Lab 02: Using Primitive Data Types

Objective

This lab will begin to introduce the Java syntax to the new Java programmer. You will create your first Java program and experiment with using different primitive data types.

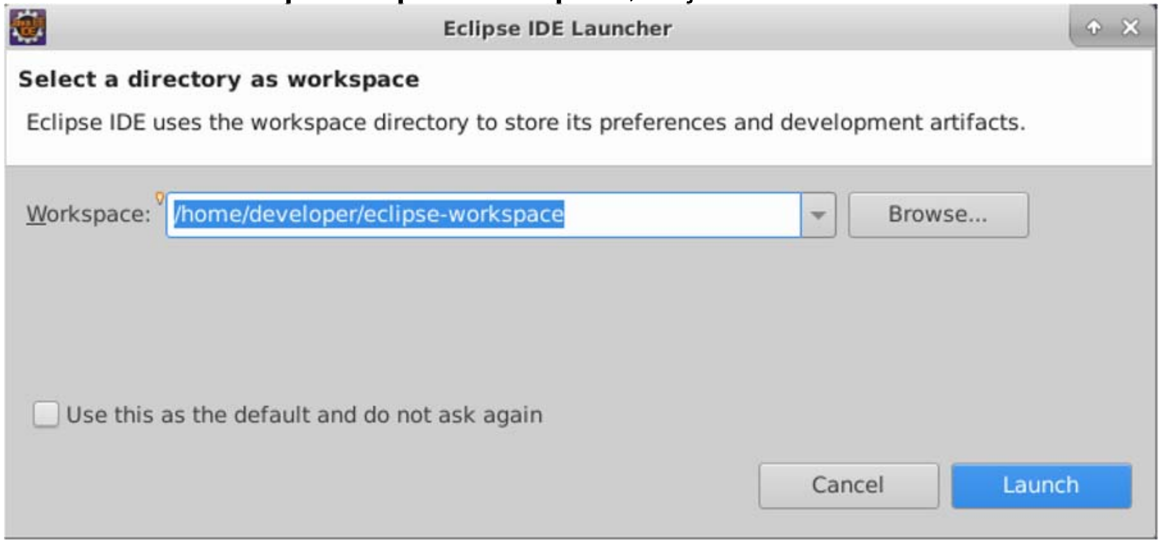
Overview

In this lab you will create a new Java project and your first Java class. While the details of object-oriented programming and Java class structure are yet to come, this initial Java class will allow you to begin writing some basic Java code.

Step by Step Instructions

Create a Workspace, Project, and Package

1. Launch your IDE and use the default workspace for your lab exercises. The default workspace location should be left unchanged in this environment and it is: **/home/developer/eclipse-workspace**, so just click the **Launch** button.



Eclipse IDE Launcher

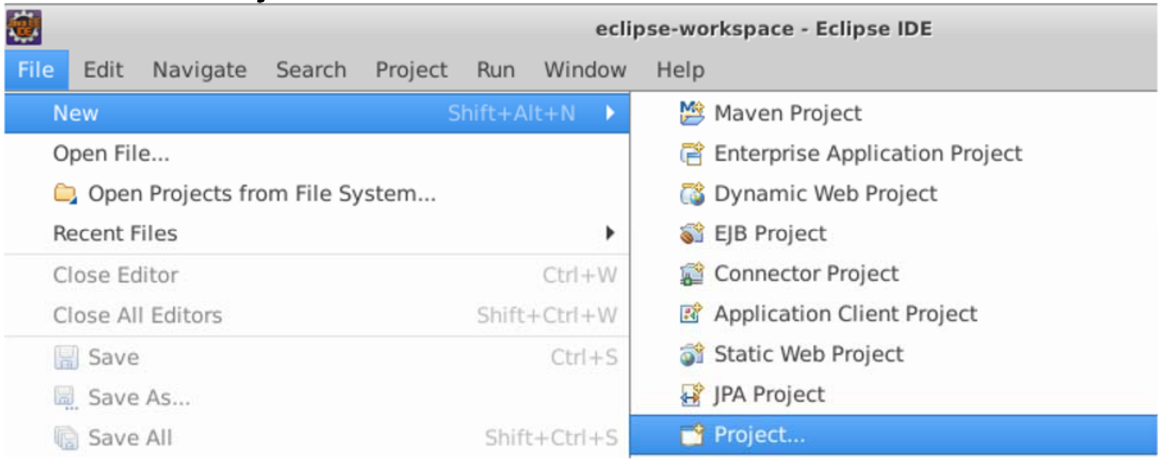
Select a directory as workspace

Eclipse IDE uses the workspace directory to store its preferences and development artifacts.

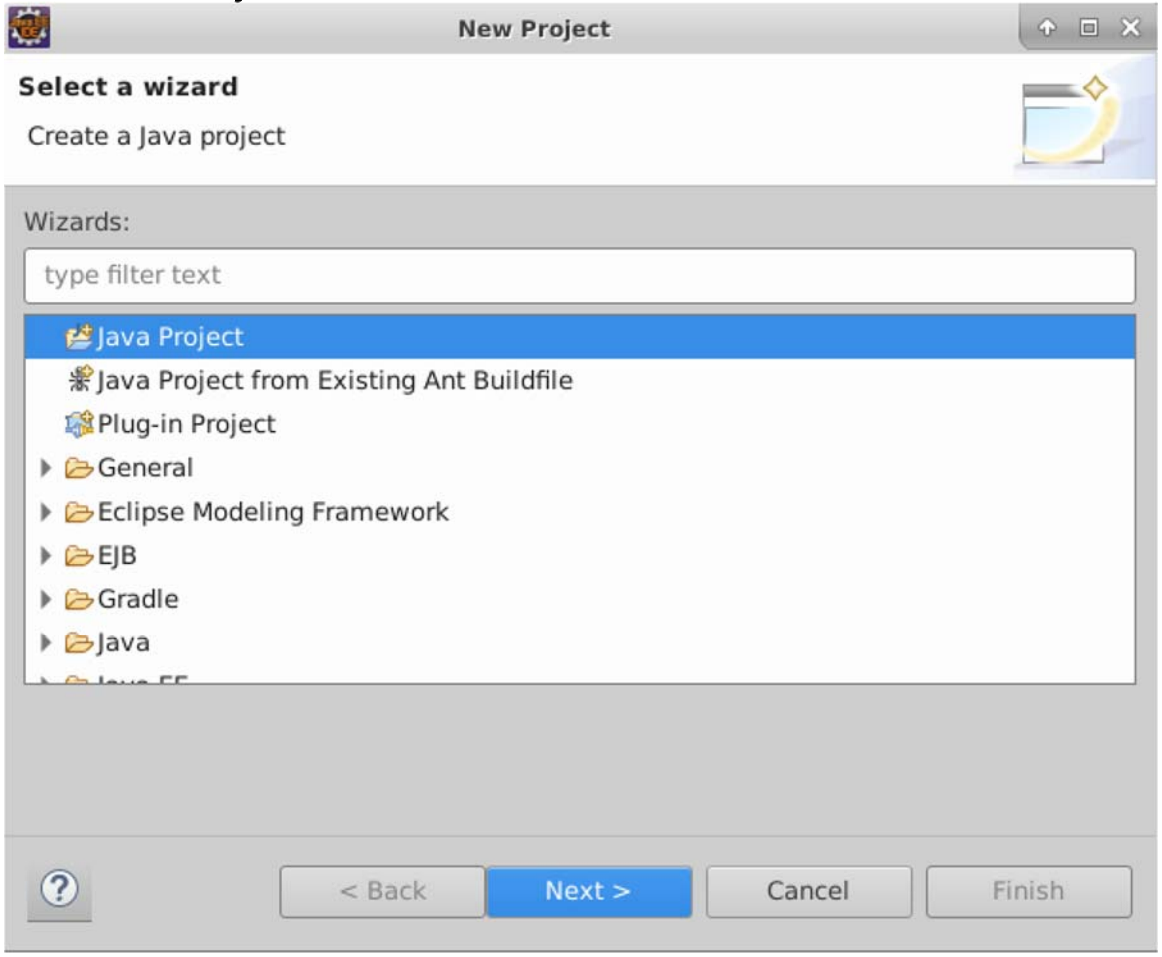
Workspace: /home/developer/eclipse-workspace

2. Create a new Java project for the first few exercises. Name the project **ClassExercises**.

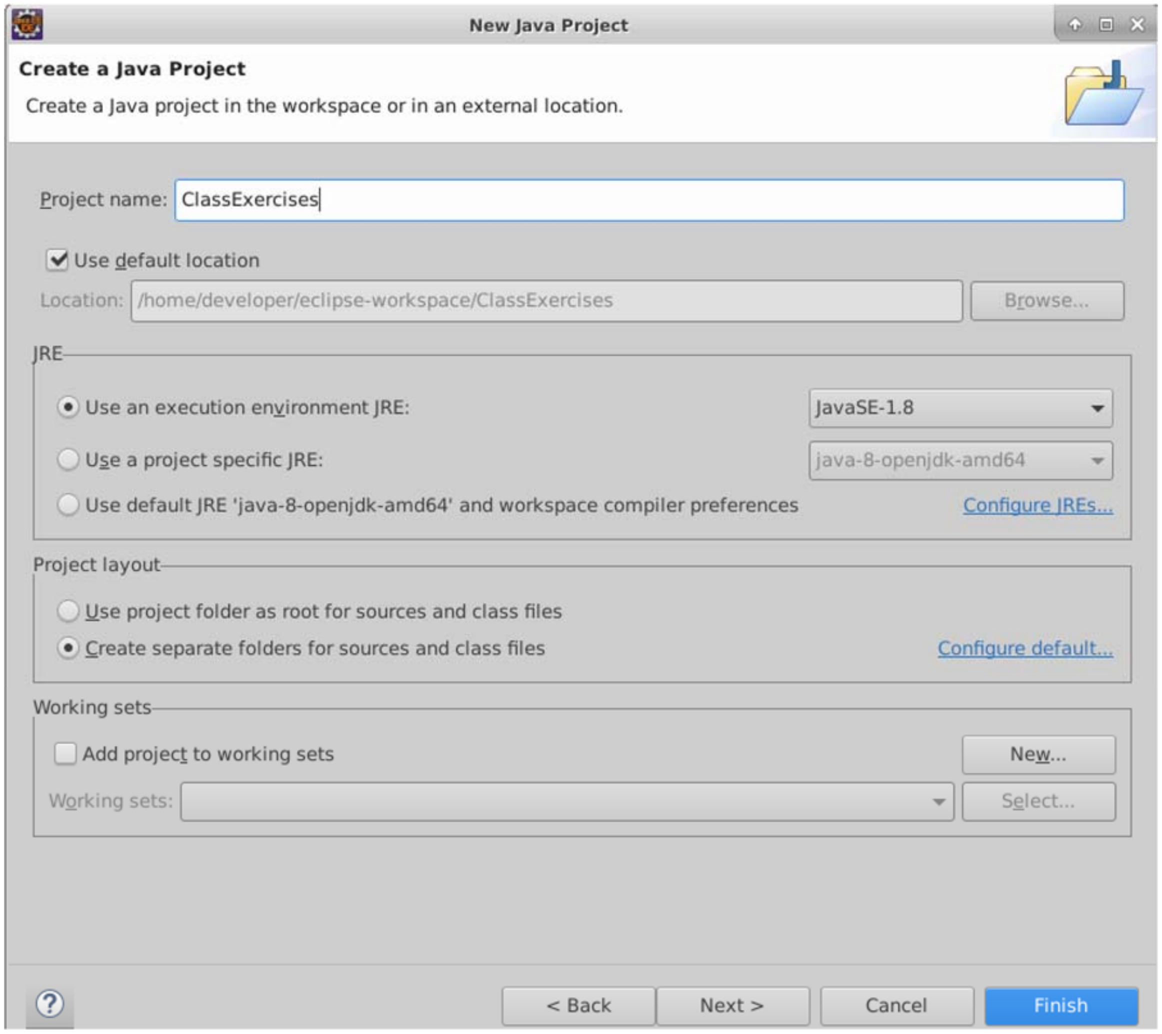
    a. Select **File** -> **Project…**



    b. Select **Java Project**



c. Enter **ClassExercises** as the Project Name and click the **Finish** button.



Create a Java Project

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

Project name: ClassExercises

[✓] Use default location

Location: /home/developer/eclipse-workspace/ClassExercises

JRE

(●) Use an execution environment JRE: JavaSE-1.8

( ) Use a project specific JRE: java-8-openjdk-amd64

( ) Use default JRE 'java-8-openjdk-amd64' and workspace compiler preferences

Project layout

( ) Use project folder as root for sources and class files

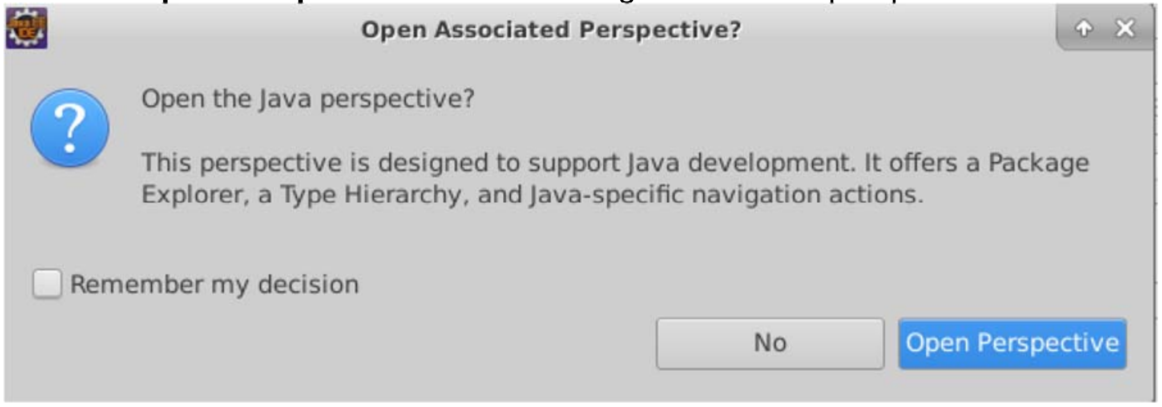
(●) Create separate folders for sources and class files

Working sets

[ ] Add project to working sets

Working sets:

d. Click the **Open Perspective** button to change to the Java perspective.

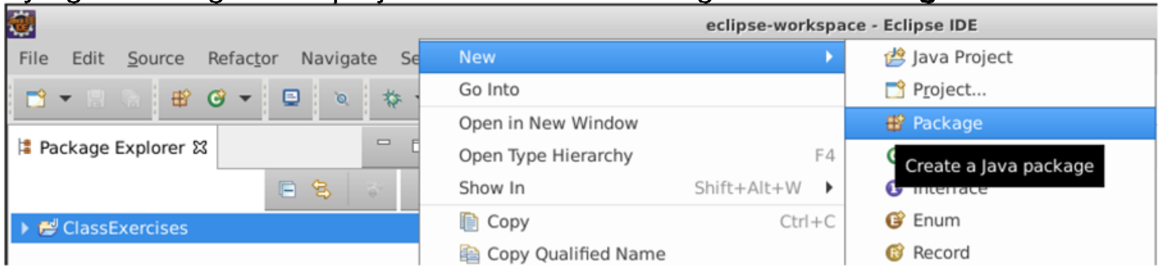


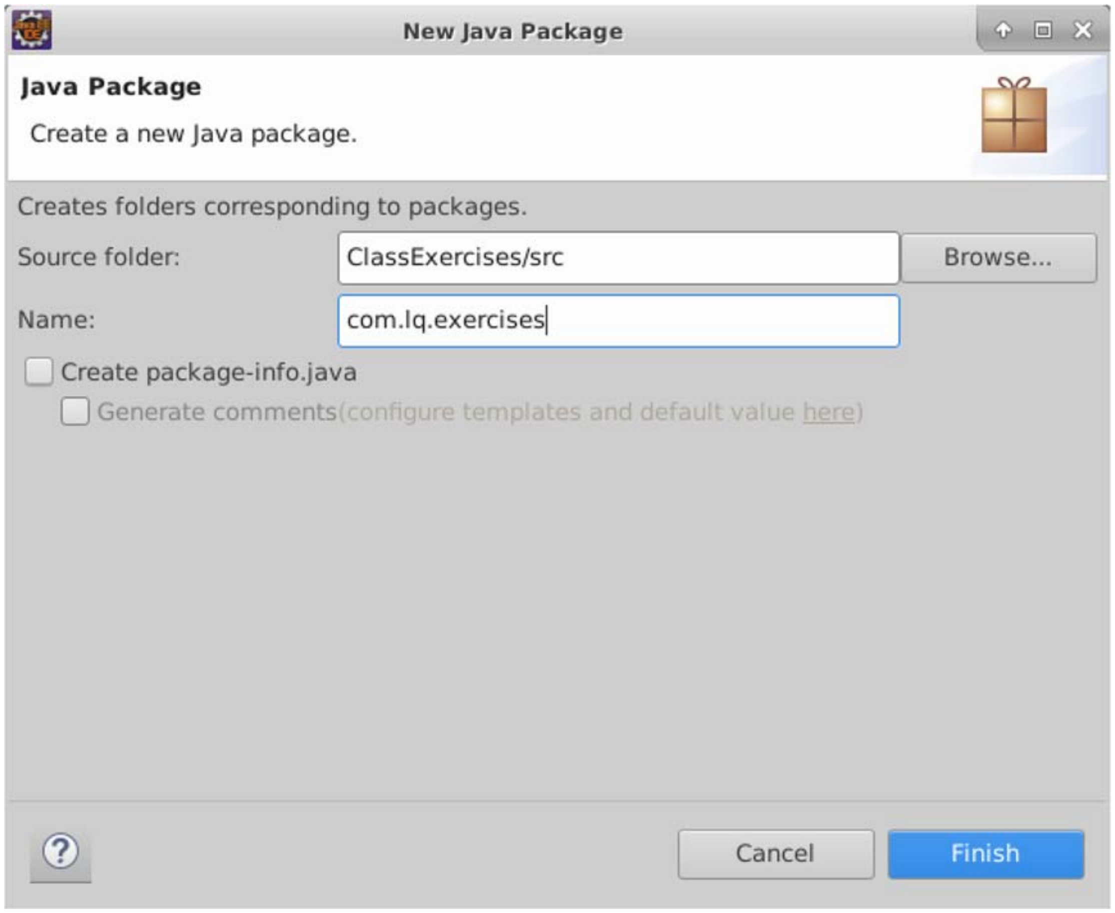
Open the Java perspective?

This perspective is designed to support Java development. It offers a Package Explorer, a Type Hierarchy, and Java-specific navigation actions.

[ ] Remember my decision

3. Create a package named **com.lq.exercises**. This can be done in most IDEs by right-clicking on the project name and selecting **New > Package**.




New Java Package

Java Package

Create a new Java package.

Creates folders corresponding to packages.

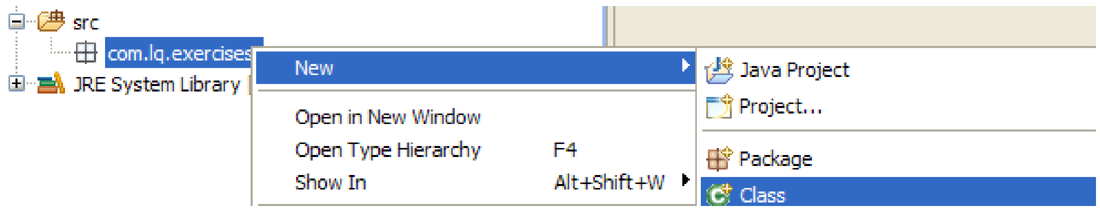
Source folder: ClassExercises/src

Name: com.lq.exercises

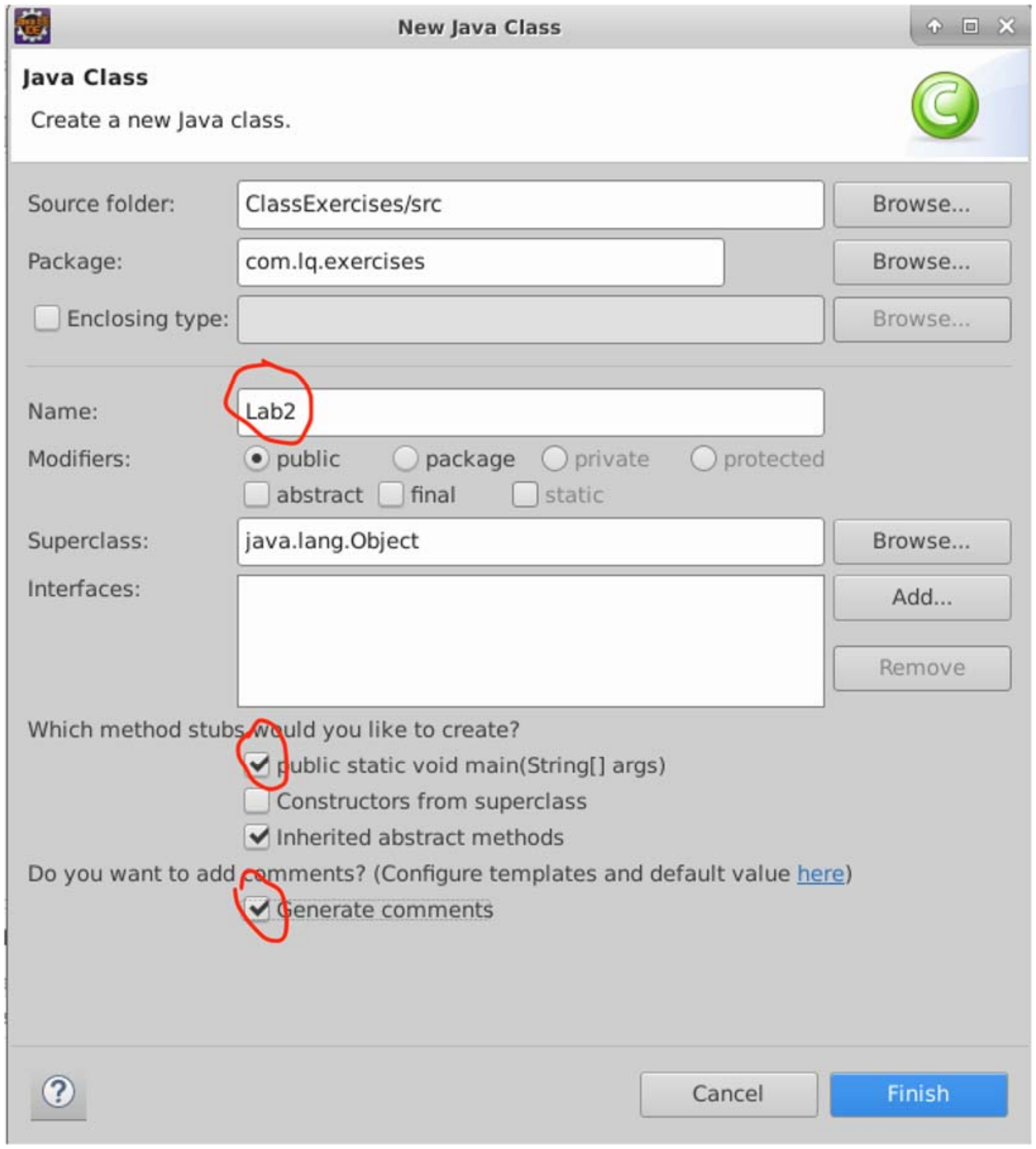
Create package-info.java

Generate comments(configure templates and default value here)

4. Create a new class named **Lab2** in the **com.lq.exercises** package. This can be done in most IDEs by right-clicking on the package and selecting **New > Class**.



The new class wizard will appear. Enter the name of the class, check the box for a main method, and check the generate comments method. Click the **Finish** button.


New Java Class

Java Class

Create a new Java class.

Source folder: ClassExercises/src

Package: com.lq.exercises

☐ Enclosing type:

Name: Lab2

Modifiers: (● public) ○ package ○ private ○ protected
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

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

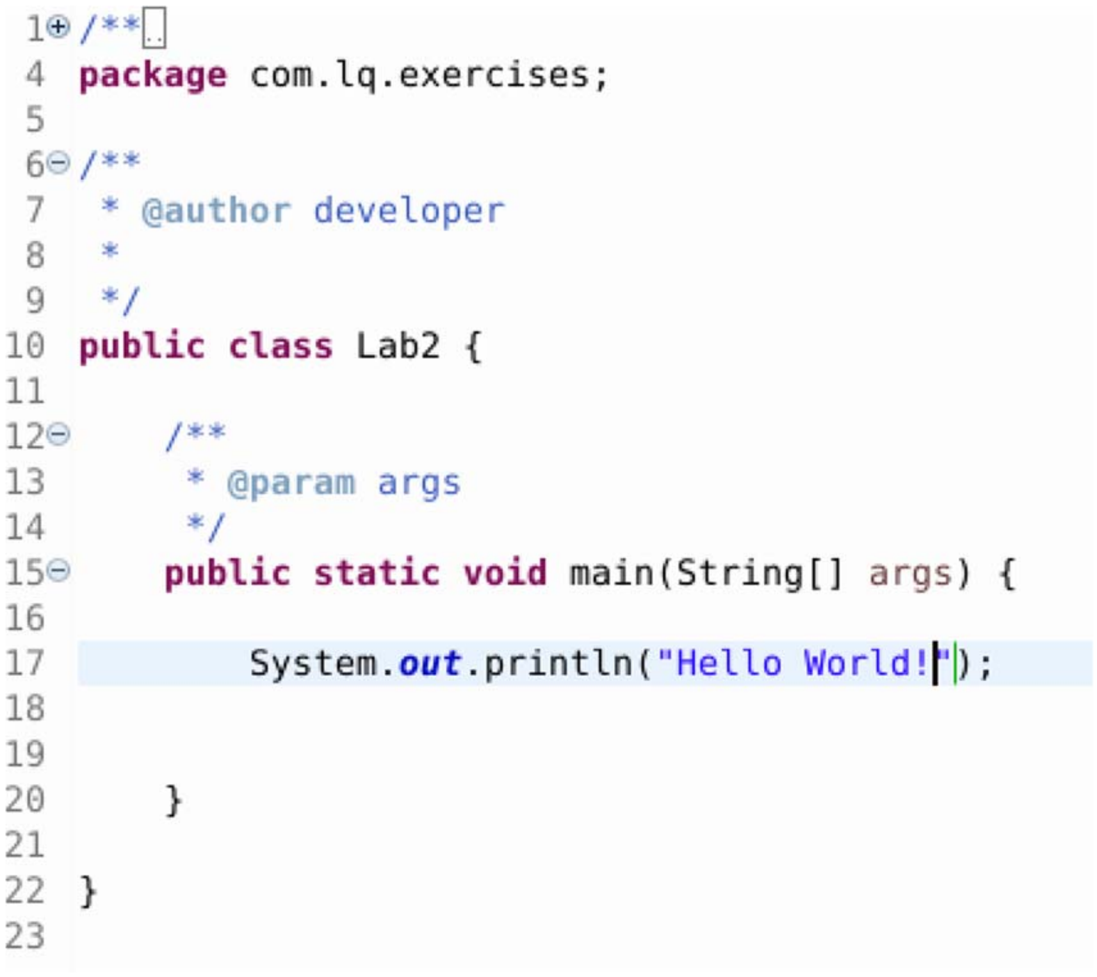
Do you want to add comments? (Configure templates and default value here)
☑ Generate comments

5. The **Lab2** class will open in the editor. It will contain a beginning curly brace '{'and an ending curly brace '}' for the class definition. Proceed with the lab exercises that follow.

Implement the main() method

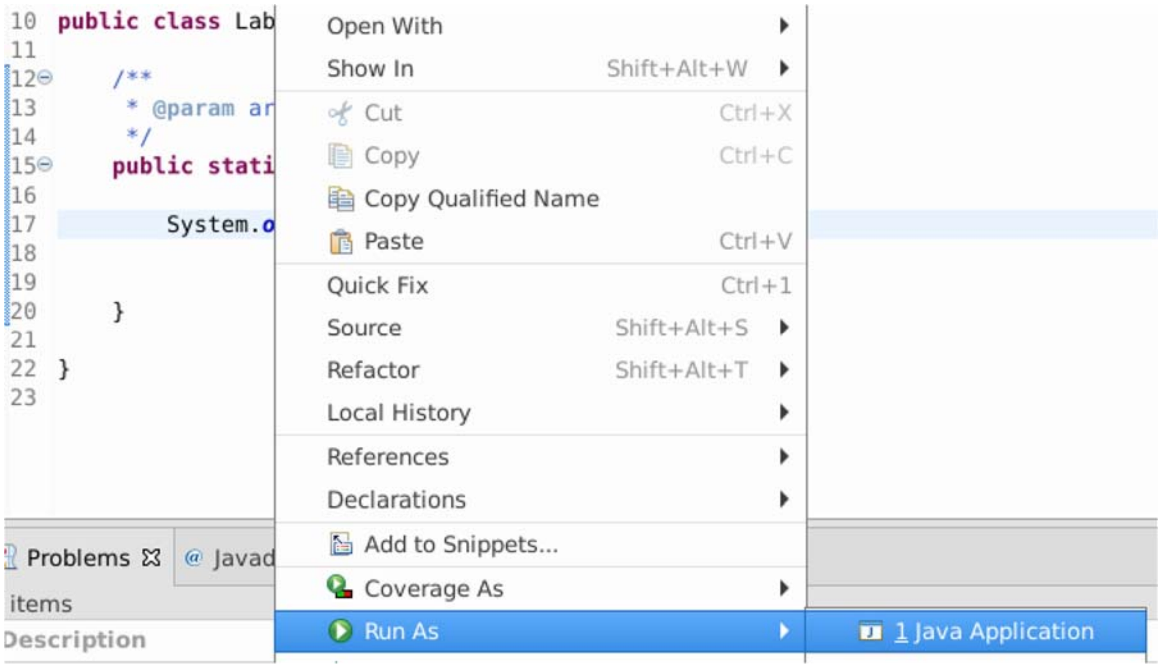
6. Remove the TODO comment. It is there only to remind you that the method was automatically created for you and that you must implement what the method is to do.

7. Add a print statement to your `main()` method that says, "**Hello World!**". We will cover the details of how this all works later but for now it will look like the following:



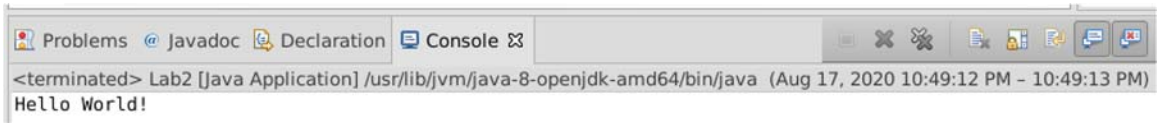
```java
/**
package com.lq.exercises;

/**
 * @author developer
 *
 */
public class Lab2 {

    /**
     * @param args
     */
    public static void main(String[] args) {

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


    }

}

```

8. Save your changes and execute your small Java program. In most IDEs this can be accomplished by right-clicking on the class name and selecting **Run As > Java Application** or by clicking the run button on the toolbar.



```
public class Lab

    /**
     * @param ar
     */
    public stati

        System.o


    }

}
```

9. View the results in the console view. It should be similar to the following:



```
<terminated> Lab2 [Java Application] /usr/lib/jvm/java-8-openjdk-amd64/bin/java  (Aug 17, 2020 10:49:12 PM – 10:49:13 PM)
Hello World!
```

<u>Define Primitive Data Types</u>

10. At the beginning of the main() method, declare the following variables with the specified characteristics:

    a. An int with the name **width** and no initial value.

b. An int with the name **height** and no initial value.
c. An int with the name **area** and no initial value.
d. A double with the name **radius** and an initial value of **10.0.**
e. A double with the name **pi** and an initial value of **3.14.**
f. A boolean with the name **result** and an initial value of **true.**

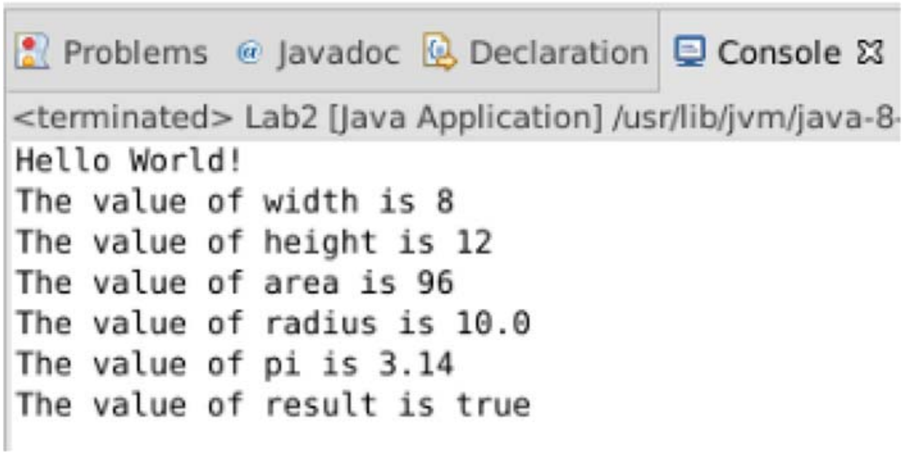
In the body of the main() method, perform the following variable assignments:

g. Assign a value of **8** to **width.**
h. Assign a value of **12** to **height.**
i. Assign a value of **96** to **area.**

11. Near the bottom of the main() method, print the values of each variable using `System.out.println().` Use the following as a guide:

```
System.out.println("The value of width is " + width);
```

12. Execute your program. Your output should be similar to the following:



```
<terminated> Lab2 [Java Application] /usr/lib/jvm/java-8
Hello World!
The value of width is 8
The value of height is 12
The value of area is 96
The value of radius is 10.0
The value of pi is 3.14
The value of result is true
```

Define Arrays

13. Near the top of your main() method, define the following arrays:

j. An array of 12 **ints** named **daysInMonths**.
k. An array of 12 **String** references named **monthNames** – initialize this array at the time it is declared with the names of the 12 months (refer to your course book for the syntax).

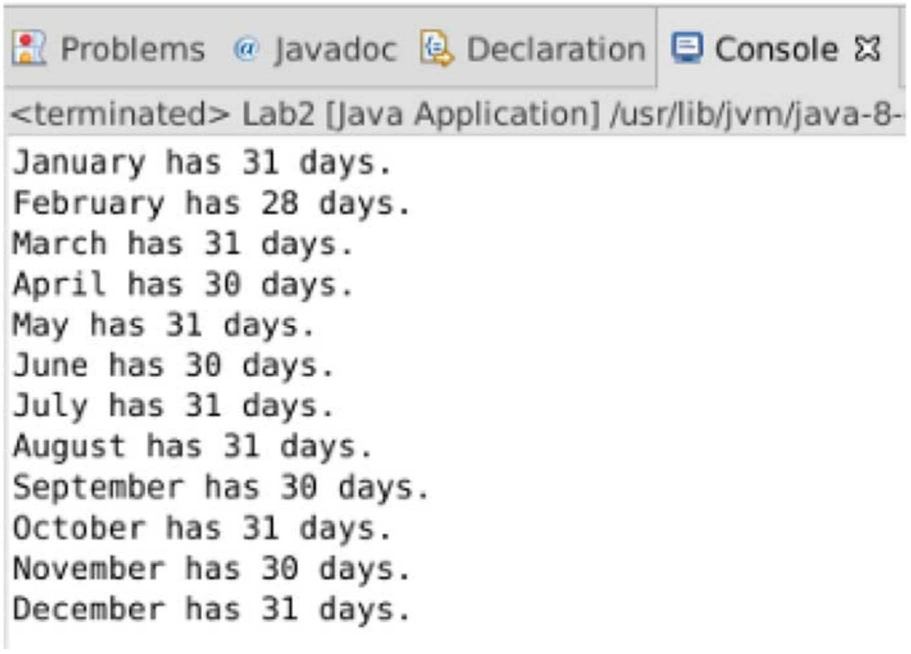
14. Write 12 lines of code to assign the number of days in each month to the **`daysInMonths[]`** array elements (do not worry about leap year!). See output in step 16 below for days in months.

15. Write a print statement for each month that will display the name of the month and the number of days it contains. Use the following as a guide:

```
System.out.println( monthNames[0] + " has " + daysInMonths[0] + " days." );
```

If you are already familiar with loops, you may use them. If not, then use 12 output statements.

16. Execute your program. Your output should be similar to the following:



```
<terminated> Lab2 [Java Application] /usr/lib/jvm/java-8-
January has 31 days.
February has 28 days.
March has 31 days.
April has 30 days.
May has 31 days.
June has 30 days.
July has 31 days.
August has 31 days.
September has 30 days.
October has 31 days.
November has 30 days.
December has 31 days.
```

Lab 03: Flow Control

Objective

This lab will continue to introduce the Java syntax to the new Java programmer. You will focus on understanding and practicing with flow control syntax.

Overview

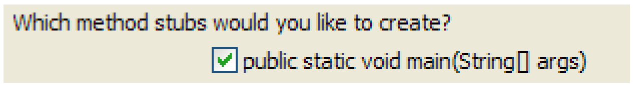
In this lab you will experiment with the following:

- Flow control using if
- Loops using for and while
- Flow control using break and continue
- Flow control using switch

Step by Step Instructions

Using Java Flow Control

1. Create a new class named **Lab3** in the **com.lq.exercises** package. Ensure that the new **Lab3** class has a `main()` method. This can usually be added automatically in the IDE when you are creating the class. Use the following as a guide:



Which method stubs would you like to create?

✓ public static void main(String[] args)

Exercise 1: for loop

2. Create an array of **String** references named **daysOfWeek** and initialize it with the names of the days of the week starting with Sunday.
3. Write a for loop to print out each element of the array. Do this two ways – the first using the standard for loop, the second using the for-next construct.
4. Write another for loop to print out each element of the array in reverse. Note that the only way this can be done is by using the standard for loop.

Exercise 2: while loop and continue

5. Create a **while** loop that will print out only even numbers between 1 and 20.
6. Use the **continue** statement and an **if** statement to skip printing the odd numbers (i.e. increment the counter by 1 each time through the loop).

Exercise 3: for loop

7. Create a **for** loop that will print out all numbers between 1 and 100 except 50 thru 60.

Exercise 4: switch statement

8. Create a **while** loop that will execute the following code 12 times using a counter that starts at 1 and is incremented by 1 each time through the loop.

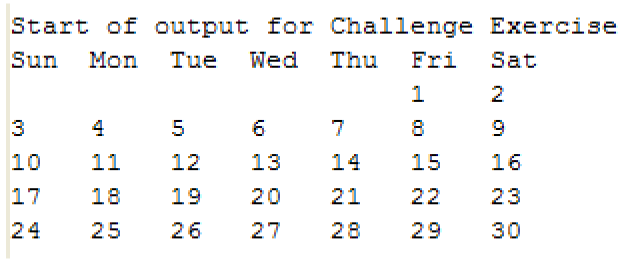
9. Within the while loop, create a switch statement that will print out the number of days in each of the months of the year.

10. Use fall through capabilities so that you only have one print statement for all of the months that contain 31 days. Do not worry about leap years.

11. Print out the name of the month as well. You can copy and paste your monthNames array from the previous lab to help make this easier.

Challenge Exercise

12. Create the necessary variables, loops and flow control to print March as a calendar. The calendar will have:
    a. A heading with the days of the week
    b. 5 rows with the days of the month in the correct column
    c. Assume that March 1 starts on a Friday
    d. Each week on the calendar should flow from Sunday – Saturday (left to right).
    e. You can use the function `System.out.print()` to print information to the console without adding a carriage return.
13. The output should look as follows:



```
Start of output for Challenge Exercise
Sun  Mon  Tue  Wed  Thu  Fri  Sat
                         1    2
3    4    5    6    7    8    9
10   11   12   13   14   15   16
17   18   19   20   21   22   23
24   25   26   27   28   29   30
```

Lab 05.1: Creating Classes

Objective

This lab will provide practice in creating classes, methods and attributes. You will also implement several small methods in your new class during this exercise.

Overview

In this lab you will:

- Create new classes, attributes and methods
- Implement your methods

Step by Step Instructions

Exercise 1: Creating Classes and Attributes

1. Create a new class named **Box** in the **com.lq.exercises** package.

2. A **Box** has three pieces of information
   a. height
   b. width
   c. length

3. Create **private** attributes in your **Box** class for each of these pieces of information. Use `double` as the data type for each attribute.

4. Create **public** methods to set each attribute and retrieve (get) each attribute. Right click on the editor window and select the Source menu. From that menu select “Generate Getters and Setters”. Check the “Select all” and “Generate Comments” boxes. From the location dropdown choose last member and then click finish.

Exercise 2: Create Constructors

5. Create two constructors for the **Box** class.

    a. The first should accept 3 parameters and assign them to the **length**, **width** and **height** attributes respectively. Note: there is a wizard on the "Source" menu that will create this without typing.

    b. The second should accept 1 parameter and assign it to all 3 attributes (creating a cube). This constructor should make a call to the other constructor to avoid duplication of code.

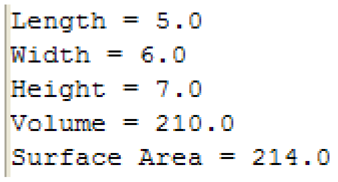
Exercise 3: Create Business Methods

6. Create a method named `getVolume` that accepts no parameters and returns a `double` containing the volume of the **Box**. [Hint: Volume is height * width * length.]
7. Create a method named `getSurfaceArea` that accepts no parameters and returns a `double` containing the surface area of the **Box**. [Hint: Surface area is the sum of the areas of the 6 sides of the **Box**.]

Exercise 4: Create a printBox Method

8. Create a method named `printBox` which does the following:
    a. If any one of the 3 attributes is less than or equal to 0, it will print a message stating that the box contains invalid properties.

    b. If all 3 sides are set correctly (i.e. greater than 0), it will print a message in the following form:



```
Length = 5.0
Width = 6.0
Height = 7.0
Volume = 210.0
Surface Area = 214.0
```

Note: we will not test any of our code at this point. Once we cover creating objects and calling class methods, we will ensure that all the code is operating correctly.

Challenge Exercises:

Refactor your code to eliminate duplicate logic.

Lab 05.2: Creating Calculator Classes

Objective

This lab will introduce the first of the larger systems that you will be programming. This system is a set of classes that perform basic math-oriented calculations. You will utilize the existing Java and **Math** class functionality to implement these classes and it will provide practice in creating classes, methods and attributes.

Overview

In this lab you will:

- Create the necessary packages
- Create the necessary calculator classes
- Add the appropriate attributes
- Provide the appropriate method definitions (minus the actual code to make them work at this point)

Step by Step Instructions

Create a Java Project and Package

1. Create a new Java Project within the same workspace named **Calculators**.

2. Create a package named **com.javaoo.calculators**.

Exercise 1: Creating the Calculator classes

3. Create each of the following classes in the **com.javaoo.calculators** package:
    a. **BasicCalculator**
    b. **ScientificCalculator**
    c. **TrigonometricCalculator**

Exercise 2: Adding Attributes and Methods

4. For each of the calculator classes, add the necessary attributes and methods as specified below.

5. **BasicCalculator**:

   a. Declare the following **public** methods
      i. `add()`
      ii. `subtract()`
      iii. `multiply()`
      iv. `divide()`

   b. Each method must accept two parameters, both of type `double`

   c. Each method must return a `double.` Add code to the return statement that calculates and returns the correct value. Example:

```
public final double multiply(double x, double y) {
        return x * y;
}
```

6. **ScientificCalculator**:

   d. Must declare the following attributes:
      i. A `double` named `PI` to hold the value of **PI (3.14159).** This attribute will be shared by all instances of the class and will be constant so it should be declared as `static` and `final`. Since it is a final, it can have public visibility so that anyone can use it.
      ii. A `double` named `holdValue` to hold a value in memory. Since it will be managed within the **ScientificCalculator** class, it should have `private` visibility.

   e. Declare the following **public** methods
      i. `exp()` which has one parameter of type `double` and returns a `double` [This method will be used to calculate e^x]
      ii. `log()` which has one parameter of type `double` and returns a `double` [This method will be used to calculate `ln` x]
      iii. `putValueInMemory()` which has one parameter of type `double` and returns a `void.` Implement this method.
      iv. `getValueFromMemory()` which has no parameters and returns a `double` Implement this method

f. We will not provide the details of each method in this lab. In order for your code to compile, add the following single statement to **each method block that returns a `double`**:

```
     return(0);
```

7. **TrigonometricCalculator**:

   g. Declare the following **public** methods
      i. `sine()`
      ii. `cosine()`
      iii. `tangent()`
      iv. `arcsine()`
      v. `arccosine()`
      vi. `arctangent()`

   h. Each method must accept one parameter of type `double`
   i. Each method must return a `double`
   j. We will not provide the details of each method in this lab. For your code to compile, add the following single statement to each method block:

```
     return(0);
```

8. **BRAIN TEASER: Could and should any of these classes or methods be made *static*?**

Lab 06: Creating Objects

Objective

This lab will provide practice in creating objects and calling methods and constructors.

Overview

In this lab you will:

- Create a test class
- Instantiate several objects using constructors
- Utilize get and set methods
- Utilize other business methods
- Fine tune the behavior of the **Box** Class

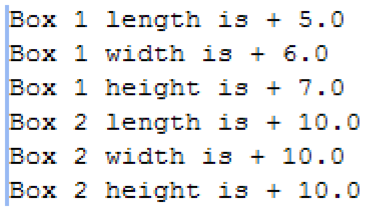
Step by Step Instructions

Exercise 1: Creating Objects

1. Create a new class named **BoxDriver** in the **com.lq.exercises** package. Ensure that this class has a `main()` method defined.

2. In the main method of **BoxDriver**, create two **Box** objects using the keyword `new`.
    a. A box named **box1** with a length of 5, a width of 6 and a height of 7
    b. A box named **box2** where all three sides are 10

3. Using the get methods, print out the value for each attribute of both boxes. Ensure that the attributes were set correctly by the **Box** constructors. Execute the main method for **BoxDriver**. The output should be like the following. If there are any errors in your **Box** code at this point, correct them and re-run the test.



```
Box 1 length is + 5.0
Box 1 width is + 6.0
Box 1 height is + 7.0
Box 2 length is + 10.0
Box 2 width is + 10.0
Box 2 height is + 10.0
```

Exercise 2: Test Business Methods

4. Ensure that your set methods work correctly. Call each set method for `box1` changing the values to the following and then use the get methods to ensure that the attributes were set correctly.
    a. Length = 3
    b. Width = 4
    c. Height = 5

5. Call the `getVolume()` and `getSurfaceArea()` methods on `box1` and ensure they are functioning correctly. If not, fix the code and re-test.

6. Execute the `printBox()` for `box1`. Ensure that it is functioning correctly. If not, fix the code and re-test.

7. Using the `setLength()` method, change the length of `box1` to -5. Execute the `printBox()` method again. What happens? You should receive a message stating that the box contains invalid attributes.

8. In order to avoid setting attributes to invalid values, re-code the set methods of the **Box** class to only accept values greater than 0. If a value less than or equal to zero is detected, print an error message like the following:

```
System.out.println("Length must be greater than 0");
```

9. Re-run your **BoxDriver** main method. You should now receive an error message when you try and set the **length** to -5. Test the other two set methods in the same manner.

Challenge Exercise

10. Change the **Box** constructors so that new boxes can only be created with valid attributes. If a constructor detects an invalid parameter, use a default value of 1 (one).

Lab 07: Creating Calculator Objects

Objective

This lab will continue to work with the **Calculator** classes you created earlier. This system is a set of classes that perform basic math-oriented calculations. You will utilize the existing Java and **Math** class functionality to implement these classes and it will provide practice in creating classes, methods and attributes.

Overview

In this lab you will:

- Implement your methods
- Create a test class
- Test all of your methods

Step by Step Instructions

Exercise 1: Implement methods

Currently, most methods in the **Calculator** classes return 0. Using the functionality provided in the **Math** class, implement each of the methods by following these instructions.

1. **ScientificCalculator:**

    a. Use the **Math** class to implement the following methods.
        i. `exp()` which has one parameter of type `double` and returns a `double` [This method will be used to calculate e^x]
        ii. `log()` which has one parameter of type `double` and returns a `double` [This method will be used to calculate `ln x`]

2. **Trigonometric Calculator:**

    a. Use the **Math** class to implement the following methods.
        i. `sine()`
        ii. `cosine()`
        iii. `tangent()`
        iv. `arcsine()`
        v. `arccosine()`
        vi. `arctangent()`

3. **Test your work:**

   a. Create a new class named **CalculatorDriver** in the **com.javaoo.calculators** package.
   b. Ensure that this class has a `main()` method.
   c. Instantiate each of your **Calculator** classes in the `main()` method and call each of the methods above to ensure they are working correctly. Use print statements to verify correct operation.

Lab 09.1: Working with Inheritance

Objective

This lab will provide some practice in using inheritance. You will also experiment with overriding methods and using information from an inherited class.

Overview

In this lab you will:

- Create a new class that extends an existing class
- Implement methods using overriding
- Add new functionality
- Create objects and test your new code

Step by Step Instructions

Exercise 1: Using the extends keyword

1. Create a new class named **Cube** that extends **Box** and place it in the **com.lq.exercises** package in the **ClassExercises** project.

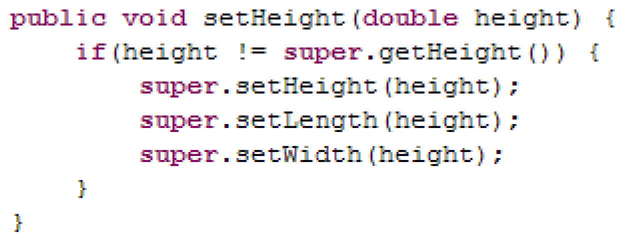
2. A **Cube** is like a **Box** however all three dimensions of a **Cube** must be the same. We simulated this in our **Box** class by providing a constructor that would set all 3 sides to the same value. We will formalize this here. We don't really need any new attributes since we inherit everything we need from **Box**.

3. The first thing we need to do is define a constructor for **Cube**. Since a **Cube** must have all 3 sides set to the same value, our constructor only needs one parameter. We do not need to set any data in the **Cube** class so all we need is a call to our super class constructor [Hint: Use the `super` keyword as the first line in the **Cube** constructor].

4. **Cube** inherits all the `public` methods from **Box**. However, if we set the `length` of a **Cube** to a new value, we also want to guarantee that the `height` and `width` are updated as well. Override each of the following methods in the **Cube** class to guarantee this functionality. (Remember, an overridden method must have the exact same signature as the base class method it is overriding).

   a. `setLength()`
   b. `setWidth()`
   c. `setHeight()`

Here is what it might look like for one of the methods. Note that the author chose to only change the value if a different one has been passed in.



```
public void setHeight(double height) {
    if(height != super.getHeight()) {
        super.setHeight(height);
        super.setLength(height);
        super.setWidth(height);
    }
}
```

Exercise 2: Adding Functionality to Cube

5. Add two new methods to get and set the sides of the **Cube** to a value. These methods work in a similar manner to the **Box** methods but will only be available to cubes.

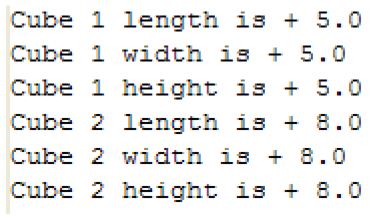
    a. Create a `setSide()` method that takes one parameter of type `double` and returns a `void`. Ensure that the sides are set to a valid value (>0).
    b. Create a `getSide()` method that takes no parameters and returns a `double`.

Exercise 3: Try out the Cube class

6. Create a new class named **CubeDriver** in the **com.lq.exercises** package. Ensure that this class has a `main()` method defined.

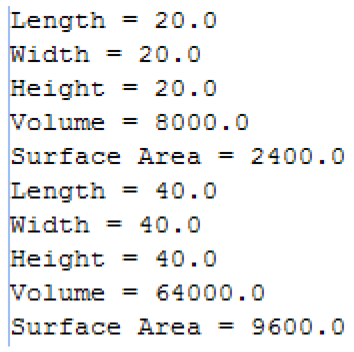
7. In the main method of **CubeTest**, create two **Cube** objects using the keyword `new`.
    a. A cube named **cube1** with all sides set to 5
    b. A cube named **cube2** with all sides set to 8

8. Using the get methods, print out the value for each attribute of both cubes. Ensure that the attributes were set correctly by the **Cube** constructor. Execute the `main()` method for **CubeDriver.** The output should be similar to the following. If there are any errors in your **Cube** code at this point, correct them and re-run the test.



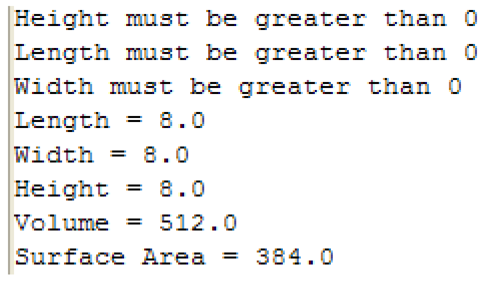
```
Cube 1 length is + 5.0
Cube 1 width is + 5.0
Cube 1 height is + 5.0
Cube 2 length is + 8.0
Cube 2 width is + 8.0
Cube 2 height is + 8.0
```

9. Ensure that your set methods work correctly. Call the `setLength()` method for **Cube** `cube1` and change the `length` to 20. Use the inherited `printBox()` method to ensure that all sides were set correctly. Do the same test again calling the `setSide()` method with a value of 40. Your output should look as follows:



```
Length = 20.0
Width = 20.0
Height = 20.0
Volume = 8000.0
Surface Area = 2400.0
Length = 40.0
Width = 40.0
Height = 40.0
Volume = 64000.0
Surface Area = 9600.0
```

10. Using the `setWidth`() method, change the width of **Cube** cube2 to -5. Did you receive the appropriate error message? Execute the `printBox()` method again and ensure that the value was not changed. Your output should look as follows:



```
Height must be greater than 0
Length must be greater than 0
Width must be greater than 0
Length = 8.0
Width = 8.0
Height = 8.0
Volume = 512.0
Surface Area = 384.0
```

<u>**Challenge Exercises:**</u>

Refactor your code to eliminate duplicate logic.

Experiment with different algorithms so that you don’t receive 3 error messages when you try and change the size of a cube to an invalid value. There are many ways of handling this situation. Explore different techniques.

Is there any way that inheritance might be put to good use in your **Calculator** classes? Write a driver to exercise your Calculator classes.

Lab 09.2: Creating an Inventory System

Objective

This lab will introduce the second of the larger systems that you will be creating. This system is a set of classes that manages an inventory. You will define a class hierarchy using inheritance to manage an inventory for a CD/Book store.

Overview

In this lab you will:

- Create the necessary packages
- Create the necessary system classes
- Add the appropriate attributes
- Provide the appropriate method definitions
- Utilize inheritance

Step by Step Instructions

Description of the Inventory System

This system will model a store that sells books and CDs and needs to create a system to track its inventory.

All items in inventory need to track a title, price and quantity. This will be managed by a generic class named **Item**.

For books, we also need to know the author, publisher and category. This will be managed by a **Book** class that inherits from **Item**.

For pop music CDs we also need to know the artist and release date. This will be managed by a **CD** class that inherits from **Item**.

The artists on the pop music CDs are assumed to be bands with a band name and several members. The store wants to track the band name, each member's name, and the musical instrument each member plays. This will be managed by a standalone **Artist** class.

For classical CDs we also need to know the composer, performer(s), location of recording and release date. This will be managed by a **ClassicalCD** class that inherits from **Item**.

Create a Java Project and Package

1. Create a new Java Project within the same workspace named **Store**.

2. Create a package named **com.javaoo.store**.

Exercise 1: Creating the Inventory System Classes

3. Create each of the following classes in the **com.javaoo.store** package:
    a. **Item**
    b. **Artist**
    c. **Book** which is a sub-class of **Item**
    d. **CD** which is a sub-class of **Item**
    e. **ClassicalCD** which is a sub-class of **Item**

Exercise 2: Adding Attributes and Methods

4. For each of the inventory system classes, add the necessary attributes and methods as specified below.

5. **Item**:

    a. Declare the following **private** attributes.
        i. `title` which is a `String`
        ii. `price` which is a `double`
        iii. `quantity` which is an `int`
    b. Create getter and setter methods for each of the attributes. In most IDEs, this can be done quite easily by right-clicking in the editor and choosing **Source > Generate Getters and Setters**. If you have any difficulty, please ask your instructor.

6. **Artist**:

    a. Declare the following **private** attributes.
        i. `name` which is a `String`
        ii. `memberNames[]` which is an array of 20 `Strings`
        iii. `memberInstruments[]` which is an array of 20 `Strings`
    b. For now, create a getter and setter method for the **name attribute only**

7. **Book**:

    a. Declare the following **private** attributes.
        i. `author` which is a `String`
        ii. `publisher` which is a `String`
        iii. `category` which is a `String`
    b. Create getter and setter methods for each of the attributes.

8. **CD**:

    a. Declare the following **private** attributes.
        i. `artist` which is an object of type `Artist`
        ii. `releaseDate` which is an object of type `Date`
    b. You will need to import the `java.util.Date` class
    c. Create getter and setter methods for each of the attributes.

9. **ClassicalCD**:

    a. Declare the following **private** attributes.
        i. `composer` which is a `String`
        ii. `performers[]` which is an array of 5 `Strings`
        iii. `recordingLocation` which is a `String`
        iv. `releaseDate` which is an object of type `Date`
    b. You will need to import the `java.util.Date` class
    c. Create getter and setter methods for each of the attributes **except for `performers[]`.**

10. At this point, all of your code should compile. We haven't created a `main()` method yet so we can't execute any of it but ensure that you do not have any compilation errors before continuing.

Exercise 3: Working with Array Attributes

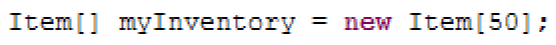
11. For the `performers[]` attribute of the **ClassicalCD** class we are going to create a method called `addPerformer(String performer)`. This method will need to know how many elements in the array are already being used. The length attribute won't help us – it just tells us how many slots there are for holding data. So:
    a. Create a private attribute named `performerCount` which is of type `int` and initialize it to zero.
    b. Create the `addPerformer()` method.
    c. Within the body of `addPerformer()` test to make sure there is still room in the `performers[]` array. Use `performers.length` and `performerCount` to manage this test.
    d. If there is no room left in the array, display a message warning the user.
    e. If there are available slots in the array, add the performer name.
    f. Ensure that you increment `performerCount` !

12. We also want a method to **display** all of the performers for a **ClassicalCD**. Write a method called `showPerformers()` which loops through the elements of the array and prints them to the console. USE THE `performerCount` attribute! Do not print out empty array slots.

Exercise 4: Creating Constructors and Testing

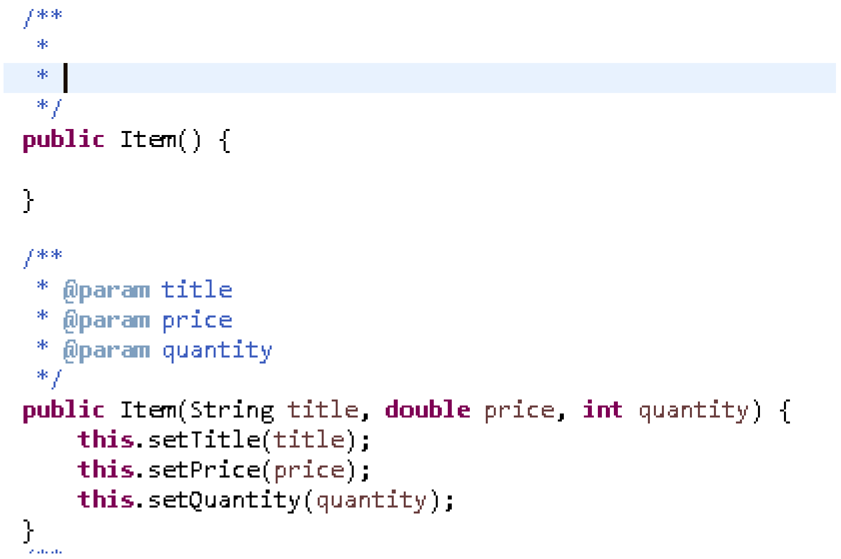
13. Create a new package named **com.javaoo.store.drivers** . Create a new class named **InventoryDriver** in the **com.javaoo.store.drivers** package. Ensure that this class has a `main()` method.

14. In the **InventoryDriver** main method create an array of 50 **Items** named `myInventory`.



```
Item[] myInventory = new Item[50];
```

15. In the **Item** class, create a constructor which accepts three parameters in its signature. These parameters should be `title, price and quantity`. In the constructor use the set methods to set up each of these attributes. Also, create a constructor that takes no arguments. This is needed in order to allow our other code to compile for now. We will address this in more detail later. When done, it should look as follows.



```
/**
 *
 *
 */
public Item() {

}

/**
 * @param title
 * @param price
 * @param quantity
 */
public Item(String title, double price, int quantity) {
    this.setTitle(title);
    this.setPrice(price);
    this.setQuantity(quantity);
}
```

16. In the **Book** class, create a constructor which accepts six parameters in its signature. These parameters should be `title, price, quantity, author, publisher and category. Pass title`, `price` and `quantity` to the super-class constructor. In the **Book** constructor use the set methods to set up the other three attributes.

```
/**
 * @param title
 * @param price
 * @param quantity
 */
public Book(String title, double price, int quantity,
        String author, String publisher, String category) {
    super(title, price, quantity);
    this.setAuthor(author);
    this.setPublisher(publisher);
    this.setCategory(category);
}
```

17. Back in **InventoryDriver** `main()` method, create five **Book** objects using the constructors you just created. Give the books any titles that you want. Here is an example:

```
//Add some books to Inventory
myInventory[0] = new Book("Godzilla on Holiday", 24.95,5,
        "Wesley Wynham-Price", "Ransom House","FICTION");
myInventory[1] = new Book("Loch Ness Memories", 49.95,1,
        "Fred MacMurray", "Penquin Press","FICTION");
myInventory[2] = new Book("MVS JCL",89.95,3,
        "Steve Balmer", "Microsoft Press","NON-FICTION");
myInventory[3] = new Book("Lingo in a Nutshell", 19.95,8,
        "Bill Bates", "O'Reilly","NON-FICTION");
myInventory[4] = new Book("Grid Computing", 79.95,2,
        "Bobby Beowold", "Trouser Press","NON-FICTION");
```

18. Using the same technique as above, create a constructor for the **Artist** class that only assigns the name attribute for now. Remember, **Artist** does NOT inherit form **Item**.

19. Create a constructor in the **CD** class similar to what you did in the **Book** class. This constructor should accept 5 parameters which are `title`, `price`, `quantity`, `artist` and `releaseDate`.

20. Go back to **InventoryDriver** and create a **CD** object. When creating a **CD**, you will need to pass the constructor an instance of **Artist** and a **Date**. Before creating the **CD**, create an instance of **Artist** for that particular CD**.** Also, we are using a deprecated technique for creating dates for now. Since we have not discussed exceptions yet, it would be too confusing to use the **DateFormat** class which is a more sophisticated way of managing dates. Here is an example of what you code might look like. Add 2 more **CDs**.

```
myInventory[5] = new CD("Going For The One", 12.95, 4,
                    new Artist("YES"), new Date("07/07/1977"));
```

21. We will finish up the **ClassicalCD** class and write a report to make sure everything is working correctly in the next lab. However, at this point you should not have any fatal compilation errors.

Lab 10: Extending the Inventory System

Objective

This lab will provide some additional functionality to assist us in testing the inventory system as well as the final constructor that is needed.

Overview

In this lab you will:

- Create the **ClassicalCD** constructor
- Create a report method
- Use the report method to verify that all of your inventory system classes are coded correctly

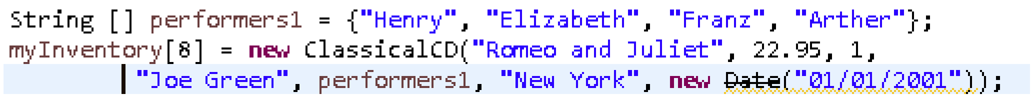
Step by Step Instructions

Exercise 1: ClassicalCD Constructor

1. Write a constructor for the **ClassicalCD** class that takes 7 arguments (`title`, `price`, `quantity`, `composer`, an array of `performers`, `location` and `release date`).

2. Ensure that you use your super class constructor to manage the `title`, `price` and `quantity`. Use your `addPerformer()` method to help with setting the array of performers so that you do not overflow the array.

3. In the `main()` method of **InventoryDriver**, create two **ClassicalCDs**. The **ClassicalCD** constructor will require an array for performers. Create an array for each new CD and pass it to the constructor. Here is an example:



```
String [] performers1 = {"Henry", "Elizabeth", "Franz", "Arther"};
myInventory[8] = new ClassicalCD("Romeo and Juliet", 22.95, 1,
        "Joe Green", performers1, "New York", new ~~Date~~("01/01/2001"));
```

Exercise 2: Creating the Inventory Class

4. Create a new class in the **com.javaoo.store** package named **Inventory**. This class will not have any data members at this point. It will just provide some general functionality for our inventory system.

5. Create a `public static` method in the **Inventory** class named `produceReport().` This method will accept an array of **Item** objects and will return a `void`.

6. The `produceReport()` method should print out the **Title**, **Price** and **Quantity** for each item in the array. In addition, at the end of the report there should be line which displays the total count of items and the total value of all items. The beginning of the report should have an appropriate header.

7. From the `main()` method in **InventoryDriver**, call the `produceReport()` method passing your array of **Items** that you created in the previous lab. Remember that this is a static method and must be called a bit differently than the other methods you have been working with. Verify that your items are printed and the totals are calculated correctly.

Challenge Exercise

8. Experiment with different techniques for making the output line up well and look professional. This is not as easy as it might seem and will take some additional processing. A `switch` statement based on the title length might help.

9. Also, you might want to experiment with the `java.text.DecimalFormat` class for formatting your total price to two decimal places.

Lab 11: Working with Interfaces

Objective

This lab will take us back to the **Box** and **Cube** example but add some additional classes, interfaces and behavior. We will also need to re-factor the existing functionality somewhat in order to be more consistent with the new design.

Overview

In this lab you will:

- Create several new classes in an inheritance hierarchy
- Create interfaces to model common behavior between unrelated classes
- Test your new functionality

Step by Step Instructions

Exercise 1: Expand the existing system

1. For this lab, you will be working in the **com.lq.exercises** package of the **ClassExercises** project.

2. We currently have a **Box** class in place as well as a **Cube** class that extends **Box**. However, we also want to model other objects. This forces us to re-think the current architecture. To start the process, create an `abstract` class named **Shape**. We will never just create **Shape** objects. It will only be used to contain common characteristics in our new class structure. Your new **Shape** class should:
   a. Be declared as `public` and `abstract.`
   b. Contain two `private` attributes, `color` and `name`, which are both of type `String`.
   c. Provide `public` getter and setter methods for the new attributes.
   d. Do not worry about writing any constructors for **Shape**. We will just use the set methods to change the `name` and `color`.

3. Modify the **Box** class so that it extends **Shape**.

4. Create a **Rectangle** class that also extends **Shape**. It should have the following characteristics:
   a. Contains two `private` attributes, `length` and `width`, which are both of type `double`.
   b. Has public getter and setter methods for the new attributes.

   c. Has a `public` constructor that accepts two arguments to set the `length` and `width`. This constructor should also set default values for name and color, “Unknown” and “White”.

   d. Has a public constructor to accept all four attributes of a **Rectangle(**two for **Rectangle** and two for **Shape)**

5. Create a **Square** class that extends **Rectangle**. It should have the following characteristics:
   a. No new attributes.
   b. Contain a `public` constructor that accepts one argument and calls the base class constructor.

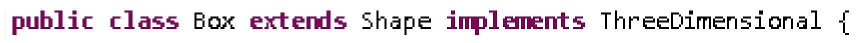
<u>**Exercise 2: Working with Interfaces**</u>

6. As we think about our system, we know it will grow considerably as we begin to add additional shapes. We want to guarantee that all of our 3-dimensional shapes provide methods that calculate the volume and the surface area. We want to guarantee that all of our 2-dimensional shapes provide methods to calculate the area and perimeter. This is less practical to do with inheritance since each shape will calculate these values differently. We will use interfaces instead. This will leave the details to the implementing class but ensure that classes provide common behavior.

7. Create a new interface in the **com.lq.exercises** package named **ThreeDimensional**. Declare the following public methods in this interface:
   a. `public double getVolume();`
   b. `public double getSurfaceArea();`

8. Remember, an interface cannot actually implement a method. It can only declare a common signature for a method that must be written by any class that chooses to implement the interface. Don’t forget to end each method declaration with a semi-colon.

9. Go back to the **Box** class and change the class declaration as follows:



```
public class Box extends Shape implements ThreeDimensional {
```

10. Notice that we do not even need to change any of the code in **Box**. We have already written the methods that implement this interface.

11. Now we will do the same for our 2 dimensional classes. Create another interface and name it **TwoDimensional**. Declare the following public methods in this interface:
   c. `public double getArea();`
   d. `public double getPerimeter();`

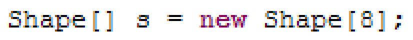
12. Go back to the **Rectangle** class and change the class declaration to implement the new **TwoDimensional** interface.

13. This time you should see that there are compilation errors in your **Rectangle** class. This is because **Rectangle** does not actually implement the interface yet. Add the definitions for the methods in the interface to **Rectangle**. They should be fairly easy to write.

Exercise 3: Test your design

14. Create a new class named **ExerciseShapes**. Ensure that it has a `main`() method.

15. Create an array of 8 **Shape** objects.



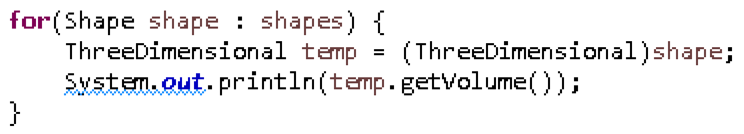
```
Shape[] s = new Shape[8];
```

Add 2 Rectangles, 2 Squares, 2 Cubes and 2 Boxes to the array.

16. Use a for-each to loop through the array and call `setColor()` for each object. Why does this work? Since all objects in the array inherit from **Shape**, they all have a `setColor()` method defined.

17. Now loop through the array and call `getVolume()` for each object. Does it work? No! The **ThreeDimensional** getVolume() method is not defined in the **Shape** class. It is part of the **ThreeDimensional** interface.

18. Change the loop so that it looks as follows:

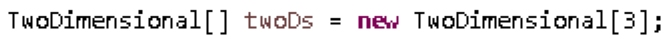


```
for(Shape shape : shapes) {
    ThreeDimensional temp = (ThreeDimensional)shape;
    System.out.println(temp.getVolume());
}
```

This will compile because we are temporarily casting the object in the array to the interface type. However, when we run the code, this will cause a run-time exception because the first 4 objects in the array are 2 dimensional. They do not implement the **ThreeDimensional** interface and cannot be cast that way. We can fix this by starting the loop with `j=4` but this is dangerous because it forces us to know exactly which element type we are working with. Let's fix that. **[Comment out the loop above since it will continue to fail when we run it.]**

19. Create a new class named **Circle**. **Circle** should extend **Shape** and implement **TwoDimensional**. Provide the appropriate attribute (`radius`) and constructor. Also, implement the `getArea()` method [Hint: PI * `radius`^{2}] and getP`erimeter`() method [Hint: 2 * PI * `radius`].

20. Go back to the **ExerciseShapes** class and create a new array that will hold three objects that are all 2 dimensional. It should look as follows:



```
TwoDimensional[] twoDs = new TwoDimensional[3];
```

21. Add a **Circle**, **Rectangle** and **Square** to the new array. Now, loop through the array calling the `getArea()` and getP`erimeter()` methods (print out the info they return). Does it work? Yes it does, because all of the objects are related by the interface that declares that `getArea()` and getP`erimeter()` must exist.

Lab 12: Polymorphism in Action

Objective

This lab will utilize our **Shape** example again. We will demonstrate polymorphic behavior by implementing a similar method several different ways and showing how polymorphism allows us to work with many different objects in a common and flexible manner.

Overview

In this lab you will:

- Add new behavior to the **Shape** hierarchy
- Create an array of many shape objects
- Test your new functionality

Step by Step Instructions

Exercise 1: Creating polymorphic methods

1. For this lab, you will be working in the **com.lq.exercises** package of the **ClassExercises** project.

2. When you created the **Shape** class in the previous lab, it did not explicitly inherit from any other class. In this case, it implicitly inherits from **Object**. We will experiment with one of the methods that all of our shapes inherit from **Object** which is called `toString()`. This is a method that provides a default way of printing out information about an object as a **String**. The default behavior of the method provides some basic information.

3. To test this behavior, create a new class in the **com.lq.exercises** package named **ExercisePrint**. Ensure that this method has a `main()` method.

4. Create an array of **Shape** objects and place each of your shape types in the array. [Remember, the **Shape** class is abstract and cannot be created as a stand alone object.]
   a. Add constructors to Rectangle, Square, Cube, Box and Circle which accept name and color as well as the specific attributes for that Shape.

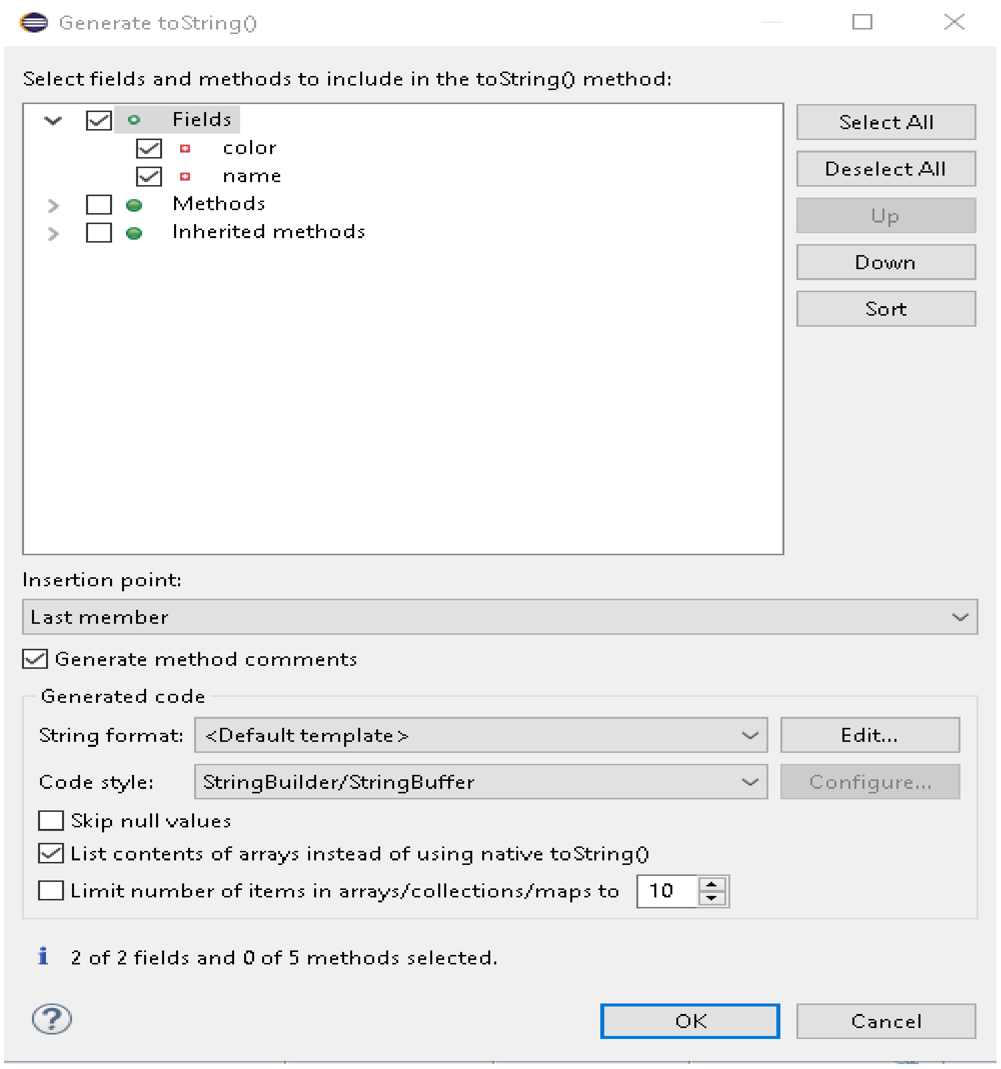
5. Loop through your array of objects and call the `toString()` method for each object in the array.

```
for(Shape shape: shapes) {
    out.println( shape );
}
```

6. Run the program and observe the output. Notice that the inherited (from Object) `toString()` method just provides some system information about the type and location of the object in memory. Not very useful! An example below.

```
com.lq.exercises.Box@27562756
com.lq.exercises.Cube@29322932
com.lq.exercises.Circle@2f6a2f6a
com.lq.exercises.Rectangle@31393139
com.lq.exercises.Square@32b732b7
```

7. In your **Shape** class, provide a toString() method using the Source -> Generate toString wizard. Make sure the Fields box is checked, the insertion point is set to “Last member”, the Generate method comments box is checked, The String format is “<Default template>”, and the Code Style is set to “StringBuilder/StringBuffer”. It should look like this:


Generate toString()

Select fields and methods to include in the toString() method:

- [✓] Fields
  - [✓] color
  - [✓] name
- [ ] Methods
- [ ] Inherited methods

Insertion point: Last member

[✓] Generate method comments

Generated code

String format: <Default template>

Code style: StringBuilder/StringBuffer

[ ] Skip null values

[✓] List contents of arrays instead of using native toString()

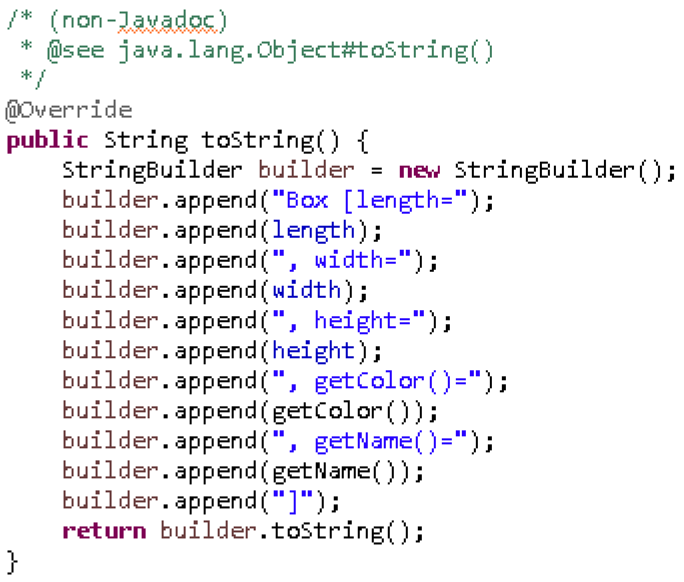
[ ] Limit number of items in arrays/collections/maps to 10

2 of 2 fields and 0 of 5 methods selected.

8. If you have created toString() methods for your Shapes in earlier labs, you may have to recreate them now that we are extending Shape, with color and name;

9. Re-run your test program. Now the output is coming from the `toString()` method defined in **Shape**. We are already seeing the benefits of polymorphism. In the previous run of the method, the only behavior we had available was the default system behavior. Now we have customized that functionality and will utilize this new method for any object of type **Shape**.

10. Let's extend this concept even further. In the **Box** class, add an implementation for `toString()` using the Source->Generate toString() wizard. Add in the Inherited methods for getColor() and getName(); Your code should look like this:



```
/* (non-Javadoc)
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    builder.append("Box [length=");
    builder.append(length);
    builder.append(", width=");
    builder.append(width);
    builder.append(", height=");
    builder.append(height);
    builder.append(", getColor()=");
    builder.append(getColor());
    builder.append(", getName()=");
    builder.append(getName());
    builder.append("]");
    return builder.toString();
}
```

11. Re-run your test program. Notice how the output changed. When we encounter a **Box** or **Cube**, `toString()` from the **Box** class is used. Otherwise, `toString()` from **Shape** is used. This is polymorphic behavior. Even though we are working with an array of **Shape** objects, the behavior from the class that defines the actual object type in the array is used when methods are called.

12. Extend this concept by writing `toString()` methods for **Circle** and **Rectangle**. [Hint: When you created **Circle**, you may not have created a get and set method for `radius`. Do that now in order to implement `toString()` appropriately.]

Your output should look similar to the followingwhen done:

```
Rectangle [length=5.0, width=4.0, color=Blue, name=Rectangle1]
Rectangle [length=8.0, width=8.0, color=Red, name=Square1]
Box [length=5.0, width=5.0, height=5.0, color=Yellow, name=Cube1]
Box [length=12.0, width=15.0, height=6.0, color=Orange, name=Box1]
Circle [radius=3.0, color=Lavender, name=Circle1]
```

This is polymorphism at work! We have a container that holds many different object types but as we call methods on those objects, the correct implementation of that method is called for the appropriate object.

Lab 13: Generics

Objective

In this lab, we will look at three different ways that generics are commonly used in Java.

Overview

In this lab you will:

- Create a generic class, a generic method, and a use of generics to provide an upper bound for the classes that can be used with a method.

Step by Step Instructions

Exercise 1: Creating a generic class

There may be many times when we want to use a common class that may be used with several different types at runtime. These classes look like a regular Java class except the class name is followed with a parameter section. This is why these types of classes in Java are sometimes called parameterized classes or parameterized types. The parameter section may contain a list of one of more generics that may then be referenced in the body of the class.

1. Create a new project named **Generics** in your workspace as described previously. In the src folder create a class named **GenericBox** with a package name of **com.lq.generics**.

2. We want an instance of the **GenericBox** class to be used with different types. Indicate that the **GenericBox** class is parameterized by adding a generic parameter after the class name. You may use any name that you like for the generic parameter, but it is by convention formatted as a single cap – usually **“T”** or **“E”**.

   Generic declarations take the form *public class ClassName<T>* where *T* represents the parameter that is provided when the class is used i.e.

   ClassName<T> cn = new ClassName<>();

3. The T in your declaration is provided in code and to be used must be “bound” to an attribute of the class. In your class, add an attribute of type **T** (or whatever name you used for the generic) named **t**. Code or generate the getters and setters for t. Your code should be like the example below.

```java
public class GenericBox<T> {
    private T t;

    /**
     * @return the t
     */
    public T getT() {
        return t;
    }

    /**
     * @param t the t to set
     */
    public void setT(T t) {
        this.t = t;
    }
}
```

4. Create a driver named **GenericBoxExerciser** to see how the **GenericBox<T>** works in practice.

5. In the main method add code to create two instances of **GenericBox<T>**. The 1st instance should be of type **Integer** and the 2nd of type **String**. Name the instances as desired.

6. On the **GenericBox<String>** instance invoke **setT()** to pass the value “Hello World” to the instance. Call **setT()** on the Integer instance and pass 10.

7. Use **System.out.printf** to print the values of the parameter type to the console from both the **Integer** and **String GenericBox** instances. If you are not familiar with the printf method, look it up in the Standard Edition Javadoc. (If you would rather use println, that will work fine as well).

8. Run GenericBox as a Java Application to exercise. You should see output to the console that reflects the values that were passed using **setT().** Your completed class should be like the following:

```java
    public static void main( String... args) {
        GenericBox<Integer> integerBox = new GenericBox<>();
        GenericBox<String> stringBox = new GenericBox<>();
        integerBox.setT(10);
        stringBox.setT("Hello World");
        out.println("IntegerBox value: " + integerBox.getT());
        out.println("StringBox value: " + stringBox.getT());
    }
```

9. Generic methods allow us to write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

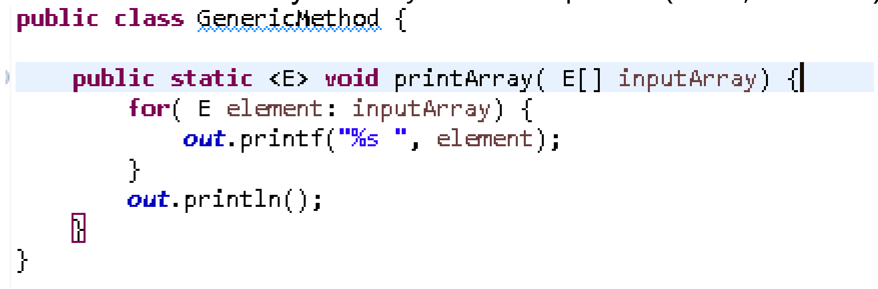
.

10. Create a class named **GenericMethod** in the **com.lq.generics** package Add this method to the class:

    ```
    public static <E> void printArray(E[] inputArray)
    ```

    Here you have declared a method named **printArray** that is bound to the parameterized type **E**. The instance of the parameterized type is named **inputArray**.

11. In the body of the method iterate over the inputArray and print each of the elements of the array. Use `System.out.printf("%s",element)` for the output..

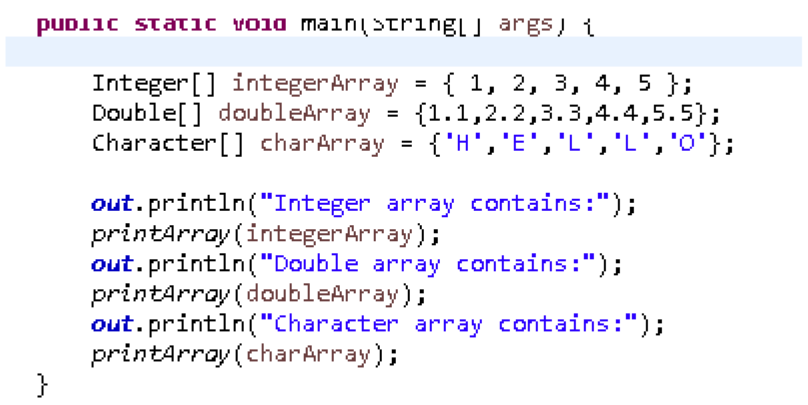


```
public class GenericMethod {

    public static <E> void printArray( E[] inputArray) {
        for( E element: inputArray) {
            out.printf("%s ", element);
        }
        out.println();
    }
}
```

12. Create a new class named **GenericMethodExerciser** In the body of the main method create three arrays. The 1st array is of type Integer, the 2nd is of type Double and the 3rd is of type Character. In the declaration of the arrays use {v1,v2,v3…} to populate the arrays.

13. Call the printArray<E>() method passing each of the arrays.

14. Exercise the application to verify that the values of the associated array print as expected. Your class should look similar to the following:



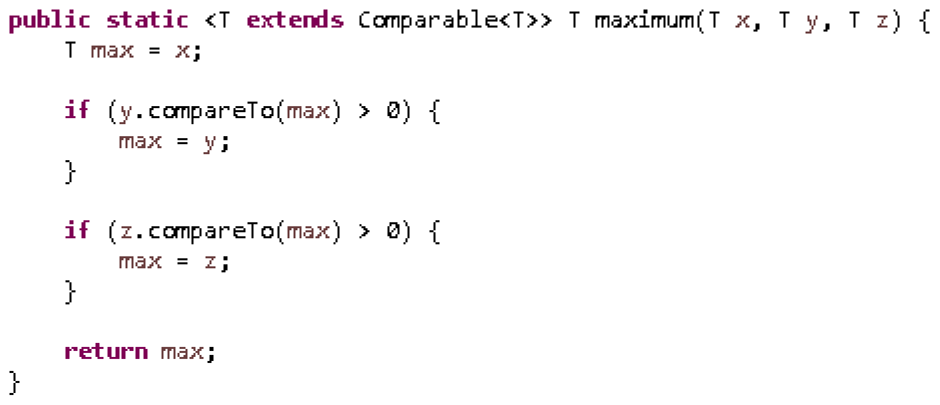
```
public static void main(String[] args) {

    Integer[] integerArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = {1.1,2.2,3.3,4.4,5.5};
    Character[] charArray = {'H','E','L','L','O'};

    out.println("Integer array contains:");
    printArray(integerArray);
    out.println("Double array contains:");
    printArray(doubleArray);
    out.println("Character array contains:");
    printArray(charArray);
}
```

15. There are times in code when you want to restrict what types may be passed as a type parameter. This is what "bounded type parameters" provide in a Java application. To declare a bounded type parameter, list the type

parameter's name followed by the **extends** keyword, followed by its upper bound.

Add a class named **Maximum** to the **com.lq.generics** package.

16. Add a method named **maximum** as follows:



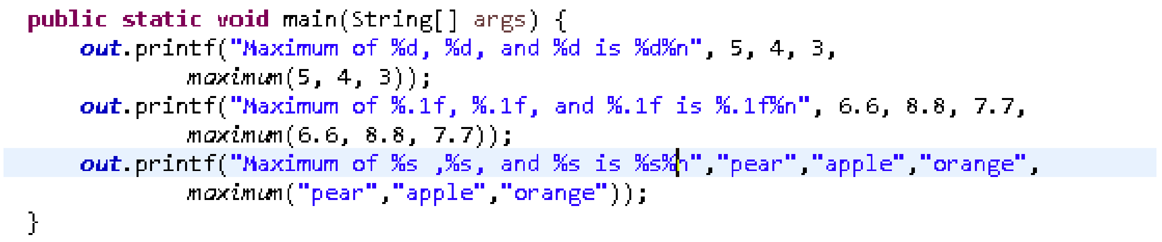
```java
public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
    T max = x;

    if (y.compareTo(max) > 0) {
        max = y;
    }

    if (z.compareTo(max) > 0) {
        max = z;
    }

    return max;
}
```

In this method we have declared that this class is parameterized by a type that must be a type or subtype of Comparable. Since Comparable is an interface, this means that any class that implements Comparable or is a subtype of a class that implemented Comparable may be the type bound to this method.

All the method does is use the **compareTo** method (as defined by the Comparable Interface) to compare the three method arguments to see which is the largest and to then return that value.

.

17. Add a class named **MaximumExerciser** to the **com.lq.generics** package Add a main method to the class to test the parameterized generic method as follows:



```java
public static void main(String[] args) {
    out.printf("Maximum of %d, %d, and %d is %d%n", 5, 4, 3,
            maximum(5, 4, 3));
    out.printf("Maximum of %.1f, %.1f, and %.1f is %.1f%n", 6.6, 8.8, 7.7,
            maximum(6.6, 8.8, 7.7));
    out.printf("Maximum of %s ,%s, and %s is %s%n","pear","apple","orange",
            maximum("pear","apple","orange"));
}
```

The exercise should display the arguments in the correct order.

Lab 14: Collections

Objective

In this lab, we will take advantage of the **Collection** interface in order to add more flexibility to some of our **Inventory** system classes.

Overview

In this lab you will:

- Change some arrays to collection types
- Add objects to collections
- Retrieve objects from collections

Step by Step Instructions

Exercise 1: Using Collections

The Inventory system contains several arrays. We will convert some of these to collections. In the interest of time, we will leave the others as arrays. We will make sure that the different collections specify their generic type to help ensure runtime type safety.

1. This lab will enhance the inventory application in the **Store** project. The **Artist** class should contain two arrays at this point -- `memberNames` and `memberInstruments`. Convert `memberNames` to a **SortedSet**. Don't create any get or set methods for it yet. We'll handle that a little later. Make sure to specify that this **SortedSet** is of generic type String.

2. Convert `memberInstruments` to a **Map**. We want a **Map** so that we can key the instruments to the names of the members. Each entry in the **Map** will contain a key which is the member name, and a value which is a **SortedSet** holding a list of the instruments the member plays.

   Specifying the generic type for the **Map** is a little tricky. We have to tell the compiler what types the **Map** will use for its key and value. The key is easy – it is a **String**. The value, however, is a **SortedSet**, another generic type. So as part of the generic declaration of the **Map**, we need to provide a generic declaration of one of its generic parts. We are declaring that the **Map** will contain a key value pair in which the key is a **String** and the value is a **SortedSet** that contains **String** objects

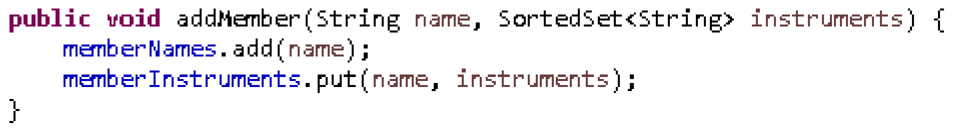
   ```
   private Map<String, SortedSet<String>> memberInstruments;
   ```

3. Add a default constructor which initializes memberNames to a **TreeSet** containing only String references; It should in similar fashion initialize memberInstruments to a **TreeMap**.

   Add a constructor using the Source->Generate Constructor Using fields wizard to include all fields.

4. Create a new method called `addMember()` which accepts a **String** for the member name and an **ArrayList** of instruments. Be sure to specify the generic type of instruments.

   Add `name` to the `memberNames` **ArrayList**. Add an entry to the `memberInstruments` **Hashtable** as well. Your code should look as follows:



```
public void addMember(String name, SortedSet<String> instruments) {
    memberNames.add(name);
    memberInstruments.put(name, instruments);
}
```

5. Right about now, it has probably occurred to you that `memberNames` is redundant and therefore unnecessary. You're right, and this will happen at times in programming. As you get deeper into the details, you will re-factor your design and your code in order to add missing details or increase efficiency. For our system, we will just leave it as is since we want to experiment with two different types of containers.

6. Now let's exercise the **Artist** class to see if our **Collections** are working properly. We will add two members with a list of instruments for each and then extract one of them to print out the name and instruments. Create a class named **ArtistExerciser** that has a `main` method. Create an instance of **Artist** named `hotPlate`.

7. In the `main()` method, create a **TreeSet** called `instruments1` with a generic type of **String** and add these instruments to it: "Piano", "Clarinet", "Hurdy Gurdy" and "Tuba". [Hint: don't forget to import `java.util.*;`].

8. Call `addMember()`, passing it the name "**Tom**" and the **TreeSet** `instrument1`.

9. Repeat steps 7 and 8, creating another **TreeSet** of generic type **String** and adding the instruments of your choice. Add these via `addMember()`, using the name "**Steve**".

10. Now, how will we extract Tom's instruments? First, we need to get a list of band member names. Write a method in the **Artist** class named

`getMembers()` that returns the **SortedSet** that contains all of the member names. Remember that the **SortedSet** contains **Strings**, so **String** should be used as the generic in the declaration of the method. This should look as follows:

```java
/**
 * @return the memberNames
 */
public SortedSet<String> getMembers() {
    return memberNames;
}
```

11. Write a method in the **Artist** class named `getInstruments()` that accepts a member name as a **String** and returns the list of instruments as an **SortedSet**. Examine the API for the **Map** class to discover the correct method to access the **Map** and return a value based on a key.

12. Go back to the **ArtistExerciser** class and print out information for the `hotPlate` **Artist**.. create a private static void method call printMemberInstruments. to the class. It should accept an Artist and a String for memberName. Print out the member name .Then using an enhanced for loop, print out the instruments that member plays. Example below:

```java
private static void printMemberInstruments(Artist artist, String memberName) {
    out.println("HotPlate band member " + memberName + " plays: ");
    for (String instrument : artist.getInstruments(memberName)) {
        out.println('\t' + instrument);
    }
}
```

Lab 15: Throwing Exceptions

Objective

In this lab, we will experiment with some of the basic coding involved with creating new exceptions and throwing and catching them.

Overview

In this lab you will:

- Create a new exception.
- Write some code that throws the new exception
- Write some code that tries to execute the new code and handles the new exception.

Step by Step Instructions

Exercise 1: Creating an Exception

An exception is typically just a marker class which means that we tell different exceptions apart simply by their name.

1. For this lab, we will work in the **ClassExercises** project and the **com.lq.exercises** package. Create a new exception class named **TooHotException**. Ensure that this new class extends **Exception**. Your class should have two constructors (Default and accepts String message. Delete any others as we will not use them. Make sure you have the appropriate super calls in each.

2. Create another new class named **Coffee**. Add a method named `setTemperature()` that accepts one parameter of type `int` and returns a `void`. Also, add an attribute to the **Coffee** class of type `int` named `temperature`. Add a `getTemperature()` method that returns the contents of `temperature`.

3. In the body of the `setTemperature()` method, check to see if the `temperature` that was passed in is greater than 120 degrees. If it is, `throw` a new **TooHotException** back to the caller of the method with a message that the coffee is too hot.. Otherwise, set the `temperature`. Remember to add the `throws` declaration to your `setTemperature()` method.

4. Add a constructor to accept a temperature.

5. Create a new class named **CoffeeExerciser**. Ensure that it has a `main()` method. In the body of the main method, create a **Coffee** object with a value of 110. You will have to do this inside of a `try` block and will also need a `catch` block to deal with the possibility of the exception getting thrown. At the very end of the `main()` method, call `getTemperature()` and print out the value that it contains to ensure things are working well. Your `main()` method should look similar to the following. Run your program and observe the output.

```java
public static void main(String[] args) {
    Coffee coffee = null;
    try {
        coffee = new Coffee(110);
    } catch (TooHotException e) {
        out.println(e.getMessage());
    }
    out.println("Coffee is set to " + coffee.getTemperature());

}
```

6. Let's make a change so that we can see how `finally` blocks get processed. Move the print statement which is currently at the end of the `main()` method into a `finally` block. Re-run your program. Does the print statement still execute?

```java
public static void main(String[] args) {
    Coffee coffee = null;
    try {
        coffee = new Coffee(110);
    } catch (TooHotException e) {
        out.println(e.getMessage());
    } finally {
    out.println("Coffee is set to " + coffee.getTemperature());
    }
}
```

7. Now, change the temperature from 110 to 125. Re-run your program. What prints this time? It should look as follows:

```
Coffee is too hot
Exception in thread "main" java.lang.NullPointerException
	at com.lq.exercises.CoffeeExerciser.main(CoffeeExerciser.java:25)
```

Because an exception was thrown, we never actually created the Coffee object and set `temperature` to a new value. But we did execute the `catch` block and the `finally` block. And since the `finally` block used the coffee reference, which was never set up an instance due to the **TooHotException**, a **NullPointerException** was thrown.

This kind of logic error is very common. In fact, it has been latent in the code since Step 5 introduced the original version of `main()`. Tracking down the root causes of exceptions is important, as is proper exception handling to prevent exceptions from catastrophically cascading.

8. Challenge Exercise: Can you make it so that the **CoffeeExerciser** does not fail with an exception, but rather reports that the temperature is 0, *i.e.*,

```
Coffee is too hot
Coffee is set to 0
```

Hint: there are at least four different ways to make that happen.

Lab 16: Input, Output and Exceptions

Objective

In this lab, we will take advantage of the **java.io** package in order to add more flexibility to our **Inventory** system class.

Overview

In this lab you will:

- Populate a file with information about books.
- Read from that file and use the information to create Book objects.
- Exercise your code to ensure it is working correctly.

Step by Step Instructions

Exercise 1: Reading Information From a File

For our Inventory system, we'd like to load our **Inventory** from a text file. Doing so for all of the different inventory types would be a lengthy exercise and we will leave that open as a challenge to the student with time on his/her hands. Instead we will simply load some **Book** objects.

1. Create a new class named **BookExerciser** in project Store and package com.javaoo.store. Ensure that this new class has a `main()` method. Create a variable in the `main()` method of type **List<Book>** named `books`. Remember to import `java.util.*.`

2. Add a method named `readBooksFromFile()` in the **BookExerciser** class. This method will be used to populate our **List<Book>** with book information that is read from a file. It should have the following signature:

   ```
   public List<Book> readBooksFromFile(String name)
   ```

3. We are going to store information about our book inventory in a text file. We will pass the name of the text file to the `readBooksFromFile()` method. The first thing we need to do is open the file and ensure that we can read from it. We would like to read from the file line by line so we will do that with a **LineNumberReader** object. Do the following:

   a. Import `java.io.*;`
   b. Create a **FileInputStream** from the file name passed into `readBooksFromFile()`
   c. Create an **InputStreamReader** from the **FileInputStream**

d. Create a **LineNumberReader** from the **InputStreamReader**
e. Ensure that all of this is done inside of a `try` block
f. This portion of the code should look as follows:

```
try {
    FileInputStream inFile = new FileInputStream(name);
    InputStreamReader inReader = new InputStreamReader(inFile);
    LineNumberReader lineReader = new LineNumberReader(inReader);
```

4. Add a simple while loop to go through the file line by line and output the information to the console. Add a `catch` block to handle any exceptions of type **IOException** that might be generated by the code that is opening and reading the file. The entire method at this point should look as follows:

```
public static ArrayList readBooksFromFile(String name) {

    ArrayList books = new ArrayList();
    try {
        FileInputStream inFile = new FileInputStream(name);
        InputStreamReader inReader = new InputStreamReader(inFile);
        LineNumberReader lineReader = new LineNumberReader(inReader);

        String line = lineReader.readLine();
        while(line != null) {
            System.out.println(line);
            line = lineReader.readLine();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return books;
}
```

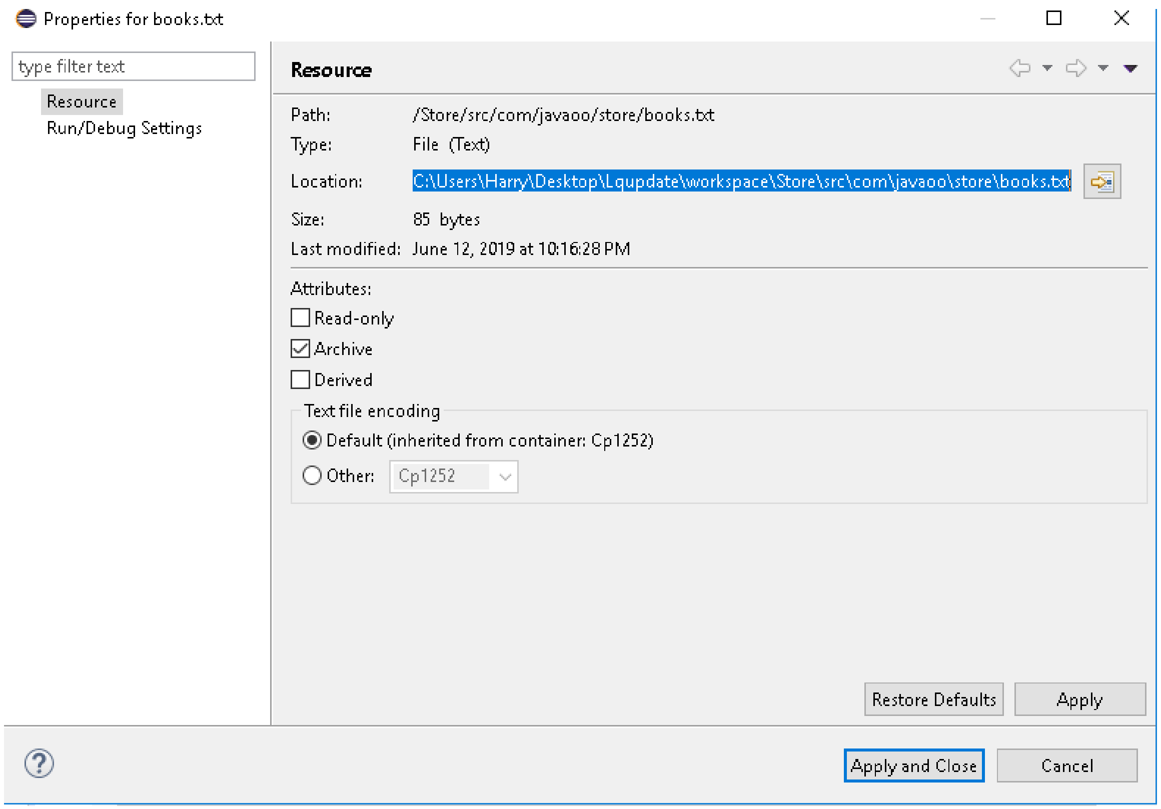
5. Now we'll need to set up the text file to hold **Book** information Using the New→File wizard, create a text file named `books.txt.` Add some books to the text file. To keep it simple we'll store title, author and price. Put one piece of data per line and add two or three books.

   Here's an example of what the file should look like:

```
COBOL for Java Programmers
Isaac B. Mortimer
45.95
3D Game Programming for the IPod
Hawley Smoot
111.95
```

Save the file;

6. In the main() method of BookExerciser, create a String variable called location. Right click on books.txt in the Package Explorer, select Properties and copy the file location from the wizard.


Properties for books.txt

Resource

Path: /Store/src/com/javaoo/store/books.txt
Type: File (Text)
Location: C:\Users\Harry\Desktop\Lqupdate\workspace\Store\src\com\javaoo\store\books.txt
Size: 85 bytes
Last modified: June 12, 2019 at 10:16:28 PM

Attributes:
- [ ] Read-only
- [x] Archive
- [ ] Derived

Text file encoding
- ● Default (inherited from container: Cp1252)
- ○ Other: Cp1252

Select cancel. Paste the clipboard as a value for the location variable;

7. In the `main()` method in **BookExerciser**, call the `readBooksFromFile()` method, passing it location as the file name. (Backslash means 'escape the next character' in Java so we need to escape the backslash.) Your main method should look as follows:

```
public static void main(String[] args) {
    String location = "C:\\Users\\Harry\\Desktop\\Lqupdate\\workspace\\Store\\src\\com\\javaoo\\store\\books.txt";
    List<Book> books = new ArrayList<>();
    readBooksFromFile(location);
}
```

Note: We are not really populating the **ArrayList** at this point. We will do that next.

8. Execute the program. You should see the contents of your text file in the console.

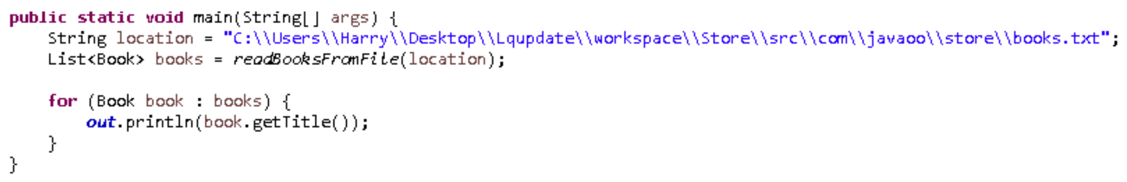
9. For each book, we're going to read three lines of text and the third line will need to be converted to a `double`. Think carefully about the logic you'll need to make this work (assuming all the data in the file is correct). Look at the **Double** class in `java.lang` for guidance. Modify the `while` loop so that it

reads the data three lines at a time storing each piece of information in a local variable. Print out each of these local variables after the 3rd line is read to ensure you are parsing the data correctly.

10.

11. Now we need to use that data and create a **Book** object. You can use the constructor that takes all 6 fields. Set the `quantity` to **5**, the `publisher` to **null** and the `category` to **NON-FICTION**. Place the book in the **ArrayList** `books`.

12. Add a loop to your `main()` method to print out the titles of all of the books in the `books` collection. Here is how your `main()` method should look.



```
public static void main(String[] args) {
    String location = "C:\\Users\\Harry\\Desktop\\Lqupdate\\workspace\\Store\\src\\com\\javaoo\\store\\books.txt";
    List<Book> books = readBooksFromFile(location);

    for (Book book : books) {
        out.println(book.getTitle());
    }
}
```

13. Execute the program and ensure that it prints out your titles.

Challenge Exercise

14. What happens if the file does not contain valid data? Experiment with different techniques for detecting bad data, and more importantly, recovering from it. This is a very open-ended problem and can be one of the most challenging aspects of writing good code.

Lab 17: Annotations

Objective

In this lab, we will look at some of uses of Annotations in Java.

Overview

In this lab you will:

- Experiment with **@SuppressWarnings** to show the impact of location on the scope of the annotation.
- Create an annotation and see how **@Document** impacts the Javadoc generated for the project.
- Show how **@Target** can constrain where an annotation may be used.

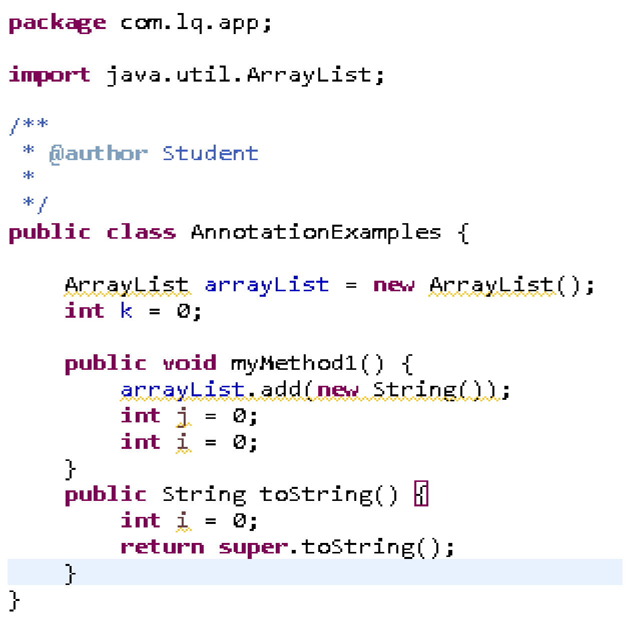
Step by Step Instructions

Exercise 1: @SuppressWarnings

The annotation **@SuppressWarnings** is used to turn off the compiler-generated warnings that crowd the Problems view and clutter the Editor sidebar. The location of the tag determines the scope of the annotation.

1. Create a new Java Project named **Annotations**. In the **src** folder create two packages: **com.lq.annotations** and **com.lq.app**.

2. Create a new class in the **com.lq.app** package named **AnnotationExamples** and implement it as follows:



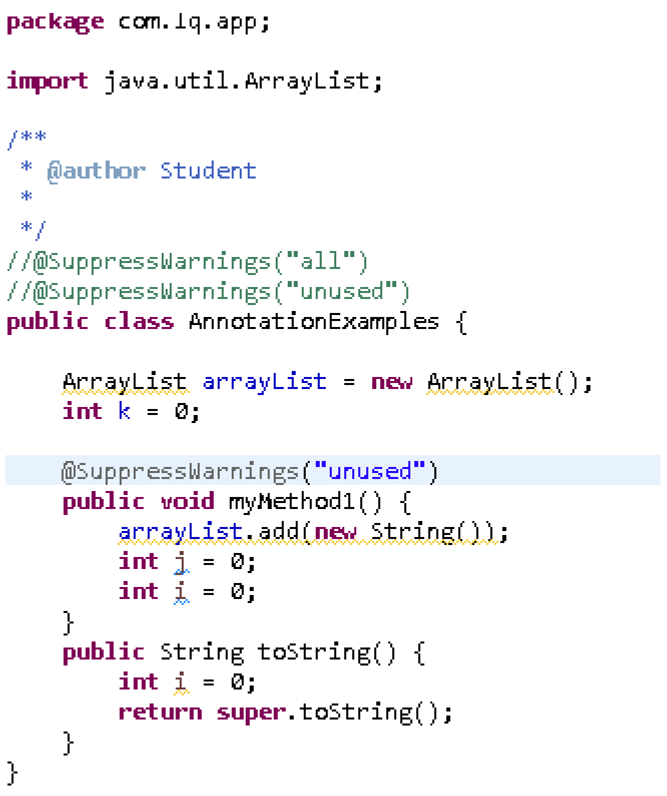
```
package com.lq.app;

import java.util.ArrayList;

/**
 * @author Student
 *
 */
public class AnnotationExamples {

    ArrayList arrayList = new ArrayList();
    int k = 0;

    public void myMethod1() {
        arrayList.add(new String());
        int j = 0;
        int i = 0;
    }
    public String toString() {
        int i = 0;
        return super.toString();
    }
}
```

You will notice several warnings on the sidebar of the Editor view.

3. Add a @SuppressWarnings (“all”) annotation after the import statement. Note that all the warnings have been *elided* (hidden from display).

4. Comment out the first annotation (to save it) and add a second immediately following @SuppressWarnings(“unused”). Note that class-wide, the warnings for variables declared but not used in code are gone.

5. Comment out the annotation and save your changes. Note that the warnings reappear. Make a copy of the second annotation and paste it immediately before **myMethod1**. Your code should now look similar to the following:



```
package com.lq.app;

import java.util.ArrayList;

/**
 * @author Student
 *
 */
//@SuppressWarnings("all")
//@SuppressWarnings("unused")
public class AnnotationExamples {

    ArrayList arrayList = new ArrayList();
    int k = 0;

    @SuppressWarnings("unused")
    public void myMethod1() {
        arrayList.add(new String());
        int j = 0;
        int i = 0;
    }
    public String toString() {
        int i = 0;
        return super.toString();
    }
}
```

Note that the warnings in **myMethod1** are now gone, but the one in the **toString** method is still there. The scope of the annotation is now at the method level.

6. Comment out the latest annotation and now add a copy to the code immediately before the **int I = 0**; declaration in **myMethod1**. Note that the warning for **i** is now gone, but the others remain. The annotation is now at field level.

7. Add a *type* level (above the class declaration) @SuppressWarnings(“unchecked”). You should now see that the warnings about generics are now gone. Different values for the annotation drive the suppression of warnings of different types.

.

Exercise 2: @Documented and Javadoc

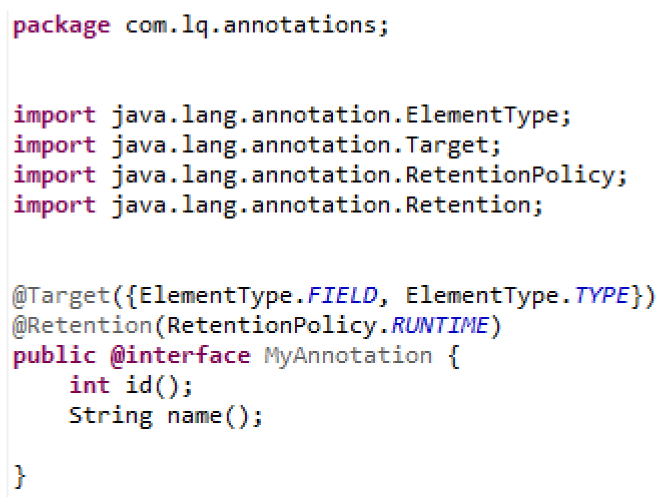
Whether or not an annotation’s values are part of the API Java documentation is determined by the presence of a **@Documented** tag in the declaration.

8. Right-click the **com.lq.annotations** package and select **New** > **Annotation**. The **New Annotation Type** wizard will appear. Name the new annotation **MyAnnotation** and click **Finish**.

9. The new annotation will open in the editor view. Declare two members of the annotation, an **int** method named **id** and a **String** method named **name**. Make sure to declare the RetentionPolicy as RUNTIME, so the Annotation is recorded in the class file by the compiler and retained by the VM at run time by using the @Retention annotation.

When we define an annotation, we sometimes want to specify where that annotation should be used in the code body of our types. The **@Target** meta-annotation provides that mechanism. Use the @Target(ElementType.TYPE) to define that the annotation can applied to another class type.

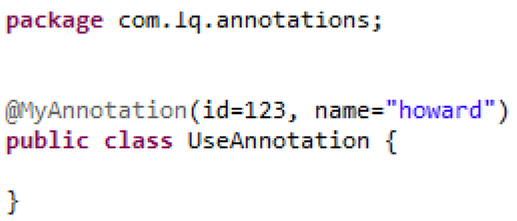
Your annotation should look like the following:



```java
package com.lq.annotations;


import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Retention;


@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    int id();
    String name();

}
```

10. Create a new class in **com.lq.annotations** package named **UseAnnotation**. Add your new annotation to the class; choose an **id** and **name**.

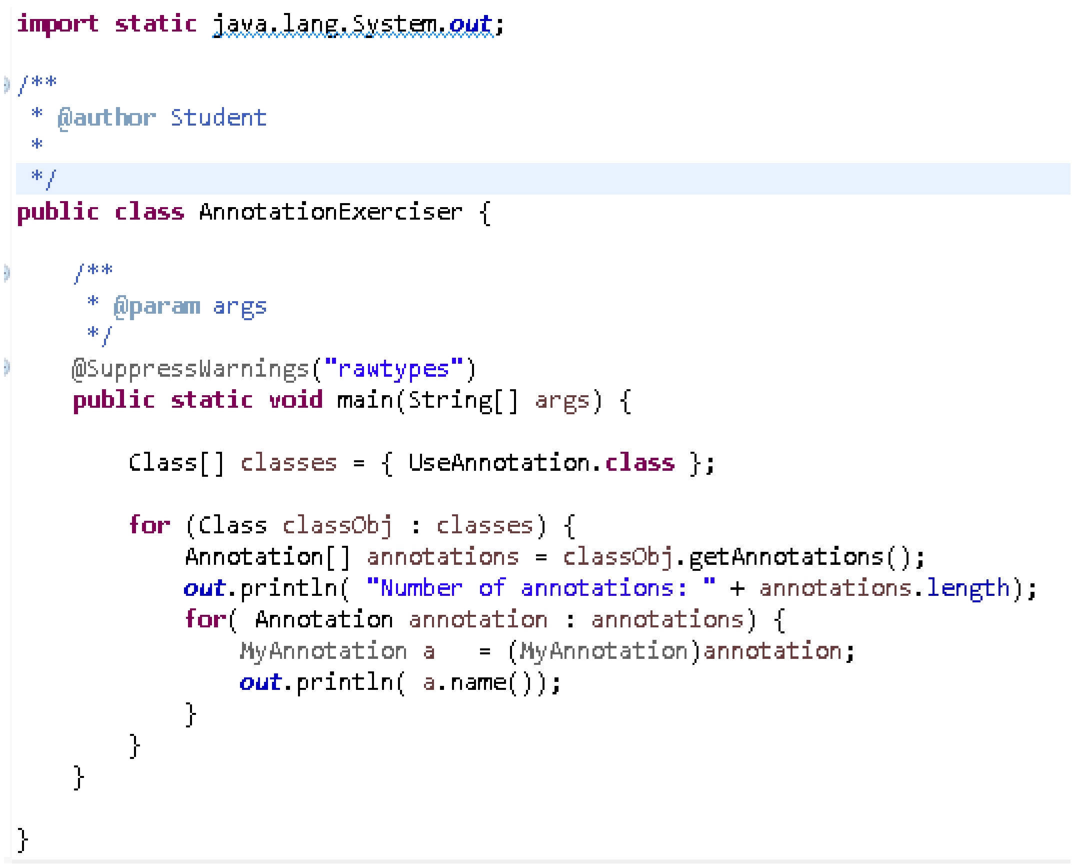
Your code should look something like the following:



```java
package com.lq.annotations;


@MyAnnotation(id=123, name="howard")
public class UseAnnotation {

}
```

11. Create a new class in the **com.lq.annotations** package (be sure to select the checkbox in the new class wizard that generates a **public static void main(String[] args)** method) named **AnnotationExerciser**. Add the following code within the main method to iterate the annotations defined with the UseAnnotation class. Display the length and the name property values of MyAnnotation type.

    Your annotation code should look like the following:



```java
import static java.lang.System.out;

/**
 * @author Student
 *
 */
public class AnnotationExerciser {

    /**
     * @param args
     */
    @SuppressWarnings("rawtypes")
    public static void main(String[] args) {

        Class[] classes = { UseAnnotation.class };

        for (Class classObj : classes) {
            Annotation[] annotations = classObj.getAnnotations();
            out.println( "Number of annotations: " + annotations.length);
            for( Annotation annotation : annotations) {
                MyAnnotation a   = (MyAnnotation)annotation;
                out.println( a.name());
            }
        }
    }

}
```

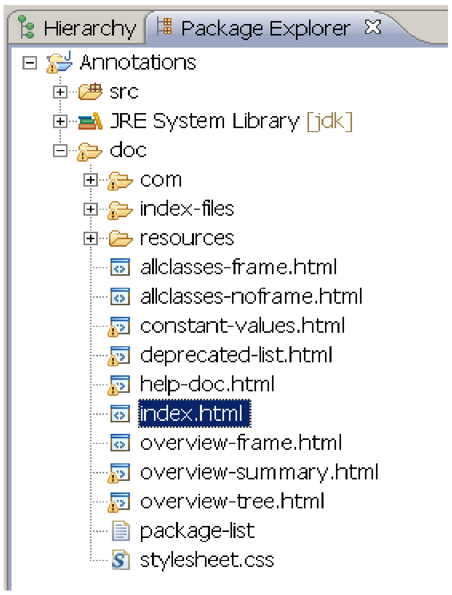
Remember to Organize Imports to clear up any import errors.

12. Select **Project** > **Generate Javadoc.** The Generate Javadoc window will appear.

13. Leave all of the settings at their default values and select the Annotations project in the **Select type for which Javadoc will be generated** list. Click **Finish**. In the Console view, you will see a considerable amount of activity as the Javadoc compiler runs.

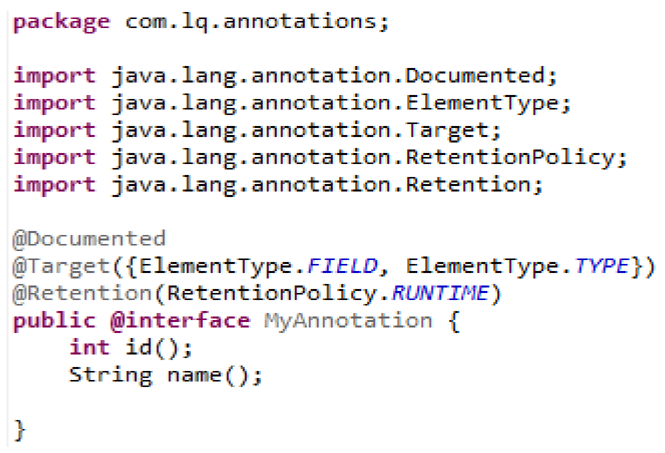
    The output of the compiler is the generated html that represents the public view of our application's API. Without the **@Documented** annotation in **MyAnnotation**, there is no display of the annotation or its values on classes where it is used, in this case the **UseAnnotation** class.

    Note that there is now a new folder in the application. The **doc** folder contains the generated html from the Javadoc compiler.



    Right-click **index.html** and select **Open With > Web Browser** to open the Javadoc in the browser. There is no reference to the annotation.

14. Add the **@Documented** meta-annotation to the **MyAnnotation** source and import the required package. **MyAnnotation** should now look like the following:



```
package com.lq.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Retention;

@Documented
@Target({ElementType.FIELD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    int id();
    String name();

}
```

15. Save your changes and re-generate the Javadoc. You may be asked to confirm the overwrite of the first Javadoc set—be sure to respond **Yes to all** when asked.

16. Re-open index.html to view the Javadoc once again and note the use of **MyAnnotation** is now part of the Javadoc.

Lab 18: Enums

Objective

In this lab, we will look at creating enums in Java.

Overview

In this lab you will:

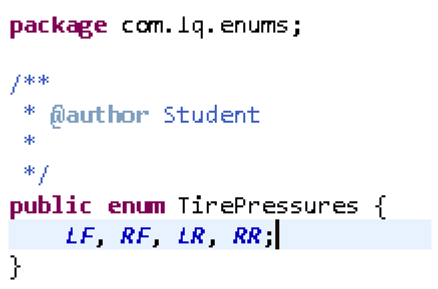
- Create an enum to represent tire pressures and look at how to override the values of the enum members.

Step by Step Instructions

Exercise 1: Create the enum

1. Create a new Java Project named **Enums**. In the **src** folder create two packages: **com.lq.enums** and **com.lq.app**.

2. Create a new **Enum** in the **com.lq.enums** package named **TirePressures** and implement it as follows:

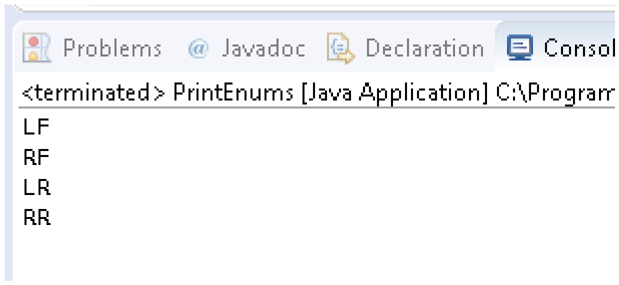


```
package com.lq.enums;

/**
 * @author Student
 *
 */
public enum TirePressures {
    LF, RF, LR, RR;
}
```

3. In the **com.lq.app** package, create a new driver class named **PrintEnums**. The code should look like this:

```
public class PrintEnums {
    public static void main(String[] args) {
            for (TirePressures tp: TirePressures.values()) {
                    System.out.println(tp);
            }
    }
}
```

Run **PrintEnums**. You should see the names of the enums appear in the console.

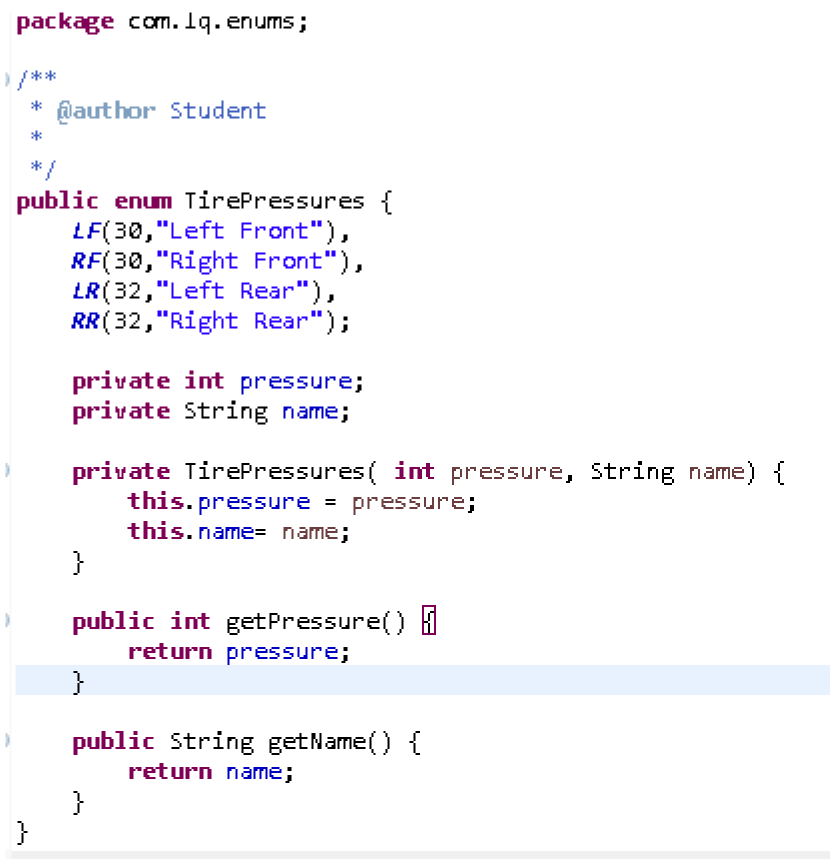


```
<terminated> PrintEnums [Java Application] C:\Progran
LF
RF
LR
RR
```

4. Tires need pressure, so add a constructor that passes the pressure for each tire. You will need to add an **int** named **pressure** to hold the pressure for each enum instance. Add a public getter for the pressure;

```
public enum TirePressures {
    LF(30), RF(30), LR(32), RR(32);
    int pressure;

    private TirePressures(int pressure) {
        this.pressure = pressure;
    }
```

5. Add a String attribute for **name** with a public String getName().
6. Add another parameter to the constructor for the name. Add values to each member to provide the parameter to the constructor call. Your code should be like this:



```
package com.lq.enums;

/**
 * @author Student
 *
 */
public enum TirePressures {
    LF(30,"Left Front"),
    RF(30,"Right Front"),
    LR(32,"Left Rear"),
    RR(32,"Right Rear");

    private int pressure;
    private String name;

    private TirePressures( int pressure, String name) {
        this.pressure = pressure;
        this.name= name;
    }

    public int getPressure() {
        return pressure;
    }

    public String getName() {
        return name;
    }
}
```

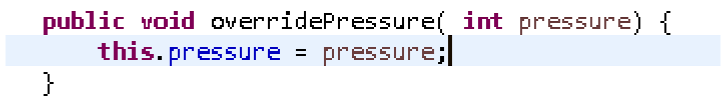
7. In PrintEnums, change the main() method to print out the value of the tire pressure and name.

```
public class PrintEnums {
    public static void main(String[] args) {
            for (TirePressures tp: TirePressures.values()) {
                    System.out.printf("%s is %d pounds%n",
                    tp.getName(), tp.getPressure());
            }
    }
}
```

8. Create a new class in the **com.lq.app** folder name **EnumTester**. Make sure that the class has a **public void main(String[] args)** method.

```
package com.lq.app;
import com.lq.enums.TirePressures;

public class EnumTester {

    public static void main(String[] args){
        for(TirePressures t : TirePressures.values()) {
            System.out.println(t + " " + t.getPressure());
        }
    }
}
```

Though relatively rare, we might also want to override some of the pressures in the enum. Add an **overridePressure(int pressure)** method to the **TimePressures** enum.



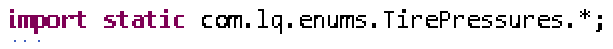
```
    public void overridePressure( int pressure) {
        this.pressure = pressure;
    }
```

9. Now add code to the **PrintEnums** class to find the “RR” member of the **TirePressures** enum using the enum **valueOf(String name)** method. Once the member has been returned, use the **overridePressure** method to change the tire pressure for that member and then loop through the enum to display the pressures for all the members. Your code should resemble the following:

```
public static void main(String[] args){

    TirePressures tp = TirePressures.valueOf("RR");
    tp.overridePressure(22);

    for(TirePressures t : TirePressures.values()) {
        System.out.println(t + " " + t.getPressure());
    }
}
```

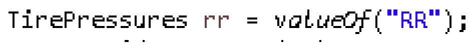
Test your code and you should see that the pressure for the right rear has now been changed from the original pressure of 32 to a new pressure of 22.

10. The members of an enum can be reference statically. You could add a static import for all the members of the **TirePressures** enum in the **PrintEnums** class.



```
import static com.lq.enums.TirePressures.*;
```

and you could then use it in your method:



```
   TirePressures rr = valueOf("RR");
```

But does that look like good, maintainable, code to you? And what if you were using more than one Enum? Static imports can be good and bad.

11. Challenge Exercise: Consider that your company only did business in three states? How would you limit addresses to those three states? Add full state name and state capitol to your answer. Code your answer with a driver to prove.

Lab 08: Setup JUnit

Objective

This lab will help you become familiar with some of the new testing features of the Eclipse. Specifically, we will look at how to:

1. Create and run a JUnit 4 Test Case.

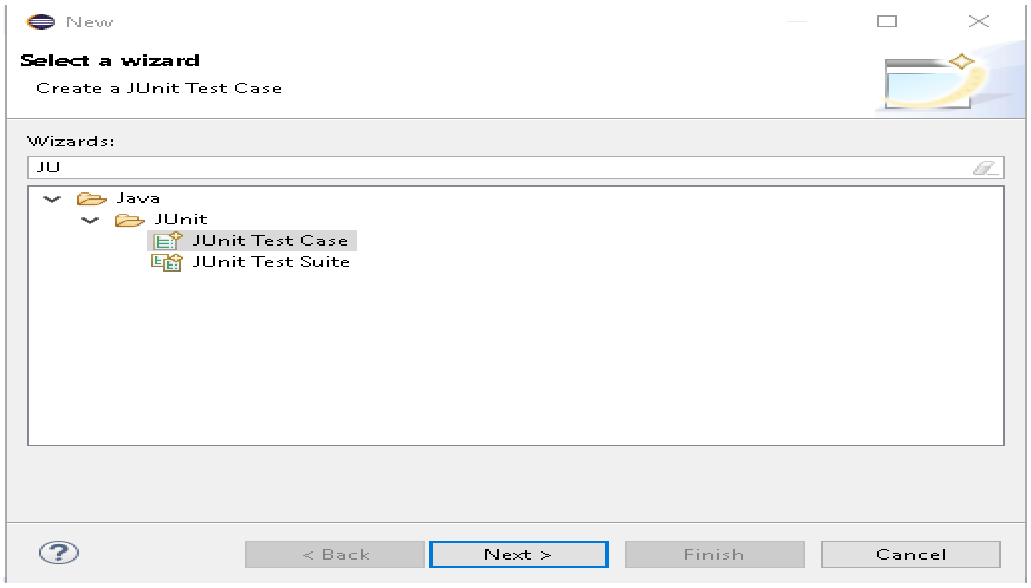
Step by Step Guide

Step by Step Instructions

Before starting this lab exercise, your instructor will demonstrate the key features of the IDE that will be used for this class. A how-to document is also available in your course book if you are unfamiliar with the IDE being used. Please review that document before beginning the lab exercise.

Add a new project

1. Create a new Project in your workspace named **TDDExerciser**.
2. Add a new package to the project named **com.lq.testcases**
3. In the package, open the New Wizard by selecting **File** > **Other**
4. When the New Wizard appears, enter "JU" in the **Wizards** field—upper or lower case is fine. This will constrain the view to show only those wizards that start with the letters "JU." The wizard should appear as follows:


New

Select a wizard

Create a JUnit Test Case

Wizards:

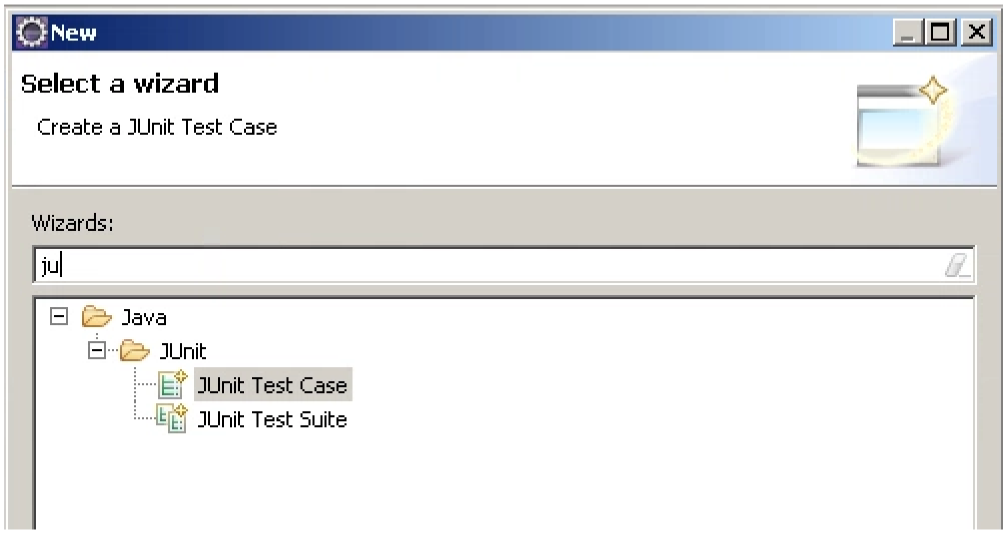
JU

5. Select Java project using the arrow keys or mouse, and click **Next**. You should see the **Java Project Wizard** appear as seen below:

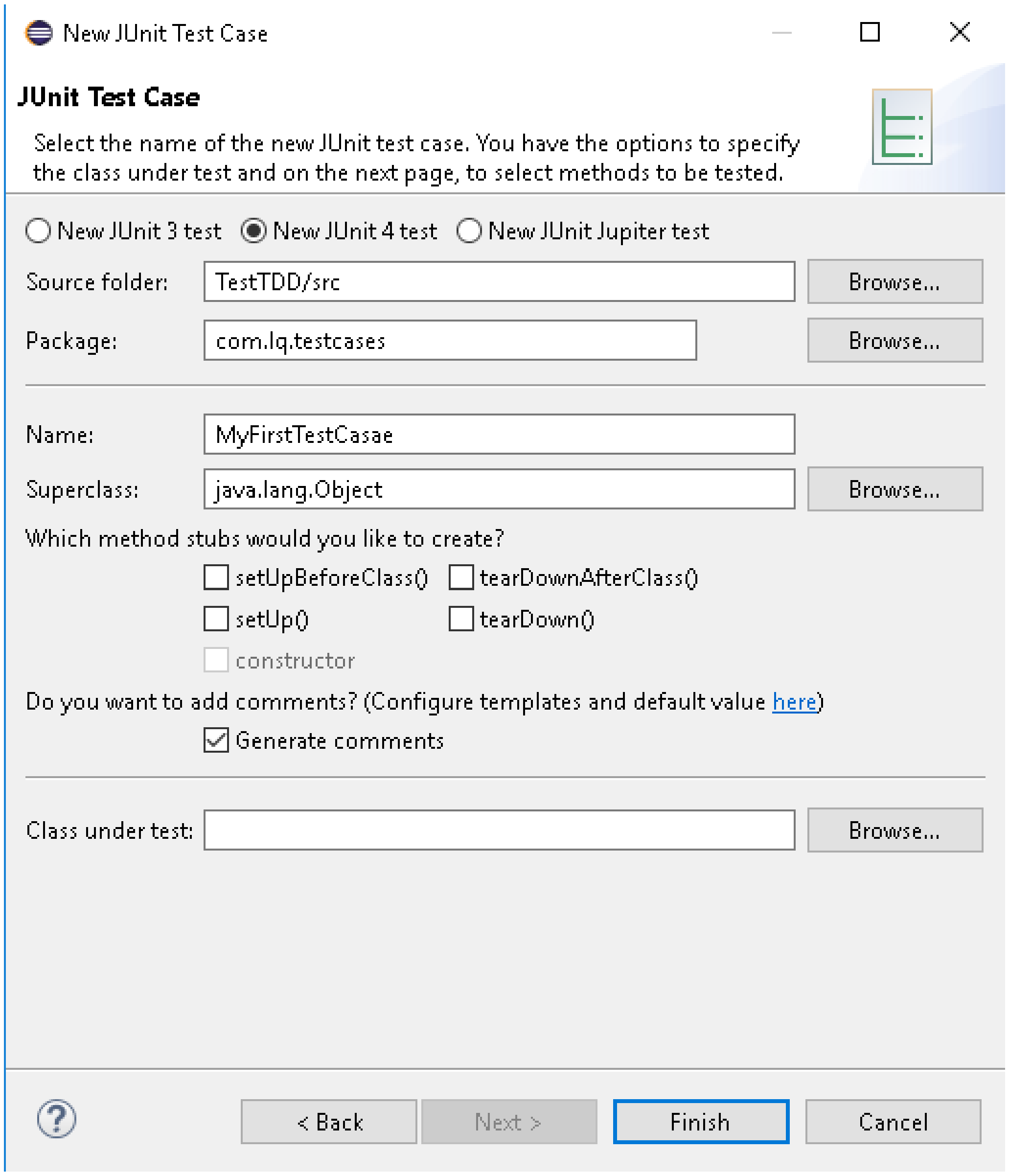
Create a Class to Test and a JUnit 4 Test Class

6. In the Package Explorer view, expand the project by clicking on the ▷ to the left of the project name. Right click on the project source folder (**src**) and select **New** > **Other** from the pop-up menu.

7. Enter "**JU**" (upper or lower case) into the **Wizards** field – you should now see the following:



8. Select JUnit Test Case, and click **Next**. Click the **new JUnit 4 test** radio button, and enter a package name of **com.lq.testcases** and class name of **MyFirstTestCase**. Leave all other fields and checkboxes as they are as none are needed. Click **Finish**. The **JUnit Test Case** wizard should appear as below:



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test] New JUnit Jupiter test

Source folder: TestTDD/src

Package: com.lq.testcases

Name: MyFirstTestCasae

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

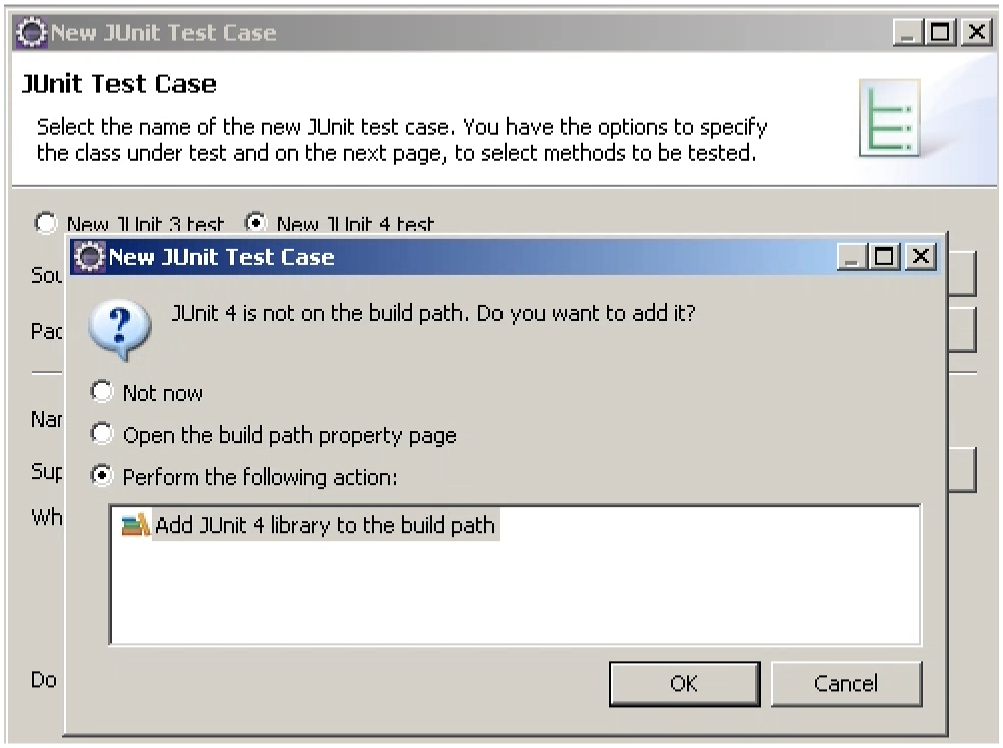
☐ constructor

Do you want to add comments? (Configure templates and default value here)

☑ Generate comments

Class under test:

9. Click OK to add the JUnit 4 library to the build path.



Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

○ New JUnit 3 test ● New JUnit 4 test

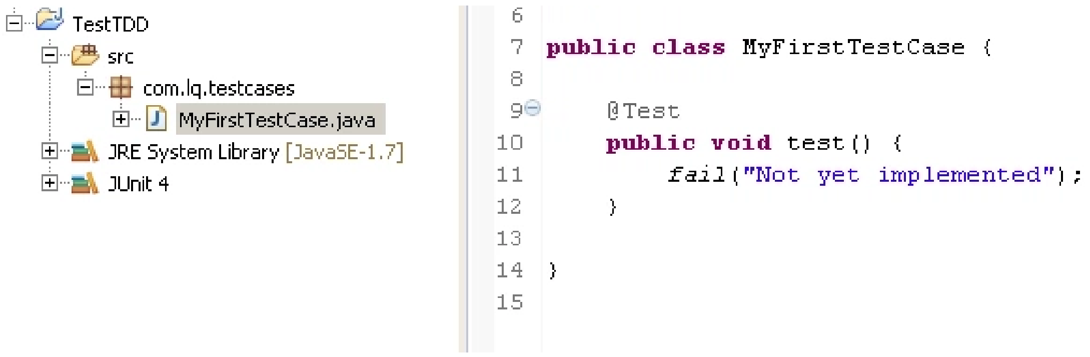
New JUnit Test Case

JUnit 4 is not on the build path. Do you want to add it?

○ Not now
○ Open the build path property page
● Perform the following action:

Add JUnit 4 library to the build path

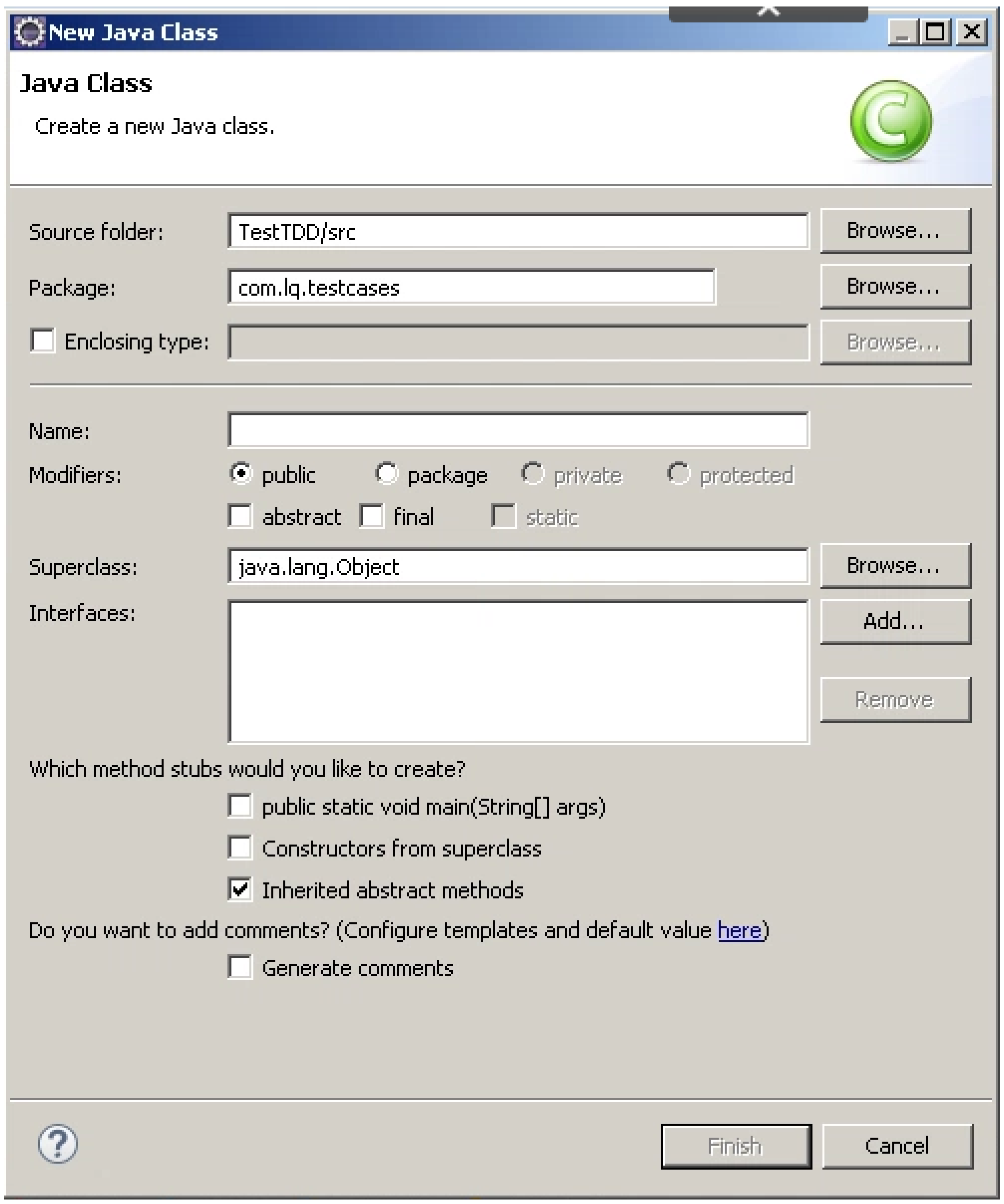
10. You should now see the following:



```

public class MyFirstTestCase {

    @Test
    public void test() {
        fail("Not yet implemented");
    }

}

```

11. Right-click on the **src** folder, and again select New > CLASS.

12. You should now see the **New Java Class** wizard:


New Java Class

Java Class

Create a new Java class.

Source folder:	TestTDD/src	Browse...
Package:	com.lq.testcases	Browse...
☐ Enclosing type:		Browse...
Name:		
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)

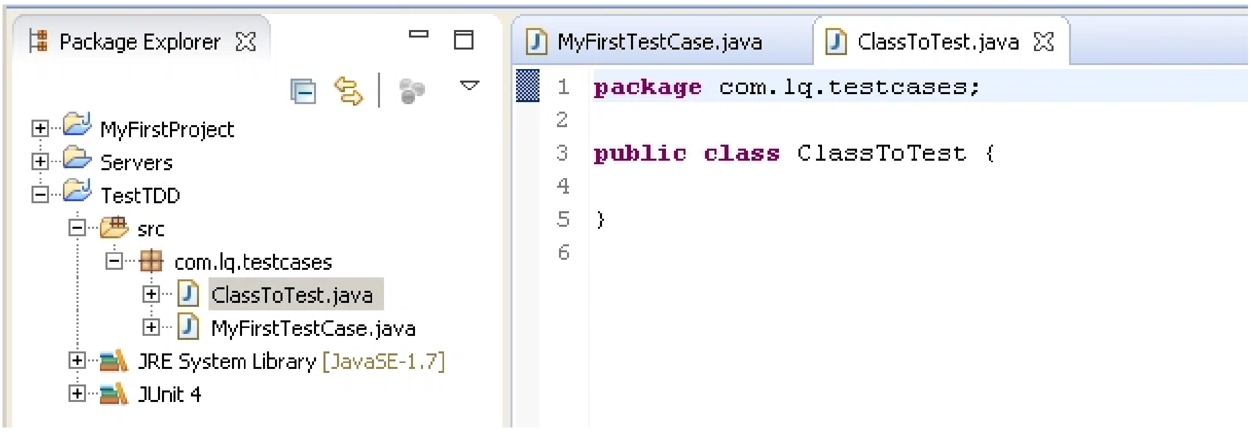
☐ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

13. Enter a package name of **com.lq.testclasses** and class name of **ClassToTest**. Click **Finish**. You should now see the following:



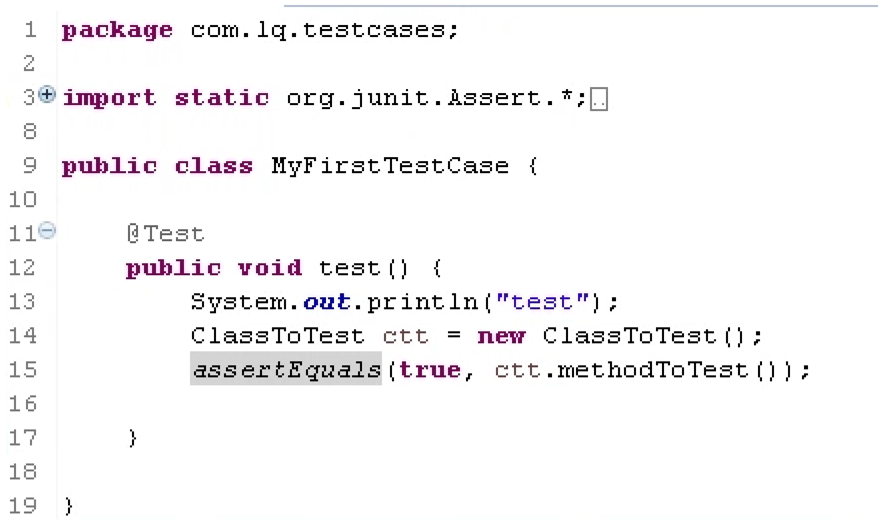
```java
package com.lq.testcases;

public class ClassToTest {

}
```

14. Be sure to save your work on a regular basis: press **control + shift + s** to save all modified resources.

Implement the Method to Test and the Test Methods

15. In the ClassToTest, create a single method named **methodToTest**, which returns a value of Boolean true.

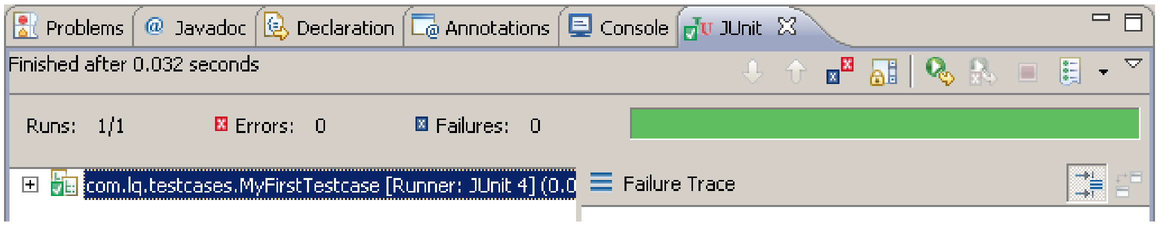
16. In MyFirstTestcase, add the following code to implement the test case:



```java
package com.lq.testcases;

import static org.junit.Assert.*;

public class MyFirstTestCase {

    @Test
    public void test() {
        System.out.println("test");
        ClassToTest ctt = new ClassToTest();
        assertEquals(true, ctt.methodToTest());

    }

}
```

17. Save your changes, and then right-click on MyFirstTestcase in the Package Explorer view, and select Run **as** > **JUnit Test**.

18. Look for two things: in the Console view, you should see "testMethod" from the message sent to sysout, and, secondly, you should see the appearance of a new view—the JUnit view, which should look as follows:



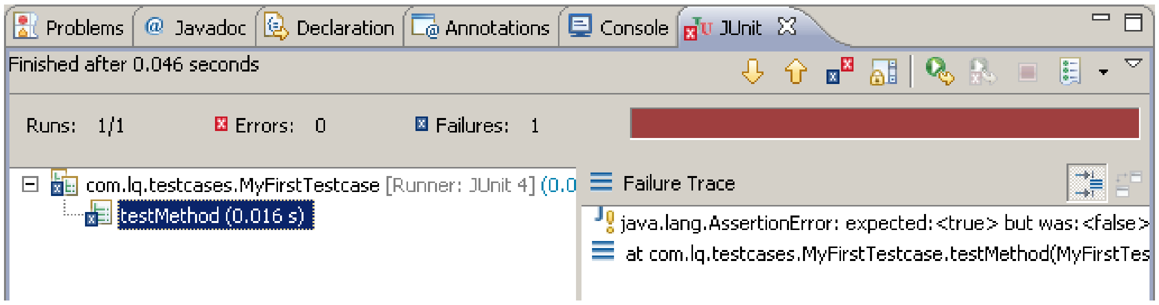
Finished after 0.032 seconds

Runs: 1/1 Errors: 0 Failures: 0

com.lq.testcases.MyFirstTestcase [Runner: JUnit 4] (0.0

19. Notice the green checkbox in the view; this indicates that the test case ran successfully.

20. Change the return value of methodToTest to **boolean false**. Run the test once again. You should now see the following:



Finished after 0.046 seconds

Runs: 1/1 Errors: 0 Failures: 1

com.lq.testcases.MyFirstTestcase [Runner: JUnit 4] (0.0
testMethod (0.016 s)

```
java.lang.AssertionError: expected:<true> but was:<false>
at com.lq.testcases.MyFirstTestcase.testMethod(MyFirstTes
```

21. This time, we see that the test has failed. Notice that as you expand the view of the test case in the JUnit view, you see displayed all failed test methods and the amount of time the test took to run.