COMP5111 – Fundamentals of Software Testing and Analysis

Unit Testing & JUnit (with Theories)

Shing-Chi Cheung

Computer Science & Engineering

HKUST

Facts of a real enterprise project

- Eclipse is known for its high reliability
- Defect density of Eclipse 3.0
  - Scariest defects: 30 per million LOC
  - Scary: 160 per million LOC
  - Troubling: 480 per million LOC
  - Of concern: 6,000 per million LOC

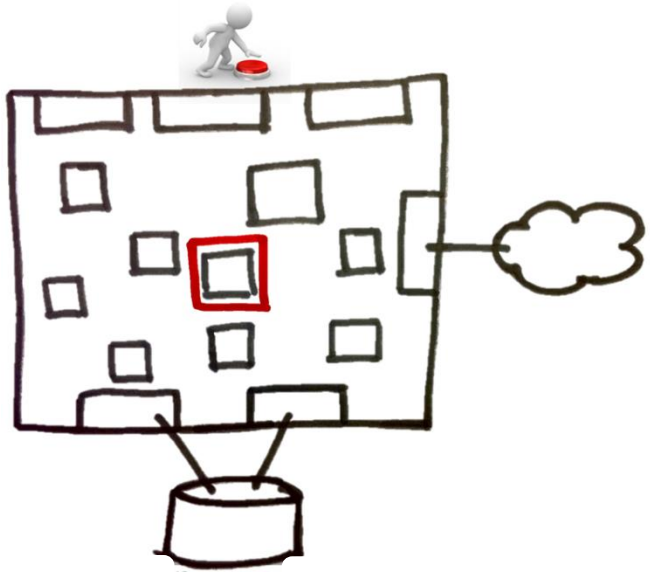
Lines of code

What is unit testing?

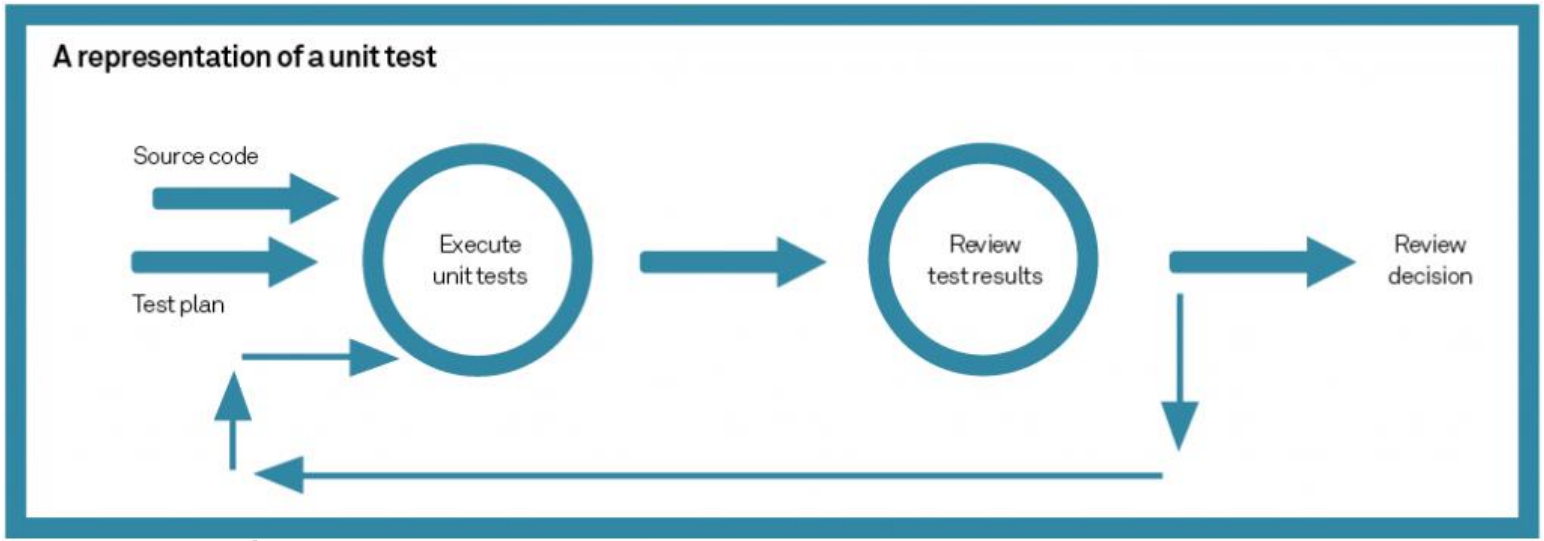
- A **test unit** typically refers to a class or one of its methods
- A unit test case specifies a test execution of the unit
  - **class under test (CUT)**: class is a test unit
  - **method under test (MUT)**: method is a test unit
- A unit **test suite** is a selected set of unit test cases

What is unit testing?

- To isolate each part of program and show that individual parts are correct
- Unit testing is a de facto practice in DevOp adopted by most practitioners to test their code
- JUnit is a bundled plug-in in most Java IDEs, including Eclipse and IntelliJ



How to conduct unit testing



Test plan
- Test objectives: regression/functional/security/GUI/…
- Test coverage criterion and threshold: statement/branch/exception/activity/all-def/…
- Test suites
- Resources and time budget

Why unit testing?

Verify the unit works as intended	Structure code as testable units (especially with TDD)	Serve as requirements documentation	Raise early alerts for bad changes
✓			

What is JUnit ?

- JUnit is a regression testing framework written by Kent Beck and Enrich Gamma
- Its latest version is 5
- It is a de facto framework used by Java practitioners to test their code
- We will use
  - JUnit 5 in class to learn latest features
  - JUnit 4 in assignments for tool compatibility

Kent Beck

Enrich Gamma

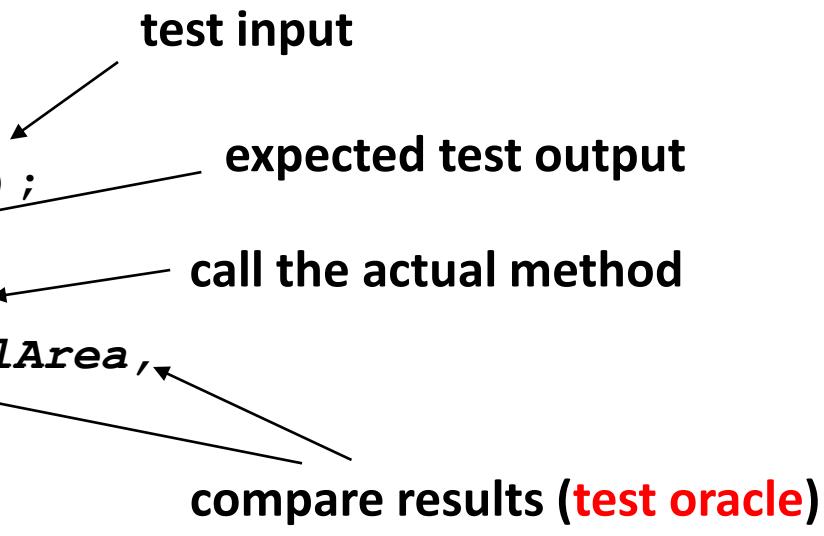
Requirement of a JUnit Test Case

```
@Test
public void testGetArea() {
  Rectangle r = new Rectangle(2, 4);
  double expectedArea = 2 * 4;
  double actualArea = r.getArea();
  assertEquals(expectedArea, actualArea,
  0.000001, "getArea fails");
}
```

- Test runs automatically with little human participation
- Results are either pass or fail
- Results are reproducible and consistent

What is in a test case?

```
@Test
public void testGetArea() {
  Rectangle r = new Rectangle(2, 4);
  double expectedArea = 2 * 4;
  double actualArea = r.getArea();
  assertEquals(expectedArea, actualArea,
  0.000001, "getArea fails");
}
```



- Describe the method's input
- Describe the expected output
- Call the method and observe actual output
- Compare expected with actual output

Putting it in Eclipse

```
public class RectangleTest {

 Rectangle r = new Rectangle(2, 4);
 final int height = 2, width = 4;
 final double tolerance = 0.000001;

@Test
public void testGetArea() {
  double expectedArea = width * height;
  double actualArea = r.getArea();
  assertEquals(expectedArea, actualArea,
      tolerance, "getArea fails");
}
```

Putting it in Eclipse

```
public class RectangleTest {

 Rectangle r = new Rectangle(2, 4);
 final int height = 2, width = 4;
 final double tolerance = 0.000001;

@Test
public void testGetArea() {
  double expectedArea = width * height;
  double actualArea = r.getArea();
  assertEquals(expectedArea, actualArea,
      tolerance, "getArea fails");
}
```

```
@Test
public void testResize() {
  r.resize(0.25, 2);
  assertEquals(0.25*height, r.getHeight(), tolerance,
      "resize - height fails");
  assertEquals(2*width, r.getWidth(), tolerance,
      "resize - width fails");
}
… // other tests
}
```

BeforeEach and AfterEach test case

```
@BeforeEach
public void setUp() throws Exception {
        // create objects and common variables for your tests here
        // setUp() is called before EACH test case is run
}

@AfterEach
public void tearDown() throws Exception {
        // put code here to reset or release objects, e.g., p1=null;
        // to garbage collect unused objects or resources
        // tearDown() is called after EACH test case is run
}
```

How is a test suite executed?

1. `setUp()`
2. `Test case 1`
3. `tearDown()`
4. `setUp()`
5. `Test case 2`
6. `tearDown()`

... until all test cases are executed

→ **Test cases should be independent of each other**

→ **JUnit does not guarantee the execution order of test cases**

Test Oracle: expected == actual?

- To compare any variables of the 8 Java primitive types (i.e., boolean, byte, short, int, long, char, float, double) use:

```
assertEquals(expected, actual)
```

- The (optional) last argument is a string printed when the assertion fails:

```
assertEquals(expected, actual, message)
```

Comparisons for Primitive Types

- assertEquals(int expected, int actual)
- assertEquals(boolean expected, boolean actual)
- assertEquals(byte expected, byte actual)
- assertEquals(char expected, char actual)
- assertEquals(short expected, short actual)
- assertEquals(long expected, long actual)
- assertEquals(double expected, double actual, double tolerance)
- assertEquals(float expected, float actual, float tolerance)
- Each of them supports a variant method by specifying a failure message string as the first formal parameter.

More Assertions

- assertSame(expected, actual)
- assertSame(expected, actual, message)
- assertTrue(condition)
- assertTrue(condition, message)
- fail()
- fail(message)
- failNotEquals(expected, actual, message)
- failNotSame(expected, actual, message)
- failSame(message)
- assertFalse(condition)
- assertFalse(condition, message)
- assertNotNull(object)
- assertNotNull(object, message)
- assertNotSame(expected, actual)
- assertNotSame(expected, actual, message)
- assertNull(object)
- assertNull(object, message)

API Documentation (JUnit 4): https://junit.org/junit4/javadoc/4.10/
API Documentation (JUnit 5) https://junit.org/junit5/docs/current/api/index.html?overview-summary.html

Eclipse Demonstration

```
public class RectangleTest {

@BeforeAll
public static void setUpBeforeClass() throws Exception {
}
@AfterAll
public static void tearDownAfterClass() throws Exception {
}
@BeforeEach
public void setUp() throws Exception {
  r = new Rectangle(4, 4);
}
@AfterEach
public void tearDown() throws Exception {
  r = null;
}
```

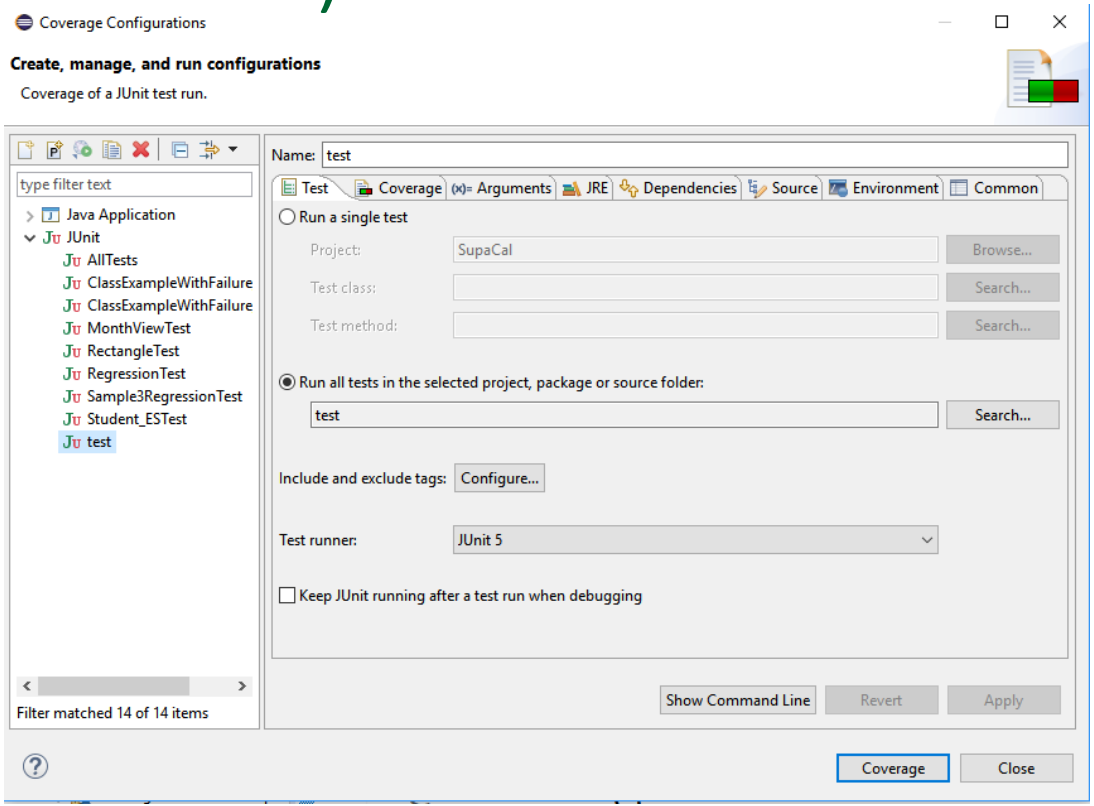
```
@Test
public void testGetArea() {
  double expectedArea = width * height;
  double actualArea = r.getArea();
  assertEquals(expectedArea, actualArea,
                tolerance, "getArea fails");
}
@Test
public void testResize() {
  r.resize(0.25, 2);
  assertEquals("resize - height fails", 0.25*height,
  r.getHeight(), tolerance);
  assertEquals(2*width, r.getWidth(),
                tolerance, "resize - width fails");
}
  Rectangle r;
  final int height = 4, width = 4;
  final double tolerance = 0.000001;
}
```

Test Suites (JUnit 5)

test folder -> Coverage As -> Coverage Configurations …

@Tag: tests filtering - Divide tests into groups using different tags

RectangleTest.java



Test Suite (JUnit 5)

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.IncludeTags;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
//@IncludeTags("Small")
@SelectClasses({
   GeometricObjectTest.class
   , RectangleTest.class
   …
})
public class TestMain { }
```

AllTests5

Test Suites (JUnit 4)

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
  GeometricObjectTest.class,
  RectangleTest.class,
  ...
})
public class TestMain { }
```

AllTests

Selected Useful JUnit 5 Features

Inject Dependency using @ExtendWith

```
@ExtendWith(MockitoExtension.class)
class MyMockitoTest {
   @BeforeEach
   void init(@Mock Person person) {
      when(person.getName()).thenReturn("Dilbert");
   }
   @Test
   void simpleTestWithInjectedMock(@Mock Person person) {
      assertEquals(person.getName(), "Dilbert");
   }
}
```

Person is a class whose getName() has not been implemented

Flexibility to mock getName() to return a different value in another test class

return the mocked value

Usage:
To supply a mocked test value for getName() when it has not been implemented or is to be debugged or returns a different value at different executions, such as using system time or random.

@RepeatedTest

```
@RepeatedTest(5000) // repeat a test multiple times for burn-in testing
public void testGetPerimeter() {
        Rectangle r1 = new Rectangle();
        double width = Math.random()*10000;
        double height = Math.random()*10000;
        r1.setWidth(width);
        r1.setHeight(height);
        double expectedPerimeter = 2*(width+height);
        double actualPerimeter = r1.getPerimeter();
}
```

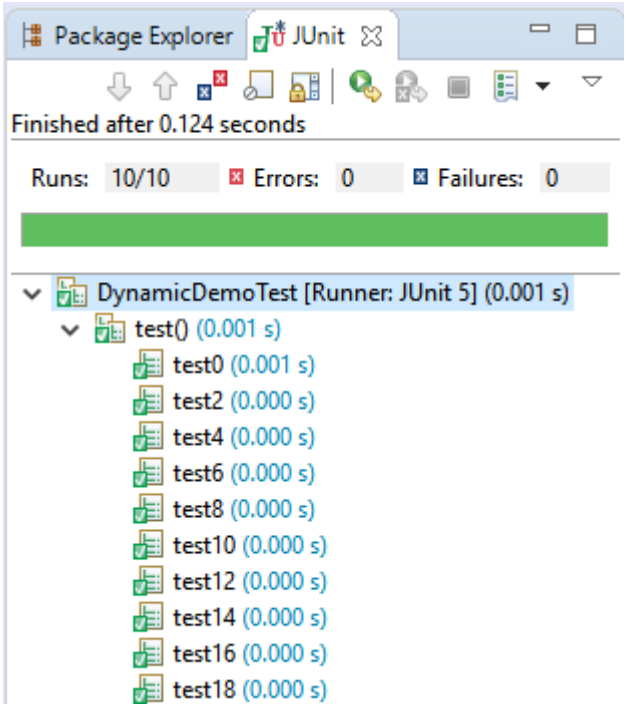
RectangleTest

Dynamic Tests

```
class DynamicDemoTest {

  @TestFactory
  Stream<DynamicTest> test() {
    // Generates tests for the first 10 even integers.
    return IntStream
        .iterate(0, n->n+2)
        .limit(10)
        .mapToObj(n ->
            DynamicTest.dynamicTest("test"+n, () -> assertTrue(n%2 ==0)));
  }
}
```

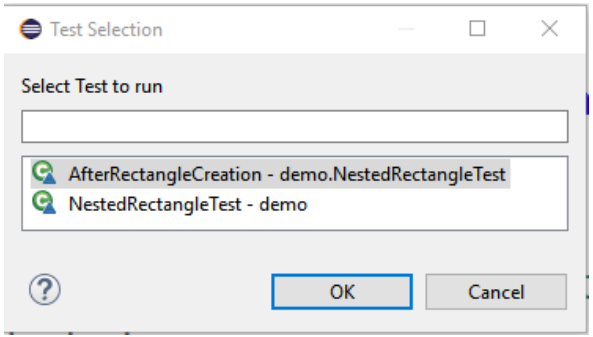
specify the test scripts to be generated

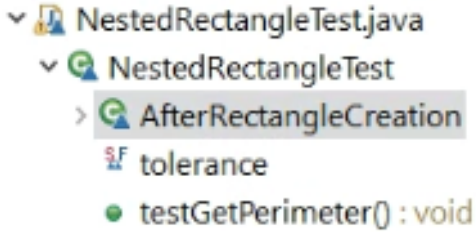


DynamicDemoTest

@Nested, assertAll() and assumingThat()

- Nest related tests into an inner class
- Support multiple nesting levels





NestedRectangleTest
NestedLambdaAssumingThatRectangleTest

The Use of AssertAll

- Sometimes we want to test a set of fields as one unit.
  - E.g., The city, street and number in an address.

```
Address address = unitUnderTest.methodUnderTest();
assertEquals("Redwood Shores", address.getCity());
assertEquals("Oracle Parkway", address.getStreet());
assertEquals("500", address.getNumber());
```

  - The above test code does not work because JUnit will not check the second assertEquals if the first fails.

The Use of AssertAll

- We want something like the following:

```
Address address = unitUnderTest.methodUnderTest();
assertEquals("Redwood Shores", address.getCity());
assertEquals("Oracle Parkway", address.getStreet());
assertEquals("500", address.getNumber());
```

☞ org.opentest4j.MultipleFailuresError:
Should return address of Oracle's headquarter (3 failures)
expected: <Redwood Shores> but was: <Walldorf>
expected: <Oracle Parkway> but was: <Dietmar-Hopp-Allee>
expected: <500> but was: <16>

The Use of AssertAll

- We want something like the following:

```
Address address = unitUnderTest.methodUnderTest();
assertEquals("Redwood Shores", address.getCity());
assertEquals("Oracle Parkway", address.getStreet());
assertEquals("500", address.getNumber());
```

```
Address address = unitUnderTest.methodUnderTest();
assertAll("Should return address of Oracle's headquarter",
    () -> assertEquals("Redwood Shores", address.getCity()),
    () -> assertEquals("Oracle Parkway", address.getStreet()),
    () -> assertEquals("500", address.getNumber()));
```

NestedLambdaAssertAllRectangleTest

assertThrows

- JUnit 4 supports exception testing at test method level.
  - @Test(expected = SomeException.class)
- JUnit 5 introduces assertThrows() to support exception testing at statement level
  - assertThrows()

RectangleWithJUnit4ExceptionTest
RectangleWithJUnit5ExceptionTest

Parameterized JUnit Tests & JUnit Theories

Write compact test cases for multiple scenarios

Parameterized JUnit Tests

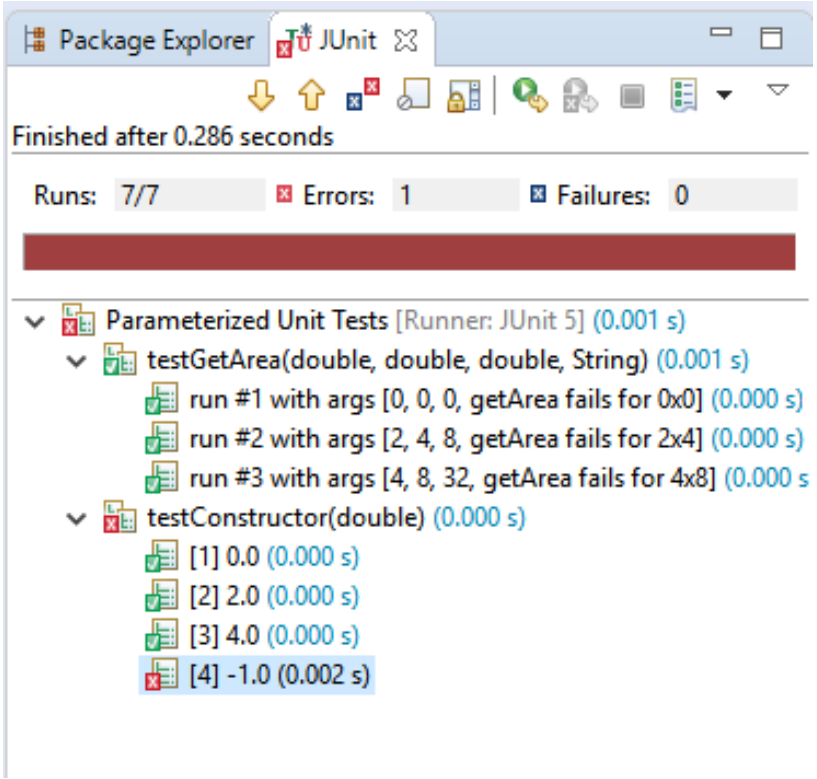
- Need to test a family of scenarios
  - Avoid writing a family of test cases
  - Add new scenarios without adding test cases
- Prepare a family of parameters
- Write test cases that act upon these parameters

Parameterized JUnit Tests

```
@ParameterizedTest
@ValueSource(doubles = {0, 2, 4, -1})  // support single parameter for strings, ints, longs, doubles
public void testConstructor(double width) throws Exception {
   var r = new Rectangle(width, 2*width);
   assertTrue(r != null && r instanceof Rectangle);
}
@ParameterizedTest(name = "run #{index} with args [{arguments}]")
@CsvSource( {
   "0, 0, 0, 'getArea fails for 0x0'",
   "2, 4, 8, 'getArea fails for 2x4'",
   "4, 8, 32, 'getArea fails for 4x8'"
})
// @CsvFileSource(resources = "parameters.csv")
   public void testGetArea(double width,
    double height, double expectedArea, String
    msg) {
   Rectangle r1 = new Rectangle(width, height);
   assertEquals(expectedArea, r1.getArea(),
      tolerance, msg);
   }

   final static double tolerance = 0.0001;
}
```

Parameterized JUnit Tests

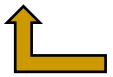



```
public Rectangle(double width, double height) {
  this.width = width;
  this.height = height;
  if (width < 0 || height < 0)
  throw new IllegalArgumentException();
 }
```

Parameterized JUnit Tests

- To add scenarios, we need only add more parameters to:

```
@CsvSource( {
"0, 0, 0, 'getArea fails for 0x0'",
"2, 4, 8, 'getArea fails for 2x4'",
"4, 8, 32, 'getArea fails for 4x8'"
})
```

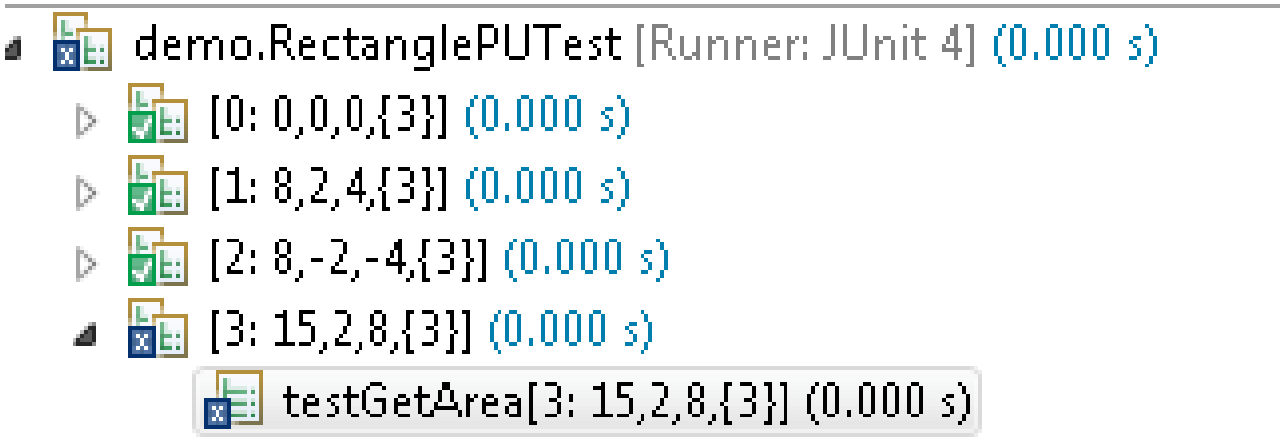


Add parameters here

Parameterized JUnit Tests (In JUnit4)

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.*;
import java.util.*;
@RunWith(Parameterized.class)
public class RectanglePUTest {
@Parameters(name="{index}: {0},{1},{2}")
public static Collection<Object[]> parameters() {
   return Arrays.asList(new Object[][] {
      {0,0,0}, {8,2,4}, {8,-2,-4}});
}
@Parameter // default value=0
public double expectedArea;
@Parameter(value=1)
public double width;
@Parameter(value=2)
public double height;
@Test
public void testGetArea() {
  Rectangle r1 = new Rectangle(width, height);
  assertEquals("getArea fails", expectedArea,
    r1.getArea(), 0.0000001);
 }
}
```

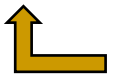
Parameterized JUnit Tests (in JUnit 4)



Parameterized JUnit Tests (In JUnit 4)

- To add scenarios, we need only add more parameters to:

```
@Parameters(name="{index}: {0},{1},{2}")
public static Collection<Object[]> parameters() {
   return Arrays.asList(new Object[][] {
      {0,0,0}, {8,2,4}, {8,-2,-4}});
}
```



Add parameters here

Parameterized JUnit Tests

- Often include both corner and normal scenarios/cases in the parameters
- Useful to check error handling and functional correctness

JUnit Theories

- So far, all test methods do not take any arguments.
- Can JUnit tests become more powerful if test methods can take arguments?

What is JUnit Theories

- A theory is a statement of test intent, which holds for a large range of input values.
  - Example - ∀e: stack.pop(push(e)) = e
- It allows more flexible and expressive assertions.

What is JUnit Theories

- Captures an intended behavior in possibly infinite potential scenarios
  - Note that an assertion in a traditional JUnit test captures an intended behavior in only one scenario.

```
@Test
public void testGetArea() {
  Rectangle r1 = new Rectangle(4,4);
  double expected = 16;
  double actual = r1.getArea();
  assertEquals(expected, actual,
    tolerance,"getArea fails");
}
```

JUnit Theories

- Suppose we know a theory:
  - Given a circle C and a rectangle R, C has a larger perimeter than R implies C has a larger area than R.
- Can we use the theory to help validate the implementation of constructors, getPerimeter() and getArea() in Circle and Rectangle?

JUnit Theories

- Adopt a contract model
  - **Assume, Act, Assert**
  - Assumptions (Preconditions) limits values appropriately
  - Action performs the required activities
  - Assertions (Postconditions) check result

JUnit Theories

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

@RunWith(Theories.class)
public class RectangleCircleTheoriesTest {

 // One set of datapoints for each datatype used
 @DataPoints
 public static double[] samples={-1,0,1,2,100,200};
```

```
@Theory
 public void testTheory(double r, double w,
 double h) {
 // state assumptions
 assumeTrue(r>0);
 assumeTrue(w>0);
 assumeTrue(h>0);
 assumeTrue(Circle.PI*r > (w+h));
 // perform actions
 double x = new Circle(r).getArea();
 double y = new Rectangle(w,h).getArea();
 // assert the theory
 assertTrue(x > y);
 }
}
```

Many tests instantiated

One test covers many scenarios

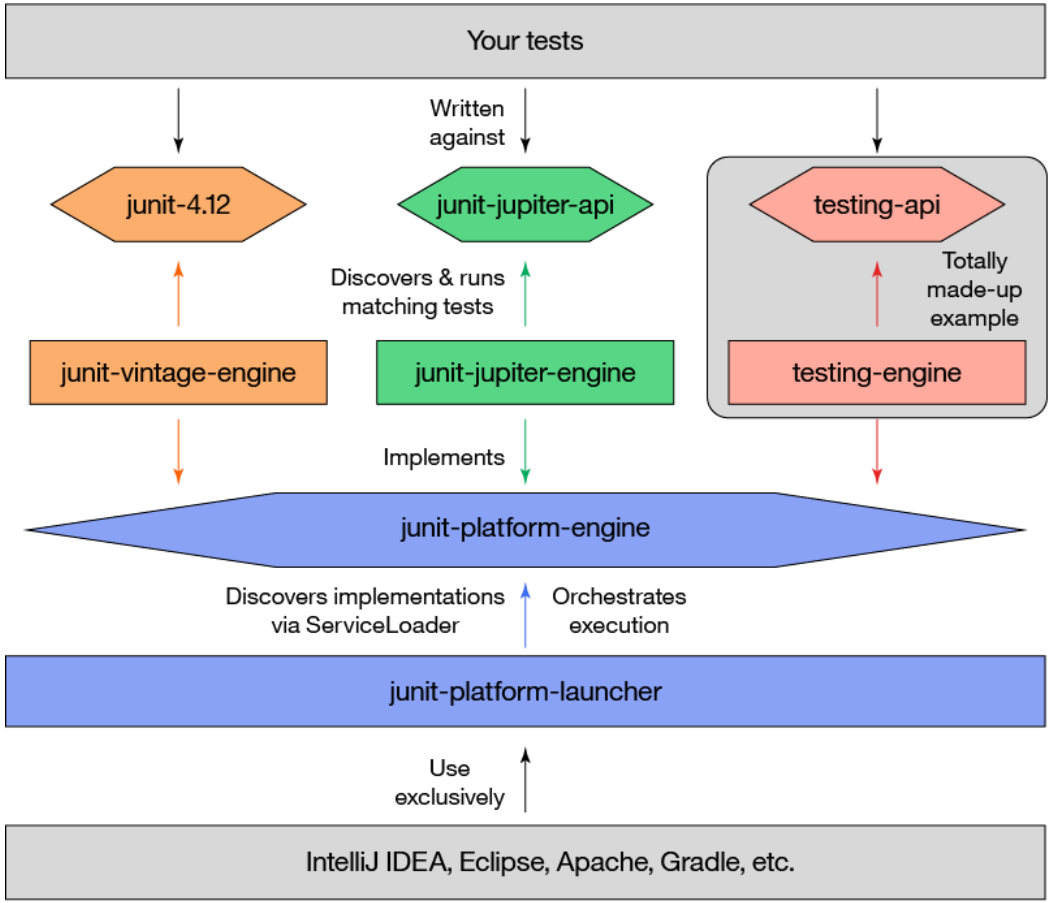
Instantiated test: r=1.0, w=1.0, h=1.0
Instantiated test: r=1.0, w=1.0, h=2.0
Instantiated test: r=1.0, w=2.0, h=1.0
Instantiated test: r=2.0, w=1.0, h=1.0
Instantiated test: r=2.0, w=1.0, h=2.0
Instantiated test: r=2.0, w=2.0, h=1.0
Instantiated test: r=2.0, w=2.0, h=2.0
Instantiated test: r=100.0, w=1.0, h=1.0
Instantiated test: r=100.0, w=1.0, h=2.0
Instantiated test: r=100.0, w=1.0, h=100.0
Instantiated test: r=100.0, w=1.0, h=200.0
Instantiated test: r=100.0, w=2.0, h=1.0
Instantiated test: r=100.0, w=2.0, h=2.0
Instantiated test: r=100.0, w=2.0, h=100.0
Instantiated test: r=100.0, w=2.0, h=200.0
Instantiated test: r=100.0, w=100.0, h=1.0
Instantiated test: r=100.0, w=100.0, h=2.0
Instantiated test: r=100.0, w=100.0, h=100.0
Instantiated test: r=100.0, w=100.0, h=200.0
Instantiated test: r=100.0, w=200.0, h=1.0
Instantiated test: r=100.0, w=200.0, h=2.0
Instantiated test: r=100.0, w=200.0, h=100.0
Instantiated test: r=200.0, w=1.0, h=1.0
Instantiated test: r=200.0, w=1.0, h=2.0
Instantiated test: r=200.0, w=1.0, h=100.0
Instantiated test: r=200.0, w=1.0, h=200.0
Instantiated test: r=200.0, w=2.0, h=1.0
Instantiated test: r=200.0, w=2.0, h=2.0
Instantiated test: r=200.0, w=2.0, h=100.0
Instantiated test: r=200.0, w=2.0, h=200.0
Instantiated test: r=200.0, w=100.0, h=1.0
Instantiated test: r=200.0, w=100.0, h=2.0
Instantiated test: r=200.0, w=100.0, h=100.0
Instantiated test: r=200.0, w=100.0, h=200.0
Instantiated test: r=200.0, w=200.0, h=1.0
Instantiated test: r=200.0, w=200.0, h=2.0
Instantiated test: r=200.0, w=200.0, h=100.0
Instantiated test: r=200.0, w=200.0, h=200.0

JUnit Design Objectives

- A simple framework that encourages developers to write unit tests.
- Minimalist framework – essential features, easier to learn, more likely to be used, flexible
- Test Cases & Test Results are objects
- Patterns – high “density” of patterns around key abstractions : mature framework

JUnit 5 Framework

- Designed to be backward compatible with JUnit 4



Resources

- JUnit: www.junit.org
- JUnit User Guide: https://junit.org/junit5/docs/current/user-guide/
- JUnit API documentation: https://junit.org/junit5/docs/current/api/
- Testing Frameworks : http://c2.com/cgi/wiki?TestingFramework
- cppUnit: http://cppunit.sourceforge.net/doc/1.8.0/index.html
- Unit Testing in Java – How Tests Drive the Code, Johannes Link
- Test-Driven Development by Example, Kent Beck
- Pragmatic Unit Testing in Java with JUnit, Andy Hunt & Dave Thomas
- "Learning to Love Unit Testing", Thomas & Hunt: www.pragmaticprogrammer.com/articles/stqe-01-2002.pdf

Advice

Whenever you are tempted to type something into a print statement,

write it as a test instead.

by Martin Fowler

Advice (Don’ts)

- Assume the order in which tests would run.
  - Strong dependencies between tests.
- Depend on externals (db, network connection, etc)…
- Depend on system date and random.
- Hardcode file paths or time delays.

Advice (Dos)

- Test everything that could possibly break.
  - Test exceptions, boundary conditions, assertions, …
- Write strong assertions in tests.
- Run tests as often as possible.
  - Run tests once you have made changes or every night.
- Write tests before implementation.

JUnit 5 vs JUnit 4

JUnit 5

```
import org.junit.Jupiter.api.Test;
public class JUnit5Test {
  @Test
  public void aJUnit5Test() {
    assertTrue(5 > 6, “message”);
  }
}
```

JUnit 4

```
import org.junit.Test;
public class JUnit4Test {
  @Test
  public void aJUnit4Test() {
    assertTrue(“message”, 5 > 6);
  }
}
```

Other Major New JUnit 5 Annotations

- @TestFactory: the method is a test factory for dynamic tests
- @DisplayName: declares a custom display name
- @BeforeEach, @AfterEach: replaces @Before and @After
- @BeforeAll, @AfterAll: replaces @BeforeClass and @AfterClass
- @Nested: declares a nested, non-static test class
- @Disabled: disables a test class or test method
- @ExtendWith: registers custom extensions

Selected New Features

- assertThrows() and expectThrows
- assertAll()
- assumingThat()
- Message is now the last parameter
- Accepts lambda syntax

Ready for Practice?

Alibaba Group | 社招官网 首 页 社会招聘 校园招聘 了解阿里 个人中心

蚂蚁集团-软件测试开发专家/测试算法高级工程师 新

发布时间：	2021-02-01	工作地点：	杭州,上海,北京	工作年限：	三年以上
所属部门：	蚂蚁集团	学 历：	本科	招聘人数：	若干

团队介绍：

团队主要对接蚂蚁集团核心基础安全的技术团队，深耕密码技术、可信计算、云原生安全技术等领域，通过建设下一代的密码基础设施、可信的身份体系、下一代的访问控制体系，构建新一代的零信任安全架构，打造世界领先的金融级可信安全平台。在企业内部构建联合作战网络，打造新一代企业纵深防御体系，助力金融级安全基础设施建设，保护全球数亿级用户的数字资产。

我们正在寻找富有激情和经验的优秀安全测试人员加入，以迎接包括移动、云、IoT、数据安全和AI安全在内的众多场景的安全挑战。如果您对零信任、应用密码学、访问控制、容器安全、安全计算等领域质量工作感兴趣，欢迎加入我们，共同打造安全可信的基础设施!

岗位描述：

面向蚂蚁集团金融级的高可靠性需求，负责数亿行软件代码的测试保障。研发软件测试算法和测试策略，并将其作为工具实现，接入现有的持续集成平台，既服务于底层的基础设施，也服务于公司的各业务线。

岗位要求：

1、软件工程或相关领域学士学位（有硕士、博士学位者优先）；
2、编程能力强，动手能力强，学习能力强。爱技术、好钻研、解决问题的能力强；
3、有自动化测试开发的工作经验；
4、有软件测试领域论文者优先；
5、有机器学习、数据挖掘经验者优先；
6、了解 metamorphic testing、combinatorial testing、adaptive random testing、mutation analysis、test case prioritization、regression testing 等测试技术者优先。

Enterprises are paying increasing attention to test automation

← A recent job advertisement from the Ant Group for Software Testing Experts

https://job.alibaba.com/zhaopin/position_detail.htm?trace=qrcode_share&positionCode=GP702541