Introduction to Software Testing
(*2nd edition*)
Chapter 3

Test Automation

Paul Ammann & Jeff Offutt

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

Updated October 2018

What is Test Automation?

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

Software Testability (3.1)

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

Components of a Test Case (3.2)

- A test case is a multipart artifact with a definite structure

- Test case values

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

- Expected results

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

  - A *test oracle* is a mechanism that 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 Framework (3.3)

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

What is JUnit?

- Open source Java testing framework used to write and run repeatable automated tests
- JUnit is open source (junit.org)
- JUnit features include:
  - Assertions for testing expected results
  - Test fixtures for sharing common test data
  - Graphical and textual test runners
    - The ability to run tests from either a command line or a GUI
- 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

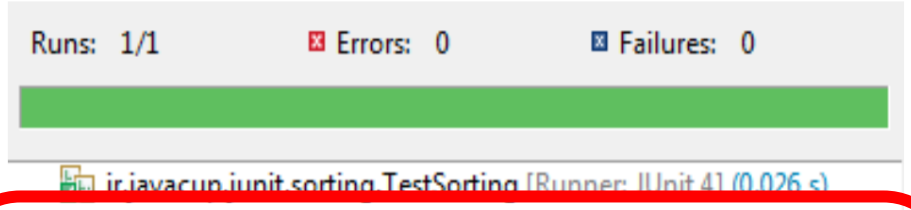
JUnit Tests

- 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

مثالی از آزمون واحد در JUint

- هر تست در قالب یک test method (متد آزمون) نوشته می‌شود.
  - در اینجا متد testSort
- در محیط‌های برنامه‌نویسی (مثل Eclipse)، برای اجرای متدهای تست نیازی به متد main نیست و با کمک TestRunner در خود IDE اجرا می‌شوند.
  - برای اجرا در command line، باید خودمان یک متد main جهت فراخوانی TestRunner بنویسیم.
- هر متد تست با حاشیه @Test مشخص می‌شود.

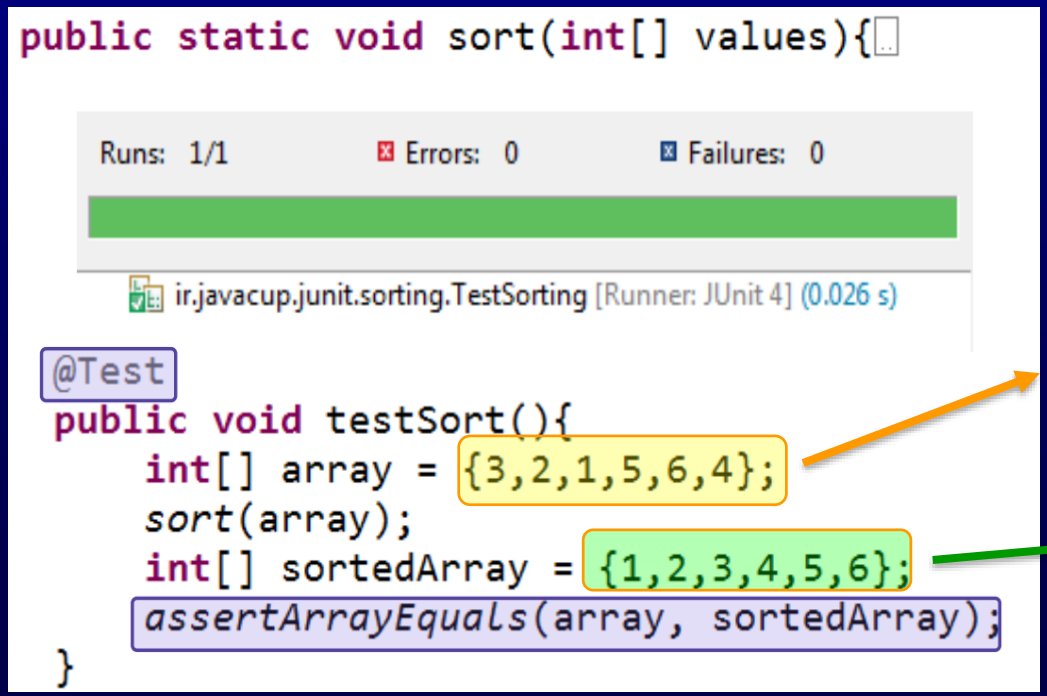
```
public static void sort(int[] values){
```




```
@Test
public void testSort(){
    int[] array = {3,2,1,5,6,4};
    sort(array);
    int[] sortedArray = {1,2,3,4,5,6};
    assertArrayEquals(array, sortedArray);
}
```

مثالی از آزمون واحد در JUint

- موفقیت‌آمیز بودن آزمون به صورت خودکار بررسی می‌شود
  - نه به صورت دستی: سندروم printl
- نتیجه آزمون در Junit به کمک assertion بررسی می‌شود
  - مثال: assertEquals



داده ورودی
(Test case value)

خروجی مورد انتظار
(Expected result)

روال اجرای خودکار یک تست

- تعیین یک ورودی برای متد مورد آزمون (test case value)
- تعیین خروجی مورد انتظار برای این ورودی (expected result)
- متد آزمون با این ورودی فراخوانی می‌شود
- خروجی متد با خروجی مورد انتظار مقایسه می‌شود (assertion)
- اگر اجرای تست موفقیت‌آمیز باشد: تست pass شده است
- اگر اجرای تست موفقیت‌آمیز نباشد: تست fail شده است

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)*

Some JUNIT ASSERTIONS

- assertNull(x)
- assertNotNull(x)
- assertTrue(boolean x)
- assertFalse(boolean x)
- assertEquals(x, y)
  - x.equals(y)
- assertSame(x, y)
  - x ==y
- assertNotSame(x, y)
- fail()
- ,…

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

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 allocated or initialized in an @Before method
- Can be deallocated or reset in an @After method

Testing the Min Class

The code of Min class

```
import java.util.*;

public class Min
{
  /**
   * Returns the mininum 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;
}
```

The code of min method in Min class

آشنایی مختصر با کد متد min در کلاس Min

```
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;
}
```

اگر لیست خالی باشد، خطای آرگومان غیرقانونی (توجه: اگر لیست نال باشد، اجرای دستور اول منجر به NullPointer خواهد شد)

تکرارگری که لیست را از اولین عضو پیمایش می‌کند

اولین عضو لیست در متغیر result ذخیره می‌شود

اگر اولین عضو لیست نال باشد، باید خطای NullPointer بدهد

در این محل قطعه کدی نوشته می‌شود که اگر عضو بعدی لیست نال باشد، باید خطای NPE بدهد و اگر دو المان متوالی، هم‌نوع نباشند باید خطای CCE بدهد

آشنایی مختصر با کد متد min در کلاس Min

```
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;
}
```

اگر لیست خالی باشد، خطای آرگومان غیرقانونی (توجه: اگر لیست نال باشد، اجرای دستور اول منجر به NullPointer خواهد شد)

تکرارگری که لیست را از اولین عضو پیمایش می‌کند

اولین عضو لیست در متغیر result ذخیره می‌شود

اگر اولین عضو لیست نال باشد، باید خطای NullPointer بدهد

عضو بعدی لیست در متغیر comp ذخیره می‌شود

اگر comp از result، کوچکتر باشد، مقدار comp جایگزین مقدار قبلی result می‌شود

کوچکترین عضو لیست برگردانده می‌شود

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) :

```
public class MinTest
 {
   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 each test method.
   @After
   public void tearDown()
   {
      list = null;   // redundant in this example
   }
   …
 }
```

Test Fixture in MinTest Class

- در این مثال، داده مشترک بین تست‌متدهای مختلف، لیستی از اشیا است.
- مثلا این‌جا می‌خواهیم با لیستی از رشته‌ها، متد Min.min را تست کنیم.
- در این دستور، صرفا یک شی با نام list از جنس List<String> تعریف می‌شود که اشاره‌گر آن به nothing است (یعنی لیست نال)

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

// Set up – Called before every test method.
@Before
 public void setUp()
 {
     list = new ArrayList<String>();
 }
// Tear down – Called after every test method.
@After
public void tearDown()
{
   list = null;   // redundant in this example
}
```

Test Fixture in MinTest Class

The “ @Before” method puts the test object into a proper initial state by creating a new List object.

- در این دستور، یک لیست خالی ایجاد می‌شود و به این ترتیب لیست جدیدی بوجود می‌آید.

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

// Set up – Called before every test method.
@Before
public void setUp()
{
    list = new ArrayList<String>();
}
// Tear down – Called after every test method.
@After
public void tearDown()
{
   list = null;   // redundant in this example
}
```

Test Fixture in MinTest Class

- "@After" method encodes the postfix part of the test.

- It resets the state of the test object by setting the object reference to null.

- Here , @After method is redundant since @Before method resets the reference anyway, but good engineering practice is to be conservative:

"measure twice, cut once."

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

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

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

This NullPointerException test catches an easily overlooked special case

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 don't 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 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

JUnit Parameterized mechanism implements data-driven testing

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

  public DataDrivenCalcTest (int a, int b, int sum)
  { // Note Constructor
    this.a = a;
    this.b = b;
    this.sum = sum;
  }

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

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

Constructor

table of inputs and expected outputs

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[]> calValues()
  { 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

Tests with Parameters: JUnit Theories

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

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

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 values 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
}
```

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

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


   @Theory
   public void removeThenAddDoesNotChangeSet
                  (Set<String> set, String string)  // Parameters!
   {
      assumeTrue (set != null);            // Assume
      assumeTrue (set.contains (string));  // Assume
// Uncomment the following line to see which tests that pass the precondition
// System.out.println ("Set, string: " + set + ", " + string);
      Set<String> copy = new HashSet<String>(set);   // Act
      copy.remove (string);
      copy.add (string);
      assertTrue (set.equals (copy));      // Assert
    }
}
```

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 …
  - See Next Slide…

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 ({ MinTest.class })  // Add test classes here.
                                         // Add more test classes by inserting a .class file name
                                         // inside the curly brackets, separate by commas.

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 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 third-party extension APIs
    - google "property based testing"
- See MinTestJUnit5.java on the book website

How to Run Tests

- A test framework should support for the test driver
- A test driver runs a test set by executing the software repeatedly on each test
- JUnit provides test drivers
  - Character-based test driver runs from the command line
    - *junit.textui.TestRunner.run (suite())*
  - 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.laliluna.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)
- JUnit: Download, Documentation
  - http://www.junit.org/

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
- However test automation is not "silver bullet" … it does not solve the hard problem of testing :

What test values to use ?

- This is the subject of test design … the purpose of test criteria
- *[After test driven development in Chapter 4, Chapter 5 discusses criteria-based test design in general, then the next four chapters give specific test criteria for designing tests]*

توضیحات تکمیلی

توضیحات تکمیلی در رابطه با

JUnit Theories
و
Property-based Testing

Adding Parameters to Unit Tests

- Allowing the use of parameters in test methods is extremely powerful
- The JUnit Theory mechanism allows test engineers to define test methods with parameters
- To better understand JUnit Theories, we first must separate example-based testing and property-based testing

Adding Parameters to Unit Tests

- Example-based vs Property-based testing
  - Example-based tests hinge on a single scenario. Property-based tests get to the root of software behavior across multiple parameters
  - Traditional (example-based) testing specifies the behavior of your software by writing examples of it—each test sets up a single concrete scenario and asserts how the software should behave in that scenario
  - Property-based tests take these concrete scenarios and generalize them
- To prove the correctness of a code:
  - Property based testing relies on properties. It checks that a function, program or whatever system under test abides by a property

Adding Parameters to Unit Tests

- A property is just something like:

```
for all (x, y, ...)
such as precondition(x, y, ...) holds
property(x, y, ...) is true
```

- Here is a simple property:

```
for all (a, b, c) strings
the concatenation of a, b and c always contains b
```

Adding Parameters to Unit Tests

Universal quantification assertion

$$\forall x \in X \bullet P(x) \rightarrow Q(x)$$

- The normal approach to prove the correctness of this assertion (i.e., to show that the assertion is a theorem) is using a mathematical proof
- Testing is usually considered ill-suited to showing such an assertion
  - The domain of X in this example, is often very large, and hence X cannot be enumerated exhaustively
- Unit tests with parameters explore a middle ground between mathematical proof and ordinary testing
  - They promise to use the practical power of testing, at least partially, to demonstrate the validity of universally quantified assertions

Adding Parameters to Unit Tests

$$\forall x \in X \bullet P(x) \rightarrow Q(x)$$

- Unit tests with parameters
  - pattern: Precondition, Action, Postcondition
  - for all possible combinations of parameters that satisfy the preconditions, the postcondition is also true for whatever action the test implements
- Of course, there is no way to try all possible values, or else our tests will never finish running
- Test engineer should not be concerned with where values come from, or how many there are, when specifying the theory itself
  - These concerns are addressed differently in different approaches to test methods with parameters
- You should focus on writing a valid test method—and leave it up to the test engine to find a counterexample if possible

Theories in JUnit 5?

- Class Theories works in JUnit 4
- JUnit 5's default test engine: Jupiter
  - JUnit Jupiter is the API for writing tests using JUnit 5
- The theory concept is not being supported by Jupiter
- In JUnit 5, the modern concept of property-based testing is supported by 3rd party extension APIs
  - For example jqwik - a 3rd party Test engine for JUnit 5

```
@Property
boolean joiningTwoLists(
    @ForAll List<String> list1,
    @ForAll List<String> list2
) {
    List<String> joinedList = new ArrayList<>(list1);
    joinedList.addAll(list2);
    return joinedList.size() == list1.size() + list2.size();
}
```

آشنایی بیشتر با کد متد min

آشنایی با کد متد min در کلاس Min

- در زبان جاوا برای مقایسه میان دو شی هم‌نوع، یک متد از پیش آماده تحت عنوان compareTo وجود دارد که بصورت انتزاعی (فاقد پیاده‌سازی) در یک کلاس واسط تحت عنوان Comparable نوشته شده است.
- کلاس های مختلف از قبیل Integer و String، از کلاس Comparable ارث‌بری دارند و متد compareTo را به طرق مختلف پیاده‌سازی کرده‌اند و قابل استفاده است.
- برای هر کلاس جدیدی که خودمان می‌نویسیم (مثلا کلاس کتاب) نیز می‌توانیم آن را به گونه‌ای تعریف کنیم که از Comparable ارث‌بری داشته باشد و متد compareTo را بر حسب نیاز (مثلا با مقایسه میان سال انتشار کتاب‌ها)، پیاده‌سازی کند.
- پیاده‌سازی متد compareTo باید طبق این اصل باشد:

x.compareTo(y)
if x > y, it returns positive number
if x < y, it returns negative number
if x == y, it returns 0

آشنایی با کد متد min در کلاس Min

```
public static <T extends Comparable<? super T>> T min (List<? extends T> list)
{…}
```

- آرگومان ورودی متد min: List<? extends T> list
  - لیستی از اشیا با نوع جنریک T یا زیرنوع‌های T (هر آنچه که فرزند T است)
  - مثلا لیستی از اعداد صحیح (Integer)

```
List<Integer> list1 = new ArrayList<Integer>();
Min.min(list1) // it is ok
```

  - یا لیستی از رشته‌ها (String)

```
List<String> list2 = new ArrayList<String>();
Min.min(list2) // it is ok
```

  - یا لیستی از کلاس‌های تعریف شده توسط کاربر (مثلا لیستی از Book یا فرزندان آن)

```
List<Book> list3 = new ArrayList<Book>();
Min.min(list3) // it is ok
```

آشنایی با کد متد min در کلاس Min

```
public static <T extends Comparable<? super T>> T min (List<? extends T> list)
{…}
```

- نوع بازگشتی متد min: `<T extends Comparable<? super T>> T`
  - شیای از نوع **جنریک** T (کوچکترین عضو list)
- در اینجا، نوع جنریک T (کلاس T) یا یکی از سوپرکلاس‌های آن، از Comparable باید ارث‌بری داشته باشد (آن را extend کند) تا بتواند به متد compareTo دسترسی داشته باشد و از آن استفاده کند:

```
<T extends Comparable<? super T>>
```

توضیحات جانبی درباره:
نوع جنریک، نوع خام
و
Lambda Expression
در جاوا

Generic Type

- Generics means parameterized types.
  - With a suitable type argument (e.g. List<String>).
- The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces.
- We use <> to specify parameter types in generic class creation:

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

- Example: `List<Integer> list = new ArrayList<Integer>();`

Raw Type (non-generic)

- A "raw" type is a class which is non-generic with "raw" Objects, rather than type-safe generic type parameters.
  - Without specifying a type argument(s) for its parameterized type(s).
  - e.g. using List instead of List<String>.
- Example:

```
List  list = new ArrayList();
```

Generic type vs Raw type (non-generic)

- Raw types are used for backwards compatibility.
  - Using these classes allow legacy code to still compile.
- Their use in new code is not recommended.
  - Generic classes allow compiler to enforce better type-safety, which in turn may improve code quality and lead to catching potential problems earlier (compile-time errors instead of run-time errors).

Example:

```
// Java program to demonstrate that NOT using
// generics can cause run time exceptions
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        // Creatinga an ArrayList without any type specified
        ArrayList al = new ArrayList();
        al.add("Sachin");
        al.add("Rahul");
        al.add(10); // Compiler allows this

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);

        // Causes Runtime Exception
        String s3 = (String)al.get(2);
    }
}
```

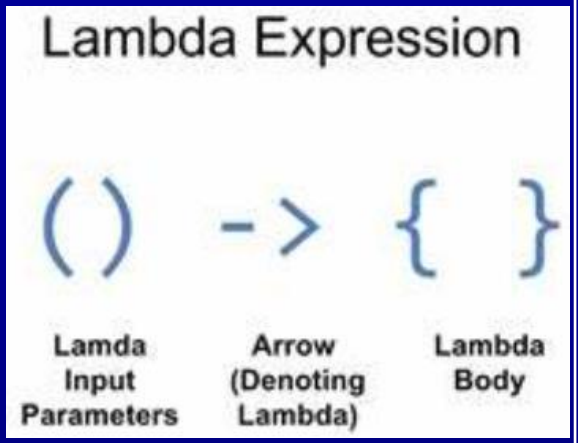
Raw Type

Generic Type

```
List<String> names = new ArrayList<String>();
names.add("John");          // OK
names.add(new Integer(1));  // compile error
```

Java Lambda Expression

- Lambda Expressions were added in Java 8
- A lambda expression is a short block of code which takes in parameters and returns a value
- Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method



Lambda Syntax

- No arguments:
```
() -> System.out.println("Hello")
```
- One argument:
```
s -> System.out.println(s)
```
- Two arguments:
```
(x, y) -> x + y
```
- With explicit argument types:
```
(Integer x, Integer y) -> x + y
```
- Multiple statements:
```
(x, y) -> {
    System.out.println(x);
    System.out.println(y);
    return (x + y);
}
```