the **new** keyword is used.

Without knowing it, we have been using the constructor on our BankAccount class. This may seem strange, because we never created one. This is because, for every Java class we create, Java itself *implicitly* gives us a constructor 'for free'. This 'free' constructor is the one that has been used whenever we were initializing our class.

If it is not adequate for us, we can always create our own constructor or constructors. Why would we do this? Look at the previous code we just entered.

Every time we create a new instance of a BankAccount, we have to explicitly set the **accountID**, the **ownerName** and the **balance**, using the set methods. Four lines of code; one for the initialization, and 3 for the *sets*. The *sets* are necessary because we do not want a BankAccount instance to exist without having the data in place; after all, how much sense would it make for an account to not have an **accountID**, or a **balance** or an **ownerName**?

Keeping this in mind, we can create a new constructor. We will make this new constructor require that the **accountID**, the **balance** and the **ownerName** be passed in as parameters. This means that a BankAccount will always have some appropriate data set on it after construction.

Let us do this now.

Part 2 - Create The New Constructor

- __1. Open the **BankAccount.java** class.
- __2. Add the following method right after the field declarations.

```
public BankAccount(int accountID, String ownerName, float balance) {  
    super();  
    this.accountID = accountID;  
    this.ownerName = ownerName;  
    setBalance(balance);  
}
```

Even though this looks like a regular Java method, this is actually a constructor. How do we know this? First of all, there is no return type for the method. Secondly, the name of the method is simply the name of the class.

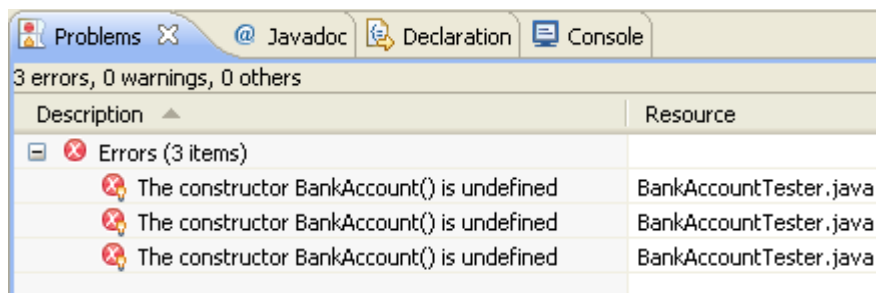
How does this method work? Firstly, note that it takes 3 arguments; an int, a String and a float. Coincidentally, these are the same types as the three fields on the class.

Secondly, the code makes a call to the **superclass's** constructor. Superclasses will be discussed in a later chapter, so you can ignore this for now.

Finally, the method merely sets the three fields on the class to the three arguments passed in. The balance is set using the 'setBalance' method since there is validation logic in this method.

Note the use of the **this** keyword here. In this context, **this** is used to separate the field (**this.accountID**) from the passed-in parameter (**accountID**). Imagine how the code would look if **this** was not used.

3. Save the code. There should be no errors on the **BankAccount** class. However, examining the *Problems* view shows something:



The same error appears in the BankAccountTester class. What is happening? Look at the error. It is complaining that the constructor BankAccount() is undefined. Recall that BankAccount() was the “free” constructor that Java was supplying for us, and we were using it. There is one error for account, account2 and account 3.

However, since we have written our own 3 argument constructor, *Java has removed the “free” constructor* so it can no longer be used. **If you create your own constructor for a class, the “free” constructor is no longer available.**

What now? Simple. We change our code to use our new constructor. Our new constructor is more practical than the old default 'free' constructor. Let us see it.

__ 4. Open the **BankAccountTester.java** class.

__ 5. Delete the following code:

```
BankAccount account = new BankAccount();
account.setAccountID(1);
account.setOwnerName("Jeff Lebowski");
account.setBalance(100f);
```

__ 6. Replace it with the much simpler:

```
BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);
```

The new code uses our new 3 argument constructor. We pass in the 3 parameters that we would have otherwise set using the setter methods, and we know what the constructor code will do with them. Four lines of code have now become one line.

__ 7. Use this constructor elsewhere in the code. Delete this code:

```
BankAccount account2 = new BankAccount();
account2.setAccountID(2);
account2.setOwnerName("Bunny Lebowski");
account2.setBalance(5000f);
```

__ 8. Replace it with:

```
BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
```

__ 9. Finally, delete this code:

```
BankAccount account3 = new BankAccount();
account3.setAccountID(3);
account3.setOwnerName("Walter Sobcheck");
account3.setBalance(1000000f);
```

__ 10. Replace it with this code:

```
BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);
```

__11. Save the file. Your code has now been greatly streamlined. It should look like the following:

```
package com.simple.test;

import com.simple.account.BankAccount;

public class BankAccountTester {

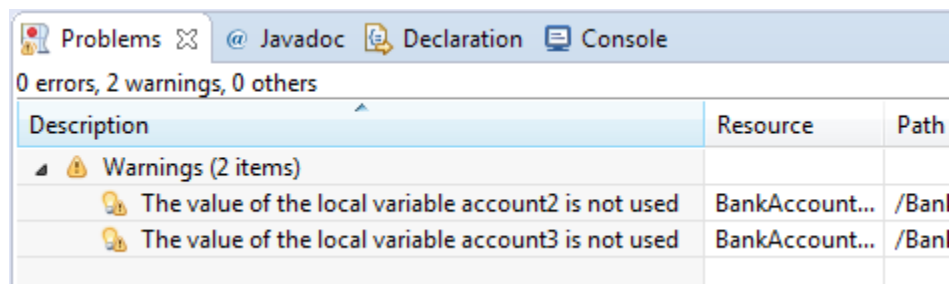
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);

        System.out.println("A Bank Account");
        account.deposit(50f);
        System.out.println("ID: " + account.getAccountID());
        System.out.println("Balance: " + account.getBalance());
        System.out.println("Owner: " + account.getOwnerName());

        BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
        BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);
    }
}
```

__12. Run the code. It shouldn't look any different, but it is definitely cleaner in terms of style.

__13. If you look in the *Problems* view, you will see there are some warnings.



The screenshot shows the Eclipse IDE's Problems view. At the top, it says "0 errors, 2 warnings, 0 others". Below this is a table with three columns: "Description", "Resource", and "Path". The table contains two rows of warnings, both with a yellow warning icon. The first row's description is "The value of the local variable account2 is not used", with resource "BankAccount..." and path "/Banl". The second row's description is "The value of the local variable account3 is not used", with resource "BankAccount..." and path "/Banl".

Description	Resource	Path
⚠ The value of the local variable account2 is not used	BankAccount...	/Banl
⚠ The value of the local variable account3 is not used	BankAccount...	/Banl

These are normal. eclipse is giving us a hint that we've declared these variables **account2** and **account3** but we do not use them for anything. This is a possible waste of memory. We will use them later and so can ignore the warnings for now.

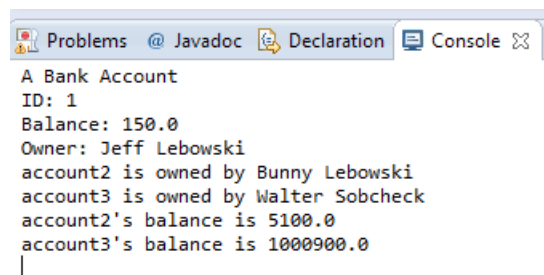
Part 3 - Object Instances

Let us now play around a bit with the various instances.

1. Go back to the **BankAccountTester.java** class and add some `println` statements following the construction of **account2** and **account3**, like the following:

```
System.out.println("account2 is owned by " + account2.getOwnerName());
System.out.println("account3 is owned by " + account3.getOwnerName());
account2.deposit(100f);
account3.deposit(900f);
System.out.println("account2's balance is " + account2.getBalance());
System.out.println("account3's balance is " + account3.getBalance());
```

2. Save the file and run the code.



```
Problems @ Javadoc Declaration Console
A Bank Account
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
account2 is owned by Bunny Lebowski
account3 is owned by Walter Sobcheck
account2's balance is 5100.0
account3's balance is 1000900.0
```

Nothing really interesting here; we are just working with the various instances. You should be seeing here that each instance is considered unique. Changing the fields (e.g. depositing money) into **account2** does not affect the fields of **account3**.

Part 4 - Static Fields

So far, we have seen fields on a class; the **BankAccount** class had fields **ownerName**, **balance** and **accountID**. Those were *member fields* (also known as instance variables), meaning each instance can have a unique value. As you have seen in your **BankAccountTester** class, we created three instances of a **BankAccount**, each one having its own **accountID**, **ownerName** and **balance**.

What if, however, we wanted a variable that was the same for **all** instances of an object? For example, all **BankAccounts** may have different balances, ownerNames and accountIDs, but they all might have the same *interest rate*.

We could create a field called **interestRate** on the class as we have done so before, and treat it as a normal field. Let us do this now.

- ___ 1. Open the **BankAccount.java** class.
- ___ 2. Add a new field after the other field declarations as follows:

```
private float interestRate;
```

___ 3. Generate accessors for this class as you did in the previous lab. (This was done by right clicking on the new **interestRate** field in the outline window and selecting **Source | Generate Getters and Setters** and clicking **Generate** in the Generate Getters and Setters window.

- ___ 4. Save the file. There should be no errors.
- ___ 5. Go back to the **BankAccountTester.java**.
- ___ 6. Let's try out these fields. We will set the field on account2. Locate the following line:

```
BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
```

- ___ 7. Immediately after, insert the following line:

```
account2.setInterestRate(5.5f);
```

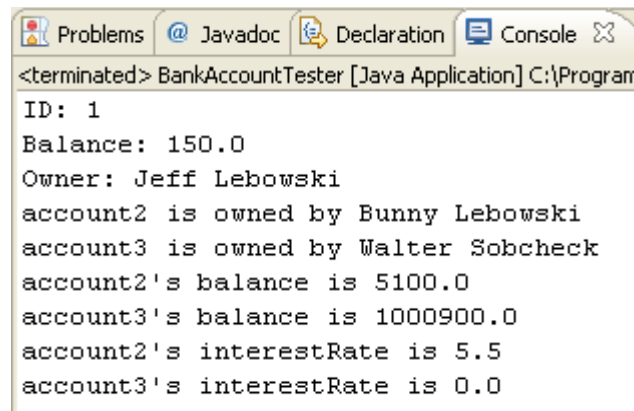
We are simply setting the field value on the instance **account2**.

- ___ 8. Now, at the end of the method, after the println of account3's balance, add the following 2 lines of code.

```
System.out.println("account2's interestRate is " +  
account2.getInterestRate());  
System.out.println("account3's interestRate is " +  
account3.getInterestRate());
```

- ___ 9. You may want to format the source code after typing these lines. See Lab 1 if you do not remember how to do this.
- ___ 10. Save the file and run the class.

What should happen here? The printlns should show the interest rate for both instances. What do you expect to see?



```
<terminated> BankAccountTester [Java Application] C:\Program
ID: 1
Balance: 150.0
Owner: Jeff Lebowski
account2 is owned by Bunny Lebowski
account3 is owned by Walter Sobcheck
account2's balance is 5100.0
account3's balance is 1000900.0
account2's interestRate is 5.5
account3's interestRate is 0.0
```

We see two interest rates ... which is what we expected to see. The interest rate for **account2** is 5.5 (as we stipulated in our code), but the interest rate for **account3** is 0.0. The 0.0 is because that is the *default* value for a float (i.e. the value that is assigned if no other value is assigned).

This is nothing new. We have already seen that two different instances will have two different values. If, however, we wanted the interest rate for **account2** and **account3** (and even **account1**) to be the same (as it should be), we would have to call the **setInterestRate()** method on all three instances. Having to do this would be highly impractical.

So a better approach would be to use a *static field*. A static field (also known as a *class field*) is a field that is the same for *all* instances of the class. Think of it as a field on the actual *class* object as opposed to the *instance* of the class. A field can be made static by using the **static** keyword. We will do this now.

__11. In **BankAccount.java**, locate the field definition for **interestRate** and change it to the following:

```
private static float interestRate;
```

We add the keyword **static** here.

__12. Also change the **setInterestRate** method and change it as follows:

```
public static void setInterestRate(float interestRate) {  
    BankAccount.interestRate = interestRate;  
}
```

Note: This sets the static field and also modifies the method so it is declared as static.

__13. Modify the '**getInterestRate**' method so it is declared static also.

```
public static float getInterestRate() {  
    return interestRate;  
}
```

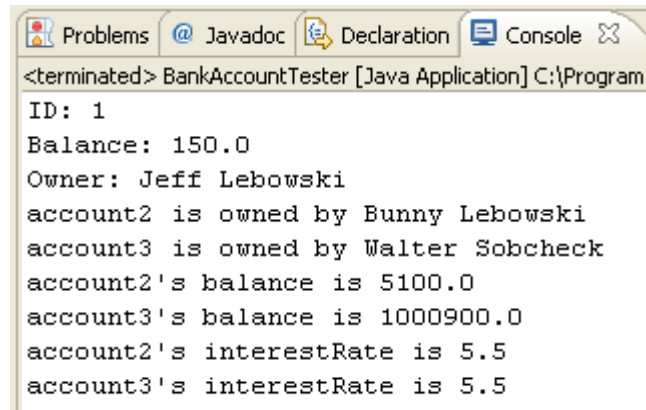
__14. Save the class.

__15. Go back to the **BankAccountTester.java** class and modify all of the places where the interest rate is used to use the '**BankAccount**' class instead of one of the instance objects. You should have code like:

```
BankAccount.setInterestRate(5.5f) ;  
... interestRate is " + BankAccount.getInterestRate() ;
```

__16. Save the class.

__17. Now, let us test this. Run the code again. What do you see?



The screenshot shows a Java IDE console window with the following tabs: Problems, Javadoc, Declaration, and Console. The console output is as follows:

```
<terminated> BankAccountTester [Java Application] C:\Program I  
ID: 1  
Balance: 150.0  
Owner: Jeff Lebowski  
account2 is owned by Bunny Lebowski  
account3 is owned by Walter Sobcheck  
account2's balance is 5100.0  
account3's balance is 1000900.0  
account2's interestRate is 5.5  
account3's interestRate is 5.5
```

Notice that the interestRates are now the same! By making the field static, we have made it the same for all instances of the class. You could now go and change the value (by calling **setInterestRate()**) on any of the instances (account1, account2 or account3) and all 3 instances would share the same value.

Congratulations! You have created your first static field.

Part 5 - Review

In this lab, you saw how Java provides a default zero-argument constructor for every class. You then created your own constructor which took parameters and used that in place of the default constructor. This new constructor made it easier to create and initialize new BankAccount objects.

Additionally, you saw how a **static** field could be used to maintain states across all instances of a class.

Lab 10 - Project - Add Constructors (Optional)

Time for lab: 15 minutes

This lab will continue the project. Modifying the state of the StockAccount object is currently fairly difficult. This is mainly because there are not good constructors to use to create the object. This lab will add constructors to the class to make this easier.

Part 1 - General Requirements

- Add a constructor to the StockAccount class that takes the name and initial balance as parameters. You may get some compiler errors when you do this until you implement the next few requirements.
- Add a constructor to the StockAccount class that takes only the name for the account. Set the initial balance to \$1000.
- Change the code of the StockTracker class to use the new constructors of the StockAccount class. Rather than constructing a StockAccount object and then setting each property separately, call the constructor that will create the object with the initial state from the user. This should simplify the code. You should be able to compile and run the code after making these changes.
- In the early versions of the project the StockTracker class would set the balance of the account to 1000 if the user input a negative balance. It is better for the StockAccount class to decide what the default balance should be if no balance is provided. Modify the StockTracker class to call the constructor with only the 'name' parameter in this case and have that constructor initialize the balance to 1000.
- **Bonus:** Define the default value the account balance should be when not provided as input to the constructor by defining a constant in the StockAccount class. Remember declaring a constant uses the 'static' and 'final' keywords in addition to the type and value of the constant. Names of constants are also often all capitals by convention.

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- You will have to use some temporary variables in the StockTracker class even if you removed these in earlier project labs. These will mainly be to hold values read in from the user before calling the appropriate constructor of the StockAccount class. Now that you need to pass the initial value for the account name and balance to the constructor you need somewhere to store these temporarily until that point.
- Once the StockTracker is no longer deciding that the default balance should be \$1000 it will be harder to print out the message to the user that a default balance has been used. The best way to do this is to construct the StockAccount object using the constructor which only takes the name and then print out a message which uses the balance of the new account as the value of what the balance was initialized to.
- When you have code that is deciding which constructor to call based on whether the input balance was negative you will have an if/else statement. Since you have a separate method that is returning the StockAccount object after calling the constructor you can simply have 'return' statements inside the if/else blocks to return the appropriately constructed object.

Part 3 - Sample Output

The output should be the same as previous versions of the project. The changes in this part of the project are all internal.

Part 4 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- It is possible to create constructors of just about all possible combinations of instance variables. Which constructors would not make sense in the scope of this project?
- Where should the decision about what the default initial balance of the StockAccount class be made? This is the balance that is used when the user attempts to open an account with a negative balance. What must be done to prevent inconsistencies if the value of this default initial balance changes?

Lab 11 - Project – Create a Stock Class (Optional)

Time for lab: 15 minutes

This lab will continue the project. Right now the project is actually a little boring. Even though we promised a program that will track stock purchases we have only implemented function to collect information about the account and spit that info back out. This was done on purpose to keep the project simple while important concepts like writing classes, fields and get/set methods, constructors, etc were introduced. Now that you know about some of these basic concepts we can make the project more interesting and implement it in an object oriented way.

Part 1 - Design Discussion

Before deciding what to create it might be good to discuss with the class or think about on your own some of the following questions. This will help bring up decisions that need to be considered when deciding how to design a class to represent the data. Even though the requirements section will give you some direction about what assumptions to make and how to proceed thinking about the questions below will help understand how other implementations might have been chosen under different assumptions.

- How are stocks identified? What kind of Java data type would make sense for this?
- When you buy and own stock what indicates how much you have? What data types might work for this?
- What other data is required to figure out what the total cost of buying the stock will be?
- If this data is going to be stored by fields in a class somewhere would it make sense in StockAccount, a new class, multiple new classes?
- If the data is stored in a new class (which means a separate object type) how would we relate this class to StockAccount to indicate that the account "owns" the stock after it is bought?

Part 2 - General Requirements

Although the previous section had some pretty open-ended questions, this section will provide some assumptions and guidance on what class definition to create and data it should contain.

- Create a new class called Stock in the same package as the other classes.
- Since a stock is identified by the stock symbol add a field to the Stock class for this. Since the stock symbol is more than one character a String should be the type used.
- The amount of stock you own is indicated by the number of shares. In this case we are going to make the assumption that you can only buy whole shares of stock so the type for this new field in the Stock class should be an 'int' type.
- We are assuming that this program is tracking your portfolio and is not actually buying and selling stock directly from the stock market. Because of this the assumption is that you will always know the price of the stock when you buy or sell stock and this should be tracked for the stock also. Since this is a monetary value create a new field in the Stock class of type double for the price of the Stock.
- Generate get/set methods for the three new fields defined above.
- Define a Stock constructor that has three parameters for the properties of a Stock and initializes the fields of the Stock class with the values from the parameters.
- In the StockAccount class, add a new field of the Stock type to store a link to the stock held by the account. For right now we are only going to own one stock to keep it simple until later when you see ways to store multiple values for data.
- Add a 'getStock' method to the StockAccount class for the new Stock field but NOT a 'set' method.

Part 3 - Sample Output

There is no new output as the class defined in this part of the project is not yet used in the rest of the program. The next few parts will actually add the features to buy and sell stock.

Part 4 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- Why did we decide to create a new Stock class instead of simply adding these new properties to the StockAccount class?
- Why did we add a field of the Stock type to the StockAccount class? What relationship does this create between the StockAccount data and the Stock data?
- Why did we create a 'getStock' method but not a 'set' method? If the other parts of the program can't 'set' the Stock field directly how will this value change? (Think ahead to the operations we will implement next)
- Since we didn't have you change the constructors for the StockAccount class to account for the new Stock field what will the field be initialized to (remember it is an object reference)? What would having a StockAccount that doesn't yet refer to a valid Stock object imply?

Lab 12 - Project – Buy Stock (Optional)

Time for lab: 45 minutes

This lab will continue the project. You will add the ability to buy a stock. This will include prompting for the stock information and checking if the account has enough money to buy the stock.

Part 1 - General Requirements

Modify the code to do the following:

- In the 'main' method of the StockTracker class, after the call to print out the account info, add a new line that prints a message to the user that they will now be buying stock.
- Add a new method to the StockTracker class that will collect stock info from the user and return a Stock object initialized with that information. This method should take no parameters and return a Stock object. You can use whatever name you think is good but something like 'promptStockInfo' might be appropriate. Within the body of the method you should prompt the user for the stock symbol, the "whole" number of shares, the price, and call the constructor of the Stock class to create the object to return from the method. Make sure this code compiles before moving on.
- In the 'main' method of StockTracker, declare a local variable of the Stock type and initialize the value to what is returned from calling the new method to prompt for stock info. This should be a similar pattern to how you declared and initialized the StockAccount variable.
- In the StockAccount class, create a new method called 'buyStock'. This should return 'void' and take a Stock object as a parameter. For right now leave the method empty as the next bullets will describe the requirements for the method.
- In the new 'buyStock' method of the StockAccount class check if buying the stock would cost more than the balance of the account. If it will inform the user they can't buy that much of the stock and don't alter the balance.
- If the purchase is allowed, subtract the amount of the purchase from the balance. Also set the field in the StockAccount that stores a reference to a Stock to be the Stock object passed into the 'buyStock' method. Storing a reference to the Stock will associate the Stock object with the StockAccount after the method completes.

- In the 'buyStock' method print the details of the purchase if the purchase is allowed.
- From the 'main' method of the StockTracker class, use the variable that refers to the StockAccount to call the 'buyStock' method and pass in the Stock object that was created earlier as the parameter to pass in.
- In the 'main' method, after calling the 'buyStock' method call the method to print the details of the account again. At this point if you run the program you will just see the balance of the account decrease.
- Alter the StockTracker method that prints the account details to get the Stock object from the account and print out details of the stock that is owned. You will have to test if the account returns a valid Stock object to the method that prints the account details since the account could not yet refer to a stock purchase.
- **Bonus:** If the user attempts to buy more shares than they can afford buy as many (whole) shares as they can afford with the current balance instead of rejecting the entire purchase.

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- It is best to create a new Scanner object from System.in at the start of the 'promptStockInfo' method which collects info about the stock to buy.
- You may want to create some temporary variables in the 'buyStock' method for calculations related to buying the stock.
- Since it is possible to have a StockAccount object that does not yet have any Stock you will need to check for 'null' and output appropriate details if that is the case. If you don't do this you will likely get 'NullPointerException's.

Part 3 - Sample Output

The following is an example of normal output:

```
Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.
```

```
Please enter your name and hit <ENTER>
```

```
Bob Broker
```

```
Please enter your initial account balance and hit <ENTER>
```

```
3000
```

```
Your account details:
```

```
Name: Bob Broker
```

```
Account Balance: 3000.0
```

```
You do not own any stock.
```

```
You will now buy stock for your account
```

```
Please enter the stock symbol and hit <ENTER>
```

```
IBM
```

```
Please enter the number of (whole) shares and hit <ENTER>
```

```
26
```

```
Please enter the price of the stock and hit <ENTER>
```

```
64.25
```

```
You have bought 26 shares of IBM at $64.25 per share
```

```
Your account details:
```

```
Name: Bob Broker
```

```
Account Balance: 1329.5
```

```
You own 26 shares of IBM
```

This is some sample output showing various errors:

```
Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.
```

```
Please enter your name and hit <ENTER>
```

```
Bob Broker
```

```
Please enter your initial account balance and hit <ENTER>
```

```
-245
```

```
Negative balances are not allowed.
```

```
An account has been opened with $1000.0
```

Your account details:
Name: Bob Broker
Account Balance: 1000.0
You do not own any stock.

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

100

Please enter the price of the stock and hit <ENTER>

50

You can't buy that much stock.

Your account details:
Name: Bob Broker
Account Balance: 1000.0
You do not own any stock.

Part 4 - Discussion Questions

- Can you see any future use of the method that is used to prompt for Stock info and construct an object from this info?
- Why is it important that the StockAccount class refer to the Stock that is bought if the purchase is allowed? How would the method that prints the account info behave if it didn't?
- What is the implication of passing the Stock object as a parameter to the 'buyStock' method called on the account object?

Lab 13 - Project – Sell Stock (Optional)

Time for lab: 45 minutes

This lab will continue the project. You will add the ability to sell a stock. This will include prompting for the stock information and checking if the stock is the same as the stock held.

Part 1 - General Requirements

Modify the code to do the following:

- In the 'main' method of the StockTracker class, after the last call to print out the account info, add a new line that prints a message to the user that they will now be selling stock.
- In the 'main' method of StockTracker, declare a second local variable of the Stock type and initialize the value to what is returned from calling the method to prompt for stock info.
- In the StockAccount class, create a new method called 'sellStock'. This should return 'void' and take a Stock object as a parameter. For right now leave the method empty as the next bullets will describe the requirements for the method.
- In the new 'sellStock' method check that the symbol of the stock being sold matches the symbol of the stock owned. If it doesn't inform the user they don't own that stock and return from the method.
- If they are selling the same stock symbol as what they own check that they are not trying to sell more shares than they own. If they are print an error and return from the method without selling any stock.
- If the sale is allowed, adjust the account balance to add the money gained from selling the stock and adjust the number of shares owned. Also adjust the price of the stock still owned to match the latest sale price.
- In the 'sellStock' method print the details of the sale if the sale is allowed.
- From the 'main' method of the StockTracker class, use the variable that refers to the StockAccount to call the 'sellStock' method and pass in the Stock object that was created earlier as the parameter to pass in.

- In the 'main' method, after calling the 'sellStock' method call the method to print the details of the account again.
- **Bonus:** If the user is attempting to sell more than they own simply sell off all of what they own instead of ignoring the sale completely.

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- Since it is possible to have a StockAccount object that does not yet have any Stock you will need to check for 'null' before you attempt to access the stock symbol to check if the user is selling the same stock.
- There is the 'equals' method that can be used to compare two Strings. It may be easiest to write code that compares the stock symbol of the stock held to what is sold by ignoring case. The String class has an 'equalsIgnoreCase' method for this kind of comparison in addition to the normal 'equals' method.

Part 3 - Sample Output

The following is an example of normal output:

```
Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.

Please enter your name and hit <ENTER>
Bob Broker
Please enter your initial account balance and hit <ENTER>
3000

Your account details:
Name: Bob Broker
Account Balance: 3000.0
You do not own any stock.

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>
IBM
Please enter the number of (whole) shares and hit <ENTER>
26
Please enter the price of the stock and hit <ENTER>
64.25
You have bought 26 shares of IBM at $64.25 per share
```

Your account details:
Name: Bob Broker
Account Balance: 1329.5
You own 26 shares of IBM

You will now sell stock from your account
Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

15

Please enter the price of the stock and hit <ENTER>

65.75

Your account details:
Name: Bob Broker
Account Balance: 2315.75
You own 11 shares of IBM

This is some sample output showing various errors:

Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.

Please enter your name and hit <ENTER>

Bob Broker

Please enter your initial account balance and hit <ENTER>

-245

Negative balances are not allowed.

An account has been opened with \$1000.0

Your account details:
Name: Bob Broker
Account Balance: 1000.0
You do not own any stock.

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

10

Please enter the price of the stock and hit <ENTER>

56.34

You have bought 10 shares of IBM at \$56.34 per share

Your account details:
Name: Bob Broker
Account Balance: 436.59999999999999
You own 10 shares of IBM

You will now sell stock from your account
Please enter the stock symbol and hit <ENTER>

BAC

Please enter the number of (whole) shares and hit <ENTER>

34

Please enter the price of the stock and hit <ENTER>

24.7

You don't own that stock to sell

Your account details:
Name: Bob Broker
Account Balance: 436.59999999999999
You own 10 shares of IBM

Example of selling off all shares and entering lowercase or uppercase stock symbol:

Your account details:
Name: Bob Broker
Account Balance: 3000.0
You do not own any stock.

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

10

Please enter the price of the stock and hit <ENTER>

56.97

You have bought 10 shares of IBM at \$56.97 per share

Your account details:
Name: Bob Broker
Account Balance: 2430.3
You own 10 shares of IBM

```
You will now sell stock from your account
Please enter the stock symbol and hit <ENTER>
ibm
Please enter the number of (whole) shares and hit <ENTER>
10
Please enter the price of the stock and hit <ENTER>
59.43
```

```
Your account details:
Name: Bob Broker
Account Balance: 3024.6000000000004
You do not own any stock.
```

Part 4 - Discussion Questions

- Think about different ways to handle the situation where the user is selling off all of the stock they own. Based on the output of the method printing the account information what might be the output that makes the most sense if a user has sold off all the stock they bought?
- In the 'main' method you may have two different variables that refer to the Stock objects created from user prompts. What is the benefit of having two separate variables? What would be reasons to reuse the variable that was used for the bought stock to refer to the sold stock?

Lab 14 - Looping

Time for Lab: 30 minutes

In this lab, you will experiment with looping constructs in Java. Using a simple **for** loop, you will see how to repeat logic in a structured and controlled manner.

Part 1 - Create the Class

We will create an entirely new Java class just to handle this logic, and we will create it in the **SimpleProject**. It will be similar to the **SimpleArithmetic** and **HelloWorld** classes we created earlier in that they will simply have a **main** method.

The class will be very simple in design. It will simply be an “adder” class. We will use a **for** loop to repeatedly ask the user to enter a number (10 times). Each number entered will be added to a running sum; at the end of the loop, the sum total will be displayed.

- ___ 1. Close all open files.
- ___ 2. In the *Package Explorer*, right click on **SimpleProject** and select **New | Class**.
- ___ 3. For the *Package*, enter **com.simple**
- ___ 4. For the *Name*, enter **SimpleAdder**
- ___ 5. Check the box marked **public static void main(String args[])**
- ___ 6. Click **Finish**.

The code editor will open on the new class.

- ___ 7. We will now start coding. Enter the following code in the **main** method:

```
int sum = 0;
Scanner scan = new Scanner(System.in);
for(int x = 0; x < 10; x++) {

}
System.out.println("The sum is " + sum);
scan.close();
```

- ___ 8. Save the class.

We first declare an **int** variable (called **sum**) that will represent our running total.

We then declare an instance of a **Scanner** that we will use to obtain input from the user.

We then begin a **for** loop and set up (but do not code) the loop body. We will add the body in a moment. First, notice the **for** loop syntax. The first part of the loop statement (**int** `x = 0`) is the *initial condition*. In it, we declare a loop variable which in our case is a simple **int** called **x**.

The next part of the statement (`x < 10`) is the *continuation condition*; the loop will continue until this condition is reached. In this case, the code is saying that while the value of **x** is less than 10, the loop should continue.

Finally, the last part of the statement (`x++`) is the *increment clause*. This reflects what should be done on every iteration through the loop. In this case, we merely increment the value of **x** by one.

From this, you should see that our loop will iterate 10 times; once for **x** being 0, another for **x** being 1, another for **x** being 2, and so on.

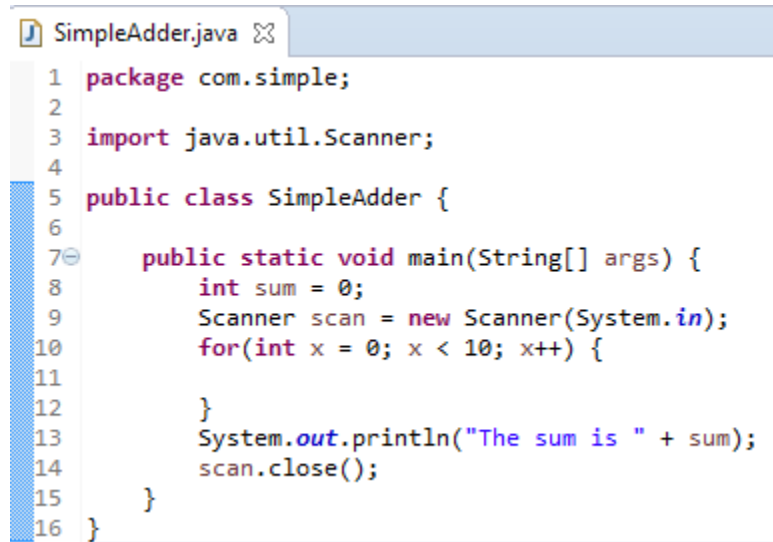
(We place an open and close curly brace pair after the **for** statement. This is the loop body, and we will place the code we want repeated within these curly braces. We will do that later, however.)

Finally, we print out a simple statement showing the sum and close the input scanner.

__9. Examine the *Problems* view.

__10. Eclipse will not be able to resolve **Scanner** as a type. Click anywhere in the source window and hit **Ctrl-Shift-O** to organize imports and make the error go away. Select **java.util.Scanner** when prompted and click **Finish**.

__11. Save the file. There should be no errors. Your code should now look like this:



```
SimpleAdder.java
1 package com.simple;
2
3 import java.util.Scanner;
4
5 public class SimpleAdder {
6
7     public static void main(String[] args) {
8         int sum = 0;
9         Scanner scan = new Scanner(System.in);
10        for(int x = 0; x < 10; x++) {
11
12        }
13        System.out.println("The sum is " + sum);
14        scan.close();
15    }
16 }
```

We can now go ahead and add the loop body code. The loop body will be quite simple.

As stated earlier, we will just prompt the user for a number (using a **Scanner** for **System.in** as we did in a previous exercise) and then add that number to the sum.

__12. Add the following code into the body of the **for** loop.

```
System.out.println("Please enter integer #" + (x+1));
int input = scan.nextInt();
sum += input;
```

Three lines of code. The first line merely displays a meaningful prompt for the user. It tells the user which “number” to enter (e.g. the first number, second number, third number all the way to the 10th number).

The second line performs the actual input of data from the user, and the third line adds it to the current sum.

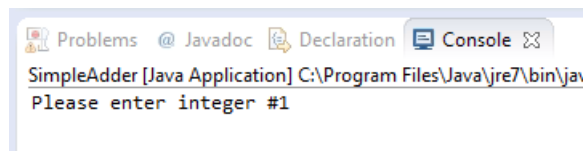
__13. Save the code. There should be no errors regarding to the SimpleAdder class.

The for statements should look like this:

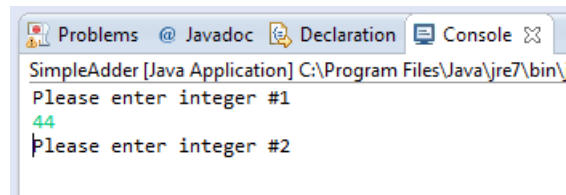
```
for(int x = 0; x < 10; x++) {  
    System.out.println("Please enter integer #" + (x+1));  
    int input = scan.nextInt();  
    sum += input;  
}
```

__14. We can now run and test the code by right clicking on the code editor and selecting **Run As | Java Application**.

Examine the *Console* view.



__15. Enter a number.



The program prompts you to enter the next number.

__16. Keep entering numbers until the program completes.

```
Please enter integer #7  
8  
Please enter integer #8  
19  
Please enter integer #9  
84  
Please enter integer #10  
22  
The sum is 593
```

Success! Our loop is working.

Note that at the moment, there is no concept of error handling.

__17. Try running the program again, but entering some non-numeric (e.g. some character data) into the program.

```
Please enter integer #1
One
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at com.simple.SimpleAdder.main(SimpleAdder.java:12)
```

Java throws a nasty message, raising an **exception**. The program then terminates ungracefully. Naturally, this is not a good user interface; instead of behaving this way, the program should handle itself more gracefully, and that can be done with the use of *exception handling*. We will learn about proper exception handling later on. For now, we will proceed.

Part 2 - Break and Continue

So far, we have seen a simple **for** loop which shows how we can iterate through code. A loop also has two other constructs that can be helpful: **break** and **continue**.

Within a loop, issuing a **continue** statement will tell the current iteration to cease executing, and go to the next iteration.

Issuing a **break** statement will cause the loop to terminate completely. We will use these in our code now.

Let us change the logic of our code a little. Right now, we are simply adding integers, with no regard for error handling. What happens right now if the user enters a *negative* number? The program will still execute, adding the negative number to our sum, thereby *decreasing* it. We will now change our code to disallow this.

Specifically, if a negative number is entered, we will want our program to merely skip over the addition of the entered number to the sum, and continue on with the next iteration. This can be achieved with a **continue** statement.

__1. Within the loop, right *before* the line:

```
sum += input;
```

__2. Enter the following:

```
if (input < 0) {  
    continue;  
}
```

This simply checks to see if the number is less than zero. If it is, skip the remainder of the code in this iteration and go to the next one.

__3. Save the code. There should be no errors.

__4. Test the code by running the program. Enter 10 for the first number, and -1 for all the remaining others.

```
Please enter integer #9  
-1  
Please enter integer #10  
-1  
The sum is 10
```

You should see that the negative numbers have been ignored, as expected.

(Note that you could also have achieved this behavior just by using an **if** statement and proper use of true/false clauses; the **continue** was just one way of achieving this)

Now, let us consider another modification to the program. Right now, we are forced to enter 10 numbers; the loop will continue until all 10 have been entered. Let us change the design allowing the user to complete the program 'early'. Specifically, we will have the program quit the loop immediately if the number 0 is added. Let us add code to this now.

__5. Insert the following code immediately *before* the line `sum += input;` and right after the last **if** statement:

```
if(input == 0) {  
    break;  
}
```

This code is similar to what we entered before. We check to see if the input was some value (0 in this case). If it is, we issue a **break** statement. Recall that a **break** signals that Java should quit the loop completely.

__6. Save the code. There should be no errors. It should look like this:

```
7 public static void main(String[] args) {
8     int sum = 0;
9     Scanner scan = new Scanner(System.in);
10    for(int x = 0; x < 10; x++) {
11        System.out.println("Please enter integer #" + (x+1));
12        int input = scan.nextInt();
13        if (input < 0) {
14            continue;
15        }
16        if(input == 0) {
17            break;
18        }
19        sum += input;
20    }
21    System.out.println("The sum is " + sum);
22    scan.close();
23 }
24 }
25 }
```

__7. Run the code. Enter two integers, a few negative numbers and then 0.

```
Please enter integer #1
15
Please enter integer #2
20
Please enter integer #3
-4
Please enter integer #4
-55
Please enter integer #5
0
The sum is 35
```

As you can see, the negative numbers were ignored (thanks to the **continue** statement) and the 0 caused the loop to complete immediately (due to the **break** statement).

__8. Close all open files.

Part 3 - Review

In this lab, you examined a looping construct; specifically, you saw how to use a **for** loop to repeatedly iterate through code. In addition to that, you saw how you can use the **break** and **continue** statements to exhibit a finer sense of control over loop execution.

Lab 15 - Project - Loop Until Quit (Optional)

Time for lab: 30 minutes

This lab will continue the project. Previously the user has had no control over the operations that are performed. They are prompted to open an account, buy stock, sell stock and then the program ends. In this part of the project the program will be changed so the user can select what operation they want to perform. The program will keep asking the user what they want to do until they indicate they want to quit.

Part 1 - General Requirements

- Modify the code of the StockTracker class to prompt the user for what action they want to perform. The program should keep prompting the user for what action to perform until they indicate they want to quit. You can leave the part of the class that opens the account alone as this only happens once.
- Since the StockAccount class only tracks one stock right now, any attempt to buy a stock with a different symbol than what is already owned should print some type of error message. Modify the 'buyStock' method of the StockAccount class to prevent this.
- Make sure that if the user attempts to buy more of the same stock the program will allow it and simply add the amount of the shares owned together.
- Modify the code of the user prompt to handle what happens when the user inputs an incorrect option.
- **Bonus:** Handle lower and upper case characters for the choice.
- **Bonus:** Modify the operations available to the user so they can deposit or withdraw money from the account. Make sure they can't withdraw more than what they have as cash on hand. Look for ways to refactor code out of the 'main' method to keep this code simple. You may also need to add methods to the StockAccount class.
- **Bonus:** Alter the method that prints account info to print the cash on hand and the value of the entire account, which includes the current value of the stock owned. Assume that the price of the stock is what the last buy or sell price was. If you still hold stock after a transaction you can update the price of the held stock with the last price from the transaction (which you might already be doing).

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- One of the easiest ways to implement the options for the user action may be a “switch” statement. You can have the user type in a character for what action they want to perform and set up different “cases” for each option. Make sure you use a 'break' statement at the end of each case so execution does not execute all the cases. The “default” case can handle what happens if they don't input a correct choice. Since a switch statement does not loop you will likely combine it with some type of loop. It may be easier to implement the selection of actions before adding the looping functionality.
- Look for ways to refactor the code to collect the user choice into a separate method.
- There are many ways to implement prompting the user and looping until they quit. The most important part of this is how they indicate they are done. This should stop the loop and allow the program to finish. In fact, it may be easier to implement and test what happens when a user wants to quit before the code for the buying and selling situations.
- You can declare a boolean variable before the loop and then check this variable to see if the loop should continue. Inside the loop change the value of the variable when the user wants to quit and the loop will stop.
- Since you are now looping you have to handle the situation of multiple purchases and multiple sales of stock. Simply altering the state of the existing Stock object that is linked to the account may not be as simple as linking to the Stock object passed into the 'buyStock' or 'sellStock' methods and simply adjusting the value of the shares to represent the new share total. This will also automatically pick up the new price for the stock object associated with the account.

```
public void buyStock(Stock toBuy) {  
    // other code  
    int newShares = toBuy.getShares();  
    if (heldStock != null) {  
        newShares += heldStock.getShares();  
    }  
    heldStock = toBuy;  
    heldStock.setShares(newShares);  
}
```

- If you are using the Scanner class to read in user input, it does not have a method to read in a single character. You may need to read in the entire line and use methods of the String class to get the first character entered.

Part 3 - Sample Output

This is the normal output with just a few operations:

Your account details:

Name: Bob Broker

Account Balance: 3000.0

You do not own any stock.

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account

Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

25

Please enter the price of the stock and hit <ENTER>

45.8

You have bought 25 shares of IBM at \$45.8 per share

Your account details:

Name: Bob Broker

Account Balance: 1855.0

You own 25 shares of IBM

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

s

You will now sell stock from your account

Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

23

Please enter the price of the stock and hit <ENTER>

56.9

Your account details:

Name: Bob Broker

Account Balance: 3163.7

You own 2 shares of IBM

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

q

Thank you for using the Stock Tracker program

This is sample output showing multiple purchases of the same stock:

Your account details:

Name: Bob Broker

Account Balance: 3000.0

You do not own any stock.

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account

Please enter the stock symbol and hit <ENTER>

MSFT

Please enter the number of (whole) shares and hit <ENTER>

5

Please enter the price of the stock and hit <ENTER>

56.34

You have bought 5 shares of MSFT at \$56.34 per share

Your account details:

Name: Bob Broker

Account Balance: 2718.3

You own 5 shares of MSFT

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account

Please enter the stock symbol and hit <ENTER>

MSFT

Please enter the number of (whole) shares and hit <ENTER>

14

Please enter the price of the stock and hit <ENTER>

58.67

You have bought 14 shares of MSFT at \$58.67 per share

Your account details:

Name: Bob Broker

Account Balance: 1896.92

You own 19 shares of MSFT

This is sample output of trying to buy a different stock than what is already owned:

Your account details:

Name: Bob Broker

Account Balance: 3000.0

You do not own any stock.

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

Please enter the stock symbol you want to buy and hit <ENTER>

IBM

Please enter the number of (whole) shares you want to buy and hit

<ENTER>

24

Please enter the price of the stock and hit <ENTER>

35.2

You have bought 24 shares of IBM at \$35.2 per share

Your account details:

Name: Bob Broker

Account Balance: 2155.2

You own 24 shares of IBM

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

Please enter the stock symbol you want to buy and hit <ENTER>

MSFT

Please enter the number of (whole) shares you want to buy and hit

<ENTER>

36

Please enter the price of the stock and hit <ENTER>

23.5

You can only own shares of one stock at a time

This shows a sample of what happens when the user inputs an incorrect choice:

Your account details:

Name: Bob Broker

Account Balance: 3000.0

You do not currently own any stock

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

d

You have entered an incorrect option

Part 4 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- What did you consider “incorrect input” for the user choice? Can your program handle lowercase and uppercase letters for the legal choices?
- Does it make sense to only be able to buy one stock? This limitation has been placed mainly because you do not currently have a good way of storing information about multiple stocks.

Lab 16 - Subclasses

Time for Lab: 45 minutes

In this lab exercise, you will examine the concept of subclasses and inheritance.

Earlier, we created a **BankAccount** class, that had a very simple design. It had 3 fields (**ownerName**, **accountID** and **balance**), one static field (**interestRate**) and one method (**deposit()**).

Now, let us extend the model a little. We wish to represent a **Savings** account. A savings account is more or less the same as a **BankAccount** (same fields, and method), but with a few extra bells and whistles.

Firstly, a savings account has an operation to calculate interest and add it to the balance. Secondly, every savings account has a minimum balance that must be maintained if interest is to be calculated.

How could we model this? The easy thing to do would be to create a new **SavingsAccount** class, give it the same fields/operations as the **BankAccount** class, and then add the savings account specific facets. We could save some code by copying and pasting a lot of the code from the **BankAccount** into the new **SavingsAccount** class.

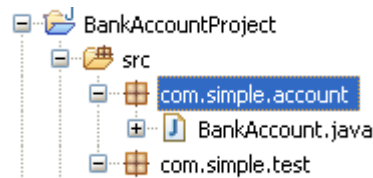
A better approach is to use *inheritance*. In object oriented programming, a class can inherit from another class. An inheriting class (also known as the *subclass*) automatically shares all the code from its parent (also known as the *superclass*), and so has all its fields and methods.

So, if we created **SavingsAccount** as a subclass of **BankAccount**, **SavingsAccount** would automatically get the fields and method “for free”. All we would have to do is write the **SavingsAccount** specific code. We will do this now.

Part 1 - Create the SavingsAccount Class

We will now create the **SavingsAccount** class, subclassing the existing **BankAccount** class.

__1. In the *Package Explorer* view, expand the **com.simple.account** package under *BankAccountProject > src*.



__2. Right click on **BankAccount.java** and select **New | Class**.

The *New Java Class* wizard will appear. The *Package* should be **com.simple.account**

__3. For the *Name*, enter **SavingsAccount**

__4. For the *Superclass*, enter **com.simple.account.BankAccount**. You can also use the **Browse** button and type a few letters in the filter to find the correct superclass.

This is how we specify that this new class should inherit from **BankAccount**.

__5. Make sure the box for *public static void main(String [] args)* is unchecked. We will not need a **main** method here. We will test this class using our BankAccountTester.

Java Class
Create a new Java class.

Source folder: BankAccountProject/src Browse...

Package: com.simple.account Browse...

☐ Enclosing type: com.simple.account.BankAccount Browse...

Name: SavingsAccount

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: com.simple.account.BankAccount Browse...

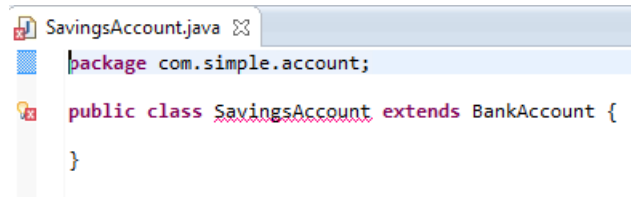
Interfaces: Add...
Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

__6. Click **Finish**.

A code editor will open on the new class.

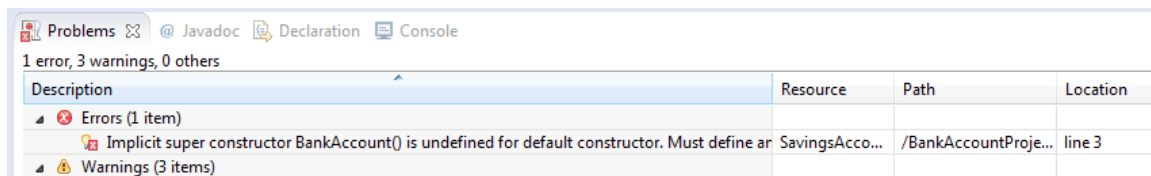


```
package com.simple.account;

public class SavingsAccount extends BankAccount {
}
```

Notice that the class declaration now has a new keyword: **extends**. This keyword is used to denote that the class being defined is a subclass of the class specified. In our case, the **SavingsAccount** **extends** **BankAccount**.

__7. Notice that right away, there is a problem.



Description	Resource	Path	Location
1 error, 3 warnings, 0 others			
Errors (1 item)			
Implicit super constructor BankAccount() is undefined for default constructor. Must define an explicit constructor.	SavingsAccount.java	/BankAccountProject	line 3
Warnings (3 items)			

Java is complaining that there is no default constructor here. Why is this happening?

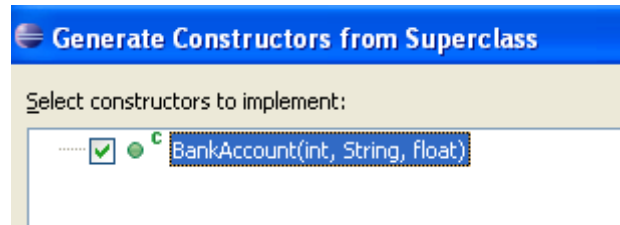
Remember that a class gets the default constructor “for free” **if** it has no other constructor defined for it. The problem is because the **BankAccount** class **does not** have a “default” constructor since we defined a constructor in the **BankAccount** class. When Java tries to define a “default” constructor for the **SavingsAccount** class with an implicit call to the “default” constructor in the superclass this creates problems.

So, in effect, we need to create a constructor for this new class. We will do this now.

We could write a constructor from scratch, but Eclipse can help us.

__8. Right click anywhere in the source and select **Source | Generate Constructors from Superclass**.

The *Generate Constructors from Superclass* window appears.



Eclipse has noticed that the superclass (BankAccount) had the 3 argument constructor and will use it.

__9. Click **Generate**.

__10. A new method appears in the source code.

```
public SavingsAccount(int accountID, String ownerName, float balance) {  
    super(accountID, ownerName, balance);  
    // TODO Auto-generated constructor stub  
}
```

Eclipse has inserted a constructor that takes three arguments. This constructor merely calls the superclass's constructor (as shown by the call to **super**), and we know that will set the three fields accordingly. Our work is done here!

__11. Save the file. There should be no errors.

Part 2 - Test the New Subclass

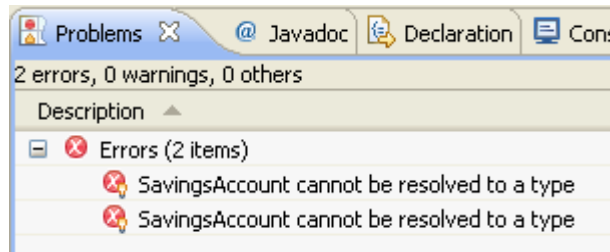
Let us now see if the new SavingsAccount class works. We will create an instance.

__1. Open **BankAccountTester.java**.

__2. Add the following code at the end of the **main** method.

```
SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 6000f);
```

__3. Save the code. A problem appears.



Eclipse is complaining that it cannot locate our new class. This is because we have not imported it yet. Even though we have imported **com.simple.accounts.BankAccount** we need to import our **SavingsAccount** class. We will do this now.

__4. Change the import statement (the second line of code) to read:

```
import com.simple.account.*;
```

By using the * symbol instead of an actual class name, Java will import *all* the classes from that package, including our new **SavingsAccount** – because it belongs to the same package.

__5. Save the class and the errors should disappear.

__6. Continue coding. Add the following code after the instantiation of the **SavingsAccount**.

```
System.out.println("The SavingAccount's balance is " +  
sAccount.getBalance());
```

__7. Save the file and run the code.

```
A Bank Account  
ID: 1  
Balance: 150.0  
Owner: Jeff Lebowski  
account2 is owned by Bunny Lebowski  
account3 is owned by Walter Sobcheck  
account2's balance is 5100.0  
account3's balance is 1000900.0  
account2's interestRate is 5.5  
account3's interestRate is 5.5  
The SavingAccount's balance is 6000.0
```

Note that we use the **getBalance()** method on **sAccount**. Even though we never wrote a **getBalance()** method for **SavingsAccount**, the invocation still works.

We see that the **SavingsAccount** is a new class – but it has the same behavior and understands the same methods as the **BankAccount** class. This is due to the inheritance structure.

Part 3 - Adding A Method

Now, let us add some business logic to the **SavingsAccount** class. Specifically, we will create a method called **payInterest()** that will update the balance based on the interest rate.

__1. Open **SavingsAccount.java**

__2. Before the closing curly brace **}** for the class, add the following method:

```
public void payInterest() {  
    float newBalance = this.getBalance() * (1  
    + (this.getInterestRate() / 100));  
    this.setBalance(newBalance);  
}
```

This method is quite straightforward. It calculates the interest based on the existing balance plus the **interestRate**.

__3. Save the class. There should be no errors.

We have now created a method on **SavingsAccount** class!

Part 4 - Adding A Field

We can now add the field representing the minimum balance to the class.

__1. Open **SavingsAccount.java**.

__2. Add the following field declaration, right after the class declaration, before the constructor:

```
private float minimumBalance = 1000f;
```

Ordinarily, we would generate accessors for this field, but in the context of our code – we will never allow any other class to get or set this **minimumBalance**. We will leave it **private** and hence inaccessible to other classes. The only other code that will use this minimum balance will be our **payInterest()** method.

What is new here is the setting of the default value for the field. The “= 1000” chunk of code states that when the class is initialized, it should automatically set the **minimumBalance** to be 1000. This is handy since we are not allowing public accessors, so nobody else will be able to set it to a different value.

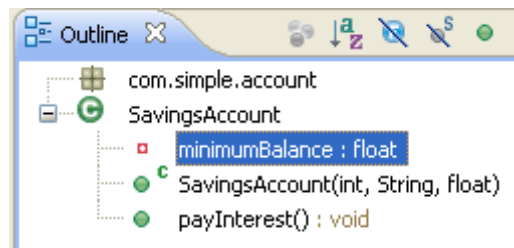
__3. Change the **payInterest()** method to check the minimum balance before updating the balance, as follows:

```
public void payInterest() {  
    float newBalance = this.getBalance() * (1 +  
        (this.getInterestRate() / 100));  
    if (this.getBalance() >= this.minimumBalance) {  
        this.setBalance(newBalance);  
    }  
}
```

All we have done is added an **if** clause right before the call to **setBalance()**

__4. Save the file. There should no errors.

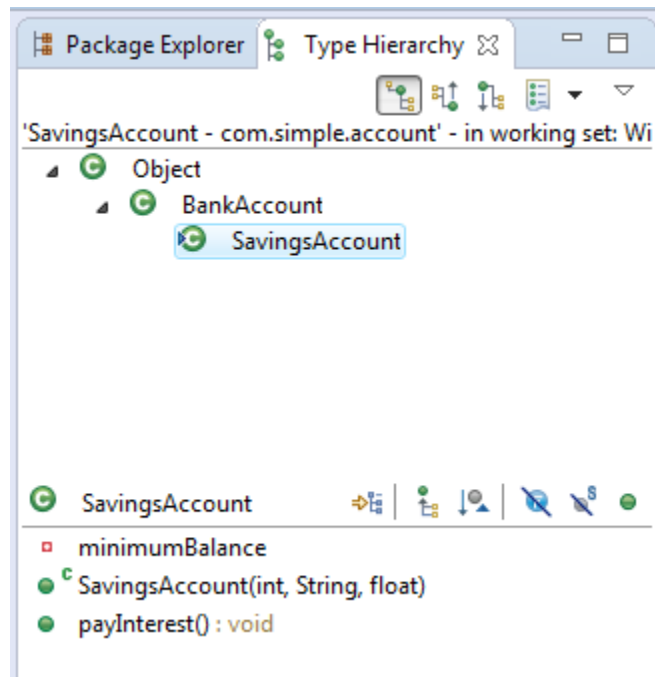
Now, examine the *Outline* view



We see the new field (**minimumBalance**) and methods (constructor and **payInterest()**). What we do not see are any of the *inherited* fields or methods. By default, Eclipse hides that for us.

__5. In this *Outline* view, right click on **SavingsAccount** and select **Open Type Hierarchy**.

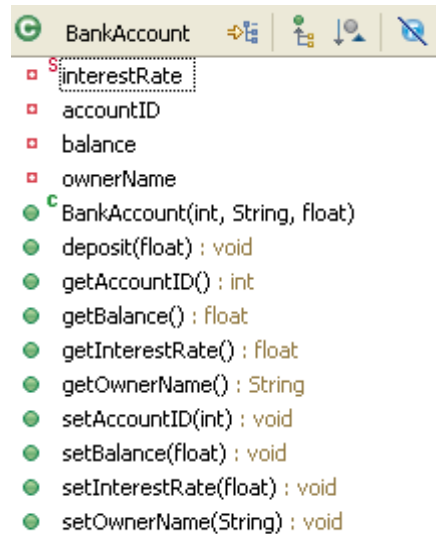
In the far left, the *Hierarchy* view appears.



This shows the relationship between the **SavingsAccount** and the **BankAccount** classes. Note that it is a tree-like structure with **SavingsAccount** being beneath **BankAccount**.

___6. In this *Hierarchy* view, click on **BankAccount**.

What happens?



All the inherited fields and methods are shown. This is a good way to see exactly how an inheritance structure is working.

__7. Close the *Hierarchy* view. We will not be needing it.

Part 5 - Test the Code

We can now test the new features of our `SavingsAccount` class.

__1. Go back to the **`BankAccountTester.java`** class and add the following code to the **main** method, immediately after the **println** showing the **sAccount**'s balance.

```
sAccount.deposit(500f);  
sAccount.payInterest();  
System.out.println("The SavingAccount's new balance is " +  
sAccount.getBalance());
```

Note that we are invoking the **deposit()** method, which is from the superclass (**`BankAccount`**). We then invoke the **payInterest()** method and print out the new interest.

__2. Save and run the code.

```
The SavingAccount's balance is 6000.0  
The SavingAccount's new balance is 6857.4995
```

As expected, the balance is updated. The call to **deposit()** worked, and so did the **payInterest()** call.

__3. Now, let us try setting the balance below the minimum. Locate the line where the constructor is called for the **SavingsAccount** and change it to:

```
SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 1f);
```

We set the balance to \$1, which is below the `minimumBalance` as defined on the class. The deposit that takes place after is only for \$500, which is still below the minimum balance.

__4. Save and run the code.

```
The SavingAccount's balance is 1.0  
The SavingAccount's new balance is 501.0
```

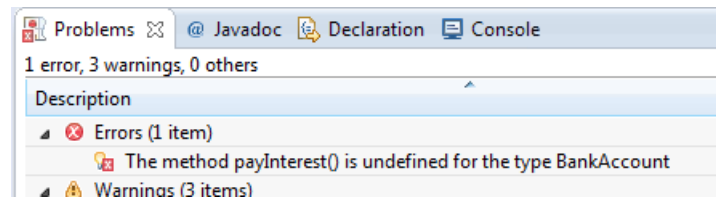
We see that the balance was not updated, even though we called **payInterest()**. It looks like the **payInterest()** logic is working.

Now, let us do an experiment. We know that the **SavingsAccount** inherited all the **BankAccount's** methods. We've seen that we could call **deposit()** and the **get/set** methods on **SavingsAccount**. But does it work the other way? Can a **BankAccount** use the **payInterest()** method? Let us see.

__5. Add the following code to the **main** method:

```
account3.payInterest();
```

__6. Save the file. Notice there is a problem.



It looks like the answer to our question is **no**. A superclass cannot use methods from its subclass. This is one of the rules of inheritance; it only works “down”, not up.

__7. Delete the line you just entered.

__8. Save the file. There should no errors.

__9. Close all open files.

Part 6 - Review

In this exercise, you created a *subclass* of **BankAccount** called **SavingsAccount**. You saw that **SavingsAccount** *inherited* all the fields and methods of the superclass. You also added a method and field to the subclass, while keeping all the behavior of the superclass.

From this, you should see that inheritance is an excellent mechanism for sharing code between common classes.

Lab 17 - Project - Dividend Stocks (Optional)

Time for lab: 30 minutes

This lab will continue the project. One type of stock that is commonly bought pays a dividend. Right now our program can't represent that because it only defines the Stock class. A stock that pays a dividend is the same as any other stock it just has some additional properties about the dividend.

This is a perfect example of a subclass. By creating a DividendStock class that extends the basic Stock class we can add the additional properties without redefining everything in the Stock class. The DividendStock subclass will extend the Stock superclass to add the additional properties.

Part 1 - General Requirements

- Create a new class in the 'com.stock' package and call the class DividendStock. Make sure that the 'com.stock.Stock' class is listed as the superclass and not the Object class.
- Create a private instance variable of the 'double' type and a pair of public 'getter/setter' methods for the dividend paid by the stock. This is the only instance variable required since all others are inherited from the Stock class.
- Create a constructor for the DividendStock class. This constructor should take the stock symbol, number of shares, price, and dividend value as parameters. Use these parameters to set the various instance variables of the DividendStock object. Use the 'super' keyword to call the constructor of the Stock class, but make sure you do this as the first line of the constructor.
- Currently you have a method in the StockTracker class that prompts the user for various stock information and creates a Stock object based on the input values. Modify this method to prompt the user if the stock is a dividend stock (y/n) and if it is collect the amount of the dividend. If the stock is a dividend stock construct and return an object of the DividendStock class.
- The 'buyStock' and 'sellStock' methods should handle the situation where if you are buying and selling the same stock symbol but answer the "dividend stock?" question differently the type of stock (dividend or regular) currently associated with the account is set to the type of stock last bought or sold. The idea is that the stock may begin or stop offering a dividend over the lifetime of the account and the last transaction will determine this.

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- The prompt for whether or not the stock has a dividend may be difficult to interpret. The Scanner class has a 'nextBoolean' method but it looks for 'true' and 'false' as input. If you are using 'yes' or 'no' you may have to look at the first character of what the user inputs to decide what answer they gave.
- When you are prompting for the stock information you are asking if the stock is a dividend stock and reading the input as a String after getting the shares and price. The problem is that since the 'nextInt' and 'nextDouble' methods do not consume the end of the line you may need to call an extra 'nextLine' method simply to ignore the new line character after the int or double read in.
- Remember that the parameter for the 'buyStock' and 'sellStock' method will always represent the most recent answer to the "dividend stock?" question. You will get a DividendStock object when the answer was 'y' and a Stock object when the answer was 'n'. By setting the 'heldStock' field in the account to reference the object coming in as a parameter you can automatically pick up when this changes. You will have to get the value of the shares previously held though before losing the reference to the previous Stock object.
- Note that right now you may not be able to easily tell if the account holds a regular or dividend stock but we will change that in the next project lab.

Part 3 - Sample Output

This is the normal output with just a few operations:

Your account details:

Name: Bob Broker

Account Balance: 3000.0

You do not own any stock.

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account

Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

23

Please enter the price of the stock and hit <ENTER>

78.0

Is the stock a dividend stock? (y/n)

y

Enter the dividend value and hit <ENTER>

0.04

You have bought 23 shares of IBM at \$78.0 per share

Your account details:

Name: Bob Broker

Account Balance: 1206.0

You own 23 shares of IBM

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

s

You will now sell stock from your account

Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

12

Please enter the price of the stock and hit <ENTER>

108.0

Is the stock a dividend stock? (y/n)

y

Enter the dividend value and hit <ENTER>

0.04

Your account details:

Name: Bob Broker

Account Balance: 2502.0

You own 11 shares of IBM

The output for error conditions is similar to previous parts of the project.

Part 4 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- Does it make sense to ask about a dividend when the user is selling stock? What would need to change to only ask about a dividend when buying stock?
- What areas of the program needed to change to account for using dividend stocks? Why can you return a DividendStock object from a method that declares that it returns a Stock object?

Lab 18 - Arrays

Time for Lab: 35 minutes

In this lab exercise, you will examine the use of arrays in Java.

An array is a consecutive collection of 0 or more elements. So far, we have only been dealing with single **int**, **String** and even **BankAccount** instances. Often, however, it is desirable to deal with many at once. For example our **BankAccount** may have multiple owners, meaning that it should have multiple **ownerNames** instead of just the one that it has right now. This can be achieved through the use of arrays. We could modify our **BankAccount** class to use an array of **String** objects for the **ownerName** field instead of just the **String**.

Let us start with something simpler, however.

Part 1 - Object Arrays

We will now experiment with some simple array operations by creating an array and inserting some **BankAccount** objects.

Think of an array as a box of consecutive 'things' (e.g. **ints**). An array has a fixed size (e.g. 10) and each element can be accessed via an index which represents its position in the array. Elements are inserted into an array by means of an assignment operator (=), and can be retrieved via square brackets and the index of the element being retrieved.

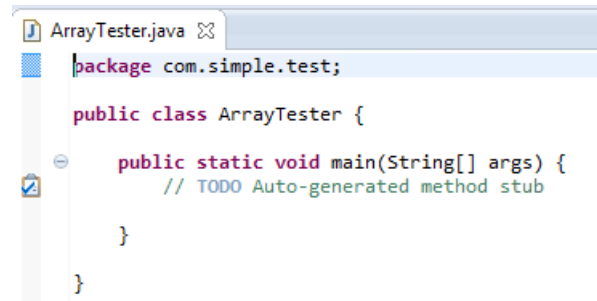
Arrays must first be declared. Once they are declared, objects can be placed inside them.

We will create a new class for our experiments; this class will create an array of **BankAccounts**. and will then insert some **BankAccount** instances into this array. We will then see how to use indexes to access them.

- ___ 1. In the *Package Explorer*, right click on **BankAccountProject** and select **New | Class**.
- ___ 2. In the *New Java Class* window, enter **com.simple.test** for the *Package*.
- ___ 3. Enter **ArrayTester** for the *Name*.
- ___ 4. Check the box marked **public static void main(String[] args)**

__5. Click **Finish**.

The code editor should open.



__6. Add the following code to the **main** method.

```
BankAccount[] accounts = new BankAccount[4];
```

Here, we declare the array. Note that we use the square brackets [] to indicate this is an array. Also note that we initialize the array by giving it a *size*. In this case, the size of the array is 4. So, at this moment, the variable **accounts** points to an array of 4 **BankAccounts**. It is ready for use!

Note that an array is *typed*, meaning that only one type of object can be placed in an array; that type of object is specified at declaration time. So, this declaration states that our array can only hold instances of BankAccounts. Four of them, to be precise.

Eclipse will complain that it cannot resolve BankAccount.

__7. Hit **Ctrl-Shift-O** to organize imports and save the file. There should be no errors.

__8. Continue by adding the following code:

```
accounts[0] = new BankAccount(1, "Jeff Lebowski", 100f);
```

This line tells Java to create a new instance of a BankAccount (with id 1) and places that at element 0 in the array. Here, we see that we use the square brackets and a number [0] to indicate index position.

It is important to realize that arrays use a zero-based index. This means that the first element in the array is considered to be at index (i.e. location) 0. The second element is considered to be at index 1, and so on.

__9. Add the following two lines of code:

```
accounts[1] = new BankAccount(2, "Maude Lebowski", 5000f);  
accounts[2] = new BankAccount(3, "Bunny Lebowski", 1f);
```

Nothing special here; we create 2 more instances and then add them to the array.

__10. Save and run the code. There should be no errors – but no output either. That is fine for now.

Realize now that we have an array of size 4, but we have only inserted 3 elements.

__11. Now, let us try something else. Add the following line of code:

```
accounts[4] = new BankAccount(5, "Ulli Wenk", 14354f);
```

This does not look like anything special, but notice the index; we are trying to access `accounts[4]` which would be the 5th position in the array; this is *out of bounds* for the array since the array is only 4 elements long.

__12. Save and run the code. What happens?

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4  
    at BankAccountProject/com.simple.test.ArrayTester.main(ArrayTester.java:12)
```

A nasty error has popped up, telling us that we tried to access an array outside its bounds. In the case of our code, it makes sense; we tried to index a position that was not within the array. Such a mistake (“out of bounds exception”) is a common and care must be taken to avoid this whenever possible.

__13. Delete the offending line of code.

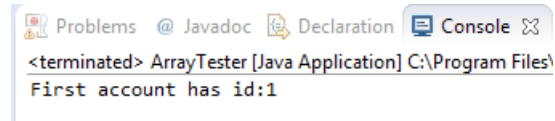
We've seen how to insert elements into an array (by using the assignment = operator). Let us now see how to get elements out of an array.

__14. Add the following code:

```
BankAccount first = accounts[0];  
System.out.println("First account has id:" + first.getAccountID());
```

The first line declares a new variable called **first** (of type `BankAccount`) and sets it to the first element in the array – which should be our first `BankAccount`. This, in effect, “pulls out” the first element of the array and sticks it inside that **first** variable. We then print out **first**'s id – which should be **1**. (We could also print out **first**'s `ownerName` and `balance`, but we will assume that if the ID is correct, the rest will be as well)

__15. Save and run the code. As expected:

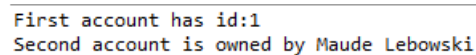
A screenshot of an IDE's console window. The window has tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active, showing the output: '<terminated> ArrayTester [Java Application] C:\Program Files\ First account has id:1'. The text is in a monospaced font with some color coding (blue for the path, red for the error/terminated status).

__16. Now add the following code:

```
BankAccount second = accounts[1];
System.out.println("Second account is owned by " +
second.getOwnerName());
```

Similar to the previous code, we 'pull out' the account at position 1 (the second element) and then print out its **ownerName**.

__17. Save and run the code.

A screenshot of an IDE's console window showing two lines of output: 'First account has id:1' and 'Second account is owned by Maude Lebowski'. The text is in a monospaced font with color coding (blue for the name, red for the first line).

As expected, the correct account was located.

__18. Finally, let us try something different. Add the following code:

```
BankAccount fourth = accounts[3];
System.out.println("Fourth account has balance " +
fourth.getBalance());
```

Notice that we are attempting to access index 3, which is the *fourth* position in the array. However, we *never inserted* anything into this position! So what should happen when we execute this code? Let us see.

__19. Save and run the code. What happens?

```
First account has id:1
Second account is owned by Maude Lebowski
Exception in thread "main" java.lang.NullPointerException
    at BankAccountProject/com.simple.test.ArrayTester.main(ArrayTester.java:20)
```

We see Java has raised a **NullPointerException**. What happened?

Since we never assigned **accounts[3]** to be anything, it currently has a value of **null** which is a special Java state meaning “nothing”. So, the variable **fourth** is pointing to null. Then, when we try to print out **fourth.getBalance()**, we are essentially calling **null.getBalance()** which should return an error – you cannot obtain a balance from **null**!

From this, we see that when declaring an array of objects, it initially sets its elements to null. (Note that if the array is composed of primitives, the *default value* of the primitive will be used instead).

__20. Remove the two lines of code added above ('fourth' variable) that introduced the errors.

Part 2 - Use Subclass in Array

An array can store any object as an element that can be treated like the type of object declared for the type of the array. An instance of a subclass can also be treated like the superclass type. Storing an instance of subclasses in an array of the superclass type can be a powerful way to write better code. We can write code to deal with all of the objects in the group, for example all of the accounts, even if some of the objects are a more specific type of object. We will see this in action.

__1. Modify the code of the first object to be created and stored in the array to be a **SavingsAccount** object instead of a **BankAccount** object as shown below. You will have errors until the next steps.

```
accounts[0] = new SavingsAccount(1, "Jeff Lebowski", 100f);
```

__2. Hit **Ctrl-Shift-O** to organize imports and save the file. There should be no errors.

__3. Run the ArrayTester class and notice the output is the same as before.

```
First account has id:1  
Second account is owned by Maude Lebowski
```

__4. Add the following line of code in bold

```
BankAccount first = accounts[0];  
System.out.println("First account has id:" + first.getAccountID());  
first.payInterest();
```

__5. Save the code and notice it doesn't compile. This is because the 'payInterest' method is only defined in the SavingsAccount class. Even though the object is actually a SavingsAccount object, the 'first' variable and the element of the array that refer to the object only can be sure that the object is a BankAccount.

__6. Delete the line with the 'payInterest' method so the code again compiles and runs.

__7. Save the file.

Part 3 - Add An Array To The SimpleAdder

Let us now use our knowledge of arrays to tackle something a little more complex.

__1. Open the **SimpleAdder.java** code that you wrote in a previous lab under SimpleProject.

Currently, it just loops and prompts the user to enter a number 10 times before printing out the sum. Each number that is entered by the user is added to the sum and then “thrown away”. What if we wished to change the design of the program such that it remembered all of the numbers that are entered? That way, at the end of the program, the code could display all of the numbers that were entered in addition to the sum.

We will make this change and use an array to help us with this.

As we iterate through the loop, we will want to “save” each number that is entered. We can do this using an array.

At the start of our code, we will declare a new array (of size 10). As we iterate through the loop, we will take every number that the user enters and insert it into the array.

Then, at the end of the code (after the prompting loop), we will iterate through the array and print out its contents. Let us do this now.

__2. In the **SimpleAdder.java** main method, locate the following line:

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

__3. Immediately after this line, add the following code:

```
int numbers[] = new int[10];
```

__4. Inside the loop body, locate the line:

```
sum += input;
```

__5. Immediately after this line, add the following code:

```
numbers[x] = input;
```

This line is saying “Put the number that was just entered into the **x**th position in the array”.

This works nicely for us because our loop starts with **x** being zero. Recall that **x** is our loop variable; we are using it to keep track of how many times we have been through the loop. So, on the first iteration, the value of **x** is zero; accordingly, the first number that the user enters will be placed in **numbers[0]** which is the *first* element in the array. The second iteration will place the number in **numbers[1]** and so on. It is important that although the array has a size of 10, it is indexed from 0 to 9!

Finally, after the loop has completed, we should iterate through the array and print out all the numbers that were entered.

__6. Locate the line:

```
System.out.println("The sum is " + sum);
```

__7. And enter the following code immediately before it:

```
System.out.println("The numbers you entered were: ");  
for(int eachNumber : numbers) {  
    System.out.println(eachNumber);  
}
```

This code sets up another loop, which uses the "enhanced for loop" or "for-each loop" syntax to iterate over the 'numbers' array and set a variable 'eachNumber' to the current element of the array to display.

__8. Save the code. There should be no errors.

__9. Run the code.

```
Please enter integer #9  
9  
Please enter integer #10  
10  
The numbers you entered were:  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
The sum is 55
```

__10. Close all open files.

Notice that the numbers you entered were printed out. We have used the array as a storage mechanism and iterated through it using loops.

Part 4 - Review

In this lab, you examined arrays. You saw how an array can be used to store a series of objects, and how they can be accessed using indexes. You also saw how loops can be used to iterate through arrays.

Lab 19 - Method Overriding

Time for Lab: 45 minutes

A common practice in object oriented programming – especially in inheritance cases – is to *override* methods. Simply put, to override a method means to change the logic of an inherited method with the same name. We will examine this here.

Part 1 - Adding A print Method

__ 1. Open the **BankAccountTester.java** class.

Right now, every time we wish to inspect a **BankAccount** instance, we are using a sequence of **println** statements. This is tedious and unnecessarily complex.

A better approach would be to write a *method* to do this for us. Ideally, we could write a method on the **BankAccount** class, called **print()** which would print the data of the current **BankAccount** to **System.out**. Let us do this now.

__ 2. Open **BankAccount.java**.

__ 3. Add the following method to the class.

```
public void print() {  
    System.out.println("\nAn Account");  
    System.out.println("Account ID:" + this.getAccountID());  
    System.out.println("    Owner:" + this.getOwnerName());  
    System.out.println("    Balance:" + this.getBalance());  
}
```

This method is very simple; it prints out the field information to the console. (The '\n' character is to force the print to go to a new line. This makes the output easier to read.)

__ 4. Save the class.

We can now make our **BankAccountTester** quite a bit more simple.

__ 5. Open **BankAccountTester.java**.

__6. Right now, it is a mess of mostly **println** statements. Let us start clean. Delete *all* the code inside the **main** method *except* for all the constructor calls. When you are done, it should look like this:

```
package com.simple.test;

import com.simple.account.*;

public class BankAccountTester {

    public static void main(String[] args) {
        BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);
        BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);
        BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);
        SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 1f);
    }
}
```

__7. Let us now test our new **print** method. Add the following code to the **main** method after the constructor calls:

```
account.print();
account2.print();
account3.print();
```

__8. Save and run the code.

```
An Account
Account ID:1
    Owner:Jeff Lebowski
    Balance:100.0

An Account
Account ID:2
    Owner:Bunny Lebowski
    Balance:5000.0

An Account
Account ID:3
    Owner:Walter Sobcheck
    Balance:1000000.0
```

Examine the output. You may have to re-size the console view to see the entire output. It looks like our **print** method is working.

__9. Let us see if the **print** statement works on our **SavingsAccount**. Add the following code:

```
sAccount.print();
```

__10. Save and run the code.

```
An Account  
Account ID:3  
Owner:Donny K  
Balance:1.0
```

We see that our **SavingsAccount** is indeed printed out. This is thanks to inheritance.

Part 2 - Printing the SavingsAccount

What if, however, we wanted the **SavingsAccount** to print out the **interestRate** as well as the **minimumBalance** required?

We could go back to the **BankAccount** class and edit the **print** method to include those fields, but that may not be a good solution for two reasons: firstly, we are not interested in seeing the **interestRate** for a regular **BankAccount** and secondly, the **BankAccount** itself does not have a **minimumBalance** field – and hence would not be able to print it out.

It is clear that we will need to add a method to the **SavingsAccount** class. So, we could add a method called **savingsAccountPrint()** that would print all the fields. That, however, is not convenient because any class using the **SavingsAccount** would have to be aware of both the **print()** and the **savingsAccountPrint()** methods. A better approach, however, would be *override* the **print** method.

Overriding the method means to implement an inherited method with the same name. This means we will create a method called **print** (with the same method signature) as the superclass's **print** method. Now, when code calls **.print()** on a **SavingsAccount** instance, this code will be executed instead. We will override this method now.

We will keep things simple for now. Our code will just print the **interestRate** and **minimumBalance**.

__1. Open **SavingsAccount.java**.

__2. Add the following method to the class:

```
public void print() {  
    System.out.println("\nSavings Summary:");  
    System.out.println("  Interest rate:" + this.getInterestRate());  
    System.out.println("Minimum Balance:" + minimumBalance);  
}
```

Note that we use the *get* method to get the interest rate, but do not use a *get* method for the minimum balance. Can you explain why?

__3. Save the code. There should be no errors.

__4. Switch back to the **BankAccountTester**. Without changing any code, run the class.

```
Savings Summary:  
  Interest rate:0.0  
Minimum Balance:1000.0
```

It looks like the **print** method is working well! However, as we stipulated, it is only printing the interest rate and minimumBalance. What if we want the account id, owner name and balance to be printed as well? This is easy to fix.

__5. Go back to **SavingsAccount.java**.

__6. In the **print** method, add the following bolded line:

```
System.out.println("\nSavings Summary:");  
super.print();  
System.out.println("  Interest rate:" + this.getInterestRate());  
System.out.println("Minimum Balance:" + minimumBalance);
```

Here, we invoke the **print** method from the superclass. Let us see the effect this has.

__7. Save the class.

__ 8. Switch back to and then run the **BankAccountTester**.

```
Savings Summary:  
  
An Account  
Account ID:3  
    Owner:Donny K  
    Balance:1.0  
    Interest rate:0.0  
Minimum Balance:1000.0
```

Much better!

Now, when we call the **print** method on either a **BankAccount** or **SavingsAccount** instance, the appropriate text will be printed. The one **print** method will behave differently, based on whichever class is invoking it. Anyone using either of these classes will only have to know about the one **print** method. This makes for a much simpler programming model.

Part 3 - Using **System.out.println**

So, we have now written a **print** method for our classes. There is one tiny issue, however. When printing to the console it is not common to call a special method on a class instance; instead, it is more common to call **System.out.println()** on the object we wish to see on the console. We have not done that so far; why not? Why have we not just been able to call **System.out.println(account)**? After all, we could call this method on other **Strings** and **ints**.

Let us see why.

__ 1. Open **BankAccountTester.java**.

__ 2. Add the following line of code at the end **main** method.

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

__3. Save and run the code. What do you see?

```
Owner:Donny K  
Balance:1.0  
Interest rate:0.0  
Minimum Balance:1000.0  
com.simple.account.BankAccount@1d1acd3
```

Not quite what we were expecting, is it?

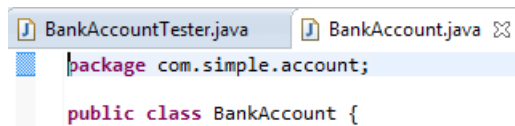
When **System.out.println()** is invoked on an object, Java tries to convert that object into a String first, and it is that String that is sent to the console.

So why is our **BankAccount** being turned into that funny looking String? How is this String conversion taking place?

The answer is well hidden. To convert an object to a String, the **System.out.println()** will call a method called **toString()** on the class being printed. That method should return a String representation of the class being printed, and it is that String that is sent to the console.

But wait a moment: we have no such **toString()** method on our classes. Where is it coming from? Answer: **Object**

In Java, all classes ultimately inherit from a class called **Object**. If you look at the class declaration for **BankAccount**, it looks like this:



```
package com.simple.account;  
  
public class BankAccount {
```

Even though we do not see any **extends** use here, it is implicit that **BankAccount** is a subclass on **Object**. This means that any methods that **Object** has will be inherited by **BankAccount**. One of the methods on **Object** is a method called **toString()**.

What about SavingsAccount? **SavingsAccount** itself extends **BankAccount** and thus also inherits all of the **Object** methods, including **toString()**

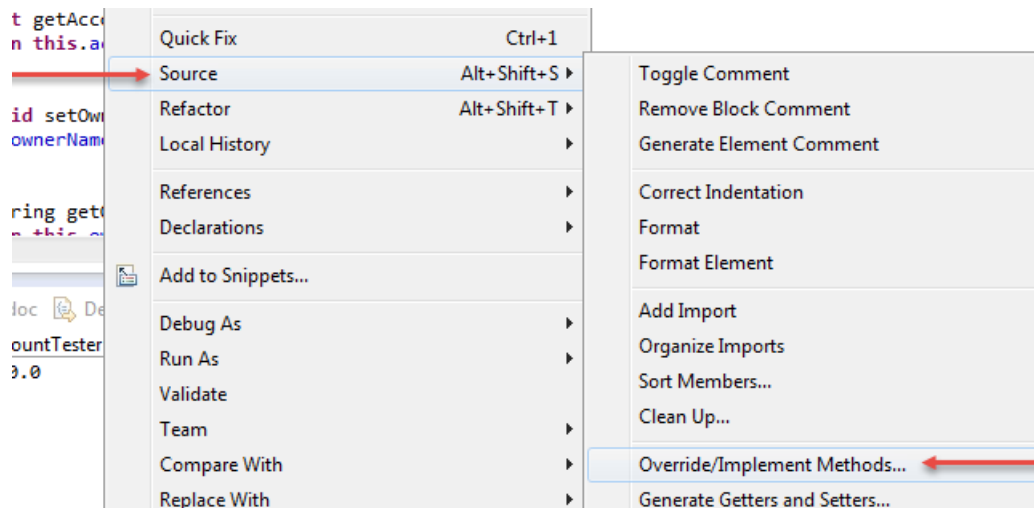
So, the text that is printed right now "com.simple.account.BankAccount@addbf1" is the result of the **toString()** call of **Object**.

If we do not like this, we can simply *override* the **toString** method in our classes! We will do this now.

___4. Open **BankAccount.java**.

We will override the method here.

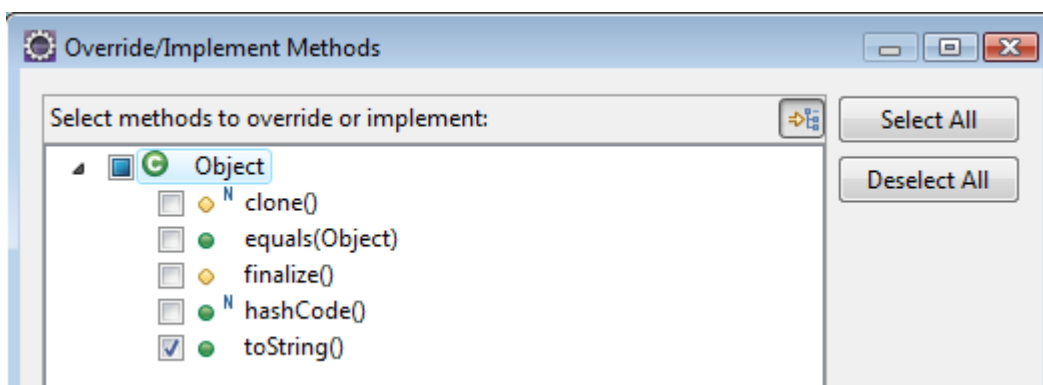
___5. Right click anywhere on the code editor and select **Source | Override/Implement Methods...**



The *Override/Implement Methods* window appears.

Eclipse has examined the superclass and listed all the methods that it has. This dialog allows you to select which of these methods you want to override.

___6. Check the box for **toString()** and click **OK**.



A new method should appear in the source.

```

@Override
public String toString() {
    // TODO Auto-generated method stub
    return super.toString();
}

```

__7. Delete the code inside this method, and change it to the following:

```

return "An account with id " + this.getAccountID() +
" with balance " + this.getBalance() +
" owned by " + this.getOwnerName();

```

Note the use of the + signs to concatenate a longer String. It is this String that is returned.

When you are done, the method should look like this:

```

@Override
public String toString() {
    return "An account with id " + this.getAccountID() +
        " with balance " + this.getBalance() +
        " owned by " + this.getOwnerName();
}

```

__8. Save the code. There should be no errors.

__9. Let us test the code. Switch back to **BankAccountTester**.

__10. Delete *all* the calls to **print()**, but leave the **System.out.println()** line.

__11. Then, add the following lines:

```

System.out.println(account2);
System.out.println(account3);

```


Your code should now look like:

```
public static void main(String[] args) {  
    BankAccount account = new BankAccount(1, "Jeff Lebowski", 100f);  
    BankAccount account2 = new BankAccount(2, "Bunny Lebowski", 5000f);  
    BankAccount account3 = new BankAccount(3, "Walter Sobcheck", 1000000f);  
    SavingsAccount sAccount = new SavingsAccount(3, "Donny K", 1f);  
  
    System.out.println(account);  
    System.out.println(account2);  
    System.out.println(account3);  
}
```

__12. Save and run the code. What do you see?

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski  
An account with id 2 with balance 5000.0 owned by Bunny Lebowski  
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
```

From a functional perspective, it is more or less the same as the old **print** methods that we were using. However, from a coding perspective, this was a better approach. **System.out.println()** is commonly used by developers everywhere, and adding the **toString()** method as we did guaranteed that calling **System.out.println()** on the **BankAccount** will behave in a proper way.

Let us see what happens when we try this on the **SavingsAccount** class.

__13. Add the following line to the **main** method of the **BankAccountTester** class.

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

__14. Save and run the class.

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski  
An account with id 2 with balance 5000.0 owned by Bunny Lebowski  
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck  
An account with id 3 with balance 1.0 owned by Donny K
```

It worked in exactly the same way as calling **println** on the **BankAccount** did. This is because the **SavingsAccount** *inherited* the **toString()** method from **BankAccount**.

Part 4 - Override toString In The SavingsAccount

As before, we should see the `interestRate` and `minimumBalance` for the `SavingsAccount`. How can we make this happen? Simple; as we did with the `print` method, we can override the `toString` method in `SavingsAccount`.

___ 1. Open `SavingsAccount.java`.

___ 2. Right click anywhere on the code editor and select **Source | Override/Implement Methods...**

The *Override/Implement Methods* window appears.

Notice that there are now a lot more methods that we can override than there were for the `BankAccount`. This makes sense because we are now working with the `SavingsAccount`, which could override all of `BankAccount`'s methods in addition to `Object`'s methods.

___ 3. Scroll down in the list and check `toString()` and click **OK**.

The method should appear in the class code.

___ 4. Change the method to look like the following:

```
public String toString() {  
    return super.toString() +  
        " with interest rate " + this.getInterestRate() +  
        " and minimum balance " + this.minimumBalance;  
}
```

Note the call to the `super.toString()` method.

___ 5. Save the code. There should be no problems.

___ 6. Go back to the `BankAccountTester` and run the code. You should see the following:

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski  
An account with id 2 with balance 5000.0 owned by Bunny Lebowski  
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck  
An account with id 3 with balance 1.0 owned by Donny K with interest rate 0.0 and minimum balance 1000.0
```

As expected, the `SavingsAccount` output has changed.

Congratulations! You have successfully overridden a method from **Object** and made using **System.out.println()** much simpler!

__7. Close all open files.

Part 5 - Review

In this exercise, you examined method overriding and saw how it can be used to increase functionality of classes in an inheritance structure.

You also saw one of the core methods on the **Object** class (**toString()**) and used method overriding on that method to make **System.out.println()** usable with our classes.

Lab 20 - Project - Improved Stock Output (Optional)

Time for lab: 30 minutes

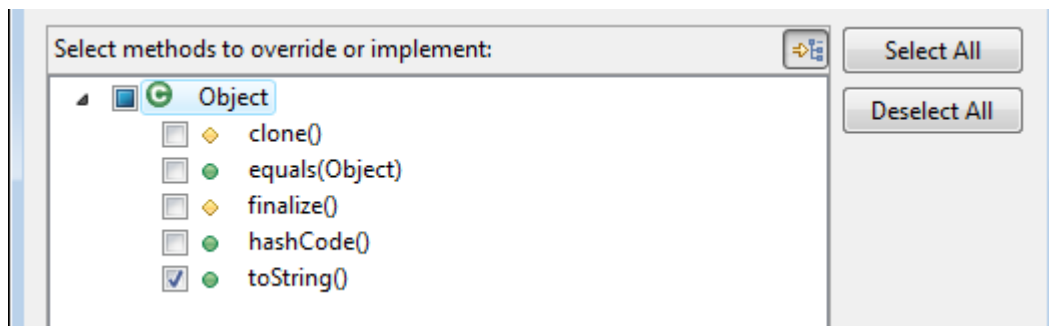
This lab will continue the project. The output of the Stock details still leaves a little to be desired. The StockTracker class must retrieve the individual properties of the stock to print the details and there is no (easy) way to print out the value of the dividend if it is a dividend stock.

One way you can make printing out the details of objects easier is to override the 'toString' method of the Object class. This method returns a String which represents the current state of the object. In this lab you will do that for the Stock and DividendStock classes.

Part 1 - Override Method

Rather than manually defining a method that is meant to override another method it is generally better to use the tools in Eclipse to do this. That way you can be guaranteed that the method signature matches exactly.

- ___ 1. Open the code of your Stock class if it is not already.
- ___ 2. Make sure the editor for the Stock class is active and select **Source** → **Override/Implement Methods...** from the Eclipse menus.
- ___ 3. Check off the **toString** method as shown below and click the **OK** button.



4. Find the method that was added to the class. Even though there may be a default implementation you would be free to change the implementation of the method and simply leave the signature as it is.

```
@Override
public String toString() {
    // TODO Auto-generated method stub
    return super.toString();
}
}
```

Note: Notice the '@Override' annotation. This tells the compiler to indicate a compilation error if the method does not actually override a method of the same name and signature. Thinking that a method overrides another method but not having that be the case because the signature of the method is slightly different can be a difficult problem to track down.

Part 2 - General Requirements

- Modify the generated 'toString' method in the Stock class so that the String returned should include the number of shares, the stock symbol, and the current price. The 'toString' method should have been created already by the steps in the last part.
- Provide a 'toString' method for the DividendStock class. This method can call the Stock version of the 'toString' method and simply add to the end information about the dividend price.
- Modify the method in the StockTracker class that prints account information to use the 'toString' methods of the stock class. The code will use the 'getStock' method to get the Stock owned and then use the 'toString' method on the object returned. The StockTracker class may still print out some type of output at the beginning of each line showing stock data but it should not call any 'get' methods of the stock object.

Part 3 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- Make the 'toString' methods as simple as possible. The String returned by these methods should only apply to the Stock. In particular it would not mention anything about “owning” a stock because this implies association with a StockAccount, information the Stock classes do not have. The StockTracker class can output some information about the fact the stock is “owned” before the details.
- You will still need to check if the Stock object returned by the StockAccount.getStock method is null. If you do not check this you may get errors when trying to call the 'toString' method on an object that doesn't exist.

Part 4 - Sample Output

This is the normal output with just a few operations:

```
Welcome to the Stock Tracker Program!
This program will help you track information
about your investments.
```

```
Please enter your name and hit <ENTER>
```

```
Bob Broker
```

```
Please enter your initial account balance and hit <ENTER>
```

```
4000
```

```
Your account details:
```

```
Name: Bob Broker
```

```
Account Balance: 4000.0
```

```
You do not own any stock.
```

```
You can (b)uy stock, (s)ell stock, or (q)uit
```

```
Enter the first letter of your choice above and hit <ENTER>
```

```
b
```

```
You will now buy stock for your account
```

```
Please enter the stock symbol and hit <ENTER>
```

```
IBM
```

```
Please enter the number of (whole) shares and hit <ENTER>
```

```
12
```

```
Please enter the price of the stock and hit <ENTER>
```

```
134.0
```

```
Is the stock a dividend stock? (y/n)
```

```
n
```

```
You have bought 12 shares of IBM at $134.0 per share
```

Your account details:
Name: Bob Broker
Account Balance: 2392.0
You currently own 12 shares of IBM at \$134.0 per share

Output showing the output of a dividend stock:

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>
b
You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>
IBM
Please enter the number of (whole) shares and hit <ENTER>
12
Please enter the price of the stock and hit <ENTER>
125.0
Is the stock a dividend stock? (y/n)
y

Enter the dividend value and hit <ENTER>
0.05
You have bought 12 shares of IBM at \$125.0 per share

Your account details:
Name: Bob Broker
Account Balance: 2500.0
You currently own 12 shares of IBM at \$125.0 per share, dividend of \$0.05

The output for error conditions is similar to previous parts of the project.

Part 5 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- Now that we are using the 'toString' method, how does the fact the program may refer to a DividendStock show up? Notice that the rest of the code can still work with just Stock objects but if it is a DividendStock you will be able to see that in the output.

Lab 21 - Exception Handling

Exceptions are a mechanism that Java uses as a notification that something has gone wrong. Erroneous code can *throw* an exception; any code that calls that erroneous code can *catch* that exception. Catching an exception typically entails handling it; handling an exception might include things like re-attempting the code, or even ignoring the exception completely.

An Exception in Java is a regular Object, like String or BankAccount. They can be initialized and created using a constructor, and have their own hierarchy. It is also possible to write your own Exception class.

This lab exercise will examine exception handling in Java.

Part 1 - The Try-Catch Block

Any code that can potentially throw an exception (i.e. any code that could cause a failure) should be surrounded with a **try** block. Immediately following the **try** block should be a **catch** block.

Any code throwing an error within the **try** block will cause the code in the **try** block to cease executing, and control will be turned over to code in the **catch** block.

Think back to the **SimpleAdder.java** that we wrote in an earlier lab. If you recall, this program would iterate through a loop and ask the user to enter a number.

- __ 1. Open **SimpleAdder.java** from **SimpleProject**.
- __ 2. Run it and enter a String instead of a number. What is returned?

```
Please enter integer #1
One
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at com.simple.SimpleAdder.main(SimpleAdder.java:14)
```

We see here that Java has thrown a **java.util.InputMismatchException** and exited the program ungracefully.

Immediately following the exception is a *stack trace*. This shows the sequence of method calls that led up to the exception. If you look at the bottom line of the stack trace, we can see that the exception was initially caused by code on line 13 of our SimpleAdder.java class, in the **main** method.

The error is actually occurring because of the call to **scan.nextInt()**. This method is expecting an **int** to be entered; when a String was entered instead, Java threw an exception; our code did not handle the exception, so Java handled it in its own way (i.e. by exiting ungracefully).

What we need to do is modify the code and **try** the call to **scan.nextInt()** and follow that with a **catch** of our particular exception type (**java.util.InputMismatchException**)

__ 3. In **SimpleAdder.java** locate the line

```
int input = scan.nextInt();
```

__ 4. Delete it and replace it with the following code:

```
int input = 0;
try {
    input = scan.nextInt();
} catch (java.util.InputMismatchException exception) {
    System.out.println("Please enter only numeric values.");
    break;
}
```

This is a **try/catch** block. The **try** block (denoted by the curly braces) is the code that Java will 'try' to execute. The **catch** block specifies what sort of error it is looking for; in this case, our code is on the lookout for an **InputMismatchException**.

Note that the syntax of the catch clause; it specifies the *type* (**InputMismatchException**) of the exception it is expecting and assigns it a variable name – in this case, the variable is simply named **exception**. This variable exists because the exception object itself may need to be referred to from within the body of the catch message. We will see an example of this later.

The **InputMismatchException** is an exception that is a part of the Java language. Java itself contains an entire hierarchy of exceptions; **InputMismatchException** is one of them, and it just so happens that it is potentially thrown by **scan.nextInt()**. Other exceptions that we have seen so far: **NullPointerException** and **ArrayIndexOutOfBoundsException**

If the code within the **try** block encounters a String instead of the expected **int**, it will 'throw' an **InputMismatchException** which will then be 'caught' by the **catch** block. There, we put our error handling code.

In this case, our handling code in the **catch** block is simple; it merely notifies the user that the wrong input type was entered, and then exits the loop. It may seem quite simple, but note that it is much nicer than seeing a stack trace and then exiting ungracefully.

Let us now test our code and see what happens.

___ 5. Save and run the code.

___ 6. Again, enter a String instead of a number. What do you see?

```
Please enter integer #1
One
Please enter only numeric values.
The numbers you entered were:
0
0
0
0
0
0
0
0
0
0
0
The sum is 0
```

This is much better. Instead of seeing the stack trace and ungraceful exit, our code continues and prints out the data that it had received. (A nasty side effect is all the extra 0's that are printed out because they were in the array; this is easily fixed, however, and we will not worry about this for now)

This shows that by adding a simple **try/catch** block, we have started using good (but not great) error handling code.

___ 7. Close the file.

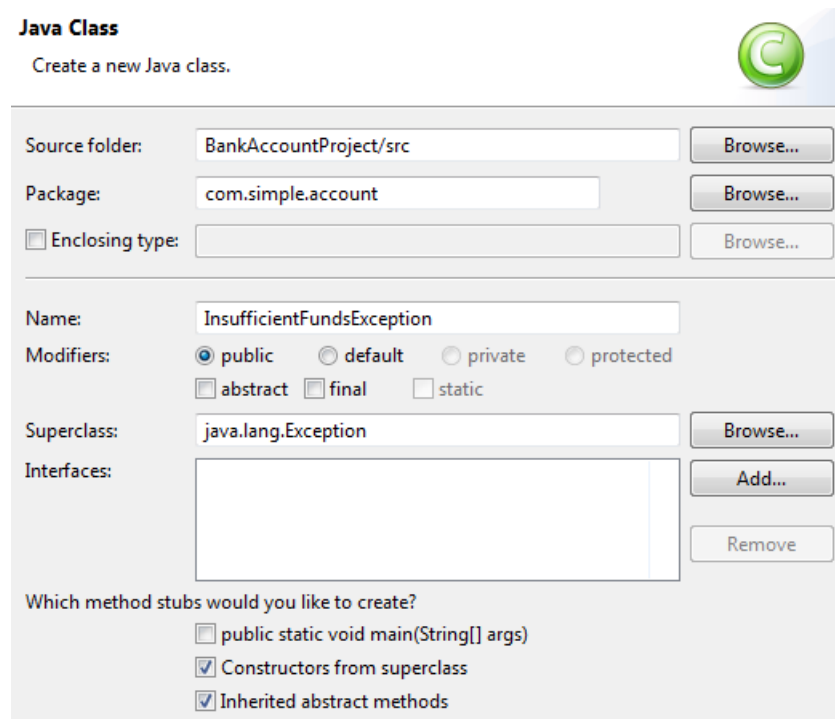
Part 2 - Creating Your Own Exception

In addition to using the exceptions provided by Java itself, it is possible to create your own exception class to handle situations specific to your program. We will do this now.

Firstly, we will add a new method on our BankAccount called **withdraw**, which will take one parameter: the *amount* to be withdrawn. This method will have some very simple error checking code; if the amount to be withdrawn is larger than the current balance, an **InsufficientFundsException** will be thrown. We will create the InsufficientFundsException class ourselves.

We will start things off by creating our exception class. An exception class can, at its simplest, merely extend the **java.lang.Exception** class. Recall that extending another class implies inheriting all its functionality, including all its methods.

- __ 1. In the *Package Explorer*, right click on **BankAccountProject** and select **New | Class**.
- __ 2. In the *New Java Class* wizard, set the *Package* to be **com.simple.account**
- __ 3. Set the *Name* to be **InsufficientFundsException**
- __ 4. Change the *Superclass* to be **java.lang.Exception**
- __ 5. If it is checked, uncheck *public static void main(String args[])*.
- __ 6. Check the box marked **Constructors from superclass**



Java Class
Create a new Java class.

Source folder: BankAccountProject/src Browse...

Package: com.simple.account Browse...

☐ Enclosing type: Browse...

Name: InsufficientFundsException

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Exception Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?
☐ public static void main(String[] args)
☒ Constructors from superclass
☒ Inherited abstract methods

__7. Click **Finish**.

The class should be created and the editor will open on it.

```
package com.simple.account;

public class InsufficientFundsException extends Exception {

    public InsufficientFundsException() {
        // TODO Auto-generated constructor stub
    }

    public InsufficientFundsException(String arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public InsufficientFundsException(Throwable arg0) {
        super(arg0);
        // TODO Auto-generated constructor stub
    }

    public InsufficientFundsException(String arg0, Throwable arg1) {
        super(arg0, arg1);
        // TODO Auto-generated constructor stub
    }

    public InsufficientFundsException(String arg0, Throwable arg1,
        boolean arg2, boolean arg3) {
        super(arg0, arg1, arg2, arg3);
        // TODO Auto-generated constructor stub
    }

}
```

Notice that a few constructors have been created for us. This was done by the wizard (by checking the *Constructors from superclass* checkbox).

And our exception class is done! We do not actually have to add any code here; all the required functionality will be inherited from the superclass.

Now, we can add the method to our BankAccount.

__8. Open **BankAccount.java**.

__9. Add the following method to the class:

```
public void withdraw(float amount) throws InsufficientFundsException {
    if (amount > this.getBalance() ) {
        throw new InsufficientFundsException("Amount " +
            amount + " exceeds balance " + this.getBalance());
    }
    this.setBalance(this.getBalance() - amount);
}
```

A few things to note here. Firstly, note that the method declaration has a **throws** clause. This specifies that any code trying to use this method must be aware that an **InsufficientFundsException** might occur and that it **must** be handled. This is a good way of forcing any code that calls this to have proper error handling logic in place.

Secondly, note the error checking logic; we check to see if the amount to be withdrawn is larger than the balance; if it is, a new instance of our **InsufficientFundsException** is created and **thrown**. Note that the constructor of the class takes a single string as a parameter, and the string is simply a message saying what the problem is.

Finally, the code goes ahead and decrements the balance. Note that this code will *not* be executed if the exception is thrown.

__10. Save the code. There should be no errors.

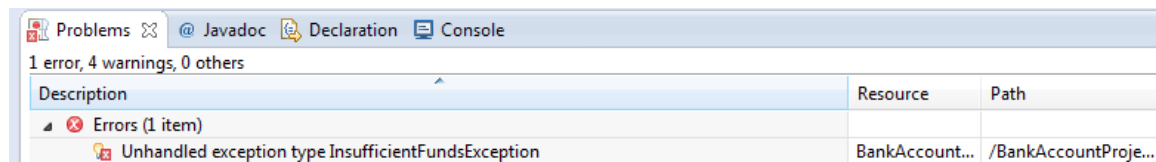
Now, let us test this method.

__11. Open **BankAccountTester.java**.

__12. Add the following code at the end of the **main** method.

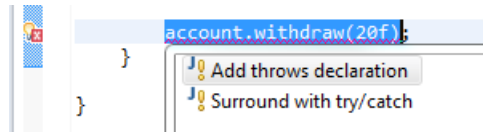
```
account.withdraw(20f) ;
```

__13. Save the file. Look at the *Problems* view. What is happening?



Eclipse is complaining that the line of code we just entered does not handle the `InsufficientFundsException`. Recall that our **withdraw** method has a **throws** clause in its declaration; this means that any code calling this method must catch this exception in case of an error. This is forcing our `BankAccountTester` class to **try** / **catch** this code.

__ 14. Left click on the red **X** in the margin next to this line.



__ 15. In the box that appears, double click **Surround with try/catch**.

Eclipse adds some code for us.

```
try {
    account.withdraw(20f);
} catch (InsufficientFundsException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Specifically, eclipse has added a **try** statement, and a **catch** statement that is trying to catch our **InsufficientFundsException**. All we have to do is put our error handling code in there.

Remember: the code in this **catch** clause will only be executed if the code within the try block throws the appropriate exception.

We will change the code inside the catch block; instead of printing the stack trace, we want to see the *message* of the exception that was caught.

__ 16. Locate the line:

```
e.printStackTrace();
```

__ 17. Delete it. In its place, enter:

```
System.out.println("There was an error withdrawing funds");
System.out.println(e.getMessage());
```

Note that we use the variable `e`. `e` is the exception object; it is declared in the **catch** clause. It refers to the actual exception object which has been thrown, and we can get information from it. Here we get the *message* from the exception. Where does this *message* come from? Simple: it was passed into the constructor of the exception instance, in our **withdraw** method.

__18. Save the file. There should be no errors.

Part 3 - Run the Code and Trigger the Exception

__1. Let us see what this code now does. Run it.

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
An account with id 3 with balance 1.0 owned by Donny K with interest rate 0.0 and minimum balance 1000.0
```

Nothing interesting; pretty much the same as before. The **withdraw** is working properly because we are trying to withdraw 20.0 from account 1 which had a balance of 100.0

Let us trigger a defect by changing the amount to be withdrawn.

__2. Locate the line:

```
account.withdraw(20f);
```

__3. Change it to:

```
account.withdraw(200f);
```

This should trigger a defect, which our code should catch.

__4. Save and run the code.

```
An account with id 1 with balance 100.0 owned by Jeff Lebowski
An account with id 2 with balance 5000.0 owned by Bunny Lebowski
An account with id 3 with balance 1000000.0 owned by Walter Sobcheck
An account with id 3 with balance 1.0 owned by Donny K with interest rate 0.0 and minimum balance 1000.0
There was an error withdrawing funds
Amount 200.0 exceeds balance 100.0
```

It looks like our error handling code was indeed executed. The **withdraw** method noticed an error condition and raised an appropriate exception; the BankAccountTester which was using the method caught that exception and then handled it gracefully if rather simplistically. (Better error handling code might ask the calling class to re-attempt the deposit with a smaller amount)

Congratulations! You have created your own exception class and seen how to throw/handle it.

__5. Close all open files.

Part 4 - Review

In this lab, you examined exceptions. You saw how code can be surrounded with a **try** block and a particular exception can be caught with a **catch** block. You also saw how to create your own exception class; finally, you saw how declaring a **throws** clause in a method declaration can force any calling code to handle a potential exception scenario.

Lab 22 - Project - Better Error Handling (Optional)

Time for lab: 45 minutes

This lab will continue the project. One thing that has been difficult in the project for a while has been how to handle errors. As the program becomes more complex and there are more error conditions to look for this becomes increasingly difficult. This has mainly been due to the fact that we have not used the mechanism of Java designed to handle errors, the Exception.

In this part of the project you will improve the error handling of the program by allowing it to work with Exceptions. You will first create a class to represent errors that may occur in the stock tracker program. You will then add code to “throw” and “catch” these errors in various parts of the program.

Part 1 - General Requirements

- Create a new class in the 'com.stock' package called `StockException`. When creating the class change the superclass to 'java.lang.Exception'. Use the option in the wizard to automatically create the constructors for you. Make sure there are several constructors created in the code generated for the `StockException` class.
- Modify the method signatures of the 'sellStock' and 'buyStock' methods of the `StockAccount` class to indicate that these methods can “throw” a `StockException`. You will have compilation errors until you finish other requirements.
- Modify the code of the 'sellStock' and 'buyStock' methods so that rather than printing out an error directly a new `StockException` is thrown with the error message. Any return statements you might have after the code to throw the exception might need to be removed as throwing the exception will halt the method.
- Modify the code of the `StockTracker` class to “try” to execute the methods on the `StockAccount` class and “catch” the Exception if it is thrown. If an exception is thrown, print out some type of message that there has been an error and include the detailed message that is part of the exception. Once you have "caught" all of the places where a `StockException` might get thrown your code should compile and run.
- Have the `StockTracker` class print the details of the transactions instead of the `StockAccount` class. This should remove the last of the 'System.out.println' code from 'sellStock' and 'buyStock' methods. This will be a better design as it will allow these operations to be independent of displaying the results to the user.

- **Bonus:** Add code to the StockTracker class to test that the input provided by the user can be read as the type expected. You can use the various 'hasNextXXX' methods of the Scanner class to “read ahead” and see if the input is the correct type. If it is not, inform the user of this and prompt them again. Note that the 'hasNextXXX' methods do not advance the input so you will need to add a call to other Scanner methods like 'next' to advance the input.

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- There are a few different ways to add the try/catch code. You can surround just the methods that throw the exception or you can surround a larger block of code to have fewer try/catch blocks. However you do it, remember that the code “skips” any other lines after the exception is thrown until it reaches the 'catch' block. Be sure your program still functions as expected when this happens.
- You may be able to use the development tool to automatically add appropriate try/catch code but be sure you know what code it has added and make sure it is appropriate for how you want the program to function.
- If the cases of the 'switch' statement for the various operations available to the user are becoming fairly complex you might look for ways to refactor that code out into separate methods in the StockTracker class that are called to perform the operation (if you haven't done this already). If you do this with the development tool make sure you carefully select what lines of code should be extracted to the new method.

Part 3 - Sample Output

The normal output is similar to previous parts of the project since exceptions are only thrown for errors.

The sample output for error conditions is also similar since the Exception simply communicates internally about the error.

```
Your account details:  
Name: Bob Broker  
Account Balance: 4000.0  
You do not own any stock.
```

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

45

Please enter the price of the stock and hit <ENTER>

123.56

Is the stock a dividend stock? (y/n)

n

You can't buy that much stock.

Your account details:

Name: Bob Broker

Account Balance: 4000.0

You do not own any stock.

Part 4 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- How has having the StockAccount class throw exceptions and the StockTracker class catch them improved the code? Why is it better for the StockTracker class to decide what to do when an error occurs?
- Having only one custom exception class really only allows one type of error to be “thrown”. What would be required if you had several types of errors that you wanted to be able to handle? What more specific types of errors might the program be expected to handle? Think about the different conditions under which the exceptions are thrown. In what ways might the program be able to adjust to these types of errors rather than just printing out “there has been an error”?

Lab 23 - Interfaces

Time for Lab: 30 minutes

In this exercise, you will examine the use of Java *interfaces*.

So far, when dealing with methods, we have seen that Java is a typed language. When you write a method that takes parameters, you specify the parameter types. Similarly, when a method returns an object, that object type is specified in the method signature.

Consider the following two methods, taken from our BankAccount class.

`public String getOwnerName()` - the method signature here indicates that a **String** is returned

`public void setOwnerName(String newName)` - here, we state the argument must be of type String

In either case, not using the correct type would cause a problem. Consider:

`int result = account.getOwnerName();` would fail because we are attempting to assign an int to a String return.

`account.setOwnerName(2333);` would fail because we are passing an int instead of a String.

Both of these statements would cause compile time errors, and the code would never run.

We can see that typing enforces a degree of correctness in code; after all, if we force the correct object type into a method signature, it is less likely that careless Java coding can occur.

However, there are some cases where we want some degree of flexibility; it might be desirable for a method to accept different types. We will see how this is possible using *interfaces*.

Let us consider a completely new class to add to our BankAccount model. Think of an **AccountManager** class. An **AccountManager** is essentially just a holder for an account. It has one field (called **account**) which represents the account it manages. It can perform operations on the account (like managing storage, batch updating, etc) and even broker withdrawals. We can say that the **AccountManager** will act as a public interface to the account that it holds.

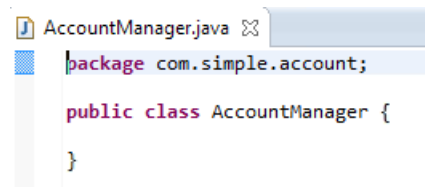
Right now, to perform deposits and withdrawals on an account, we are accessing the `BankAccount` directly; for encapsulation purposes, we will want our `AccountManager` to act as the front end for that. An end user could operate an ATM teller, and the ATM teller would operate on an instance of an `AccountManager`. The `AccountManager` would then pass deposit/withdrawal requests to the `BankAccount` instance that it manages.

We will use this class as a basis for our *interface* experiments. Let us create this class.

Part 1 - Create the AccountManager Class

__1. Create a new class in the **BankAccountProject** and call the class **AccountManager**. It should belong to the package **com.simple.account**. It should *not* have a **main** method, make sure *Constructors for superclass* is unchecked.

The class should look like this.



```
AccountManager.java
package com.simple.account;

public class AccountManager {
}
```

__2. Add a field called **account** of type **BankAccount** as follows:

```
private BankAccount account;
```

__3. Generate getter/setter methods for this field.

```
public BankAccount getAccount() {
    return account;
}

public void setAccount(BankAccount account) {
    this.account = account;
}
```

__4. Now, add the deposit/withdraw methods.

```
public void deposit(float amount) {
    this.getAccount().deposit(amount);
}

public void withdraw (float amount) throws InsufficientFundsException {
    this.getAccount().withdraw(amount);
}
```

Note that these methods are just *wrappers* for the deposit/withdraw methods on the account themselves. These methods do not actually have any withdraw/deposit business logic, but rather delegate it to the account itself.

Remember, this **AccountManager** class is essentially a public wrapper class for our old **BankAccount**.

__5. Save the code. There should be no errors.

Part 2 - Test the AccountManager

We will now create a tester class to make sure our new **AccountManager** works.

__1. In the **BankAccountProject**, create a class called **AccountManagerTester** that belongs to the package **com.simple.test** and it should have a **main** method.

__2. Add the following code to the **main** method. (Delete the **//TODO** comment)

```
AccountManager manager = new AccountManager();
BankAccount account = new BankAccount(1, "Jeff Lebowski", 1000f);
```

We create a new instance of a **BankAccount** as well as an **AccountManager**.

__3. Perform an “organize imports” (**Ctrl-Shift-O**).

__4. Continue adding the following code:

```
manager.setAccount(account);
```

Remember that the manager will act as a broker for our account; accordingly, we set the account as an instance of the manager.

Now, we can actually perform withdrawals and deposits via the manager.

__5. Add the following code to test this.

```
manager.deposit(400f);
try {
    manager.withdraw(20f);
} catch (InsufficientFundsException e) {
    System.out.println(e.getMessage());
}
```

We perform a simple deposit and then a withdrawal. Notice that the **withdrawal** invocation has to be wrapped with a **try/catch**. (Look at the method signature for **withdrawal** in the **AccountManager** class; it has the **throws** clause.)

__6. Finally, let's add some output.

```
System.out.println("Managing " + manager.getAccount());
```

__7. Perform an “organize imports” (Ctrl-Shift-O).

__8. Save and run the code. The output should be expected:

```
Managing An account with id 1 with balance 1380.0 owned by Jeff Lebowski
```

Looks like it worked.

We can now use the AccountManager to handle all operations on the BankAccount. This might seem strange; after all, why add this extra level of abstraction? What was wrong with accessing the BankAccount directly?

The reasons are twofold: firstly, this AccountManager could act as a **UI** (user interface) class which is capable of interacting with the end user (drawing menus, handling button presses, etc) while leaving the actual account logic on the BankAccount itself. While we will not actually be doing this, it is a common practice.

Secondly, this AccountManager will act as a perfect example of using *interfaces*, which is the whole reason we are performing this lab exercise.

Part 3 - Changing The AccountManager

We've seen that the **AccountManager** can work with instances of **BankAccounts**. But can it work with **SavingsAccounts**? Let's try this.

__1. Open the **AccountManagerTester.java**.

__2. Locate the line:

```
BankAccount account = new BankAccount(1, "Jeff Lebowski", 1000f);
```

__3. Change it to:

```
SavingsAccount account = new SavingsAccount(1, "Jeff Lebowski", 1000f);
```

What we are doing here is making the **AccountManager** handle an instance of a **SavingsAccount** instead of a regular **BankAccount**.

__4. Organize imports.

__5. Save the code. There should be no errors.

__6. Run the code. The output returns:

```
Managing An account with id 1 with balance 1380.0 owned by Jeff Lebowski with interest rate 0.0 and minimum balance 1000.0
```

Looks like it worked! So, the **AccountManager** can work with **BankAccounts** and **SavingsAccounts**.

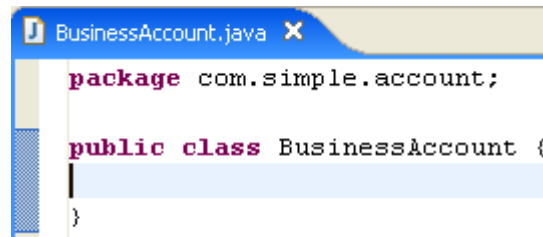
This is possible because the **BankAccount** and **SavingsAccount** are related by inheritance; since a **SavingsAccount** is a subclass of **BankAccount**, any method that can work with a **BankAccount** can work with a **SavingsAccount**.

Let us consider, however, another type of account. Say we now wish to have a new account type called **BusinessAccount**. A **BusinessAccount** is an account that is used by corporate entities. It is similar to the regular **BankAccount**, in that it has a **balance**, and an **ID**; however it will not have an **ownername**; instead it will have **companyName** and **companyAddress** fields.

Could we model this class as subclass of BankAccount? Unfortunately, not; because our BusinessAccount does not have an **ownerName**, forcing it to inherit would not make for good object oriented behavior. So, it will not be able to belong to the same inheritance hierarchy. Let us go ahead and create this class.

Part 4 - Create the BusinessAccount Class

__1. In the **BankAccountProject**, create a new class called **BusinessAccount**, belonging to package **com.simple.account**. Leave it as extending **java.lang.Object** and do not create a **main** method for it.



```
BusinessAccount.java X
package com.simple.account;

public class BusinessAccount {
    |
}
```

__2. Add the following fields:

```
private float balance;
private int accountID;
private String companyName;
private String companyAddress;
```

__3. Generate getters/setters for these fields.

__4. Create a constructor for this class. The best way to do this is to right click anywhere on the code editor and select **Source | Generate Constructor using Fields** . Click the **Select All** button and then click **Generate** in the *Generate Constructor using Fields* window that appears.

__5. Add the **toString** method as follows:

```
public String toString() {
    return "A BusinessAccount belonging to " + this.getCompanyName()
        + " with balance " + this.getBalance();
}
```

__6. Add the deposit method as follows:

```
public void deposit(float amount) {
    this.setBalance(this.getBalance() + amount);
}
```

__7. Finally, add the withdraw methods as follows: (If you want, you can copy and paste it from the `BankAccount` class to save yourself some typing.)

```
public void withdraw(float amount) throws InsufficientFundsException {
    if (amount > this.getBalance() ) {
        throw new InsufficientFundsException("Amount " + amount + " exceeds
            balance " + this.getBalance());
    }
    this.setBalance(this.getBalance() - amount);
}
```

__8. Save the code. There should be no errors.

We have now created a new class; even though it is semantically similar to the **BankAccount** and **SavingsAccount** classes, it was not similar enough in implementation to warrant being a subclass of either of those two classes.

Part 5 - Managing the **BusinessAccount**

We want the **AccountManager** to still be able to manage a **BusinessAccount**. Can this be done? Let us try this.

__1. Open **AccountManagerTester.java**.

__2. Locate the line:

```
SavingsAccount account = new SavingsAccount(1, "Jeff Lebowski", 1000f);
```

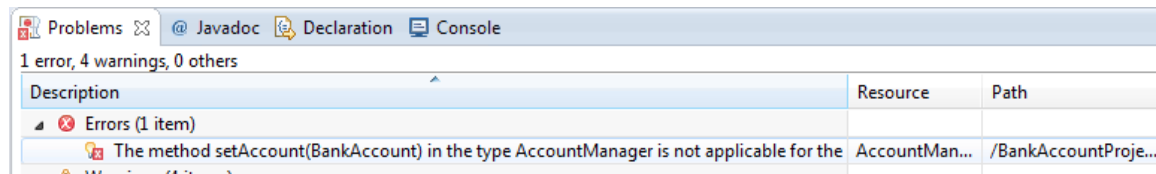
__3. Change it to the following:

```
BusinessAccount account = new BusinessAccount(1000f, 1, "SimpleCorp",
    "123 Fake Street");
```

Note here that we use the 4 argument constructor for the **BusinessAccount**.

__4. Organize imports.

__5. Save the code. Look at the *Problems* view. There should be one error:



The problem is in the **setAccount** call. **setAccount** expects an instance of a **BankAccount** or one of its subclasses; we are passing in a **BusinessAccount** which is neither of those. Java's *type* behavior is now causing a problem.

So what can we do? We already said that we cannot make the **BusinessAccount** a subclass of the **BankAccount** hierarchy. We could try *method overloading*; this means writing duplicate methods for each type (e.g. **setAccount(BankAccount)** as well as **setAccount(BusinessAccount)** but that would lead to unnecessarily complicated code.

A better idea is to use an *interface*.

(Ignore this error for now; it will be fixed shortly.)

Part 6 - Examining Interfaces

An interface is an abstract definition of a class. An interface has no code, but instead lists several methods and fields. A class can declare that it conforms to an interface by **implementing** it.

Think of an interface as a contract. You want to ensure that classes behave a certain way by having certain methods, but you do not want to use inheritance. You can write this contract to stipulate that any implementing class has code for those methods (but you do not actually specify the logic for the methods; just the names). So if a class implements an interface, it conforms to the contract.

This helps us because we can use an interface in a method signature.

Think about our account problems. The **AccountManager** needs some form of account. But really, according to the **AccountManager**, all an account really is is a class that has a **deposit** and a **withdrawal** method. Our **BankAccount**, **SavingsAccount** and **BusinessAccount** all conform to this.

What we can now do is write an interface to represent such an account. Then we can change our **AccountManager** to operate on the *interface* instead of the actual object type.

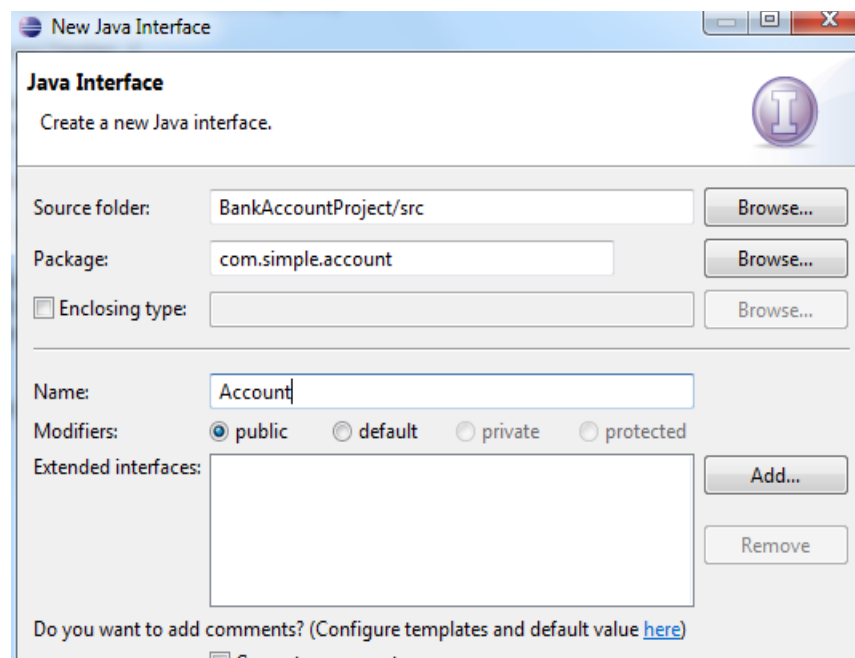
So, what we will do is this: we will firstly create an *interface* (called **Account**) that has two operations: **deposit** and **withdraw**. We will then change the **BankAccount** and **BusinessAccount** classes to say they **implement** the **Account** interface. Finally, in the **AccountManager** class, we will change the method signature of the **setAccount()** method to use our new more general interface **Account** instead the more specific **BankAccount** class. Our tester code should then work; the **AccountManager** will be able to use any of our **Account** types.

Let us start by creating the interface.

__1. In the *Package Explorer*, right click on **BankAccountProject** and select **New | Interface**. The *New Java Interface* window will appear.

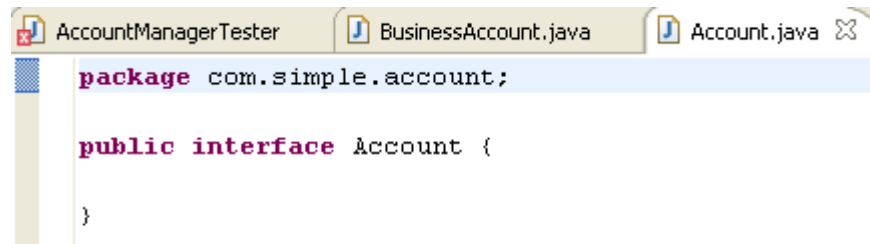
__2. For the *Package*, specify **com.simple.account**

__3. For the *Name*, specify **Account**



__4. Click **Finish**.

The code editor will open on our new interface.



Remember, an interface is just a contract that outlines what methods an implementing class must provide code for. It does not provide code itself.

__5. Add the following two method declarations to the interface.

```
public void withdraw(float amount) throws InsufficientFundsException;
public void deposit(float amount);
```

__6. Save the code. There should be no errors in this class.

Our contract is complete.

We can now state that our classes (**BankAccount** and **BusinessAccount**) conform to this contract.

__7. Open **BankAccount.java** and change the class declaration to the following:

```
public class BankAccount implements Account{
```

All we do here is add the **implements** keyword and specify what interface this class is implementing.

__8. Save the code.

__9. Now, open **BusinessAccount.java**. As before, change the class declaration to the following:

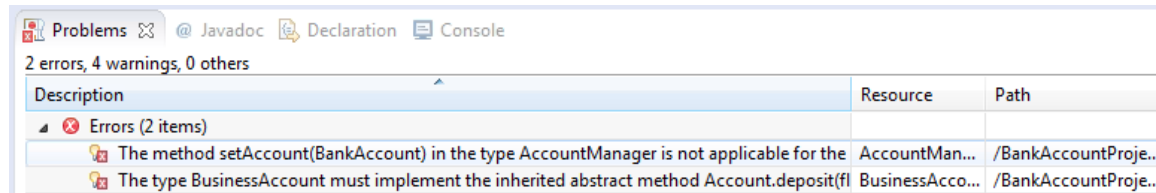
```
public class BusinessAccount implements Account {
```

What exactly is the implication of doing this? The implication is that these **BusinessAccount** and **BankAccount** classes *must* have code for the **deposit()** and **withdraw** method. By doing so, they can say that they conform to the contract specified by the **Account** interface.

__10. Test this. In the **BusinessAccount.java** file, locate the method **deposit()** and rename it to something else (like **deposit2()**).

```
public void deposit2(float amount) {
```

__11. Save the file. If you then look at your *Problems* view, a new error appears.



This is showing that unless the methods specified on the interface exist in the implementing class, Java will throw an error. This forces us to conform to the contract.

__12. Undo the change (i.e. rename the method back to **deposit()**) and save the file. This error should go away.

Part 7 - Update the AccountManager

We can now tell the **AccountManager** class to use the more generic **Account** interface in its method signatures.

__1. Open **AccountManager.java**.

__2. Locate the field declaration:

```
private BankAccount account;
```

__3. Change it to use the interface as follows:

```
private Account account;
```

__4. Now, locate the code for the getter method:

```
public BankAccount getAccount() {
```

__5. Change its return type as follows:

```
public Account getAccount() {
```

__6. Finally, locate the code for the setter method:

```
public void setAccount(BankAccount account) {
```

__7. Change the parameter declaration as follows:

```
public void setAccount(Account account) {
```

__8. Save the file.

There should be no errors. Additionally, the error that we had before (when we first switched the **AccountManagerTester** to use a **BusinessAccount**) should be gone.

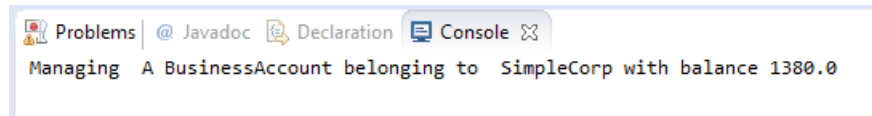
What have we done here? Simple. We have changed the **AccountManager** to no longer work on instances of **BankAccount**. Instead, we have coded it to use any class that implements the **Account** interface – which is precisely a category that our **BusinessAccount** and **BankAccount** both fit into.

Note that in the **deposit** and **withdraw** methods, our **AccountManager** is still able to invoke the **deposit** and **withdraw** methods of the field; this is because the field **account** is an implementation of our **Account** interface – which we know for a fact will have **deposit** and **withdraw** methods on it.

__9. Switch back to the **AccountManagerTester**. There should be no errors here.

Note. If you see a red underline on **setAccount**, close the class and open it again, then the error will gone.

__10. Run the code.



Everything works. The **AccountManager** is working with the **BusinessAccount**, as it did with the **BankAccount** before it. Switching to an interface has given us flexibility.

__11. Close all open files.

Part 8 - Review

In this exercise, we examined the concept of interfaces. An interface is a contract; a class that **implements** an interface is guaranteed to conform to the contract (i.e. has code for methods specified in the contract).

We saw how interfaces can help with class 'compatibility'. Even though classes may not be related by an inheritance relationship, they might both have a similar functional appearance, and that common appearance can be expressed using an interface.

Method signatures can then be changed to use the interface for return and parameter types, allowing for greater flexibility and cleaner code.

Lab 24 - Collections

Time for Lab: 30 minutes

In this lab exercise, you will examine Java *collections*.

Recall that in an earlier lab, we examined arrays. An array was a sequential series of objects of the same type. We could put elements of the same type into an array, and access them afterwards. Using an array, we could maintain a *collection* of objects.

Let us re-visit our **AccountManager** class. Right now, it only manages a single account instance (via a field called **account**). What if, however, we want one **AccountManager** to manage multiple account instances? We could change the field declaration from being a simple **Account** to an *array* of accounts. Then we could add accounts to an **AccountManager**'s array and it manage the array.

Arrays, for all their good are actually quite primitive. While they can serve as a simple collection, they have some problems. Firstly, the size of an array must be known at declaration time. So, when we initialize our class, we would have to specify how many accounts the AccountManager could handle – and that size would be fixed.

Also, arrays have a very simple API; the only operations you can perform on an array are the indexing operations. If you wanted to search an array for a particular element, or sort it, you would have to write the code yourself. So, while arrays can be handy, they are a little primitive. Fortunately for us, Java has the *collection API*.

Java itself comes with a variety of collection classes (e.g. **Vector**, **HashMap**, **ArrayList**, etc) to handle our many collection needs. They all work more or less the same way; a collection is just a container for multiple object instances. Different collections, however, offer different features; some are sorted; some are indexed; some are optimized for different operations.

In this lab, we will examine one such collection: the **HashMap**.

Part 1 - Revise the AccountManager

As we have stated before, we want to change the AccountManager to handle a collection of accounts instead of just one. What effect will this have on the AccountManager's external API?

Firstly, the **setAccount** method will be removed. Instead, we will have a method called **addAccount** which takes as parameter, an instance of an **Account**.

Secondly, the **getAccount** method will change; currently it just returns the one account associated with the **AccountManager**. We will have to update it so it can return a particular account instance.

Thirdly, the **deposit** and **withdraw** methods will have to be beefed up a bit. Right now, they only operate on a single parameter, which is the amount. We still want these functions to work, but we will now have to pass in a new parameter; the account to be updated. (Remember, our **AccountManager** will be tracking multiple accounts)

Immediately, we have to decide how we are going to identify account instances. What piece of information can we use to query an **AccountManager** to obtain a particular account instance?

How about the ID? We could specify that all accounts are located by their ID. The API would then take that into account. When calling **getAccount()** an ID would be passed in as a parameter, and the appropriate instance would be returned. Also, when calling **deposit/withdraw**, the **id** of the account to be updated would be passed in, along with the amount. Some sample API:

```
manager.getAccount(1);
```

This would tell the manager to retrieve the account instance with ID 1.

```
manager.deposit(1, 100);
```

This would tell the manager to deposit 100 into account with ID 1.

Let us start coding these changes!

__ 1. Open **AccountManager.java**.

We will first change the declaration of the field to be a **HashMap** instead of a regular account.

__ 2. Locate the line

```
private Account account;
```

__3. Delete it. In its place, add the following code.

```
private HashMap<Integer, Account> accounts =  
    new HashMap<>();
```

As mentioned earlier, **HashMap** is a Java-provided collection class. It has some interesting API that we will use.

This code also uses “Generics” to qualify exactly how this Collection will be used. The code indicates that Integer objects will be used for keys and Accounts will be stored as values.

Notice we initialize the variable right away. Whenever this class is instantiated, this **accounts** variable will be initialized into being a new empty **HashMap**.

__4. An error will appear because **HashMap** has not been imported. Perform an organize imports.

__5. Delete the getAccount() and setAccount() methods. They will not be used any more.

__6. We now need a method to add an account to the **HashMap**. Add the following method.

```
public void addAccount(Account account) {  
    int key = account.getAccountID();  
    this.accounts.put(key, account);  
}
```

This code is interesting. Remember that **this.account** is referring to the field **HashMap**.

HashMaps do not use indexes per se. Remember that arrays could be traversed by using the **[index]** operation. **HashMaps** use *keys* instead of indexes. A key can actually be any object value, not just an **int**. To locate (or place) a particular instance, we use a key. So, similar to an array we can say to a **HashMap**, “find me the object at *key*”

Remember that we specified that we want to use an account's ID as its key. So an account with ID 1 could be found “at” position 1 and account with id 2 could be found “at” position 2, and so on.

This method first extracts the passed-in account's ID and saves it as an **int**. The code then inserts the account into the **HashMap**. Since the **HashMap** was created using Generics, Java can automatically “autobox” the **int** into an **Integer** when storing in the Collection.

Note that this insertion is done using the HashMap's **put()** method, which takes two parameters: the key, and the object to place at the key. In this case, the account will be placed 'at' the key indicated by the account id. This is a part of the collection API.

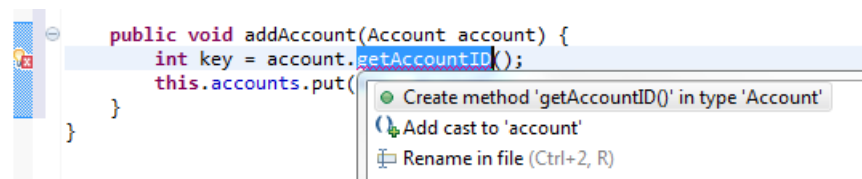
__7. There is still an error on this method. If you float the mouse cursor over the red X in the margin next to the line

```
int key = account.getAccountID();
```

You will see a message stating that method **getAccountID()** is not understood. Why is this happening? Simple – the account that is passed into this method is actually an instance of the **Account** interface. Recall that our interface only has 2 methods: **deposit** and **withdraw**. Now, we are trying to call a method **getAccountID()** on it; this means we will have to add the **getAccountID()** method to the interface.

Let us add the method in a quick way.

__8. Click the red X in the margin you were just floating the mouse over. A box appears.



__9. In the white box, double click the line **Create method 'getAccountID' in type 'Account'**

__10. A code editor will open on the **Account** interface. Notice a new method has been added:

```
package com.simple.account;

public interface Account {
    public void withdraw(float amount) throws
        InsufficientFundsException;
    public void deposit(float amount);
    public int getAccountID();
}
```

__11. Save the file.

__12. Switch back to the **AccountManager.java** class. Save the file. There should now be no errors in the **addAccount()** method. Fortunately, the two implementors of the class (**BankAccount** and **BusinessAccount**) already have this **getAccountID()** method, so there is no need to actually add them.

__13. We now need to have a method that allows for the retrieval of an **Account**, based on a passed-in account ID parameter. Add the following method:

```
public Account getAccount(int id) {  
    return (Account) this.accounts.get(id);  
}
```

Notice the use of **this.accounts.get(id)**. This is the actual collection API; this is telling the **HashMap** to return the object 'at' the “key” indicated by the id. Since the **HashMap** was declared using Generics, the id can be passed in directly and Java will know to “autobox” the value into an **Integer** object.

Note also that since we defined the accounts reference using generics, the compiler knows that this **get(...)** method will return an instance of some class that implements the **Account** interface. We do not need to perform an explicit cast on the return value from **get(...)**, so we can just return the value directly.

__14. Now we can tackle the **deposit** and **withdraw** methods. Locate the two following methods:

```
public void deposit(float amount) {  
    this.getAccount().deposit(amount);  
}  
  
public void withdraw (float amount) throws InsufficientFundsException {  
    this.getAccount().withdraw(amount);  
}
```

__15. Delete the methods.

__16. Now, add the following two methods:

```
public void deposit(int id, float amount) {
    Account account = this.accounts.get(id);
    account.deposit(amount);
}

public void withdraw (int id, float amount) throws
InsufficientFundsException {
    Account account = this.accounts.get(id);
    account.withdraw(amount);
}
```

These two methods work in the same way. They take two parameters as input. One is the ID of the account to be deposited into/withdrawn from and the other is the amount. They both use the **get()** method of the **HashMap** to retrieve the actual **Account** instance.

Again, notice we can pass the id directly into the call to the 'get' method because of Generics and autoboxing. We then call the appropriate **deposit/withdraw** method on the retrieved instance.

Finally, let us write a **toString** method that will allow us to see the accounts that an **AccountManager** is managing.

__17. Add the following method:

```
public String toString() {
    return this.accounts.toString();
}
```

This is pretty simplistic code; all it does is call the **toString()** method of the **HashMap**. All **toString()** method on the **HashMap** does is print out the contents. This isn't very elegant (or very readable) but will do the job for a quick validation.

__18. Our code is done. Save it. There should be no errors.

It should look like this:

```
package com.simple.account;

import java.util.HashMap;

public class AccountManager {

    private HashMap<Integer, Account> accounts =
        new HashMap<>();

    public void addAccount(Account account) {
        int key=account.getAccountID();
        this.accounts.put(key, account);
    }

    public Account getAccount(int id) {
        return this.accounts.get(id);
    }

    public void deposit(int id, float amount) {
        Account account= this.accounts.get(id);
        account.deposit(amount);
    }

    public void withdraw(int id, float amount) throws InsufficientFundsException {
        Account account= this.accounts.get(id);
        account.withdraw(amount);
    }

    public String toString() {
        return this.accounts.toString();
    }
}
```

We have completed our revised collection-savvy AccountManager. The next step is to test it.

Part 2 - Test the New AccountManager

We will now see if we can successfully add multiple Accounts to our AccountManager. We will also see if we can perform a deposit to a particular account.

- ___ 1. Open **AccountManagerTester.java**.
- ___ 2. Locate the **main** method and delete all the code inside it. We will write new test code from scratch.

__3. In the now empty **main** method, add the following code.

```
AccountManager manager = new AccountManager();
BankAccount account = new BankAccount(1, "Jeff Lebowski", 90f);
SavingsAccount sAccount = new SavingsAccount(2, "Maude Lebowski",
10000f);
BusinessAccount busAccount = new BusinessAccount(1000f, 3,
"SimpleCorp", "123 Fake Street");
```

Here, we set everything up. We create a new instance of an AccountManager and then an instance of a BankAccount, a SavingsAccount, and a BusinessAccount. Our next step will be to *add* them to the AccountManager.

__4. Organize imports.

__5. Add the following code:

```
manager.addAccount(account);
manager.addAccount(sAccount);
manager.addAccount(busAccount);
```

Here, we use the new AccountManager API to add the three accounts we have just created. Remember that the AccountManager API is using a **HashMap** (and its API) to actually store these instances.

__6. Now let us take a look at the contents of the AccountManager. Add the following code:

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

__7. Save the code. There should be no errors.

It should look like this:

```
public class AccountManagerTester {  
    public static void main(String[] args) {  
        AccountManager manager = new AccountManager();  
        BankAccount account = new BankAccount(1, "Jeff Lebowski", 90f);  
        SavingsAccount sAccount = new SavingsAccount(2, "Maude Lebowski",  
            10000f);  
        BusinessAccount busAccount = new BusinessAccount(1000f, 3,  
            "SimpleCorp", "123 Fake Street");  
  
        manager.addAccount(account);  
        manager.addAccount(sAccount);  
        manager.addAccount(busAccount);  
  
        System.out.println(manager);  
    }  
}
```

__8. Run the code. You should see some interesting results.

```
{1=An account with id 1 with balance 90.0 owned by Jeff Lebowski, 2=An  
account with id 2 with balance 10000.0 owned by Maude Lebowski with  
interest rate 0.0 and minimum balance 1000.0, 3=A BusinessAccount  
belonging to SimpleCorp with balance 1000.0}
```

This may look a little strange, but the information we are looking for is actually there. It looks strange because we are calling the **toString()** method of the **HashMap** – but that in turn is calling the **toString()** method of our accounts. The **toString()** method of the **HashMap** shows an open curly brace, and then a list of key/value pairs for every object in the **HashMap**. Note that the key for the **BusinessAccount** is 3, the key for the **SavingsAccount** is 2 and the key for the **BankAccount** is 1. This is the expected behavior.

So, it looks like our **AccountManager** can successfully add accounts.

Let us now try the **deposit** method of the account manager. We will deposit 100 into the **SavingsAccount** (which has an id of 2).

__9. In **AccountManagerTester**, delete the line:

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

__10. In its place, add the code:

```
manager.deposit(2, 100f);
```

This should deposit 100 into the SavingsAccount.

How do we know this worked? Simple: let us try using the **getAccount()** method to get the freshly deposited-into account 2 and print it out.

__ 11. Add the following code:

```
System.out.println(manager.getAccount(2));
```

Simple code. We call the **getAccount()** method and **println** it.

__ 12. Save the code. There should be no errors.

__ 13. Run the code.

```
An account with id 2 with balance 10100.0 owned by Maude Lebowski with interest rate 0.0 and minimum balance 1000.0
```

It worked! The balance was updated by a 100, and the call to **getAccount()** returned the correct instance, showing the new amount.

Congratulations! You have used the collection API!

__ 14. Close all open files.

Part 3 - Review

In this lab, we examined the collection API. Specifically, we looked at the **HashMap**. We changed our **AccountManager** class to use a **HashMap** to store a collection of **Account** instances instead of just a single **Account**. We saw API (methods **put** and **get**) to see how to insert objects into and retrieve objects from the **HashMap**. This is a good example of how you might typically use a collection class in your every day Java coding.

Lab 25 - Project – Multiple Stocks (Optional)

This lab will continue the project. Up until now you have added a lot of features to the StockTracker but it still only allows you to own one stock at a time. This is obviously not reasonable. Although you could have implemented this once you learned about arrays we did not have you do this since Collections can also be used to implement this functionality. Now that you know about both mechanisms we can discuss which might be best and implement that. Each approach can have advantages and disadvantages.

Part 1 - Implementation Discussion

This section provides some questions about how to implement multiple stocks that can be discussed with the class or thought about on your own. The answers to these questions should help point out the strengths and weaknesses of different ways to implement tracking multiple stocks.

- When you are implementing multiple stocks will the StockAccount or Stock/DividendStock class(es) change the most? Think about how currently the program "remembers" what stock is owned by the account.
- If the StockAccount class referred to an array of Stock objects how would that be implemented? What would need to happen before you can store a reference to a Stock in the array?
- Arrays have a fixed size. If you need to store more items in an array than it can hold you must create a new array and copy the elements from the old array into the new array. What implications would this have on the stock program?
- If you decided to start with an array of a size that was bigger than the number of stocks owned, what would the uninitialized elements of the array contain? How could this complicate other parts of the program?
- Right now the assumption is that if the user sells off all of the stock they own the 'null' value is used instead of setting the number of shares to zero. This has the effect of showing that the user doesn't own any stock when the account details are printed similar to when the account was first opened. Now that you can have multiple stocks how would we "sell off" all of a stock if an array were used?
- If you "sold off" stocks that were in the "middle" of the array between other stocks that are still owned what impact would this have on other parts of the program?

- One of the primary things you need to do when buying or selling stock is to figure out if the user already owns that stock. How would this be done if you were using an array?
- Overall what are the strengths and weaknesses of using an array to hold multiple stocks?
- How would you figure out if a user already owns a stock when using one of the types from the Collections framework? Is there something from this framework that would naturally let you check for a particular stock using the property that indicates the "identity" of the stock?
- How would you "sell off" a stock if using one of the options from the Collections framework?
- Overall what are the strengths and weaknesses of using some type of structure from the Collections framework? What things are provided for you automatically that you might have had to implement yourself with an array?

Part 2 - General Requirements

Hopefully through the questions in the previous section you came to the realization that since the number of stocks owned can vary and "selling off" a stock meant "nulling out" the reference from the StockAccount to the Stock object that an array was not going to be a very easy way to implement multiple stocks. Many features of the Collections framework will simplify the implementation of multiple stocks. Combine this with the fact that we can use the String of the stock symbol as a key to the Stock object and that Strings already have a "natural ordering" the choice of a SortedMap to store references to the stocks owned makes sense for the implementation of the multiple stocks feature.

- Modify the code of the StockAccount class so the instance variable that holds the stock information is a SortedMap and not a single object. You may also want to use the refactoring tools to rename the instance variable of the StockAccount class to a plural variable name (like 'heldStocks') to imply the change of owning multiple stocks. You will get compilation errors until you complete other requirements. You may also need to add an import statement since the SortedMap class is in the java.util package.
- Use generics to declare that the SortedMap should only contain String keys and Stock objects.

- Modify the constructor(s) of the StockAccount class to initialize the instance variable that tracks stocks to a new TreeMap. This implements the SortedMap interface. Use generics in the same way you did with the declaration of the instance variable to indicate that the TreeMap will hold String keys and Stock object values. Again you may need to add an import statement.
- Modify the return type (and also perhaps the name) of the method that was previously returning the Stock object referenced by the StockAccount. This 'getHeldStocks' method should now return a Collection<Stock> instead of a single Stock object. You will also need to call the 'values' method on the SortedMap instance variable to get a collection of just the Stock objects instead of the whole map with keys. Make sure this method compiles when you have made changes even though there may be other compile errors. Changing this method will also introduce errors in the StockTracker class but those will be fixed later.
- Comment out all of the code in the body of the 'sellStock' method but leave the method signature. This should let you ignore the compilation errors in that method for now so we can concentrate on the 'buyStock' method. Make sure that you comment out all of the method code correctly and after this the 'buyStock' method is the only place with compilation errors in the StockAccount class.
- In the 'buyStock' method remove any logic that was throwing a StockException if the user was trying to buy more than one stock as this will now be allowed.
- For right now change the implementation of the 'buyStock' method so that if the user has enough balance the stock to be bought is placed in the collection of stocks owned. Ignore for now what happens when the user buys a particular stock a second time. You will need to use the 'put' method on the SortedMap interface and use the symbol of the stock as the key.
- Make sure all of the code in the StockAccount class compiles. There will be errors in the StockTracker class. You will not have all features like selling or buying a stock multiple times implemented but we want to test the basic buy operation first.
- Open the StockTracker class and change the code that was printing out the details on the account to adjust to the fact that the 'getHeldStocks' method (or whatever you called it) is now returning a Collection<Stock> object. You will need to check if this collection is empty and if it isn't go through the entire collection and print details for every stock.
- Make sure that all of your code compiles and run the program just checking that you can buy stock.

- Go back to the 'buyStock' method in the StockAccount class and make adjustments so if a stock currently held is bought again the method can handle that situation. Remember that if the user has answered the 'dividend stock?' question differently between times buying stock that the program will use the type of stock based on the most recent purchase. You might use methods like 'remove', 'put', and 'containsKey' from the Map interface to implement the proper behavior.
- Check that you can compile and run the program and it now handles purchasing the same stock more than once correctly.
- Modify the code of the 'sellStock' method to use the SortedMap. The first thing you should be able to do easily is to check (using the 'isEmpty' method) if any stocks are owned and throw the appropriate exception if none are owned.
- Modify the code of the 'sellStock' method to use the symbol of the stock to be sold as the key and look to see if the stock is owned. Both the 'remove' and 'get' methods of the Map interface will take the key as a parameter and either return null if the stock is not owned or return the Stock object if it is. If the value returned is null the method should throw the appropriate exception that the stock is not owned.
- Now that you know the stock being sold is owned check that the number of shares being sold is not greater than the amount owned and throw the appropriate exception if this is the case. Make sure that if the user tries to sell too much stock the original amount they owned is retained.
- Finally handle the situations where the user is "selling off" all that is owned or less than what is owned. Remember that if all is "sold off" the account should have no link to that type of stock anymore and that you need to store the type of stock ("regular" or dividend) depending on the last answer to the 'dividend stock?' question. For both of these reasons you are likely to be using the 'remove' method of the Map interface.
- Finally, delete any commented out code that was from before you implemented multiple stocks to clean up the code and make it more readable. This should not change the fact that the program compiles and runs correctly.

Part 3 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- When using generics for a Map (or subclass) you need to declare the type of the key and the values that will be stored. These are separated by a comma inside the generic modifier.
- Now that the 'heldStocks' instance variable in the StockAccount class refers to the collection of stocks and this is always initialized to an empty collection when creating the StockAccount you will need a new way to test if there are any stocks owned. There is an 'isEmpty' method available that will help with this. 'heldStocks == null' is no longer what you want to test though.
- When modifying the 'printDetails' method remember that you can use a “foreach” loop to loop over all elements in a collection.
- Remember that before we were using a collection of multiple stocks the way you got the program to account for the fact the user might change their response to the 'dividend stock?' question was to "forget" about the reference to the previous stock object and store the new stock object constructed from their responses in the instance variable of the StockAccount class. Now that we are using a group of stocks the way to do this would be to remove the previous object and add in the new object after adjusting the number of shares appropriately.

Part 4 - Sample Output

Sample output from only buying a few different stocks:

```
This program will help you track information
about your investments.
```

```
Please enter your name and hit <ENTER>
```

```
Bob Broker
```

```
Please enter your initial account balance and hit <ENTER>
```

```
3000
```

```
Your account details:
```

```
Name: Bob Broker
```

```
Account Balance: 3000.0
```

```
You do not own any stock.
```

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

5

Please enter the price of the stock and hit <ENTER>

78.5

Is the stock a dividend stock? (y/n)

y

Enter the dividend value and hit <ENTER>

0.05

You have bought 5 shares of IBM at \$78.5 per share

Your account details:

Name: Bob Broker

Account Balance: 2607.5

You currently own the following stocks:

- 5 shares of IBM at \$78.5 per share, dividend of \$0.05

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>

MSFT

Please enter the number of (whole) shares and hit <ENTER>

10

Please enter the price of the stock and hit <ENTER>

45.98

Is the stock a dividend stock? (y/n)

n

You have bought 10 shares of MSFT at \$45.98 per share

Your account details:

Name: Bob Broker

Account Balance: 2147.7

You currently own the following stocks:

- 5 shares of IBM at \$78.5 per share, dividend of \$0.05
- 10 shares of MSFT at \$45.98 per share

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

q

Your account details:

Name: Bob Broker

Account Balance: 2147.7

You currently own the following stocks:

- 5 shares of IBM at \$78.5 per share, dividend of \$0.05
- 10 shares of MSFT at \$45.98 per share

Output that shows purchasing the same stock more than once. Notice the type of stock changes (first "regular" then dividend):

Your account details:

Name: Bob Broker

Account Balance: 3000.0

You do not own any stock.

You can (b)uy stock, (s)ell stock, or (q)uit

Enter the first letter of your choice above and hit <ENTER>

b

You will now buy stock for your account

Please enter the stock symbol and hit <ENTER>

IBM

Please enter the number of (whole) shares and hit <ENTER>

5

Please enter the price of the stock and hit <ENTER>

89.56

Is the stock a dividend stock? (y/n)

n

You have bought 5 shares of IBM at \$89.56 per share

Your account details:

Name: Bob Broker

Account Balance: 2552.2

You currently own the following stocks:

- 5 shares of IBM at \$89.56 per share

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>
b
You will now buy stock for your account
Please enter the stock symbol and hit <ENTER>
IBM
Please enter the number of (whole) shares and hit <ENTER>
12
Please enter the price of the stock and hit <ENTER>
105.76
Is the stock a dividend stock? (y/n)
y
Enter the dividend value and hit <ENTER>
0.05
You have bought 12 shares of IBM at \$105.76 per share

Your account details:
Name: Bob Broker
Account Balance: 1283.0799999999997
You currently own the following stocks:
- 17 shares of IBM at \$105.76 per share, dividend of \$0.05

Output showing "selling off all stock. The behavior should be the same even if other stocks are still owned.

Your account details:
Name: Bob Broker
Account Balance: 2277.96
You currently own the following stocks:
- 6 shares of IBM at \$98.34 per share, dividend of \$0.05
- 10 shares of MSFT at \$45.98 per share

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>
s
You will now sell stock from your account
Please enter the stock symbol and hit <ENTER>
IBM
Please enter the number of (whole) shares and hit <ENTER>
6
Please enter the price of the stock and hit <ENTER>
123.65
Is the stock a dividend stock? (y/n)
n
You have sold 6 shares of IBM at \$123.65 per share

Your account details:
Name: Bob Broker
Account Balance: 3019.86
- 10 shares of MSFT at \$45.98 per share

Output showing trying to sell a stock not owned:

Your account details:
Name: Bob Broker
Account Balance: 2690.72
You currently own the following stocks:
- 12 shares of IBM at \$98.54 per share, dividend of \$0.04
- 20 shares of MSFT at \$56.34 per share

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>
s
You will now sell stock from your account
Please enter the stock symbol and hit <ENTER>
ORCL
Please enter the number of (whole) shares and hit <ENTER>
5
Please enter the price of the stock and hit <ENTER>
28.76
Is the stock a dividend stock? (y/n)
n
You don't own that stock to sell

Output showing trying to sell off too much stock:

Your account details:
Name: Bob Broker
Account Balance: 2690.72
You currently own the following stocks:
- 12 shares of IBM at \$98.54 per share, dividend of \$0.04
- 20 shares of MSFT at \$56.34 per share

You can (b)uy stock, (s)ell stock, or (q)uit
Enter the first letter of your choice above and hit <ENTER>
s
You will now sell stock from your account
Please enter the stock symbol and hit <ENTER>
IBM
Please enter the number of (whole) shares and hit <ENTER>
15

Please enter the price of the stock and hit <ENTER>

123.65

Is the stock a dividend stock? (y/n)

n

You can't sell that much stock

Your account details:

Name: Bob Broker

Account Balance: 2690.72

You currently own the following stocks:

- 12 shares of IBM at \$98.54 per share, dividend of \$0.04

- 20 shares of MSFT at \$56.34 per share

Part 5 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- Besides changing the StockAccount class what other changes were need to work with multiple stocks? What does this say about the overall design of the program?
- Why is using a Map rather than a Collection better for the StockAccount class? What would be different if a Collection were used?
- How did you use various methods like 'put', 'remove', 'isEmpty', etc on the Map interface? Could you use different combinations of these when buying and selling?

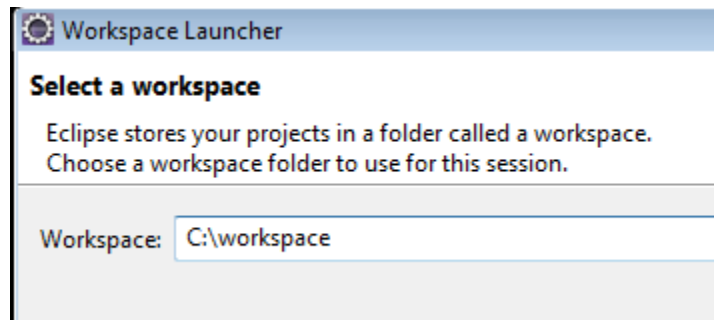
Lab 26 - Introduction to Lambda Expressions

One of the most significant changes in Java 8 was the introduction of “Lambda expressions”. These expressions provide an easy way to implement “functional interfaces”, interfaces that have only one method, and replace many needs for anonymous inner classes.

In this lab you will begin to see Lambda expressions and how code that uses an anonymous inner class can be modified to utilize them.

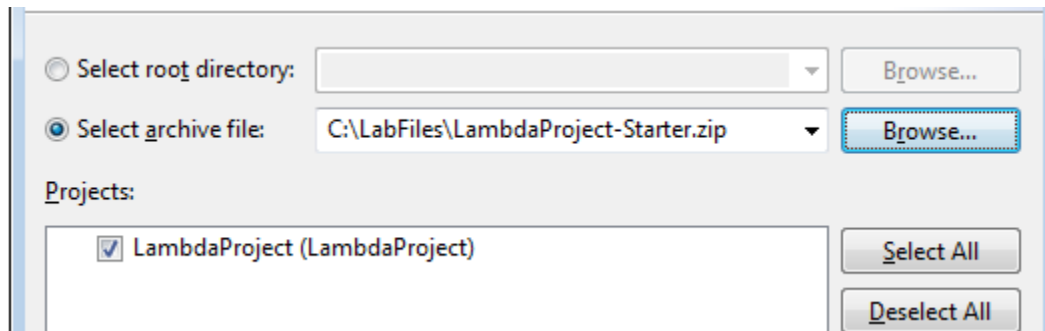
Part 1 - Import Starting Project

- __1. If Eclipse is not already running, from **C:\Software\eclipse** execute **eclipse.exe**
- __2. If you are prompted to select a workspace. Change the the workspace to **C:\workspace**



- __3. Click the **OK** button.
- __4. You may see the **Welcome** screen. Close it.
- __5. From the menu, select **File > Import**.
- __6. Expand and select **General** → **Existing Projects into Workspace** and click the **Next** button.
- __7. Select the radio button next to the **Select archive file** option and use the matching **Browse** button to find and open the file '**C:\LabFiles\LambdaProject-Starter.zip**'.

__ 8. Select the project listed in the file if it is not already as shown below.



__ 9. Press the **Finish** button to import the project.

__ 10. Make sure you are in the **Java** perspective.

__ 11. Expand **LambdaProject** from the *Package Explorer* view.

Part 2 - Examine and Run Current Code

The imported project has functioning code that does not use Lambda expressions. You will examine and run it to understand the functionality before converting to Lambda expressions.

__ 1. From the **LambdaProject**, expand **src** → **com.webage.lambda.person**.

__ 2. Open and examine the purpose of some of the following classes:

- Gender – An Enumeration for MALE and FEMALE
- Person – A JavaBean class that will hold various data about a person
- PersonContact – A utility class for printing out various contact details about a person
- PersonDatabase – A utility class that has a method to generate a List of people for sample data. Examine the details of some of the sample data.
- PersonOperations – A utility class that declares a few methods that will be used in various labs. This is provided to avoid the need to create this code yourself since it does not demonstrate Lambda expressions.

__ 3. Close any classes that are currently open.

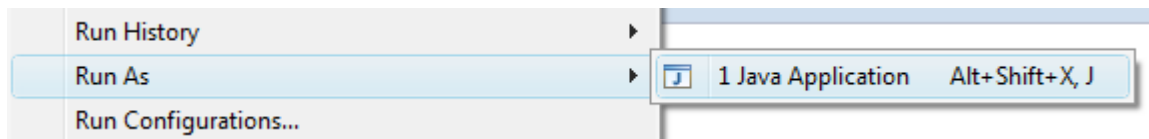
__ 4. In the project, expand **LambdaProject** → **src** → **com.webage.lambda.intro** and open the **PersonSearch.java** file.

__5. Notice how the various methods all take a List of Person objects, select some of them based on some criteria, and then print out various details about the person. Although there are only 3 methods here currently, imagine how difficult it would be to have a method for all three contact options (mail, email, and call) for all three groups (voters, people in New York, and draftees).

__6. Open the **PersonSearchTest.java** file from the **com.webage.lambda.intro** package.

Notice how this class simply gets the list of sample data for people and passes it into the various methods in the PersonSearch class.

__7. With the **PersonSearchTest** class still open, select **Run** → **Run As** → **Java Application**.



__8. The *Console* view should appear and show the output of the application. You may want to maximize the view (double click the view's tab or use the maximize button in the upper right corner of the view) to see all of the output.

```
----- Calling Voters -----  
Calling Bob Thompson age 46 at 555-123-4567  
Calling Susan Brown age 27 at 222-444-1267  
Calling John Adams age 24 at 333-111-6767  
Calling Devon Jasso age 68 at 520-376-7538  
Calling Nicole Allen age 53 at 450-876-9435  
----- Mailing New York -----  
Mailing Bob Thompson age 46 at 100 Broadway, New York, NY 10005  
Mailing Wally Schumaker age 15 at 100 Broadway, New York, NY 10005  
----- Emailing Draftees -----  
Emailing John Adams age 24 at john@mycompany.com
```

Note: It is important to note that the code changes you will be making in this lab will not really impact the output. Using Lambda expressions will mainly improve the code design itself so it is easier to change the search conditions if needed.

__9. Restore the view to normal if you maximized it before going to the next section.

Part 3 - Refactor Test Logic Into Methods

One difficulty in the current code is that the logic to decide if a person matches certain criteria is more difficult to read since it is done within the 'if' statement. Refactoring this to separate methods might improve the code.

___1. Open the **PersonSearch.java** file from the **com.webage.lambda.intro** package if it is not already open.

___2. Add the following three methods to the class. Make sure they are outside other methods but within the brackets for the class itself.

```
private boolean isVoter(Person p) {
    return p.getAge() >= 18;
}

private boolean isNewYork(Person p) {
    return p.getCity().equalsIgnoreCase("New York");
}

private boolean isDraftee(Person p) {
    return p.getAge() >= 18
        && p.getAge() <= 25
        && p.getGender() == Gender.MALE;
}
```

___3. Save the code and make sure you don't have any errors before moving on.

___4. Find the existing '**callVoters**' method and replace the condition of the 'if' statement with a call to one of the new methods as shown in bold below.

```
public void callVoters(List<Person> people) {
    for (Person p : people) {
        if (isVoter(p)) {
            contact.callPerson(p);
        }
    }
}
```


___5. Find the existing '**mailNewYorkCity**' method and replace the condition of the 'if' statement with a call to one of the new methods as shown in bold below.

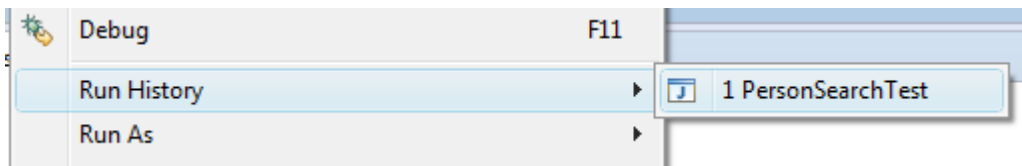
```
public void mailNewYorkCity(List<Person> people) {  
    for (Person p : people) {  
        if (isNewYork(p)) {  
            contact.mailPerson(p);  
        }  
    }  
}
```

___6. Find the existing '**emailDraftees**' method and replace the condition of the 'if' statement with a call to one of the new methods as shown in bold below.

```
public void emailDraftees(List<Person> people) {  
    for (Person p : people) {  
        if (isDraftee(p)) {  
            contact.emailPerson(p);  
        }  
    }  
}
```

___7. Save the code and make sure you don't have any errors.

___8. From the Eclipse menus, select '**Run → Run History → PersonSearchTest**'



__9. Check that the output in the *Console* view is the same as it was before.

```
----- Calling Voters -----
Calling Bob Thompson age 46 at 555-123-4567
Calling Susan Brown age 27 at 222-444-1267
Calling John Adams age 24 at 333-111-6767
Calling Devon Jasso age 68 at 520-376-7538
Calling Nicole Allen age 53 at 450-876-9435
----- Mailing New York -----
Mailing Bob Thompson age 46 at 100 Broadway, New York, NY 10005
Mailing Wally Schumaker age 15 at 100 Broadway, New York, NY 10005
----- Emailing Draftees -----
Emailing John Adams age 24 at john@mycompany.com
```

__10. Restore the *Console* view to normal if you maximized it before going to the next section.

Part 4 - Pass Predicate From Calling Code

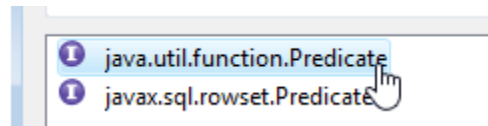
Although the code is certainly simpler to read, it could still be improved. Currently you would need to define additional methods to have a different combination of conditions and the action to take (emailing voters for example). Java 8 defines a 'functional interface' called `Predicate` that can be used to define a test that will return a boolean value for given input data. By adding a new parameter to the existing `PersonSearch` methods, you can have the class that calls the methods pass in the condition that a `Person` should meet in order to have a certain action taken. This will make the `PersonSearch` class much more generic and flexible for different situations. At first you will pass implementations of this `Predicate` interface as Anonymous Inner Classes (AIC).

__1. Open the **PersonSearch.java** file from the **com.webage.lambda.intro** package if it is not already open.

__2. Find the existing '**callVoters**' method and make the following changes to the method signature. Notice this includes changing the name of the method, since it will no longer be specific to voters, AND adding a new parameter for a `Predicate` object. You will have compilation errors for several steps.

```
public void callMatchingPeople(List<Person> people,
    Predicate<Person> aTest) {
```

___3. Organize imports (**Source** → **Organize Imports** or **CTRL+SHIFT+O**) and pick '**java.util.function.Predicate**' when prompted.



___4. Within the body of the method, change the condition of the 'if' statement to use the test of the Predicate passed in as a parameter. Note how this makes the method more generic as any condition can be passed as a parameter.

```
public void callMatchingPeople(List<Person> people,
    Predicate<Person> aTest) {
    for (Person p : people) {
        if (aTest.test(p)) {
            contact.callPerson(p);
        }
    }
}
```

___5. Save the code and make sure you don't have any errors in this file before moving on. You will have errors elsewhere in the project though.

___6. Find the existing '**mailNewYorkCity**' method and make the following changes to the method signature. Again you are changing the name of the method AND adding a new parameter for a Predicate object.

```
public void mailMatchingPeople(List<Person> people,
    Predicate<Person> aTest) {
```

___7. Within the body of the method, change the condition of the 'if' statement to use the test of the Predicate passed in as a parameter.

```
public void mailMatchingPeople(List<Person> people,
    Predicate<Person> aTest) {
    for (Person p : people) {
        if (aTest.test(p)) {
            contact.mailPerson(p);
        }
    }
}
```

__8. Find the existing '**emailDraftees**' method and make the following changes to the method signature. Again you are changing the name of the method AND adding a new parameter for a Predicate object.

```
public void emailMatchingPeople(List<Person> people,  
    Predicate<Person> aTest) {
```

__9. Within the body of the method, change the condition of the 'if' statement to use the test of the Predicate passed in as a parameter.

```
    public void emailMatchingPeople(List<Person> people,  
        Predicate<Person> aTest) {  
        for (Person p : people) {  
            if (aTest.test(p)) {  
                contact.emailPerson(p);  
            }  
        }  
    }  
}
```

__10. Remove the '**isVoter**', '**isNewYork**' and '**isDraftee**' methods as you will no longer need these since the conditions will be passed as a Predicate parameter instead of being defined by the PersonSearch class. You can also remove the '**Gender**' import if you wish as it will no longer be used.

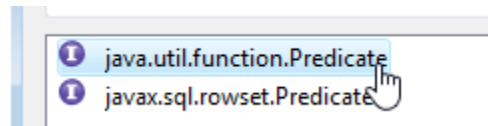
__11. Save the code and make sure you don't have any errors in this file before moving on. You will have errors elsewhere in the project though.

__12. Open the **PersonSearchTest.java** file from the **com.webage.lambda.intro** package if it is not already open. This class now has compilation errors that need to be fixed.

__13. Find the code in the '**main**' method that currently calls the '**callVoters**' method and make the following changes. Note you are changing the name of the method being called AND creating a new instance of the Predicate interface as an Anonymous Inner Class (AIC). Be careful with the syntax as defining the AIC within the parameters of the method is tricky.

```
System.out.println("----- Calling Voters -----");  
search.callMatchingPeople(people, new Predicate<Person>() {  
    @Override  
    public boolean test(Person p) {  
        return p.getAge() >= 18;  
    }  
});
```

__14. Organize imports (**Source** → **Organize Imports** or **CTRL+SHIFT+O**) and pick '**java.util.function.Predicate**' when prompted.



__15. Save the code and make sure you no longer have any compilation errors in the code you just modified. You will still have other compilation errors though.

__16. Find the code in the '**main**' method that currently calls the '**mailNewYorkCity**' method and make the following changes. Again note you are changing the name of the method being called AND creating a new instance of the Predicate interface as an Anonymous Inner Class (AIC).

```
System.out.println("----- Mailing New York -----");
search.mailMatchingPeople(people, new Predicate<Person>() {
    @Override
    public boolean test(Person p) {
        return p.getCity().equalsIgnoreCase("New York");
    }
});
```

__17. Save the code and make sure you no longer have any compilation errors in the code you just modified. You will still have other compilation errors though.

__18. Find the code in the '**main**' method that currently calls the '**emailDraftees**' method and make the following changes. Again note you are changing the name of the method being called AND creating a new instance of the Predicate interface as an Anonymous Inner Class (AIC).

```
System.out.println("----- Emailing Draftees-----");
search.emailMatchingPeople(people, new Predicate<Person>() {
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18
            && p.getAge() <= 25
            && p.getGender() == Gender.MALE;
    }
});
```

__19. Organize imports (**Source** → **Organize Imports** or **CTRL+SHIFT+O**) to import the **Gender** enumeration definition.

__20. Save the code and make sure there are no compilation errors in any files.

- __21. From the Eclipse menus, select '**Run** → **Run History** → **PersonSearchTest**'
- __22. Check that the output in the *Console* view is the same as it was before.

```
----- Calling Voters -----  
Calling Bob Thompson age 46 at 555-123-4567  
Calling Susan Brown age 27 at 222-444-1267  
Calling John Adams age 24 at 333-111-6767  
Calling Devon Jasso age 68 at 520-376-7538  
Calling Nicole Allen age 53 at 450-876-9435  
----- Mailing New York -----  
Mailing Bob Thompson age 46 at 100 Broadway, New York, NY 10005  
Mailing Wally Schumaker age 15 at 100 Broadway, New York, NY 10005  
----- Emailing Draftees -----  
Emailing John Adams age 24 at john@mycompany.com
```

- __23. Restore the *Console* view to normal if you maximized it before going to the next section.

Part 5 - Use Lambda Expression

Although introducing the Predicate interface as a parameter to the methods in the PersonSearch class has made it more flexible to use, actually using it by creating an instance of an AIC (Anonymous Inner Class) is not very easy. This is where the Lambda expressions introduced in Java 8 can greatly simplify things. These expressions are stripped down to just the unique code involved in implementing the “functional interface” (an interface with one method) and do not need all of the more complex syntax involved with defining an AIC.

- __1. Open the **PersonSearchTest.java** file from the **com.webage.lambda.intro** package if it is not already open.
- __2. Find the code in the '**main**' method that currently calls the '**callMatchingPeople**' method with voters and make the following changes. Note that you are removing the AIC instance and replacing it with a Lambda expression. Make sure the syntax for the method parameters is still correct.

```
System.out.println("----- Calling Voters -----");  
search.callMatchingPeople(people, p -> p.getAge() >= 18);
```

- __3. Save the code and make sure you do not have any compilation errors.

__4. Find the code in the '**main**' method that currently calls the '**mailMatchingPeople**' method with residents of "New York" and make the following changes. Again note that you are removing the AIC instance and replacing it with a Lambda expression.

```
System.out.println("----- Mailing New York -----");
search.mailMatchingPeople(people,
    p -> p.getCity().equalsIgnoreCase("New York"));
```

__5. Save the code and make sure you do not have any compilation errors.

__6. Find the code in the '**main**' method that currently calls the '**emailMatchingPeople**' method with draftees and make the following changes. Again note that you are removing the AIC instance and replacing it with a Lambda expression. The syntax of this expression is a bit more complex.

```
System.out.println("----- Emailing Draftees -----");
search.emailMatchingPeople(people, p -> p.getAge() >= 18
    && p.getAge() <= 25
    && p.getGender() == Gender.MALE);
```

__7. Save the code and make sure you do not have any compilation errors.

__8. From the Eclipse menus, select '**Run → Run History → PersonSearchTest**'

__9. Check that the output in the *Console* view is the same as it was before.

```
----- Calling Voters -----
Calling Bob Thompson age 46 at 555-123-4567
Calling Susan Brown age 27 at 222-444-1267
Calling John Adams age 24 at 333-111-6767
Calling Devon Jasso age 68 at 520-376-7538
Calling Nicole Allen age 53 at 450-876-9435
----- Mailing New York -----
Mailing Bob Thompson age 46 at 100 Broadway, New York, NY 10005
Mailing Wally Schumaker age 15 at 100 Broadway, New York, NY 10005
----- Emailing Draftees -----
Emailing John Adams age 24 at john@mycompany.com
```

__10. Restore the *Console* view to normal if you maximized it before going to the next section.

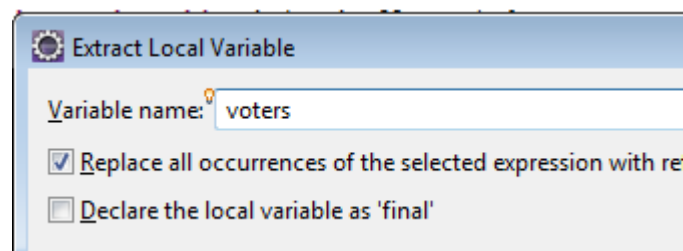
Part 6 - Reuse Lambda Expressions

Note how much simpler it is to use Lambda expressions as opposed to instances of an AIC. The only thing that is not done in the previous section is declaring a variable that can be used to reference and reuse the Lambda expressions. That will be done in this section.

- ___ 1. Open the **PersonSearchTest.java** file from the **com.webage.lambda.intro** package if it is not already open.
- ___ 2. Select the code of the Lambda expression for the voters by highlighting it as shown below. Make sure to only select the Lambda expression and not the comma or parenthesis before and after it.

```
System.out.println("----- Calling Voters -----");
search.callMatchingPeople(people, p -> p.getAge() >= 18);
```

- ___ 3. From the Eclipse menus select '**Refactor** → **Extract Local Variable**'. If this option is not available or gives an error double check what you have selected and try again.
- ___ 4. Change the name of the local variable that will be created to '**voters**' and click the **OK** button to define the variable.



- ___ 5. Check that the code is modified to match that shown below.

```
System.out.println("----- Calling Voters -----");
Predicate<Person> voters = p -> p.getAge() >= 18;
search.callMatchingPeople(people, voters);
```

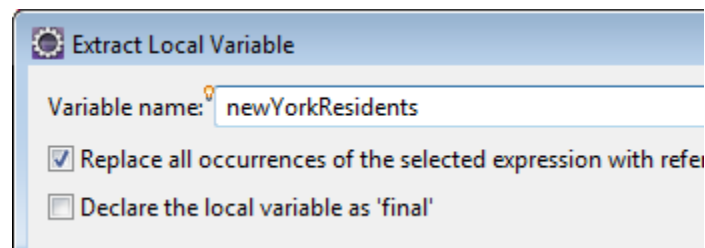

__6. Select the code of the Lambda expression for the New York residents by highlighting it as shown below. Make sure to select only one of the parenthesis at the end.

```
people(people, voters);

----- Mailing New York -----");
people(people, p -> p.getCity().equalsIgnoreCase("New York"));
----- Emailing Draftees -----");
```

__7. From the Eclipse menus select '**Refactor** → **Extract Local Variable**'. If this option is not available or gives an error double check what you have selected and try again.

__8. Change the name of the local variable that will be created to '**newYorkResidents**' and click the **OK** button to define the variable.



__9. Check that the code is modified to match that shown below.

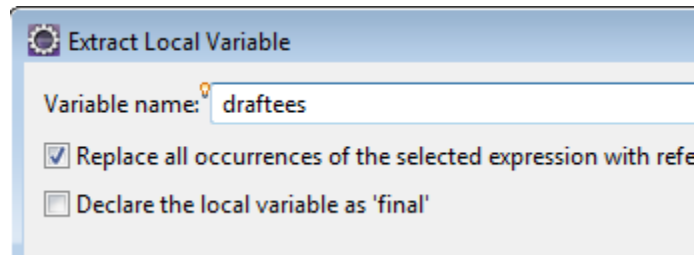
```
System.out.println("----- Mailing New York -----");
Predicate<Person> newYorkResidents =
    p -> p.getCity().equalsIgnoreCase("New York");
search.mailMatchingPeople(people, newYorkResidents);
```

__10. Select the code of the Lambda expression for the draftees by highlighting it as shown below. Make sure not to select the parenthesis at the end.

```
System.out.println("----- Emailing Draftees -----");
search.emailMatchingPeople(people, p -> p.getAge() >= 18
    && p.getAge() <= 25
    && p.getGender() == Gender.MALE);
```

__11. From the Eclipse menus select '**Refactor** → **Extract Local Variable**'. If this option is not available or gives an error double check what you have selected and try again.

__12. Change the name of the local variable that will be created to '**draftees**' and click the **OK** button to define the variable.



__13. Check that the code is modified to match that shown below.

```
System.out.println("----- Emailing Draftees -----");
Predicate<Person> draftees = p -> p.getAge() >= 18
    && p.getAge() <= 25
    && p.getGender() == Gender.MALE;
search.emailMatchingPeople(people, draftees);
```

__14. Save the file.

__15. From the Eclipse menus, select '**Run → Run History → PersonSearchTest**'.

__16. Check that the output in the *Console* view is the same as it was before.

```
----- Calling Voters -----
Calling Bob Thompson age 46 at 555-123-4567
Calling Susan Brown age 27 at 222-444-1267
Calling John Adams age 24 at 333-111-6767
Calling Devon Jasso age 68 at 520-376-7538
Calling Nicole Allen age 53 at 450-876-9435
----- Mailing New York -----
Mailing Bob Thompson age 46 at 100 Broadway, New York, NY 10005
Mailing Wally Schumaker age 15 at 100 Broadway, New York, NY 10005
----- Emailing Draftees -----
Emailing John Adams age 24 at john@mycompany.com
```

__17. Modify the code that had called voters to instead call retirees as shown below.

```
System.out.println("----- Calling Retirees -----");
Predicate<Person> retirees = p -> p.getAge() >= 65;
search.callMatchingPeople(people, retirees);
```

__18. Save the code and make sure you do not have any errors.

- __19. From the Eclipse menus, select '**Run** → **Run History** → **PersonSearchTest**'.
__20. Check that the output changes to reflect calling the new group.

```
----- Calling Retirees -----  
Calling Devon Jasso age 68 at 520-376-7538  
----- Mailing New York -----  
Mailing Bob Thompson age 46 at 100 Broadway, New York, NY 10005  
Mailing Wally Schumaker age 15 at 100 Broadway, New York, NY 10005  
----- Emailing Draftees -----  
Emailing John Adams age 24 at john@mycompany.com
```

- __21. Modify the code that had mailed New York residents to instead mail retirees as shown below.

```
search.mailMatchingPeople(people, retirees);
```

- __22. Save the code and make sure you do not have any errors.
__23. From the Eclipse menus, select '**Run** → **Run History** → **PersonSearchTest**'.
__24. Make sure the output changes as shown below.

```
----- Calling Retirees -----  
Calling Devon Jasso age 68 at 520-376-7538  
----- Mailing New York -----  
Mailing Devon Jasso age 68 at 133 W 23rd St., Tucson, AZ 85713  
----- Emailing Draftees -----  
Emailing John Adams age 24 at john@mycompany.com
```

- __25. Close all open files.

Part 7 - Review

This lab showed you how using Lambda expressions can simplify code. First you moved the logic for how to determine which people to call, mail, or email to the class calling these methods and passed the logic as an instance of the Predicate interface. Without Lambda expressions you saw how this would require Anonymous Inner Classes which is more difficult syntax. Using Lambda expressions you were able to simplify the syntax while retaining the same functionality.

Lab 27 - Writing To A File

Time for Lab: 35 minutes

In this lab, you will examine how to write simple objects to a file.

Java has a very comprehensive *stream* API. A stream is a serial destination that can be output to. An example of a stream is the console, which we have been accessing whenever we called **System.out.println()**. Indeed, **System.out** is a stream representing the console, and we call the **println()** method on it.

The stream hierarchy is very flexible; we have seen it is easy to print to the console. We could also easily print data to a stream representing a network connection, for network communication.

Even more interestingly, a file on the file system can also be treated as a stream; this means we could write data to a file, using the stream API.

We will use this stream feature to write our **AccountManager** data to a file. We will do this by writing a method to write the contents of the **AccountManager**'s collection out to disk. We will then write a method to read in the contents from that same file.

Part 1 - Serialization

Ordinarily, writing an object out to disk would involve a lot of hand-written code. Fortunately, Java is quite good at doing it for us. After all, an object is essentially just a collection of its fields and their state; these are simply bytes in memory. For simple objects, writing its state out to disk is just a matter of *serializing* it (which means to take its shape and write it out as a sequential series of bytes). Simple objects (like our **Account** classes) can be easily serialized, by simply declaring that the class is serializable. Once a class is serializable, writing the class is very simple, requiring only a few lines of code.

The problem at hand is this: we want to take an **AccountManager** and want to serialize its collection of accounts. Remember that the collection of accounts is actually a **HashMap**, so in reality we need to serialize a **HashMap**. Luckily for us, the Java-provided **HashMap** class is already serializable. The class has its own code for writing itself out to disk; if we serialize a **HashMap**, the code will simply serialize its contents.

Recall, however that our **HashMap** will contain instances of **BusinessAccount**, **BankAccount**, and **SavingsAccount**. Those are currently *not* serializable. The good news is that it is very easy to make these classes serializable. We will do this now.

__1. Open **BusinessAccount.java** from **BankAccountProject**.

__2. Change the class declaration as follows:

```
public class BusinessAccount implements Account, Serializable {
```

All we do here is specify that the class implements the **Serializable** interface, which is provided by the Java language. Notice the comma (,) allows for a class to implement multiple interfaces.

__3. An error will appear. Organize imports and select **java.io.Serializable** if prompted.

__4. Save the file. There should be no errors although you may have warnings.

We have now made the **BusinessAccount** serializable.

Now we need to make the **BankAccount** class serializable.

__5. Open **BankAccount.java**.

__6. As before, change the class declaration as follows:

```
public class BankAccount implements Account, Serializable {
```

__7. Again, organize imports. In the *Organize Imports* window and select **java.io.Serializable** if prompted.

__8. Save the file. There should be no errors although you may have warnings.

What about the **SavingsAccount**? Do we not need to make that **Serializable** as well? The answer is no; remember that **SavingsAccount** extends from **BankAccount**, and so inherits the **Serializable** implementation. Job done.

Our **Account** classes are now serializable.

Part 2 - Add the Write/Read Methods

We can now add the actual code that performs the file write and file read to the **AccountManager** class.

- ___1. Open AccountManager.java.
- ___2. Add the following method to the class.

```
public void persist() throws IOException {
    File f = new File("C:/Workspace/users.dat");
    OutputStream os = new FileOutputStream(f);
    try (ObjectOutputStream output = new ObjectOutputStream(os))
    {
        output.writeObject(this.accounts);
    }
}
```

Note: This code uses the Java 7 "try with resources" statement so that the 'output' object will be closed automatically when it is done being used.

This is simple code that looks complex. The method name “persist” means to 'write out to permanent storage', like a disk.

The first line declares an instance of a **File** object, and maps it to the location C:/Workspace/users.dat. This is where our data will be written to.

The second line creates a new **OutputStream** on the file we just created.

The third line then sets up an **ObjectOutputStream** and maps it to the **OutputStream** (which has already been mapped to the file). This **ObjectOutputStream** will be responsible for performing the actual serialization.

The final line performs the actual serialization via the **writeObject()** method. Note that we write the **HashMap**.

Note that the method throws an **IOException**. This exception will be raised if the code has any problems opening the file for writing. We will have to remember that when invoking this method.

__3. Add the following method to read data back in from the file.

```
public void load() throws IOException, ClassNotFoundException {
    File f = new File("C:/Workspace/users.dat");
    InputStream is = new FileInputStream(f);
    try (ObjectInputStream input = new ObjectInputStream(is)) {
        this.accounts=(HashMap<Integer, Account>)
            input.readObject();
    }
}
```

The code here is very similar to the **persist** method, except we are using **InputStreams** instead of **OutputStreams**. Also notice the call to **readObject()** instead of **writeObject()**.

Finally notice the *cast* of the read-in object to a **HashMap**. The **AccountManager's accounts** variable is then assigned to the result of this read-in **HashMap**. Unfortunately you will get a warning about “type safety” because the list of accounts in the **AccountManager** class is qualified by Generics but the 'readObject' method does not return an Object with this qualification. The code will still run, the compiler is just telling you that there is no way to enforce that whatever comes back from the 'readObject' method is always a HashMap of Account objects using Integers for keys.

__4. Organize imports and select the following imports: **java.io.InputStream** and **java.io.OutputStream**.

__5. Save the file. There should be no errors.

Part 3 - Test the Serialization

We will now test these methods. We will create an **AccountManager** and add some accounts to it. We will then try writing this **AccountManager** out to disk. Next, we will try reading it back in to see if the serialization worked.

__1. Open **AccountManagerTester.java**.

__2. Locate the line:

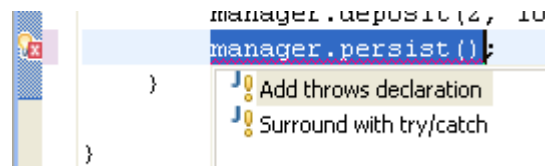
```
System.out.println(manager.getAccount(2));
```


__3. Delete it. In its place, add the following code:

```
manager.persist();
```

An error appears; this is happening because `persist` throws an **IOException**. We have to catch it.

__4. We will be lazy and let Eclipse generate a **try/catch** clause for us. Click on the red X in the margin next to the line. In the pop-up box that appears, double click on **Surround with try/catch**



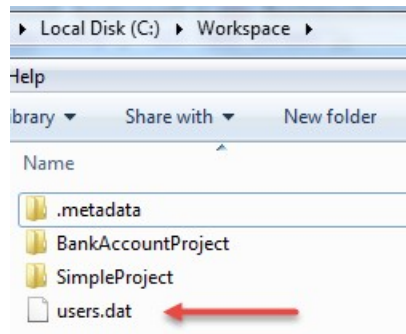
Java will generate some code for you. Your code should now look like this:

```
...
    manager.deposit(2, 100f);
    try {
        manager.persist();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
...
```

__5. Save the code. There should be no errors.

__6. Run the code. You should see nothing.

__7. We have not printed anything to the console, so there will be no feedback here. However, open a Windows file explorer and navigate to **C:\Workspace**. You should see a new file called **users.dat**. This is our persisted AccountManager's collection!



__8. Open this file with a text editor (like Notepad). It looks like a jumbled mess of binary and character data. Believe it or not, that is our persisted **HashMap**! Close the text editor.

Now, let us see if it can be read back in.

__9. Back in Eclipse, add the following code to the **AccountManagerTester**'s main method.

```
AccountManager readTest = new AccountManager();
readTest.load();
System.out.println(readTest);
```

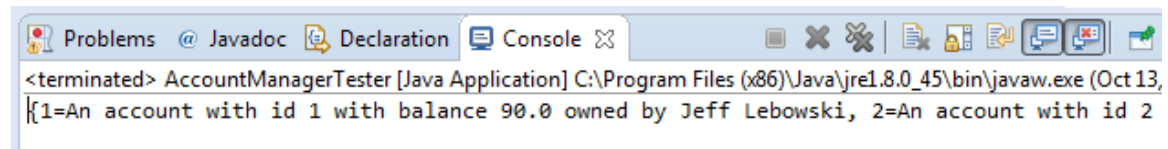
Here, we declare a new instance of an **AccountManager** and then call the **load()** method. If it works, the accounts should be read in from disk and the next call to **println** should show this.

__10. There is an error on the call to **load()**. As with the **persist()** method, we have to handle potential exceptions. Left click on the red X in the margin, and again select **Surround with try/catch**

__11. Save the code. There should be no errors.

__12. Run the code. You should see:

```
{1=An account with id 1 with balance 90.0 owned by Jeff Lebowski, 2=An  
account with id 2 with balance 10100.0 owned by Maude Lebowski with  
interest rate 0.0 and minimum balance 1000.0, 3=A BusinessAccount  
belonging to SimpleCorp with balance 1000.0}
```



Note. If you don't see the message in the console, restart Eclipse and run the class again; then you should see the message in the console.

Looks like it works! You have successfully serialized an object to disk, and then de-serialized it back in from disk. This shows you how easy it is to persist objects in Java.

__13. Close all open files.

Part 4 - Review

In this exercise, we saw how Java can write objects out to disk using the serialization technique. We saw that a class has to be declared as serializable (by implementing the appropriate interface). Once it has been declared thusly, stream API can be used to write the object, as well as read it back in.

Lab 28 - Project - Saving to a File (Optional)

This lab will continue the project. You have built up a number of good features in the Stock Tracker program. The only problem right now is that once the user quits all of the data about their stocks is lost. This part of the project will have you save that information to a file so it can be retrieved later.

Even though there are many different features we could add related to a file, the requirements given will be fairly simple just so you get a sense for working with a file.

Part 1 - General Requirements

- Create a new class called `StockFileManager` in the `'com.stock'` package. Since the program has split different responsibilities between classes up until now it will make sense to add the implementation of working with a file to a separate class.
- In the new `StockFileManager` class add a `String` constant for the name of the file that information will be saved to. It is generally better to use forward slashes (/) as the separation between directories so the file name can be something like `"C:/temp/stocks.dat"`.
- Modify the `StockAccount`, and `Stock` classes to implement the `java.io.Serializable` interface. This does not involve adding any methods but simply the correct `'implements'` declaration in the class declaration. You may also need to add an import statement. You may have some compiler warnings after this but these are OK.
- Add a `'storeAccountInfo'` method to the `StockFileManager` class. This method should take a `StockAccount` object as a parameter and return no return type. The method should throw a `StockException`. The method should also be a static method as it will be called without an instance of the `StockFileManager` class. The implementation will be described next.
- In the implementation of the `'storeAccountInfo'` method, check to see if the file indicated by the constant exists using the `java.io.File` class. If it does, delete the old file.

- Create an `ObjectOutputStream` object remembering that you must first create a `FileOutputStream` to pass into the `ObjectOutputStream` constructor. Make sure to store the object created in a variable for use within the rest of the method. Use the `'writeObject'` method of the `ObjectOutputStream` to write out the current `StockAccount` object to the stream. Close the output stream. Surround all of the code that can throw exceptions with a try/catch block. You should catch a `'FileNotFoundException'` and an `'IOException'`. When catching these exceptions, throw a new `StockException` with a meaningful message.
- Modify the code of the `StockTracker` class so that when the user indicates to quit the static `'storeAccountInfo'` method of the `StockFileManager` class is called with the current account as the parameter passed. You will also need to catch the `StockException` that the method may throw and print out the error message if this happens.
- Make sure all the code is saved and compiles without errors. Run the program and after quitting the program make sure a file is written out.
- Add a `'getStoredAccount'` method to the `StockFileManager` class. This method should be a static method. This method should return a `StockAccount` object. This method does not take any parameters. Indicate that the method throws a `StockException`. Further details on how to implement the method are given in the next requirements, for now just declare the method. You will get a compiler error since the method does not yet return a `StockAccount`.
- In the implementation of the `'getStoredAccount'` method check to see if the file at the location described by the constant exists using the `java.io.File` class. If the file does not exist return a `'null'` from the `'getStoredAccount'` method.
- If the file does exist, open an `ObjectInputStream` based on it. Remember that you must first create a `FileInputStream` to pass into the `ObjectInputStream` constructor. Use the `'readObject'` method of the `ObjectInputStream` class to read in an object. Close the input stream. Use the `'instanceof'` operator to see if the object read in is a `StockAccount` object. If it is, cast it to a `StockAccount` object and return it otherwise simply return `'null'`. Surround all of the code that can throw exceptions with a try/catch block. You should catch a `'ClassNotFoundException'`, a `'FileNotFoundException'` and an `'IOException'`. When catching these exceptions, throw a new `StockException` with a meaningful message.

- Modify the code of the StockTracker class to first attempt to get account information using the 'getStoredAccount' method of the StockAccount class. The StockAccount variable used in the class should be initialized with the return value of this method. If the value returned is 'null', no account data has been saved and the user should create a new account as before. If an exception is thrown from the 'getStoredAccount' method, print out the error and then prompt the user to create a new account.
- Run the program several times and make sure when you run it again the stock account starts with the same info as the end of the previous run.

Part 2 - Tips

This section will contain some tips about things that may cause problems during this part of the project.

- Remember to use the 'static' and 'final' keywords when declaring the constant for the file name. Also remember the convention with constants is to have all capitals letters with an underscore before a new word in the constant name.
- You can call the 'storeAccountInfo' either within the loop or immediately after but it should only be called once.
- Use the options of the development tool to automatically create the try/catch blocks if possible.
- Use a "try with resources" statement if possible to automatically close the resources for the file. Don't close the Scanner used for input though.
- Make sure you test the program several times to be sure it can save and read the file correctly. If you modify and recompile your classes you may have errors reading a previously saved file. You may need to manually delete a previously saved file to test how the program behaves when the file does not exist.

Part 3 - Sample Output

The output is similar to previous parts of the project. The main difference is if account data already exists in a file you should not be prompted to create a new account, the program should just automatically list the account data.

Part 4 - Discussion Questions

This section will contain questions that will be good to think about and discuss with the rest of the class. These questions will highlight things that may be solved in different ways or things that may be currently difficult to do.

- What would be some other ways to work with data in a file?
- What changes might be needed if the user were able to pick the file name to store and retrieve account info from?
- What happens if the file exists but stores other data instead of stock account data? What would be other ways to handle this error?
- Does the StockTracker class know the details of how the account data is being stored in a file? How might this design lead to flexibility in the future?

Lab 29 - Build Script Basics

Gradle was already configured for you under **C:\Software\gradle-6.3** folder and added in the environment variables; Gradle can be downloaded by following this URL: <https://gradle.org/gradle-download/>. **Complete Distribution** contains the binary files, documentation, and source code. If you prefer, you can just download the binary files.

In this lab you will build a Java Project, manage dependencies, and run unit tests with Gradle.

Part 1 - Gradle installation verification

- __ 1. Open a Command Prompt.
- __ 2. Verify Gradle is installed.

```
gradle -version
```

In this lab you will create build scripts and utilize various features.

Part 2 - Create directory structure for storing build script

In this part you will create directory structure for storing build script.

- __ 1. Switch to the Workspace directory.

```
cd C:\Workspace
```

- __ 2. Create the directory structure.

```
mkdir labs\gradle\basics
```

- __ 3. Switch to the newly created directory.

```
cd labs\gradle\basics
```


Part 3 - Create Gradle build script and use shortcut task definition technique for defining a task

In this part you will create a Gradle build script and add a task using shortcut task definition technique.

__ 1. Create build.gradle file.

notepad build.gradle

__ 2. Click Yes to create the file.

__ 3. Enter following code.

```
// let's add a simple task
task hello1 {
    println 'Hello World!'
}
```

Here you have added a single-line comment and a task which displays a message on the screen.

__ 4. Save the file.

__ 5. Run Gradle.

gradle

Notice it's asking us to pass it more parameters.

```
C:\Workspace\labs\gradle\basics>gradle

> Configure project :
Hello World!

> Task :help

Welcome to Gradle 6.3.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

For troubleshooting, visit https://help.gradle.org

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

__ 6. Get task list.

```
gradle tasks --all
```

Notice it lists 'hello1' under "Other tasks" section.

__ 7. Execute hello1 task.

```
gradle hello1
```

Notice it displays "Hello World!" message on the screen, but there are additional messages logged as well.

__ 8. Execute the task by suppressing the log.

```
gradle -q hello1
```

Notice this time it just displays the message.

Part 4 - Create a task with actions

In this part you will create another task which uses actions.

__1. In the Notepad window, which contains build.gradle code, append following code.

```
// add another task with actions
task hello2 {
    doFirst {
        print 'This is '
    }
    doLast {
        println 'a test!'
    }
}
```

Note: doFirst and doLast are the predefined actions. As the name suggests, doFirst code is executed before doLast. You don't need to use both together. You can use one or the other.

__2. Save the code.

__3. Execute hello2 task.

```
gradle -q hello2
```

It displays "This is a test!" message on the screen.

Note: "Hello World!" will be displayed in every test.

Part 5 - Define Task Dependency

In this part you will define task dependency. A task will make use of another task.

__ 1. In the Notepad window, which contains build.gradle code, append following code.

```
task welcome(dependsOn: hello1) {  
    doLast {  
        println "Welcome Bob!"  
    }  
}
```

Note: welcome task depends on hello1. It will execute hello1 then welcome task. In this case hello1 task already exists. In case if you define 'welcome' task before defining the dependent hello1 task, then use following syntax.

```
task welcome(dependsOn: 'hello1')
```

Notice here you have used quotes for defining the dependent task. This technique is also called Lazy dependsOn.

__ 2. Save the code.

__ 3. Execute welcome task.

```
gradle -q welcome
```

Notice it displays 'Hello Bob!'

Part 6 - Alternative technique for defining task dependency

In this part you will use a different technique for defining task dependency.

___ 1. In the Notepad window, which contains build.gradle code, append following code.

```
task hello {  
}  
  
hello.dependsOn hello1, hello2
```

Note: hello task depends on hello1 and hello2. It will execute hello1, hello2, then if hello task.

___ 2. Save the code.

___ 3. Execute hello task.

```
gradle -q hello
```

Notice it displays 'This is a test!'

Part 7 - Using Task Properties

In this part you will define task properties and reuse them in other tasks.

___ 1. In the Notepad window, which contains build.gradle code, append following code.

```
task myProperties {  
    ext.greeting = "Hello, "  
    ext.userName = "Bob"  
}
```

Note: Here you have created a custom task named myProperties and added 2 properties to it. Task name can be anything, but you must use ext object to add properties to the task.

__2. Append follow code to reuse the properties.

```
task useProperties {  
    doLast {  
        println myProperties.greeting + myProperties.userName  
    }  
}
```

Note: Here you have used the properties by specifying the task name.

__3. Save the code.

__4. Execute useProperties task.

```
gradle -q useProperties
```

Notice it displays 'Hello, Bob' message on the screen.

Part 8 - Create a method and reuse it in Gradle build script

In this part you will create a method and then reuse it in the Gradle script.

__1. In the Notepad window, which contains build.gradle code, append following code.

```
String toUpper(String userName) {  
    userName.toUpperCase()  
}
```

Note: This method takes string based input, converts it to upper case, and returns it.

__2. Append following code to reuse the method.

```
task callMethod {  
    doLast {  
        println myProperties.greeting + toUpper(myProperties.userName)  
    }  
}
```

Note: The custom toUpper method is called by the doLast action of custom task.

__3. Save the code.

__4. Execute the callMethod task.

```
gradle -q callMethod
```

Notice it displays 'Hello, BOB' message on the screen.

Part 9 - Defining default tasks

In this part you will set default tasks so they should execute even without specifying them as part of command-line arguments.

__1. In the Notepad window, which contains build.gradle code, append following code.

```
task defaultTask1 {
    doLast {
        println 'Default Task 1!'
    }
}

task defaultTask2 {
    doLast {
        println 'Default Task 2!'
    }
}
```

Note: Here you have defined 2 custom tasks. Next, you will set them as default tasks.

__2. In the beginning of the file, add following code.

```
defaultTasks 'defaultTask1', 'defaultTask2'
```

__3. Save the code.

__4. Execute the build script without specifying any task name.

```
gradle -q
```

Notice it displays 'Default Task 1! Default Task 2!' message on the screen.

Part 10 - Create Tasks Dynamically

In this part you will create tasks dynamically by using a loop.

__ 1. In the Notepad window, which contains build.gradle code, append following code.

```
3.times { i ->
    task "task$i" {
        doLast {
            println "Task #: $i"
        }
    }
}
```

Note: Here you are using groovy syntax to define a loop which runs 3 times. The current index location is stored in variable i. String interpolation is using to retrieve value of i and append it to "task".

__ 2. Append following code.

```
task allTasks {
}

allTasks.dependsOn task0, task1, task2
```

Note: Here you have created allTasks which aggregates task0, task1, and task2.

__ 3. Save the file.

__ 4. Execute allTasks.

```
gradle -q allTasks
```

Notice it displays 'Task #0, Task #1, Task #2' message on the screen.

Part 11 - Using list and .each loop

In this part you will create a list, use each loop to go over it, create directories, and delete directories.

- ___ 1. In the Notepad window, which contains build.gradle code, append following code.
- ___ 2. Append following code.

```
task createDirs {  
    doLast {  
        ext.myDirList = ["dir1", "dir2"]  
        ext.myDirList.each() {  
            new File("${it}").mkdir()  
        }  
    }  
}
```

Note: You have used ext object to create a custom list of string then used each loop to iterate over the items and created directory for each item. `${it}` is a special variable which contains the current item being processed by the each loop.

- ___ 3. Save the file.
- ___ 4. Execute the task.

```
gradle -q createDirs
```

- ___ 5. Get directory list.

```
dir
```

Notice dir1 and dir2 are created.

__ 6. In the Notepad window, which contains build.gradle code, append following code.

```
task removeDirs {  
    doLast {  
        ext.myDirList = ["dir1", "dir2"]  
        ext.myDirList.each() {  
            new File("${it}").delete()  
        }  
    }  
}
```

Note: In the code you are removing the directories which you created previously.
--

__ 7. Save the file.

__ 8. Execute the task.

```
gradle -q removeDirs
```

__ 9. Get the directory list.

```
dir
```

Notice dir1 and dir2 are removed.

__ 10. Close all.

Part 12 - Review

In this lab you utilized core features of groovy in build scripts.

Lab 30 - Build Java Project, Management Package Dependencies, and Run Unit Tests with Gradle

Part 1 - Create directory structure for storing project files

In this part you will create directory structure for storing Java files. It's the same structure used by Maven as well.

- __ 1. Open a Command Prompt.
- __ 2. Switch to the \Workspace\labs\gradle directory.

```
cd C:\Workspace\labs\gradle
```

- __ 3. Create directory structure for storing Java code.

```
mkdir MyProject\src\main\java\hello
```

MyProject is the name of your project. It's the base directory you will use for storing your project files.

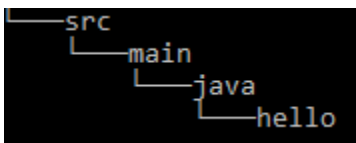
main and java directories must exist with same names. hello is custom Java package name.

- __ 4. Switch to MyProject directory.

```
cd MyProject
```

- __ 5. View directory tree.

```
tree
```



```
├── src
│   ├── main
│   │   ├── java
│   │   │   └── hello
```

Part 2 - Create a Java project

In this part you will write simple Java code. There will be 2 files. One will contain a custom class and second will contain the main function which make use of the custom class.

__1. Create Greeter.java by using Notepad. (Note: Click 'Yes', if prompted to do so)

```
notepad src\main\java\hello\Greeter.java
```

__2. Enter following code.

```
package hello;

public class Greeter {
    public String sayHello() {
        return "Hello world!";
    }
}
```

__3. Save the file and close Notepad window.

__4. Create HelloWorld.java file by using Notepad. (Note: Click 'Yes', if prompted to do so)

```
notepad src\main\java\hello\HelloWorld.java
```

__5. Enter following code.

```
package hello;

public class HelloWorld {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        System.out.println(greeter.sayHello());
    }
}
```

__6. Save the file and close Notepad window.

Part 3 - Create Gradle build script

In this part you will create a Gradle build script.

__ 1. Run Gradle, without creating the build script, and get tasks list.

```
gradle tasks
```

Notice there is no section labeled "Build tasks" containing build, clean, clean, ... tasks.

__ 2. Create build.gradle file. (Note: Click 'Yes', if prompted to do so)

```
notepad build.gradle
```

__ 3. Enter following code.

```
apply plugin: 'java'
```

You have included Java plugin which will make more tasks available to Gradle.

__ 4. Save and close the file.

__ 5. Get Gradle tasks list again.

```
gradle tasks
```

Notice there's a new section available with bunch of useful Java related tasks.

```
Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.
```

Part 4 - Build and Clean the Gradle generated files

In this part you will build the Java project with Gradle, run the generated compiled code, and clean the files.

__1. Build the project using Gradle.

```
gradle build
```

__2. View "build" directory tree.

```
tree build
```

```
C:\Workspace\labs\gradle\MyProject>tree build
Folder PATH listing
Volume serial number is CC19-8684
C:\WORKSPACE\LABS\GRADLE\MYPROJECT\BUILD
|_ classes
|   |_ java
|       |_ main
|           |_ hello
|_ generated
|   |_ sources
|       |_ annotationProcessor
|           |_ java
|               |_ main
|       |_ headers
|           |_ java
|               |_ main
|_ libs
|_ tmp
|   |_ compileJava
|   |_ jar
```

Here are some important directories:

* classes: compiled .class files are generated here.

* libs: jar file is stored here.

__3. Run the compiled class in the jar file.

```
java -cp build\libs\MyProject.jar hello.HelloWorld
```

Notice it displays "Hello World!" message on the screen.

__4. Clean the build generated files.

```
gradle clean
```

__5. Get directory list.

```
dir
```

Notice build directory is gone.

Part 5 - Dependency Management

In this part you will include package dependency and then manage it using the Gradle build script.

__1. Edit HelloWorld.java file.

```
notepad src\main\java\hello\HelloWorld.java
```

__2. Below "package hello;" add following code.

```
import org.joda.time.LocalDateTime;
```

Note: Although you can use native Java classes for obtaining date and time, but just so you can see dependency management you will utilize a 3rd party package, Joda, for obtaining the date and time.

__3. Above "Greeter greeter = new Greeter();" add following code.

```
LocalTime currentTime = new LocalTime();  
System.out.println("The current time is: " + currentTime);
```

__4. Save and close the file.

__5. Try to build the project with Gradle.

```
gradle build
```

Notice the build has failed. It's unable to find LocalTime class dependency.

__6. Edit the build script.

```
notepad build.gradle
```

__7. Append following code.

```
repositories {  
    mavenLocal()  
    mavenCentral()  
}
```

Note: Here you are adding local and online Maven Central repository (<http://search.maven.org>) from where the dependencies will be downloaded from. You can also use other repositories, such as Jcenter(<https://bintray.com/bintray/jcenter>).

Custom repositories can also be defined like this:

```
repositories {  
    maven {  
        url "http://repo.mycompany.com/maven2"  
    }  
}
```


__8. Below repository add following code to define source and target Java version. (Note: this step is optional)

```
sourceCompatibility = 1.7
targetCompatibility = 1.7
```

__9. Add following code to add Joda time 2.2 package dependency.

```
dependencies {
    compile "joda-time:joda-time:2.2"
}
```

__10. Save and close the file.

__11. Run Gradle build.

```
gradle build
```

Notice there are no errors this time.

Part 6 - Using Application Plugin

In this part you will use Application plugin. It makes more tasks available to Gradle. You can use it for executing your application.

__1. Edit the build script file.

```
notepad build.gradle
```

__2. Add following line in the beginning of the file.

```
apply plugin: 'application'
```

__ 3. Add following code in the end of the file.

```
mainClassName = "hello.HelloWorld"
```

Note: Here you have specified the main class which should be executed.

__ 4. Save and close the file.

__ 5. Execute the build script.

```
gradle build
```

__ 6. Run the application.

```
gradle -q run
```

Notice you have used the run task made available by Application plugin. Also notice it displays current date & time and Hello world! message on the screen.

Part 7 - Running unit tests with Grade

In this part you will write a JUnit test and run it with Gradle.

__ 1. Create directory structure for storing unit test.

```
mkdir src\test\java\hello
```

__2. Create a unit test. (Note: Click 'Yes', if prompted to do so)

```
notepad src\test\java\hello\TestGreeting.java
```

__3. Enter following code.

```
package hello;

import org.junit.Assert;
import org.junit.Test;

import hello.Greeter;

public class TestGreeting {
    @Test
    public void testGreeter() {
        Greeter msg = new Greeter();
        Assert.assertEquals("Hello world!",
msg.sayHello());
    }
}
```

__4. Save and close the file.

__5. Run the test.

```
gradle test
```

Notice it displays error message that JUnit is not recognized. You need to add JUnit dependency in order to make it work. You will do it in the next step.

__6. Open build.gradle file.

```
notepad build.gradle
```

__7. In dependencies section, after compile "joda-time:joda-time:2.2", add following line.

```
testCompile "junit:junit:4.12"
```

__ 8. Save and close the file.

__ 9. Run test again.

```
gradle test
```

Notice test is executed successfully and no error is displayed.

__ 10. View test result.

```
notepad build\test-results\test\TEST-hello.TestGreeting.xml
```

Notice testGreeter test case is listed. It also shows time it took to run the test.

__ 11. Close the Notepad window and command prompt.

```
exit
```

Part 8 - Review

In this lab you built a Java Project, managed dependencies, and ran unit tests with Gradle.

