CCIS كلية علوم الحاسب والمعلومات COLLEGE OF COMPUTER & INFORMATION SCIENCES

جامعة الأمير سلطان PRINCE SULTAN UNIVERSITY

Unit Testing and JUnit

SE401: Software Quality Assurance and Testing

Outline

- Unit Testing and JUnit
- Assertion Methods
- JUnit Best Practices
    - JUnit documentation
        - http://junit.org/junit5
        - https://junit.org/junit5/docs/snapshot/user-guide/
    - An introductory tutorial
        - http://www.vogella.com/tutorials/JUnit/article.html
    - Using JUnit in Eclipse
        - https://www.eclipse.org/community/eclipse_newsletter/2017/october/article5.php
        - https://www.educative.io/courses/java-unit-testing-with-junit-5/B892KY261z2
    - How to use JUnit with NetBeans
        - https://testingandlearning.home.blog/2019/01/30/how-to-use-junit-with-netbeans/

Unit Testing and JUnit

Unit Testing

- Testing of an individual software unit
  - usually a class & its helpers
- Focus on the functions of the unit
  - functionality, correctness, accuracy
- Usually carried out by the developers of the unit
  - can use black-box and white-box techniques to design test cases

Unit Testing

- Unit testing: Looking for errors in a subsystem in isolation.
  - Generally a "subsystem" means a particular class or object.
  - The Java library JUnit helps us to easily perform unit testing.
- The basic idea:
  - For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.
- JUnit provides "assert" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

Why JUnit

- Allows you to write code faster while increasing quality
- Elegantly simple
- Check their own results and provide immediate feedback
- Tests are inexpensive
- Increase the stability of software
- Developer tests
- Written in Java
- Free
- Gives proper understanding of unit testing

JUnit

- JUnit helps the programmer:
  - Define and execute tests and test suites
  - Formalize requirements and clarify architecture
  - Write and debug code
  - Integrate code and always be ready to release a working version
- What JUnit does
  - JUnit runs a suite of tests and reports results

JUnit

- JUnit is a framework for writing unit tests
  - A unit test is a test of a single class
    - A test case is a single test of a single method
    - A test suite is a collection of test cases
- Unit testing is particularly important when software requirements change frequently
  - Code often has to be refactored to incorporate the changes
  - Unit testing helps ensure that the refactored code continues to work

A JUnit test class

```
import static
  org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class name {
    ...

    @Test
    void name() {   // a test case method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

The structure of a test method

- A test method doesn't return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an AssertionFailedError
- The JUnit framework catches the error and deals with it; you don't have to do anything

Test suites

- In practice, you want to run a group of related tests (e.g. all the tests for a class)
- To do so, group your test methods in a class with annotations @SelectPackages or @SelectClasses

Organize The Tests

- Create test cases in the same package as the code under test
- For each Java package in your application, define a TestSuite class that contains all the tests for validating the code in the package
- Define similar TestSuite classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application
- Make sure your build process include the compilation of all tests

JUnit Best Practices

- Separate production and test code
- But typically in the same packages
- Compile into separate trees, allowing deployment without tests
- Don’t forget OO techniques, base classing
- Test-driven development
  1. Write failing test first
  2. Write enough code to pass
  3. Refactor
  4. Run tests again
  5. Repeat until software meets goal
  6. Write new code only when test is failing

JUnit Best Practices

- Tests need *failure atomically* (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.

Problems with unit testing

- JUnit is designed to call methods and compare the results they return against expected results
  - This ignores:
    - Programs that do work in response to GUI commands
    - Methods that are used primarily to produce output
- Heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that just return results, without side effects (such as printing), are simpler, more general, and easier to reuse

An Introduction to JUnit
Part 1: The Basics

JUnit – Java Unit Testing Tool

- A unit testing tool for Java programs
  - JUnit home page: http://junit.org
- A simple framework to write repeatable tests
  - Test cases, test suites, assertions, etc.,
- Automated execution of test suites
  - Run all test cases, generate reports
- Development methodology neutral
  - Often used in agile development/test-driven development

JUnit 5

- Requires Java 8 (or higher) at runtime
  - Can still test code that has been compiled with previous versions of the JDK
- Supported by popular IDEs
  - IntelliJ IDEA
  - Eclipse
  - NetBeans
  - Visual Studio Code
- Supported by build tools
  - Gradle
  - Maven
  - Ant

Running JUnit

- JUnit has been integrated into most IDE's
  - We will use the latest Eclipse/NetBeans IDE
  - Download and install Eclipse/NetBeans IDE for Java Developers
- JUnit can also be run independently
  - Command-line, builder server
  - Using a simple build tool *Ant*

You can use both methods of running JUnit.

Test Case Verdicts

A *verdict* is the result of executing a single test case.

Pass

- The test case execution was completed
- The function being tested performed as expected

Fail

- The test case execution was completed
- The function being tested did *not* perform as expected

Error

The test case execution was not completed, due to an unexpected event, exceptions, or improper set up of the test case, etc.

JUnit Tests

- A *JUnit test* is represented as a class (test class).
- Each *test case* is a method in a test class.
- A typical test case does the following
  - create some objects/data to test
  - do something interesting with the objects
  - determine pass or fail based on the results
- A *test suite* may consist of multiple test classes.

JUnit Assertions

- *Assertions* are Boolean expressions
  - An *AssertionFailedError* is thrown if the assertion is false
- Can check for many conditions, such as
  - equality of objects and values
  - identity of references to objects
- Determine the test case verdict
  - **Pass:** all assertions are true
  - **Fail:** one or more assertions are false

A Simple JUnit Test Case

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
   Value v1 = new Value();

   v1.setName("Y");

   String expected = "Y";
   String actual = v1.getName();

   Assert.assertEquals(expected, actual);
}
```

A Simple JUnit Test Case

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
   Value v1 = new Value();

   v1.setName("Y");

   String expected = "Y";
   String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

Identify this Java method as a test case

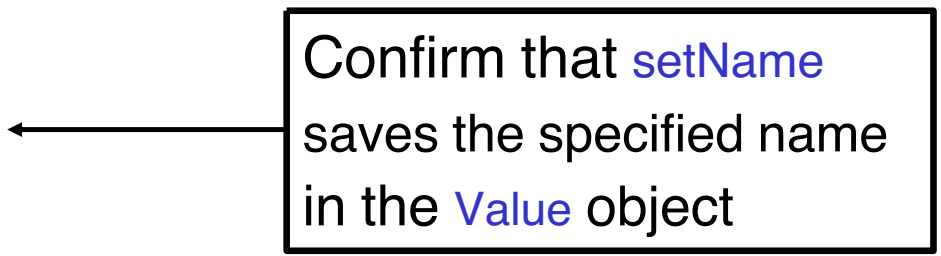
A Simple JUnit Test Case

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
   Value v1 = new Value();

   v1.setName("Y");

   String expected = "Y";
   String actual = v1.getName();

   Assert.assertEquals(expected, actual);
}
```



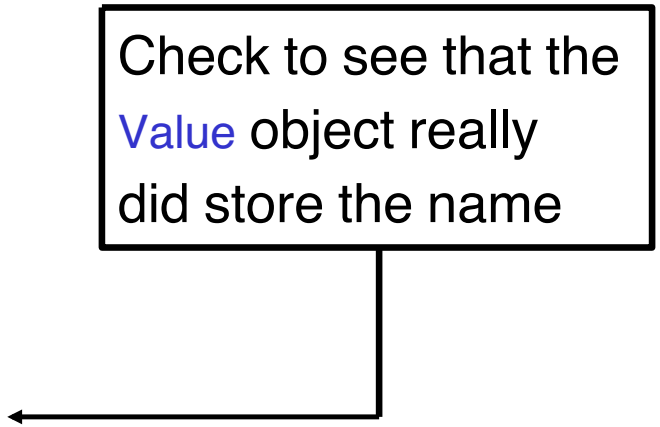
A Simple JUnit Test Case

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
   Value v1 = new Value();

   v1.setName("Y");

   String expected = "Y";
   String actual = v1.getName();

   Assert.assertEquals(expected, actual);
}
```



A Simple JUnit Test Case

```
/** Test of setName() method, of class Value  */
@Test
public void createAndSetName() {
   Value v1 = new Value();

   v1.setName("Y");

   String expected = "Y";
   String actual = v1.getName();

    Assert.assertEquals(expected, actual);
}
```

Assert that the expected and actual should be equal. If not, the test case should fail.

Organization of JUnit Test

- Each test method represents a single test case
  - can independently have a verdict (pass, error, fail).
- The test cases for a *class under test* (CUT) are usually grouped together into a test class.
- Naming convention:
  - Class under test: Value
  - JUnit test for the class: ValueTest
  - Test classes are sometimes placed in a separate package.

Using JUnit in Eclipse/NetBeans

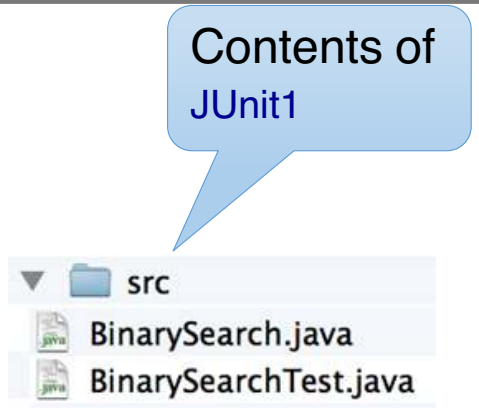
- Download and install JDK

  Download and install Eclipse/NetBeans IDE for Java Developers

  - JUnit is included in Eclipse/NetBeans

Run JUnit in Eclipse: An Example

- Download the *Sample Code*
  - JUnit1.rar
  - Unzip to the Eclipse workspace folder
    - A subfolder named JUnit1
- The example contains
  - BinarySearch.java
  - BinarySearchTest.java



[see JUnit1.rar]

The Example Program: The Class Under Test

```
public class BinarySearch {
  public static int search(int[] a, int x) {
        …
  }

  public static int checkedSearch(int[] a, int x) {
        …
  }
}
```

The JUnit Test

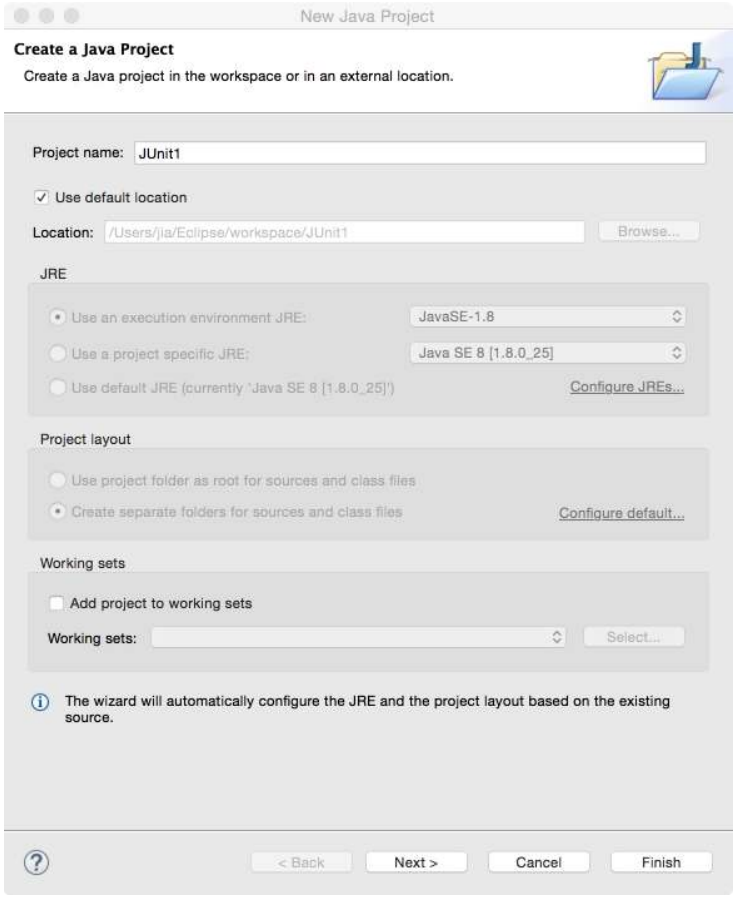
```
public class BinarySearchTest {
  @Test
  public void testSearch1() {
        int[] a = { 1, 3, 5, 7 };
        assertTrue(search(a, 3) == 1);
  }
```

The JUnit Test (cont'd)

```
  @Test
  public void testSearch2() {
            int[] a = { 1, 3, 5, 7 };
            assertTrue(search(a, 2) == -1);
  }
  @Test
  public void testCheckedSearch1() { … }
  @Test
  public void testCheckedSearch2() { … }
  @Test
  public void testCheckedSearch3() { … }
}
```

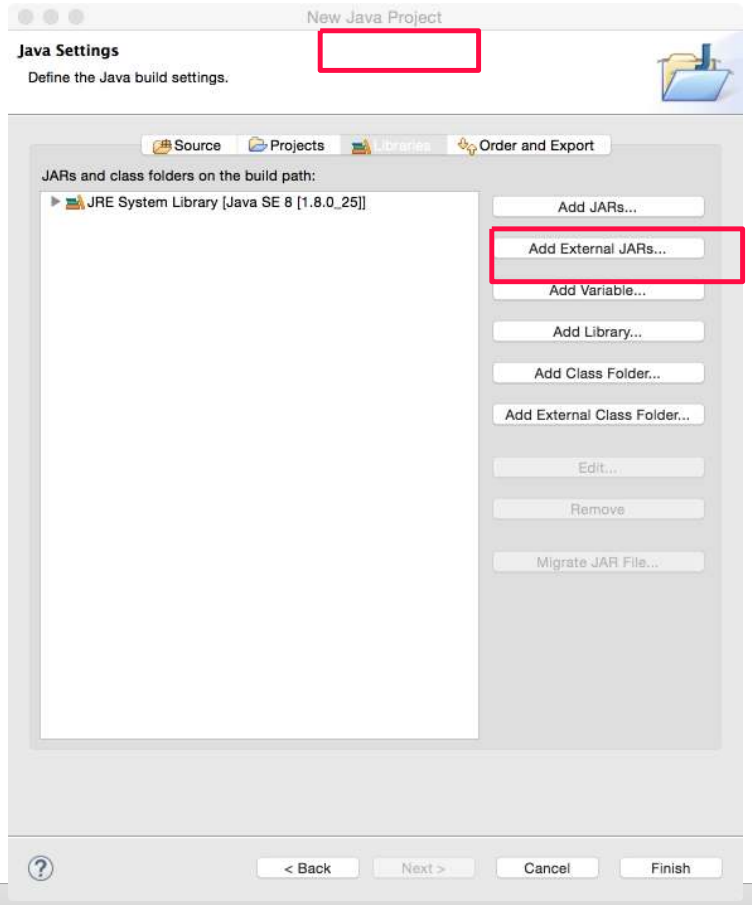
Run JUnit in Eclipse: An Example

- Start Eclipse IDE
- New Java Project
  - Project name: JUnit1
  - **Important**: The project name matches the name of the folder that contains the sample code



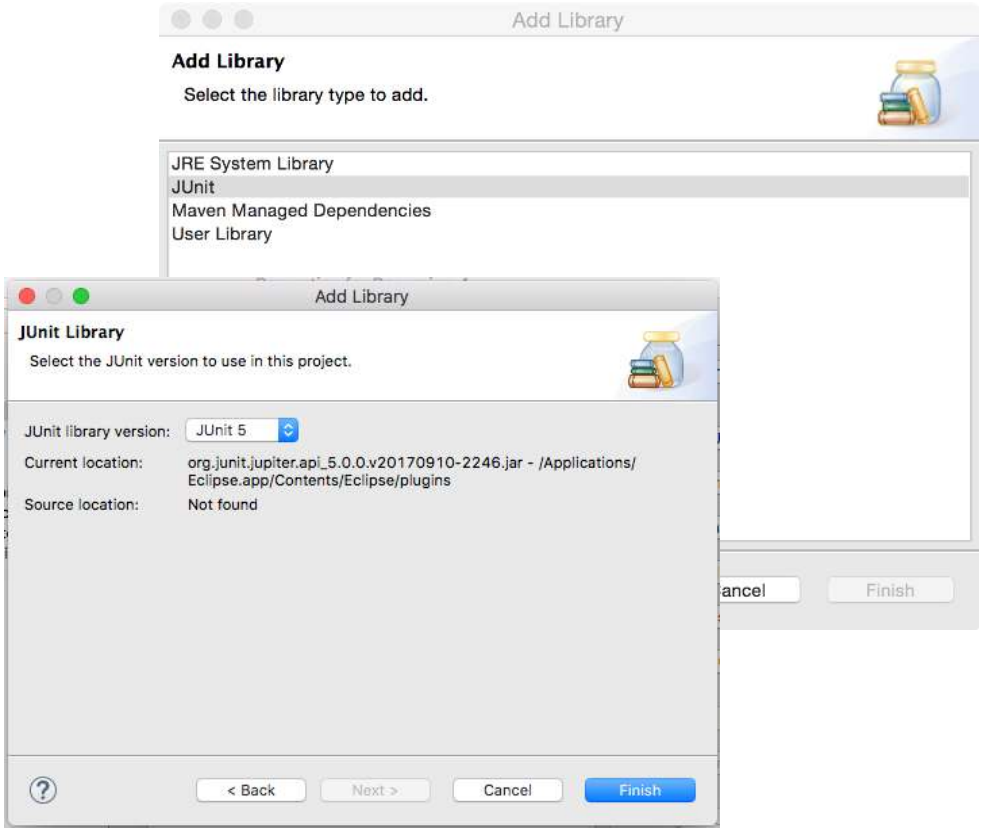
Run JUnit in Eclipse: An Example

- Click “Next”
- Java Settings
  - Click “Libraries”
  - Click “Add Library …”



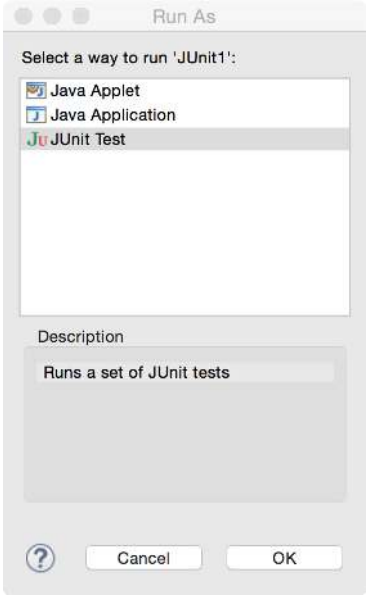
Run JUnit in Eclipse: An Example

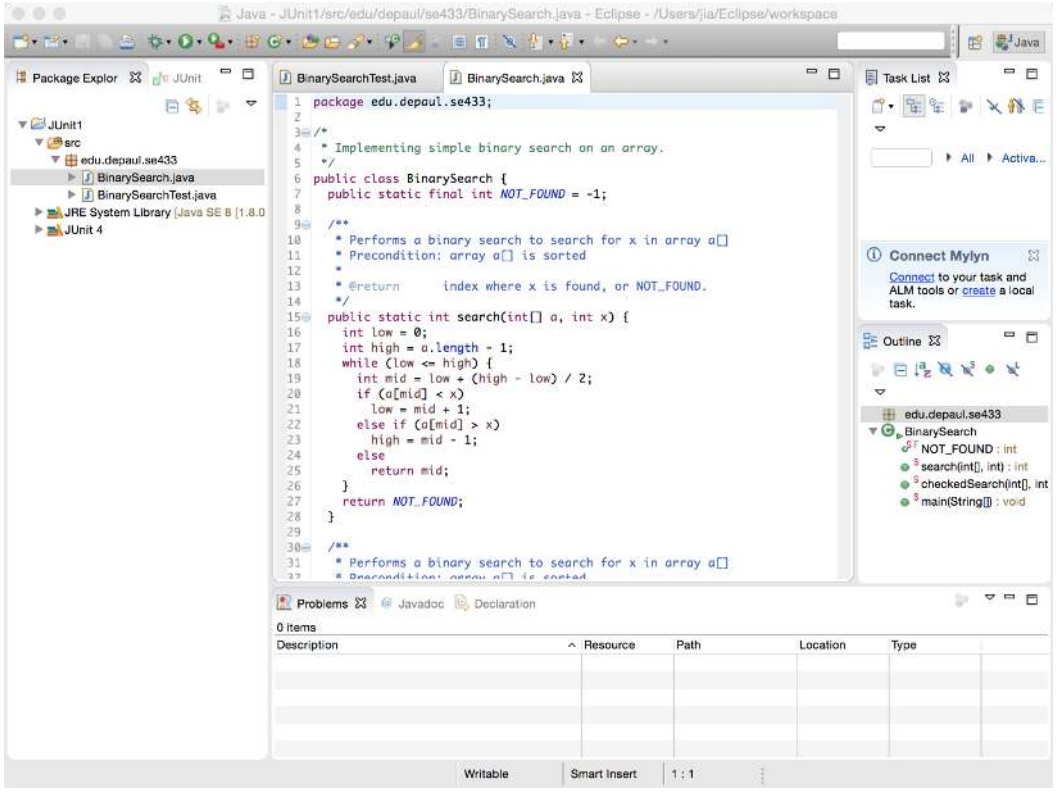
- Add Library
  - Choose “JUnit”
- JUnit Library
  - Choose “JUnit 5”
- Click “Finish”



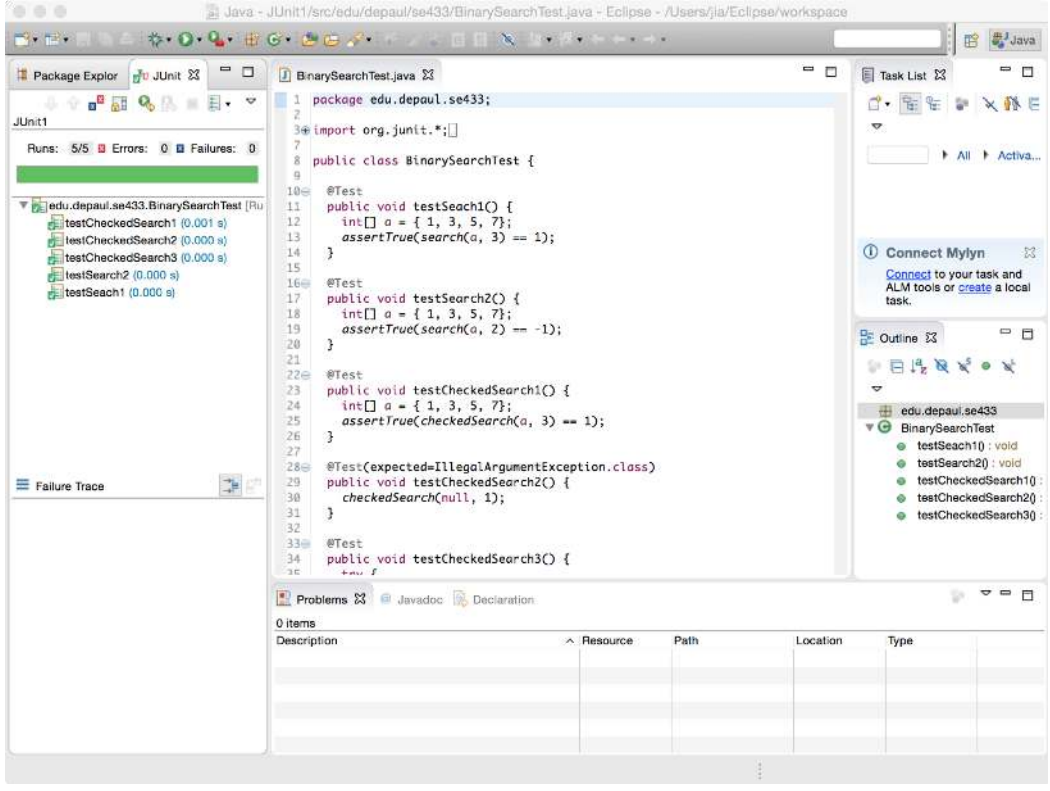
Run JUnit in Eclipse: An Example

- Run as
  - JUnit test





Run JUnit in Eclipse: An Example



Assertion Methods

Assertions in Test Cases

During execution of a test case:

- If an assertion is **true**,
  - Execution continues
- If any assertion is **false**,
  - Execution of the test case stops
  - The test case fails
- If an *unexpected* exception is encountered,
  - The verdict of the test case is an error.
- If all assertions were true,
  - The test case *passes*.

Assertion Methods: Boolean Conditions

- Static methods defined in org.junit.Assert
- Assert a Boolean condition is true or false

  assertTrue(*condition*)

  assertFalse(*condition*)

- Optionally, include a failure message

  assertTrue(*message, condition*)

  assertFalse(*message, condition*)

- Examples

```
assertTrue(search(a, 3) == 1);
assertFalse("Failure: 2 is not in array.", search(a, 2) >= 0);
```

Assertion Methods: Null Objects

- Assert an object references is null or non-null

  assertNull(*object*)

  assertNotNull(*object*)

- With a failure message

  assertNull(*message, object*)

  assertNotNull(*message, object*)

- Examples

```
assertNotNull(”Should not be null.", new Object());
assertNull(”Should be null.", null);
```

Assertion Methods: Object Identity

- Assert two object references are identical

  assertSame(*expected, actual*)

  - True if: expected == actual

  assertNotSame(*expected, actual*)

  - True if: expected != actual

- The order does not affect the comparison,
  - But, affects the message when it fails

- With a failure message

  assertSame(*message*, *expected, actual*)

  assertNotSame(*message*, *expected, actual*)

Assertion Methods: Object Identity

- Examples

```
assertNotSame("Should not be same.",
                    new Object(), new Object());

Integer num1 = Integer.valueOf(2013);
assertSame("Should be same.", num1, num1);

Integer num2 = Integer.valueOf(2014);
assertSame("Should be same.", num1, num2);
```

```
java.lang.AssertionError:
Should be same. expected same:<2013> was not:<2014>
```

Assertion Methods: Object Equality

- Assert two objects are equal:

  assertEquals(expected, actual)

  - True if: expected.equals(actual)
  - Relies on the equals() method
  - Up to the class under test to define a suitable equals() method.

- With a failure message

  assertEquals(*message*, *expected, actual*)

Assertion Methods: Object Equality

- Examples

```
assertEquals("Should be equal.", "JUnit", "JUnit");

assertEquals("Should be equal.", "JUnit", "Java");
```

```
org.junit.ComparisonFailure:
Should be equal. expected:<J[Unit]> but was:<J[ava]>
```

Assertion Methods: Equality of Arrays

- Assert two arrays are equal:

  assertArrayEquals(expected, actual)

  - arrays must have same length
  - Recursively check for each valid index i,

    assertEquals(expected[i],actual[i])

    or

    assertArrayEquals(expected,actual)

- With a failure message

  assertArrayEquals(*message*, *expected, actual*)

Assertion Methods: Equality of Arrays

- Examples

```
int[] a1 = { 2, 3, 5, 7 };
int[] a2 = { 2, 3, 5, 7 };
assertArrayEquals("Should be equal", a1, a2);

int[][] a11 = { { 2, 3 }, { 5, 7 }, { 11, 13 } };
int[][] a12 = { { 2, 3 }, { 5, 7 }, { 11, 13 } };
assertArrayEquals("Should be equal", a11, a12);
```

Assertion Methods: Floating Point Values

- For comparing floating point values (double or float)
  - assertEquals requires an additional parameter **delta**.

assertEquals(*expected, actual, delta*)
assertEquals(*message*, *expected, actual, delta*)

- The assertion evaluates to true if

Math.abs(expected – actual) <= delta

- Example:

الفرق يكون

```
double d1 = 100.0, d2 = 99.99995;
assertEquals("Should be equal within delta.", d1, d2, 0.0001);
```

Exception Testing

- A.k.a., robustness testing
- The expected outcome of a test is an exception.

```
public static int checkedSearch(int[] a, int x) {
  if (a == null || a.length == 0)
    throw
    new IllegalArgumentException("Null or empty array.");
  ...
}
```

```
checkedSearch(null, 1);
```

Exception Testing: Specify the Excepted Exception

- Specify an expected exception in a test case
  - A particular class of exception is expected to occur

```
@Test
  void exceptionTesting() {
    Exception exception = assertThrows(ArithmeticException.class, () ->
          calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
  }
```

- The verdict
  - Pass: if the expected exception is thrown
  - Fail: if no exception, or an unexpected exception

Exception Testing: The fail() Assertion

- Assertion methods
  - fail()
  - fail(*message*)
- Unconditional failure
  - i.e., it always fails if it is executed
- Used in where it should not be reached
  - e.g., after a statement, in which an exception should have been thrown.

Exception Testing: Use fail() Assertion

- Catch exceptions, and use fail() if not thrown

```
@Test
public void testCheckedSearch3() {
 try {
            checkedSearch(null, 1);
            fail("Exception should have occurred");
 } catch (IllegalArgumentException e) {
            assertEquals(e.getMessage(), "Null or empty array.");
 }
}
```


- Allows
  - inspecting specific messages/details of the exception
  - distinguishing different types of exceptions

Summary: Key Concepts

- Unit testing refers to the practice of testing certain functions and areas – or units – of our code. This gives us the ability to verify that our functions work as expected.
- Testing needs to be thorough
- Eclipse/NetBeans provides a platform for doing unit tests using JUnit as a built-in feature.

JUnit Best Practices

JUnit Best Practices

- Each test case should be independent.
- Test cases should be independent of execution order.
- No dependencies on the state of previous tests.

JUnit Test Fixtures

- The context in which a test case is executed.
- Typically include:
  - Common objects or resources that are available for use by any test case.
- Activities to manage these objects
  - Set-up: object and resource allocation
  - Tear-down: object and resource de-allocation

Set-Up

- Tasks that must be done prior to each test case
- Examples:
  - Create some objects to work with
  - Open a network connection
  - Open a file to read/write

Tear-Down

- Tasks to clean up after execution of each test case.
- Ensures
  - Resources are released
  - the system is in a known state for the next test case
- Clean up should not be done at the end of a test case,
  - since a failure ends execution of a test case at that point

Method Annotations for Set-Up and Tear-Down

- @BeforeEach annotation: set-up
  - code to run before each test case.
- @AfterEach annotation: Teardown
  - code to run after *each* test case.
  - will run regardless of the verdict, even if exceptions are thrown in the test case or an assertion fails.
- Multiple annotations are allowed
  - all methods annotated with @BeforeEach will be run before *each* test case
  - but no guarantee of execution order

Example: Using a File as a Test Fixture

```
public class OutputTest {
   private File output;

   @BeforeEach
   public void createOutputFile() {
     output = new File(...);
   }

   @AfterEach
   public void deleteOutputFile() {
      output.close();
      output.delete();
   }

   @Test
   public void test1WithFile() {
      // code for test case
      …
   }

   @Test
   public void test2WithFile() {
      // code for test case
      …
   }
}
```

Method Execution Order

1. createOutputFile()
2. test1WithFile()
3. deleteOutputFile()
4. createOutputFile()
5. test2WithFile()
6. deleteOutputFile()

Not guaranteed:

test1WithFile runs before test2WithFile

Once-Only Set-Up

- @BeforeAll annotation on a *static* method
  - one method only
- Run the method *once only* for the entire test class
  - *before* any of the tests, and
  - *before* any @BeforeEach method(s)
- Useful for starting servers, opening connections, etc.
  - No need to reset/restart for each test case
  - Shared, non-destructive

```
@BeforeAll
public static void anyName() {
    // class setup code here
}
```

Once-Only Tear-Down

- @AfterAll annotation on a *static* method
  - one method only
- Run the method *once only* for the entire test class
  - *after* any of the tests
  - *after* any @AfterEach method(s)
- Useful for stopping servers, closing connections, etc.

```
@AfterAll
public static void anyName() {
   // class clean up code here
}
```

Timed Tests

- Useful for simple performance test
  - Network communication
  - Complex computation
- The @Timeout annotation
  - Time unit defaults to seconds but is configurable

```
@Test
@Timeout(5)
public void testLengthyOperation() {
      ...
}
```

in seconds

if she asked you to use the method and she specified a time, you need to convert it to seconds.

- The test fails
  - if timeout occurs before the test method completes

JUnit 5 Unit Testing Framework

- JUnit 5 Documentation
- Use JUnit 5 annotations to mark test methods

Annotation	Description
@Test public void method()	The annotation @Test identifies that a method is a test method.
@BeforeEach public void method()	Will execute the method before each test. Can prepare the test environment (e.g. read input data, initialize the class).
@AfterEach public void method()	Will execute the method after each test. Can cleanup the test environment (e.g. delete temporary data, restore defaults).

JUnit 5 Unit Testing Framework

Annotation	Description
@BeforeAll public void method()	Will execute the method once, before the start of all tests. Can be used to perform time intensive activities, for example to connect to a database.
@AfterAll public void method()	Will execute the method once, after all tests have finished. Can be used to perform clean-up activities, for example to disconnect from a database.
@Timeout(5)	Fails if the method takes longer than 5 seconds.
@Timeout(value = 100, unit = TimeUnit.MILLISECONDS)	Fails if the method takes longer than 100 milliseconds

Parameterized Tests

- Repeat a test case multiple times with different data
- Define a parameterized test
  - Declared just like regular @Test methods but use the @ParameterizedTest annotation instead
  - Must declare at least one source that will provide the arguments for each invocation
  - Consume the arguments in the test method

Parameterized Test Example

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "able was I ere I saw elba" })
void palindromes(String candidate) {
    assertTrue(StringUtils.isPalindrome(candidate));
}
```