Tutorial 6: Testing

Due   Mar 12 by 5:30pm   **Points**   10   **Submitting**   a file upload
Available   until Mar 12 at 6:30pm

This assignment was locked Mar 12 at 6:30pm.

Testing

**There are two parts to this tutorial.** The first part involves setting up and using JUnit. The second part involves using Selenium to perform web testing.

Do part 1, which is shown below before doing part 2, which can be found [here](https://docs.google.com/document/d/1bDwVsBeHTqKVWbuGF67HHEoqmsKMtOnh4eubgNgUkLs/edit?usp=sharing) **(https://docs.google.com/document/d/1bDwVsBeHTqKVWbuGF67HHEoqmsKMtOnh4eubgNgUkLs/edit?usp=sharing)** .

Executive Summary

A robust set of tests is essential in helping to ensure that new changes do not negatively impact how a software system functions. One of the most popular tools for writing unit tests is the jUnit testing framework. jUnit provides an extremely simple yet powerful way to define and run tests for Java software. In this lab, we will learn how to use jUnit.

The tutorial is based on materials from Dr. Miryung Kim and Dr. Adnan Aziz.

You can use the command line, Eclipse, or IntelliJ IDEA to complete this tutorial.

Getting jUnit

jUnit is available as a single Java archive (jar) file which contains everything you need to both define and execute tests. The jUnit jar is included with Eclipse's Java Development Tools, IntelliJ IDEA's Java Library or can be downloaded from the project's **[repository](https://github.com/junit-team/junit5)**   **(https://github.com/junit-team/junit5)** .

jUnit Basics

Tests in jUnit are written using Java code. Starting with version 4.0 of jUnit, all that is needed to write a test is to annotate a method containing test code with the "Test" annotation available from the "org.junit" package. When jUnit is run on a class, it will call all of the methods it finds annotated with the "Test" annotation. Any given test method passes if it returns without raising an exception.

The concept of annotations may be new to some so we'll digress for a moment to describe them. Annotations are a facility Java provides that allow metadata to be added to the compilation output of a

class. This metadata is then accessible within the virtual machine as part of the class. Tools such as jUnit access this metadata to perform their given tasks.

This is all actually simpler than it sounds, so we'll consider an example that tests the following simple class:

```java
public class Philadelphia {
    public static boolean isItSunny() {
        return true;
    }
}
```

```java
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;
public class PhiladelphiaTest {
    @Test
    public void testIsItSunny() {
        assertTrue(Philadelphia.isItSunny());
    }
}
```

Note that we've imported the "org.junit.jupiter.api.Test" annotation and applied it to the test method. Also, notice that we've used one of the static helper methods found in the "org.junit.jupiter.api.Assertions" class to assert that it is always sunny in Philadelphia. There are quite a few such helper methods for different types of assertions. jUnit tests typically make heavy use of these assertion helpers, so you'll want to become familiar with what is available. When this test is run, it will pass only if the assertion holds (which in this example always will).If you wanted to write a test to check the output of the "isItSunny" method it would look something like:

jUnit5.0 Basic

[Documentation](https://junit.org/junit5/docs/5.1.0-M1/user-guide/) **[(https://junit.org/junit5/docs/5.1.0-M1/user-guide/)](https://junit.org/junit5/docs/5.1.0-M1/user-guide/)**

Starting with version 5.0 of jUnit, JUnit 5 is composed of several different modules from three different sub-projects.

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

The **JUnit Platform** serves as a foundation for **[launching testing frameworks (https://junit.org/junit5/docs/5.1.0-M1/user-guide/#launcher-api)](https://junit.org/junit5/docs/5.1.0-M1/user-guide/#launcher-api)** on the JVM. It also defines the `TestEngine` `[(https://junit.org/junit5/docs/5.1.0-M1/api/org/junit/platform/engine/TestEngine.html)](https://junit.org/junit5/docs/5.1.0-M1/api/org/junit/platform/engine/TestEngine.html)` API for developing a testing framework that runs on the platform. Furthermore, the platform provides a **[Console Launcher (https://junit.org/junit5/docs/5.1.0-M1/user-guide/#running-tests-console-launcher)](https://junit.org/junit5/docs/5.1.0-M1/user-guide/#running-tests-console-launcher)** to launch the platform from the command line and build plugins for **[Gradle (https://junit.org/junit5/docs/5.1.0-M1/user-guide/#running-tests-build-gradle)](https://junit.org/junit5/docs/5.1.0-M1/user-guide/#running-tests-build-gradle)** and **[Maven (https://junit.org/junit5/docs/5.1.0-M1/user-](https://junit.org/junit5/docs/5.1.0-M1/user-)**

guide/#running-tests-build-maven) as well as a **JUnit 4 based Runner (https://junit.org/junit5/docs/5.1.0-M1/user-guide/#running-tests-junit-platform-runner)** for running any `TestEngine` on the platform.

JUnit Jupiter is the combination of the new **programming model** **(https://junit.org/junit5/docs/5.1.0-M1/user-guide/#writing-tests)** and **extension model** **(https://junit.org/junit5/docs/5.1.0-M1/user-guide/#extensions)** for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a `TestEngine` for running Jupiter based tests on the platform.

JUnit Vintage provides a `TestEngine` for running JUnit 3 and JUnit 4 based tests on the platform.

Running jUnit

Many prefer to execute their tests from the convenience of an automated integrated development environment such as Eclipse, IntelliJ IDEA.

Eclipse

Fortunately, Eclipse includes jUnit as a part of the standard Eclipse Java Development Tools package and provides a clean front-end to defining and running tests. To see Eclipse's jUnit capabilities in action, create a new Java project and add the Philadelphia class from above. Now to define the test class, use Eclipse's "File -> New -> JUnit Test Case" menu option. This will open a "New JUnit Test Case" dialog as seen below. We want the following options:

- Select "New JUnit Jupiter test"
- Name: PhiladelphiaTest
- Class under test: Philadelphia

New JUnit Test Case   —   ☐   ✕

JUnit Test Case

⚠ The use of the default package is discouraged.

◯ New JUnit 3 test ◯ New JUnit 4 test ◉ New JUnit Jupiter test

Source folder: | tutorial9-ecl/src | Browse...

Package: | | (default)   Browse...

Name: | PhiladelphiaTest |

Superclass: | java.lang.Object | Browse...

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp()   ☐ tearDown()

☐ constructor

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

☐ Generate comments

Class under test: | Philadelphia | Browse...

⍰   < Back   Next >   Finish   Cancel

Click "Next" and you'll be given the option to select the methods from the Philadelphia test that should be tested. We'll be testing the "isItSunny" method, so select the checkbox next to that method as seen below:

New JUnit Test Case   —   □   ×

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- ☑ Philadelphia
  - ☑ isItSunny()
- ☐ Object
  - ☐ Object()
  - ☐ getClass()
  - ☐ hashCode()
  - ☐ equals(Object)
  - ☐ clone()
  - ☐ toString()
  - ☐ notify()
  - ☐ notifyAll()
  - ☐ wait(long)
  - ☐ wait(long, int)
  - ☐ wait()
  - ☐ finalize()

[Select All]

[Deselect All]

1 method selected.

☐ Create final method stubs
☐ Create tasks for generated test methods
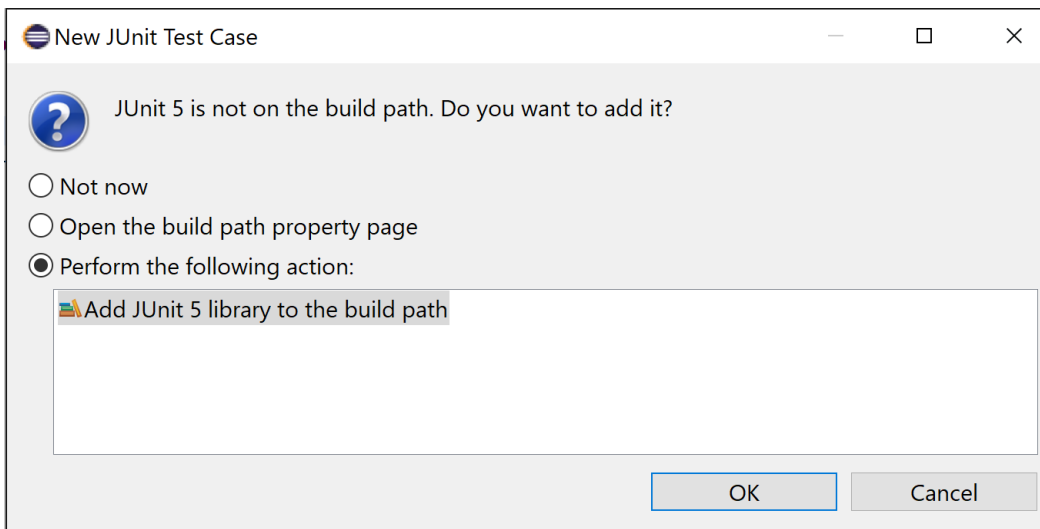
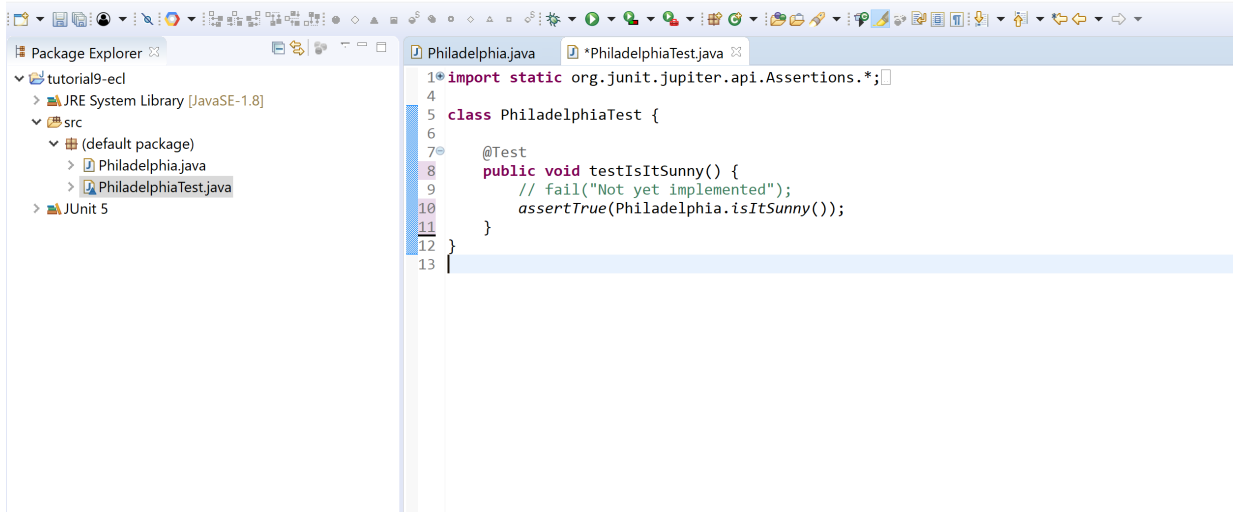(?)   [< Back]   [Next >]   [Finish]   [Cancel]

Click Finish, and Eclipse will create a new Test class for you. (Note: the first time you define a test in a project Eclipse will inform you that jUnit is not on the build path and ask if you would like it to be added. The jUnit library is required to run the tests, so we do indeed want it to be added to the build path).

New JUnit Test Case   —   □   ×

(?)   JUnit 5 is not on the build path. Do you want to add it?

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

Add JUnit 5 library to the build path

[OK]   [Cancel]

All that remains is to actually define the test from above. In the PhiladelphiaTest class that Eclipse created and opened for you, replace the call to "fail" with the original test statement as seen below:
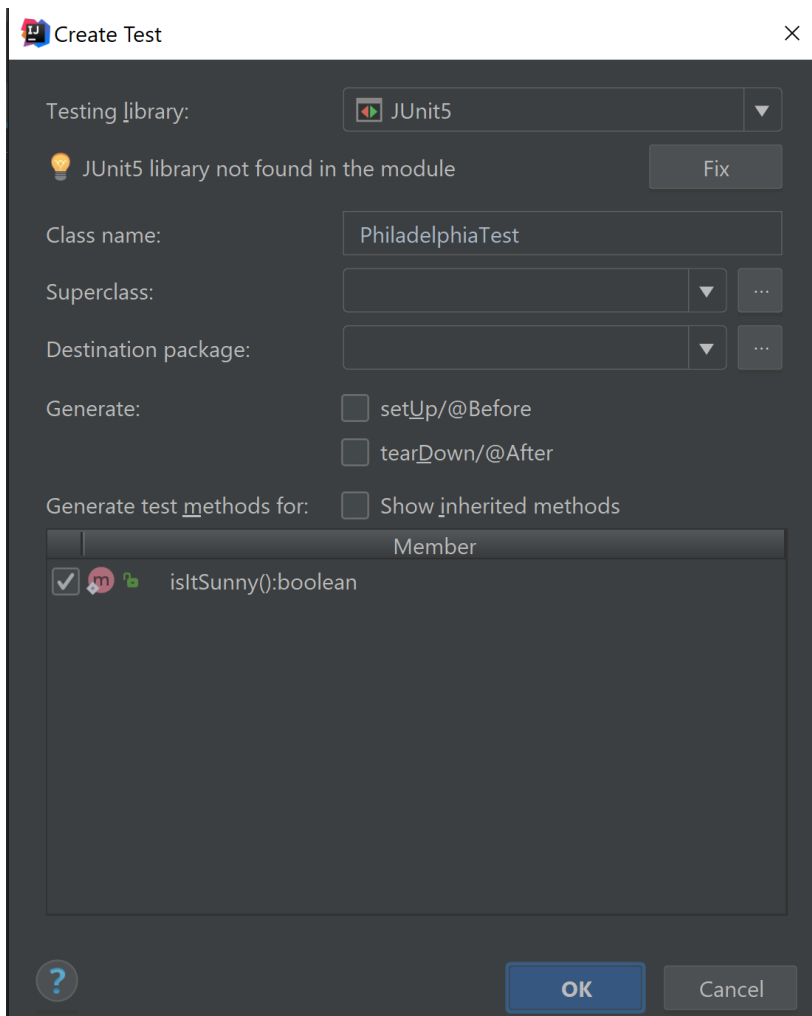
Project - tutorial9-ecl/src/PhiladelphiaTest.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

```
Package Explorer                              Philadelphia.java    *PhiladelphiaTest.java
 tutorial9-ecl                            1 import static org.junit.jupiter.api.Assertions.*;
   JRE System Library [JavaSE-1.8]        4
   src                                    5 class PhiladelphiaTest {
     (default package)                    6
       Philadelphia.java                  7     @Test
       PhiladelphiaTest.java              8     public void testIsItSunny() {
   JUnit 5                                9         // fail("Not yet implemented");
                                         10         assertTrue(Philadelphia.isItSunny());
                                         11     }
                                         12 }
                                         13
```
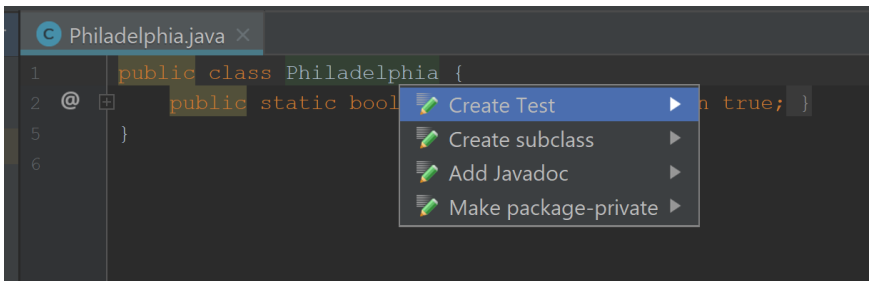
IntelliJ IDEA

IntelliJ IDEA has jUnit integrated into its IDE. To create a new test class, put the cursor in the class name. Then, **Navigate > Test > Create New Test**, select create new test. This will open the "Create Test" window.

Create Test   ×

Testing library:	JUnit5
JUnit5 library not found in the module	Fix
Class name:	PhiladelphiaTest
Superclass:	
Destination package:	
Generate:	☐ setUp/@Before
	☐ tearDown/@After
Generate test methods for:	☐ Show inherited methods

Member
☑ isItSunny():boolean

OK   Cancel

(Another way to open this window is to press **alt + enter** when the cursor is in the class name.)

Exercise 1

Perform the following tasks to become familiar with jUnit basics (you can use either Eclipse or IntelliJ IDEA for these). Type the answers to the questions in a document under "Exercise 1" (you'll submit this document, with all of you exercise solutions in it, at the end of this tutorial.)

- Run the test described above and verify that you get the same output.
- Add a test in which an assertion fails (hint: use assertFalse and your sunny Philadelphia code). Describe (very briefly) what has changed in the test results.
- Choose one other static assertion method from the jUnit Assert API and call it (**this (https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html)** will be useful). Copy your test in your document and describe the results.
- Write a new test that throws an exception when it is triggered. Run the tests again. Copy your test in your document and describe the results.

Test Fixtures

Often when testing you'll need to do the same basic set-up or tear-down activities for several tests. jUnit provides an easy way to aggregate tests and handle the set-up and tear-down activities in separate methods that get called before and after running each test. To use this capability, you need to group such tests into the same containing class and annotate the set-up (pre-test) and tear-down (post-test) methods with "org.junit.jupiter.api.BeforeEach" and "org.junit.jupiter.api.AfterEach" respectively. For example, the following class represents a very simple test fixture that exercises some of the methods found in the "java.util.ArrayList" class:

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

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class ArrayListTest {

    List<Integer> testArray;
```

```java
    /**
     * This method is invoked before each test is run to set up the test array
     * with a known set of values.
     */
    @BeforeEach                           //@Before in jUnit 4
    // Informs JUnit that this method should be run before each test
    public void setUp() {
        testArray =new ArrayList<Integer>(Arrays.asList(3, 1, 4, 1, 5));
    }

    /**
     * This method is invoked after each test is run to perform tear down
     * activities (not needed in this test fixture).
     */
    @AfterEach                            // @After in jUnit 4
    // Informs JUnit that this method should be run after each test
    public void tearDown() {
        // No tear down needed for this test
    }

    /**
     * Adds a value to the array and verifies the add was successful.
     */
    @Test
    public void testAdd() {
        testArray.add(9);

        List<Integer> expected =
            new ArrayList<Integer>(Arrays.asList(3, 1, 4, 1, 5, 9));

        assertEquals(testArray, expected);
    }

    /**
     * Removes a value from the array and verifies the remove was successful.
     */
    @Test
    public void testRemoveObject() {
        testArray.remove(new Integer(5));

        List<Integer> expected =
            new ArrayList<Integer>(Arrays.asList(3, 1, 4, 1));

        assertEquals(testArray, expected);
    }

    /**
     * Tests the indexOf method and verifies the expected return value.
     */
    @Test
    public void testIndexOf() {
        assertEquals(testArray.indexOf(4), 2);
    }
}
```
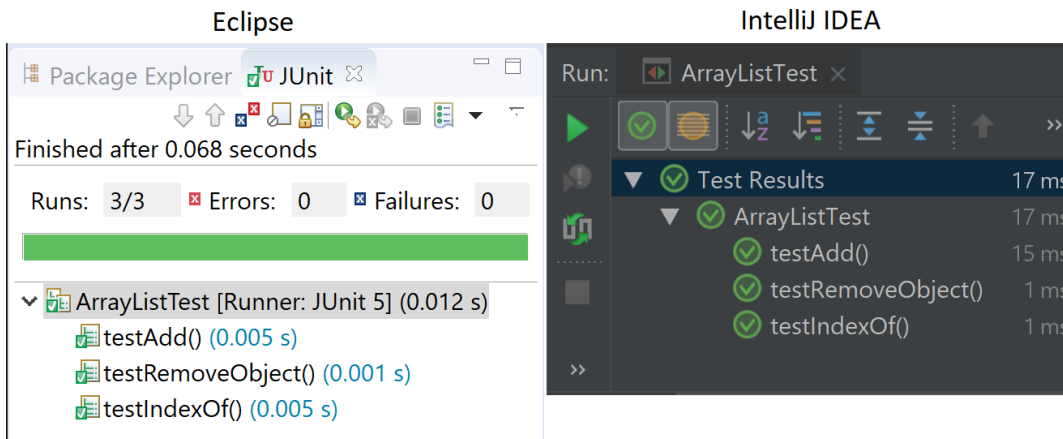
Eclipse	IntelliJ IDEA

Each of the tests above works on the same pre-initialized test array named "testArray". Also, it's important to note that each jUnit test is run independently: actions taken in one test are not seen in any other test. In other words, the set-up and tear-down are run for each test. Upon running the tests in this class, we'll get the following output:

Exercise 2

Perform the following tasks to become familiar with jUnit basics (you can use either IntelliJ IDEA or Eclipse for these). Type the answers to the questions in the same document as above, now under "Exercise 2."

- Run the test fixture described above and verify that you get the same output.
- Add a test that uses the testArray, tests the "clear" method, and verifies that the array is empty. Copy your test into your document for submission.
- Add a test that uses the testArray and tests the "contains" method by verifying that it returns true when supplied a value that exists in the array. Copy your test into your document for submission.
- Add a test that uses the testArray and tests the "contains" method by verifying that it returns false when supplied a value that does not exist in the array. Copy your test into your document for submission.
- Add a test that uses the testArray and tests the "get" method by verifying that it returns the correct value for a given index. Copy your test into your document for submission.

Coverage

Coverage is a metric of how much of a program's source code is exercised by a given test suite. There are many different types of code coverage, but we'll discuss three of the most popular coverage notions here:

- **Statement Coverage:** This coverage metric measures the percentage of statements being exercised by tests.
- **Branch Coverage:** Also known as condition coverage, this coverage metric measures the percentage of true and false evaluations of branches exercised by tests. For example, full branch coverage is achieved when each conditional expression is evaluated to both true and false.
- **Path Coverage:** This coverage metric measures the percentage of program paths exercised by the tests. For example, if the code has three branches, full path coverage would be achieved by testing each possible combination of branch decisions (e.g.: true-true-true, true-true-false, true-false-true, etc., leading to eight candidate paths).

The following example will make this concept clearer. Consider the method below that defines the "identity" method. This implementation has a bug, but the intent is that whatever integer value is supplied as the "x" parameter is returned by the method. The conditionals change the way that the input value is handled on the way to being output, but the *desire* is that under all input values you get the same value back.

```java
public class CoverageExample {
   /**
    * A simple function that should return exactly the provided integer input.
    *
    * The function below is expected to take in a value and return exactly the
    * same value. The manner in which the return value is calculated is
    * affected by three boolean values, but regardless of the boolean values
    * input, the function should already return the originally provided integer
    * value.
    *
    * The implementation below has a bug, but the bug only manifests itself
    * under a specific set of inputs. The intent here is to demonstrate that
    * not all test coverage types are sufficient to find all bugs.
    *
    * @param x - the value to be returned.
    * @param cond1 - the boolean that causes x to be incremented when true.
    * @param cond2 - the boolean that causes x to be decremented when true.
    * @param cond3 - the boolean that causes x to be multiplied by 1 when true.
    * @return the original value of x.
    */
   public static int identity(int x, boolean cond1, boolean cond2, boolean cond3) {
       if (cond1) {      // Statement 1 (S1), Branchpoint 1 (B1)
           x++;       // Statement 2 (S2)
       }

       if (cond2) {       // Statement 3 (S3), Branchpoint 2 (B2)
           x--;        // Statement 4 (S4)
       }

       if (cond3) {       // Statement 5 (S5), Branchpoint 3 (B3)
           x *= 1;  // Statement 6 (S6)
       }
```

```
        return x;           // Statement 7 (S7)
    }
}
```

The example above can be translated to the following control flow graph, which visually represents the possible control flow for the method:
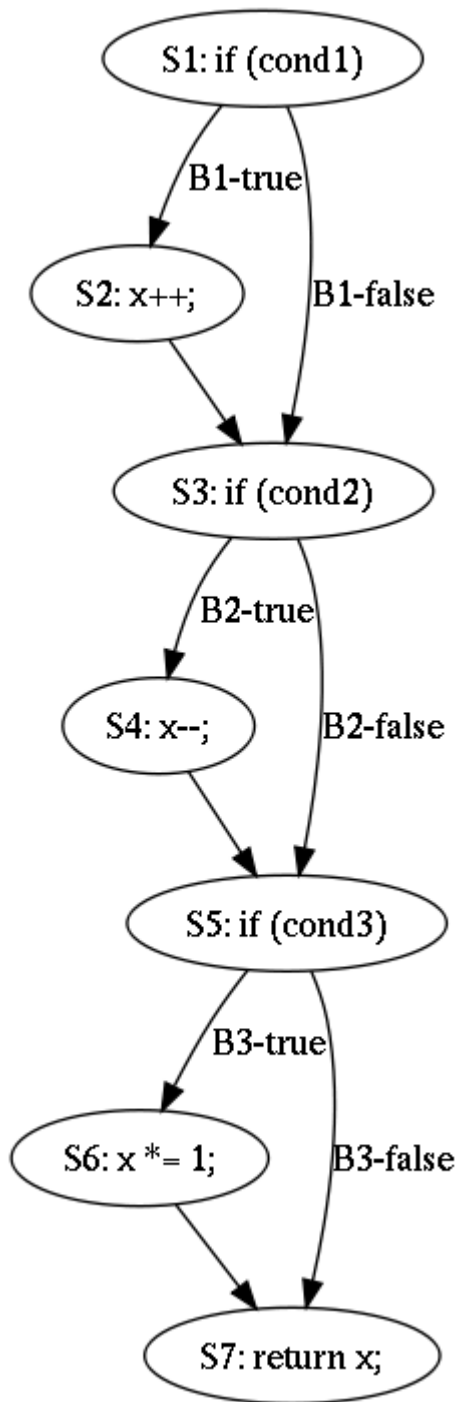
The control flow graph is useful for reasoning about test adequacy with respect to different types of coverage. We'll start with statement coverage. To achieve 100% statement coverage, our tests must cause control to visit every bubble in the control flow graph. Fortunately, this can be achieved with a

single test if we set all the boolean inputs to true. The following shows a test method that provides 100% statement coverage:

```java
@Test
public void testIdentityAllTrue() {
  assertEquals(0, CoverageExample.identity(0, true, true, true));
}
```

It is important to notice that this test passed and provides 100% statement coverage, and yet it did not uncover our bug! Perhaps branch coverage will work better. In branch coverage, we must ensure that our test suite covers every possible branch outcome. Returning to our control flow graph, branch coverage is achieved if all the arrows coming out of each branch expression are covered by the tests. We already have a test that covers all the true branches, so we can achieve 100% branch coverage by writing a similar test but providing all boolean inputs as false. The following shows a test method that can be combined with the previous to achieve 100% branch coverage:

```java
@Test
public void testAllFalse() {
   assertEquals(0, CoverageExample.identity(0, false, false, false));
}
```

We have now achieved branch coverage, yet we have yet to encounter the bug. For that, we need to turn to path coverage. In path coverage, we must cover every possible path through the control flow graph. This requires the following additional tests:

```java
@Test
public void testTrueTrueFalse() {
   assertEquals(0, CoverageExample.identity(0, true, true, false));
}

@Test
public void testTrueFalseTrue() {
   assertEquals(0, CoverageExample.identity(0, true, false, true));
}

@Test
public void testTrueFalseFalse() {
   assertEquals(0, CoverageExample.identity(0, true, false, false));
}

@Test
public void testFalseTrueTrue() {
   assertEquals(0, CoverageExample.identity(0, false, true, true));
}

@Test
public void testFalseTrueFalse() {
```

```
        assertEquals(0, CoverageExample.identity(0, false, true, false));
    }

    @Test
    public void testFalseFalseTrue() {
        assertEquals(0, CoverageExample.identity(0, false, false, true));
    }
```

Now when we execute the test, we finally have found our bug, namely that the identity function does not perform as desired whenever cond1 != cond2.

Path coverage is ideal, but even with this simple example, you can get a feel for how quickly the numbers of required test cases for full path coverage can grow quite large (indeed it usually represents exponential growth).

Exercise 3

The following class provides the "parseTimeToSeconds" method, which takes in a string representing a given time (e.g., "1:23:45 pm") and attempts to parse it and return the number of seconds past midnight that the specified time represents. Your job is to write test suites with various levels of coverage. Specifically, you should perform the following tasks:

- Write a minimal test suite that provides full statement coverage (e.g., write the shortest test suite you can think of that provides full statement coverage).
- Write a minimal test suite that provides full branch coverage.
- Write a minimal test suite that provides full path coverage. If any paths are not possible to test, describe why.

You should submit all three test suites separately in your document under a heading "Exercise 3."

```java
public class TimeParser {
    private static final int MINS_PER_HR = 60;
    private static final int SECS_PER_MIN = 60;

    private static final int MIN_HRS = 0;
    private static final int MAX_HRS = 23;
    private static final int MIN_MINS = 0;
    private static final int MAX_MINS = 59;
    private static final int MIN_SECS = 0;
    private static final int MAX_SECS = 59; // ignore leap seconds

    public static int parseTimeToSeconds(String time) throws NumberFormatException {
        // Normalize the input (trim whitespace and make lower case)
        time = time.trim().toLowerCase();

        int firstColon = time.indexOf(':');
        if (firstColon == -1) {
            throw new NumberFormatException("Unrecognized time format");
```

```
        }

        int secondColon = time.indexOf(':', firstColon+1);
        if (secondColon == -1) {
            throw new NumberFormatException("Unrecognized time format");
        }

        // Interpret everything up to the first colon as the hour
        int hours = Integer.valueOf(time.substring(0, firstColon));
        // Interpret everything between the two colons as the minute
        int minutes = Integer.valueOf(time.substring(firstColon+1, secondColon));
        // Interpret the two characters after the second colon as the seconds
        int seconds = Integer.valueOf(time.substring(secondColon+1, secondColon+3));

        // Adjust hours if 'pm' specified
        if (time.contains("pm") && hours != 12) {
            hours += 12;
        } else if (time.contains("am") && hours == 12) {

            hours = 0;

         }

        // Range check the values
        if ((hours < MIN_HRS || hours > MAX_HRS) ||
            (minutes < MIN_MINS || minutes > MAX_MINS) ||
            (seconds < MIN_SECS || seconds > MAX_SECS)) {
                throw new IllegalArgumentException("Unacceptable time specified");
        }

        // Calculate number of seconds since midnight
        return (((hours * MINS_PER_HR) + minutes) * SECS_PER_MIN) + seconds;
    }
}
```

Invariants

Testing is also useful in assuring that an object's invariant holds—namely that after all method invocations the invariant property remains true. For example, consider the following HeapArray class which implements the heap data structure (specifically a min heap):

```
import java.lang.reflect.Array;
import java.util.Arrays;
import java.util.Iterator;

/**
 * A HeapArray holds element on a priority heap, which orders the elements
 * according to their natural order.
 * <p>
```

```
 * The heap data structure maintains the invariant that in its underlying tree
 * structure the parent is less than all of its children. This implementation
 * maps the tree structure into an array which maintains the heap invariant such
 * that array[n] <= array[2*n + 1] and array[n] <= array[2*n + 2] for all elements in
 * the array, when the root is at n = 0.
 *
 * @param <E> the type of the objects stored in the array (must implement
 *            Comparable)
 */
public class HeapArray<E extends Comparable<E>> implements Iterable<E> {
    private static final int DEFAULT_CAPACITY = 11;

    private static final int DEFAULT_CAPACITY_RATIO = 2;

    private int size;

    private E[] elements;

    /**
      * Constructs a heap array with an initial capacity of 11.
      */
    public HeapArray() {
        this(DEFAULT_CAPACITY);
    }

    /**
      * Constructs a heap array with the specified capacity.
      *
      * @param initialCapacity - the desired initial capacity.
      */
    public HeapArray(int initialCapacity) {
        if (initialCapacity < 1) {
            throw new IllegalArgumentException();
        }

        elements = newElementArray(initialCapacity);
    }

    /**
      * Remove all elements from the heap array
      */
    public void clear() {
        Arrays.fill(elements, null);
        size = 0;
    }

    /**
      * Adds the specified object to the heap array.
      *
      * @param o - the object to be added
      */
    public void add(E o) {
        if (null == o) {
            throw new NullPointerException();
```

```
        }
        int i = size;
        if (i >= elements.length)
            growToSize(i + 1);
        size = i + 1;
        if (i == 0)
            elements[0] = o;
        else
            siftUp(i, o);
    }

    /**
     * Remove the smallest item from the heap array.
     *
     * @return - the smallest item in the heap array.
     */
    public E pop() {
        if (isEmpty()) {
            return null;
        }
        E result = elements[0];
        removeAt(0);
        return result;
    }

    /**
     * Get the element stored at the specified index in the heap array.
     *
     * @param index - the index of the element.
     * @return the element stored at the specified index.
     */
    public E get(int index) {
        return elements[index];
    }

    /**
     * Peek at the smallest item in the heap array (but do not remove it).
     *
     * @return the smallest item in the heap array.
     */
    public E peek() {
        if (isEmpty()) {
            return null;
        }
        return elements[0];
    }

    /**
     * Remove the specified object from the heap array.
     *
     * @param o - the object to be removed.
     * @return whether or not the request object was found and removed.
     */
    public boolean remove(Object o) {
```

```java
            if (o == null) {
                return false;
            }
            for (int i = 0; i < size; i++) {
                if (elements[i].equals(o)) {
                    removeAt(i);
                    return true;
                }
            }
            return false;
        }

        /**
          * Check whether or not the heap array contains the specified object.
          *
          * @param object - the object to be checked.
          * @return whether or not the specified object is in the heap array.
          */
        public boolean contains(Object object) {
            for (int i = 0; i < size; i++) {
                if (elements[i].equals(object)) {
                    return true;
                }
            }
            return false;
        }

        /**
          * Check if the heap array is empty.
          *
          * @return whether or not the heap array is empty.
          */
        public boolean isEmpty() {
            return size == 0;
        }

        /**
          * Return the size of the heap array (how many items are in it).
          *
          * @return the size of the heap array.
          */
        public int size() {
            return size;
        }

        /**
          * Returns a new array containing all the elements contained in the heap
          * array.
          *
          * @return the new array containing all the elements contained in the heap
          *         array
          */
        public Comparable<E>[] toArray() {
            return Arrays.copyOf(elements, elements.length);
```

```
    }

    /**
     * Returns an array containing all the elements contained in the heap array.
     * If the specified array is large enough to hold the elements, the
     * specified array is used, otherwise, an array of the same type is created.
     * If the specified array is used and is larger than the heap array, the
     * array element following the heap array contents is set to null.
     *
     * @param <T> - the array element type used for the return array.
     * @param destinationArray - the array used as specified above.
     * @return the new array containing the elements contained in the heap
     *         array.
     */
    @SuppressWarnings("unchecked")
    public <T> T[] toArray(T[] destinationArray) {
        if (size > destinationArray.length) {
            Class<?> ct = destinationArray.getClass().getComponentType();
            destinationArray = (T[]) Array.newInstance(ct, size);
        }
        System.arraycopy(elements, 0, destinationArray, 0, size);
        if (size < destinationArray.length) {
            destinationArray[size] = null;
        }
        return destinationArray;
    }

    @SuppressWarnings("unchecked")
    private E[] newElementArray(int capacity) {
        return (E[]) new Comparable[capacity];
    }

    private void removeAt(int index) {
        size--;
        if (size == index)
            elements[index] = null;
        else {
            E moved = elements[size];
            elements[size] = null;
            siftDown(index, moved);
            if (elements[index] == moved) {
                siftUp(index, moved);
            }
        }
    }

    /**
     * Inserts target at position childIndex, maintaining heap invariant by
     * promoting target up the tree until it is greater than or equal to its
     * parent, or is the root.
     *
     * @param childIndex - the index used for the original placement of target.
     * @param target - the item to be placed and sifted.
     */
```

```java
    private void siftUp(int childIndex, E target) {
        while (childIndex > 0) {
            int parentIndex = (childIndex - 1) / 2;
            E parent = elements[parentIndex];
            if (target.compareTo(parent) >= 0)
                break;
            elements[childIndex] = parent;
            childIndex = parentIndex;
        }
        elements[childIndex] = target;
    }

    /**
     * Inserts target at position index, maintaining the heap invariant by
     * demoting target down the tree repeatedly until it is less than or equal
     * to its children or is a leaf.
     *
     * @param index the index where the target should initially be placed.
     * @param target the item to be placed and sifted down.
     */
    private void siftDown(int index, E target) {
        int half = size / 2;
        while (index < half) {
            int childIndex = index * 2 + 1;
            E child = elements[childIndex];
            int rightIndex = childIndex + 1;
            if (rightIndex < size
                    && child.compareTo(elements[rightIndex]) > 0) {
                childIndex = rightIndex;
                child = elements[childIndex];
            }
            if (target.compareTo(child) <= 0)
                break;
            elements[index] = child;
            index = childIndex;
        }
        elements[index] = target;
    }

    private void growToSize(int size) {
        if (size > elements.length) {
            E[] newElements = newElementArray(size * DEFAULT_CAPACITY_RATIO);
            System.arraycopy(elements, 0, newElements, 0, elements.length);
            elements = newElements;
        }
    }

    @Override
    public Iterator<E> iterator() {
        return Arrays.asList(elements).iterator();
    }

    @Override
    public String toString() {
```

```java
        if (isEmpty()) {
            return "[]";
        }

        StringBuilder buffer = new StringBuilder(size * 16);
        buffer.append('[');
        Iterator<E> it = iterator();
        while (it.hasNext()) {
            E next = it.next();
            if (next != this) {
                buffer.append(next);
            } else {
                buffer.append("(this HeapArray)");
            }
            if (it.hasNext()) {
                buffer.append(", ");
            }
        }
        buffer.append(']');
        return buffer.toString();
    }
}
```

One of the HeapArray's class invariants is that the minimum value in the array is at array index zero. The following test class verifies that the invariant holds. Note that, for simplicity, only public methods that modify the underlying element array are tested here. In practice, all public methods should be tested as well to ensure a more complete set of tests are in place.

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

import org.junit.jupiter.api.*;
import java.util.Arrays;
import java.util.Collections;
import java.util.Random;

public class MinHeapArrayInvariant1Test {
    private static final long SEED = 42;
    private static final long VALUES_TO_TEST = 1000;
    private Random random;
    private HeapArray<Integer> heap;

    @BeforeEach
    public void setUp() {
        random = new Random(SEED);
        heap = new HeapArray<Integer>();
    }

    @Test
    public void testWithRandomAdds() {
        for (int i=0; i < VALUES_TO_TEST; i++) {
            int addMe = random.nextInt();
```

```java
            heap.add(addMe);
            assertTrue(invariantHolds());
        }
    }

    @Test
    public void testWithRandomRemoves() {
        fillWithRandomValues(VALUES_TO_TEST);
        while (heap.size() > 0) {
            int indexToRemove = random.nextInt(heap.size());
            Integer removeMe = heap.get(indexToRemove);
            heap.remove(removeMe);
            assertTrue(invariantHolds());
        }
    }

    @Test
    public void testPop() {
        fillWithRandomValues(VALUES_TO_TEST);
        while (heap.size() > 0) {
            heap.pop();
            assertTrue(invariantHolds());
        }
    }

    @Test
    public void testClear() {
        fillWithRandomValues(VALUES_TO_TEST);
        heap.clear();
        assertTrue(invariantHolds());
    }

    private boolean invariantHolds() {
        Integer top = heap.peek();
        if (top == null) {
            return true;
        }

        Integer[] contents = new Integer[heap.size()];
        Integer min = Collections.min(Arrays.asList(heap.toArray(contents)));
        if (min > top) {
            System.out.println("Whoops!");
        }

        return min <= top;
    }

    private void fillWithRandomValues(long numValues) {
        for (int i = 0; i < numValues; i++){
            heap.add(random.nextInt());
        }
    }
}
```

Exercise 4

Write a test suite that tests the following additional class invariant:

"For all elements in the array, array[n] <= array[2*n+1] and array[n] <= array[2*n+2] when root is at n=0"

If, in executing your test, you find any problems with the implementation, suggest how to change the code to correct the error (do not need to implement, but implement the test suite to catch this failure).

Place both your test suite and your suggested corrections in your document under the heading "Exercise 4."

What to Submit

Generate a PDF from the document you've been creating and name it as FIRSTNAME_LASTNAME.pdf. Include it as your submission for this part of the tutorial.

You should submit two files: 1) The PDF from part 1, and 2) the MinWebTest Java class file from part 2.

Notes:

Please turn your submission in on time. No late submission policy.