itcourseware™

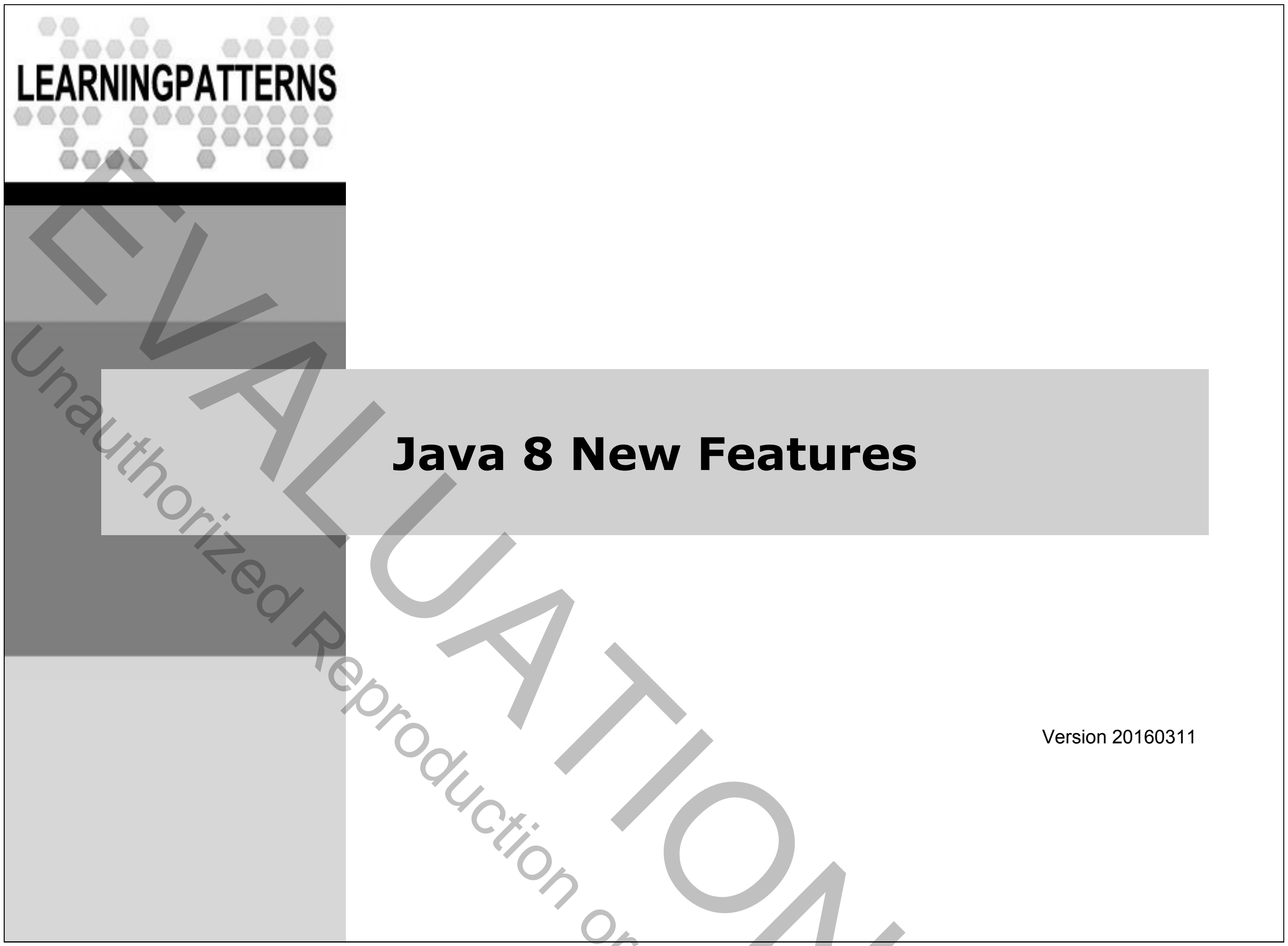
TRAINING MATERIALS FOR IT PROFESSIONALS

Java 8 New Features

For more information about Java Enterprise Java, or related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

LearningPatterns. Inc.	Services@learningpatterns.com | www.learningpatterns.com
Global Courseware Services	262 Main St. #12 | Beacon NY, 12508 USA 212.487.9064 voice and fax

Table of Contents – Java 8 New Features



LEARNINGPATTERNS

Java 8 New Features

Version 20160311

Notes:

- Version 20160311

Workshop Overview

- Intermediate level course covering the new features introduced in Java 8
- Course covers the following areas of Java 8:
  - **New interface capabilities**
  - **Lambda expressions and method references**
  - **Functional interfaces**
  - **Java Streams**
  - **New parallel processing capabilities and the Stream API**
  - **New Date / Time API**

Notes:

Workshop Agenda

- Session 1: **What's New in Interfaces**
- Session 2: **Lambda Expressions**
- Session 3: **Streams**
- Session 4: **Parallel Processing and Concurrency**
- Session 5: **Date and Time API**
- Session 6: **Other Capabilities**

Notes:

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

  `Catalog catalog = new CatalogImpl()`

  –Code fragments are the same, e.g., `catalog.speakTruth()`

  –We **bold/color** text for emphasis

  –Filenames and paths are in italics, e.g., *Catalog.java*

  –Notes are indicated with a superscript number [1] or a **star** *

  –Longer code examples appear in a separate code box (below)

```
JButton coolButton = new JButton("Press Me for Coolness");
cooltButton.addActionListener(
  (e) -> System.out.println("Lambda expressions are way cool")
);
```

Notes:

[1] If we had additional information about a particular item in the slide, it would appear here in the notes.

- We might also put related information that generally pertains to the material covered in the slide.

Labs

Lab

- The workshop has numerous hands-on lab exercises, structured as a series of labs
  - Many use types from a fictional case study called **JavaTunes**
    - An online music store
  - The lab instructions are separate from the main manual pages
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code
- Lab slides have an icon in the upper right corner of the slide
  - The end of a lab is marked with a stop sign

STOP

Notes:

LEARNINGPATTERNS

Session 1: What's New in Interfaces

Default Methods
Static Methods
Functional Interfaces

Notes:

Lesson Objectives

- Quick review of Java interfaces
- Use default methods in interfaces
  - And understand more complex inheritance scenarios
- Use `static` methods in interfaces
- Introduce definition of a ***functional interface***
  - These underlie many of Java 8's new features

Notes:

Interfaces – Quick Review

- Java interfaces specify a **type** that is (mostly) separate from any implementation
  - Often used to define ***roles*** played by an object

- An ***interface*** defines a type that is similar to a class – with some key differences:
  - Can declare methods, but **these methods are `abstract`**
    - Except for Java 8 default methods and `static` methods (covered next)
  - Can also have properties, but **all properties are `static final` constants**
  - Cannot be instantiated with the **new** keyword

Notes:

Interfaces – Example

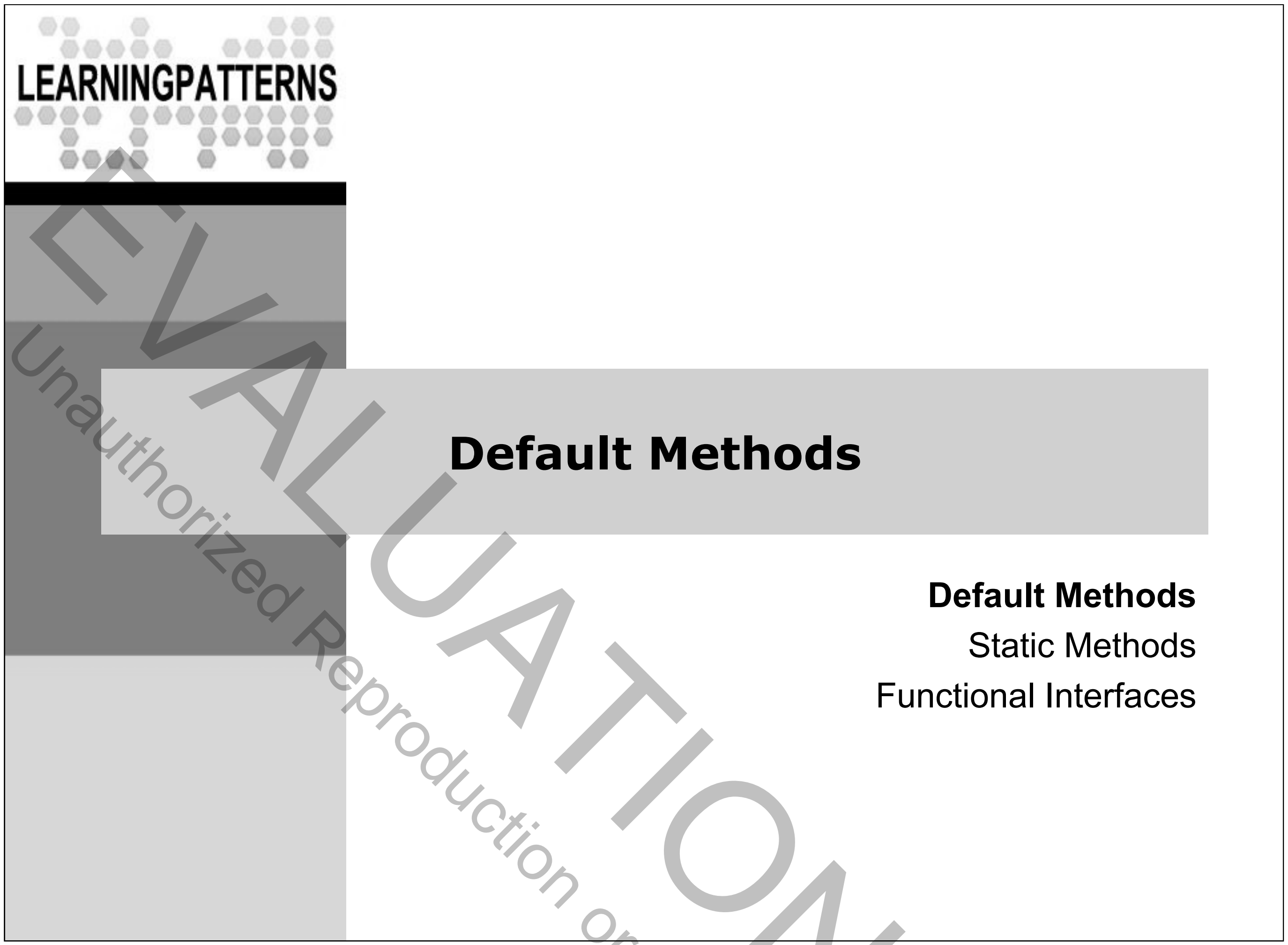
- Regular interface methods are declared without a body
  - Implicitly **abstract**, with no implementation, as shown below
- To implement an interface, you write a class with implementations for all the interface methods
  - If you don't implement all the methods, your implementation class must be declared as an abstract class

```
public interface Moveable {
  public void moveTo(String dest);
}
```

```
public class PosterTube implements Moveable {
  // provides an implemented moveTo(String) method
  public void moveTo(String dest) {
    // implementation here
  }
}
```

Notes:

- Like classes, interfaces can be placed in packages.
  - Like classes, interfaces are either `public` or not. `public` interfaces are visible to classes in other packages, whereas non-`public` interfaces are visible only within the same package.
    - In practice, most interfaces are declared to be `public`.
- Interface methods are inherently `public` and `abstract`.
  - In practice, the `public` keyword is used, but the `abstract` keyword is omitted.



Default Methods

Default Methods
Static Methods
Functional Interfaces

Notes:

Default Methods

- **Default methods** provide implementations in interface itself
  - Default methods are **not** `abstract`, and implementing classes **don't need to implement** them
    - But may override them with their own implementation

- Below, we add `getCurrentLocation()` to `Moveable`
  - **`default`** keyword and provided implementation (in `{ }`) indicate it is a default method

```
public interface Moveable {
  public void moveTo(String dest);

  default public Location getCurrentLocation() {
    // return GPS-based location - details not shown
  }
}
```

Notes:

- The details of the `Location` type, and the implementation of `getCurrentLocation()` are not relevant to how default methods work.

- Implementing classes only need to provide an implemented `moveTo()` method.
  - Optionally, they can also provide an implemented `getCurrentLocation()` method.
  - If they don't, the default implementation will be used.

- The `default` keyword does not need to appear first, though you'll usually find it written this way, for clarity. Our default method above could also be written as:

```
public default Location getCurrentLocation() { ... }
```

Default Methods – Example

- Below, `Car` implements `Moveable`
  - It provides an implemented `moveTo()` method
  - It inherits the default `getCurrentLocation()` method

```
public class Car implements Moveable {
  public void moveTo(String dest) {
    // implementation here
  }
}
```

```
class GetMoving {
  public static void main(String[] args) {
    Moveable m = new Car();
    m.moveTo("Seattle");                         // Car's impl
    System.out.println(m.getCurrentLocation());  // default impl
  }
}
```

Notes:

- Because `Car` implements `Moveable`, it can be referenced as type `Moveable`. A `Car` IS-A `Moveable`.
  - Sometimes you'll see this written as above, sometimes you'll see this as a parameter type:

```
public void ship(Moveable item) {  // a Car can be passed in
  ...
}
```

Motivation and Benefits of Default Methods

- **Provide a common implementation** for reuse
  - Implementing classes can inherit and use the functionality
  - For example, interface `java.lang.Iterable` defines a default `forEach()` method [1]
    - Inherited by `java.util.Collection`, and all implementing classes
    - Works easily with ***lambda expressions***, covered later
- Allows **easy evolution** of interfaces
  - When adding a new default method, **existing implementations continue working unchanged**
    - **Binary compatibility** is maintained
  - Adding a "regular" (abstract) interface method requires an implementation in **all** implementing classes
    - Breaks all existing implementation classes! (bad bad bad) [2]

Notes:

[1] `Iterable.forEach()` performs an action on each element in a collection.

- The method supports lambda expressions, covered later.
- The action can be defined and passed in at the point of call, using a lambda expression.
- We'll cover collections and lambda expressions later.

[2] Remember that all "regular" methods in an interface are implicitly `abstract`.

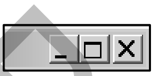
- An implementing class **must** implement them, or the class itself must be declared `abstract`.
- If we add a new, non-default method to an interface, binary compatibility is not maintained.
  - Your implementation class (which formerly compiled fine), is now flagged with compiler errors, because it doesn't implement the new method.
- If we add the new method as a default method instead, the method is available to be called, even for implementation classes written to the old interface definition.
  - The default implementation is inherited, and is used as needed.
  - No existing code is broken.

Motivation and Benefits of Default Methods

- Reduced need for adapter classes defining "empty" methods
  - Common before default methods (see notes)
  - Interfaces can now define default (or empty) implementations
- Enhances Collections API to support lambda expressions
  - Covered later
- NOTE: can't define **`Object`** methods like `equals()` as a default method [1]

Notes:

- "Interface adapter" classes are sometimes used in situations where a "large" interface is required, but in many uses cases, only one or a few of these methods is important.
  - Well-known example is the Java AWT `WindowListener` interface and accompanying `WindowAdapter` class.
    - `WindowListener` is an interface with 7 abstract methods, a "large" interface.
    - `WindowAdapter` is an implementing class with 7 empty methods, i.e., `{ }`.
    - You can subclass `WindowAdapter` and override only the methods you care about, and then use your subclass as a `WindowListener`. For example, to get the "X" button to respond by terminating the application, you need to pass a `WindowListener` to the UI window's `addWindowListener()` method. The applicable `WindowListener` method is `windowClosing(WindowEvent)`. Often, you don't care about the other 6 methods in `WindowListener`. In this case, you can subclass `WindowAdapter` and override only the `windowClosing()` method (you inherit the other 6 empty methods).



- With default methods in interfaces, the `WindowAdapter` class wouldn't be necessary.
  - You could just write a `WindowListener` implementation class with a single `windowClosing()` method. Your implementing class inherits the other 6 default methods.
  - For the record, `WindowListener` was **not** enhanced to supply default methods in Java 8.

[1] See : http://mail.openjdk.java.net/pipermail/lambda-dev/2013-March/008435.html

Inheritance Issues

- Situation: a class implements two interfaces, each with the same default method signature (return type is **not** considered)

```
public interface Moveable {
  public void moveTo(String dest);
  default public Location getCurrentLocation() {
    // impl 1
  }
}
```

```
public interface Item {
  public double getWeight();
  default public Location getCurrentLocation() {
    // impl 2
  }
}
```

```
public class Car implements Moveable, Item {
  public void moveTo(String dest) { ... }
  public double getWeight() { ... }
  // Q: which getCurrentLocation() method do I get??? (see notes)
}
```

Notes:

- **A: neither!** Compile-time error.
  - Since we have an ambiguity, the class must explicitly provide an implemented `getCurrentLocation()` method.

- Related, what happens if interface `Beta` **extends** interface `Alpha`, **both** of which define a default method named `doIt()`?
  - 1. Class `A` implements `Alpha`.
  - 2. Class `B` implements `Beta`.
    - In each case (1) and (2) above, what happens in `A` and `B` with respect to the `doIt()` method?
    - Hint: use your gut instinct – it does what you expect it to.

LEARNINGPATTERNS

Static Methods

Default Methods

Static Methods

Functional Interfaces

Notes:

Static Methods

- **Static methods** now legal in interfaces
  - Recall that `static` methods are associated with a **type**
  - Static methods in an interface work the same way as in a class
- Suppose a `Location` interface was defined as below
  - **`getNorthPole()`** is a "factory method" of `Location`, which returns a `Location` object
  - Factory methods are often `static`, and they can now be members of an interface

```
public interface Location {
  static public Location getNorthPole() {
    // return an instance of some class that implements Location
  }
}
```

```
client code:
Location northPole = Location.getNorthPole();
```

Notes:

- As usual, `static` methods are called as `Type.methodName()`, e.g., `Location.getNorthPole()` in the example above.

- The `static` keyword does not need to appear first, though you'll usually find it written this way, for clarity. Our `static` method above could also be written as:

```
public static Location getNorthPole() { ... }
```

Benefits of Static Methods

- Useful for utility methods and functions, e.g., those found in the `Collections` class and `Math` class
  - Reducing need for separate utility classes

- No need for separate "factory class"
  - The interface type serves as its own factory for instances [1]

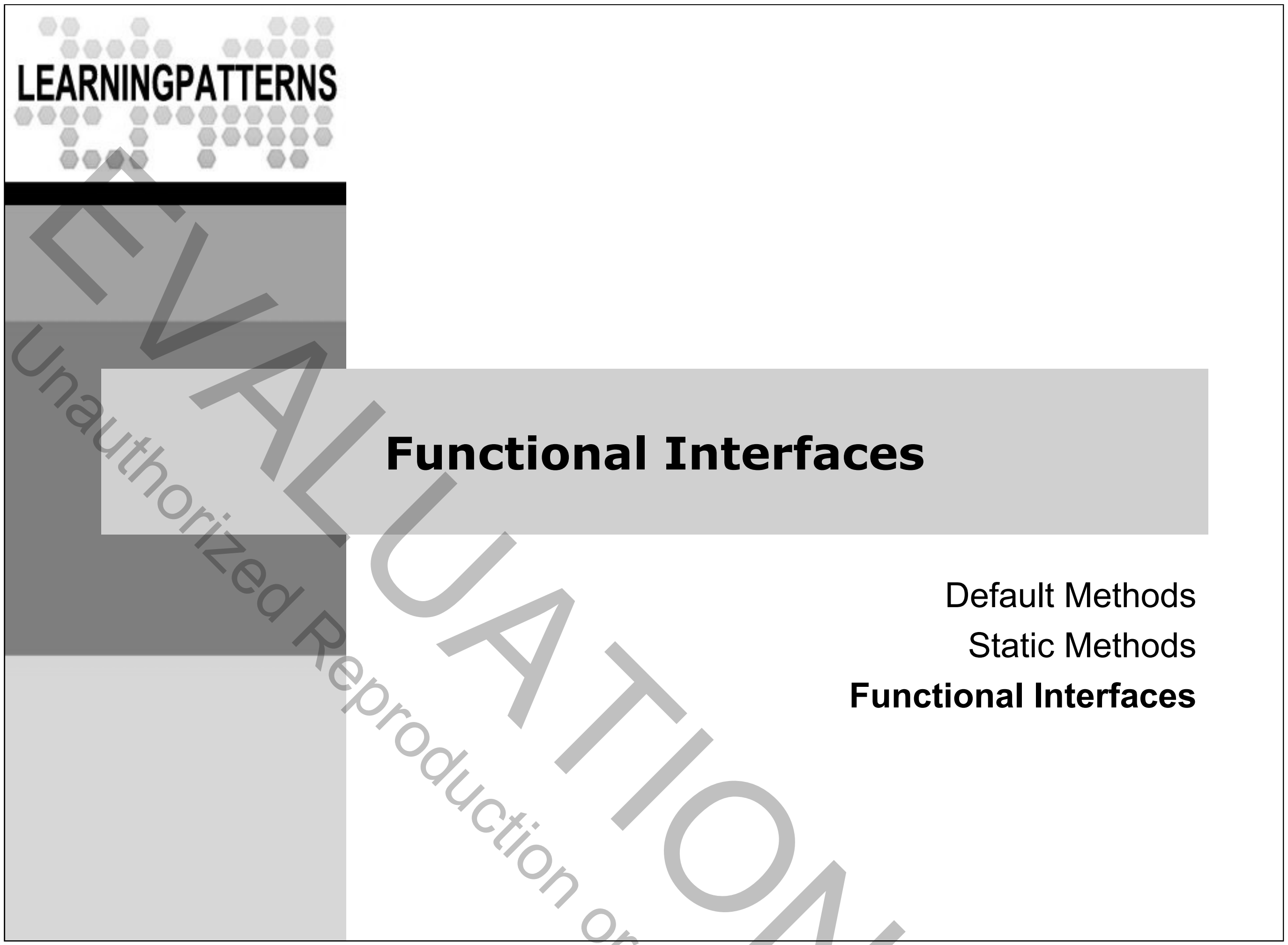
- NOTES:
  - Implementing classes can't override them
    - Just like subclasses can't, either
  - Can't define **`Object`** methods like `equals()` as a `static` method
    - Classes can't do this, either
  - Basically, all the normal rules apply

Notes:

[1] We'll see the `Comparator` interface a lot in this course. The Java 8 `Comparator` interface has added several new `static` methods that return an instance of `Comparator`, obviating the need for something like a `ComparatorFactory` class.

– You can thus get `Comparator` instances by using only the interface type.

```
Comparator c = Comparator.comparing(...);
Comparator c = Comparator.reverseOrder();
```



LEARNINGPATTERNS

Functional Interfaces

Default Methods
Static Methods
Functional Interfaces

Notes:

Functional Interface – Defined

- An interface that defines <u>**exactly one**</u> **abstract method**
  - Simple concept, but vitally important for lambda expressions
  - Used extensively later – just introducing the concept now

- **`@FunctionalInterface`** can be used to indicate a functional interface
  - **Not required** – only requirement is that interface define **exactly one** abstract method
  - If you use `@FunctionalInterface` on an interface that is not one, the compiler will flag it as an error

```
@FunctionalInterface  // optional, just gives a compile-time check
public interface ActionListener extends EventListener {
  public void actionPerformed(ActionEvent e);
}
```

Notes:

- The `ActionListener` interface has only one abstract method – `actionPerformed()`.
  - Therefore, it is a functional interface.

- In our example, we annotated `ActionListener` with `@FunctionalInterface`. However, that is not required.
  - The `@FunctionalInterface` annotation is not actually present in the definition of `ActionListener` in the Java library – we include it only for illustration. If, however, we put it on an interface that had zero or 2+ `abstract` methods, it would generate a compile-time error.
  - For example, the interface definition below results in a compilation error:

```
@FunctionalInterface
public interface MyInterface {
  public void myMethod1();
  public void myMethod2();
}
```

- Using `@FunctionalInterface` is a good idea, as it documents that this is a functional interface, plus it gives you the compile-time check.
  - Without it, the readers of your code would have to search through the methods, counting up the abstract ones to determine if it's a functional interface.

Functional Interfaces – Details

- Sometimes referred to as "SMI" or "SAM" types
  - SMI: Single Method Interface
  - SAM: Single Abstract Method

- NOTES:
  - Default and `static` methods do not count toward the "exactly one abstract method" rule
  - Nor do abstract methods defined in `Object` (see notes)

Notes:

- From the Javadoc for `@FunctionalInterface`:
  - An informative annotation type used to indicate that an interface type declaration is intended to be a *functional interface* as defined by the Java Language Specification. Conceptually, a functional interface has exactly one abstract method. Since default methods have an implementation, they are not abstract. If an interface declares an abstract method overriding one of the public methods of `java.lang.Object`, that also does *not* count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere.

Case Study – Comparator

- **`java.util.Comparator`** is a well-known interface
  - Often used for sorting a collection, e.g., a `List`, via this method:
    `void sort(`**`Comparator<E> c`**`)`
  - It's also a functional interface – **`int compare(E obj1, E obj2)`**

Mini-Lab

- See Javadoc for `Comparator` – google "Comparator Javadoc"
  - Look at the Java 7 version first
    - The `equals()` method does not count toward its abstract method count (see previous page)
  - Then look at the Java 8 version (wow!)
    - Is the Java 8 version still a functional interface? (1)
    - Note the many default and `static` methods that have been added

Notes:

(1) When looking over the Java 8 `Comparator` API doc, you will probably agree that the `@FunctionalInterface` annotation can be really useful.

- Because this interface is marked with `@FunctionalInterface`, we don't need to scroll down at all and count the number of abstract methods, making sure the sum is **exactly 1**.
- The presence of this annotation immediately answers the question, "Is the Java 8 version still a functional interface?"
  - Even though they added many more methods to it – you'll notice that these new methods are all default or `static` interface methods.

LEARNINGPATTERNS

Lab 1.1: Setup, Default Methods, Static Methods

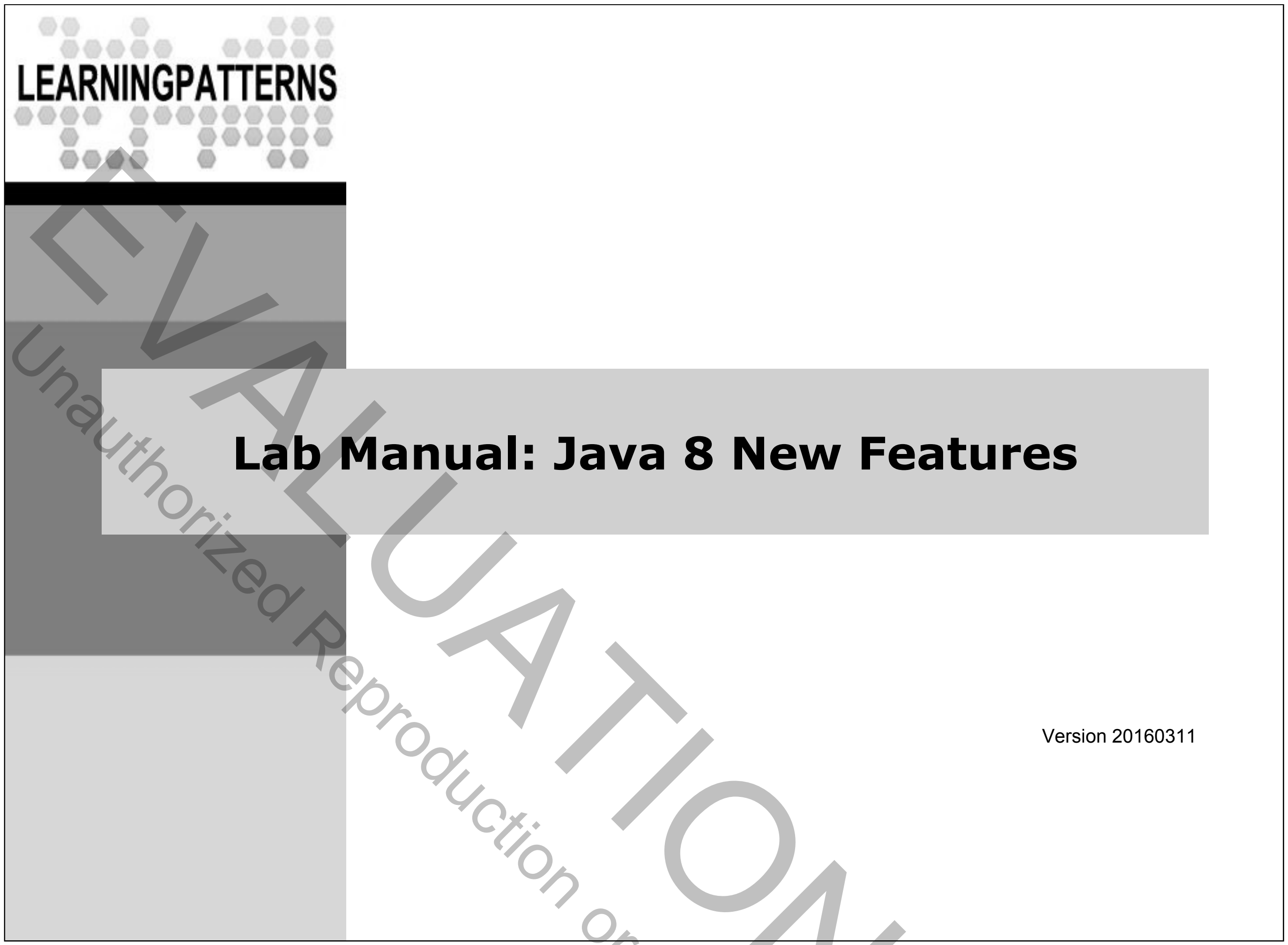
Notes:

Review Questions

1. An interface may have at most one default method. [T/F]
2. A default method in an interface can call other methods in the same interface. [T/F]
3. Explain how default methods help interfaces to evolve "cleanly," i.e., without breaking existing code.
4. Functional interfaces must include the `@FunctionalInterface` annotation. [T/F]
5. The `Comparator` interface has **2** abstract methods, yet it's a functional interface. Explain.
6. A static interface method can only return an instance of that interface type, e.g., `Shape.get()` must have `Shape` as its return type. [T/F]
7. What's wrong with the interface below?

```
public interface TaxPayer {
  public void register();
  public double payTaxes() {
    return 0.0;
  }
}
```

Notes:



LEARNINGPATTERNS

Lab Manual: Java 8 New Features

Version 20160311

Notes:

- Version 20160311

Release Level

Lab

- This manual contains instructions for creating and running the Java 8 New Features labs using the following software packages:
  - **Java 8 SDK**
  - **Eclipse Java EE – Luna (4.4) or later**
- The labs have been tested on Eclipse Luna/4.4 and Mars/4.5
  - Luna or later is required for Java 8 support

Notes:

- The instructions for the labs are geared for Eclipse Luna/4.4 or later.
  - Java 8 support is required, which is available in Eclipse 4.4+. Java 8 support can be added to previous versions of Eclipse, e.g., Kepler/4.3, with the addition of a patch.

A Word about JUnit

- Our labs and some examples use **JUnit** to run program code
  - JUnit is a popular open source Java testing framework
- JUnit tests have the following characteristics:
  - Annotate test methods with **`@Test`**
  - Make assertions using `static` methods in **`org.junit.Assert`**
  - We show an example below

```
import static org.junit.Assert.*;    // see notes on static imports
import org.junit.Test;

public class ArrayTest {
  @Test
  public void testArrayLength() {
    int[] intArray = {1, 2, 3, 4};
    assertTrue("length should be 4", intArray.length == 4);
  }
}
```

Notes:

- The `assertXXX()` methods are all `static` methods of `Assert`.
  - The familiar way to use these methods would be to import `org.junit.Assert`, and then call the `static` methods through the `Assert` class.
    `Assert.assertTrue(collection.isEmpty());`
  - This is a little cumbersome, so the *static import* feature of Java is used – which imports `static` members from a class.
  - The following `import` statement imports all the `static` members (including methods) from the `Assert` class:
    `import static` `org.junit.Assert.*;`
  - This allows us to use the `static` members without qualifying them by the classname:
    `assertTrue(collection.isEmpty());`
- There is much more capability in JUnit.
  - We won't go into that, since it's beyond the scope of the course.
  - We only cover enough to show how the labs work.

Test Cases in the Labs

- JUnit is a convenient driver for our lab code
  - It also lets us easily test our results
- In the labs, we'll generally give you the test class
  - Sometimes the tests are already written
  - Sometimes you'll need to add test code
- Note in the previous example how `testArrayLength()` is annotated with **`@Test`**
  - It creates an array, then checks the length
  - We use **`assertTrue()`** to perform the test
    - This is a `static` method of `Assert`
    - We use a ***static import*** for convenience – see notes previous page for details

Notes:

Lab 1.1: Setup, Default Methods, Static Methods

In this lab, we'll set up our lab environment, then work with default and static interface methods

Notes:

Lab Synopsis

Lab

- **Overview**:
  - Set up our **lab environment** and the **Eclipse IDE**
  - Add **default methods** and **`static` methods** to an existing interface
    - You'll first add a method as a regular (`abstract`) interface method, and see what happens – then you will change it to a default method
  - The types will be simple types based on a **`Volume`** interface that allows control of a device's volume
- **Builds on previous labs**: none
- **Approximate time**: 20-30 minutes

Notes:

The Eclipse Platform

Lab

- **Eclipse** (**www.eclipse.org**) is an 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 product line is built on top of Eclipse
- Eclipse products have two fundamental layers:
  - **Workspace**: files, packages, projects, resource connections, configuration properties
  - **Workbench**: editors, views, and perspectives
- We will set up the workspace and workbench, then do our lab

Notes:

- The workbench sits on top of the workspace and provides visual artifacts that allow you to access and manipulate various aspects of the underlying workspace resources.

Information Content and Task Content

Lab

- In a lab, information-only content is presented the same as in the student manual pages
  - Like the bullets here
- Tasks for students are in a box like the one below

Tasks to Perform

- Note the different look of this box as compared to that above
  - All future labs will use this format
- Make sure that you have **Java 8 or later installed**
  - Likely in a directory such as *C:\Program Files\Java\jdk1.8.0_51*
  - If not, you'll need to install it – it can be downloaded from:
    - ***oracle.com/technetwork/java/javase/downloads***
- OK – now **get out your setup files**, we're ready to start working

Notes:

- The setup files may be on your workstation, or they may be provided by your instructor.

Extract the Lab Setup Zip File

Lab

- To set up the labs, you'll need the course setup zip file
  - It has a name like: ***LabSetup_Java8New_yyyyMMdd.zip***
- Our base working directory for this course will be ***C:\StudentWork\JNew***
  - Gets created when we extract the lab setup zip
  - Includes a directory structure and files (Java source files and others) that will be needed in the labs
  - All instructions assume that this zip file is extracted to ***C:\***
    If you choose a different directory, please adjust accordingly

Tasks to Perform

- Unzip the lab setup file to ***C:\***
  - Creates the *StudentWork/JNew* directory structure, containing files that you will need for doing the labs

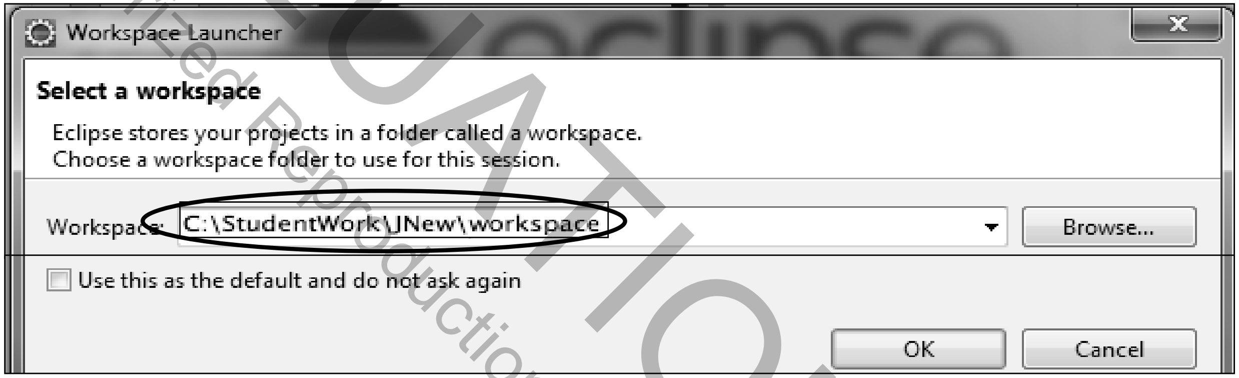
Notes:

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 workspace location to ***C:\StudentWork\JNew\workspace***
  - If a different default workspace location is set, change it


Workspace Launcher

Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\StudentWork\JNew\workspace

Use this as the default and do not ask again

Notes:

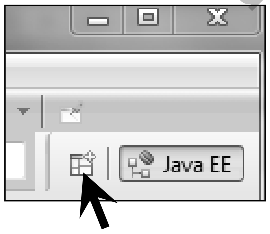
- If Eclipse was installed elsewhere, adjust the path to the Eclipse executable accordingly.
- You can also put a shortcut on your desktop to start Eclipse.
- If you need to download Eclipse, go to **http://www.eclipse.org/downloads**.
  - In the **Package Solutions** section, click on the link for the **Eclipse IDE for Java EE Developers**.
  - Save the zip file to your computer, and unzip it. The easiest location to unzip it to is ***C:***\, but another location is fine as long as you can get to it to run the *eclipse.exe* executable.

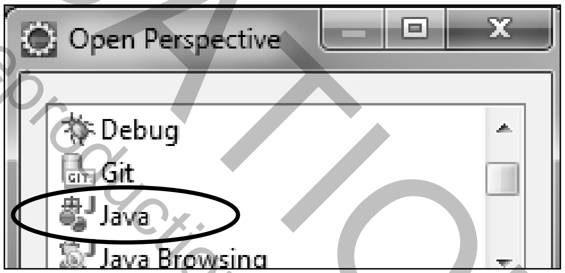
Workbench and Java Perspective

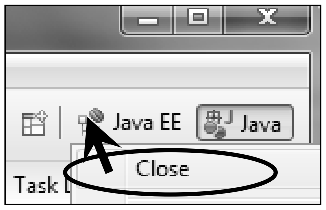
Lab

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 default for the Eclipse Java EE version
- Close the Java EE perspective by right clicking its icon, and selecting close (as shown below right)

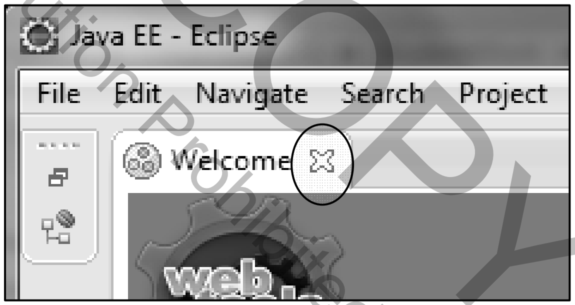






Notes:

- To close the Welcome screen, click the X in the tab, as shown below.
- The Eclipse Java EE version opens in the Java EE perspective by default.

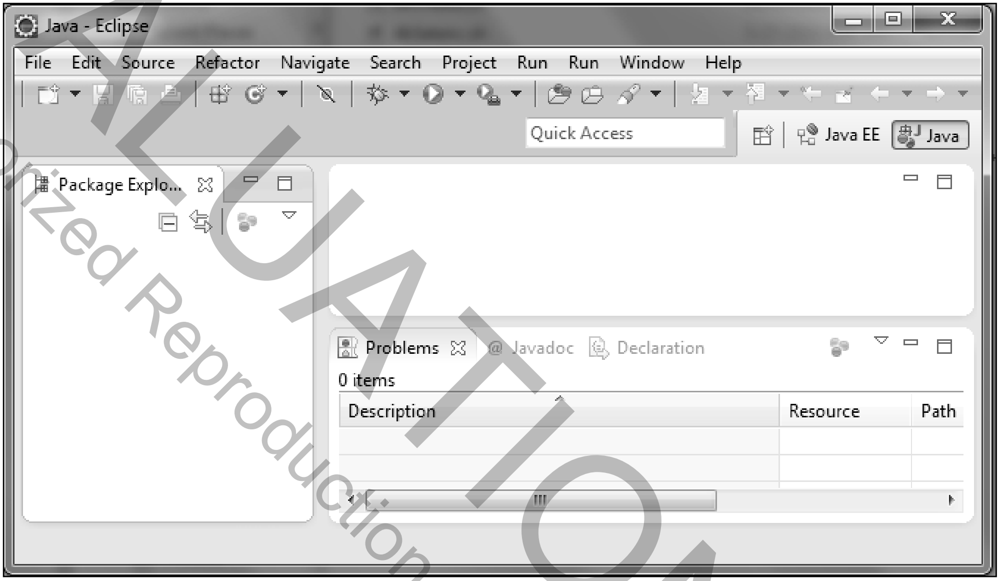


Unclutter the Workbench

Lab

Tasks to Perform

- Unclutter the Java perspective by closing some views
  - Close the Task List and Outline views (click on the X)
  - You can save these changes to the perspective (see notes)



Notes:

- Save these changes to the Java perspective via **Window → Save Perspective As → Java**.

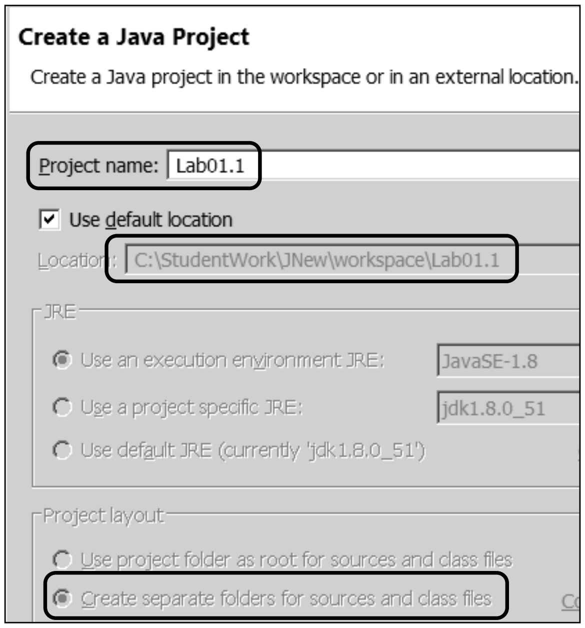
- You can reset the perspective to it defaults via **Window → Reset Perspective**.

Create New Java Project

Lab

Tasks to Perform

- Create a new **Java** project
  - **File → New → Java Project**
- Name it **exactly** **Lab01.1** (1)
  - When you name the project **Lab01.1**, it will be stored in ***workspace\Lab01.1***
    - And it will pick up some starter code provided in that directory (1)
  - Make sure "Create separate folders for sources and class files" is selected
    - May be preselected but grayed out – that's OK
- **Next**



Create a Java Project

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

Project name: Lab01.1

☑ Use default location

Location: C:\StudentWork\JNew\workspace\Lab01.1

JRE

● Use an execution environment JRE: JavaSE-1.8

○ Use a project specific JRE: jdk1.8.0_51

○ Use default JRE (currently 'jdk1.8.0_51')

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files

Notes:

(1) **IMPORTANT NOTE**: you **must** name the project **Lab01.1**, in order to pick up the starter code provided in the lab setup. That's because the starter code was preprovisioned in the *workspace/**Lab01.1*** directory (in appropriate subdirectories).

- By naming the project **Lab01.1**, the project directory defaults to *workspace/**Lab01.1***, and the starter code is automatically recognized and included in the project's fileset.

- There are multiple ways to create a new project.
  - **File → New → Project → Java → Java Project**.
  - Click on the "**New**" wizard icon in left side of the toolbar.
  - Right click in the Package Explorer view, select **New ...**

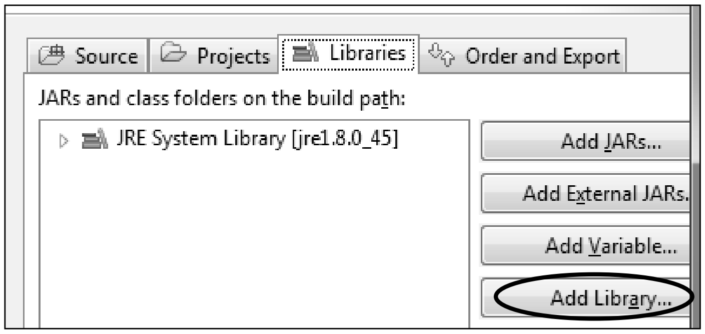
- It's usually better to create separate folders for source and class files.
  - We've done that here, and Eclipse puts the *.java* files in the lab's *src* directory, and the *.class* files in the lab's *bin* directory.
  - This is the default structure for Java projects in Eclipse.

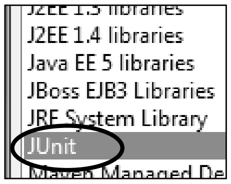
Add JUnit to Build Path

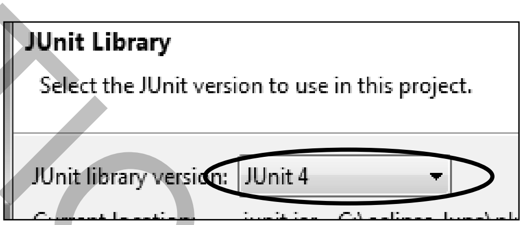
Lab

Tasks to Perform

- In the next dialog (Java Settings) click **Libraries** tab
- Click the **Add Library** button
  - In the list that comes up, select JUnit, and click Next
  - Select the JUnit 4 library and click Finish
- Back in the main wizard, click **Finish** to create the project
- See notes






JUnit Library

Select the JUnit version to use in this project.

JUnit library version: JUnit 4

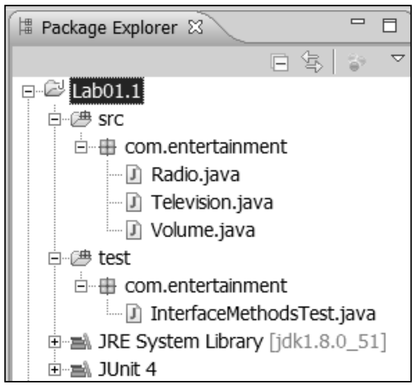
Notes:

- If you ever forget to add JUnit during initial project creation, it can also be added later.
  - In Eclipse, right-click on project directory and choose **Build Path → Add Libraries → JUnit → JUnit 4**.

New Project with Starter Code

Lab

- The lab setup pre-provisioned some starter code in the *workspace/**Lab01.1*** directory
  - By naming the project **Lab01.1**, the root project directory becomes *workspace/Lab01.1*
  - Eclipse sees the ***src*** and ***test*** directories there, and imports these files into the project
    - We will use this technique several more times when creating new projects in the labs



```
Package Explorer
Lab01.1
  src
    com.entertainment
      Radio.java
      Television.java
      Volume.java
  test
    com.entertainment
      InterfaceMethodsTest.java
  JRE System Library [jdk1.8.0_51]
  JUnit 4
```

- Once you've created the project and you see the starter code, you can rename it if you wish
  - Perhaps something to indicate its purpose, e.g., Lab01.1_InterfaceMethods
  - Rename it by pressing **F2** on the project folder in Eclipse, or **right-click → Refactor → Rename**

Notes:

The Lab Types

Lab

- **`Volume`**: interface that declares volume-control methods
- **`Television`**: class that implements `Volume`
  - Also has `brand` property (string), and some validation code
- **`Radio`**: class that implements `Volume`
  - Doesn't have much else in it
- Above types in are package `com.entertainment`

- **`InterfaceMethodsTest`**: JUnit test case
  - Also in package `com.entertainment`

```
public interface Volume {
  public void setVolume(int volume);
  public int getVolume();
  public void mute();
  public boolean isMuted();
}
```

Notes:

Add Default Interface Method

Lab

Tasks to Perform

- Open interface `com.entertainment.`**`Volume`**
  - Add a "regular" (`abstract`) interface method with the signature below: **`public void silence();`**
  - Look in the Problems view, there should be errors
    - `Television` and `Radio` don't implement this method

- Change `silence()` to be a default method
  - Set the volume to zero by calling `setVolume(0)`
    - This is a simpler form of muting – we don't save the old volume for unmute
  - Your errors should now disappear

- Open `com.entertainment.`**`InterfaceMethodsTest`**
  - Review the code
  - In **`testDefaultMethods()`**, uncomment call to `silence()` in the loop

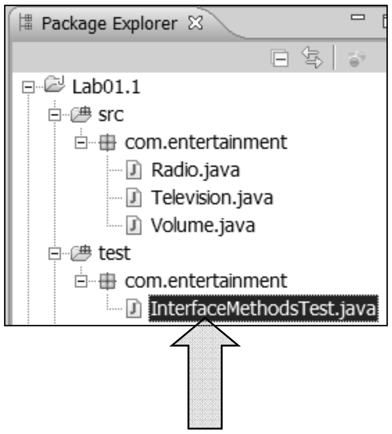
Notes:

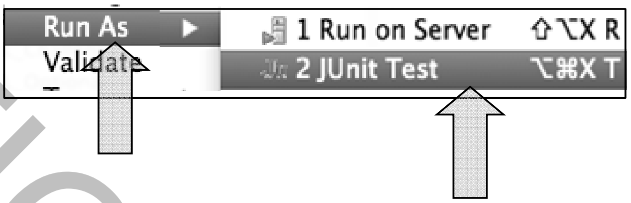
Run Your Tests

Lab

Tasks to Perform

- We use Eclipse to run the tests
- In **Package Explorer**, right-click on `InterfaceMethodsTest`
  - **Run As → JUnit Test**
  - Automatically finds and runs its test methods



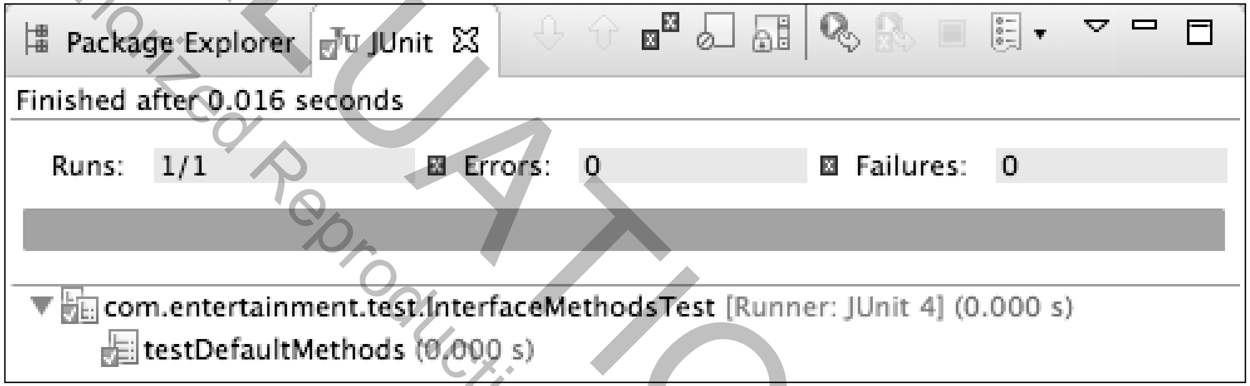


Notes:

Test Output

Lab

- You should see JUnit output in a JUnit view, as shown below
  - All tests should pass (1)
  - If your test produced output to stdout, it would be in the Console view
- **Note**: you'll follow similar procedures **whenever you have a JUnit test client to run** in the labs (2)



```
Finished after 0.016 seconds

Runs: 1/1        Errors: 0        Failures: 0

▼ com.entertainment.test.InterfaceMethodsTest [Runner: JUnit 4] (0.000 s)
      testDefaultMethods (0.000 s)
```

Notes:

(1) If you see any errors in the JUnit view, or exceptions in the Console view, then you've got something configured incorrectly.

(2) When running programs in other labs, the only thing that may change is the program class that you'll need to right click on to run.

– The general procedure will remain the same.

Add Static Interface Method and Test

Lab

Tasks to Perform

- Open interface `com.entertainment.`**`Volume`**
  - Add a `static` method, **`silenceAll()`** that takes a varargs `Volume` argument, as shown below:

    `void silenceAll(Volume... vols)`
    - The method should loop through its arguments and silence each one
    - See notes for working with the `vols` parameter
- In your JUnit test class:
  - Uncomment the **`@Test`** annotation on **`testStaticMethods()`**, as well as the call to `silenceAll()`
  - **Run** the test case again – it should run with no failures
- You can see that these new interface additions are easy to work with

STOP

Notes:

(1) An incoming varargs argument can be treated like an array – in fact, internally, it's converted to one.

– You can iterate over it as you would with any other array, here using a for-each loop:

```
for (Volume vol : vols) {
  ...
}
```

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com

9-06-00892-000-10-23-18