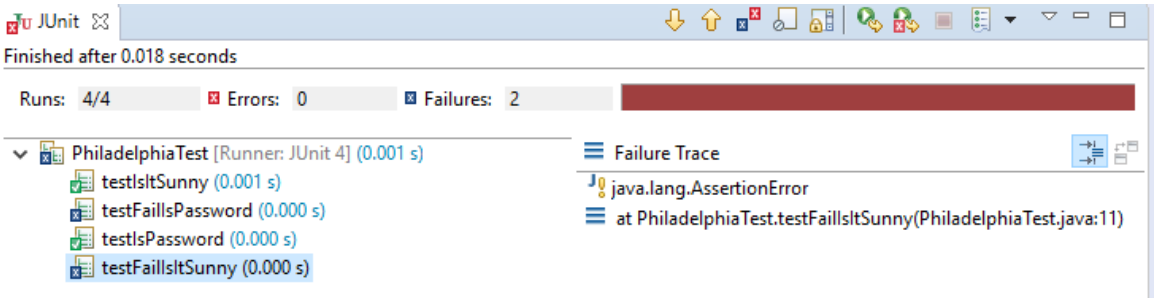
Exercise 1:

Run the test described above and verify that you get the same output.

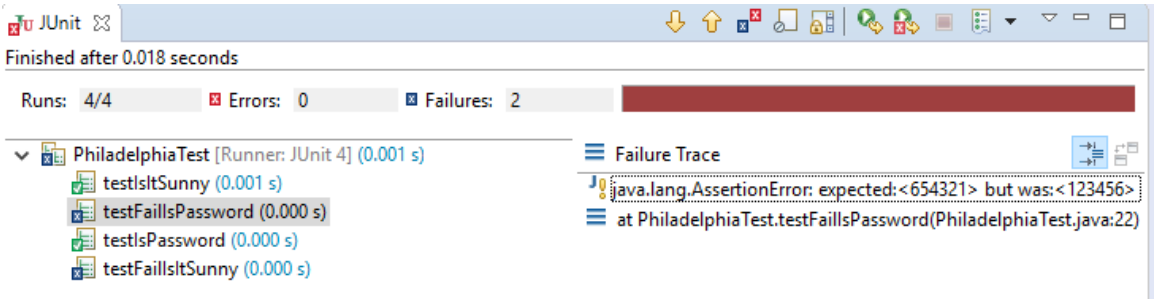
Add a test in which an assertion fails (hint: use assertFalse and your sunny Philadelphia code). Describe (very briefly) what has changed in the test results.



Choose one other static assertion method from the jUnit Assert API and call it (this will be useful). Give the test that you wrote and describe the results.

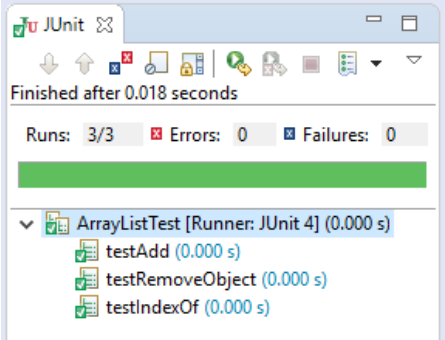
```
public static long isPassword() {
    long password = 123456;
    return password;
}

@Test
public void testIsPassword() {
    long p = 123456;
    assertEquals(p, Philadelphia.isPassword());
}
@Test   //java.lang.AssertionError: expected:<654321> but was:<123456>
public void testFailIsPassword() {
    long p = 654321;
    assertEquals(p, Philadelphia.isPassword());
}
```

Write a new test that throws an exception when it is triggered. Run the tests again. Give the test that you wrote and describe the results.



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.

```
@Test
public void testClear() {
	testArray.clear();
	assertTrue(testArray.isEmpty());
}
```

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.

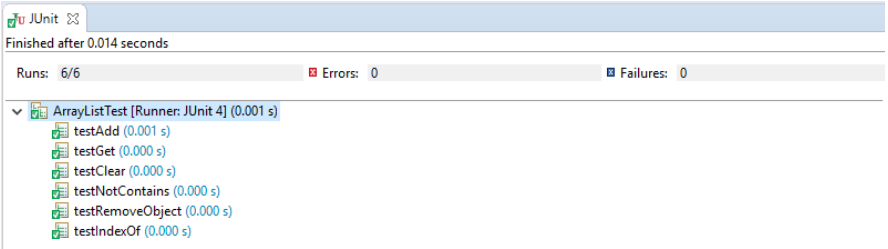
```
@Test
public void testContains() {
	assertTrue(testArray.contains(3));
}
```

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.

```
@Test
public void testNotContains() {
	assertFalse(testArray.contains(2));
}
```

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.

```
@Test
public void testGet() {
	assertEquals(testArray.get(0), new Integer(3));
}
```



Exercise 3:

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).

```
@Test
public void testStatement() {
assertEquals(48225, TimeParser.parseTimeToSeconds("1:23:45 pm"));
}
```

Write a minimal test suite that provides full branch coverage.

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void testBranch() {
	thrown.expect(NumberFormatException.class);
	thrown.expectMessage("Unrecognized time format");
	assertEquals(5025, TimeParser.parseTimeToSeconds("12345 am"));
	thrown.expect(IllegalArgumentException.class);
	thrown.expectMessage("Unacceptable time specified");
	assertEquals(5025, TimeParser.parseTimeToSeconds("1:23:99 am"));
}
```

Write a minimal test suite that provides full path coverage. If any paths are not possible to test, describe why.

```
@Test
public void testPath() {
	thrown.expect(NumberFormatException.class);
	thrown.expectMessage("Unrecognized time format");
	assertEquals(5025, TimeParser.parseTimeToSeconds("12345 pm"));
	thrown.expect(NumberFormatException.class);
	thrown.expectMessage("Unrecognized time format");
	assertEquals(5025, TimeParser.parseTimeToSeconds("1:2345 pm"));
	thrown.expect(IllegalArgumentException.class);
	thrown.expectMessage("Unacceptable time specified");
	assertEquals(5025, TimeParser.parseTimeToSeconds("1:23:99 pm"));
}
```

Some paths are not possible to test because process can't continue to proceed if preconditions goes wrong. Some conditions with wrong precondition could never be reached.