LEARNINGPATTERNS

Presentation Slides:
For INSTRUCTOR Use Only

Lab Manual: Fast Track to Java 8 and OO Development

Version: 20160729

Release Level

- This manual contains instructions for creating and running the Fast Track to Java labs using the following Java platforms:
  - **Java 8**
  - **Eclipse Java EE - Luna (4.4) or later** *
- All labs have been tested on Windows Operating Systems
- **Complete lab instructions for this platform are included in the student manual**

LEARNINGPATTERNS

Lab 1.1: HelloWorld

In this lab, we will compile and run a very simple Java program

Lab Synopsis

- **Overview**: In this lab, we will compile and run a very simple Java program
  - We will work from the command line, using the java compiler (`javac`) and JVM (`java`) directly to see how they work
  - All later labs will use the **Eclipse** IDE
  - You will also become familiar with the lab structure
- **Builds on previous labs**: None
- **Approximate Time**: 20-30 minutes

Information Content and Task Content

- In labs, information only content is presented in the normal way –
  - Like these bullets at the top of the page
- Tasks students need to do are in a box like that below
  - So you can see them clearly

Tasks to Perform

- Note the different look of this box as compared to that above
  - All labs will use this format
- Make sure that you have **Java installed**
  - Likely in a directory like *C:\Program Files\Java\jdk1.8.0_101 **
  - If not, you'll need to install it - It can be downloaded from:
  - **www.oracle.com/technetwork/java/javase/downloads/index.html**
- OK – Now **get out your setup files**; we're ready to start working

Extract the Lab Setup Zip File

Lab

- You need the course setup zip file for the labs *
  - It has a name like: ***LabSetup_FTJ8_20160729.zip***
- The base lab folder is: ***C:\StudentWork\FTJ***
  - Created when we extract the Setup zip
  - It includes a directory structure and lab starter files
- Lab folders are under: *C:\StudentWork\FTJ**workspace***
  - We'll refer to this as ***workspace***
- Instructions assume that this zip file is extracted to C:\
  - If you extracted elsewhere, then adjust accordingly

Tasks to Perform

- Unzip the lab setup file to **C:**
  - This creates the directory structure, described in the next slide, with files needed for the labs

Writing Hello World

Lab

Tasks to Perform

- The root lab folder is: **workspace\Lab01.1**
  - It should already exist,
  - Create all files in this folder, and build the application there
- Write a class called **`HelloWorld`** in a file ***HelloWorld.java***
  - Include a main method
  - Write it exactly as shown below, using whatever text editor you like
  - **JAVA IS CASE SENSITIVE!!!**
  - See note if you are using Notepad
  - Save this file in *workspace\Lab01.1*

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

Checking Your Environment

Lab

Tasks to Perform

- Open the file *Lab01.1**setEnv.cmd***
  - Set the JAVA_HOME setting for your Java version (e.g. Java 8)
  - If necessary, modify it to be correct for your install *

- Open a command prompt in the lab directory
  - Execute `setEnv`, then execute the JVM as shown below to make sure you are correctly set up to use the java tools on your system

```
> setEnv
  … Output of setEnv not shown
> java -version
```

```
C:\StudentWork\FTJ\workspace\Lab01.1>java -version
java version "1.8.0_20"
Java(TM) SE Runtime Environment (build 1.8.0_20-b26)
Java HotSpot(TM) 64-Bit Server VM (build 25.20-b23, mixed mode)
```

Compiling HelloWorld

Lab

Tasks to Perform

- Compile the Java source code from the same command prompt where you ran `setEnv`
  - Invoke the compiler to compile *HelloWorld.java* by typing:

```
>javac HelloWorld.java
```

  - If you get a message saying javac isn't found, check your path
  - If you have compilation errors, see the notes for hints

- Note: javac **expects** you to type the **.java** extension for the file you're compiling

- When compiling finishes, you should have a file in the lab folder called *HelloWorld.class*
  - This is the bytecode produced by the compiler

Running HelloWorld

Lab

Tasks to Perform

- Execute your program from the same command prompt where you ran `setEnv`
  - Using the Java interpreter (or JVM) - i.e. the **`java`** executable
  - It will execute **HelloWorld**'s `main()` method
  - Run `HelloWorld` by typing:

```
>java HelloWorld
```

  - Note: The JVM does **not** expect you to type the .class extension for the file you are using

- You should see the output "Hello World" printed in your command prompt window
  - Congratulations - Your first Java program !

LEARNINGPATTERNS

Lab 2.1: The Development Environment

In this lab, we'll become familiar with the Eclipse development environment

Lab Synopsis

- **Overview**: In this lab, we'll become familiar with the Eclipse development environment
  - We will start up the Workbench, use its capabilities, and create and use a project in Eclipse
  - We'll also set up and use a project in Eclipse
- **Builds on previous labs**: None
- **Approximate Time**: 30-40 minutes

The Eclipse Platform

Lab

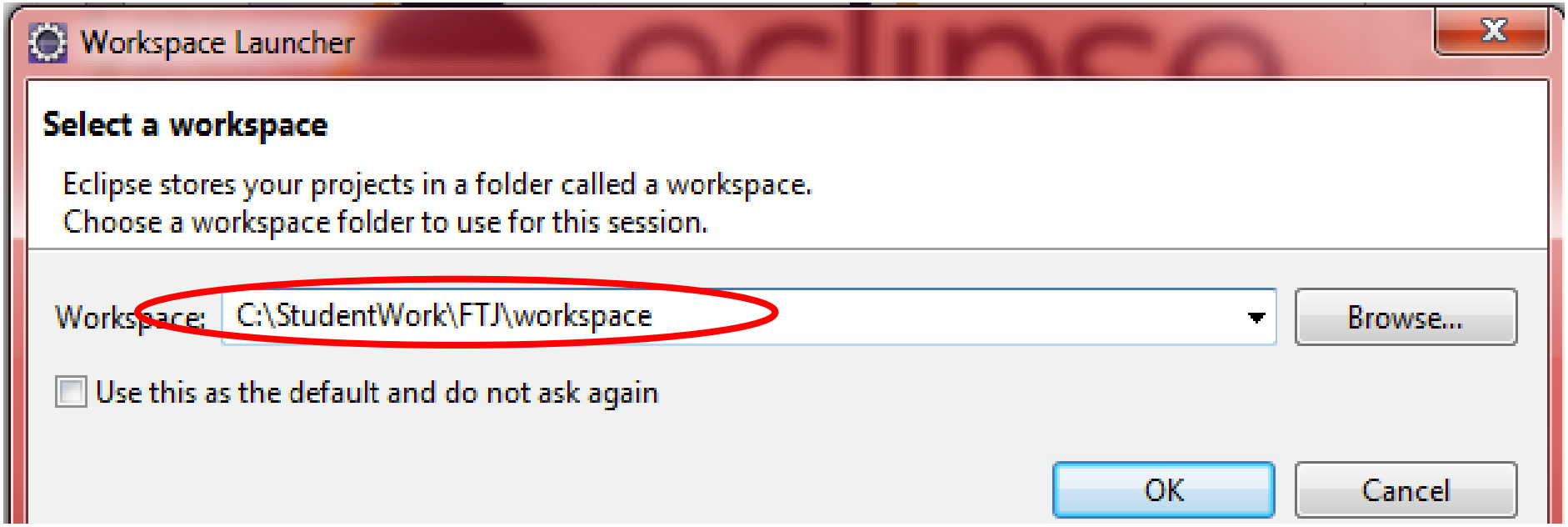
- **Eclipse** (**www.eclipse.org**): Open source platform for building integrated development environments (IDEs)
  - Used mainly for **Java development** - can be extended via plugins and used in other areas (e.g. C# programming)
- Originally developed by IBM
  - Released into open source
  - IBM's RAD environment is built on top of Eclipse
- Eclipse products have two fundamental layers
  - The **Workspace** – files, packages, projects, resource connections, configuration properties
  - The **Workbench** – editors, views, and perspectives
- We will set up the workspace and workbench, then describe it in more detail at the end of the lab

Getting Started With Eclipse

Lab

Tasks to Perform

- Make sure you have **Eclipse installed** - likely in C:\eclipse
  - If not - you'll need to install it - see instructions in notes
- **Launch Eclipse**: Go to **c:\eclipse** and run **eclipse.exe**
  - A dialog box should appear prompting for a workspace location
  - Set the workbench location to **C:\StudentWork\FTJ\workspace**
  - If a different default workspace location is set, change it

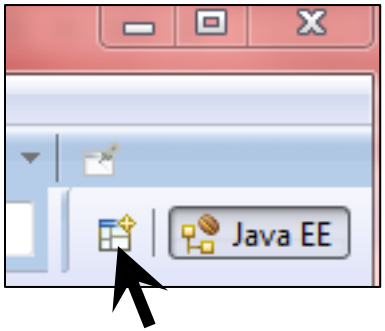


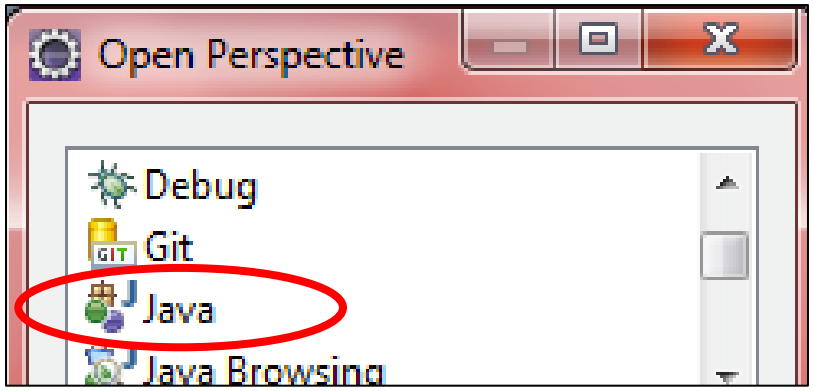
Workbench and Java Perspective

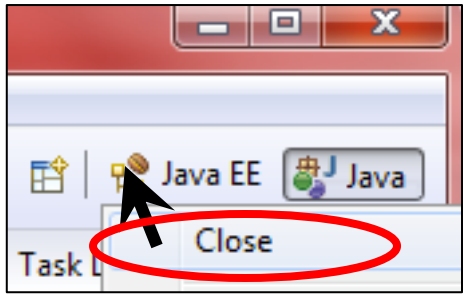
Lab

Tasks to Perform

- Close the Welcome screen (click the X on its tab – see notes)
- **Open a Java Perspective**:
  - Click the Perspective icon at the top right of the Workbench
  - Select Java (as shown below)
  - The Java EE perspective is the current default for Eclipse Java EE
- Close the Java EE perspective by right clicking its icon, and selecting close (as shown below right)





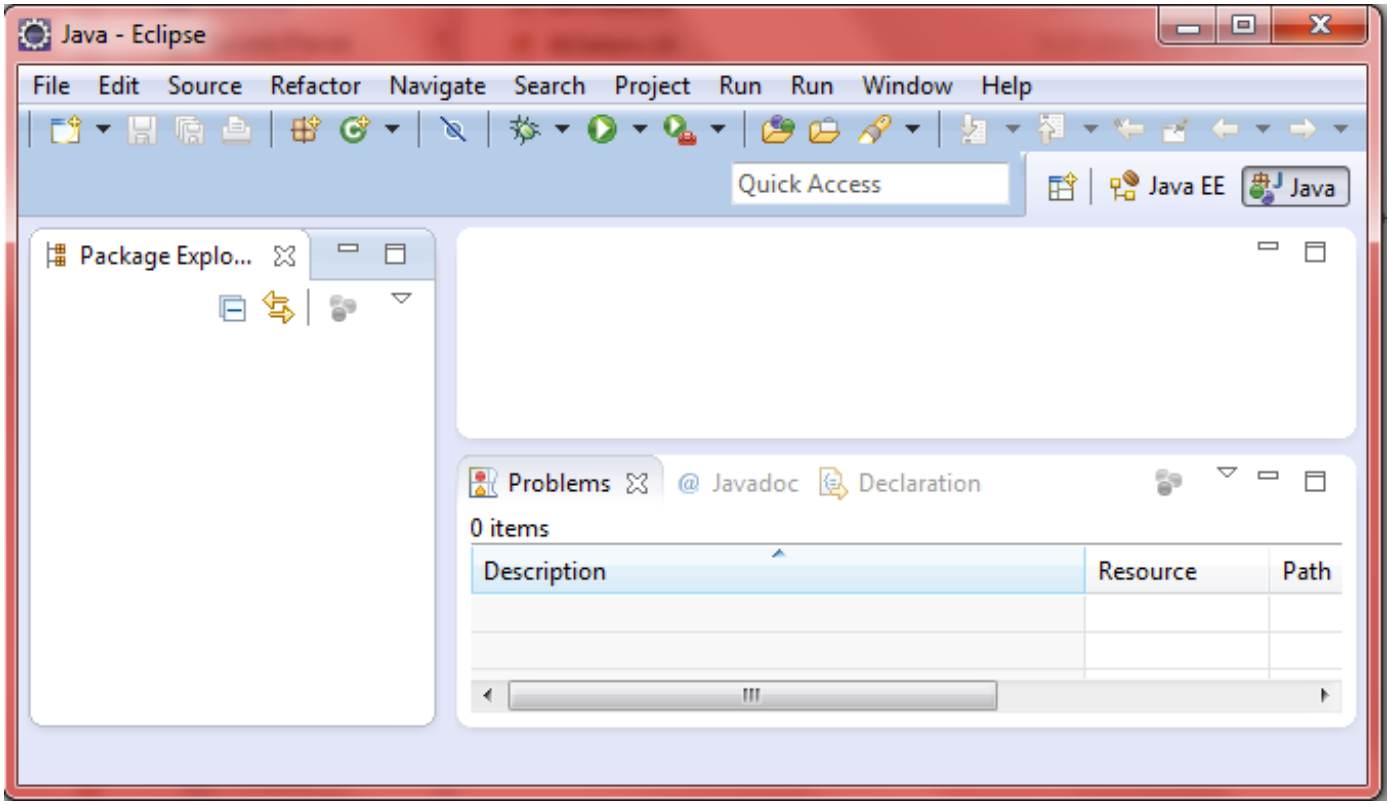


Unclutter the Workbench

Lab

Tasks to Perform

- Let's unclutter the Java Perspective by closing some views
  - Close the Task List and Outline views (click on the X)
  - Open the Navigator View (**Window | Show View | Navigator**)
  - You can save this as the default if you want (see note)

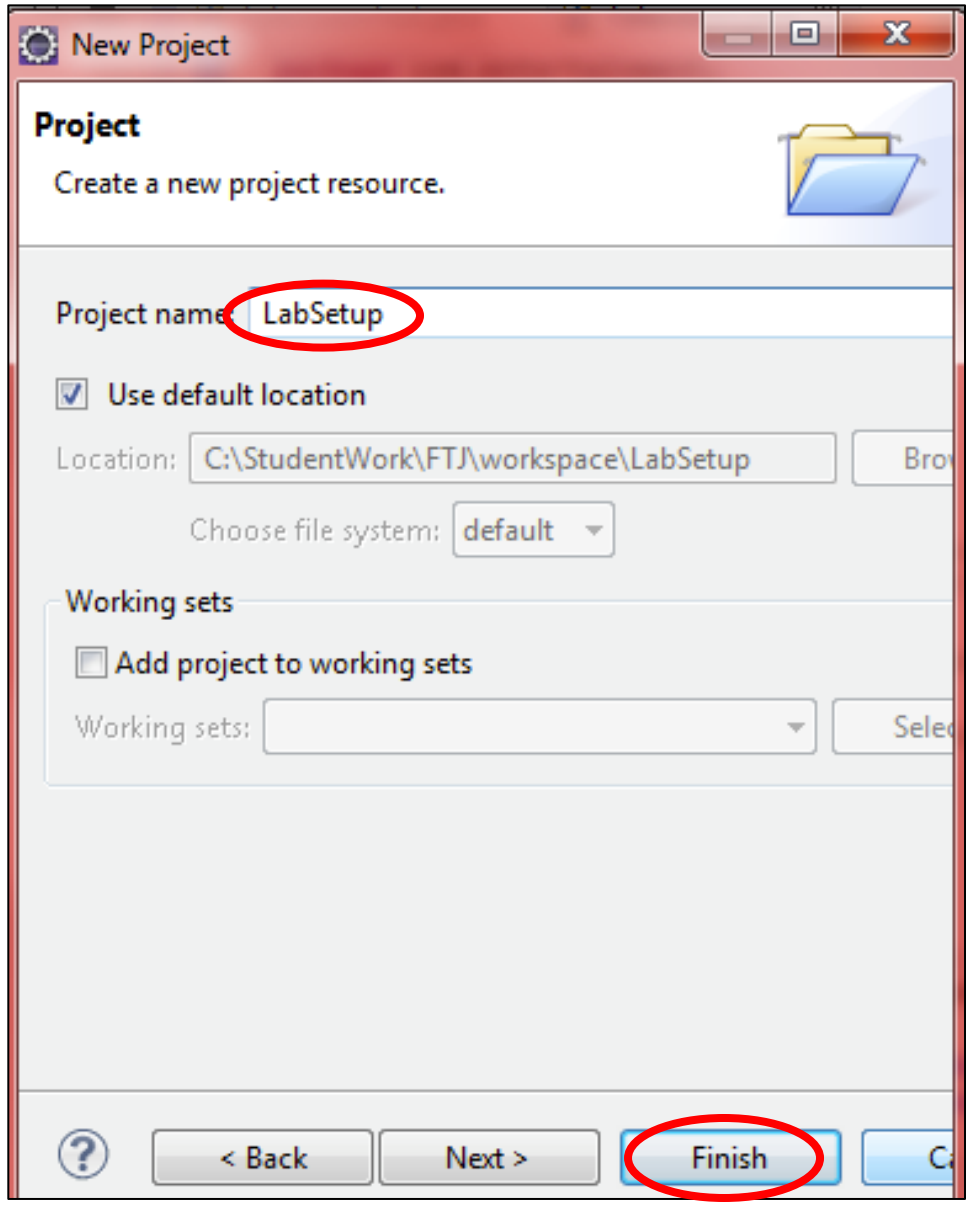


LabSetup Project

Lab

Tasks to Perform

- We'll first create a project to access lab setup files
  - You won't do any work in this project - just copy files from it
- Create a new General project
  - **File | New | Project | General | Project**
- Fill in the name as **LabSetup**
  - Click **Finish**
  - We will use this in later labs
  - It's not needed in this lab

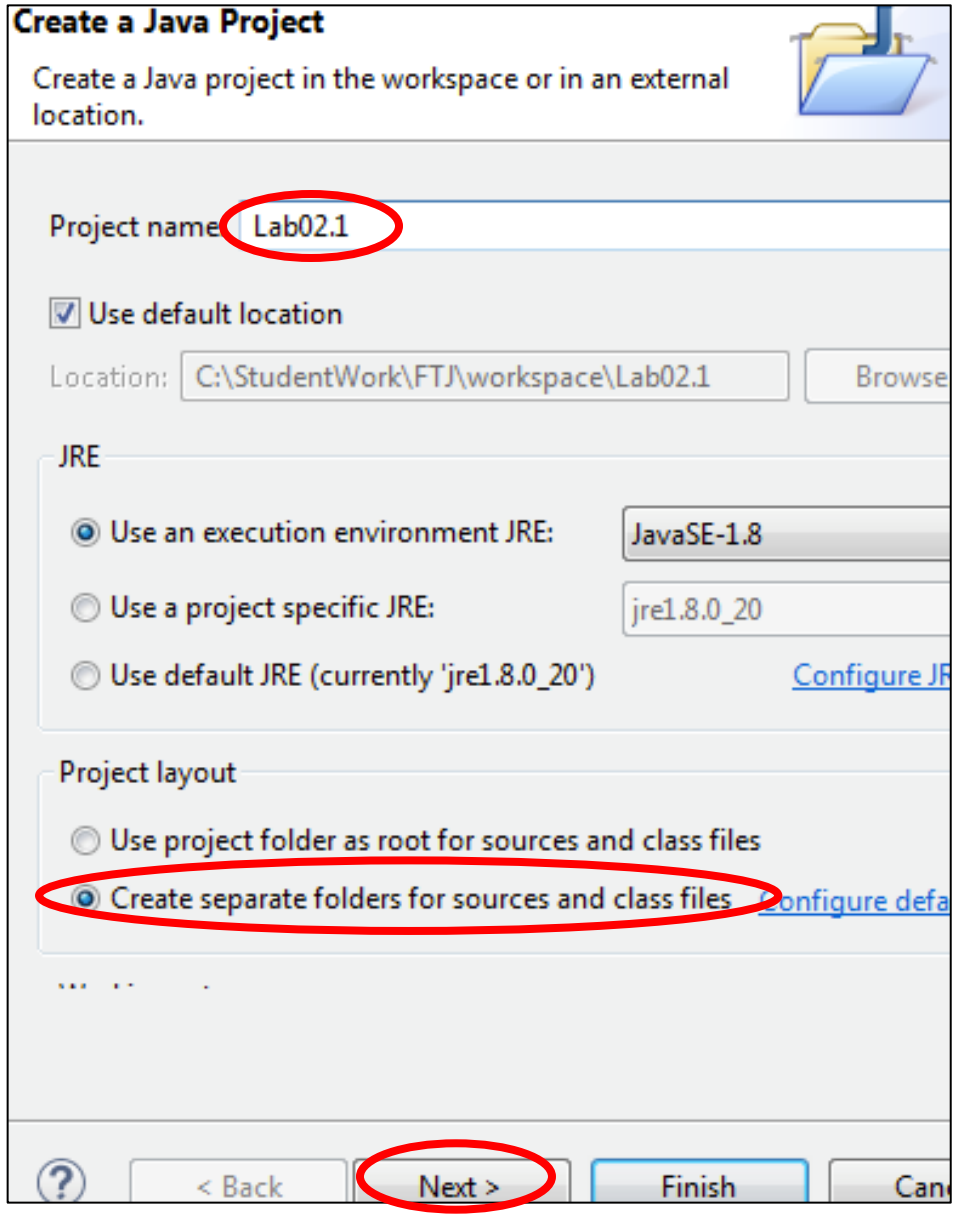


Java Project

Lab

Tasks to Perform

- Create a new Java project
  - **File | New | Java Project**
- Fill in the name as **Lab02.1**
  - Make sure "**Create separate folders for source and class files**" is selected
  - Click **Next**
- There are many ways to create a project (see notes)
  - When you call the project Lab02.1, by default it will be stored in ***workspace\Lab02.1***

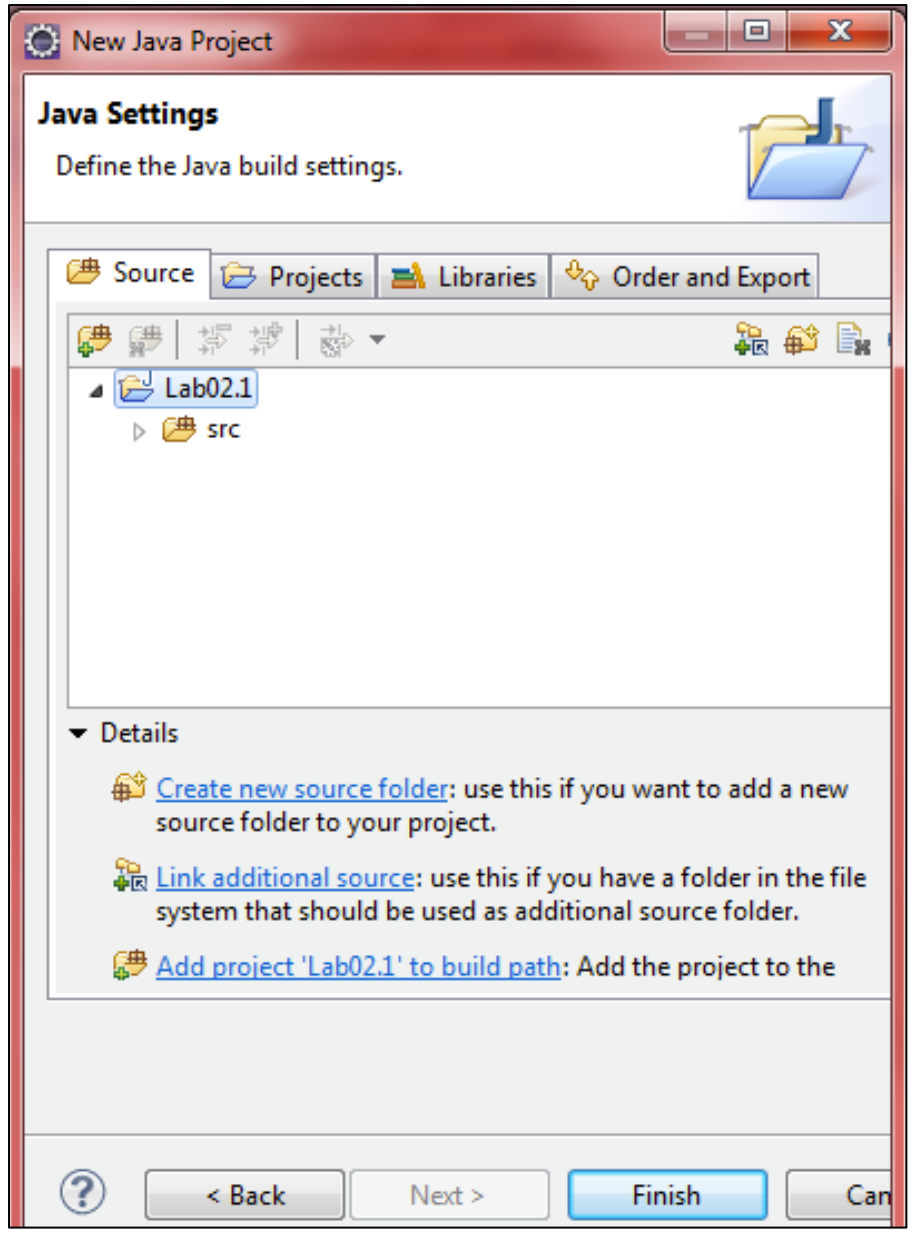


Finish Java Project

Lab

Tasks to Perform

- The defaults are fine in the next dialog
  - Click **Finish**

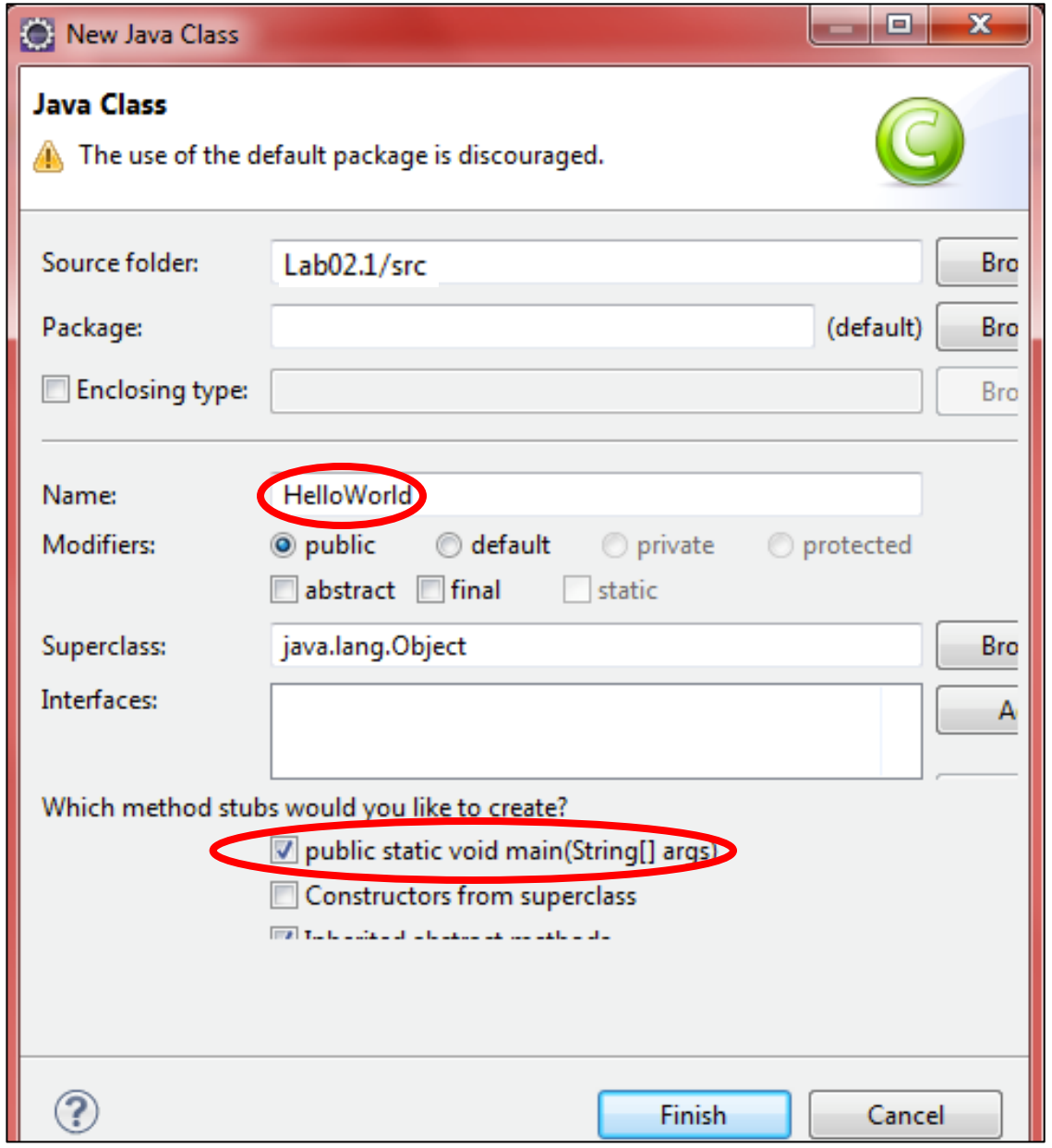


Your First Application

Lab

Tasks to Perform

- Create a new Java class within the project *
  - **File | New | Class**
- Call the class **`HelloWorld`** as in your first program
  - Generate a main method by checking the box in the dialog
- Click **Finish**
  - *HelloWorld.java* will open in the editor
  - Add the same code as in the first lab



Compilation and the Problems View

Lab

Tasks to Perform

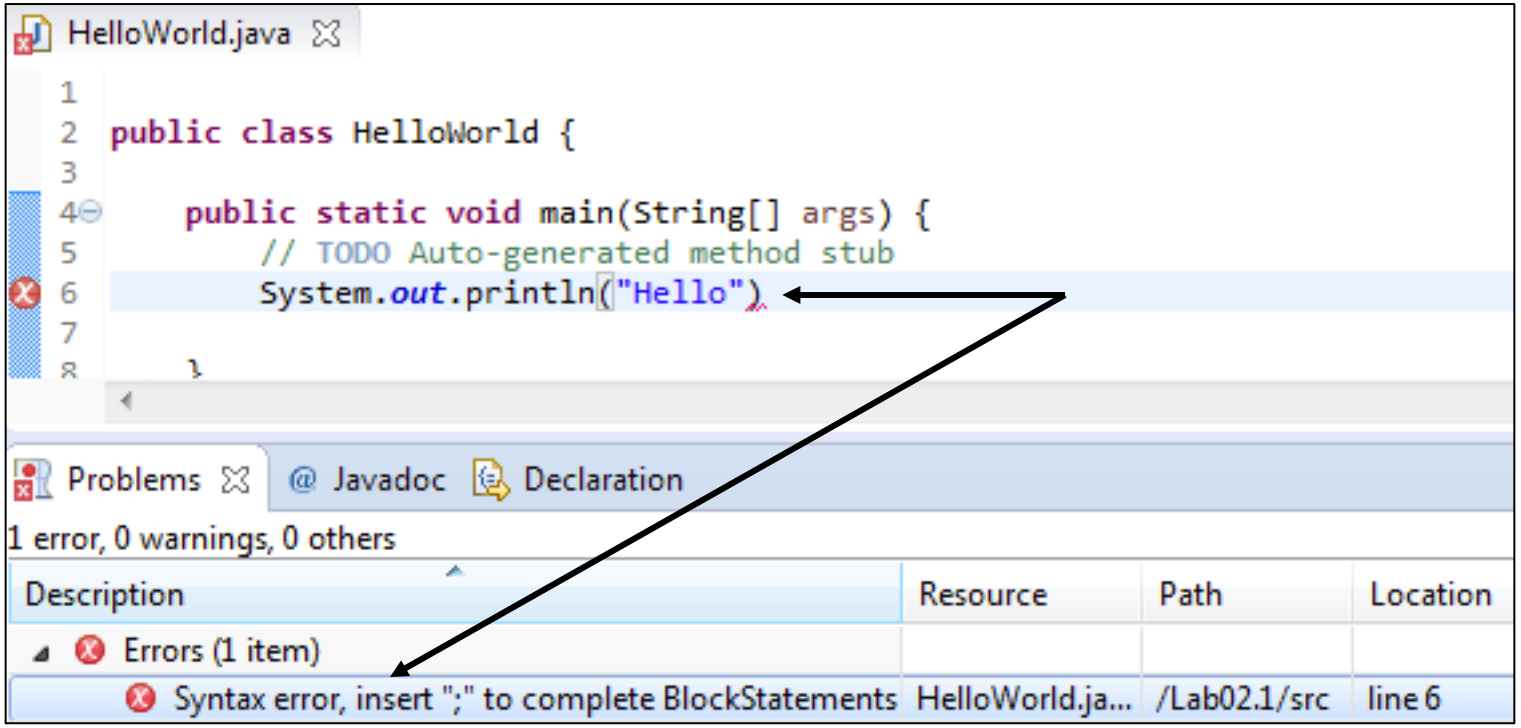
- The file should compile automatically when you save it (1)
  - Resolve any warnings or errors the compiler adds to the Problems view (see next slide)
- Add an error in your code and save the file
  - You should see the error in the Problems view
- The **Problems view** gets populated by problems Eclipse detects in your code
  - A compiler may add errors and warnings that need resolution
  - Eclipse may add it's own warnings (2)
  - **Double click** on a problem in Problems view to jump to the code associated with that problem

Seeing a Problem

Lab

Tasks to Perform

- Note the missing semicolon at the end of line 17
  - Note the error message in the problem view
  - Note: Line numbers can be turned on/off via the menu selection: ***Window | Preferences | General | Editors | Text Editors***

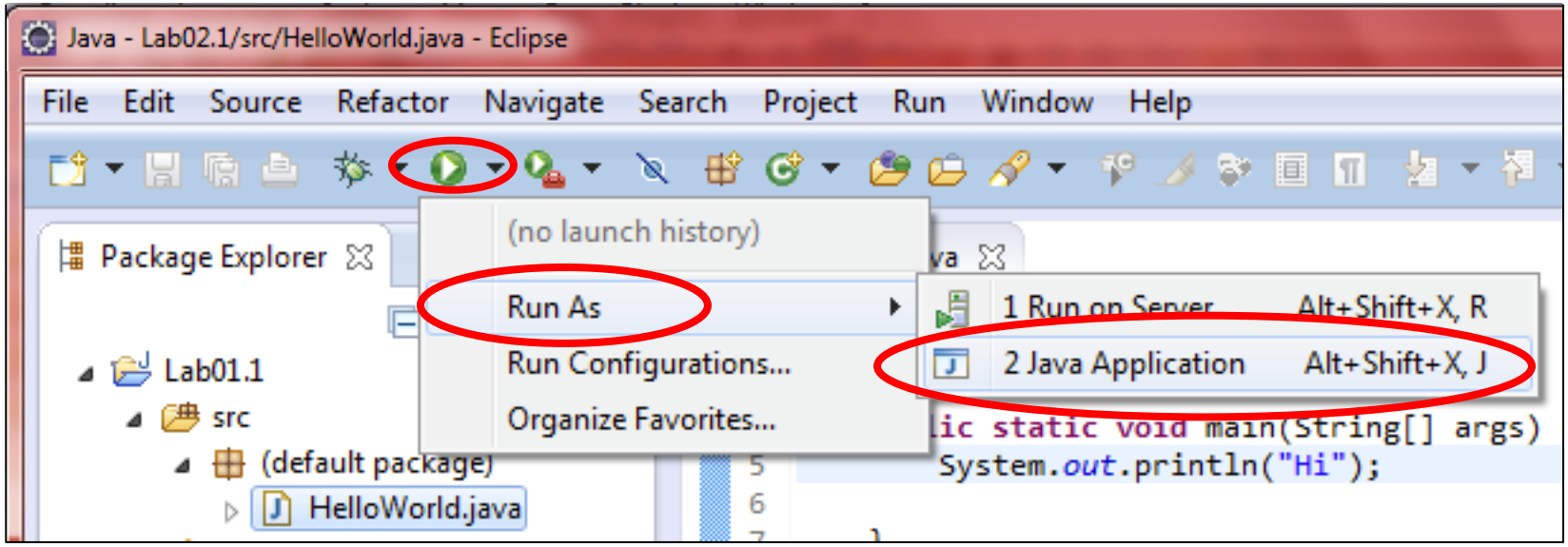


Testing your Application

Lab

Tasks to Perform

- After a clean build (error-free, but not necessarily warning-free), test the application as follows
  - Select *HelloWorld.java* in the Navigator or Package Explorer view
  - Click the run button arrow on the task bar *
  - Choose **Run As | Java Application** from the menu that appears

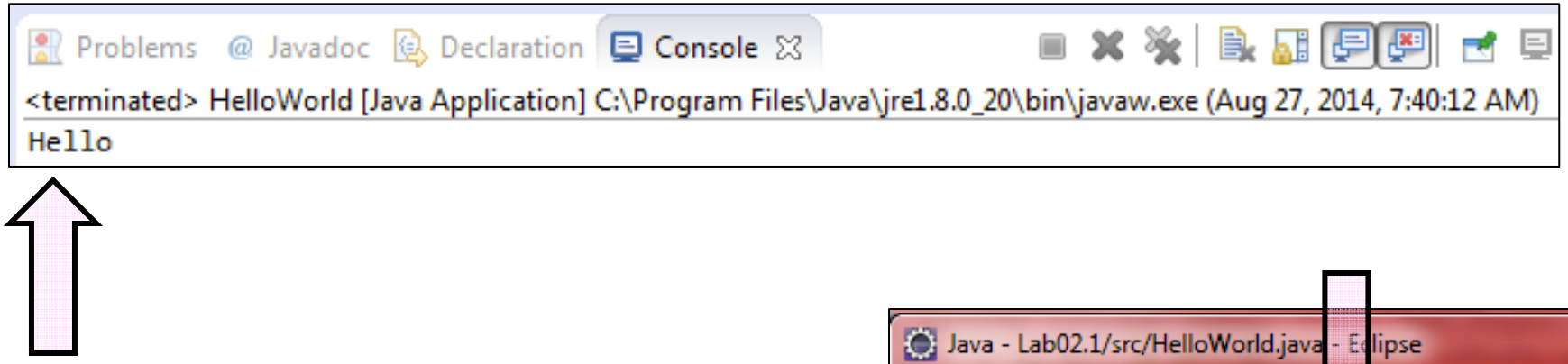


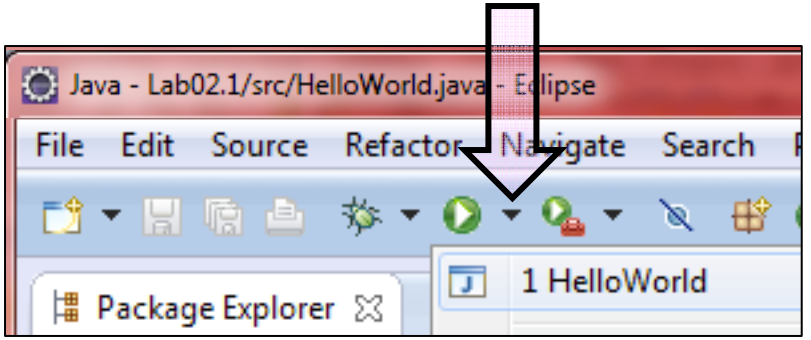
Viewing Results

Lab

Tasks to Perform

- You'll see results in the console view as shown below
  - If necessary, open the Console (**Window | Show View | Console**)
- To run again, you can press the **Run Icon arrow** as shown at bottom
  - This brings up a list of previously run programs that you can pick from
  - Just select the **`HelloWorld`** program

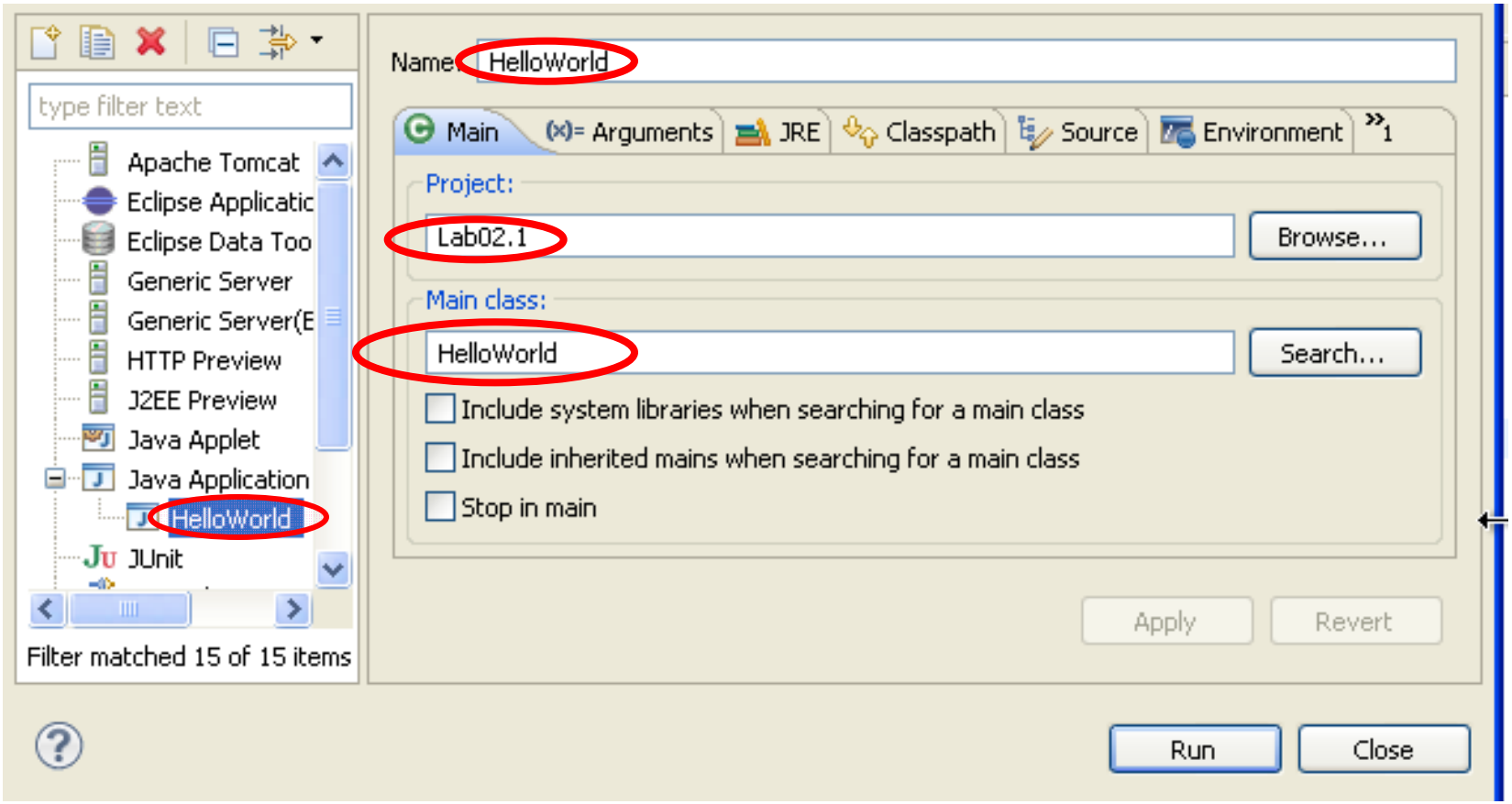




Launch Configurations

Lab

- Eclipse creates a **Launch configuration** to run a program
- Lets you customize the execution of an application
  - Review it by going to the **Run icon**, selecting **Run Configurations …**, and selecting **HelloWorld** from the next dialog

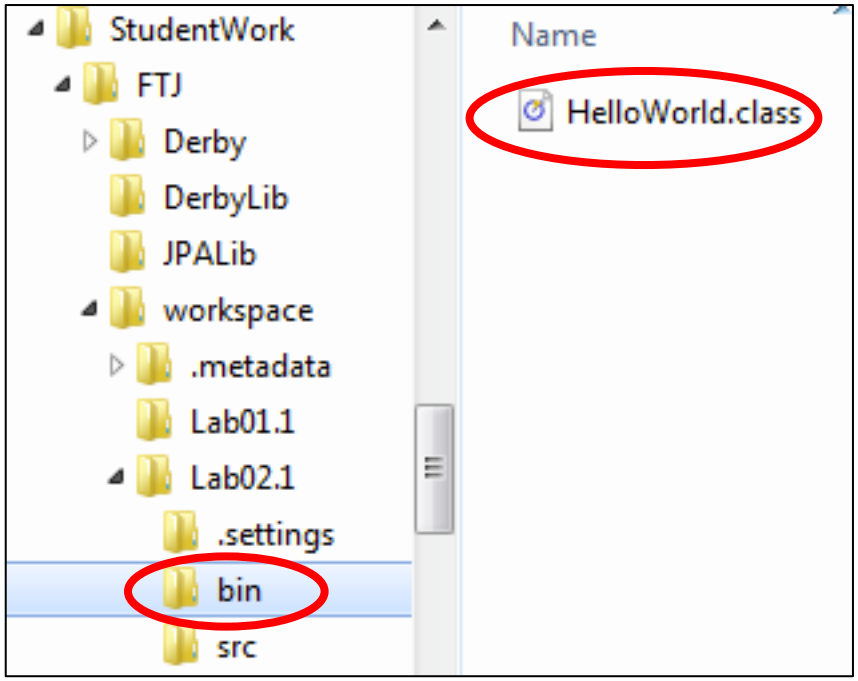


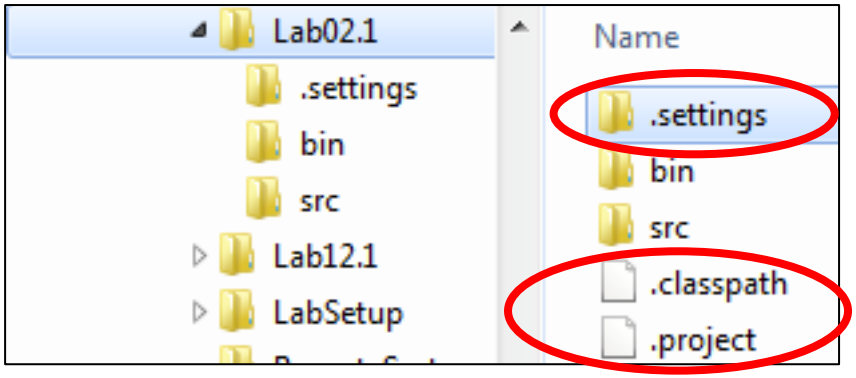
The Project Directory

Lab

Tasks to Perform

- Look at the *workspace\Lab02.1* - src and bin folders
  - You'll see *HelloWorld.java* and *HelloWorld.class* files (in src and bin)
  - In the project root, there is a*.settings* folder, *.project* and *.classpath* files
  - These are used by Eclipse to maintain the project





Important Notes for Using Eclipse

Lab

- Any lab that has a **new lab directory** will require you to create a **new Eclipse project**
  - Sometimes several labs are done one directory, in which case you will use the same project for all of them
- Do all copy/pasting of setup files **within Eclipse**
  - Usually copying files from the appropriate LabSetup subfolder
  - Pasting them into the project you are working on
- For anyone not familiar with Eclipse, the next few slides give a (very) brief overview of how Eclipse is structured
  - There is **nothing you need to do** in those slides – they are for information purposes only

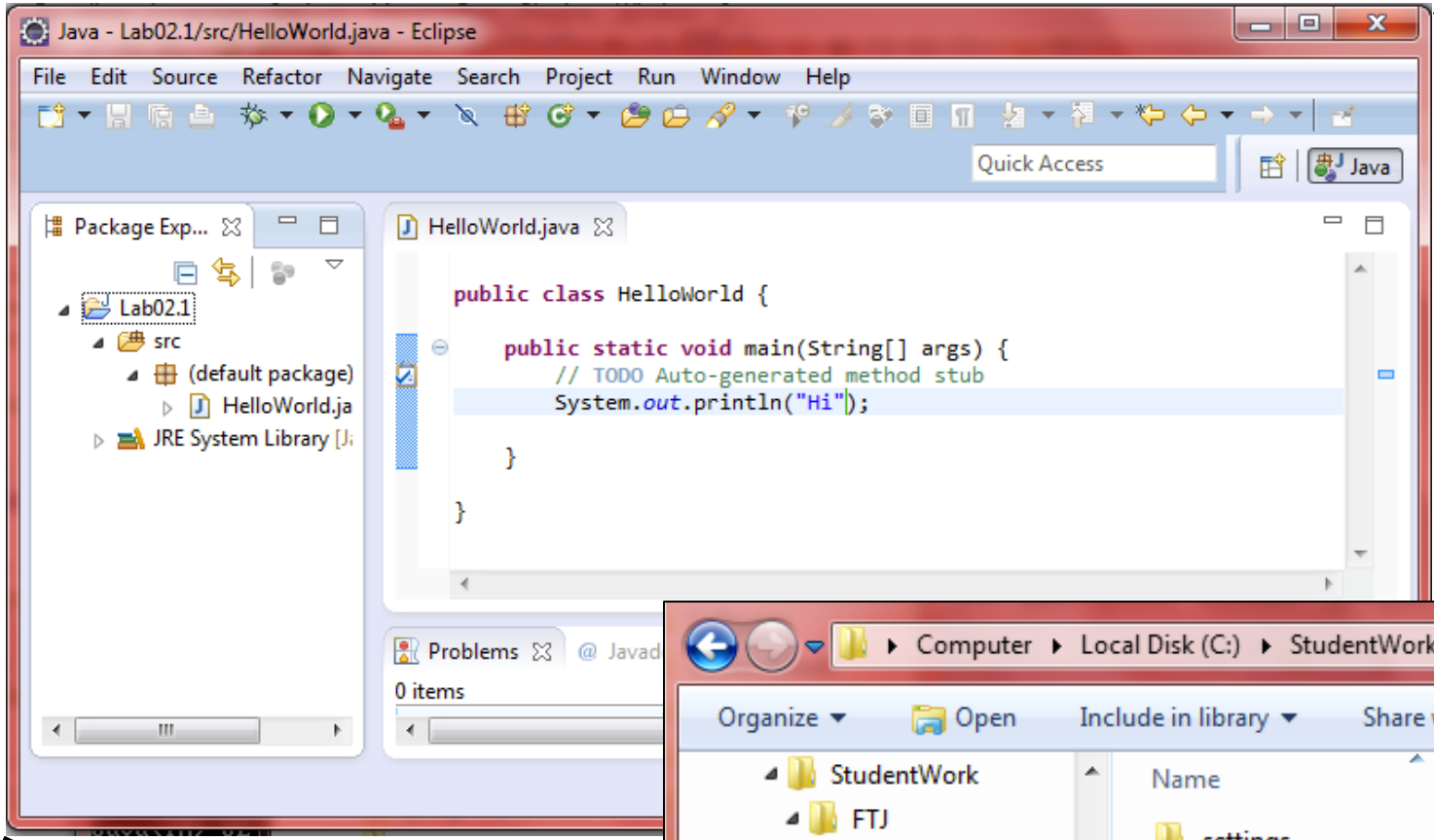
The Eclipse Paradigm

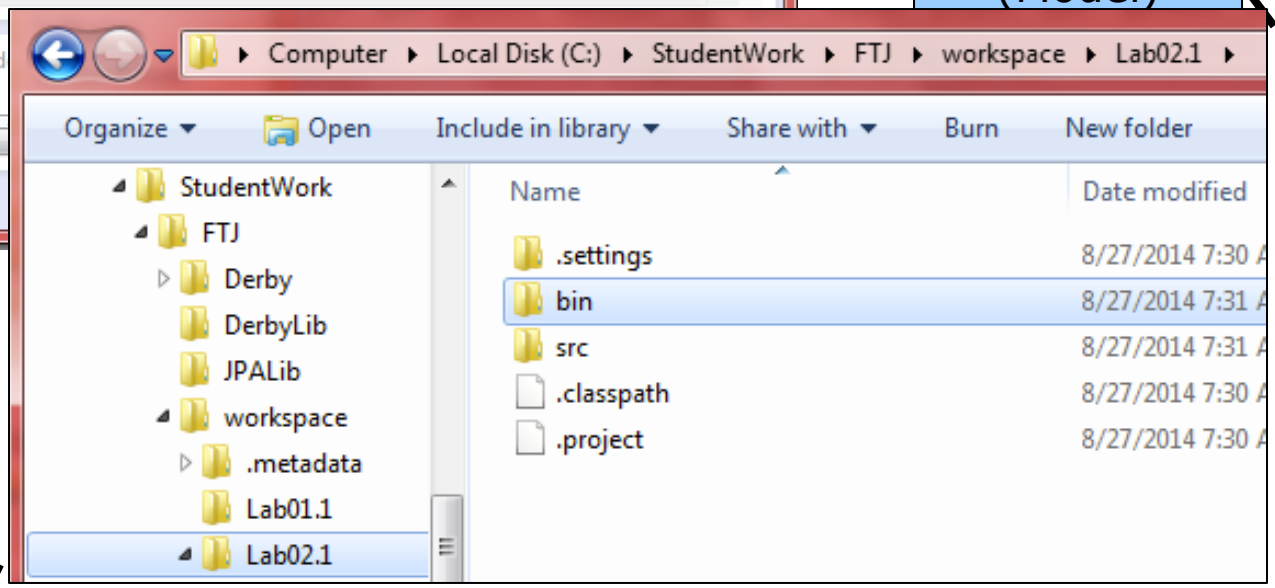
Lab

- Eclipse products have two fundamental layers
  - The **Workspace** – files, packages, projects, resource connections, configuration properties
  - The **Workbench** – editors, views, and perspectives
- The Workbench sits on top of the Workspace
  - Provides views to access/manipulate resources
  - **Editor** – A component that allows a developer to interact with and modify the contents of a file.
  - **View** – A component that exposes meta-data about the currently selected resource.
  - **Perspective** – A grouping of related editors and views that are relevant to a particular task and/or role.
- You can have multiple perspectives open to provide access to different aspects of the underlying resources

Workbench and Workspace

Lab





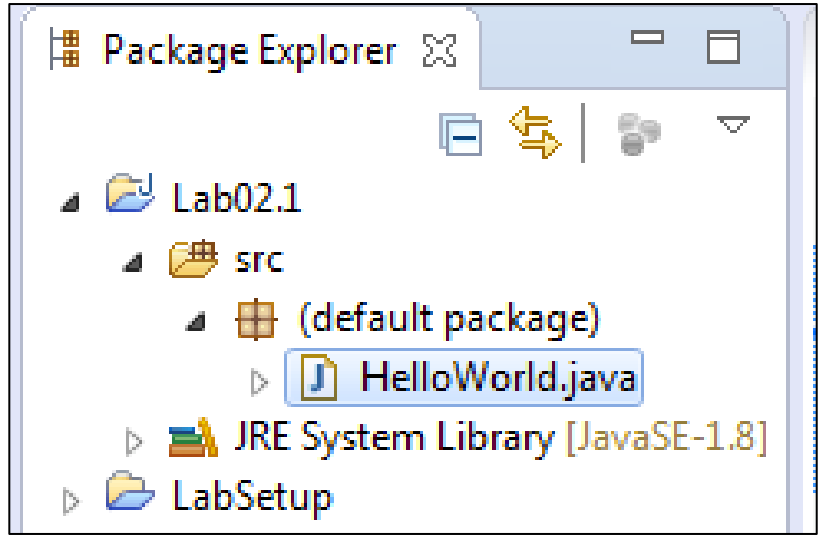
Workspace (Model)

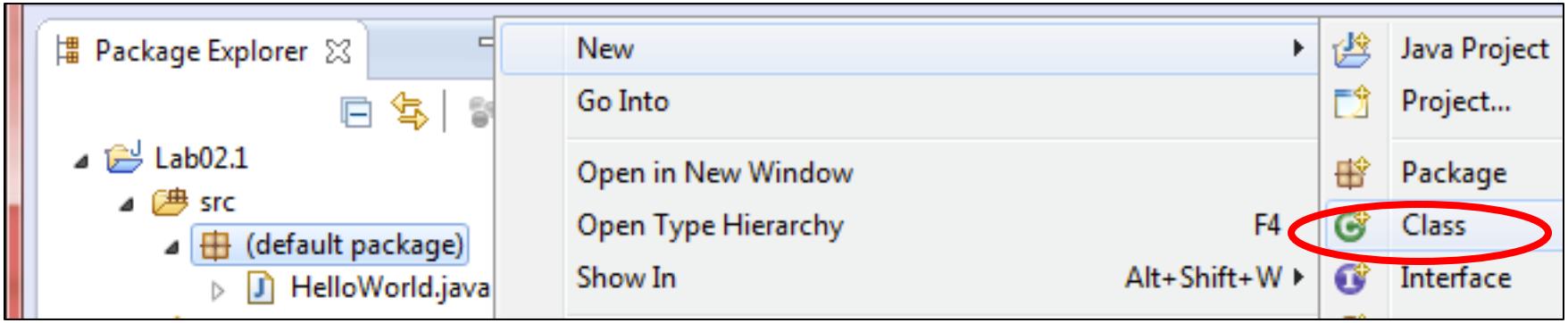
Workbench (View)

Package Explorer View

Lab

- Shown by default in the Java perspective
  - A Java-specific view of resources
  - Shows Java element hierarchy of projects
  - Easy to use with Java resources
  - e.g., to create a new class, right click on the package (1) you want and select **New | Class**

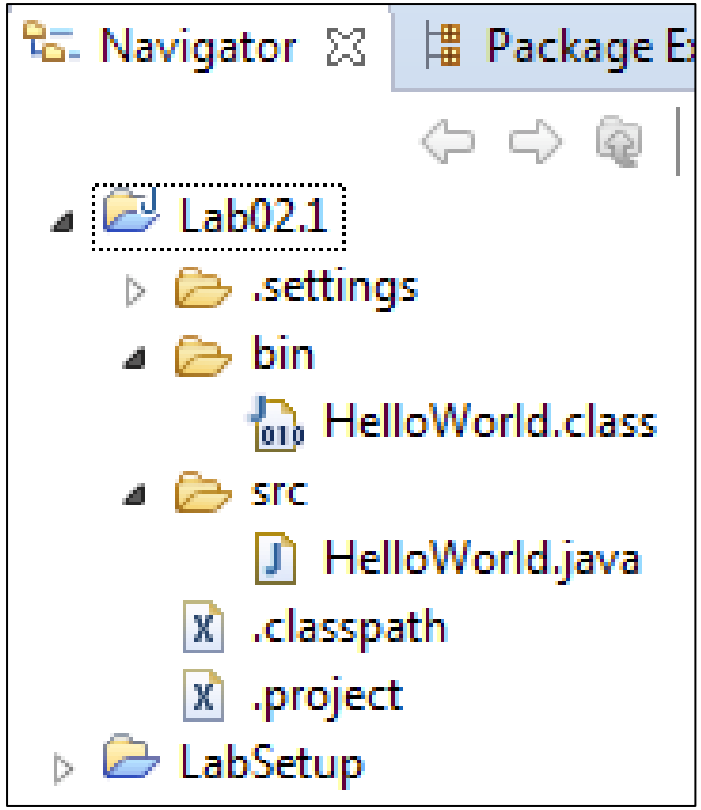




Navigator View

Lab

- Similar to file system view
  - There are three kinds of resources described below
- **Files**
  - Correspond to files on the file system
- **Folders**
  - Like folders on the file system
- **Projects**
  - Used to organize all your resources and for version control.
  - Creating a new project assigns a physical location for it on the file system.
  - A third-party SCM (Source Control Manager) may be used to properly share project files amongst developers.



Editors

Lab

- There is a customized source editor (like this one for a .java file) for all character files. (.java, .jsp, .html, etc.)

HelloWorld.java

```
1
2  public class HelloWorld {
3
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6          System.out.println("Hello");
7
8      }
9
10 }
11
```

LEARNINGPATTERNS

Lab 3.1: Exploring Types

In this (discussion only) lab, we'll explore the notion of defining types

Lab Synopsis

Lab

- **Overview**: In this (discussion only) lab, we'll explore the notion of defining types much like the ones you might define in a program
  - We'll define a type's characteristics (its name, properties, and behavior)
- **Builds on previous labs**: None
- **Approximate Time**: 15-20 minutes

Create Your Own Types

Lab

Tasks to Perform

- Split up into pairs or small groups, and work together on creating several types, including their properties and behavior
  - You can use types from projects that you are working on in your company, or make up a new problem domain
  - Discuss each type a little – does everyone agree that this is a type? Discuss its properties and behavior
  - Discuss the names of your types – names are important
    - They convey information to users of the type
  - Have one member from each of a few different teams go to the front of the room and present their types

LEARNINGPATTERNS

Lab 3.2: Identity and Object References

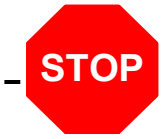
In this (discussion only) lab, we'll explore the notion of object instances and references

Lab Synopsis

- **Overview**: In this lab, which is discussion only, we'll explore the use of object references and object identity
- **Builds on previous labs**: None
- **Approximate Time**: 10 minutes

Tasks to Perform

- Everybody look at the instructor
  - Do you all agree that the instructor exists, or has identity? Does this identity depend on the instructor's name?
- Now everybody else in the room point at the instructor – we know that's not polite, but hopefully s/he will forgive us
  - You are all **referring to** this person – you are all **object references** to the **same object**
  - Hopefully, it's clear that the object references are not the object itself
  - Keep this in mind as you're programming – much of an object-oriented program consists of passing object references around



LEARNINGPATTERNS

Lab 3.3: Writing a Class Definition

In this lab, we will create a class that has methods and instance variables

Lab Synopsis

- **Overview**: In this lab, we will create a class that has methods and instance variables
  - We will also write a run a test program that creates instances of the class and works with those instances
  - NOTE: we will continue to modify and add to this class as we learn more of Java's capabilities

- **Builds on previous labs**: None
  - The new lab folder and project is **Lab03.3**

- **Approximate Time**: 25-35 minutes

Sample Class Definition and Usage

- The examples below show defining and using a class
  - This is **not the lab solution** – but shows helpful examples

```
public class AlarmClock {
  int snoozeInterval = 5;

  int getSnoozeInterval() {   // Must return an int
    return snoozeInterval;   // Returns the instance var
  }

  void snooze() { /* … */ }
}
```

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    // …
    int snoozeInterval = myClock.getSnoozeInterval();
    System.out.println("Need more sleep:" + snoozeInterval);
    myClock.snooze();
  }
}
```

Lab Preparation

- The root lab folder is **workspace\Lab03.3**
  - This is a new lab directory
- We'll create a **`Television`** class
  - It will have brand and volume fields to work with the concepts we just learned (but **no main method**)
- We'll also create **`TelevisionTest`** class
  - It **will** have a `main` method where you create instances of `Television`, get the property values using the get methods, and print them
  - Note that the word property is often used to describe the data values of an object

Tasks to Perform

- Close all files and projects you have open (1)
- Create a new Java project called **Lab03.3** in your workspace
  - See the earlier lab's instructions if you need details

Creating a Class

Lab

Tasks to Perform

- Create a new class called `Television`
  - It will be saved in a file ***Television.java***
  - Use the Eclipse wizard similarly to the earlier labs
  - In your `Television` class, declare two fields:

```
String brand;  // brand of Television
int volume;    // current volume
```

- Write getter methods to access each field
  - The method names should start with **`get`** and be followed by the logical name of the field, e.g., **`getVolume`** (1)
  - What should the complete signature of the get methods be?
  - Refer to the `AlarmClock` class code for an example

Writing a Test Class

Lab

Tasks to Perform

- Create a new class, `TelevisionTest`
  - Include a **`main`** method in `TelevisionTest`
  - Remember - you can check off the box to create a main method in the Eclipse new class wizard
- In `TelevisionTest.main()` instantiate two `Televisions`
  - You'll want `Television` references also, of course

```
Television tv1 = (what goes here?)
```

  - Use `System.out.println` to print out the values for each television instance
  - Use the get methods to access the properties (see notes)

Running the Program

Lab

Tasks to Perform

- **Run the program** as you did in the earlier lab (see notes)
  - You have two classes now, `Television` and `TelevisionTest`
  - Which class should be your class to run the program?
  - **Hint**: Use the one with the `main` method in it
  - Is the output what you expect?

- Next, add initializers in `Television` for brand and volume

```
String brand = "Sony";
int volume = 17;
```

  - Run the program again. What do you see now?

Additional Things to Do

Lab

Tasks to Perform

- Try accessing the instance variables directly instead of using the get methods
  - e.g. try something like `tv1.brand` instead of `tv1.getBrand()`
  - This will work – for now …
  - Leave at least one access like this, as we'll use it to illustrate encapsulation in the next lab

- Eclipse includes several features to help in writing code
  - Some of these can be accessed via the Edit menu and hotkeys
  - Some features are only accessible by right clicking in the editor window
  - Look at the following slide, and experiment with some of the features for a few minutes

Handy Editor Features

Lab

- Some of the more common features include
  - Undo (C-Z), Redo (C-Y), Cut (C-X), Copy (C-C), Paste (C-V), Select All (C-A]), etc.
  - Content Assistance (*C-spacebar*)
  - Code Formatting (*Select code*, *right-click, Source* | *Format*)
  - Jump to a Line (*C-L*)
  - Find/Replace (*C-F*)
  - Add bookmark (*Edit menu or right-click on source line*)
  - Add task or breakpoint (*right-click in left margin*)
  - Searching: (*C-H*)
  - Note that C- is an abbreviation for pressing the Ctrl key along with the associated key

LEARNINGPATTERNS

Lab 4.1: Encapsulation

In this lab, we will encapsulate the data in Television

Lab Synopsis

- **Overview**: In this lab, we'll encapsulate the `Television` data
  - By making the instance variables **`private`**
  - We'll use **`public`** getter and setter methods to access/modify their values
  - We'll define a **`toString()`** method for `Television`
    - It will describe a `Television` instance in `String` form
  - We'll also explore some of the other Eclipse views

- **Builds on previous labs:** Lab 3.3
  - Continue to work in the **Lab03.3** project

- **Approximate Time**: 20-30 minutes

Encapsulation Example

Lab

- This example code shows encapsulation at work
  - This is **not the lab solution** – but shows helpful examples

```
public class AlarmClock {
  // Make the data private for encapsulation
  private int snoozeInterval = 0;

 // Make the methods public for access
  public void setSnoozeInterval(int snoozeIn) {
    snoozeInterval = snoozeIn;
  }
  public int getSnoozeInterval() {
    return snoozeInterval;   // Returns the instance var
  }
}
```

```
public class EarlyMorning {
  public static void main(String[] args) {
    AlarmClock myClock = new AlarmClock();
    myClock.setSnoozeInterval(10);   // OK: Method accessible
    myClock.snoozeInterval = 10;   // ERROR: not accessible
  }
}
```

Encapsulating Data

Lab

Tasks to Perform

- Encapsulate the data in the `Television` class by **making the instance variables `private`**
  - i.e., prefix each field declaration with the `private` modifier
- **Make your getter methods public** with the `public` modifier
- Make sure you're still making a direct access to a field in your `TelevisionTest` program
  - e.g. using **`tv1.volume`** to access the volume directly
  - Make a trivial change to *TelevisionTest.java*, then save to recompile it,
  - What error messages does the compiler generate?
  - Why are you getting them?
  - Remove the direct field access in `TelevisionTest`

Complete the Accessor methods

Lab

Tasks to Perform

- Create mutator (setter) methods for each data element
  - The method names should start with **`set`**, and be followed by the logical name of the variable, e.g., **`setVolume`**
  - Method names should follow the same conventions as variables
    - i.e., first "word" in lower case, each subsequent "word" capitalized, e.g., `setVolume`, `getVolume`, `setColorTint`, etc.
  - For the method parameters of the setter methods, you may wish to use some other naming convention, like volumeIn, brandIn etc.
- What should the complete signature of the set methods be? *

Modifying TelevisionTest

Lab

Tasks to Perform

- Change `TelevisionTest` to only use the accessor methods to access television data
  - Use the **set** methods to initialize each `Television`'s data elements
  - Use the **get** methods to retrieve each `Television`'s data elements, and print them out as before, using `System.out.println`
- **Run** your program and view the output to see that it works

```
Television tv1 = new Television();
tv1.setBrand("RCA");
tv1.setVolume(10);
System.out.println("Television: brand=" + tv1.getBrand() +
  ", volume=" + tv1.getVolume());
```

Writing a toString Method

Lab

Tasks to Perform

- Create a **toString** method in class Television
  - It should return a String that includes the name of the class and the values of the instance variables
  - The signature of the method must be:

```
public String toString() { … }
```

  - Consider formatting the returned String, so it looks like this:

```
Television: brand=Sony, volume=6
```

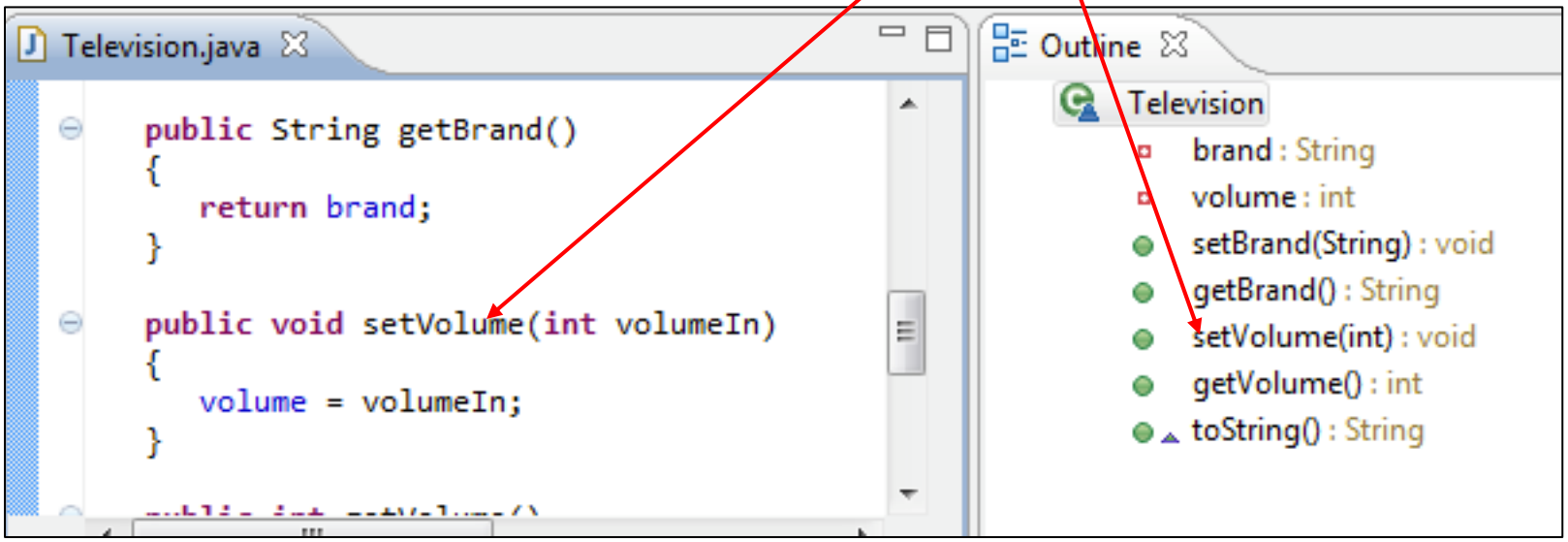
  - In TelevisionTest use this toString method to display each Television object's data
- **Run** your program and view the output

```
// for a Television reference, tv1:
System.out.println(tv1.toString());
System.out.println(tv1);  // Same result as above
```

[Optional] Review Eclipse Outline View

Lab

- **Open the Outline View (*Window | Show View | Outline*)**
  - It is an overview of the key elements that make up the resource that is being edited
  - Allows quick and easy navigation through your resource using a tree model for organizing related elements
  - Clicking in the outline view will bring you to the equivalent location in the editor
  - Try opening it and moving around *Television.java* with it

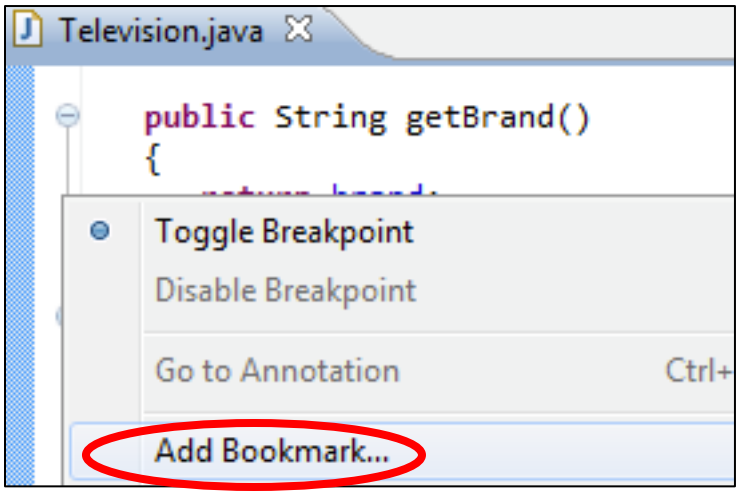


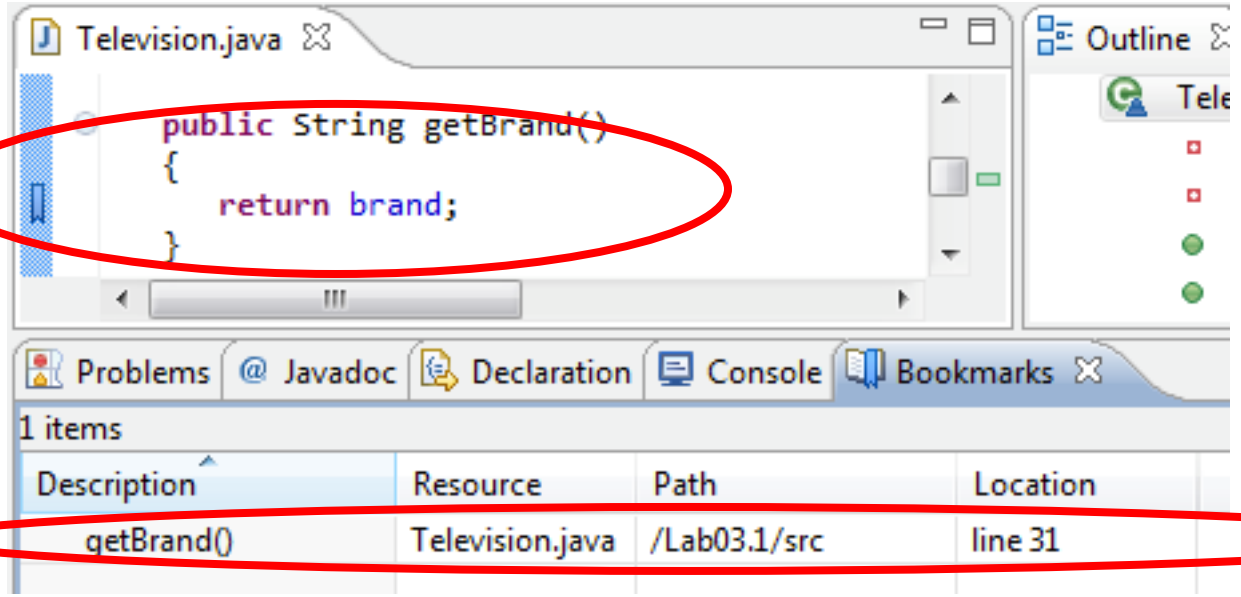
[Optional] Review Bookmarks View

- Helps organize your development environment and process
- Add bookmarks in one of two ways
  - The edit menu (**Edit | Add Bookmark …**)
  - **Right-clicking** in an editor's left-hand margin
- The bookmark view lets you jump to any resource with a bookmark, even a specific line of bookmarked code
- **Open the Bookmark view** and see how it works
  - ***Window | Show View | Other | General | Bookmarks***
  - Try adding a bookmark, then going to a bookmark using the Bookmark view *
  - See following slide

[Optional] Eclipse Bookmarks

Lab





STOP

LEARNINGPATTERNS

Lab 4.2: Adding Constructors to a Class

In this lab, we will add constructors to `Television`

Lab Synopsis

- **Overview**: In this lab, we will add constructors to `Television`
  - We will create both default (no-arg) constructors, and detailed constructors with multiple arguments
  - We'll use them in a program when creating televisions

- **Builds on previous labs**: Lab 4.1
  - Continue to work in the **Lab03.3** project

- **Approximate Time**: 20-30 minutes

Sample Code

Lab

- This example code shows constructors being used
  - This is **not** the lab solution – but shows relevant examples

```
public class AlarmClock {
  private int snoozeInterval = 0;

  // No-arg constructor creates AlarmClock with default snooze
  public AlarmClock() { }

  // Create an AlarmClock with the specified snooze time
  public AlarmClock(int snoozeIn) {
    setSnoozeInterval(snoozeIn);  // Why do it this way?
  }
  // …
}
```

```
// create an AlarmClock with no-arg (default), constructor
AlarmClock a1 = new AlarmClock();

// create a AlarmClock with constructor that takes an int
AlarmClock a2 = new AlarmClock(1500);
```

Sample Code

Lab

- One constructor can invoke another constructor
  - You must do it in the first statement
  - It is invoked by using **`this()`**, with a list of arguments in the parentheses

```
public class AlarmClock {
  private int  snoozeInterval = 0;
  private long currentTime = 0;

  public AlarmClock(int snoozeIn) {
    // invoke other constructor explicitly
    this(snoozeIn, 0);
  }

  public AlarmClock(int snoozeIn, long currentIn) {
    setSnoozeInterval(snoozeIn);
    setCurrentTime(currentIn);
  }
}
```

Add Detailed Constructor

Lab

Tasks to Perform

- Create a constructor to to set both fields of `Television`
  - We'll call this the "detailed" constructor
  - It should take two arguments – one for volume, one for brand
    - Use the arguments to initialize the fields
    - What should the signature of the constructor be?
  - Refer to the `AlarmClock` class samples for an example

- **NOTE**: Use the setter methods in your detailed constructor
  - Don't assign to the fields directly
  - Why is this important?

Add No-Arg Constructor

Lab

Tasks to Perform

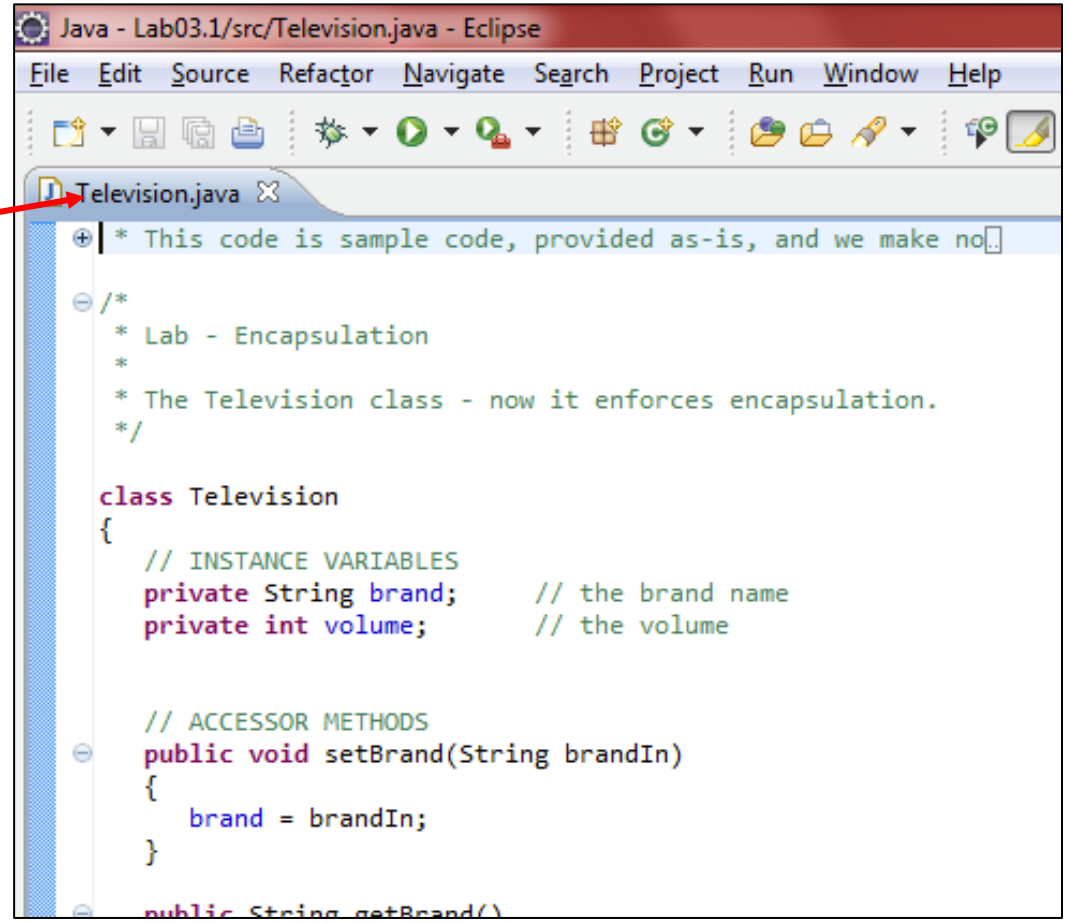
- Create another constructor that takes no arguments
  - The **default** or **no-argument** constructor
  - It should call the detailed constructor using `this()`
  - You can simply hardcode default values for brand and volume, e.g., `this("RCA", 10)`
- In `TelevisionTest` create a valid `Television` using each constructor
  - Print out some information from each television instance
- **Run** your program and view the output

```
Television tv1 = new Television();
Television tv2 = new Television("Hitachi", 17);
```

Eclipse - Maximize a View or Editor

Lab

- You can maximize a view by double clicking in its tab
  - Try it with the *Television.java* file in the editor
  - This is nice, as it lets you easily view a larger piece of a file
  - You can minimize it by double clicking again



LEARNINGPATTERNS

[Optional] Lab 4.3: Using `static` Members

In this (optional) lab, we will add static variables and methods to `Television`

Lab Synopsis

- **Overview** : In this (optional) lab, we'll add static variables and methods to `Television` and use them in our program
  - The static methods will work with the static variables
  - Note: This lab is somewhat complex
  - It should only be done if the class is moving through the labs at a good pace
- **Objectives**: Create and use `static` variables and methods
- **Builds on previous labs**: Lab 4.2
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 20-30 minutes

Objectives

Lab

- We show examples of defining and using statics below

```
// the Math class has a number of static definitions
public final class Math {
  public static final double PI = 3.14159265358979323846;
  public static int max(int a, int b) {
    return (a >= b) ? a : b;
  }
  ...
}
```

```
class UsingStatics {
  public static void main(String[] args) {
    int i = 5;
    int j = 6;

    int maxValue = Math.max(i, j);
    System.out.println("Pi equals " + Math.PI);
  }
}
```

Declaring static Variables

Lab

Tasks to Perform

- Create static (class) constants for min and max volumes in class `Television`

```
public static final int MIN_VOLUME = 0;
public static final int MAX_VOLUME = 100;
```

- It is standard to name these kinds of constants in all caps
  - Access these outside the class using the following expressions

```
Television.MIN_VOLUME and Television.MAX_VOLUME
```

  - From within the class, you don't need the Television. prefix
    - But it's standard practice to use it anyway, to clarify that this is a static constant

Use `static` Methods

Lab

Tasks to Perform

- Change the setter method for volume to make sure the volume stays between the min and max values (your static constants)
  - If an incoming value is below the min, give it the min value
  - If an incoming value is greater than the max, give it the max value
- You can use the static `Math.min()` and `Math.max()` methods to implement this
  - Look at the javadocs to see the full signature of these methods (the Math class is in the `java.lang` package)
  - See the notes for the logic
- **See notes** for accessing the javadoc online

Testing

Lab

Tasks to Perform

- In `TelevisionTest`, try setting the volume of a television instance to a negative value, and also to something greater than 100
  - Test both `setVolume()` and a constructor with these values
  - Since our constructors are calling setter methods to set the data, an invalid volume should be rejected in both tests
- **Run** your program and view the output
  - The volume should not go outside the allowed range
  - No matter what value you pass in
- **Optional**: Add declarations for default brands and volume
  - Static final constants similar to max/min volume
  - It is better to use these than "magic numbers" scattered in code

LEARNINGPATTERNS

Lab 4.4: Thinking About enums (Discussion-Only Lab)

In this lab, we will discuss a `Television` implementation that uses an enum for volume

Lab Synopsis

- **Overview** : In this lab, we consider an alternate implementation that uses an enum for a television's volume
  - We'll discuss the details of how the Television implementation changes
  - We'll discuss how to implement various capabilities, and how they change when using an enum for volume
  - We'll gain a better understanding of enum types

- **Builds on previous labs**: Lab 4.2

- **Approximate Time**: 20-30 minutes

Use an enum for Volume

- Assume that we have an enum Volume as defined at bottom

Tasks to Perform

- As a class, discuss the changes you would need to make in order to use the `Volume` enum as the volume for a television
  - What would the field declaration to store the volume look like?
  - What would the getter/setter look like?
  - Do the constructors need to change?
  - If you output a volume in `toString()`, what would the volume output look like?
  - What do you need to do to validate a volume value in `setVolume()`?
    - To make sure the incoming value is a legal value
  - What would you need to do to add another legal volume value

```
public enum Volume {
  OFF, VERY_SOFT, SOFT, MEDUIM, LOUD, VERY_LOUD, MAX
}
```

LEARNINGPATTERNS

Lab 4.5: Debugging

In this lab, we will learn about and use the debugging capabilities of Eclipse

Lab Synopsis

Lab

- **Overview** : In this lab, we will learn about and use the debugging capabilities of Eclipse
  - We'll look at the following slides on Eclipse debugging, and try out the various capabilities, including:
  - Open the Debug Perspective, Explore the views provided in the Debug Perspective, Set breakpoints in your current project, Run the program from the debug perspective, Examine and change variables, (Optional) Use conditions and hit counts

- **Builds on previous labs**: Lab 4.3
  - Continue to work in the **Lab03.3** project for this lab

- **Approximate Time**: 20-30 minutes

The Eclipse Debugger

Lab

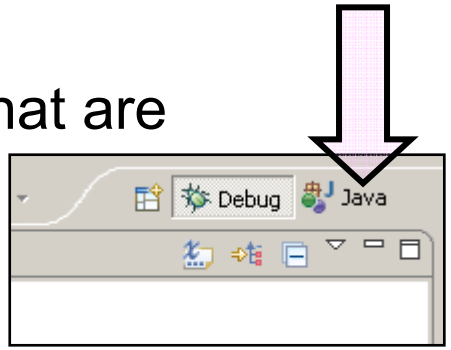
- Eclipse has support for
  - Setting breakpoints (and suspending execution)
  - Using conditions with breakpoints
  - Inspecting expressions
  - Viewing processes
  - View variables of processes
  - Displaying the thread call stack
  - Viewing the console
  - And more…

Debug Perspective

Lab

Tasks to Perform

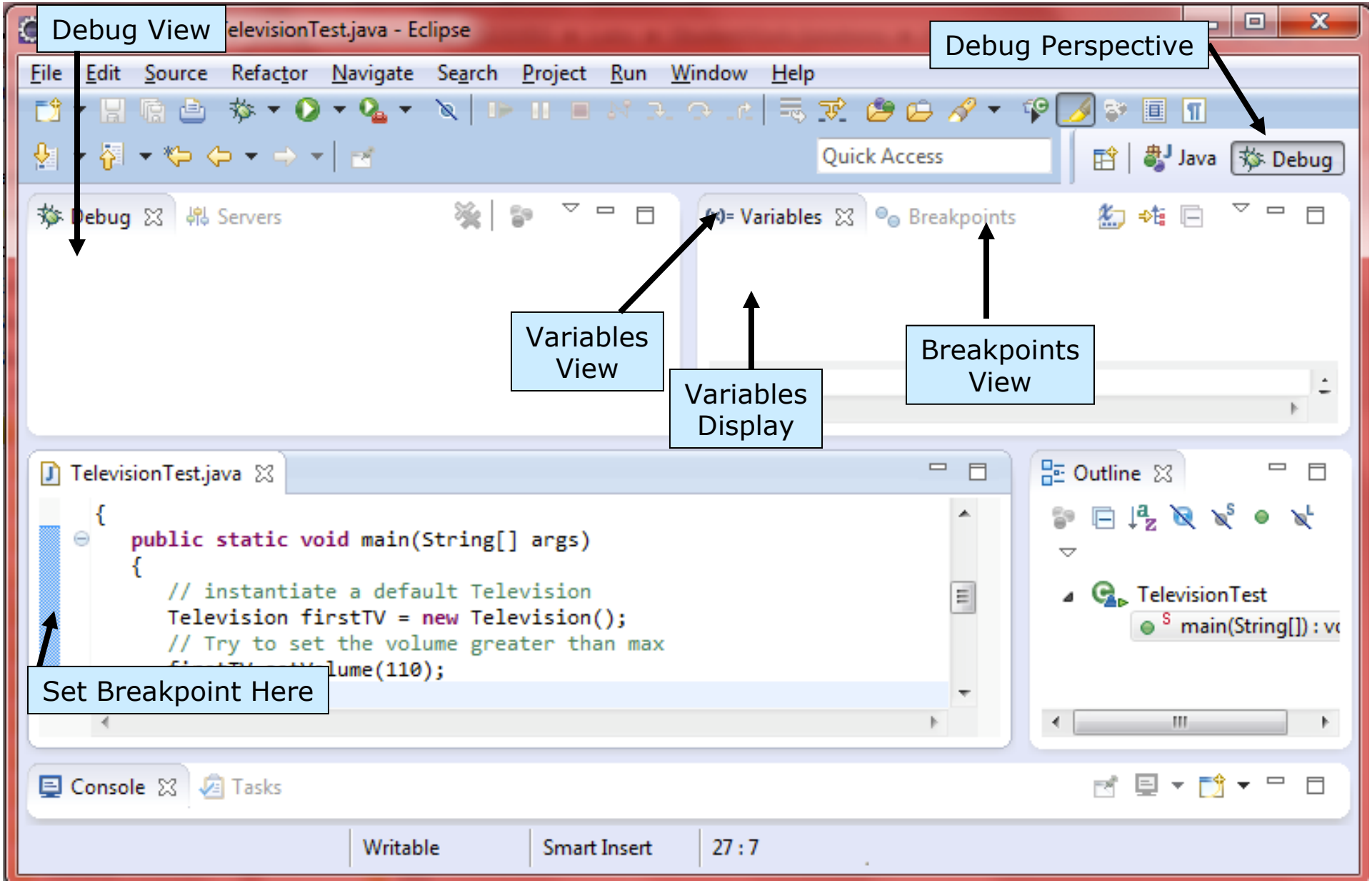
- Open the **Debug Perspective** via the menu selection:
  Window | Open Perspective | Debug
  - Look at the different views in this perspective that are described below and pictured on the next slide
  - You can switch back to the Java Perspective by selecting the icon as shown at right *



- The Debug Perspective contains views related to debugging
  - **Debug View**: Allows managing the debugging of a program
    - Including selecting stack frames, running, stepping, etc.
  - **Variables View**: Displays information about variables in currently selected stack frame
  - **Breakpoints View**: Lists all breakpoints
  - **Expressions View**: Used to inspect data

The Debug Perspective

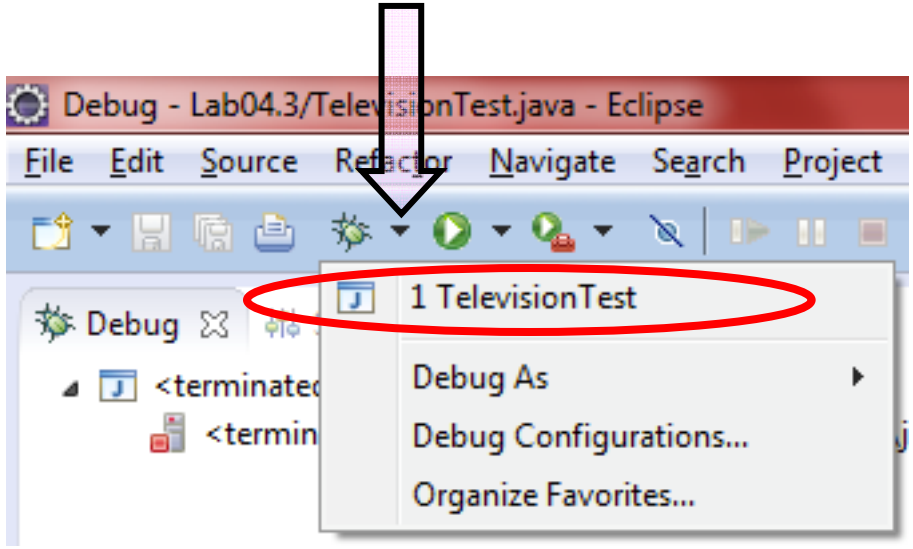
Lab



Lab

Tasks to Perform

- You can debug a program from the "Bug" toolbar button
  - Click the **Black arrow next to the Bug icon, select TelevisionTest**
  - This uses a a launch configuration in the same as running an application normally
  - The program will run to completion since no breakpoints are set



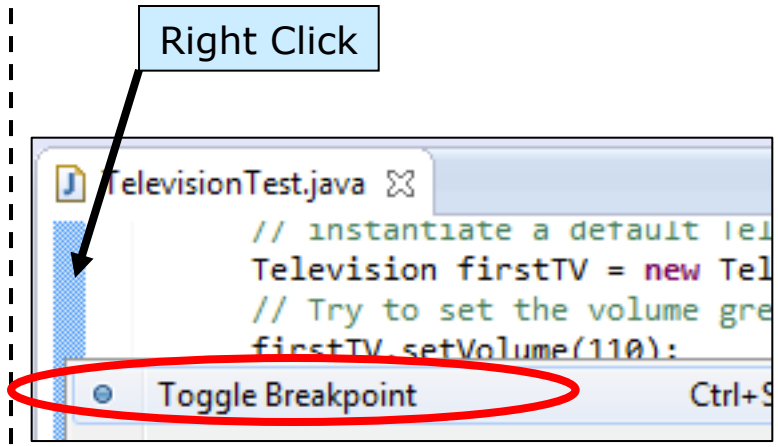
Setting Breakpoints

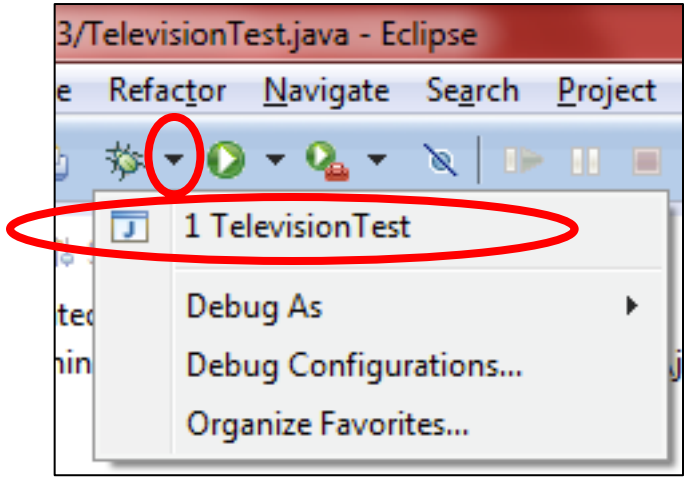
Lab

Tasks to Perform

- Breakpoints let you halt program execution at specific points
- In Debug Perspective, open `TelevisionTest` and set a breakpoint as follows:
  - Right click in margin and select

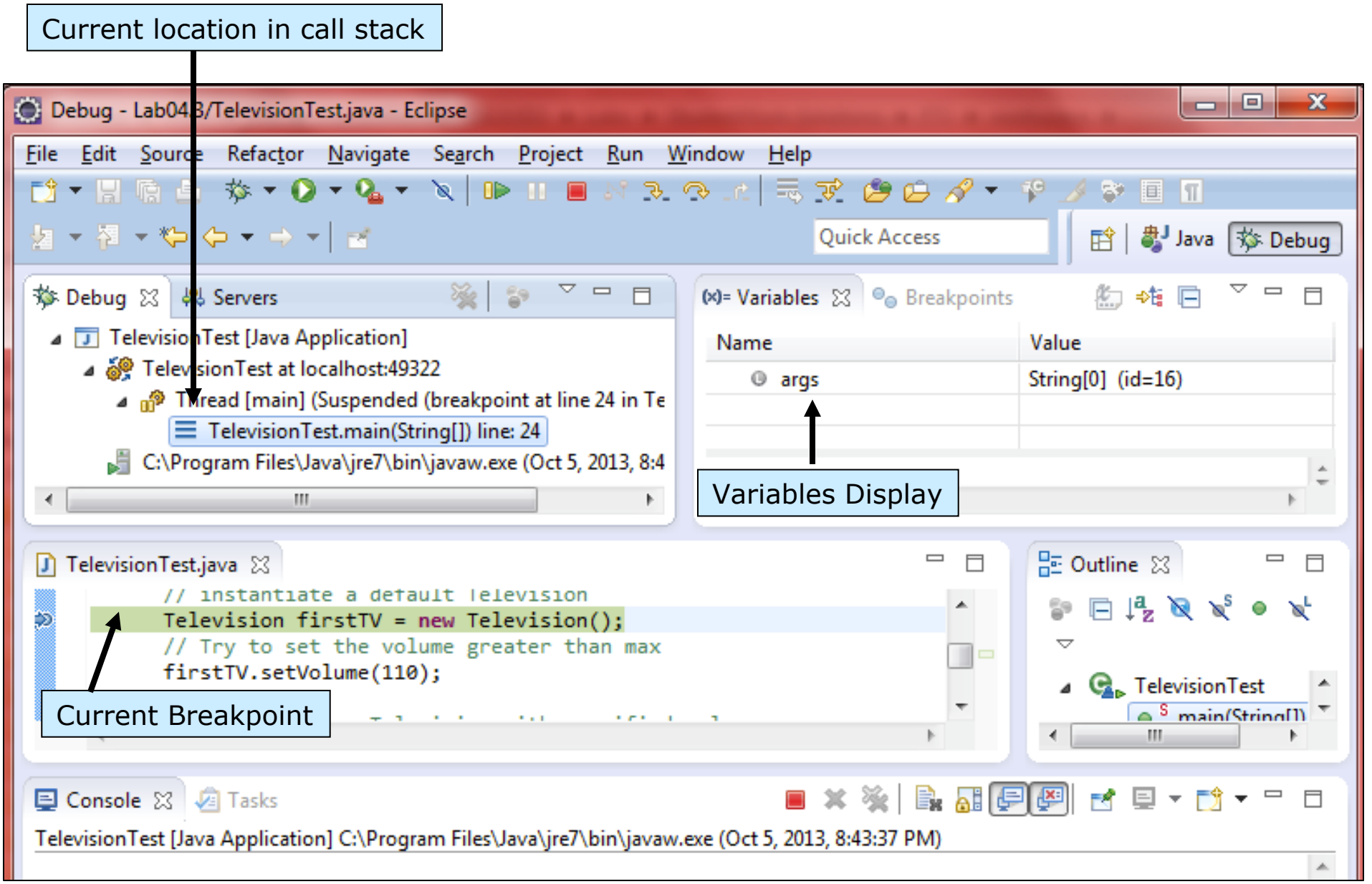
    Toggle breakpoint
- Start debugging `TelevisionTest`
  - Click the small black arrow next to the "Bug" icon and select `TelevisionTest`
- We'll work with various debugging capabilities in the next few slides





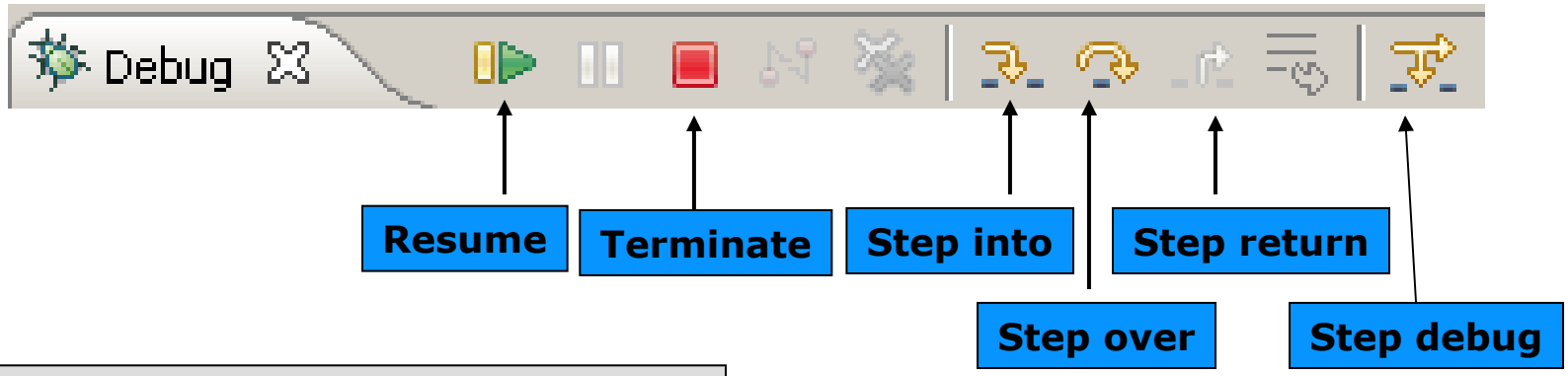
The Debug Perspective at a Breakpoint

Lab



Working with Breakpoints

Lab



Debug Controls allow:

- **Step into**

Next executable line for this thread

- **Step over**

Move to next method in stack

- **Step return**

Pause at next 'return'

- **Step debug**

Move into code containing debug info

- At a breakpoint, Eclipse provides options on how to continue running a program
  - The different controls are shown in the image above

Work with the Debugger Controls

Tasks to Perform

- Work with different controls available to continue while at a breakpoint (`Resume`, `Step into`, `Step over` ...)
  - We'll see how they work, and what the different views show while you're stepping through the program
- Note: When choosing **`Step into`**, you may step into code in the the Java runtime library (for example, the `ClassLoader loadClass()` method, or the `Object` constructor)
  - If this happens, just press the **`Step return`** icon to complete the system code and then continue debugging
- Work with these controls, and with the different capabilities of the debugger described on the next view slides

Variables View, Enable/Disable

- When a thread suspends, the top stack frame of the thread is selected
  - i.e. the place where the breakpoint occurred
- The **Variables View** displays the visible variables in that stack frame
  - If you select another stack frame in the Debug view, its variables will be displayed
- Disabling a breakpoint temporarily halts breaking on it
  - In the editor, right click on the breakpoint and select **Disable/Enable Breakpoint**
  - Can also be done in the Breakpoints view
  - You can also completely remove a breakpoint, which deletes it

Breaking on Exceptions

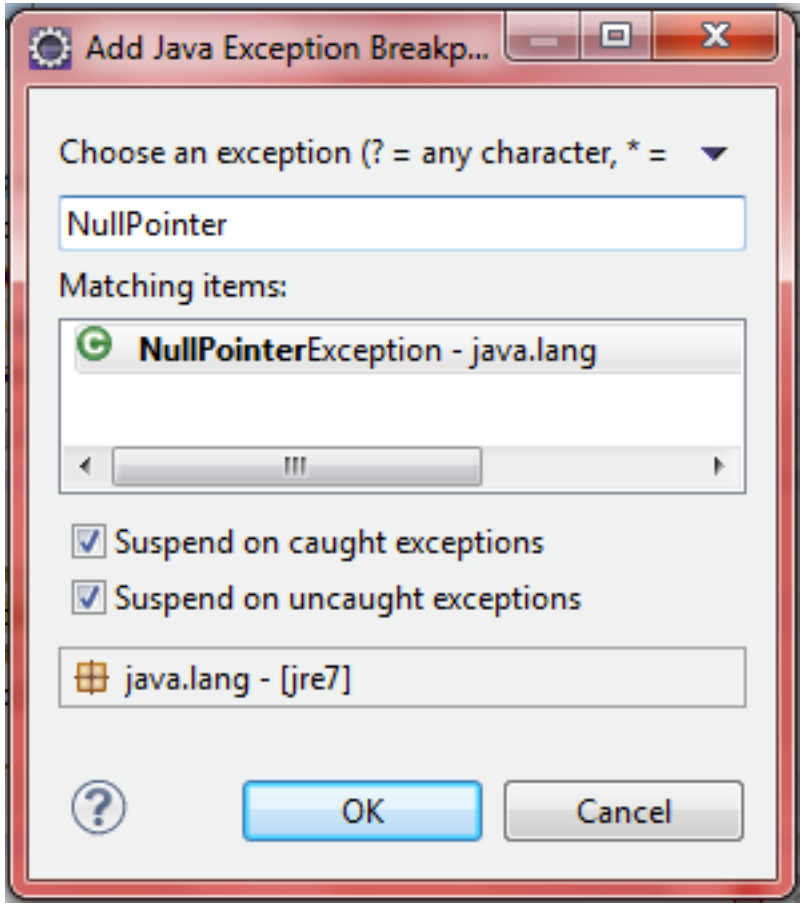
Lab

- Eclipse allows for breaking on exceptions
  - Exceptions are covered later in the course
- Can be done in a Debug or a Java perspective as follows
  - Navigate to the menu item:

Run → Add Java Exception Breakpoint

  - Type exception class name
    - e.g. `NullPointerException`
  - You can try this by setting a television ref to null and using it

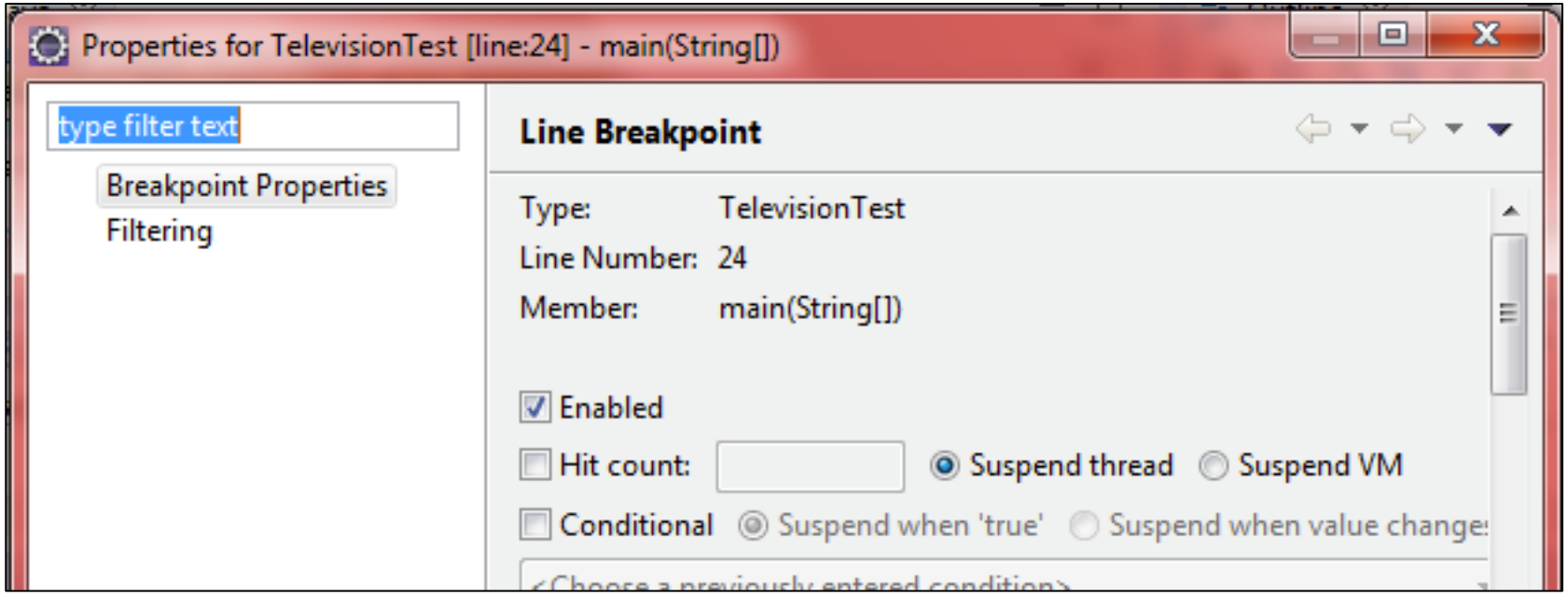
```
Television secondTV = null;
String s = secondTV.getBrand();
```



Breakpoint Properties

Lab

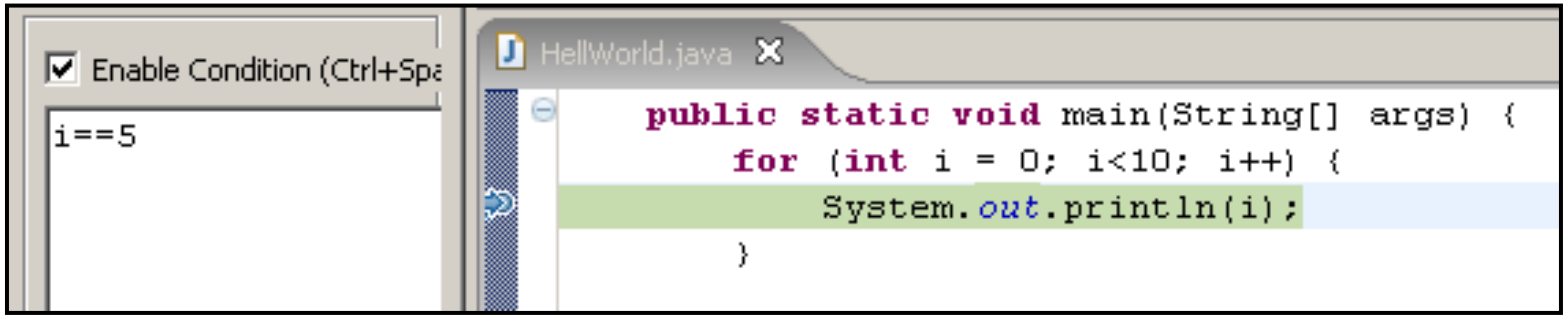
- Eclipse lets you control a breakpoint's properties
  - e.g. to set conditions on the breakponit
- Done as follows
  - Find the line of code with the breakpoint
  - Right-click on the breakpoint and select: ***Breakpoint Properties…***



Conditional Breakpoints

Lab

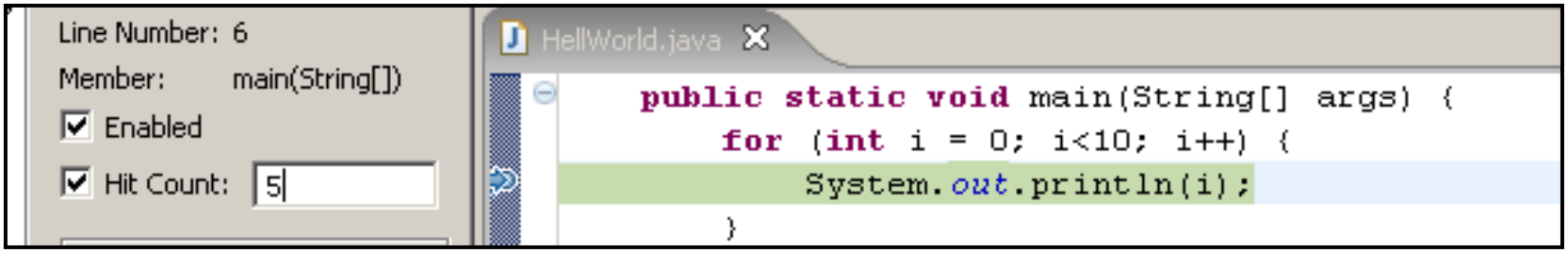
- If you set a **condition on a breakpoint**, it only breaks when the condition is met.
  - Below, you'll only break when i is equal to 5
  - Try this in your program
  - Conditions can be more complicated, e.g. `i==(res.size()-2)` will break on the next to last iteration over some collection res



Hit Counts

Lab

- **Hit counts** modify breakpoints so they only break once when the hit count is reached
  - e.g. - if the hit count is 5, the breakpoint will only break the 5th time it is reached
  - Not before, not after, until you change or disable the hit count



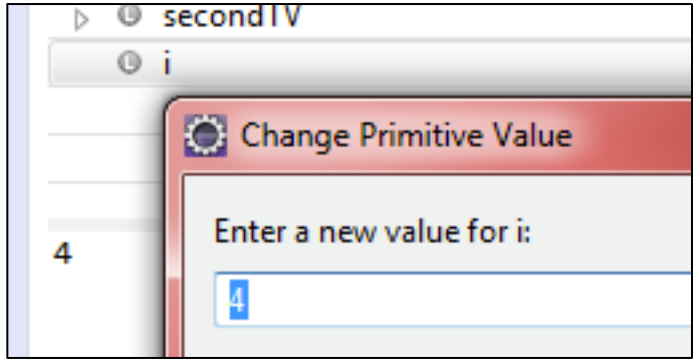
[Optional] Tasks to Perform

- Add a loop into the main method as shown above, and work with the hit count and conditional breakpoints

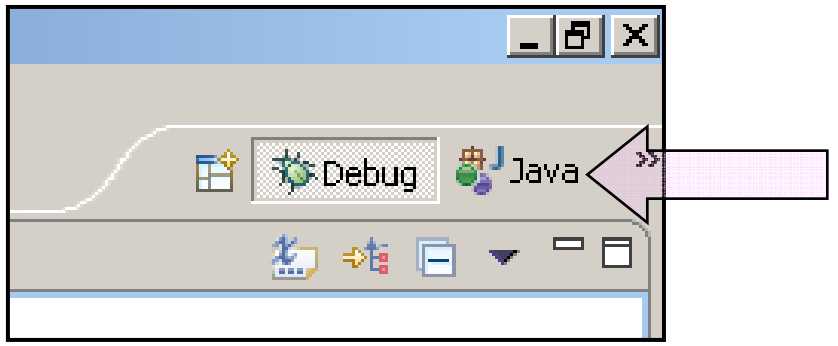
Changing Variable Values

Lab

- When execution is halted you can change the value of a thread's variables:
  - And see how the new value affects execution and results
- Done in the **Variables** tab as follows:
  - Right click on the variable (e.g. i) whose value you want to change and select **Change Value**
  - Set the variable value
  - Continue processing



  - When you're finished trying out the debugging, go back to the Java Perspective



STOP

LEARNINGPATTERNS

Lab 5.1: Data Validation

In this lab, we will add data validation to a class

Lab Synopsis

- **Overview**: In this lab, we'll become familiar with and use conditional statements to add validation to the setter methods
  - We'll (optionally) also add a `mute` method to `Television`, which is more complex than the programming we've done until now

- **Builds on previous labs**: Lab 4.2 (or Lab 4.3 if you did it)
  - Continue to work in the **Lab03.3** project

- **Approximate Time**: 20-30 minutes (45 min with optional part)

Validating Data in Setter Methods

Lab

Tasks to Perform

- Using `if` statements, write data validation routines in the brand setter method
  - In the `setBrand` method, make sure the brand is either "Sony", "Zenith", "Hitachi", or "RCA"
  - If the incoming value is not legal, then don't make any change
- **NOTE**: see notes about using the **`equals`** method, not ==, to do string comparisons
- Below is sample code using an if statement

```
String s = // Initialized to some string
if (s.equals("Hello")) {
  System.out.println("Hello to you also");
} else
  System.out.println("What's happening?");
}
```

Testing Your Validation

Lab

Tasks to Perform

- In `TelevisionTest`, set the brand to an invalid value
  - Try it via the setter method `setBrand("whatever")` and via the constructor `new Television("whatever", 44)`
- **Run** your program and view the output
  - You should always have valid values for the brand
- **Optional**: Add code to the `Television` detailed constructor to check that the brand is not null before it exits
  - If it is. set it to a default
    - This possible if someone passed in an invalid brand
  - Declare a `final static` variable for the default brand
    - Don't just using a "magic string" in the constructor
    - This is similar to the declarations in the optional static members lab

[Optional] Implement Muting

- Add muting behavior to the Television class
  - **Muting** a television sets the volume to zero
  - **Unmuting** returns the volume to what it was before muting
- Note that this section is fairly complex, and should **only be done** by classes **moving quickly** through the labs
- At a high level, you will:
  - Write a **`mute`** method to change the muted/unmuted state of a `Television` and alter the volume appropriately
  - The `mute` method should toggle muting between on and off
    - Each time you call `mute`, the mute state changes
  - **To mute**, save the television's current volume setting, change the current volume to zero, and set a flag to note that it's muted
  - **To unmute**, restore the television's current volume to the saved old volume and reset the flag to note that it's not muted

[Optional] Implement Muting

Lab

[Optional] Tasks to Perform

- Create fields to hold muting state:

```
private boolean isMuted;     // mute flag
private int     oldVolume;   // old volume
```

- Add a getter method for muted state

```
public boolean isMuted() { }
```

  –NOTE: `boolean` variables and methods that return `boolean`s are often named in the form of a question, i.e., `isSomething`?

- NOTE: what is the default value of a `boolean` instance variable if it does not have an initial value?

[Optional] Implement Muting

Lab

[Optional] Tasks to Perform

- Implement the `mute` method

```
public void mute() { }
```

  - It should execute the following logic:
    - If `isMuted` is `false`, save `volume` in `oldVolume`, change `volume` to zero and set `isMuted` to `true`
    - If `isMuted` is `true`, restore `volume` from `oldVolume` and set `isMuted` to `false`
  - You can see an example in the notes below for more help

- Modify the `toString` method to indicate a volume value of `<muted>`, not 0, if the `Television` is in the muted state

```
Television: brand=Sony, volume=<muted>
```

[Optional] Test Muting Behavior

[Optional] Tasks to Perform

- In `TelevisionTest`, test your `mute` method
  - Below is an example of how to do it:
- **Run** your program and view the output - does it work?

```
// Television tv has been created and has some volume
// print its status
System.out.println(tv.toString());

// mute it
tv.mute();

// print its status again – is it muted?

// unmute it
tv.mute();

// Print its status again
// Is the old volume restored?
```

LEARNINGPATTERNS

Lab 6.1: Arrays

In this lab, we will practice using arrays

Lab Synopsis

- **Overview**: In this lab, we will practice creating and using arrays
  - We'll use the `args` array in `main` to retrieve command line arguments
  - We will create and work with an array of a class type (`Television`)
- **Builds on previous labs**: Lab 5.1
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 30-40 minutes
- Below is a sample showing array iteration (using `for-each`) *

```
public static void main(String[] args) {
  for (String s : args) {
    System.out.println(s);
  }
}
```

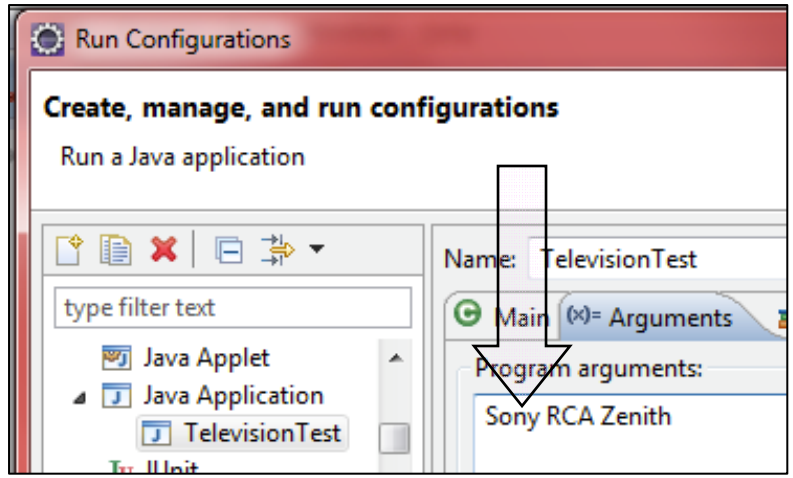
Using the args Array

Lab

- In this lab, you'll pass in brand arguments to your program
  - (e.g. `Sony RCA Zenith`)
  - Eclipse uses launch configurations to set program arguments, as in the instructions below

Tasks to Perform

- Open the launch config (1), and type in arguments (brand names) in the **Program Arguments** tab for the `TelevisionTest` configuration



- In your main method, create an **array** of `Television`
  - How do you know how big the `Television` array needs to be? *

```
Television[] tvArray = // what goes here?
```

Arrays of a Class Type

Lab

Tasks to Perform

- For each argument, create a `Television` instance using that argument as the brand value to the detailed constructor
  - Just set the volume to 5
  - Add each television instance into the array of `Television`

```
int i = 0;  // See note
for (String s : args) {
    tvArray[i++] = // What goes here?
}
```

- Iterate over the array and print out the data values for each instance
  - Use a **`for-each`** to go through the array
- **Run** your program and view the output

[Optional] Pass in Volume Values

Lab

[Optional] Tasks to Perform

- First get the lab working with hard coded volumes
  - Once that is done, you can try passing in volume args
- Pass in pairs of brand-volume arguments - e.g.:

```
>java TelevisionTest RCA 10 Zenith 20
```

- Use the volume arguments when creating your televisions
- Recall that the arguments will be strings (not integers)
  - `Integer.parseInt(String s)` is a `static` method that returns an `int`, given the string representation of an integer.

```
int seven = Integer.parseInt("7");
```

[Optional] Arrays

Lab

[Optional] Tasks to Perform

- In `Television`, initialize a static array of allowable brands using the shortcut notation

```
public static final String[] VALID_BRANDS =
  {"Sony", "Zenith", "Hitachi", "RCA"};
```

  - Use it for brand validation – only brands in `VALID_BRANDS` are ok
  - In the `setBrand` method, check if the brand passed in is one of those in `VALID_BRANDS` (see notes)

- Test that your validation with the `VALID_BRANDS` array is working
  - Try valid and invalid values for the brand, e.g.,
    `Television tv = new Television("whatever", 10);`

LEARNINGPATTERNS

Lab 7.1: Packages

In this lab, we will practice using packages

Lab Synopsis

- **Overview**: In this lab, you will learn how to put your classes in packages, and how to use classes that are in packages
  - You will also become more familiar with how `public`, `private`, and "`package-private`" (default) access works

- **Builds on previous labs**: Lab 6.1
  - Continue to work in the **Lab03.3** project

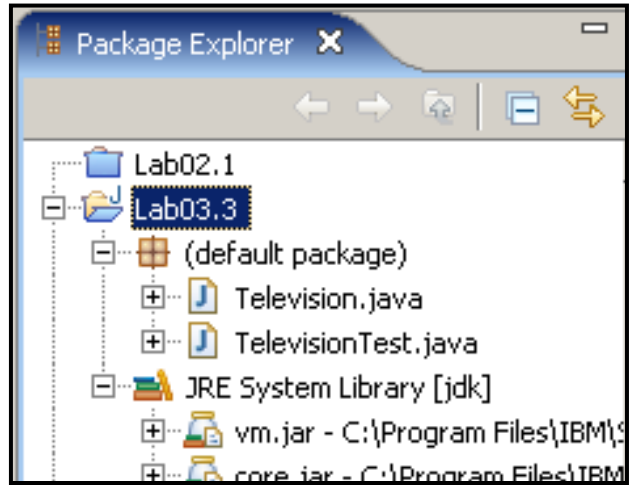
- **Approximate Time**: 25-35 minutes

Writing Classes in Packages

Lab

- We'll put class `Television` class into a package **`com.entertainment`**
  - Its source/class files have to be in a folder ***com\entertainment***
  - `TelevisionTest` will go into **`com.entertainment.test`**

- We will use the Eclipse **Package Explorer** view for this
  - It is a Java specific view including imports, classes, methods
  - Source folders and referenced jars are shown
  - Lets you modify package structure



Tasks to Perform

- Go to **Package Explorer** view *

Writing Classes in Packages

Lab

Tasks to Perform

- Create a **`com.entertainment`** package in your project
  - Right click on Lab03.3, select **New | Package**
  - Name the package **`com.entertainment`** and click **Finish**
- Move Television to the `com.entertainment` package
  - Right click on *Television.java*, select **Refactor | Move ...**
  - In the dialog, select the package `com.entertaiment`
- Next, create a **`com.entertainment.test`** package
  - Move `TelevisionTest` into this package
  - Use the instructions above as a guide
- Eclipse will change your Java code appropriately
  - Let's examine those changes

Resulting Code

Lab

Tasks to Perform

- Look at the changes from putting `Television` into the `com.entertainment` package
  - A **`package`** statement was added to the top of the source file
  - You need to declare the class **`public`** (Why?)
  - The methods (and constructors!) also need to be **`public`**. Why?
- Look at the changes to use `Television` in `TelevisionTest`
  - Look at the **`import`** statement added in *TelevisionTest.java*
  - Eclipse makes these kinds of changes very easy, as it propagates changes in a type to all uses of that type
- You may see errors in `TelevisionTest` (1)
  - Fix any errors, and **`run`** the program

LEARNINGPATTERNS

[Optional] Lab 8.1: Composition

In this lab, we will practice using composition

[Optional] Lab Synopsis

- **Overview**: In this lab, you'll practice building classes by **composing** them from other classes
  - We'll add television functionality that is handled by another class that we supply
  - We'll modify `Television` to use composition and **delegate** its new functionality to this new class
- Note: This lab is somewhat complex, and, though valuable, it does take some time, and may be skipped
- **Builds on previous labs**: Lab 7.1
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 30-40 minutes

Example of Composition (Not the Solution)

Lab

```
public class Engine {
  public void start() { /* ... */ }
  public void rev() { /* ... */ }
}
```

```
public class Transmission {
  public void shiftTo(int gear) { /* ... */ }
  public void engage() { /* ... */ }
}
```

```
public class Car {
  private Engine       engine = null;  // Car is composed of other objects
  private Transmission tranny = null;

  public Car(Engine eng, Transmission trans) {
    engine = eng;      // construct a Car with Engine and Transmission
    tranny = trans;
  }
  public void moveTo(String destination) {
    // delegate work to Engine and Transmission
    engine.start();
    engine.rev();
    tranny.shiftTo(1);
    tranny.engage();
  }
}
```

A Support Class

Lab

- We provide a **Tuner** class to handle tuning the television to a channel (passed as a string)
  - This is functionality that can easily be separated from the class using it
  - It is also something we might want to reuse
- `Tuner` has the following public methods:

```
public void setChannel(String channelIn)
public String getChannel()
```

  - These methods output some text to indicate they were called
    - They don't do anything else
  - `Tuner` has some private internal methods that it uses

Examine the Support Class

Lab

Tasks to Perform

- Find the **`com.entertainment.Tuner`** source file in the **LabSetup** project in the folder ***Lab08.1-optional***
  - Review: why is this located in a *com\entertainment* folder?
  - Copy the source file into your Java project's *com.entertainment* package, with those of `Television`
    - Right click on *Tuner.java* in Eclipse, select **Copy**, then paste it into the Lab03.3 project's `com.entertainment` package
- Open *Tuner.java* and review the source code
  - It has the two public methods described earlier
  - It has some private methods used internally
  - Note: We don't try to have real functionality in the class
- You will now **modify `Television`** to use a **`Tuner`**

Use Composition in Television

Lab

Tasks to Perform

- Modify `Television` to use a `Tuner` instance as follows:
  - Add a field to `Television` to hold a `Tuner`
  - Add the following two methods to `Television`

```
public void gotoChannel(String channelIn)
public String getCurrentChannel()
```

  - Have them delegate to the tuner's `setChannel`/`getChannel` methods
    - i.e. call those methods through the tuner instance
  - Consider any issues with packages (1)

```
// only the relevant portions are shown here
public class Television {
  private Tuner tuner = new Tuner();  // The tuner

   public void gotoChannel(String channelIn) {
     // Delegate to the tuner instance
   }
   public void getCurrentChannel() { /* Delegate to the tuner */ }
}
```

Use Composition in `Television`

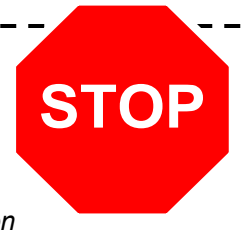
Lab

Tasks to Perform

- Modify **`TelevisionTest`** to use tuner related functionality
  - e.g., Create a television, set/get its channel to see that it works
  - The Tuner methods will output some information to the console
  - You should see this output if you delegated correctly

- **Run your program** – it should show output from the tuner
  - Because it's handling some of the functionality of changing the channel
  - You've split the functionality between the Television and the Tuner
  - This is one of the foundations of OO programming (**Divide and Conquer**)

- Note that the provided solutions include the Tuner functionality in this lab only (1)



LEARNINGPATTERNS

Lab 8.2: Inheritance

In this lab, we will practice using inheritance

Lab Synopsis

- **Overview**: In this lab, you will learn to use inheritance to specialize classes by creating subclasses of `Television`
- **Builds on previous labs**: Lab 7.1 (or 8.1 if you did it)
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 20-30 minutes

Example Code – Not the Solution

Lab

```
public class Timepiece {
  private Date currentTime;

 public Timepiece(Date d) {
    currentTime = d;
  }

  public void displayCurrentTime() {
    System.out.println(currentTime);
    // Give subclasses a chance to do some extra display
    displayExtra();  // Polymorphism!  See note
  }

}
```

```
public class AlarmClock extends Timepiece {
  int snoozeInterval;  // get/set methods not shown

  public AlarmClock(Date d) {
    super(d); // pass d to superclass constructor
  }
}
```

Creating Subclasses

Lab

Tasks to Perform

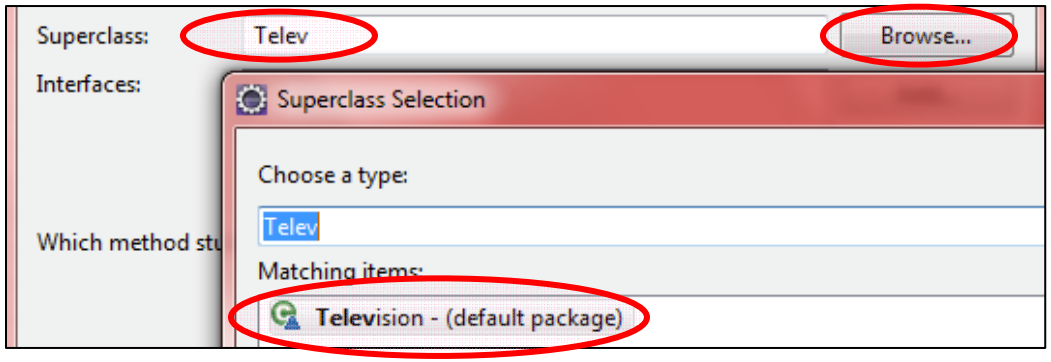
- Create subclasses of `Television` with different characteristics
  - Create two subclasses of `Television`: **`ColorTelevision`**, and **`PortableTelevision`**
  - Put them in the `com.entertainment` package – what directory will they be located in?
  - **You can specify both the superclass and package** in the Eclipse wizard to create a class *
- Create the following fields for these classes along with get/set methods (see notes and next slide for Eclipse shortcuts)

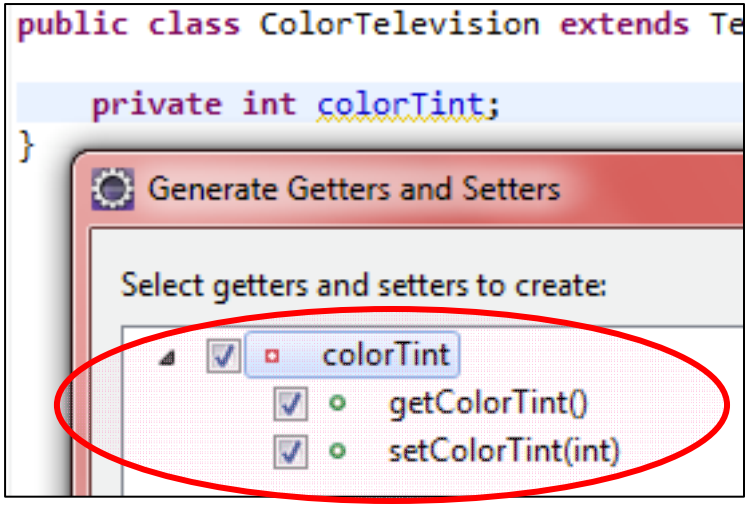
```
ColorTelevision -    int colorTint
PortableTelevision - int rechargeLevel
```

Eclipse Shortcuts

Lab

- In the **New Java Class** dialog, you can start typing in the name `Television` in the Superclass box, then hit browse
  - Eclipse will find the `com.entertainment.Television` class for you, and you can just click it to use it as the superclass (image on left below)
- Eclipse also has wizards to help you add accessor methods
  - Add in the `colorTint` variable declaration in `ColorTelevision`
  - Once done, right click in the editor, and select **Source | Generate Getters and Setters** (image on right below)





Test the Subclasses

Tasks to Perform

- In `TelevisionTest`, create instances of each subclass
  - Access a property inherited from `Television` (e.g. the volume) and output it
  - Access a property in the subclass itself (e.g. a color tint) and output it
- **Run** your program and see that it works

LEARNINGPATTERNS

Lab 8.3: Polymorphism

In this lab, we will practice using polymorphism

Lab Synopsis

- **Overview**: In this lab you'll override inherited methods in your subclasses
  - You will see polymorphism at work and gain a better understanding of it
- **Builds on previous labs**: Lab 8.2
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 20-30 minutes

Overriding Methods

Lab

Tasks to Perform

- Override **`toString()`** in each `Television` subclass
  - To override, it must have the same signature as the standard `toString` method
  - Label the new `toString()` methods with **`@Override`**
- `toString()` should return a string including all of `Television.toString()`'s information
  - It should print additional information appropriate to the subclass
    - e.g., `ColorTelevision` should report the color tint
- Leverage `Television`'s `toString` method
  - Since it already handles brand and volume,
  - You can do this by calling `super.toString` (see notes)

Using Polymorphism

Lab

- We'll now see polymorphism working by using an array
  - We'll put instances of the different Television classes in an array
  - We'll iterate through the array and call `toString` on each object
  - The appropriate `toString()` will be called on each instance
  - Even though the reference (in the array) is of type `Television`

Tasks to Perform

- In `TelevisionTest`, create a `Television` array of size 3

```
Television[] tvArray = new Television[3];
```

- Add an instance of `Television` and of each `Television` subclass to the array
  - See notes for the shortcut notation to initialize an array
  - Why can you add a `ColorTelevision` to a `Television` array?

Using Polymorphism

Lab

Tasks to Perform

- Iterate over the array and print out each instance
  - Use `toString()` and `System.out.println()`
  - The correct `toString` is called for each instance (via polymorphism)
- **Run** the program and view the output
- **OPTIONAL**: create subclass constructors which call the superclass's constructor with `super()`

```
public ColorTelevision(String brand, int volume, int color)
{
  // pass brand and volume to superclass constructor
  super(brand, volume);

  // deal with color tint here
  this.setColorTint(color);
}
```

LEARNINGPATTERNS

Lab 9.1: Interfaces

In this lab, we will work with interfaces - both creating and using them

Lab Synopsis

- **Overview**: In this lab, you will work with interfaces to gain a better understanding of them
  - You will create and implement an interface
  - You will use a supplied class (`Radio`) that implements the same interface as `Television`, but is otherwise unrelated
  - You'll see how you can access and treat both types as instances of the interface type
- **Builds on previous labs**: Lab 8.2
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 25-35 minutes

Example Code (Not the Solution)

Lab

- Below, and on the following slide, are examples of code using interfaces
  - This is **not** the lab solution – but shows relevant examples

```
public interface Moveable { // A generic movable
  public void moveTo(String dest);
}
```

```
public class PosterTube implements Moveable {
  public void moveTo(String dest) {
    // ...
  }
}
```

```
class ShippingBox implements Moveable   { /* ... */ }
class WardrobeBox extends ShippingBox { /* ... */ }
```

Example Code (Not the Solution)

Lab

```
class MovingCompany {

  Moveable[] goods = null;  // Moveable is an interface type

  MovingCompany(Moveable[] goodsIn) {
    goods = goodsIn;
  }
  void deliverAllGoods(String location) {
    for (Moveable m : goods) {
        m.moveTo(location);
    }
  }
}
```

```
class GetMoving {
  public static void main(String[] args)  {
    Moveable[] items = { new PosterTube(),
           new ShippingBox(),  new WardrobeBox() };

    MovingCompany acme = new MovingCompany(items);
    acme.deliverAllGoods("San Francisco");
  }
}
```

Creating an Interface

Lab

- First, we'll create an interface describing volume functionality
  - Consider this functionality as a "role" description for controlling volume

Tasks to Perform

- Create an interface named **Volume** with the following abstract methods:

```
public void setVolume(int volumeIn);
public int getVolume();
public void mute();          // Only if mute lab done *
public boolean isMuted(); // Only if mute lab done *
```

  - Right click on the project and select **New | Interface**
  - Put it in the `com.entertainment` package
    - What directory must it be located in?

Class Radio Implements Volume

Lab

Tasks to Perform

- Find the supplied **`com.entertainment.Radio`** source file in the LabSetup project under ***Lab09.1***
  - Copy the source files into your `com.entertainment` package, with those of `Television`
  - Do this all within Eclipse as previously
- Modify class `Radio` so it implements the **`Volume`** interface
  - It won't compile after you implement the interface - notice the error message
  - Uncomment `Radio`'s `getVolume()` method so it compiles cleanly *
  - Look over the `Radio` source code - it handles volume-related functionality the same way that `Television` currently does

Implementing an Interface

Lab

Tasks to Perform

- Implement the `Volume` interface in class `Television`
  - Since `Television` already has the `Volume` methods you don't need to add any methods to it *
- In `main()`, change the `Television` array to a `Volume` array of size 4

```
Volume[] volArray = new Volume[4];
```

- Create a `Radio` instance and add it into the array, in addition to the televisions already in the array
  - In the `for` loop, call some `Volume` methods on each element before it gets printed, e.g., mute all the elements in the array, double the volume of each element, etc.
- **Run** your program and see what happens

LEARNINGPATTERNS

[Optional] Lab 9.2: Default Method

In this lab, we will add a default method to an interface and see how it works

Lab Synopsis

- **Overview**: In this lab, you will add a default method to the `Volume` interface
  - You'll first add it as a standard method, without changing the implementing classes, and see how they don't compile
  - You'll then change it to a default method, and see how that works
- **Builds on previous labs**: Lab 9.1
  - Continue to work in the **Lab03.3** project
- **Approximate Time**: 15-20 minutes

Add a Default Method

Lab

Tasks to Perform

- Open interface `Volume` for editing
  - Add a regular method to it with the signature below, and save it

```
public void silence()
```

  - Look in the Problems view - note the errors in `Television` and `Radio`
  - They don't implement this method

- Change `silence()` to a default method
  - In it's body, set the volume to zero via `setVolume()` and save the file
    - This is a simpler form of muting - we don't save the old volume (1)
  - Your errors should now disappear
  - In your test method, in the loop that works with `Volume` objects, call `silence()` before printing out the objects
  - Run the program - all your instances should have zero volume

Summary

Lab

- You first added an additional method to the `Volume` interface
  - And saw how existing classes had errors in them
  - This is exactly the situation that default methods are designed for
- Once you added the method as a default method, then your errors disappeared
  - And you could call the default method on an instance of your classes - even though you did not change them
  - Again, this is exactly what default methods are meant for

LEARNINGPATTERNS

Lab 10.1: Using Exceptions

In this lab, we will work with exceptions - both throwing and catching them

Lab Synopsis

Lab

- **Overview**: In this lab, you will learn how to use exceptions, including learning to:
  - Throw an exception when an error condition occurs
  - Handle an exception when an error condition occurs
  - You'll also gain a better understanding of checked and unchecked exceptions

- **Builds on previous labs**: Lab 9.1
  - Continue to work in the **Lab03.3** project

- **Approximate Time**: 30-40 minutes

Sample Code

- In this lab, we'll use exceptions to indicate that an invalid brand has been passed into a `Television`
  - You're already checking if the brand is valid in `setBrand()`
  - You'll now add code to **throw an exception** if it's invalid

- We'll use the basic **`Exception`** class to represent our error
  - The base exception is suitable for our needs

- The code in the following slide shows an example of using exceptions
  - This is **not** the lab solution – but shows relevant examples

Sample Code (Not the Solution)

Lab

```
public class AlarmClock {  // Much detail omitted …
  public void setSnoozeInterval(int snoozeIn)
                throws InvalidSnoozeIntervalException {
    if (snoozeIn >=0 && snoozeIn <= 100) {
       // set snooze interval as appropriate
    } else {
      throw new InvalidSnoozeIntervalException("snoozeInteval " +
                snoozeIn  + " is invalid");
    }
  }
}
```

```
public class Worker {
  public void wakeUp(AlarmClock c) {
    try {
      c.setSnoozeInterval(10);
    }
    catch (InvalidSnoozeIntervalException e) {
        String msg = e.getMessage();
        System.out.println(msg);
    }
  }
}
```

Throwing an Exception

Lab

Tasks to Perform

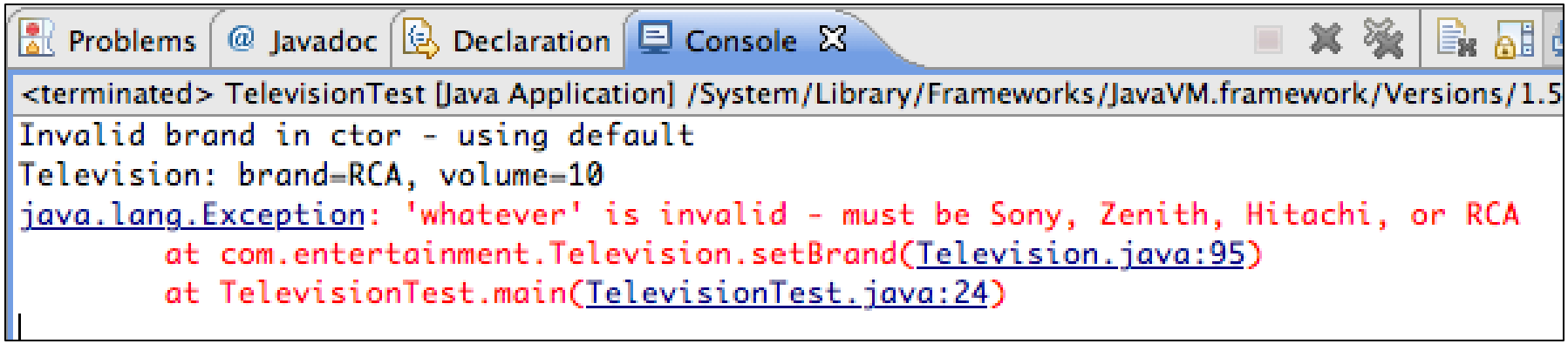
- In class `Television`:
  - Modify `setBrand()` to throw an `Exception` on an invalid brand
  - Update the method signature with a `throws` clause
  - Modify any `Television` constructors calling `setBrand()`
  - Have them catch the exception, and set the brand to the default (e.g. `brand = DEFAULT_BRAND;` (1))
    - Are you required to catch the exception here? Why or why not?
    - Does your code compile without it? What are your choices?
- Try creating a `Television` in `TelevisionTest` using any of the constructors - Does it compile and run ?
  - Try it with an invalid brand
  - Does it compile? What happens when you run it?

Catching an Exception

Lab

Tasks to Perform

- Next, call `setBrand()` on a `Television` instance in `main()`
  - Try it with an invalid brand to see what happens
  - We'll need `try` and `catch` blocks (1)
  - Add a `try-catch` block around the call to `setBrand()` in `main()`
    - Include a call to the `printStackTrace` method in the `catch` block
- **Run** the program and view the output



Optional Ideas (Time Permitting)

[Optional] Tasks to Perform

- Once the first part is all working correctly, here are some ideas for further exploration
- Try defining your own exception class **`IllegalBrandException`** that is a subclass of `Exception`
  - Have `setBrand()` throw an instance of this class instead of an instance of Exception
  - What else had to change in your code? Why?
- Try using an **`IllegalArgumentException`** (in the `java.lang` pacakge) instead of an `Exception`
  - This is a runtime exception
  - If you use this, do you still need a throws clause in `setBrand()`
  - Do you need try/catch blocks in your program?

LEARNINGPATTERNS

Lab 11.1: Using Collections

In this lab, we will introduce the usage of collections

Lab Synopsis

- **Overview**: In this lab, you will gain hands-on experience using the Java Collections Framework
  - After retrieving a collection of objects from a utility class, you will write code to iterate over the collection
  - You will also use many of the previous concepts you've learned – like polymorphism, looping constructs, and `static` methods

- **Builds on previous labs**: Lab 10.1
  - **Work back in your Lab03.3 project**

- **Approximate Time**: 25-35 minutes

A Catalog of Televisions

- We provide a **`Catalog`** class that contains an in-memory database of televisions
  - You will search the catalog for televisions with a specified brand
  - You will get your search results back as a `Collection`
  - You then need to iterate over the results and analyze them
- **Important Note**: The Catalog class expects that valid brands are one of: "**Sony**", "**Zenith**", "**Hitachi**", "**RCA**"
- The `Catalog` class has the following methods:

```
public static Collection<Television>
  searchByBrand(String brand)
public static Collection<Television> getInventory()
```

  - **NOTE**: you don't have to write these methods; they have been created for you in the `Catalog` class

Install and Use the Catalog

Lab

Tasks to Perform

- Copy the **com.entertainment.Catalog** source files from the ***LabSetup*** project's ***Lab12.1*** folder into your **Lab03.3** project
  - Into the `com.entertainment` package
  - See notes for compatibility requirements of `Television` (1)
- In `TelevisionTest.main()`:
  - Search the catalog by calling its `searchByBrand()` method
  - This is a `static` method, so you need not instantiate `Catalog`
  - Either hard-code the brand to search for, or use a command line argument (see note (2))
  - Iterate over the returned `Collection` (3) and print out the information in each `Television` (easy to do with `Television`'s `toString`)
- **Run** your program and view the output
- Test your polymorphism knowledge
  - What `toString()` method is called in the loop

[Optional] A Summary Report

Lab

[Optional] Tasks to Perform

- **OPTIONAL 1**: determine how many televisions of each brand are instances of `Television`, `ColorTelevision`, and `PortableTelevision`
  - Using the same search you just performed, you need to determine the type of each element in the collection
  - You can do this with the **`instanceof`** operator (see notes)
  - Above the iteration loop, declare 3 counter variables
  - As you iterate over the collection, determine each element's type and increment the appropriate counter
  - After the loop, report the results with `System.out.println` (see notes)

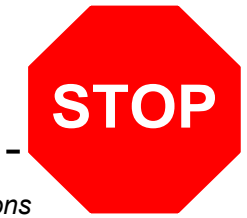
[Optional] Find the Loudest

[Optional] Tasks to Perform

- **OPTIONAL 2**: determine which television is the loudest
  - Get the entire catalog by calling `Catalog.getInventory`
  - Above the iteration loop, declare variables for maximum volume so far and for the television with that volume

```
int maxVolume = 0;
Television loudest = null;
```

  - As you iterate over the collection, compare the current television's volume to `maxVolume` and, if it's louder, replace `maxVolume` and `loudest` with the current volume and television
  - After the loop, report the result with `System.out.println` (see notes)



LEARNINGPATTERNS

Lab 11.2: Using Sets

In this lab, you will create and populate both sets and lists

Lab Synopsis

- **Overview**: In this lab, you will use a set, and see how its behavior differs from a list
  - You'll put some elements in a set, and see how it does not add duplicates
  - You'll also see how it filters out duplicates when created from another collection already containing duplicates
- **Builds on previous labs**: Lab 11.1
  - **Work back in your Lab03.3 project**
- **Approximate Time**: 15-20 minutes

Create and use a Set

Lab

Tasks to Perform

- In `TelevisionTest.main()` do the following
  - Create two television instances, and store them in variables
  - Create an `ArrayList<Television>`, add both television instances into the list **twice**, and print out the size of the list
  - Create a `HashSet<Television>`, add both instances into the set **twice**, and print out the size of the list
  - Are the results what you expect?
  - Create a second `HashSet<Television>`, passing the list you created in the first part of the lab into its constructor
    - This copies the elements of the list into the set
  - Print out the size of the list and the new set - is it what you expect?

LEARNINGPATTERNS

Lab 12.1: Mapping an Entity Class

In this lab, we will map a class to a database using JPA annotations - we won't access the DB yet

Lab Synopsis

- **Overview**: In this lab we will map a class - `MusicItem` - to the database using JPA
  - We will add JPA annotations to the `MusicItem` class
  - These will include annotations for the entity class itself, the primary key and for the properties
  - We will NOT run a program until the next lab

- **Builds on previous labs**: none
  - The new lab folder and project is **Lab12.1**

- **Approximate Time**: 30-40 minutes

Adding Classes to the Classpath

- To run the program, you need to have the **Derby JDBC driver** classes and the **JPA classes** in your classpath
  - These are packaged as jar files
    - A jar is a zip-format archive with some added capabilities
  - We include the jars in the lab setup
  - We use the open-source Hibernate implementation for JPA

- Each project has a Java build path describing its dependencies
  - i.e. the jar and class files it needs
  - We'll configure the project's build path to include the jar files we need
  - They'll be automatically included when we compile/run the project

Review - MusicItem Class / Item Table

Lab

```
// JPA details and other details omitted
public class MusicItem {
   private Long       id;
   private String     title;
   private String     artist;
   private BigDecimal price;

   public MusicItem() { /* … Detail omitted … */ }
}
```

```
CREATE TABLE Item
(
   ITEM_ID     BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY (START
                                          WITH 1, INCREMENT BY 1),
   Title       VARCHAR(40),
   Artist      VARCHAR(40),
   Price       DECIMAL(5,2),
   CONSTRAINT  PK_Item  PRIMARY KEY(ITEM_ID)
);
```

Lab Preparation

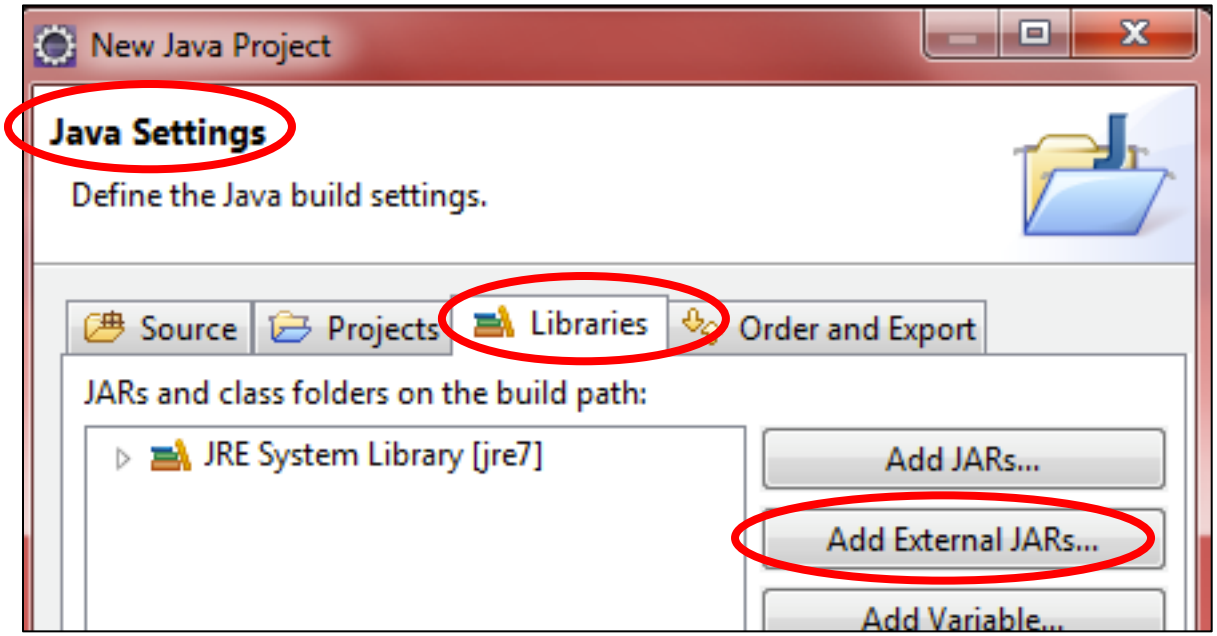
Lab

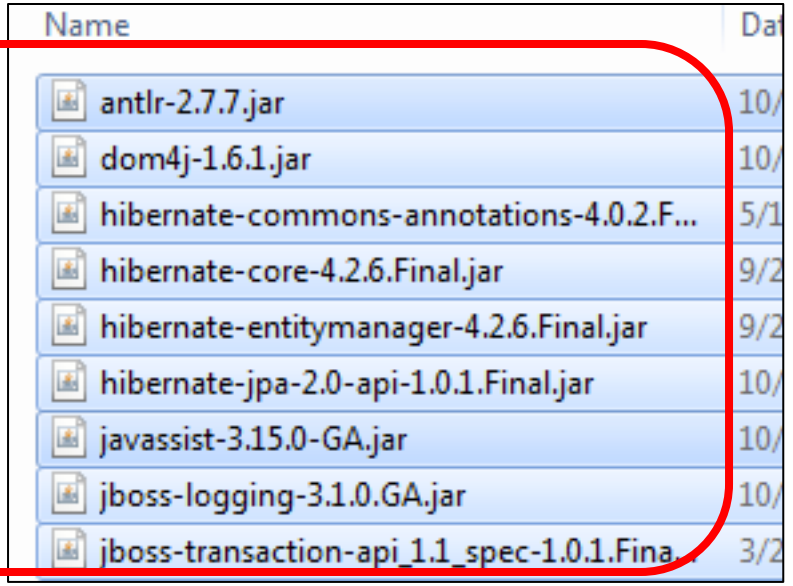
Tasks to Perform

- Close all files you have open for editing
- Create a new Java project called **Lab12.1** in your workspace
  - **File | New | Java Project**
  - Fill in the name as **Lab12.1**, and click **Next**
- In the next dialog, click the **Libraries** tab, and then click the **Add External JARs…** button (see next slide for screenshots) (1)
  - Browse to *StudentWork\FTJ**JPALib*** and add all the jars in that folder
- Click **Add External JARs…** again
  - Browse to *StudentWork\FTJ**DerbyLib*** and add the *derbyclient.jar* file
- Click **Finish** to complete the project creation

Adding Jars to Project

Lab





Finish MusicItem

Lab

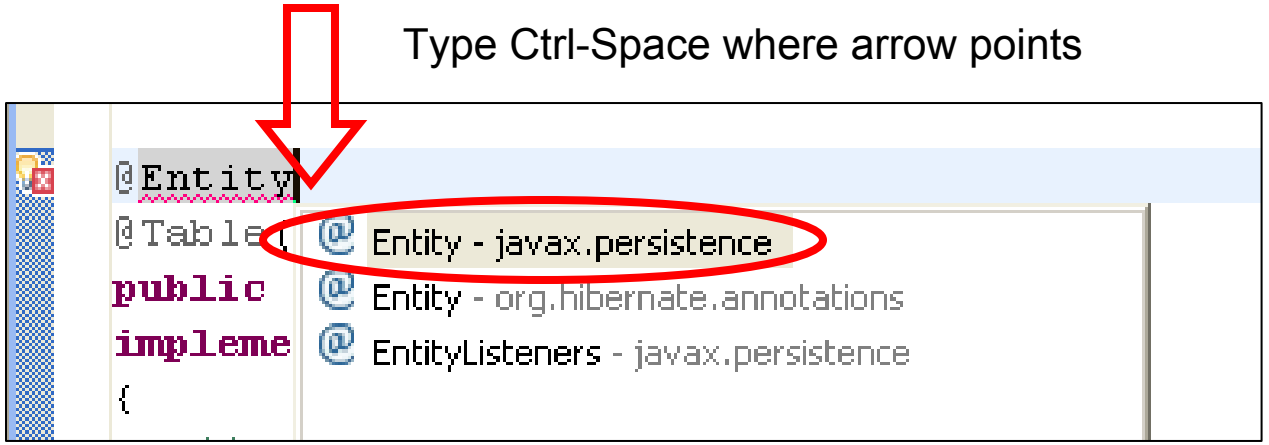
Tasks to Perform

- Open ***MusicItem.java*** and do the following:
  - Add annotations **to declare `MusicItem` an entity class**
    - You'll also need to **specify the table name**
  - You'll need to import the annotations (just like any type)
    - See the next slide for Eclipse help with importing
  - Find the annotation (already present) declaring the primary key (**`@Id`**)
    - Use **`@Column`** to specify the **correct column name** for this property
  - Review the other properties - Are the JPA defaults OK for them?
- Eclipse should **compile** the code for you when you save it
- Once the file compiles cleanly, **you're finished with this lab**
  - We'll use this class in the next lab

Eclipse Hint - Importing

Lab

- One of the nicest features of Eclipse is its auto-import feature
  - To try it out, in your *MusicItem.java* file, remove the import for `Entity` if you've already put it in (if you haven't imported it, then just continue below)
  - At the location where you're going to use the annotation, type `@Entity`, then type **Ctrl-Space**
  - Select the `javax.persistence` entry and hit return
  - An `import javax.persistence.Entity` statement will very conveniently be added near the top of your source file



LEARNINGPATTERNS

Lab 12.2: Using JPA

In this lab, we will use the JPA EntityManager

Lab Synopsis

- **Overview**: In this lab we will use the `EntityManager` to lookup `MusicItem` entities from data stored in a database
  - We provide a simple test program to do this
  - We will need to add JPA code into the main method
- **Builds on previous labs**: Lab 12.1
  - Continue working in your **Lab12.1** directory for this lab
- **Approximate Time**: 30-40 minutes

Database Notes

- We will be using the Apache open source Derby database
  - Originally Cloudscape, it was put into open source by IBM
- We supply scripts to start/stop the database server
  - As well as scripts to create the database, called JavaTunesDB, that we'll use in the labs
- You will need to configure *persistence.xml* to connect to this database
  - We supply all the information for that
  - JDBC URL, user name, password, etc.

Setup/Create the Derby Database

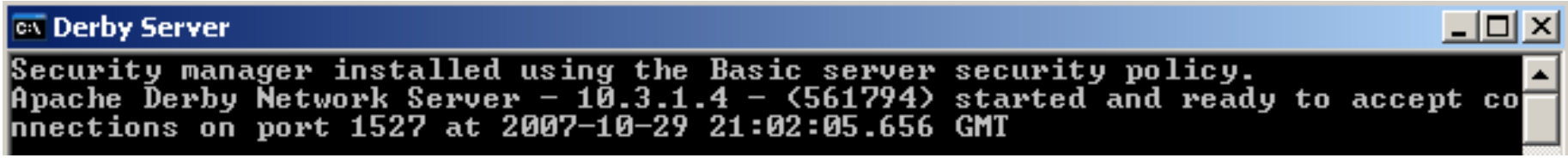
Lab

Tasks to Perform

- View the supplied Derby DB scripts in the folder:

  C:\StudentWork\FTJ\Derby

- Start the Derby database server (see note (1) for *nix)
  - Execute **`dbStart.cmd`** (you can double-click on it)
  - This starts a standalone Derby server that can accept network connections to the database
  - You should see output like that below in the window that opens



- Create the database
  - Execute **`dbCreate.cmd`** (you can double-click on it)
  - This creates the JavaTunesDB database
  - (you can ignore a DROP TABLE error, if you see it)

Using an SQL Command Tool

- **ij** is Derby's SQL command line tool
  - It allows us to create database objects and view and manipulate database data, without having to write code to do so
  - Most database packages provide such a tool

Tasks to Perform

- Execute **dbSQL.cmd** to run ij (you can double-click on it)
  - This command file is set up to connect to the correct DB
  - Using ij, you can browse the DB
  - Execute the query shown below to see the items in the DB

```
ij> select * from item;
    ... Lots of output ...
ij> exit;
```

persistence.xml

Lab

Tasks to Perform

- Open *persistence.xml* - located in the ***META-INF*** directory
  - Look for the TODOs
  - Set the name of the persistence unit to `javatunes`
  - Set the transaction type to `RESOURCE_LOCAL`
- Note the Hibernate specific properties in *persistence.xml*, e.g.

```
<property name="hibernate.dialect"
    value="org.hibernate.dialect.DerbyDialect"/>
```

  - This particular property tells Hibernate that we are using the Derby database
  - There are many other Hibernate properties
- Configure the required properties **as shown on the next slide**

Finish persistence.xml

Lab

Tasks to Perform

- Finish the properties that are circled below - look for the TODOs in your *persistence.xml*
  - This is typical of the configuration needed to connect to a DB

```
<persistence … namespaces not shown … >

 <persistence-unit name="javatunes"
                   transaction-type="RESOURCE_LOCAL">
  <properties>
  <!-- Database Connection Settings -->
   <property name="hibernate.connection.username">guest</property>
   <property name="hibernate.connection.password">password</property>
   <property name="hibernate.connection.url">
      jdbc:derby://localhost:1527/JavaTunesDB</property>
   <property name="hibernate.connection.driver_class">
      org.apache.derby.jdbc.ClientDriver</property>

  <!-- Other configuration not shown -->
  </properties>
 </persistence-unit>
</persistence>
```

Finish the Test Class

Lab

Tasks to Perform

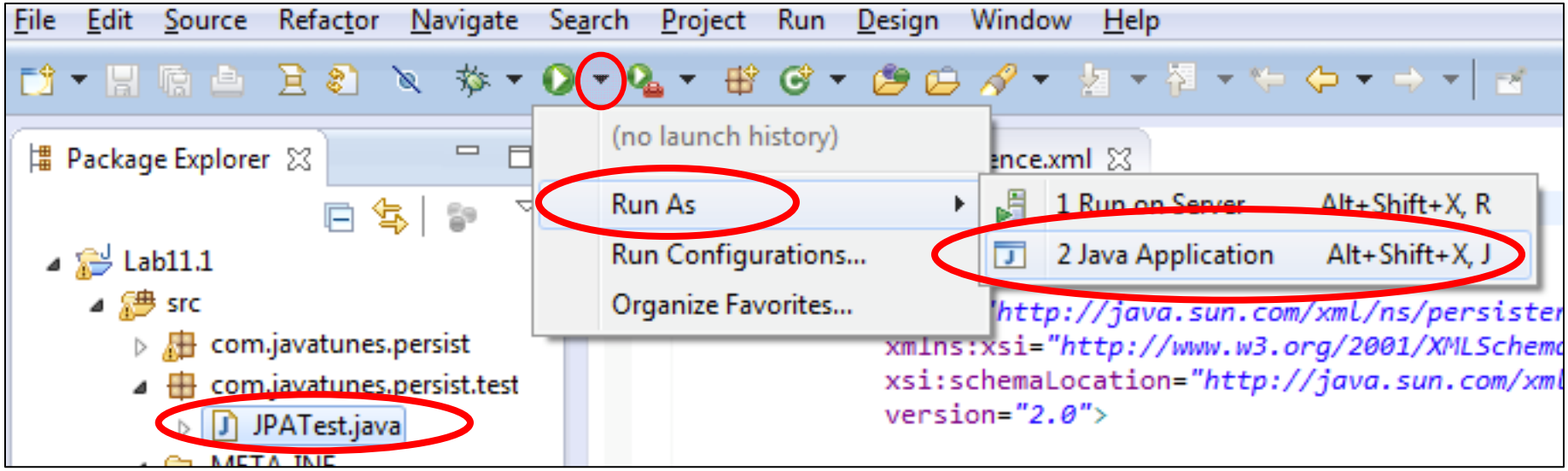
- Open *JPATest.java* (in `com.javatunes.persist.test`)
  - We'll write some fairly straightforward code to work with JPA
  - We'll add complexity later
- Do the following in `main()`
  - Create an `EntityManagerFactory` and `EntityManager`
    - Use the name of the persistence unit from *persistence.xml* (javatunes)
  - Use `EntityManager.find()` to get an instance of `MusicItem` using the id of 1
  - See the lecture manual slides for examples
  - Remember to add in any imports
  - There is already code to output the item you find

Running Your Application

Lab

Tasks to Perform

- After a clean build (one that is error-free, but not necessarily warning-free), test the application as follows
  - Select *JPATest.java* in the Navigator or Package Explorer view
  - Click the run button arrow on the task bar *
  - On the menu that appears, select **Run As | Java Application**

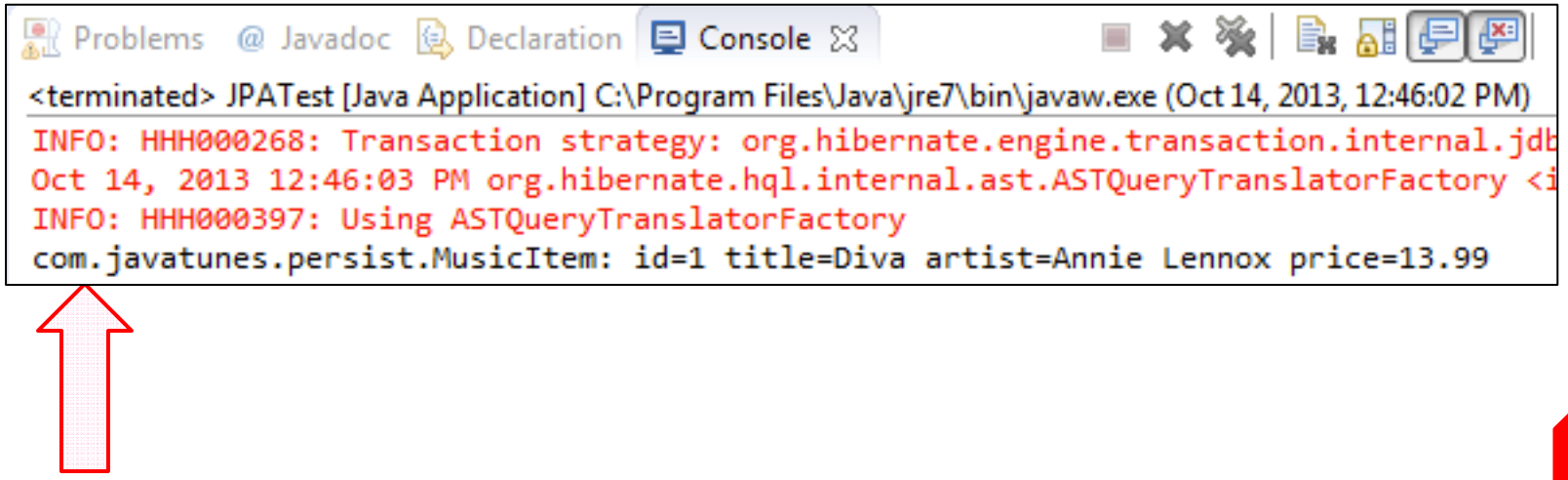


Viewing the Console

Lab

Tasks to Perform

- You will see the results in the console view as shown below
  - If necessary, open the Console (**Window | Show View | Console**)
  - There is usually quite a bit of logging output from Hibernate also
- That's it - you've retrieved an item from the database
  - JPA generated all the SQL and JDBC code that was needed



STOP

LEARNINGPATTERNS

Lab 12.3: Insert/Query Demo

In this lab, we will demonstrate some additional features of JPA

Lab Synopsis

- **Overview**: In this lab we will demonstrate inserting and querying using JPA
  - We provide all the code for this
- **Builds on previous labs**: Lab 12.2
  - Continue working in your **Lab12.1** directory for this lab
- **Approximate Time**: 10 minutes

Tasks to Perform

- Open *JPATest.java* for editing
  - Uncomment and review the code section labeled "for last JPA lab"
  - It does a query using JPQL, an insert, then the same query again
  - The second query shows the new item that was added in
- Run JPATest again, as before
  - Look in the console for the output - you should see the two query outputs

LEARNINGPATTERNS

[Optional] Lab 14.1: Formatted Output

In this lab, you will use the Java formatting capabilities

Lab Synopsis

- **Overview**: In this lab, you will do some simple formatting using the formatting capabilities of the `String` class
  - You will output both strings and numerical values
- **Builds on previous labs**: Previous Television labs
  - Work back in your **Lab03.3** project
- **Approximate Time**: 20-30 minutes

Use Formatting in Television

Tasks to Perform

- Modify `Television.toString()` to use `String.format()` to produce its output
  - Output the brand and the volume of the television
- Modify `toString()` of a subclass (e.g. `ColorTelevision`) to use `String.format()` also
- Modify `TelevisionTest` to create an instance of each, and print it out using `System.out.println()`
- **Run** your program
  - You should see the formatted output from your `toString()` method
  - Generally, this is much easier to do than concatenating strings
- Experiment with formatting other data types
  - Just do it in your main method using `printf()`
  - Look at the `java.util.Formatter` javadoc