itcourseware™

TRAINING MATERIALS FOR IT PROFESSIONALS

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

This material is copyrighted by LearningPatterns Inc. This content and shall not be reproduced, edited, or distributed, in hard copy or soft copy format, without express written consent of LearningPatterns Inc. Copyright © LearningPatterns Inc.

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 962 Main St. #Ste. 4-167| Fishkill NY, 12524 USA
212.487.9064 voice and fax

Java, and all Java-based trademarks and logo trademarks are registered trademarks of Oracle, Inc., in the United States and other countries. LearningPatterns and its logos are trademarks of LearningPatterns Inc. All other products referenced herein are trademarks of their respective holders.

Your Source for Quality Java™ Technology Courseware

Table of Contents: Java Testing with JUnit 5

Java Testing with JUnit 5 — **1**
Course Overview — 2
Course Agenda — 3
Typographic Conventions — 4
Labs — 5

Session 1: Unit Testing with JUnit 5 — **6**
Session Objectives — 7

Overview — **8**
Unit Testing Overview — 9
JUnit Overview — 10
Introducing JUnit 5 — 11
What's New in JUnit 5? — 12
JUnit Library Components — 13
JUnit Library Components – Illustrated — 14
Writing a Test – First Example — 15
Running Tests in the IDE — 16
Running Tests in Other Environments — 17
Naming Conventions and Organizing Tests — 18
Lab 1.1 – Getting Started with JUnit — **19**

Tests and Assertions — **20**
Writing Test Methods — 21
It's All about Expectations — 22
Assertions — 23
Floating Point Assertions — 24
Assertion Messages — 25
Assertion Messages – Useful or Not? — 26
Assertion Messages – a Practical Approach — 27
Lab 1.2 – Tests and Assertions — **28**

Test Fixtures and Lifecycle — **29**
Test Fixtures – `@BeforeEach` and `@AfterEach` — 30
Test Fixtures – Example — 31
`@BeforeAll` and `@AfterAll` — 32
Order of Execution – Full Test Lifecycle — 33
Test Instance Creation — 34
Per-Class Lifecycle — 35
Lab 1.3 – Test Fixtures and Lifecycle — **36**

Session 2: Writing and Running Tests — **37**
Session Objectives — 38

- Brief Aside – Java 8 New Features - — **39**
Dates and Optionals — 40
Default and Static Interface Methods — 41
Functional Interface – Defined — 42
Stripping a Method Down to its Essence — 43
Lambda Expression as Functional Interface — 44
Functional Interfaces and Lambdas — 45
JUnit Functional Interfaces — 46
Lambdas in JUnit – `Executable` — 47
Lambdas in JUnit – `ThrowingSupplier<T>` — 48

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Copyright © LearningPatterns Inc. All rights reserved

Lambdas in JUnit – `ThrowingConsumer<T>` 49

Additional Testing Needs **50**
Testing for Exceptions 51
Testing for Exceptions – Example 52
Setting Timeouts 53
Setting Timeouts – Example 54
Assertion Groups 55
Assertion Groups – Example 56
Lab 2.1 – Exceptions, Assertion Groups, and Timeouts **57**

Running Tests **58**
JUnit Library Components – Revisited 59
Lots of Ways to Run Tests 60
IDE Support – Eclipse 61
IDE Support – IntelliJ IDEA 62
Configuration for Maven Project – `mvn test` 63
Configuring `maven-surefire-plugin` 64
JUnit Platform Console Launcher 65
Using the Console Launcher 66
Running Tests Programmatically 67
Deciding Which Tests Get Run 68
Test Discovery and Selection 69
Lab 2.2 – Running Tests **70**
Display Names 71
Grouping and Filtering Tests with Tags 72
Specifying Tags at Test Run 73
Tag Expressions 74
Configuration Parameters 75
Setting Configuration Parameters 76
Nested Tests 77
Regular Test Class Results – Illustrated 78
Nested Test Class Results – Illustrated 79
Nested Tests – Example 80
Rules for Nested Test Classes 81
Lab 2.3 – Test Reporting and Filtering **82**

Advanced Capabilities **83**
Custom Composed Annotations 84
Practicing Good OO in Test Classes 85
Inheritance with Test Classes 86
It's All about Removing Redundancy 87
Extensions – Overview 88
Defining and Using Extensions 89
Callbacks and Insertion Points 90
Test Lifecycle with Extension Callbacks 91
`ParameterResolver` 92
Conditional Test Execution [new in 5.1.0] 93
Conditional Tests vs. Assumptions 94
Deactivating Conditions at Runtime 95
Parameterized Tests 96
Parameterized Tests – Details 97
Lab 2.4 – Advanced Capabilities **98**

JUnit 4 Migration **99**
The Do Nothing Case 100
Stuck with Old Software 101

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Copyright © LearningPatterns Inc. All rights reserved

Migrating JUnit 4 Tests to Jupiter API 102
Steps in Migration – API Changes 103
Steps in Migration – Runners and Rules 104
JUnit 4 Test Suites 105
JUnit 4 Test Suites on the JUnit Platform 106
JUnit 4 Test Suites – Example 107

Best Practices 108
Tests Are Real Code 109
What to Test 110
Testing `void` and Private Methods 111
Be Thorough 112
Many Small Tests vs. Fewer Large Tests 113
Try to Break It 114
Narrow and Broad Testing 115
Characteristics of Good Tests 116
Writing Testable Code 117
Writing Testable Code 118
Testing Anti-Patterns 119
Further Reading 120

Session 3: Testing with Mocks 121
Session Objectives 122

Overview 123
Overview of Mock Objects 124
Mocks as Collaborators 125
Introducing Mockito 126

Creating and Using Mocks 127
Basic Steps in Mocking 128
The `Mockito` Class 129
An Evolution in Mock Creation 130
Mock Creation with JUnit 5 131
Stubbing Method Calls 132
How Much Stubbing? 133
Example – Business Object and Collaborator 134
Example – Setting up the Mock and Testing 135
Lab 3.1 – Using Mock Objects 136

Additional Capabilities 137
Argument Matchers 138
Using Argument Matchers 139
Partial Mocking with Spies 140
Mocking the Unmockable 141
Dependency Injection of Mocks 142
Dependency Injection of Mocks 143
Lab 3.2 – Additional Capabilities [Optional HW] 144

Session 4: Testing Enterprise Components 145
Session Objectives 146

Overview 147
Types of Testing 148
Types of Testing 149
High-Level Issues Involved 150
Composition and the HAS-A Relationship 151
Composition in the Java EE Container 152

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Testing Options 153
Standalone vs. In-Container Testing 154
Standalone vs. In-Container Testing 155
Standalone vs. In-Container Testing 156
Not Necessarily Competitors 157

Testing the Persistence Layer 158
Interface vs. Implementation 159
You Really Need a Database 160
Standalone vs. In-Container Testing 161
Issues Specific to Database Testing 162
Lab 4.1 – Testing the Persistence Layer 163
In-Container Testing 164
What It Looks Like 165
Container Configuration and Deployment 166

Testing Services 167
Similar Issues, Different Layer 168
Is It Worth It? 169

Testing Web Components 170
We Saved the Hardest for Last 171
Similar Issues, Easier Answer 172
In-Container Testing Isn't Enough 173
Manual vs. Automated Testing 174
Automate as Much as Possible 175
Automating the Web Browser 176

Recap 177
Recap of What We've Done 178
Recap of What We've Done 179
Resources 180

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Java Testing with JUnit 5

Version 20180418

Copyright © LearningPatterns Inc. All rights reserved. 1

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- Version 20180418

Copyright © LearningPatterns Inc. All rights reserved.

Course Overview

- This is a 2-day course about testing in Java, using **JUnit 5**
- It covers the following high-level areas:
  - Writing and running tests
  - Using mock objects
  - Testing enterprise components
- A few notes [see notes]: ← hey, that's funny
  - The course does not require prior experience with unit testing or JUnit, and all the fundamentals of JUnit 5 are covered in detail
  - **JUnit 5 requires a Java 8 runtime** and employs several of its new language features and APIs

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- Note that the structure and implementation of JUnit 5 is substantively different than JUnit 4, though the core testing principles and techniques are the same.
  - If you have experience with previous versions of JUnit, that will certainly be helpful, **but** you'll need to be aware of the differences.
    - As one example, JUnit 5 has a `@Test` annotation, as does JUnit 4, but these are **different annotations** – they are in different packages and have different attributes.
- Java 8 was released in 2014. At a minimum, you're mostly likely aware that this is a significant release, bringing with it new language features and APIs, including:
  - Lambda expressions and the formal definition of a *functional interface*.
  - The ability to provide implemented methods in interfaces.
  - Streams API (`java.util.stream`), Date/Time API (`java.time`), `Optional` (`java.util`), and a family of predefined *functional interfaces* (`java.util.function`):
    - `Predicate`, `Function`, `Consumer`, and `Supplier`.
- It's undoubtedly the most significant release since Java 5.
  - If you've managed to steer clear of it so far, it's time to get acquainted!
  - The course includes a Java 8 primer, which should be enough to get you started. For more complete coverage, consider the 2-day Java 8 New Features course.

Copyright © LearningPatterns Inc. All rights reserved.

Course Agenda

- Session 1: **Unit Testing with JUnit 5**
- Session 2: **Writing and Running Tests**
- Session 3: **Testing with Mocks**
- Session 4: **Testing Enterprise Components**

Notes:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Copyright © LearningPatterns Inc. All rights reserved.

Typographic Conventions

- Code in the text uses a fixed-width font

```
Catalog catalog = new CatalogImpl()
```

- Code fragments are the same, e.g., `catalog.findById(4)`
- We **bold/color** text for emphasis
- New terms are often introduced in ***bold italics***
  - Filenames and paths are also 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)

```
package com.javatunes.catalog;

public class CatalogImpl implements Catalog {
   public MusicItem findById(long id) {
      ...
   }
}
```

Introduction 4

Notes:

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

- We might also include related information that pertains more generally to the slide.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Copyright © LearningPatterns Inc. All rights reserved.

Labs

- The course has numerous hands-on lab exercises
  - Many use types from a fictional case study called **JavaTunes**
    - An online music store
  - The lab instructions are separate from the main manual
- Setup files are provided with skeleton code for the labs
  - Students add code focused on the topic at hand
  - 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

Introduction 5

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Session 1: Unit Testing with JUnit 5

Overview
Tests and Assertions
Test Fixtures and Lifecycle

Session 1: Unit Testing with JUnit 5 6

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Session Objectives

- Understand basic principles of unit testing
- Understand JUnit library structure and testing model
- Set up the IDE and organize tests in an IDE project
- Write and run tests in the IDE
- Write assertions to verify correct application behavior
- Use test fixtures to manage the objects under test
- Understand the lifecycle of a test run

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Overview

Overview

Tests and Assertions

Test Fixtures and Lifecycle

EVALUATION COPY

Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Unit Testing Overview

- Testing is a critical part of software development
  - Assess **behavior** of software with respect to **expectations**
- Unit testing focuses on "individual units" of a larger system
  - In Java, this is typically a **method** of a class
  - ***Test cases*** are test methods written to perform each unit test
    - Shows that the code does what we (developers) expect it to do
  - Verifies that individual parts of a system are **correct**
    - And that they **remain correct** when **changes** are made (**this is key**)
- Unit testing allows you to write better code in less time
  - Quickly detect bugs when changes / additions are made
  - Verification of edge cases that might otherwise be overlooked
  - Encourages writing **smaller** chunks of **testable** code (**also key**)

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- Unit testing is generally done by the developers themselves.
- The testing we are talking about does not measure "quality" but rather how closely software meets requirements.
  - All requirements are captured in tests. When the code passes the tests, you are "finished" writing the code ("finished" in quotes because all code is subject to refactoring at some point).
  - The codebase of tests is like an "executable specification" of the system.
  - As requirements evolve and/or the software increases in complexity and size, testing becomes even more important.
  - Note that unit testing doesn't measure whether the code performs what the **end-user wants**. It checks that the code does what the **developer expects**.
- Ideally, unit testing is done in isolation from other parts of the system – with each test case being independent from the others.
  - Sometimes mock or fake objects are used to help perform a test case in isolation.

Copyright © LearningPatterns Inc. All rights reserved.

JUnit Overview

- Open-source Java testing framework
  - Most popular in industry – universally supported in IDEs, tools
  - See notes for a brief history
- **Automates** unit testing of Java code
- Provides standard way to write and organize tests
- Consists of classes and annotations to write and run tests
  - **Annotations** for declaring test methods (`@Test`)
  - **Assertions** for testing actual results against expected results
  - **Test fixtures** for setting up each test's environment
  - **Test engines and runners**
    - Built-in, plus extension mechanism for 3rd party implementations

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- JUnit Home: ***http://junit.org***.
- A brief history:
  - **Kent Beck** developed the SUnit automated test tool for Smalltalk in the mid-1990s.
  - Beck and Erich Gamma (of design patterns Gang of Four) supposedly developed JUnit on a flight from Zurich to Washington, DC.
    - Probably true, as Beck's main JUnit example uses a `Money` class, and many instances of `Money` in his example have currency CHF (Swiss Franc).
  - Martin Fowler has said, "Never in the field of software development was so much owed by so many to so few lines of code."
  - Many "xUnit" tools have since been developed for other languages and environments, including Perl, C++, Python, Scala, Visual Basic, C#, .NET, web testing (HttpUnit), database testing (dbUnit), etc.
- JUnit is the #1 most prevalent 3rd party library used in Java projects.
- JUnit is simple, effective, ubiquitous.

Copyright © LearningPatterns Inc. All rights reserved.

Introducing JUnit 5

- History and evolution
  - **JUnit 3:** one JAR ***junit.jar***
    - Subclass `TestCase`, write test methods named "`test`Something", optionally override designated lifecycle methods for init and cleanup
  - **JUnit 4:** two JARs ***junit.jar***, ***hamcrest.jar***
    - Annotation-based approach: `@Test`, `@Before`, `@RunWith`, etc.
    - Enhanced test run lifecycle, "matcher" assertions, cleaner testing for exceptions, parameterized tests, timeouts, external test runners
- **JUnit 5** initial release September, 2017
  - Structure is significantly more complex
    - Multiple components: platform + test engine + API (details next)
  - BUT from a developer perspective, it's fundamentally the same
  - Java 8 runtime required (older-compiled code ok)
  - Java 9 ready (where code (in JARs) is organized into modules)

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- Note that the Java package names have changed with each release:
  - JUnit 3: `junit.framework`, `junit.runner`, `junit.extensions`.
  - JUnit 4: `org.junit` and several subpackages.
  - JUnit 5: `org.junit.platform`, `org.junit.jupiter`, `org.junit.vintage` (each with several subpackages).
- As mentioned in the Introduction session, JUnit 5 requires a Java 8 runtime and employs several of its new language features and APIs.
  - But using it to test older code (Java 7 and earlier) is just fine.

Copyright © LearningPatterns Inc. All rights reserved.

What's New in JUnit 5?

- Console Launcher
  - Flexible test run environment from the command line
- Java 8 lambdas
- Test run filtering and conditional test execution
- Display names and nested tests
  - Aids in test reporting and organizing tests
- Parameterized tests
- Single standardized extension mechanism
  - Additional lifecycle callbacks, more flexible approach for 3rd party libraries, e.g., Spring, Mockito

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

JUnit Library Components

- **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

- **Platform** provides API and infrastructure for `TestEngines`, legacy test runners, and build tools (Gradle, Maven)
  – Also provides a Console Launcher for command line use

- **Jupiter** is the programming model and API for test developers
  – **This is your main interface to JUnit**
    • As a developer writing unit tests, this is what you code to
  – Also provides a `TestEngine` to run JUnit 5 tests on the Platform

- **Vintage** allows JUnit 3 and 4 tests to run on the Platform
  – Via another `TestEngine`
  – In fact, that's all Vintage is – a `TestEngine` for older tests

Session 1: Unit Testing with JUnit 5 13

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

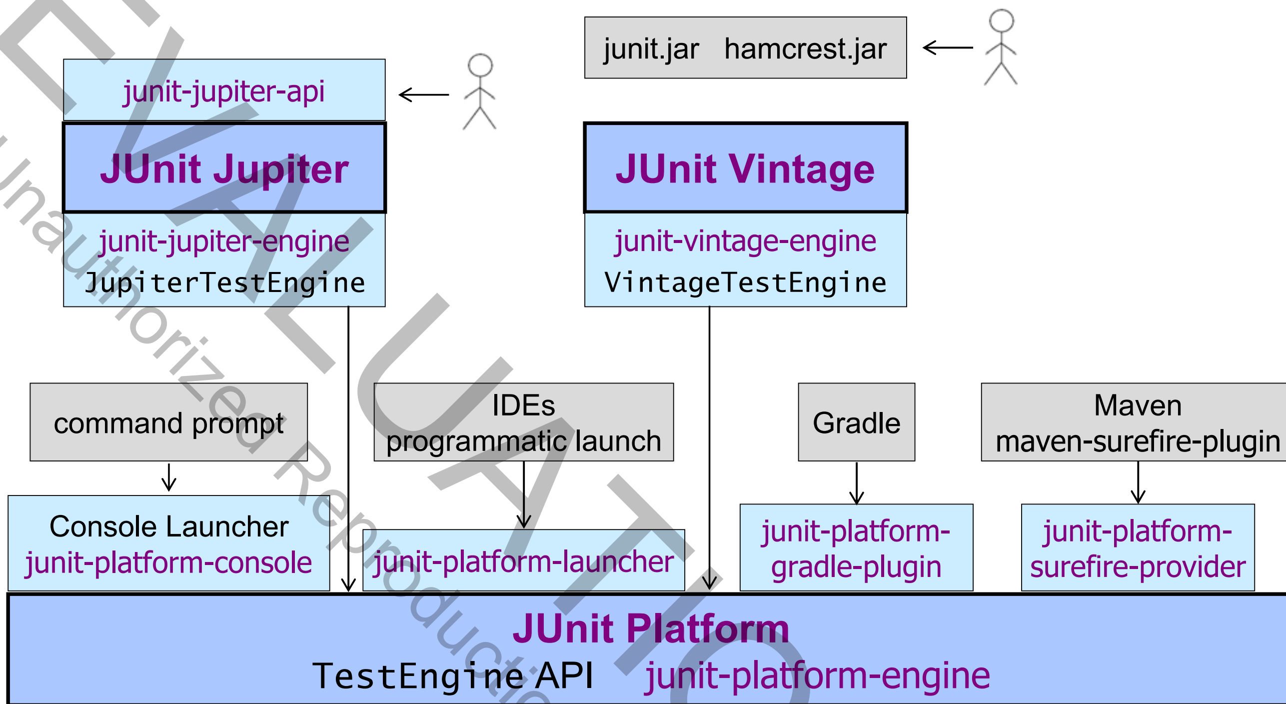
- Compared to JUnit 3 (one JAR) and JUnit 4 (two JARs), it can be difficult to figure out exactly which pieces do what, and which pieces you need in **your** project. We clarify all that here.

- From the JUnit User Guide (paraphrased, text added, and reorganized):
  – The **JUnit Platform** serves as a foundation for launching testing frameworks on the JVM.
    • It defines the `TestEngine` API for developing a testing framework that runs on the platform. JUnit Jupiter and JUnit Vintage both supply `TestEngine` implementations – for running JUnit 5 ("Jupiter") tests, and JUnit 3 and 4 ("Vintage") tests, respectively.
    • It provides a `ConsoleLauncher` to launch the platform from the command line.
    • It provides plugins for running tests in Gradle and Maven build systems.
    • It also provides a JUnit 4 `Runner` class for running JUnit 5 Jupiter tests in IDEs and build systems that support JUnit 4 but do not yet support the JUnit 5 Platform directly.
  – **JUnit Jupiter** is the combination of the new programming model and extension model (APIs) for writing tests and extensions in JUnit 5. It also provides a `TestEngine` (`JupiterTestEngine`) for running Jupiter-based (JUnit 5) tests on the platform.
  – **JUnit Vintage** provides a `TestEngine` (`VintageTestEngine`) for running JUnit 3 and JUnit 4 tests on the platform.
- As mentioned in the description above, "Jupiter test" = "JUnit 5 test" – these are synonyms.
  – The User Guide generally uses the term "Jupiter test."

Copyright © LearningPatterns Inc. All rights reserved.

JUnit Library Components – Illustrated

- You, the developer, are the stick figure coding to the JUnit API



Notes:

- INDEED, a bit more involved than putting one or two JARs on the classpath!

- JUnit provides two `TestEngine` implementations out of the box:
  - `JupiterTestEngine` runs "Jupiter" tests, aka "JUnit 5" tests on the JUnit Platform. Located in the `junit-jupiter-engine` module.
  - `VintageTestEngine` works with JUnit 4 to run "Vintage" tests (JUnit 3 and 4 tests) on the JUnit Platform.
    Located in the `junit-vintage-engine` module.

- The slide does not show every single piece. Internal and less-often-used modules have been omitted. Included in this list is `junit-platform-runner`, which allows older IDEs and tools that support JUnit 4 but do not yet support the JUnit 5 Platform directly, to work with the Platform.

EVALUATION COPY Unauthorized Reproduction or Distribution Prohibited

Copyright © LearningPatterns Inc. All rights reserved.

Writing a Test – First Example

- Create a test class, write a test method annotated with **@Test**
  - Set up a business object and invoke business method(s) on it
  - Make assertions about the results via **static** methods in the **Assertions** class
  - Main JUnit 5 base package is **org.junit.jupiter.api**

```
package demo.junit;

import static org.junit.jupiter.api.Assertions.*;   // see notes
import org.junit.jupiter.api.Test;

class StringTest {  // test class, test methods need not be public

  @Test
  void testLength() {
    String msg = "hello";                  // object under test
    assertEquals(5, msg.length());         // expected, actual
    assertTrue(5 == msg.length());         // alternative assertion
  }
}
```


EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

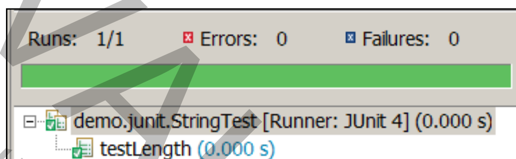
Notes:

- DISCLAIMER: this is obviously a demo example – you would never write tests to verify that Java strings work correctly(!)
- We'll discuss assertions in more detail later, but for now:
  - `assertEquals(`**`expected, actual`**`)` passes when `expected` "equals" `actual`.
    - For **primitive types**, this means **value equality (==)**, for **class types**, this means **`equals()`**. More on assertions later.
  - `assertTrue(`**`condition`**`)` passes when the supplied `condition` is true.
- **Note on static imports**: they can certainly make your code more readable, but are best used sparingly, and only when importing the static members of one (maybe two) classes. Otherwise, readers of your code (including you a few months later) won't know which class a static member comes from, and the code is now **less** clear.
  - Having said that, with regard to JUnit tests, you will almost always see (and use) the static import above. You have our blessing.
- JUnit 4 users: this is a **different package** and **different `@Test` annotation** than those in JUnit 4!

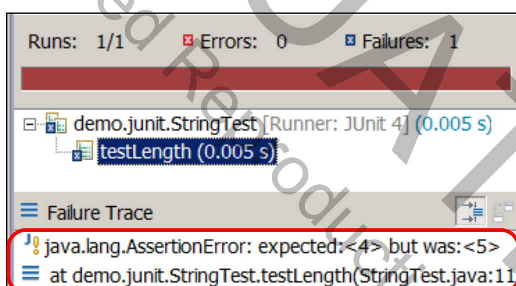
Copyright © LearningPatterns Inc. All rights reserved.

Running Tests in the IDE

- All modern IDEs provide a graphical test runner
  - Usually provide a progress bar: green = pass, red = fail
  - ***"If the bar is green, the code is clean"***



- Failed tests offer "advice" as to why it failed, and where



EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- The failed test above was deliberately triggered with an incorrect assertion.

- The screenshot of the failing test was taken with JUnit 4, very similar in JUnit 5.

Copyright © LearningPatterns Inc. All rights reserved.

Running Tests in Other Environments

- Ant `<junitlauncher>` task in *build.xml* [1]
  ```
  > ant run-tests
  ```
- Maven test phase
  ```
  > mvn test
  ```
- Standalone (command line)
  - JUnit Platform Console Launcher – single JAR with a main-class
- Programmatically
  - Write your own code to the Launcher API
- We'll discuss these other test run environments later [2]

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

[1] This new Ant task is the JUnit 5 counterpart to the older `<junit>` task for JUnit 3 and 4 tests.

- As of this writing, it is nearing completion and appears to be scheduled for inclusion in a later Ant 1.10.x release. See *http://www.mail-archive.com/dev@ant.apache.org/msg46672.html*.
- Although Ant has fallen out of favor, with preference to more modern build systems such as Maven and Gradle, there are still shops using it – and this new `<junitlauncher>` task will be useful for them.

[2] For most of the course, we'll run tests in the IDE. It's easy, fast, and flexible.

- As you write your business code, you should be writing and running tests for that code (at least a subset of them) **all the time** – if you haven't seen a green bar in more than 5 minutes, you're doing it wrong!
- In a later session, we'll work with a Maven project, use the Console Launcher, and look at code written to the Launcher API.

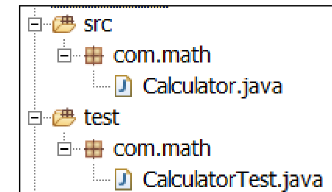
Copyright © LearningPatterns Inc. All rights reserved.

Naming Conventions and Organizing Tests

- `MyClass` is tested by `MyClassTest`
- `MyClassTest` is in **same package** as `MyClass`

why would this matter?

- Parallel ***src*** and ***test*** directories
  - Both branches are checked into source control
  - Package / deploy only the *src* classes
  - Alternatively, two IDE projects – test code in separate project



- For test methods:
  - `myMethod()` is tested by `testMyMethod()`
  - **Must** be annotated by `@Test` to be run as a test case
    - Method name technically doesn't matter, but this historical naming convention is still used – see notes
    - We'll use this convention in the labs, along with others

Session 1: Unit Testing with JUnit 5 18

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- This naming convention for **test classes** has been around since the beginning, and is still by far the most widely used.
  - Other notations include `MyClassTests` (plural) and `TestMyClass`.
  - **NOTE**: JUnit 5 uses a new ***discovery*** mechanism to find test classes, and its default matching pattern finds test classes ending with "Test" or "Tests" – so you are strongly encouraged to use this naming convention (and not `TestMyClass`).

- The naming convention for **test methods** is historical, but still used...though fading over time.
  - In JUnit 3, test runners followed an ad hoc contract, in which all public methods in the test class named "`testSomething`" were run as test cases.
    - This was a requirement. If your test method didn't begin with "test", it didn't get run.
  - That was two major releases (and many years) ago. Starting with JUnit 4, the `@Test` annotation designates test methods – the name of the method is immaterial to the JUnit test runner.
    - Current thinking is that the name of a test method should capture the **intent** and **expected outcome** of the test – and long method names are fine for test methods. We will employ this idea in some of the labs.
    - As one example, some developers use "should" as a test method prefix, e.g., `shouldThrowExceptionWhenSomeCondition()`, `shouldBeEmptyCollection()`, etc.

Copyright © LearningPatterns Inc. All rights reserved.

Lab 1.1 – Getting Started with JUnit

Session 1: Unit Testing with JUnit 5 19

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Java Testing with JUnit 5
Lab Manual – Eclipse

Version 20180418

Copyright © LearningPatterns Inc. All rights reserved.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- Version 20180418

Copyright © LearningPatterns Inc. All rights reserved.

Release Level

- This manual contains instructions for creating and running the labs using the following software packages:
  - **Java 8 SDK**
  - **Eclipse Java EE Edition – Oxygen/4.7.2 or later**
    - JUnit 5 support was added in 4.7.1a
- Instructions are geared towards Windows-based operating systems
  - You should easily be able to adapt to other environments, e.g., Mac OS

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Lab 1.1 – Getting Started with JUnit

Lab 1.1 – Getting Started with JUnit 3

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Lab Synopsis

- **Overview**:
  - Set up our lab environment and the Eclipse IDE
  - Set up an IDE project to use JUnit
  - Test a simple `Calculator` class using some basic assertions
    - We'll discuss assertions in detail after the lab
- **Builds on previous labs**: none
- **Approximate time**: 15-25 minutes

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

Copyright © LearningPatterns Inc. All rights reserved.

Information Content and Task Content

Lab

- **Informational content** is presented in the normal way – the same as in the student manual
  - Like these bullets at the top of the page
- **Tasks** you need to do are in a "Tasks to Perform" box

Tasks to Perform

- Look at these instructions, and notice the different look of the box
  - Make a note of how it looks, as 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_xxx***
  - If not, you'll need to install it – it can be downloaded from: ***http://oracle.com/technetwork/java/javase/downloads***
- OK – now **get out your setup files**, we're ready to start working

Lab 1.1 – Getting Started with JUnit 5

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

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

Extract the Lab Setup Zip File

- To set up the labs, you'll need the course setup zip file
  - It has a name like: ***LabSetup_JavaTesting_yyyyMMdd.zip***
- Our base working directory for this course will be ***C:\StudentWork\JavaTesting***
  - Gets created when we extract the lab setup zip
  - Includes a directory structure and files 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/JavaTesting*** directory structure, containing files that you will need for doing the labs

Lab 1.1 – Getting Started with JUnit 6

Notes:

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

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Copyright © LearningPatterns Inc. All rights reserved.

The Eclipse Platform

- **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 – all stored on the **filesystem**
  - **Workbench**: editors, views, and perspectives – the **UI** you work with
- We will set up the workspace and workbench, then do our lab

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

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

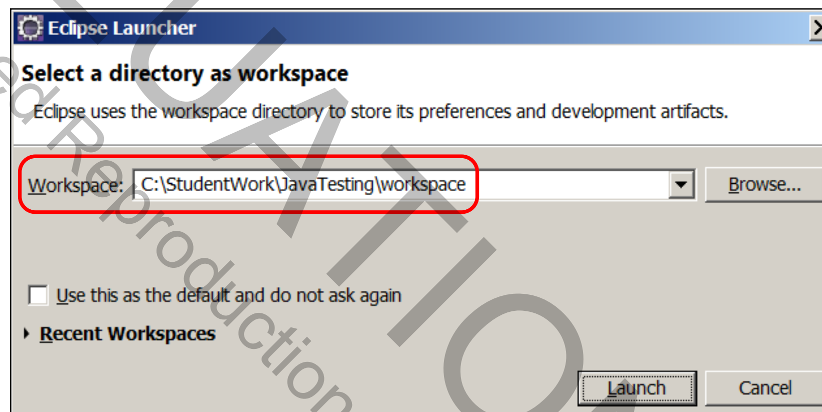
Copyright © LearningPatterns Inc. All rights reserved.

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 (1)
  – Set workspace location to *C:\StudentWork\JavaTesting**workspace*** (2)
  – If a different default workspace location is shown, change it



Lab 1.1 – Getting Started with JUnit 8

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

(1) If the workspace dialog box does **not** appear, that means that Eclipse was previously launched on your system, a workspace directory was chosen, and the "Use this as the default" box was checked.

– In this case, you should change workspace to that shown above via:
File → Switch Workspace → Other. This will open the dialog box.

(2) **NOTE: you must use our provided workspace directory**, as we've supplied starter code and other files for some of the labs here.

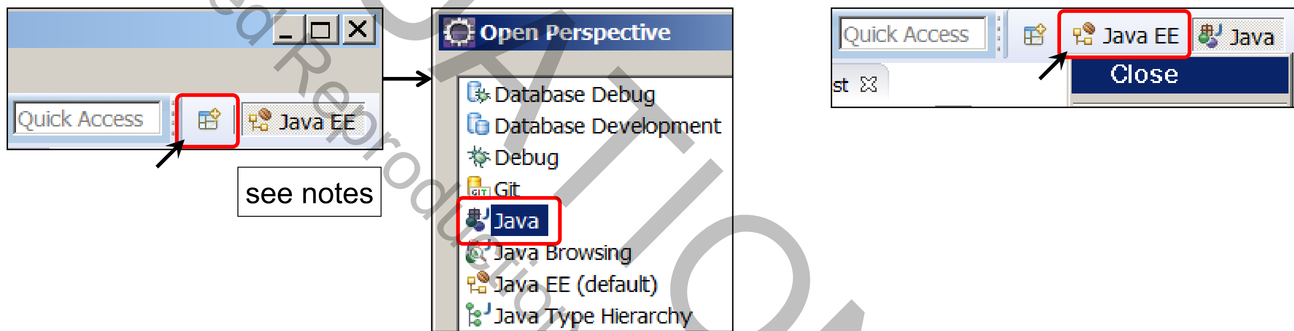
- If Eclipse was installed elsewhere, adjust the path to the Eclipse executable accordingly.
- You can also put a shortcut on your desktop to start Eclipse.
- If you need to download Eclipse, just do a search for "eclipse packages" and go from there.
  – You want the "Eclipse IDE for Java EE Developers."
  – Save the zip file to your computer, and then 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.

Copyright © LearningPatterns Inc. All rights reserved.

Workbench and Java Perspective

Tasks to Perform

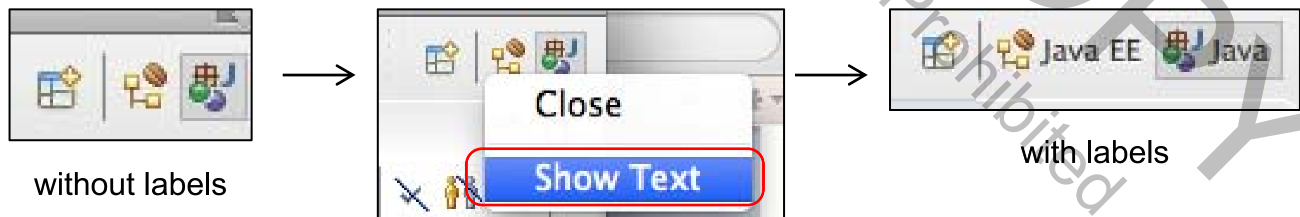
- Close the Welcome screen and **open a Java Perspective**
  - Click the Perspective icon at the top right of the Workbench and select **Java** (see below left)
  - See notes about perspective icons
- **Close the Java EE perspective** by right-clicking its icon at the top right of the Workbench and selecting Close (see below right)
  - The Java EE perspective is the default for the Eclipse Java EE version



Lab 1.1 – Getting Started with JUnit 9

Notes:

- The Eclipse Java EE version opens in the Java EE perspective by default.
- In Eclipse Neon/4.6 and later, the perspective switcher does not show the perspective name by default. To change this, right-click on any perspective icon and **Show Text** – see examples below.



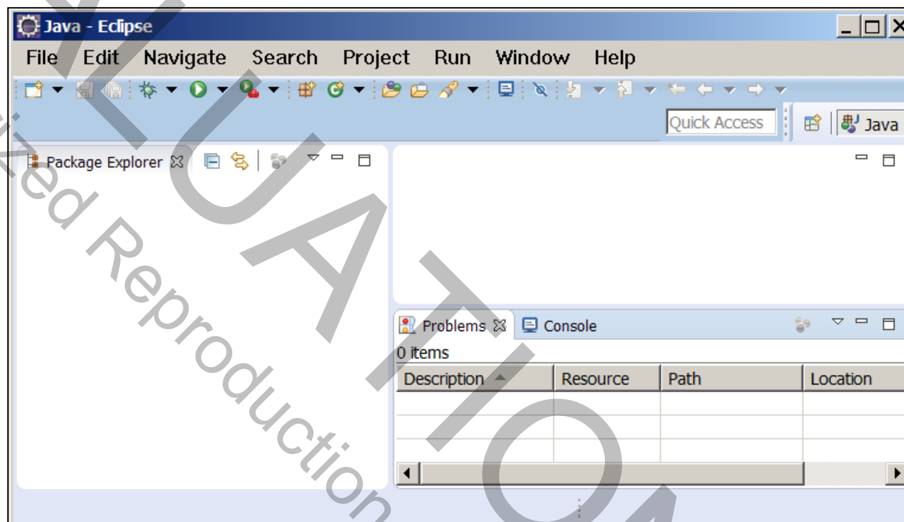
EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Customize the Perspective

Lab

Tasks to Perform

- Unclutter the perspective by closing some views
  - Close the Task List and Outline views
- Open the Console view: **Window → Show View → Console**
- You can save these changes to the perspective [see notes]



Lab 1.1 – Getting Started with JUnit 10

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- An Eclipse ***perspective*** is simply a collection of ***views***, shown as tabbed panes in the Workbench.
  - A perspective defines which views are included, where they are positioned, and what their relative sizes are in the Workbench.

- Eclipse has several predefined perspectives, which can be used as-is, or customized to suit your needs / preferences. There's a good chance you'll want to tweak the ones you're using.
  - To customize a perspective, open a predefined one, e.g., the Java perspective, close the views you don't want, open any views you do want, reposition and/or resize them, and then save the changes via **Window → Perspective → Save Perspective As**.
    - TIP: to preserve the factory-shipped perspective and create a brand new perspective based on it, use a different name, e.g., "Java [YI]", where "[YI]" are Your Initials in square brackets.
    - Window → Perspective → Save Perspective As → Java [YI].

- Your instructor may also recommend additonal customizations, which s/he will go over with you.
  - There are myriad settings, all available via **Window → Preferences**.

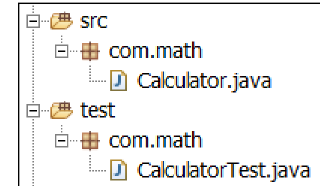
Copyright © LearningPatterns Inc. All rights reserved.

Create Project and Add JUnit Support

Lab

- We'll create a new project, presupplied with some starter code
- Your tasks include:
  - Add JUnit library to project (via JUnit support built in to Eclipse)
  - Finish the test cases and run the tests in the IDE

Tasks to Perform



- Create **new** Java project named **exactly Lab01.1**
  - **Must** name it this to pick up starter code → **Next**
    - See notes about pre-provisioning the starter code
- In the next dialog (Java Settings), click **Libraries** tab
  - Click **Add Library** button, select JUnit and then press Next
    - Select **JUnit 5** (the default) and then click Finish
  - Back in the main wizard, click **Finish** to create the project
  - See notes for adding JUnit support after the fact

Lab 1.1 – Getting Started with JUnit 11

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- **IMPORTANT NOTE**: you **must** name the project **Lab01.1**, in order to pick up the starter code we've provided for this lab.
  - That's because the starter code was pre-provisioned 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.
- After adding the JUnit library to the project, any compile errors you might see should go away. See your instructor if they don't, or if you have any problems.
- If you ever forget to add JUnit during initial project creation, it can easily be added later.
  - Right-click on project and choose **Build Path → Add Libraries → JUnit → JUnit 5**, or
  - **Build Path → Configure Build Path → Libraries tab → Add Library → JUnit → JUnit 5**.
  - Also note: you can completely ignore this step, and when you create the first test class in a project via **New → JUnit Test Case**, the wizard will prompt you to add JUnit support to the project (which of course you say "yes" to).

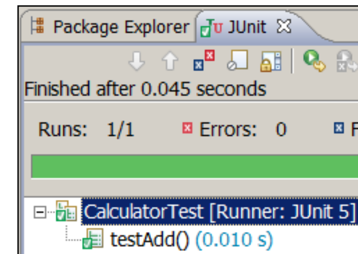
Copyright © LearningPatterns Inc. All rights reserved.

Business Class and Sample Test

Lab

Tasks to Perform

- Note the structure of the project and the names of the classes
  - Parallel *src* and *test* directories
  - Test classes and business classes in same package
- Review the **Calculator** class – yes, it's dumb-simple BUT:
  - Understand its API and what to **expect** from its methods
    - This forms the basis for your test cases
  - This simple class gives us a good example to start with
- Review the **CalculatorTest** test class
  - Run its test cases (methods) via right-click → **Run As → JUnit Test** [see notes]
  - An Eclipse JUnit view appears, with a green bar



EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- You can right-click on the *CalculatorTest.java* file in Package Explorer, or just right-click anywhere in the editor.

Copyright © LearningPatterns Inc. All rights reserved.

Finish the Test Methods

Tasks to Perform

- Write additional test methods for the `Calculator` class, following the example `testAdd()` method
  - **testDivide()**

```
assertEquals(2.5, calc.divide(5, 2));            // expected, actual
```

    - Run the tests as before: right-click → **Run As → JUnit Test**

  - Still working in `testDivide()`, try this one:

```
assertEquals(.571428, calc.divide(4, 7));       // FAIL
```

    - The expected value is a six-digit repeating decimal – what to do?

```
assertEquals(.571428, calc.divide(4, 7), .001); // include a delta
```

  - **testIsEven()**
    - Use **assertTrue()**, passing in a should-be-true condition – for example:

```
assertTrue(calc.isEven(10));              // also try an assertFalse()
```


EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- The `assertEquals()` method is heavily overloaded, with **30** variations. We'll get into the details after the lab, but what's happening here is that there are two versions for comparing `double`s:
  - `assertEquals (double expected, double actual)`
  - `assertEquals (double expected, double actual, double delta)`
    - Asserts that `expected` and `actual` are equal within the given `delta`.

- Interestingly, JUnit 4 also had these two flavors of floating point assertions, but the two-argument version was deprecated, recommending the delta version as the better alternative.

- The examples above use method in-lining for the "actual" value in the assertions – this is common, but of course not required. For example, the `testDivide()` method could also be written as:

```
@Test
public void testDivide() {
  Calculator calc = new Calculator();
  double result = calc.divide(5, 2);
  assertEquals(2.5, result);
}
```

Copyright © LearningPatterns Inc. All rights reserved.

Scoping the Test Run

Lab

- You can run only **one test**, all the way up to **all tests** in a project
  - It's simply a function of where you right-click

Tasks to Perform

- Try it out
  - Right-click ***CalculatorTest.java*** in Package Explorer (or anywhere in the editor for it) → all tests in **that class**
  - Right-click **`com.math`** package → all tests in all classes in **that package**
  - Right-click ***test*** source folder → all tests in all classes in **all packages** in *test* folder
- To run just a single test:
  - Select the **method name** in the editor
    - Just double-click on it – actually, just a single-click anywhere on it will do
  - Then right-click and run test
    - You should see **only** the results of this single test method

```
@Test
public void testSortLambda() {
  System.out.println("testSortLambda:");
```


Notes:

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

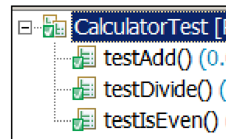
Copyright © LearningPatterns Inc. All rights reserved.

Test Setup and Order of Execution

Lab

- You want a "clean" new instance of `Calculator` for each test
  - This class is stateless (no instance variables), but principle still applies
  - We thus instantiate one in each test method [`new Calculator()`]
    - Simple, but the result is redundant setup code in each test method
    - We'll address this later
- Tests run in **isolation**, and in **no guaranteed order**
  - Even though you may think you see a pattern

Tasks to Perform



- In the JUnit view, note the order of test execution
  - It **may** appear to follow the order of methods in test class
- Write a new method, `testSomething`, as the first one in the class
  - Rerun all tests → what do you notice? [see notes]

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- GOTCHA – there **is** nothing to notice. Tests should run in **isolation** and in **no guaranteed order**.
  - Any pattern you **think** you see **cannot** and **should not** be relied upon. This is a fundamental principle of unit testing. More info coming.

Copyright © LearningPatterns Inc. All rights reserved.

JUnit API and Documentation

Lab

Tasks to Perform

referred to as **<junit>**

- Find the supplied JUnit distribution in the lab setup, in ***StudentWork/JavaTesting/Software/junit-5.1.0***
  - Eclipse uses its own built-in version
  - Maven downloads the JARs from a repository
  - But a standalone distribution is good for API docs, and just to have
- Open and bookmark the JUnit Javadoc
  - Using a file browser (Windows Explorer), locate the base ***doc*** directory ***Software/junit-5.1.0/doc***
  - From here, drill down into ***docs/current***
    - Go one level deeper into the ***api*** directory and then open ***index.html***
    - You should **bookmark** this start page for the JUnit API doc
    - Note also the ***release-notes*** and ***user-guide*** directories
  - Also available online at ***http://junit.org/junit5***

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

Notes:

- We have found **no source** for an all-in-one zip for the JUnit 5 Javadoc that can be downloaded and extracted locally.
  - Each module in JUnit 5 is available at Maven Central, and each provides *jar*, *javadoc.jar*, and *sources.jar* files. Okay, but there are **many** modules that we need to look at.
  - We started down this path, downloading the *javadoc.jar* files for `junit-jupiter-api` and `junit-jupiter-params`, but quickly realized this was an inferior solution.
  - Ultimately, we decided to pull it off *junit.org* directly, relativizing all the links and making a few tweaks. This seems to work perfectly well.
  - If you do hit a broken link or something, just start over at the *index.html* file that you bookmarked.

Copyright © LearningPatterns Inc. All rights reserved.

EVALUATION COPY
Unauthorized Reproduction or Distribution Prohibited

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

1-06-00097-000-05-25-21