UNIT TESTING BEST PRACTICES

Good unit tests: FIRST

- Fast
- Isolated
- Repeatable
- Self-validating
- Timely

Fast

- It should be comfortable to run all unit tests often
- Isolate slow tests from fast tests
  - Separate unit and integration tests

Isolated

- Only two possible results: PASS or FAIL
- No partially successful tests.
  - If a test can break for more than one reason, consider splitting it into separate tests
- Isolation of tests:
  - Different execution order must yield same results.
  - Test B should not depend on outcome of Test A

Repeatable

- A test should produce the same results each time you run it.
- Watch out for
  - Dates, times
  - Random numbers
  - Data from a datastore
- Use mock objects to give consistent data

Self-validating

- Your tests should be able to run anywhere at any time
- They should not depend on
  - Manual interaction
  - External setup

Timely

- Do not defer writing unit tests
  - For every method you write, write the corresponding unit tests at the same time
- Use test rules in your project
  - Review process
  - Test coverage tools

Unit test best practices

- Write tests for every found bug
- Fix failing tests immediately
- Make unit tests simple to run
  - Test suites can be run by a single command or a one button click.
- An incomplete set of unit tests is better than no unit tests at all.
- Don’t repeat production logic
- Reuse test code (setup, manipulate, assert)
- Don’t run a test from another test

Single Responsibility

- One test should be responsible for one scenario only.

- Test behavior, not methods:
  - One method, multiple behaviors → Multiple tests
  - One behavior, multiple methods → One test

Single Responsibility

```
@Test
public void testMethod() {
    assertTrue(behaviour1);
    assertTrue(behaviour2);
    assertTrue(behaviour3);
}
```

```
@Test
public void testMethodCheckBehaviour1() {
    assertTrue(behaviour1);
}

@Test
public void testMethodCheckBehaviour2() {
 assertTrue(behaviour2);
}

@Test
public void testMethodCheckBehaviour3() {
  assertTrue(behaviour3);
}
```

Self Descriptive

- Unit test must be easy to read and understand
  - Variable Names
  - Method Names
  - Class Names
  - No conditional logic
  - No loops

Self descriptive

- Name tests to represent PASS conditions:
  - canMakeReservation()
  - totalBillEqualsSumOfMenuItemPrices()
  - math_Divide_Throws_DivideByZeroException_When_Dividing_By_Zero()

No conditional logic

- Test should have no uncertainty:
  - All inputs should be known
  - Method behavior should be predictable
  - Expected output should be strictly defined
  - Split in to two tests rather than using “If” or “Case”

- Tests should not contain conditional logic.
  - If test logic has to be repeated, it probably means the test is too complicated.

No conditional logic

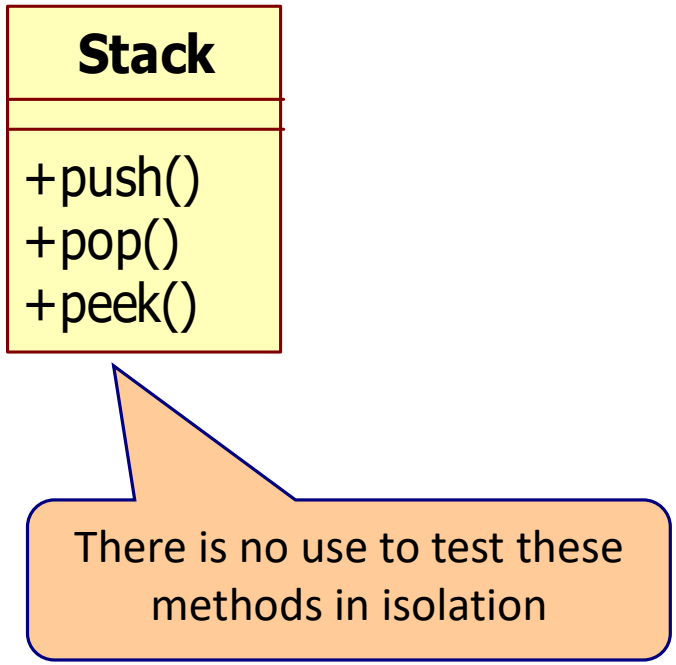
```
@Test
public void testMethod() {
  if (before)
    assertTrue(behaviour1);
  else if (after)
    assertTrue(behaviour2);
  else
    assertTrue(behaviour3);
}
```

```
@Test
public void testBefore() {
  boolean before = true;
  assertTrue(behaviour1);
}

@Test
public void testAfter() {
  boolean after= true;
  assertTrue(behaviour2);
}

@Test
public void testNow() {
   boolean before = false;
   boolean after= false;
   assertTrue(behaviour3);
}
```

Test only the public interface

- Every method has a side effect
  - Test this side effect
  - Test behavior, not methods
- What if this side effect is not visible (private attributes and methods)?
  - Do not sacrifice good design just for testing
  - Test behavior, not state

Test behavior, not methods/state



- Unit tests:
  - Pop of an empty stack should return null
  - Peek of an empty stack should return null
  - Push first x on the stack, then a peek should return x
  - Push first x on the stack, then a pop should remove x from the stack
  - Push first x, then y. A pop should return y and another pop should return x.

Summary

- Fast
- Isolated
- Repeatable
- Self-validating
- Timely
- Single responsibility
- No conditional logic
- Test behavior, not methods
  - Test the public interface

Treat test code as production code

Keep your tests

- Simple
- Short
- Understandable
- Loosely coupled

HAMCREST MATCHERS

Traditional asserts

- Parameter order is counter-intuitive
- Assert statements don't read well

assertEquals(***expected***, ***actual***)

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

@Test
public void AssertEqualToRed(){
    String color = "red";
    assertEquals("red", color);
}
```

assertThat with hamcrest matchers

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Before;
import org.junit.Test;

public class CalculatorHamcrestTest{
Calculator calculator=null;

    @Before
    public void createAcalculator(){
      calculator = new Calculator();
    }

    @Test
    public void add(){
        assertThat( calculator.add( 10, 50), equalTo (60.0));
    }

    @Test
    public void divide(){
        assertThat(calculator.divide( 10, 2 ), equalTo (5.0));
    }
}
```

Static import of matchers

matcher

assertThat

actual

expected

assert vs assertThat

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

@Test
public void AssertEqualToRed(){
    String color = "red";
    assertEquals("red", color);
}
```

assert

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

@Test
public void hamcrestAssertEqualToRed(){
    String color = "red";
    assertThat(color, equalTo("red"));
}
```

assertThat

assertThat equality tests

```
String color = "red";
assertThat(color, is("red"));
```

assertThat … is

```
String color = "red";
assertThat(color, equalTo("red"));
```

assertThat … equalTo

```
String color = "red";
assertThat(color, not("blue"));
```

assertThat … not

```
String color = "red";
assertThat(color, isOneOf("blue","red"));
```

assertThat … isOneOf

```
List myList = new ArrayList();
assertThat(myList, is(Collection.class));
```

assertThat … is a class

assertThat testing for null values

```
String color = "red";
assertThat(color, is(notNullValue()));
assertNotNull(color);
```

notNullValue

```
String color = null;
assertThat(color, is(nullValue()));
assertNull(color);
```

nullValue

assertThat testing with collections

```
List<String> colors = new ArrayList<String>();
colors.add("red");
colors.add("green");
colors.add("blue");
assertThat(colors, hasItem("blue"));
```

hasItem

```
assertThat(colors, hasItems("red","blue"));
```

hasItems

```
String[] colors = new String[] {"red","green","blue"};
assertThat(colors, hasItemInArray("blue"));
```

hasItemInArray

```
assertThat("red", isIn(colors));
```

isIn

```
List<Integer> ages = new ArrayList<Integer>();
ages.add(20);
ages.add(30);
ages.add(40);
assertThat(ages, not(hasItem(lessThan(18))));
```

Combined matchers

Hamcrest matchers

- Core
  - anything - always matches, useful if you don't care what the object under test is
  - describedAs - decorator to adding custom failure description
  - is - decorator to improve readability
- Logical
  - allOf - matches if all matchers match, short circuits (like Java &&)
  - anyOf - matches if any matchers match, short circuits (like Java ||)
  - not - matches if the wrapped matcher doesn't match and vice versa
- Object
  - equalTo - test object equality using Object.equals
  - hasToString - test Object.toString
  - instanceOf, isCompatibleType - test type
  - notNullValue, nullValue - test for null
  - sameInstance - test object identity
- Beans
  - hasProperty - test JavaBeans properties
- Collections
  - array - test an array's elements against an array of matchers
  - hasEntry, hasKey, hasValue - test a map contains an entry, key or value
  - hasItem, hasItems - test a collection contains elements
  - hasItemInArray - test an array contains an element
- Number
  - closeTo - test floating point values are close to a given value
  - greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo - test ordering
- Text
  - equalToIgnoringCase - test string equality ignoring case
  - equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace
  - containsString, endsWith, startsWith - test string matching

Hamcrest packages

Matcher Library	
org.hamcrest.beans	Matchers of Java Bean properties and their values.
org.hamcrest.collection	Matchers of arrays and collections.
org.hamcrest.core	Fundamental matchers of objects and values, and composite matchers.
org.hamcrest.internal	
org.hamcrest.number	Matchers that perform numeric comparisons.
org.hamcrest.object	Matchers that inspect objects and classes.
org.hamcrest.text	Matchers that perform text comparisons.
org.hamcrest.xml	Matchers of XML documents.

http://hamcrest.org/JavaHamcrest/javadoc/1.3/