Revised: 7/24/2019

JUnit and Unit Testing

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

What is JUnit?

- A commonly used library/API to create tests which can be automated.
- Each ***unit test*** performs an individual test of a class's functionality.
  - Like testing that a method works as designed.
- With the unit tests in place, the tests can easily be re-run every time a change is made to the class.

What is JUnit?

- By now, you probably realize testing that a program works is time consuming.
  - Especially when you need to re-test everything after making a change.

- Having an established JUnit test allows you to automatically re-run the same tests to make sure everything still works correctly.

- *Why haven't you told me about this sooner?!*
  - It's better to first have a decent grasp of object-oriented programming.

Setting up JUnit

- The JUnit libraries need to be set in your IntelliJ Project.
  - **junit.jar**
  - **junit-4.12.jar**
  - **hamcrest-core-1.3.jar**
  - All three should be in the lib folder where IntelliJ is installed on your computer.
    - Usually in C:\Program Files\JetBrains\IntelliJ Community Edition\lib

- See the JUnit Setup Document in Module 4 for a walkthrough.

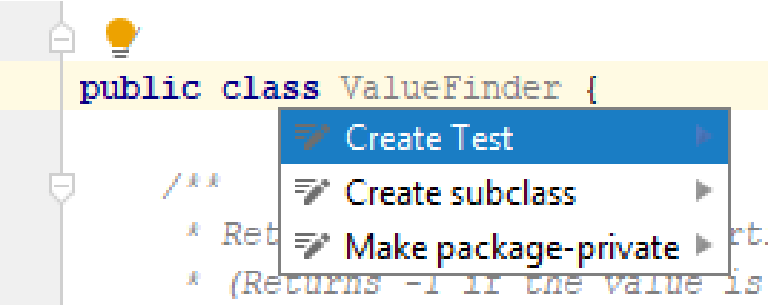
Creating a JUnit Test Class in IntelliJ

- Open the class you want to create a test for.
- Position the cursor on the **class** keyword in the class declaration.

```
*/
public class ValueFinder {
```

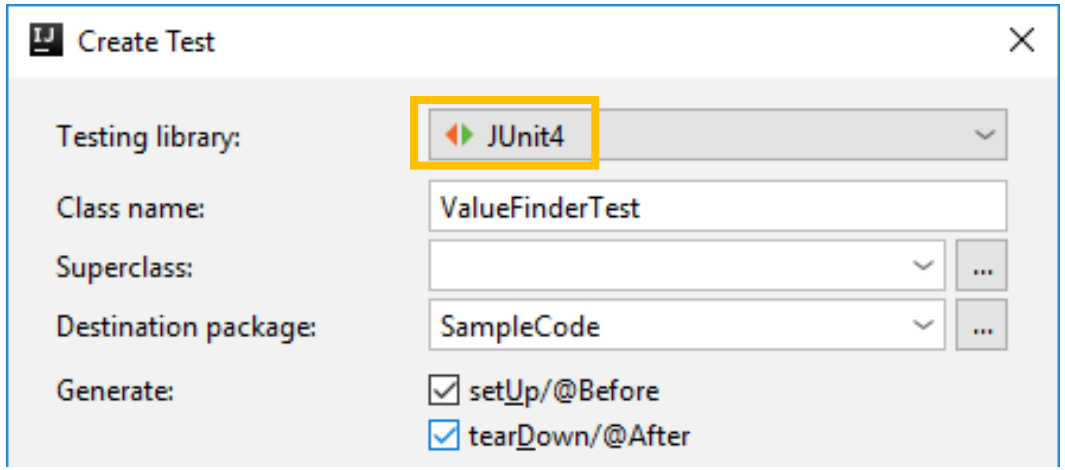
Creating a JUnit Test Class in IntelliJ

- Hold down the **Alt** key and press **Enter**
- Select **Create Test**

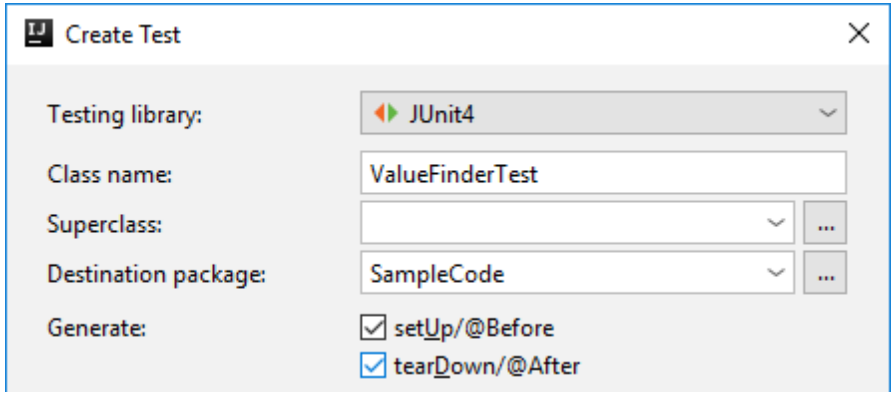


Creating a JUnit Test Class in IntelliJ

- In the Create Test window, select **JUnit 4** from the Testing library drop down menu.



Creating a JUnit Test Class in IntelliJ

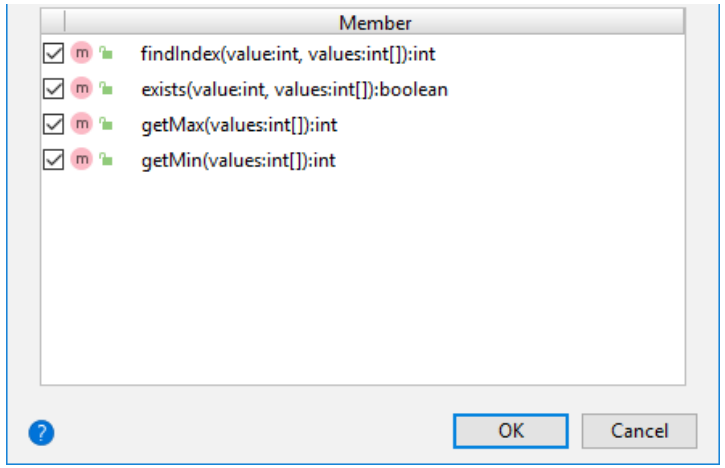


- Class name – The name of the class that will contain the unit tests.
- Destination Package – The folder where this class will be created.
- setUp method – Automatically called when the unit test program starts.
  - Used to set up any objects that are to be tested.
- tearDown method – Automatically called when the unit test program ends.
  - Used to clean up any objects that were tested.
  - More often used to close any files, database connections, etc. that were opened during the test.

Creating a JUnit Test Class in IntelliJ

- In the Member section, select any methods that you want included in the test class.
  - You can always write/add new test methods later on.
  - Selecting them here just automatically generates methods for each of them.
  - The methods are automatically called when the test runs.

- Click **OK** to create the test class.



Creating a JUnit Test Class in IntelliJ

- The test class is setup and ready for tests to be added.

ValueFinder.java | ValueFinderTest.java

```java
package SampleCode;

import ...

public class ValueFinderTest {

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void findIndex() {
    }

    @Test
    public void exists() {
    }

    @Test
    public void getMax() {
    }

    @Test
    public void getMin() {
    }
}
```

Creating a JUnit Test Class in IntelliJ

- We can add any fields needed for the test class.
  - They will be accessible by any methods in the test class.
- In this example, we'll have…
  - A field for a ValueTest object (**STOP** and review the ValueFinder class in the SampleCode folder)
  - Three arrays used to test the class

```
public class ValueFinderTest {

    private ValueFinder vf;
    private int[] array1 = {3, 6, 4, 7, 1};
    private int[] array2 = {2, 4, 6, 8, 10};
    private int[] array3 = {1, 2, 3, 8, 9, 10};
```

The setUp method

- This method is automatically called when the test class runs.
  - Here is where you'll want to instantiate any objects needed for testing.
  - It's kind of like the constructor for this test class.
  - Having this method is optional.

```
@Before
public void setUp() throws Exception {

}
```

The setUp method

- In this example, it will only instantiate an object of the ValueFinder class for the ValueFinder field named vf.

```
@Before
public void setUp() throws Exception {
    vf = new ValueFinder();
}
```

The findIndex method (automatically generated)

- The automatically generated “findIndex” method will be used to test the findIndex method of the ValueFinder class.
  - The name of the method in the test class is arbitrary; it can be named anything.
  - Just keep the @Test annotation, though.
    - <u>This is what tells JUnit that this method is a unit test</u>. The method’s name is irrelevant.

```
@Test
public void findIndex() {

}
```

Assertions

- Unit tests commonly make use of assertions about values returned by a method.
- A commonly used assertion is the JUnit assertEquals method.
- The syntax is shown below.

```
@Test
public void findIndex() {
    assertEquals(1, vf.findIndex(6, array1));
}
```

assertEquals

- Two arguments
  - The value expected
  - The value we get

```
@Test
public void findIndex() {
    assertEquals(1, vf.findIndex(6, array1));
}
```

The value that we are expecting to be returned

The value to test
In this case, the value returned by the ValueFinder object's findIndex method

assertEquals

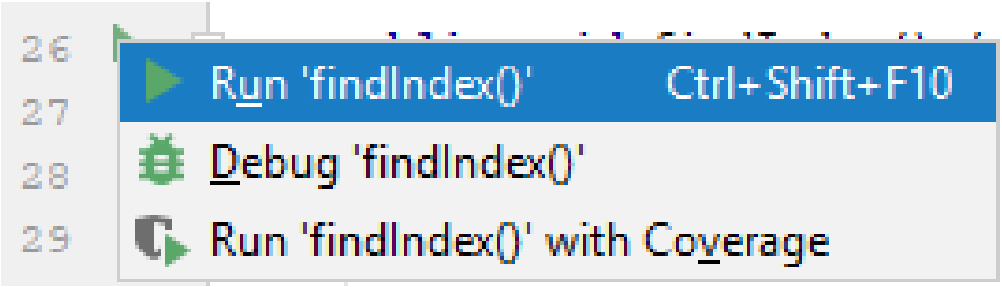
- We can add as many assertions as we want.

```
@Test
public void findIndex() {
    assertEquals(1, vf.findIndex(6, array1));
    assertEquals(3, vf.findIndex(8, array2));
    assertEquals(2, vf.findIndex(3, array3));
    assertEquals(-1, vf.findIndex(5, array1));
}
```

- If any assertions fail, this <u>entire</u> method (or *unit test*) fails.

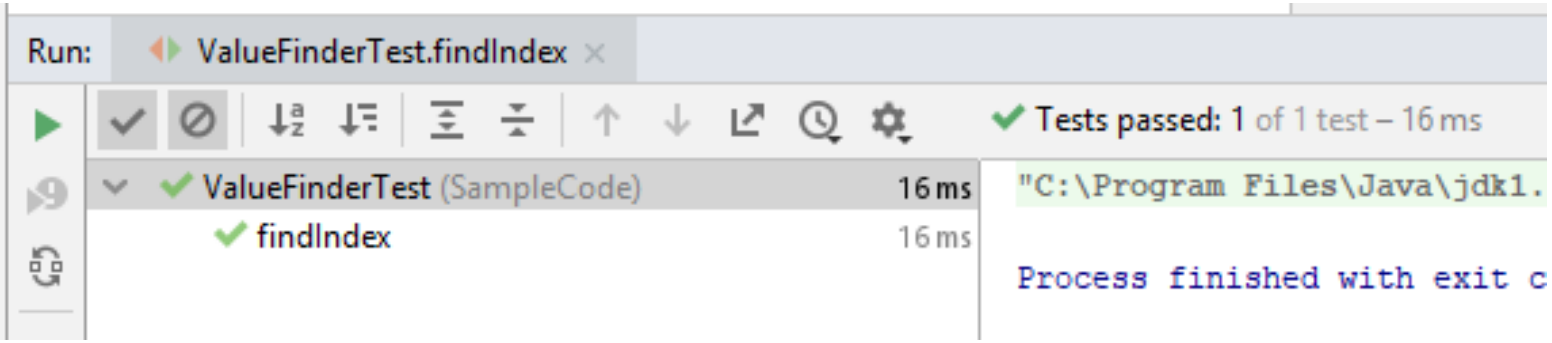
Running a test

- To run one unit test/one method, click the play button to the right of the method.
  - Select **Run 'name of method'**



Running a test

- The test results are shown at the bottom of IntelliJ.
  - All four assertions must not have failed, since this one unit test passed.




```
25      @Test
26      public void findIndex() {
27          assertEquals(1, vf.findIndex(6, array1));
28          assertEquals(3, vf.findIndex(8, array2));
29          assertEquals(2, vf.findIndex(3, array3));
30          assertEquals(-1, vf.findIndex(5, array1));
31      }
```

Running a test

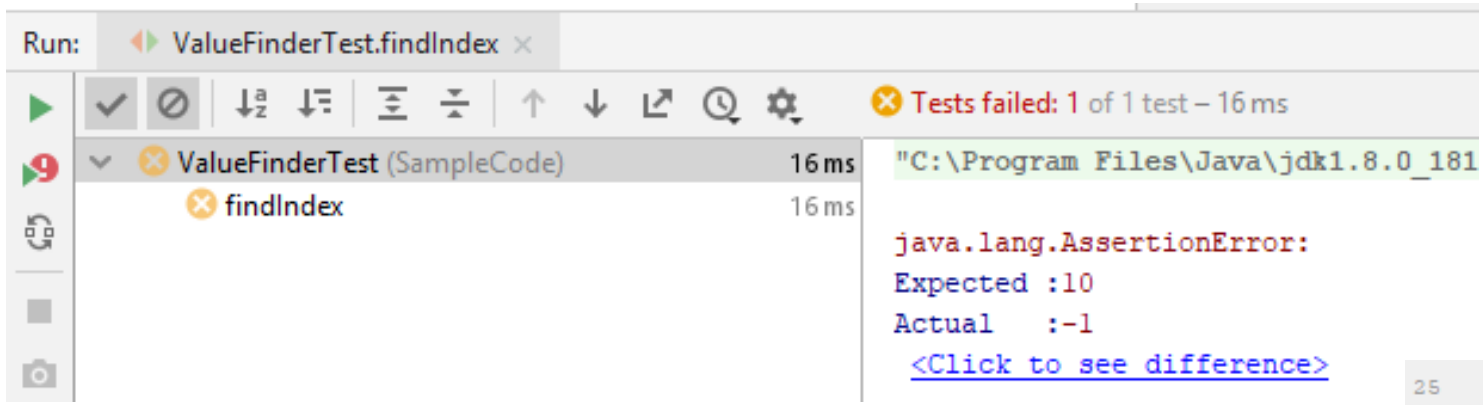
- We'll purposely cause a test to fail, just to see what happens.

```
@Test
public void findIndex() {
    assertEquals(1, vf.findIndex(6, array1));
    assertEquals(3, vf.findIndex(8, array2));
    assertEquals(2, vf.findIndex(3, array3));
    assertEquals(10, vf.findIndex(5, array1));
}
```

- We know findIndex should return -1 on that last assertion, but we'll tell JUnit to expect the value 10 to be returned.

Running a test

- Re-running the test shows that this test now failed.
  - Even though the first three assertions passed, the unit test (the method itself) failed.




```
25    @Test
26    public void findIndex() {
27        assertEquals(1, vf.findIndex(6, array1));
28        assertEquals(3, vf.findIndex(8, array2));
29        assertEquals(2, vf.findIndex(3, array3));
30        assertEquals(10, vf.findIndex(5, array1));
31    }
```

Assertions

- The assertTrue and assertFalse assertions are used to test the Boolean value returned by a method.
- They are both used in the exists method of the example test class.
  - Used to test the ValueFinder object's exists method.

```
@Test
public void exists() {
    assertTrue(vf.exists(4, array2));
    assertFalse(vf.exists(5, array1));
}
```

assertTrue and assertFalse

- One argument
  - The value we get (Must be a Boolean type)

```
@Test
public void exists() {
    assertTrue(vf.exists(4, array2));
    assertFalse(vf.exists(5, array1));
}
```

We expect this to return true

We expect this to return false

The value to test for true or false

The tearDown method

- This method is automatically called when the test class finished.
  - Here is where you'll want close any open files or resources.
  - Having this method is optional.
  - In this test, it will simply set the vf field to null.
    - Not really necessary to do, but we'll keep it for demonstration purposes.

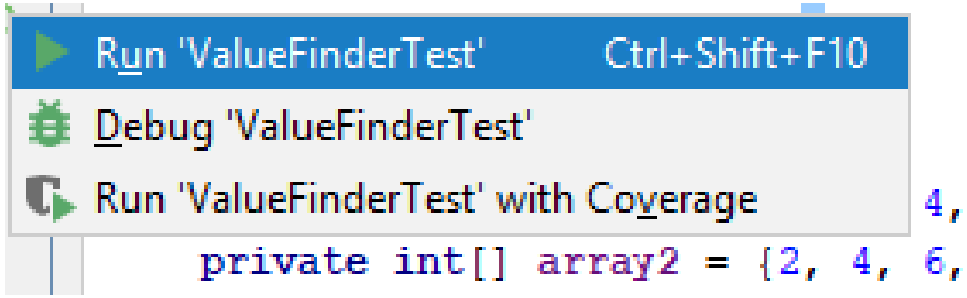
```
@After
public void tearDown() throws Exception {
    vf = null;
}
```

All four tests (for this demonstration)

```
@Test
public void findIndex() {
    assertEquals(1, vf.findIndex(6, array1));
    assertEquals(3, vf.findIndex(8, array2));
    assertEquals(2, vf.findIndex(3, array3));
    assertEquals(-1, vf.findIndex(5, array1));
}

@Test
public void exists() {
    assertTrue(vf.exists(4, array2));
    assertFalse(vf.exists(5, array1));
}

@Test
public void getMax() {
    assertEquals(7, vf.getMax(array1));
}

@Test
public void getMin() {
    assertEquals(1, vf.getMax(array1));
}
```

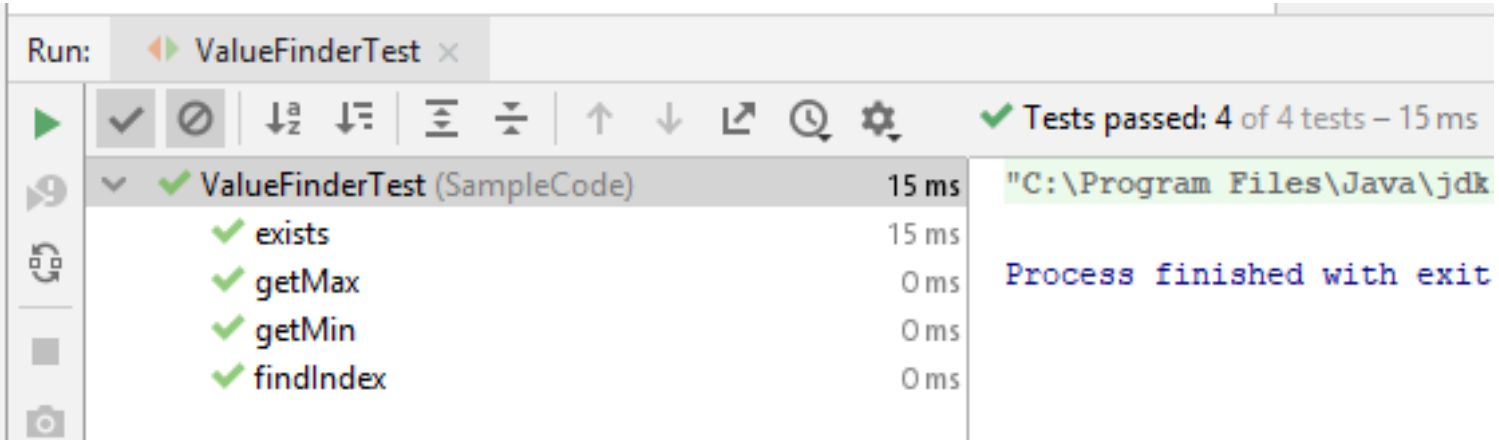
Running ALL tests

- To run all of the unit tests at once, click the play button to the right of the class header.
  - Select **Run 'name of class'**



Running ALL tests

- The test results are shown at the bottom of IntelliJ.
  - All unit tests appear to have passed.




```
10
11      private ValueFinder vf;
12      private int[] array1 = {3, 6,
```

```
public class ValueFinderTest {
```

Test Class for Bicycle.java

(Same Bicycle.java from Module 1 Sample Code)

```java
@Test
public void AccelerateTest() {
    Bicycle b = new Bicycle();
    b.speedUp();                    //Accelerate (add 5) to the speed
    b.speedUp();
    b.speedUp();                    //Speed should be at 15
    assertEquals(15, b.getSpeed());
}

@Test
public void DecelerateTest() {
    Bicycle b = new Bicycle();
    b.speedUp();                    //Accelerate (add 5) to the speed
    b.speedUp();                    //Speed should be at 10
    b.slowDown();                   //Decelerate (subtract 5) from the speed; Speed should be at 5
    assertEquals(5, b.getSpeed());
}

@Test
public void NegativeSpeedTest() {
    Bicycle b = new Bicycle();
    b.speedUp();                    //Accelerate (add 5) to the speed
    b.speedUp();                    //Speed should be at 10
    b.slowDown();                   //Decelerate (subtract 5) from the speed; Speed should be at 5
    b.slowDown();                   //Decelerate (subtract 5) from the speed; Speed should be at 0
    b.slowDown();                   //Decelerate (subtract 5) from the speed; Speed should still be at 0 (not -5)
    assertEquals(0, b.getSpeed());
}

@Test
public void GearTest() {
    Bicycle b = new Bicycle();
    b.setGear(10);                  //Sets the gear to 10
    assertEquals(10, b.getGear());  //Tests the gear field is set to 10
    b.setGear(-5);                  //Sets the gear to -5 (Should be defaulted to 1)
    assertEquals(1, b.getGear());   //Tests the gear field is set to 1
    b.setGear(20);                  //Sets the gear to 20 (Should be defaulted to 1)
    assertEquals(1, b.getGear());   //Tests the gear field is set to 1
}
```

- This test class is designed a little differently from the ValueFinder example
- Each test instantiates its own Bicycle object for testing.

Test Class for Bicycle.java

- If changes are made or new features are added to Bicycle.java, I can re-run these tests to make sure I didn't break any of the existing/working features.
  - The order of the tests in the test class doesn't matter.
  - Each runs independently of one another.