Introduction to Software Testing
(*2nd edition*)

Chapter 3

Test Automation and Testability

Instructor: Morteza Zakeri

Slides by: Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

Modified by: Morteza Zakeri

https://m-zakeri.github.io/

March 2024

What is Test Automation?

Test Automation or Automated Testing: The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions

- Reduces cost
- Reduces human error
- Reduces variance in test quality from different individuals
- Significantly reduces the cost of regression testing
- We will see a more comprehensive view later in this lecture

Software Testability (3.1)

Software Testability: The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

- Plainly speaking – how hard it is to find faults in the software
- Testability is dominated by two practical problems
  - How to provide the test values to the software
  - How to observe the results of test execution

Observability and Controllability

- **Observability**

How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

  - Software that affects hardware devices, databases, or remote files have low observability

- **Controllability**

How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

  - Easy to control software with inputs from keyboards
  - Inputs from hardware sensors or distributed software is harder

- Data abstraction reduces controllability and observability

Components of a Test Case (3.2)

- A test case is a **multipart artifact** with a definite structure
- Test case values (Test data)

> The input values needed to complete an execution of the software under test (SUT)

- Expected results (Test oracle)

> The result that will be produced by the test if the software behaves as expected

  - A *test oracle* uses expected results to decide whether a test passed or failed

Affecting Controllability and Observability

- Prefix values

> Inputs necessary to put the software into the appropriate state to receive the test case values

- Postfix values

> Any inputs that need to be sent to the software after the test case values are sent

1. *Verification Values*: Values needed to see the results of the test case values
2. *Exit Values*: Values or commands needed to terminate the program or otherwise return it to a stable state

Putting Tests Together

- Test case

The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test

- Test set

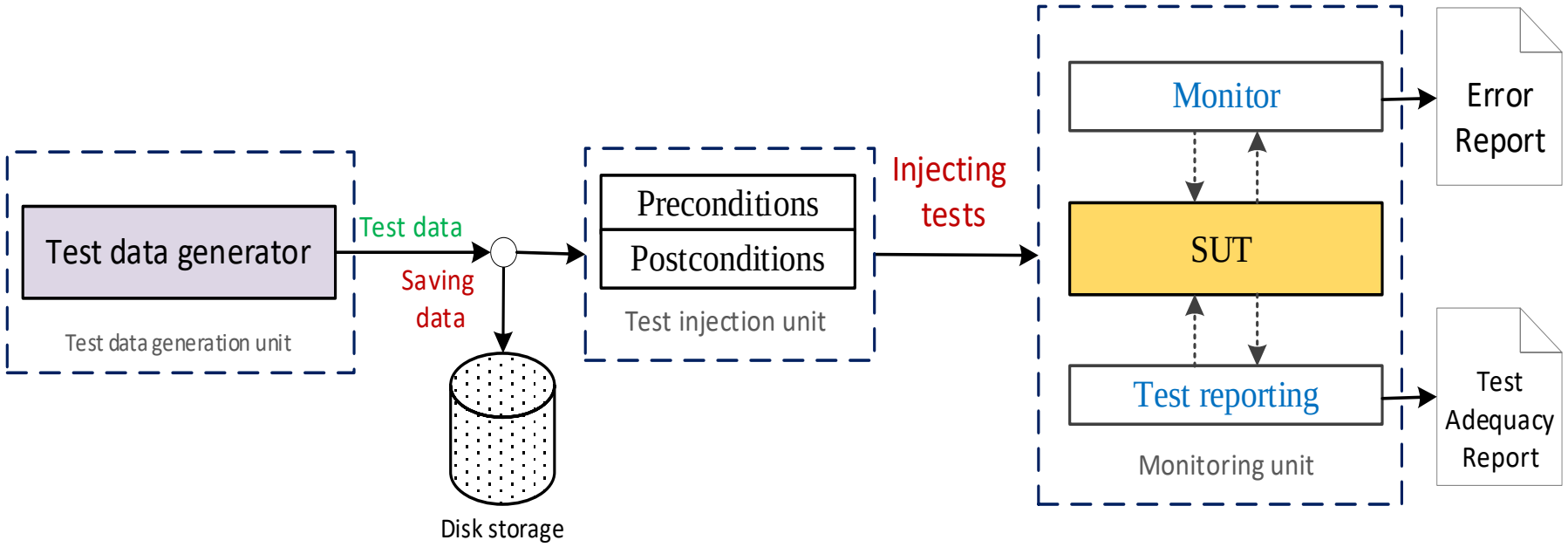
A set of test cases

- Executable test script

A test case that is prepared in a form to be executed automatically on the test software and produce a report

Test Automation: Revisited

- Test automation components and process
  - Test automation is not only test injection



Zakeri Nasrabadi, M., Parsa, S. & Kalaee, A. **Format-aware learn&fuzz: deep test data generation for efficient fuzzing**. *Neural Comput & Applic* 33, 1497–1513 (2021). https://doi.org/10.1007/s00521-020-05039-7

A sample tool: DeepFuzz (*https://m-zakeri.github.io/iust_deep_fuzz/*)

Test Automation Framework (3.3)

A set of assumptions, concepts, and tools that support test automation

Boundaries between automation framework and a testing tool:

- Tools are specifically designed to target some particular test environment, such as Windows and web automation tools, etc.
  - Serve as a driving agent for an automation process
- An automation framework is not a tool to perform a specific task, but rather infrastructure that provides the solution where different tools can do their job in a unified manner.

What is JUnit?

- Open source Java testing framework used to write and run repeatable automated tests
- JUnit is open source (junit.org)
- A structure for writing test drivers
- JUnit features include:
  - Assertions for testing expected results
  - Test features for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners
- JUnit is widely used in industry
- JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse

JUnit Tests

- JUnit can be used to test …
  - … an entire object
  - … part of an object – a method or some interacting methods
  - … interaction between several objects
- It is primarily intended for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests
- Get started at junit.org

Writing Tests for JUnit

- Need to use the methods of the junit.framework.assert class
  - javadoc gives a complete description of its capabilities
- Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded
- The test runner uses the result to report to the user (in command line mode) or update the display (in an IDE)
- All of the methods return void
- A few representative methods of junit.framework.assert
  - *assertTrue (boolean)*
  - *assertTrue (String, boolean)*
  - *fail (String)*

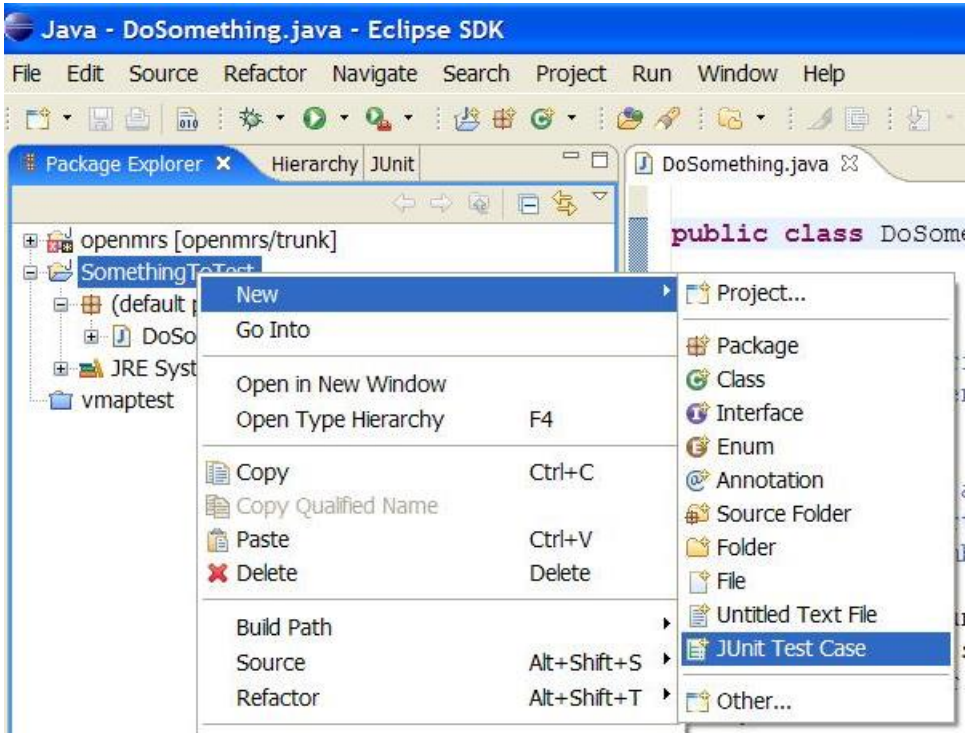
JUnit Test Fixtures

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (*prefix* values)
  - Reset values (*postfix* values)
- Different tests can use the objects without sharing the state
- Objects used in test fixtures should be declared as instance variables
- They should be initialized in a @Before method
- Can be deallocated or reset in an @After method

JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → JUnit → JUnit 4 → Finish**

- To create a test case:
  - right-click a file and choose **New → Test Case**
  - or click **File → New → JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



A JUnit test class

```
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public 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.

JUnit assertion methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals("message", expected, actual)`
  - Why is there no `pass` method?

ArrayIntList JUnit test

```
import org.junit.*;
import static org.junit.Assert.*;
public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
```

Testing for exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

– Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

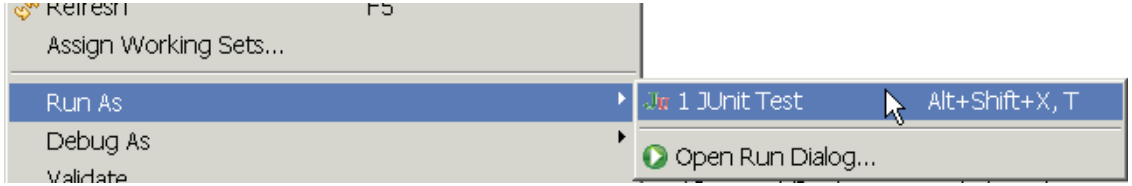
Running a test

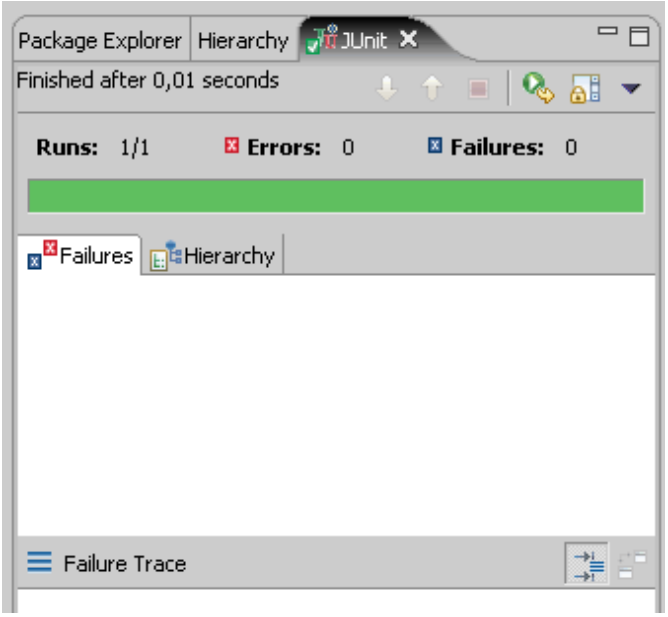
- Right click it in the Eclipse Package Explorer at left; choose:

  Run As → JUnit Test

- The JUnit bar will show **green** if all tests pass, **red** if any fail.

- The Failure Trace shows which tests failed, if any, and why.

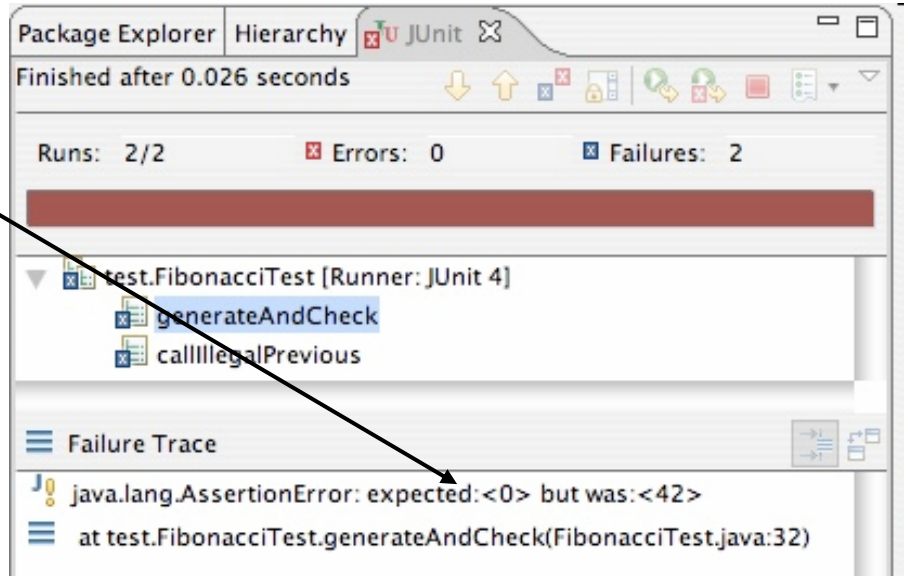




Good assertion messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

```
// JUnit will already show
// the expected and actual
// values in its output;
//
// don't need to repeat them
// in the assertion message
```



Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

– The above method will be considered a failure if it doesn't finish running within **5000 ms**

```
private static final int TIMEOUT = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

– Times out / fails after 2000 ms

Pervasive timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, d);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // almost every test should have a timeout so it can't
    // lead to an infinite loop; good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Simple JUnit Example

Note: JUnit 4 syntax

```
public class Calc
{
  static public int add (int a, int b)
  {
    return a + b;
  }
}
```

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalcTest
{
  @Test public void testAdd()
  {
    assertTrue ("Calc sum incorrect",
      5 == Calc.add (2, 3));
  }
}
```

Test values

Printed if assert fails

Expected output

Testing the Min Class

```
import java.util.*;

public class Min
{
  /**
    * Returns the minimum element in a list
    * @param list Comparable list of elements to search
    * @return the minimum element in the list
    * @throws NullPointerException if list is null or
    *         if any list elements are null
    * @throws ClassCastException if list elements are not mutually
comparable
    * @throws IllegalArgumentException if list is empty
    */
     …
}
```

Testing the Min Class

```
public static <T extends Comparable<? super T>> T min (List<? extends T>
list)
    {
       if (list.size() == 0)
       {
          throw new IllegalArgumentException ("Min.min");
       }
       Iterator<? extends T> itr = list.iterator();
       T result = itr.next();

       if (result == null) throw new NullPointerException ("Min.min");

       while (itr.hasNext())
       {   // throws NPE, CCE as needed
          T comp = itr.next();
          if (comp.compareTo (result) < 0)
          {
             result = comp;
       }  }
       return result;
}
```

MinTest Class

- Standard imports for all JUnit classes :

```
import static org.junit.Assert.*;
import org.junit.*;
import java.util.*;
```

- Test fixture and pre-test setup method (prefix) :

```
private List<String> list;   // Test fixture

// Set up - Called before every test method.
@Before
 public void setUp()
 {
    list = new ArrayList<String>();
 }
```

- Post test teardown method (postfix) :

```
// Tear down - Called after every test method.
@After
public void tearDown()
{
   list = null;   // redundant in this example
}
```

Min Test Cases: NullPointerException

```
@Test public void testForNullList()
{
   list = null;
   try {
       Min.min (list);
   } catch (NullPointerException e) {
        return;
   }
   fail ("NullPointerException expected");
}
```

This NullPointerException test uses the fail assertion

Min Test Cases: NullPointerException

```
@Test public void testForNullList()
{
   list = null;
   try {
       Min.min (list);
   } catch (NullPointerException e)
       return;
   }
   fail ("NullPointerException expe
}
```

This NullPointerException test decorates the @Test annotation with the class of the exception

```
@Test (expected =
NullPointerException.class)
public void testForNullElement()
{
     list.add (null);
     list.add ("cat");
     Min.min (list);
}
```

This NullPointerException test uses the fail assertion

Min Test Cases: NullPointerException

```
@Test public void testForNullList()
{
   list = null;
   try {
       Min.min (list);
   } catch (NullPointerException e)
       return;
   }
   fail ("NullPointerException expe
}
```

This NullPointerException test decorates the @Test annotation with the class of the exception

This NullPointerException test uses the fail assertion

```
@Test (expected =
NullPointerException.class)
public void testForNullElement()
{
    list.add (null);
    list.add ("cat");
    Min.min (list);
}
```

This NullPointerException test catches an easily overlooked special case

```
@Test (expected =
NullPointerException.class)
public void testForSoloNullElement()
{
    list.add (null);
    Min.min (list);
}
```

More Exception Test Cases for Min

```
@Test (expected = ClassCastException.class)
@SuppressWarnings ("unchecked")
public void testMutuallyIncomparable()
{
   List list = new ArrayList();
   list.add ("cat");
   list.add ("dog");
   list.add (1);
   Min.min (list);
}
```

Note that Java generics do not prevent clients from using raw types!

```
@Test (expected = IllegalArgumentException.class)
public void testEmptyList()
{
      Min.min (list);
}
```

Special case:Testing for the empty list

Remaining Test Cases for Min

```
@Test
public void testSingleElement()
{
    list.add ("cat");
    Object obj = Min.min (list);
    assertTrue ("Single Element List", obj.equals ("cat"));
}

@Test
 public void testDoubleElement()
 {
    list.add ("dog");
    list.add ("cat");
    Object obj = Min.min (list);
    assertTrue ("Double Element List", obj.equals ("cat"));
 }
```

Finally! A couple of "Happy Path" tests

Summary: Seven Tests for Min

- Five tests with exceptions
  1. null list
  2. null element with multiple elements
  3. null single element
  4. incomparable types
  5. empty elements
- Two without exceptions
  6. single element
  7. two elements

Data-Driven Tests

- Problem: Testing a function multiple times with similar values
  - How to avoid test code bloat?
- Simple example: Adding two numbers
  - Adding a given pair of numbers is just like adding any other pair
  - You really only want to write one test
- Data-driven (table-driven) unit tests call a constructor for each collection of test values
  - Same tests are then run on each set of data values
  - Collection of data values defined by method tagged with @Parameters annotation

Example JUnit Data-Driven Unit Test

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith (Parameterized.class)
public class DataDrivenCalcTest
{  public int a, b, sum;

   public DataDrivenCalcTest (int v1, int v2, int expected)
   { this.a = v1; this.b = v2; this.sum = expected; }

  @Parameters public static Collection<Object[]> parameters()
  { return Arrays.asList (new Object [][] {{1, 1, 2}, {2, 3, 5}}); }

  @Test public void additionTest()
  { assertTrue ("Addition Test", sum == Calc.add (a, b)); }
}
```

Constructor is called for each triple of values

Test 1
Test values: 1, 1
Expected: 2

Test 2
Test values: 2, 3
Expected: 5

Test method

Example JUnit Data-Driven Unit Test

- https://www.jetbrains.com/guide/java/tutorials/writing-junit5-tests/data-driven-tests/

Tests with Parameters: JUnit Theories

- Unit tests can have actual parameters
  - So far, we have only seen parameterless test methods
- Contract model: Assume, Act, Assert
  - *Assumptions* (preconditions) limit values appropriately
  - *Action* performs activity under scrutiny
  - *Assertions* (postconditions) check result

```
@Theory public void removeThenAddDoesNotChangeSet (
        Set<String> someSet, String str)  {          // Parameters!
    assumeTrue (someSet != null)                      // Assume
    assumeTrue (someSet.contains (str)) ;             // Assume
    Set<String> copy = new HashSet<String>(someSet);  // Act
    copy.remove (str);
    copy.add (str);
    assertTrue (someSet.equals (copy));               // Assert
}
```

Question: Where Do The Data Values Come From?

- Answer:
  - All combinations of values from @DataPoints annotations where assume clause is true
  - Four (of nine) combinations in this particular case
  - Note: @DataPoints format is an array

```
@DataPoints
public static String[] animals = {"ant", "bat", "cat"};


@DataPoints
public static Set[] animalSets = {
  new HashSet (Arrays.asList ("ant", "bat")),
  new HashSet (Arrays.asList (“bat", “cat", “dog“, “elk”)),
  new HashSet (Arrays.asList (“Snap”, “Crackle”, “Pop"))
};
```

Nine combinations of animalSets[i].contains (animals[j]) is false for five combinations

JUnit Theories Need BoilerPlate

```
import org.junit.*;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;
import static org.junit.Assume.*;

import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;

import java.util.*;

@RunWith (Theories.class)
public class SetTheoryTest
{
   …  // See Earlier Slides
}
```

Running from a Command Line

- This is all we need to run JUnit in an IDE (like Eclipse)
- We need a *main()* for command line execution …

Test suites

- **test suite**: One class that runs many JUnit tests.
  - An easy way to run all of your app's tests at once.

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestCaseName.class,
    TestCaseName.class,
    ...
    TestCaseName.class,
})
public class name {}
```

Test suite example

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HW2Tests {}
```

AllTests

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import junit.framework.JUnit4TestAdapter;

// This section declares all of the test classes in the program.
@RunWith (Suite.class)
@Suite.SuiteClasses ({ StackTest.class })  // Add test classes here.

public class AllTests
{
   // Execution begins in main(). This test class executes a
   // test runner that tells the tester if any fail.
   public static void main (String[] args)
   {
      junit.textui.TestRunner.run (suite());
   }

   // The suite() method helps when using JUnit 3 Test Runners or Ant.
   public static junit.framework.Test suite()
   {
      return new JUnit4TestAdapter (AllTests.class);
   }
}
```

JUnit 5 changes: min() Example

- JUnit 5 uses assertions, not annotations, for exceptions

```
@Test public void testForNullList()
{
  assertThrows(NullPointerException.class,  () -> Min.min(null));
}
```

- Other JUnit 5 differences
  - Java lambda expressions play a role
  - @Before, @After change to @BeforeEach, @AfterEach
  - imports, some assertions change
  - Test runners change (no simple replacement for AllTests.java)
  - @Theory construct moved to 3rd party extensions
    - google "property based testing"
- See MinTestJUnit5.java on the book website

How to Run Tests

- JUnit provides test drivers
  - Character-based test driver runs from the command line
  - GUI-based test driver-*junit.swingui.TestRunner*
    - Allows programmer to specify the test class to run
    - Creates a "Run" button
- If a test fails, JUnit gives the location of the failure and any exceptions that were thrown

JUnit Resources

- Some JUnit tutorials
  - http://open.ncsu.edu/se/tutorials/junit/
    - (Laurie Williams, Dright Ho, and Sarah Smith)
  - http://www.lalilуna.de/eclipse-junit-testing-tutorial.html
    - (Sascha Wolski and Sebastian Hennebrueder)
  - http://www.diasparsoftware.com/template.php?content=jUnitStarterGuide
    - (Diaspar software)
  - http://www.clarkware.com/articles/JUnitPrimer.html
    - (Clarkware consulting)
  - - https://www.jetbrains.com/guide/java/tutorials/writing-junit5-tests/data-driven-tests/
- JUnit: Download, Documentation
  - http://www.junit.org/

JUnit for other languages

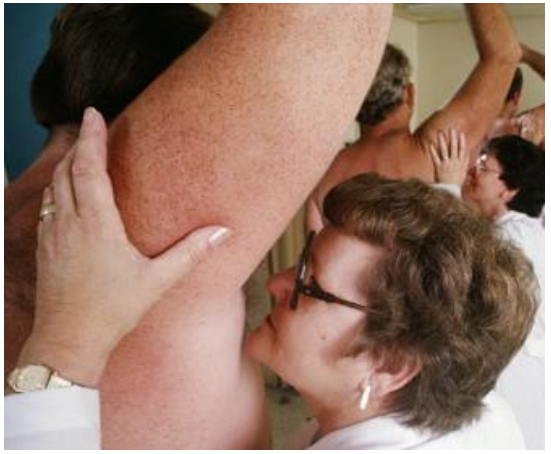
- **NUnit**
  - An open-source unit testing framework for the **.NET Framework** and Mono.
  - https://nunit.org/
- **Unittest**
  - Originally inspired by JUnit
  - https://docs.python.org/3/library/unittest.html
- **Pytest**
  - https://docs.pytest.org/en/8.0.x/

Trustworthy tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.
- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.
- Torture tests are okay, but only *in addition to* simple tests.

Test case "smells"

- Tests should be self-contained and not care about each other.



- "**Smells**" (bad things to avoid) in tests:
  - *Constrained test order* : Test A must run before Test B.
    (usually a misguided attempt to test order/flow)
  - *Tests call each other* : Test A calls Test B's method
    (calling a shared helper is OK, though)
  - *Mutable shared state* : Tests A/B both use a shared object.
    (If A breaks it, what happens to B?)

Bugs in tests

- hard to find
  - developers assume that tests are correct
- manifest in odd ways
  - sometimes test initially passes, then begins to fail much later
    - code under test may have been altered in a subtle way
    - test case may have relied on invalid assumptions
    - API of code under test may have changed
- often test wasn't written by developer
  - bug assigned back and forth

What's wrong with this?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
}
```

Well-structured assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());  // expected
        assertEquals(2, d.getMonth());    // value should
        assertEquals(19, d.getDay());     // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

JUnit summary

- Tests need *failure atomicity* (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.
- Always use a `timeout` parameter to every test.
- 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.
- Use helpers, `@Before` to reduce redundancy between tests.

Bugs

- it is very likely that our code will some bugs in it
  - bugs are inevitable in any complex software system
  - a bug can be very visible or can hide in your code until a much later date

- **debugging**: The process of finding and removing causes of failures in software.
  - **debugger**: A tool for tracing program execution to find bugs. (preferred over System.out.println statements)

Debugging in Eclipse

- To show line numbers in Eclipse:
  - **Window -> Preferences -> General -> Editors -> Text Editors -> Show Line Numbers**
- To set Eclipse UI to 'Debug' mode:
  - switch to the 'Debug' perspective
  - **Window -> Open Perspective -> Debug**
- While in Debug mode:
  - any exception will halt the program
  - you can examine the state of the program at the point of its crash
  - you can set *breakpoints* and/or *step* through execution (see next slide)

Breakpoints

- **breakpoint**: a line at which the execution of the program will halt
  - useful so that you can examine variables, objects, and other data in mid-execution
  - can step (execute each statement one at a time) to see when and how a program becomes incorrect

- To set a breakpoint in Eclipse:
  - double-click the gray margin left of the desired line
  - (or right-click it and choose Toggle Breakpoint)

QA practices

- debugging is only one part of the larger goal of assuring that our software meets quality standards
  - **quality assurance (QA)**: A planned and systematic pattern of all actions necessary to provide confidence that adequate technical requirements are established, that products and services conform to established technical requirements, and that satisfactory performance is achieved.

- we can hunt down the cause of a known bug using print statements or our IDE's **debugger** ...
  but how do we discover all of the bugs in our system, even those with low visibility?
  - ANSWER: *testing* and Quality Assurance practices

Testing exercise

- Imagine that we have a **Date** class with working methods like isLeapYear(year), getDaysInMonth(month, year), and addDays(days).
  - Let's talk about the pseudo-code for the algorithm for an addDays(days) method that moves the current Date object forward in time by the given number of days. A negative value moves the Date backward in time.
  - Come up with a set of test values for the getDaysInMonth method to test its correctness.
  - Come up with a set of test values for your addDays method to test its correctness.

JUnit exercise

Using our Date class, let's:

- Enforce a contract with invariants on our Date objects, so that they can never hold an invalid state.
- Write an addDays method that takes an int and adjusts the current Date's date by the given number of days forward in time. If the argument is negative, go backward in time.
- Write a compareTo method that compares Dates chronologically.
- Write an equals method that tells whether two Dates store the same date.
- Add getDaysFrom method that takes another Date object and returns the number of days this Date is away from the date of the given other Date object.
  - It is therefore implied that after a call of *this.addDays(-this.getDaysFrom(other))*, the statement *this.equals(other)* would be true.

- Write **JUnit** test code that checks each of the above methods for correctness. Utilize intelligent testing techniques as presented in class.

When to test

- **Test-last:** write code under test, then write test code
- **Test-first:** write test code, then write code under test (test as spec)
  - Test-driven development (TDD),
  - Behavior driven development (BDD).
  - Will be discussed in the next lecture!
- **Regression test:** when it breaks, write a test first to fix it
- **Test-coverage:** when nothing is broken but want to get coverage

  - Develop the **Date** class program in the previous slide with all above approaches.

Summary

- The only way to make testing efficient as well as effective is to automate as much as possible
- Test frameworks provide very simple ways to automate our tests
- It is no "silver bullet" however … it does not solve the hard problem of testing :

What test values to use ?

- This is test design … the purpose of test criteria