ITSS SOFTWARE DEVELOPMENT

8.2 UNIT TEST - JUNIT

Content

1. **Unit testing frameworks**
2. JUnit
3. Test Driven Development

Unit testing

- Unit testing **concentrate** on **each unit** of the software as **implemented in source code**
  - **Focus** on each **component individual**, ensuring that it functions properly as a unit
  - Unit is the **smallest part of a software** system which is testable: code files, classes, methods, etc.
  - Before testing an integrated large module or whole system

Unit testing (2)

- Divide-and-conquer approach
  - Split system into units
  - Debug unit individually
  - Narrow down places where bugs can be
  - Don’t want to chase down bugs in other units

Unit testing (3)

- Good unit test
  - Automated and repeatable
  - Simple to implement
  - Maintainable
  - Available, anyone should be able to run it
  - Runs at the push of a button (or automatically)
  - Runs quickly
  - Runs in memory

Anatomy of a unit test

```
@Test
public void WhenXHappens_GivenY_ResultShouldBeZ()
{
  //arrange
  Thing newThing = new Thing();
  //act
  newThing.doSomething();
  //assert
  Assert.Equals("Actual",newThing.toString());
}
```

Indicates a test method

Descriptive method name

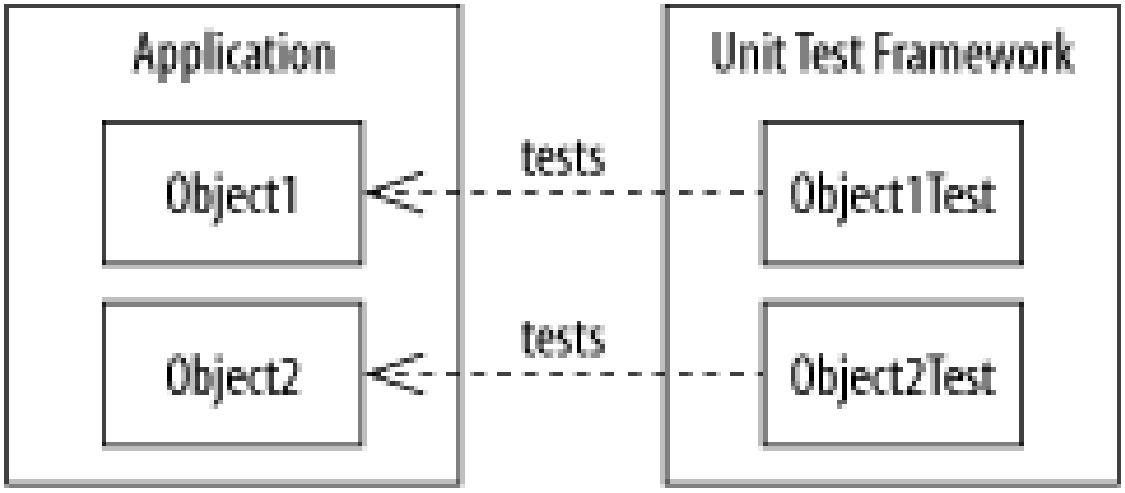
Set up the variables & objects for the test

Perform the test action(s)

Assert that the expected result occurred.

Unit testing Frameworks

- Software tools to support writing and running unit tests, including a foundation on which to build tests and the functionality to execute the tests and report their results.
- The relationship of unit tests to production code



  - The tests can be run separately from the application, so the objects can be tested in isolation.

Unit Test Frameworks by Paul Hamill

Unit testing Frameworks (2)

- JUnit is a unit-testing framework for Java
  - Developed by the same people who pioneered TDD
  - Uses source code annotations to decorate special methods to be run by the test harness
  - Integrated with Java IDEs like Eclipse and JCreator
- NUnit is a unit testing framework based on.NET platform
  - It is a free tool allows to write test scripts manually but not automatically
  - NUnit works in the same way as JUnit works for Java
  - Supports data-driven tests that can run in parallel
  - Uses Console Runner to load and execute tests

Unit testing Frameworks (3)

- JMockit is an open-source tool for Unit Testing with the collection of tools and API
  - Developers can use these tools and API to write test using TestNG or JUnit
  - JMockit is considered as an alternative to the conventional use of the mock object
- Emma is an open-source toolkit that measures Java Code Coverage
  - It enables the code coverage for each and every developer in the team rapidly
  - Emma supports class, line, method and basic block coverage and report types like text, HTML, XML etc.

Unit testing Frameworks (4)

- HtmlUnit is an open-source Java library which contains GUI-less browser for Java programs
  - This tool supports JavaScript and provides GUI features like forms, links, tables, etc.
  - It is a Java unit testing framework for testing web applications that are used within frameworks like JUnit, TestNG
  - HtmlUnit uses the JavaScript engine named as Mozilla Rhino
  - Supports protocols like HTTP, HTTPS along with a cookie, submit methods like GET, POST, and proxy server
- SimpleTest is an open-source unit testing framework dedicated to PHP Programming Language
  - This framework supports SSL, forms, proxies and basic authentication
  - The test case classes in SimpleTest are being extended from base test classes along with methods and codes
  - SimpleTest includes autorun.php.file to transform test cases into executable test scripts

SimpleTest
Unit Testing for PHP

Content

1. Unit testing frameworks
2. **JUnit**
3. Test Driven Development

JUnit

- JUnit is a framework for test-drivent development
  - Written by Erich Gamma (Design Patterns) and Kent Beck (eXtreme Programming)
- JUnit uses Java's reflection capabilities (Java programs can examine their own code) and (as of version 4) **annotations**
- JUnit allows us to:
  - Define and execute tests and test suites
  - Use test as an effective means of specification
  - Write code and use the tests to support refactoring
  - Integrate revised code into a build
- JUnit is available on several IDEs, e.g. BlueJ, JBuilder, Eclipse, DrJava...

https://junit.org/junit5/

JUnit history

- Timeline

- **JUnit 4**
  - Significant (and not compatible) change from prior versions.
  - Requires JDK 5 or later
- **JUnit 3** is different.
  - Can be used with earlier versions of Java
  - Still in use

https://www.youtube.com/watch?v=TK0H8SCSiOA

JUnit's terminologies

- A test runner is software that runs tests and reports results
  - Many implementations: standalone GUI, command line, integrated into IDE
- A test suite is a a collection of test cases
  - Usually these test cases share similar pre-requisites and configuration
  - Usually can be run together in sequence
  - Different test suites for different purposes
- A test case tests the response of a single method to a particular set of inputs
- A unit test is a test of the smallest element of code you can sensibly test, usually a single class

JUnit's terminologies (2)

- A test fixture is the environment in which a test is run. A new fixture is setup before each test case is executed, and torn down afterwards.
  - Example: if you are testing a database client, the fixture might place the database server in a standard initial state, ready for the client to connect.
- Proper unit testing would involve mock objects - fake versions of the other classes with which the class under test interacts.
  - *JUnit does not help with this*. It is worth knowing about, but not always necessary.
- Test oracle (or just oracle) is a mechanism for determining whether a test has passed or failed

JUnit annotations

- Common annotations

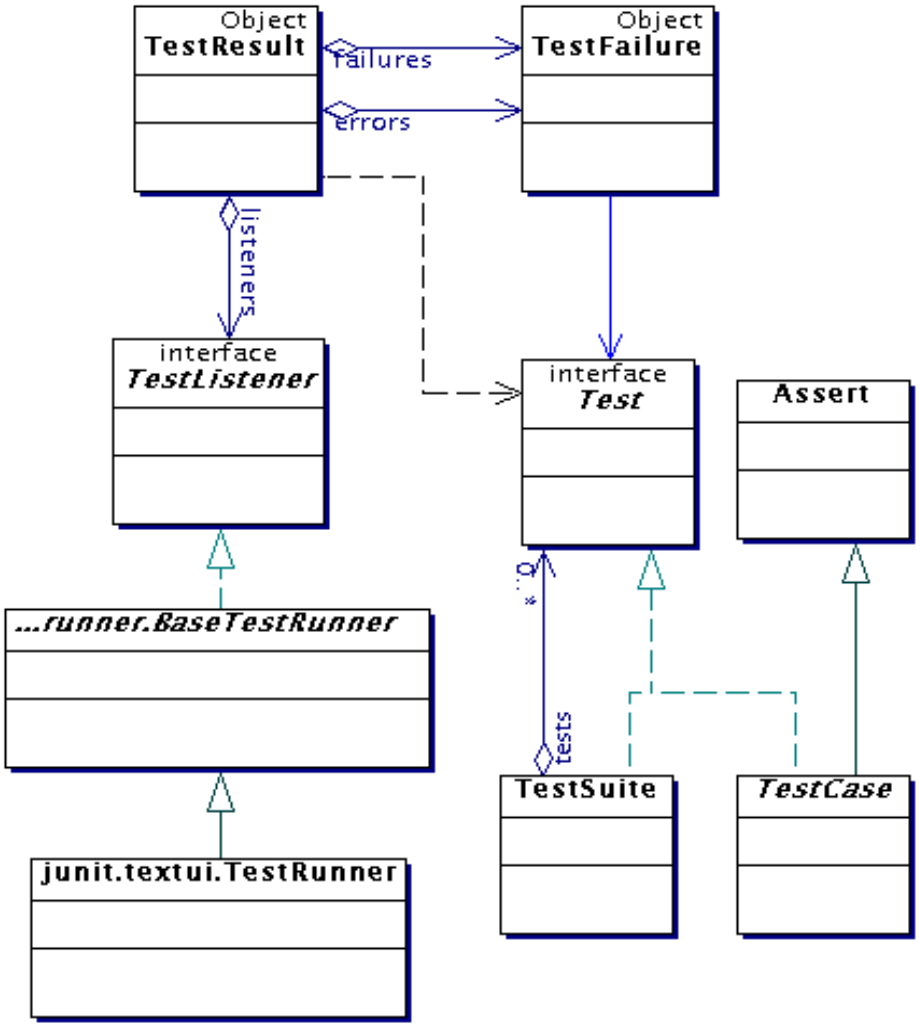
Annotation	Description
@Test	Identify test methods
@Test (timeout=100)	Fail if the test takes more than 100ms
@Before	Execute before each test method
@After	Execute after each test method
@BeforeClass	Execute before each test class
@AfterClass	Execute after each test class
@Ignore	Ignore the test method

Why create a test suite?

- Obviously you have to test your code—right?
  - You can do *ad hoc* testing (running whatever tests occur to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of a test suite
  - It's a lot of extra programming
    - True, but use of a good test framework can help quite a bit
  - You don't have time to do all that extra work
    - *False!* Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of a test suite
  - Reduces total number of bugs in delivered code
  - Makes code much more maintainable and refactorable

Architectural overview

- JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests
- **TestRunner** runs tests and reports **TestResult**s
- You test your class by extending abstract class ***TestCase***
- To write test cases, you need to know and understand the **Assert** class



A JUnit test class

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

public class 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

The structure of a test method

- A test method doesn’t return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an **AssertionFailedError**
- The JUnit framework catches the error and deals with it; you don’t have to do anything

Organize The Tests

- Create test cases in the same package as the code under test
- For each Java package in your application, define a **TestSuite** class that contains all the tests for validating the code in the package
- Define similar **TestSuite** classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application
- Make sure your build process include the compilation of all tests

A JUnit test suite

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)

@Suite.SuiteClasses({
    Test1.class,
    Test2.class
})

public class JunitTestSuite {
}
```

JUnit will invoke the class it references to run the tests in that class

class tests was running

- Create a java class file
- Attach **`@RunWith(Suite.class)`** Annotation with the class.
- Add reference to JUnit test classes using **`@Suite.SuiteClasses`** annotation.

A JUnit test runner

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result =
 JUnitCore.runClasses(JunitTestSuite.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

- Create a java class file to execute test case(s)

Test Case Verdicts

- A *verdict* is the result of executing a single test case

Pass

- The test case execution was completed
- The function being tested performed as expected

Fail

- The test case execution was completed
- The function being tested did *not* perform as expected

Error

- The test case execution was not completed, due to
  - an unexpected event, exceptions, or
  - improper set up of the test case, etc.

JUnit Assertions

- *Assertions* are Boolean expressions
  - An *assertion failure* exception is thrown if the assertion is false
- Can check for many conditions, such as
  - equality of objects and values
  - identity of references to objects
- Determine the test case verdict
  - **Pass:** all assertions are true
  - **Fail:** one or more assertions are false

Assert methods

During execution of a test case:

- If an assertion is **true**,
  - Execution continues
- If any assertion is **false**,
  - Execution of the test case stops
  - The test case fails
- If an *unexpected* exception is encountered,
  - The verdict of the test case is an error.
- If all assertions were true,
  - The test case *passes*.

Assert methods (2)

- Each assert method has parameters like these:

  message, *expected-value*, *actual-value*

  - floating point numbers get an additional argument, a tolerance
- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:
  - messages helps documents the tests
  - messages provide additional information when reading failure logs

Assert methods: Boolean Conditions

- Static methods defined in org.junit.Assert
- Assert an Boolean condition is true or false

  assertTrue(*condition*)

  assertFalse(*condition*)
- Optionally, include a failure message

  assertTrue(*message, condition*)

  assertFalse(*message, condition*)
- Examples

```
assertTrue(search(a, 3) == 1);
assertFalse("Failure: 2 is not in array.", search(a, 2) >= 0);
```

Assert methods: Null Objects

- Assert an object references is null or non-null

  assertNull(*object*)

  assertNotNull(*object*)

- With a failure message

  assertNull(*message*, *object*)

  assertNotNull(*message*, *object*)

- Examples

```
assertNotNull(”Should not be null.", new Object());
assertNull(”Should be null.", null);
```

Assert methods: Object Identity

- Assert two object references are identical

  assertSame(*expected, actual*)

  - True if: expected == actual

  assertNotSame(*expected, actual*)

  - True if: expected != actual

- The order does not affect the comparison,
  - But, affects the message when it fails
- With a failure message

  assertSame(*message*, *expected, actual*)

  assertNotSame(*message*, *expected, actual*)

Assert methods: Object Equality

- Assert two objects are equal:

  assertEquals(expected, actual)

  - True if: expected.equals(actual)
  - Relies on the equals() method
  - Up to the class under test to define a suitable equals() method.

- With a failure message

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

Examples

```
assertEquals("Should be equal.", "JUnit", "JUnit");

assertEquals("Should be equal.", "JUnit", "Java");
```

org.junit.ComparisonFailure:
Should be equal. expected:<J[Unit]> but was:<J[ava]>

Assert methods: Equality of Arrays

- Assert two arrays are equal:

  assertArrayEquals(expected, actual)

  - arrays must have same length
  - Recursively check for each valid index i,

    assertEquals(expected[i],actual[i])

    or

    assertArrayEquals(expected,actual)

- With a failure message

  assertArrayEquals(*message*, *expected, actual*)

Assert methods: Floating Point Values

- Beware of problems with comparisons: floating point arithmetic is not precise
- For comparing floating point values (double or float)
  - assertEquals requires an additional parameter **delta**.

assertEquals(*expected, actual, delta*)

assertEquals(*message*, *expected, actual, delta*)

- The assertion evaluates to true if

Math.abs(expected – actual) <= delta

- Example:

```
double d1 = 100.0, d2 = 99.99995;
assertEquals("Should be equal within delta.", d1, d2, 0.0001);
```

More stuff in test classes: test fixtures

- Typically include:
  - Common objects or resources that are available for use by any test case.
- Activities to manage these objects
  - Set-up: object and resource allocation
    - Tasks that must be done prior to each test case
    - Examples:
      - Create some objects to work with
      - Open a network connection
      - Open a file to read/write
  - Tear-down: object and resource de-allocation
    - Tasks to clean up after execution of each test case.
      - Ensures resources are released

More stuff in test classes: test fixtures (2)

- @Before annotation: set-up
  - code to run before each test case.
- @After annotation: Teardown
  - code to run after *each* test case.
  - will run regardless of the verdict, even if exceptions are thrown in the test case or an assertion fails.
- Multiple annotations are allowed
  - all methods annotated with @Before will be run before *each* test case
  - but no guarantee of execution order

Example: Using a File as a test fixture

```
public class OutputTest {
    private File output;

    @Before
    public void createOutputFile() {
       output = new File(...);
    }

    @After
    public void deleteOutputFile() {
        output.close();
        output.delete();

    }

    @Test
    public void test1WithFile() {
       // code for test case
       …
    }

    @Test
    public void test2WithFile() {
       // code for test case
       …
    }
}
```

Once-only set-up

- @BeforeClass annotation on a *static* method
  - one method only
- Run the method *once only* for the entire test class
  - *before* any of the tests, and
  - *before* any @Before method(s)
- Useful for starting servers, opening connections, etc.
  - No need to reset/restart for each test case
  - Shared, non-destructive

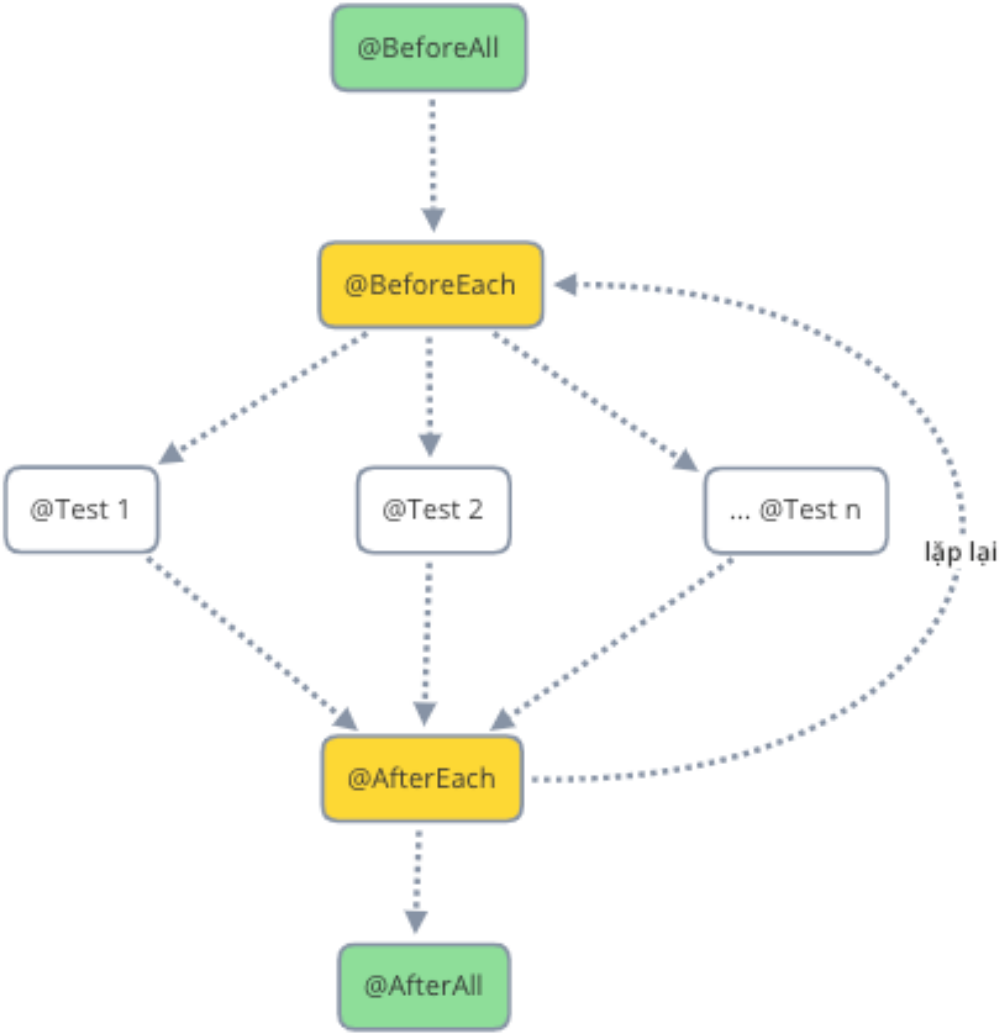
```
@BeforeClass
public static void anyName() {
    // class setup code here
}
```

Once-only tear-down

- **@AfterClass** annotation on a *static* method
  - one method only
- Run the method *once only* for the entire test class
  - *after* any of the tests
  - *after* any @After method(s)
- Useful for stopping servers, closing connections, etc.

```
@AfterClass
 public static void anyName() {
    // class clean up code here
 }
```

Các annotation trong JUnit



Timed tests

- Useful for simple performance test
  - Network communication
  - Complex computation
- The timeout parameter of @Test annotation
  - in milliseconds

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

- The test fails
  - if timeout occurs before the test method completes

Parameterized tests

- Repeat a test case multiple times with different data
- Define a parameterized test
  - Class annotation, defines a test runner

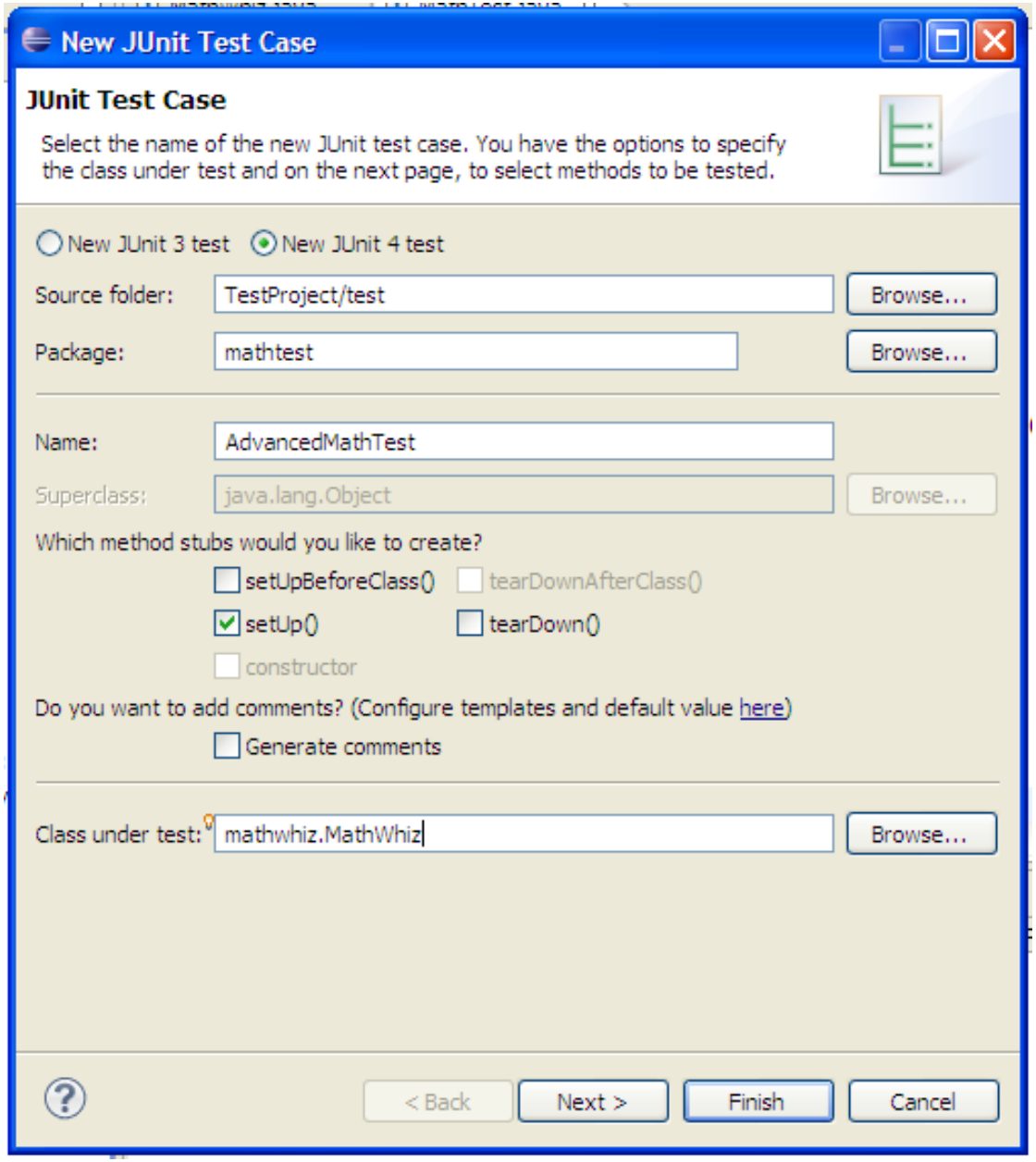
    @RunWith(Parameterized.class)
  - Define a constructor
    - Input and expected output values for one data point
  - Define a static method returns a *Collection* of data points
    - Annotation @Parameter [or @Parameters, depending]
    - Each data point:
      an array, whose elements match the constructor arguments

Running a parameterized test

- Use a parameterized test runner
- For each data point provided by the parameter method
  - Construct an instance of the class with the data point
  - Execute all test methods defined in the class

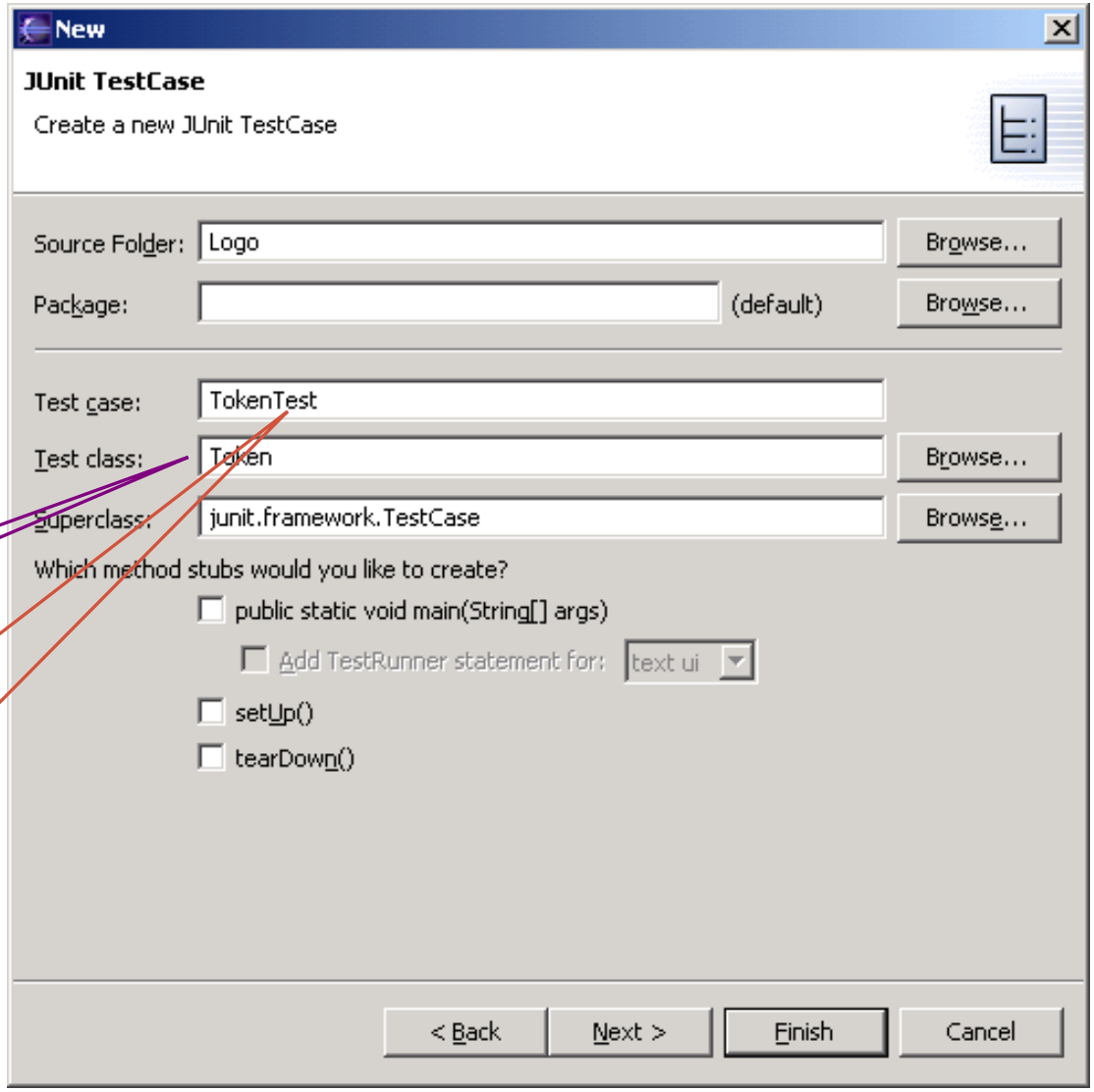
Creating JUnit Tests with Eclipse

- Using JUnit with Eclipse is easy
- Create new JUnit tests using File -> New -> JUnit
  - Specify JUnit 4
  - Select location, class you want to test, method stubs to create
  - Then select which methods you want to test, Eclipse will generate test stubs



JUnit in Eclipse

- To create a test class, select File→ New→ Other… → Java, JUnit, TestCase and enter the name of the *class* you will test



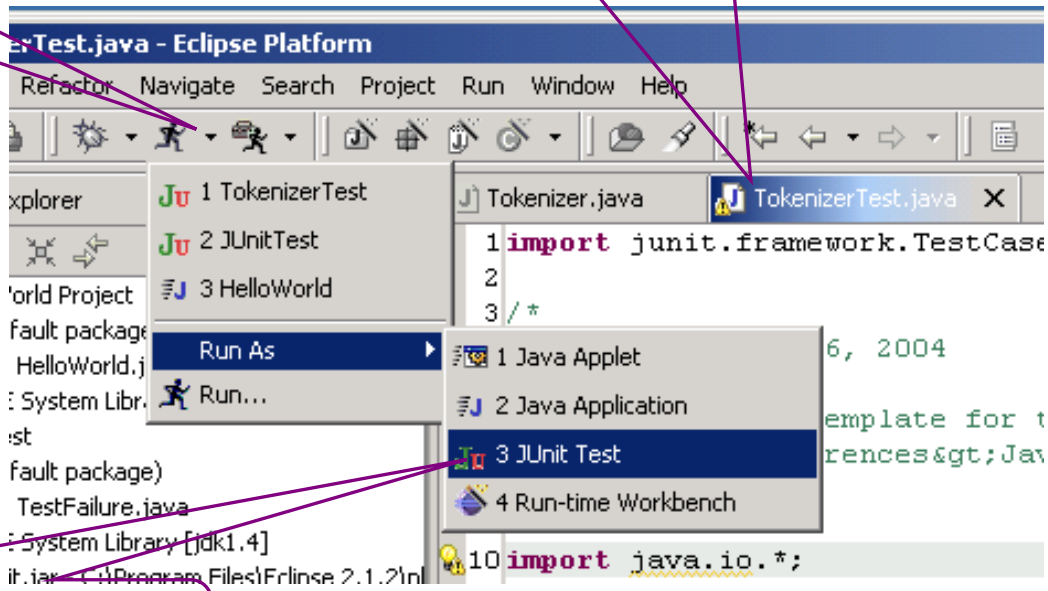
Fill this in

This will be filled in *automatically*

Running JUnit

Second, use this pulldown menu

First, select a *Test* class



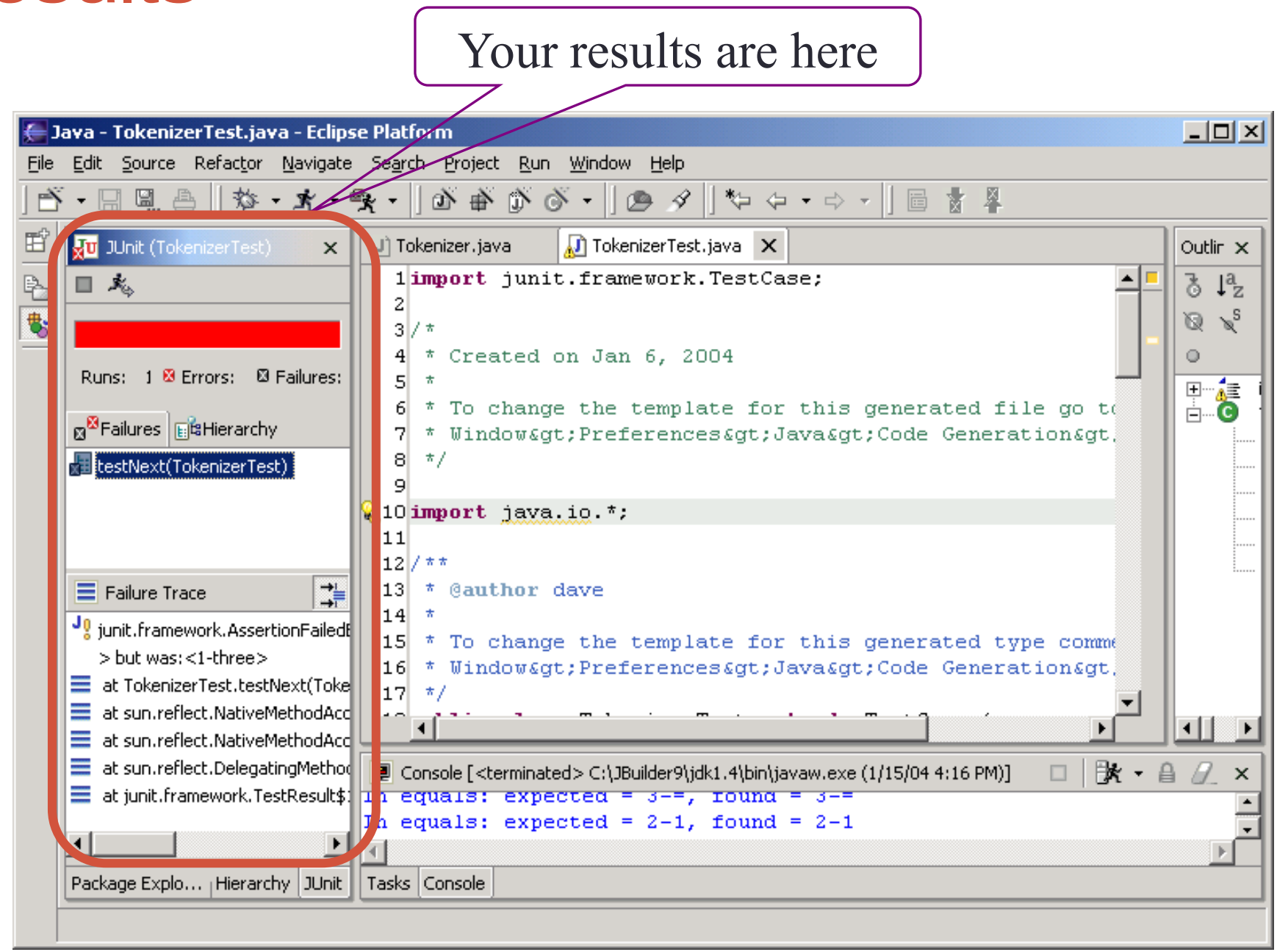
Third, Run As → JUnit Test

Your JUnit test classes should have the following import declarations:

import static org . junit . jupiter . api . Assertions .*;

import org . junit . jupiter . api . Test ;

Results


Your results are here
Java - TokenizerTest.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
JUnit (TokenizerTest)
Runs: 1 Errors: Failures:
Failures Hierarchy
testNext(TokenizerTest)
Failure Trace
junit.framework.AssertionFailedE
> but was:<1-three>
at TokenizerTest.testNext(Toke
at sun.reflect.NativeMethodAcc
at sun.reflect.NativeMethodAcc
at sun.reflect.DelegatingMetho
at junit.framework.TestResult$
Package Explo... Hierarchy JUnit
Tokenizer.java
TokenizerTest.java
1 import junit.framework.TestCase;
2
3 /*
4 * Created on Jan 6, 2004
5 *
6 * To change the template for this generated file go t
7 * Window>Preferences>Java>Code Generation>
8 */
9
10 import java.io.*;
11
12 /**
13 * @author dave
14 *
15 * To change the template for this generated type comme
16 * Window>Preferences>Java>Code Generation>
17 */
Outlir
Console [<terminated> C:\JBuilder9\jdk1.4\bin\javaw.exe (1/15/04 4:16 PM)]
In equals: expected = 3-=, found = 3-=
In equals: expected = 2-1, found = 2-1
Tasks Console

Example 1: Triangle class

```
public class Triangle {
    private int p; // Longest edge
    private int q;
    private int r;
    public Triangle(int s1, int s2, int s3) {
        if (s1>s2) {
            p = s1; q = s2;
        } else {
            p = s2; q = s1;
        }
        if (s3>p) {
            r = p; p = s3;
        } else {
            r = s3;
        }
    }
    public boolean isScalene() {
        return ((r>0) && (q>0) &&
            (p>0) &&
            (p<(q+r))&& ((q>r) || (r>q)));
    }
    public boolean isEquilateral() {
        return p == q && q == r;
    }
}
```

Example 1: a JUnit 4 test for Triangle

```
package st;

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

public class TestTriangle {

    private Triangle t;
    @Before public void setUp() throws Exception {
        t = new Triangle(3, 4, 5);
    }
    @Test public void scaleneOk() {
        assertTrue(t.isScalene());
    }
    /** Other JUnit test methods*/
}
```

Test driver

Test case

Test oracle

Example 2:

The class under test is **`UserDAO`**:

```
1  public class UserDAO {
2
3      public User save(User user) {
4          // code to persist the User object
5          return user;
6      }
7
8      public void delete(User user) {
9          // code to remove the User object
10     }
11 }
```

The class **`UserDAOTest`**:

```
1  import org.junit.Test;
2  import static org.junit.Assert.fail;
3  import static org.junit.Assert.assertNotNull;
4
5  public class UserDAOTest {
6
7      @Test
8      public void testSaveUser() {
9          UserDAO dao = new UserDAO();
10         User user = new User();
11         user = dao.save(user);
12
13         assertNotNull(user);
14     }
15
16     @Test
17     public void testDeleteUser() {
18         fail("Not yet implemented");
19     }
20 }
```

Example 2:

The class under test is **`ProductDAO`**:

```
1  public class ProductDAO {
2
3      public Product save(Product product) {
4          // code to persist the Product object
5          return product;
6      }
7
8      public void delete(Product product) {
9          // code to remove the Product object
10     }
11 }
```

The class **`ProductDAOTest`**:

```
1  import org.junit.Test;
2  import static org.junit.Assert.fail;
3  import static org.junit.Assert.assertNotNull;
4
5  public class ProductDAOTest {
6
7      @Test
8      public void testSaveProduct() {
9          ProductDAO dao = new ProductDAO();
10         Product product = new Product();
11         product = dao.save(product);
12
13         assertNotNull(product);
14     }
15
16     @Test
17     public void testUpdateProduct() {
18         fail("Not yet implemented");
19     }
20 }
```

Example 2:

The Test suite class:

```
1  import org.junit.runner.RunWith;
2  import org.junit.runners.Suite;
3  import org.junit.runners.Suite.SuiteClasses;
4
5  @RunWith(Suite.class)
6  @SuiteClasses({UserDAOTest.class, ProductDAOTest.class})
7  public class ProjectTestSuite {
8      // code relevant to test suite goes here
9  }
```

The test runner class:

```
1  import org.junit.runner.JUnitCore;
2  import org.junit.runner.Result;
3  import org.junit.runner.notification.Failure;
4
5  public class TestSuiteRunner {
6
7      public static void main(String[] args) {
8          Result result = JUnitCore.runClasses(ProjectTestSuite.class);
9
10         for (Failure failure : result.getFailures()) {
11             System.out.println(failure.toString());
12             failure.getException().printStackTrace();
13         }
14
15
16 System.out.println("Test successful? " + result.wasSuccessful());
17     }
18 }
```

Unit testing for other languages

- Unit testing tools differentiate between:
  - Errors (unanticipated problems caught by exceptions)
  - Failures (anticipated problems checked with assertions)
- Basic unit of testing:
  - *CPPUNIT_ASSERT(Bool)* examines an expression
- CPPUnit has variety of test classes (e.g. *TestFixture*)
  - Inherit from them and overload methods

Another example: sqrt

```
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)
```

What are some values or ranges of `x` that might be worth testing

- `x < 0` (exception thrown)
- `x ≥ 0` (returns normally)
- around `x = 0` (boundary condition)
- perfect squares (`sqrt(x)` an integer), non-perfect squares
- `x < sqrt(x)`, `x > sqrt(x)`
- Specific tests: say `x = {-1, 0, 0.5, 1, 4}`

Subdomains

- Many executions reflect the same behavior – for **`sqrt`**, for example, the expectation is that
  - all **`x < 0`** inputs will throw an exception
  - all **`x ≥ 0`** inputs will return normally with a correct answer
- By testing any element from each *subdomain*, the intention is for the single test to represent the other behaviors of the subdomain – *without testing them!*
- Of course, this isn't so easy – even in the simple example above, what about when **`x`** overflows?

Testing `RandomHello`

- "*Create your first Java class with a main method that will randomly choose, and then print to the console, one of five possible greetings that you define.*"

- We'll focus on the method **`getGreeting`**, which randomly returns one of the five greetings
- We'll focus on *black-box testing* – we will work with no knowledge of the implementation
- And we'll focus on unit testing using the JUnit framework

- Intermixing, with any luck, slides and a demo

Does it even run and return?

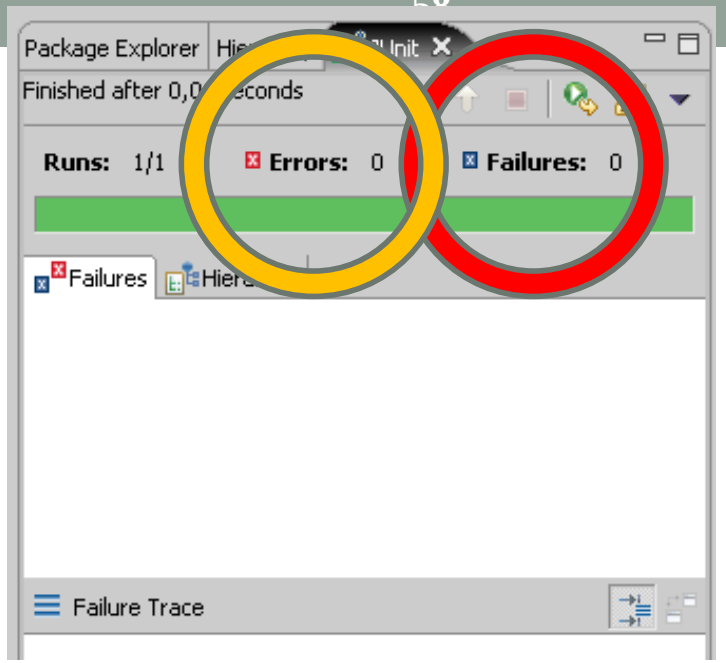
- If `getGreeting` doesn't run and return without throwing an exception, it cannot meet the specification

JUnit tag "this is a test"	`@Test`
name of test	`public void test_NoException(){`
Run `getGreeting`	`  RandomHello.getGreeting();`
JUnit "test passed" (doesn't execute if exception thrown)	`  assertTrue(true);` `}`

Tests should have descriptive (often very long) names

A unit test is a (stylized) program! When you're writing unit tests (and many other tests), you're programming!

Running JUnit tests



- There are many ways to run JUnit test method, test classes, and test suites
- Generally, select the method, class or suite and **`Run As >> JUnit Test`**
- A green bar says "all tests pass"
- A red bar says at least one test failed or was in error
- The failure trace shows which tests failed and why

- A failure is when the test doesn't pass – that is, the oracle it computes is incorrect
- An error is when something goes wrong with the program that the test didn't check for (e.g., a null pointer exception)

Does it return one of the greetings?

- If it doesn't return one of the defined greetings, it cannot satisfy the specification

```
@Test
public void testDoes_getGreeting_returnDefinedGreeting() {
  String rg = RandomHello.getGreeting();
  for (String s : RandomHello.greetings) {
    if (rg.equals(s)) {
      assertTrue(true);
      return;
    }
  }
  fail("Returned greeting not in greetings array");
}
```

A JUnit test class

Don't forget that Eclipse can help you get the right `import` statements – use Organize Imports (Ctrl-Shift-O)

```
import org.junit.*;
import static org.junit.Assert.*;
public class RandomHelloTest() {
  @Test
  public void test_ReturnDefinedGreeting() {
    …
  }
  @Test
  public void test_EveryGreetingReturned() {
    …
  }
  …
}
```

- All `@Test` methods run when the test class is run
- That is, a JUnit test class is a set of tests (methods) that share a (class) name

Does it return a random greeting?

```
@Test
public void testDoes_getGreetingNeverReturnSomeGreeting() {
  int greetingCount = RandomHello.greetings.length;
  int count[] = new int[greetingCount];
  for (int c = 0; c < greetingCount; c++)
    count[c] = 0;
  for (int i = 1; i < 100; i++) {
    String rs = RandomHello.getGreeting();
    for (int j = 0; j < greetingCount; j++)
      if (rs.equals(RandomHello.greetings[j]))
        count[j]++;
  }
  for (int j = 0; j < greetingCount; j++)
    if (count[j] == 0)
      fail(j+"th [0-4] greeting never returned");
  assertTrue(true);
}
```

Run it 100 times

If even one greeting is never returned, it's unlikely to be random (~1-0.8^{100})

What about a sleazy developer?

```
if (randomGenerator.nextInt(2) == 0) {
   return(greetings[0]);
} else
  return(greetings[randomGenerator.nextInt(5)]);
```

- Flip a coin and select **either** a random **or** a specific greeting
- The previous "is it random?" test will almost always pass given this implementation
- But it doesn't satisfy the specification, since it's not a random choice

Instead: Use simple statistics

```
@Test
public void test_UniformGreetingDistribution() {
 // …count frequencies of messages returned, as in
 // …previous test (test_EveryGreetingReturned)

  float chiSquared = 0f;
  float expected = 20f;
  for (int i = 0; i < greetingCount; i++)
    chiSquared = chiSquared +
              ((count[i]-expected)*
               (count[i]-expected))
                              /expected;
  if (chiSquared > 13.277) // df 4, pvalue .01
    fail("Too much variance");
}
```

A JUnit test suite

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

@RunWith(Suite.class)
@Suite.SuiteClasses({
  RandomHelloTest.class,
  SleazyRandomHelloTest.class
})
public class AllTests {
  // this class remains completely
  // empty, being used only as a
  // holder for the above
  // annotations
}
```

- Define one suite for each program (for now)
- The suite allows multiple test classes – each of which has its own set of `@Test` methods – to be defined and run together
- Add `tc.class` to the `@Suite.SuiteClasses` annotation if you add a new test class named `tc`
- So, a JUnit test suite is a set of test classes (which makes it a set of a set of test methods)

`ArrayIntList`: example tests

```
@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());
}
```

- High-level concept: test behaviors in combination
  - Maybe `add` works when called once, but not when call twice
  - Maybe `add` works by itself, but fails (or causes a failure) after calling `remove`

A few hints: data structures

- Need to pass lots of arrays? Use array literals

```
public void exampleMethod(int[] values) { ... }
...
exampleMethod(new int[] {1, 2, 3, 4});
exampleMethod(new int[] {5, 6, 7});
```

- Need a quick **`ArrayList`**?

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Need a quick set, queue, etc.? Many take a list

```
Set<Integer> list = new HashSet<Integer>(
                        Arrays.asList(7, 4, -2, 9));
```

A few general hints

- Test one thing at a time per test method
  - 10 small tests are much better than one large test
- Be stingy with **`assert`** statements
  - The first **`assert`** that fails stops the test – provides no information about whether a later assertion would have failed
- Be stingy with logic
  - Avoid **`try/catch`** – if it's supposed to throw an exception, use **`expected=`** ... if not, let JUnit catch it

Test case dangers

- Dependent test order
  - If running Test A before Test B gives different results from running Test B then Test A, then something is likely confusing and should be made explicit
- Mutable shared state
  - Tests A and B both use a shared object – if A breaks the object, what happens to B?
    - This is a form of dependent test order
    - We will explicitly talk about invariants over data representations and testing if the invariants are ever broken

More JUnit: exception testing

- Testing for exceptions

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
  ArrayIntList list = new ArrayIntList();
  list.get(4); // this should raise the exception
}              // and thus the test will pass
```

- Specify an expected exception in a test case
  - A parameter of @Test annotation
  - A particular class of exception is expected to occur
- The verdict
  - Pass: if the expected exception is thrown
  - Fail: if no exception, or an unexpected exception

More JUnit: exception testing

- Assertion methods
  - fail()
  - fail(*message*)
- Unconditional failure
  - i.e., it always fails if it is executed
- Used in where it should not be reached
  - e.g., after a statement, in which an exception should have been thrown.

More JUnit: exception testing

- Catch exceptions, and use fail() if not thrown

```
@Test
public void testCheckedSearch3() {
 try {
        checkedSearch(null, 1);
        fail("Exception should have occurred");
 } catch (IllegalArgumentException e) {
        assertEquals(e.getMessage(), "Null or empty array.");
 }
}
```

- Allows
  - inspecting specific messages/details of the exception
  - distinguishing different types of exceptions

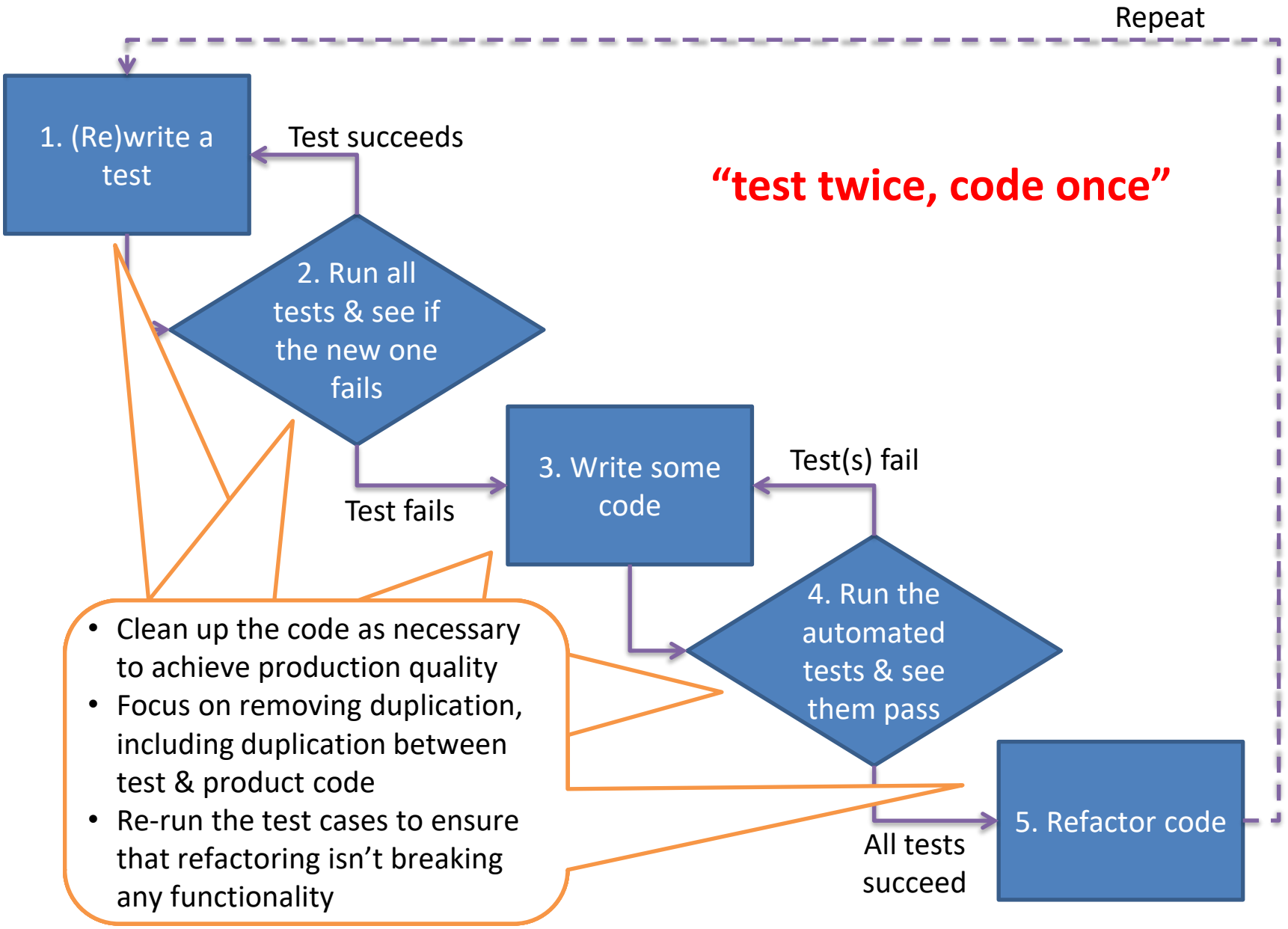
Content

1. Unit testing frameworks
2. JUnit
3. **Test Driven Development**

Test Driven Development (TDD)

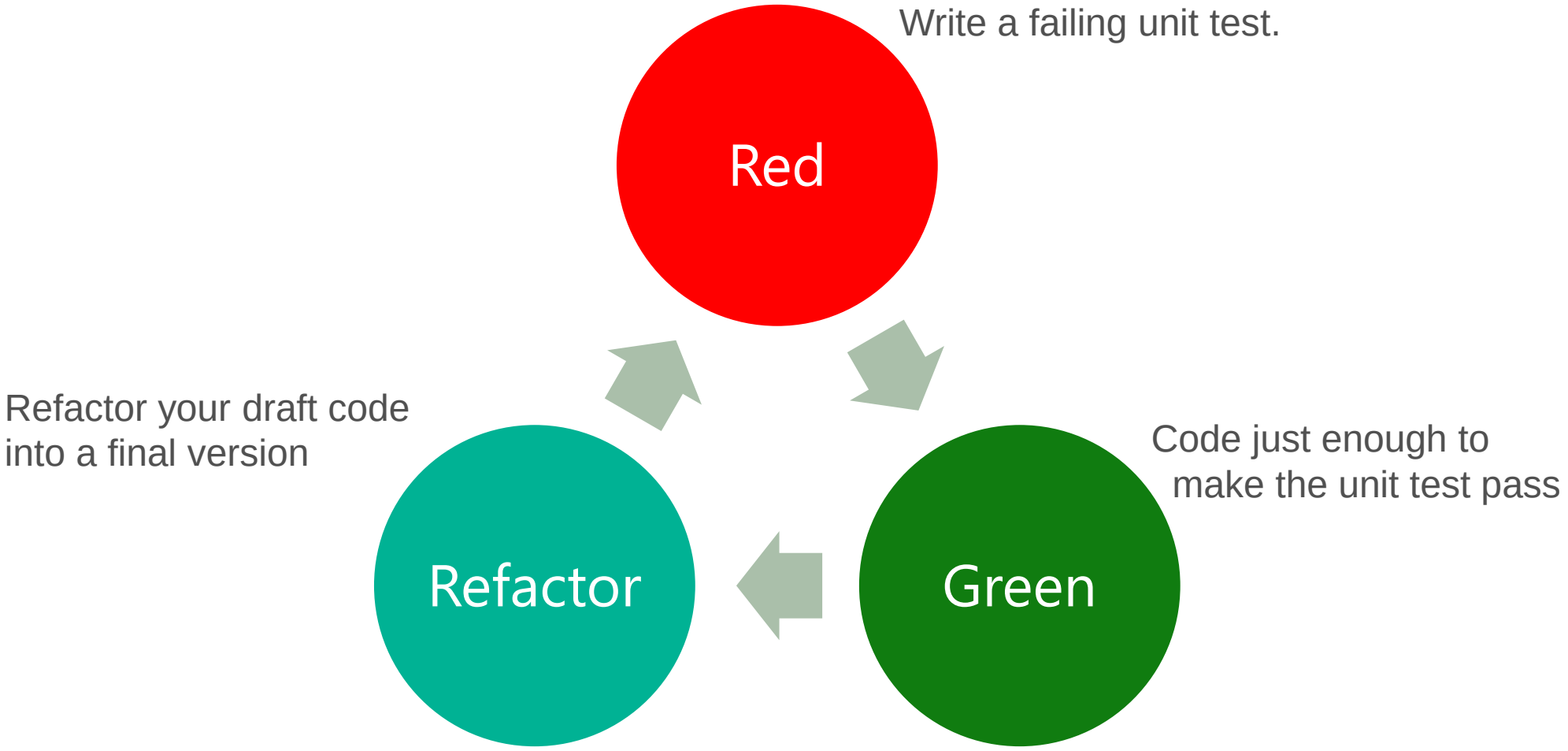
- A development technique that relies on the concept of writing the test cases ***before*** the product code
  - Validates that spec and requirements are well-understood
  - Test cases will initially fail
  - Developer writes code to make them pass
- The test is the proof that the code works
  - Developer must clearly understand user requirements in order to write tests
  - There should be no functionality in product code that isn't tested
- Encourages simple designs and inspires confidence
  - "Test Driven-Development: Clean code that works" – Ron Jeffries

TDD Workflow

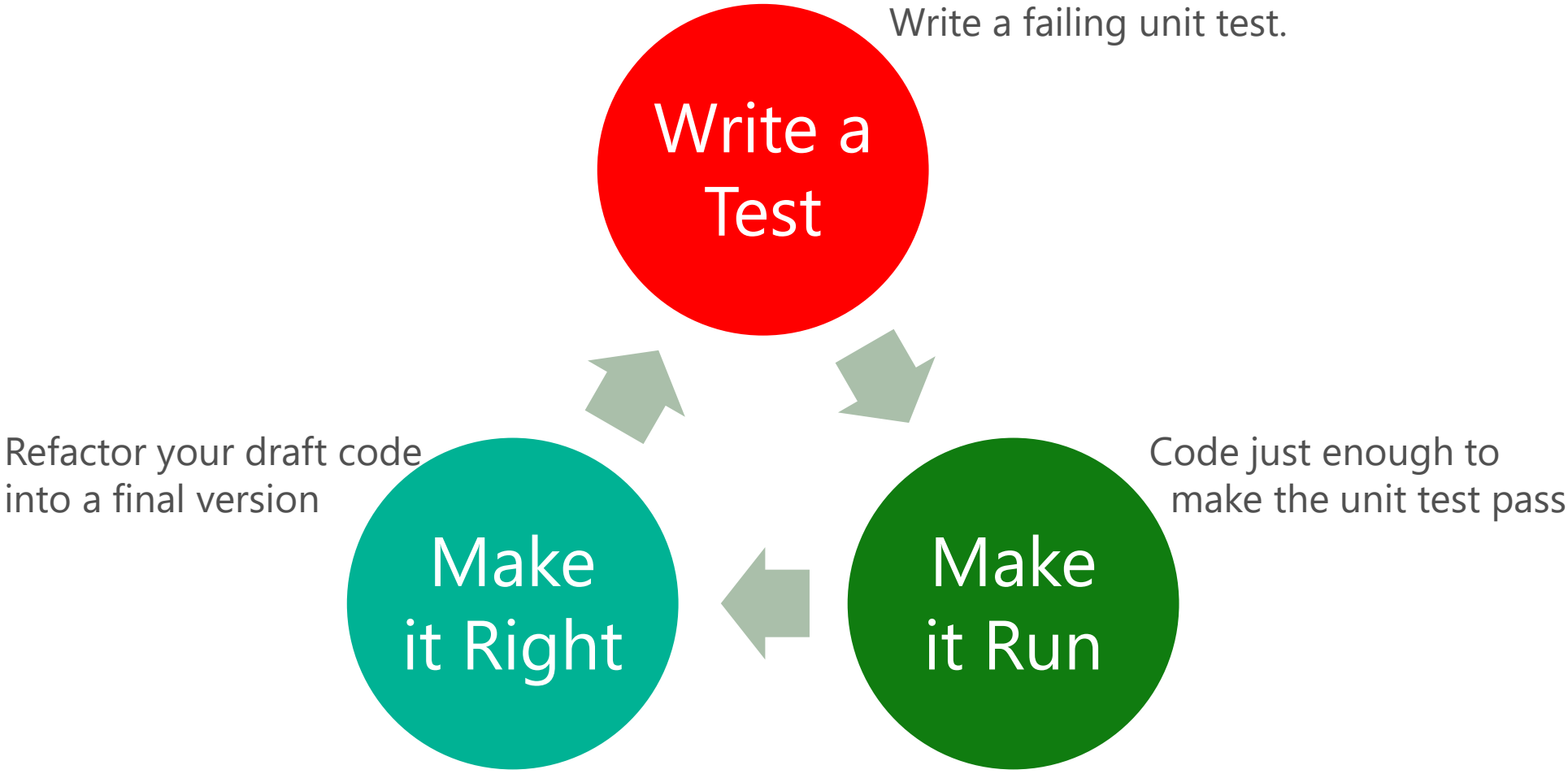


http://en.wikipedia.org/wiki/Test-driven_development

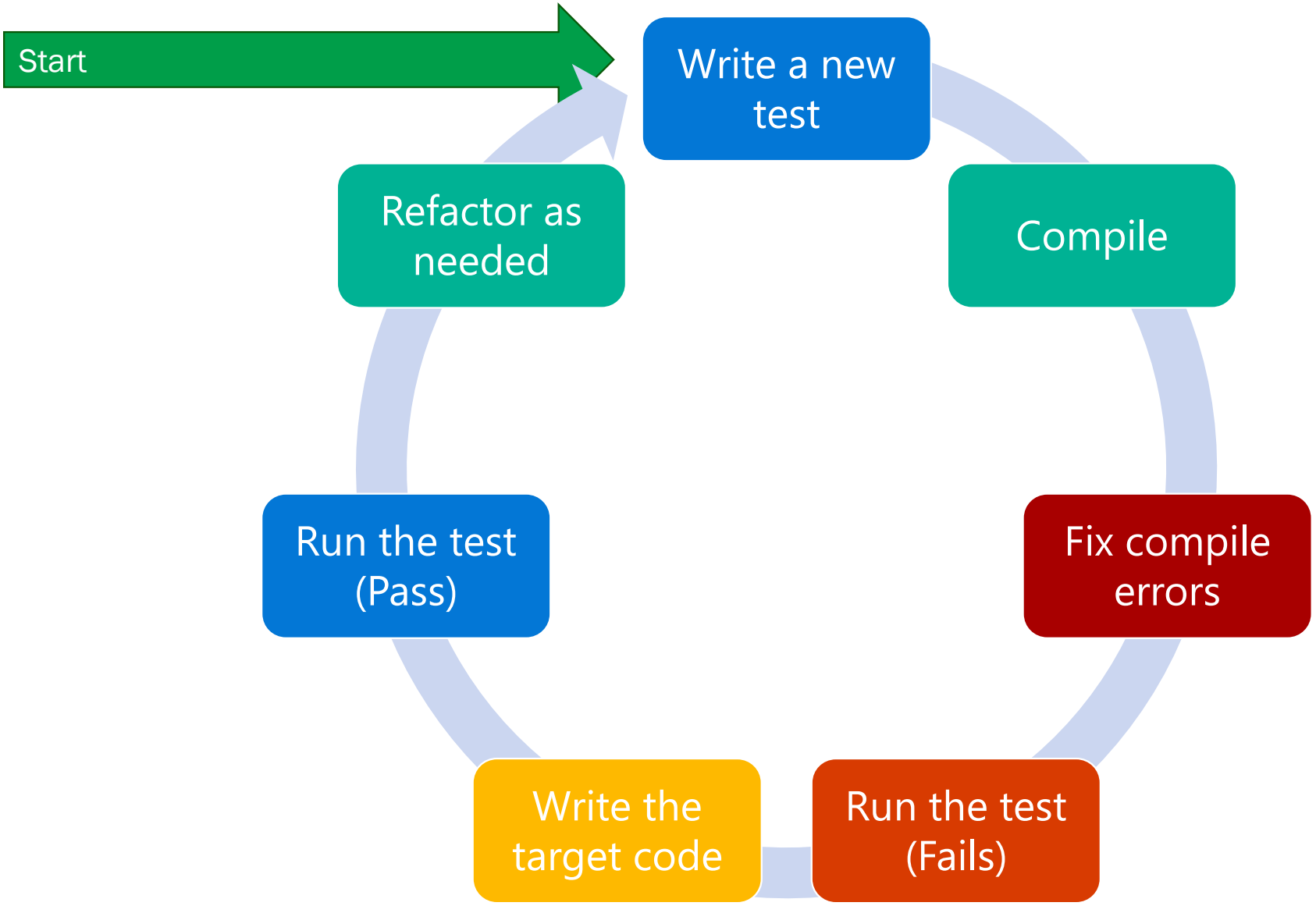
TDD - Red/Green/Refactor



TDD - Red/Green/Refactor



TDD - Red/Green/Refactor

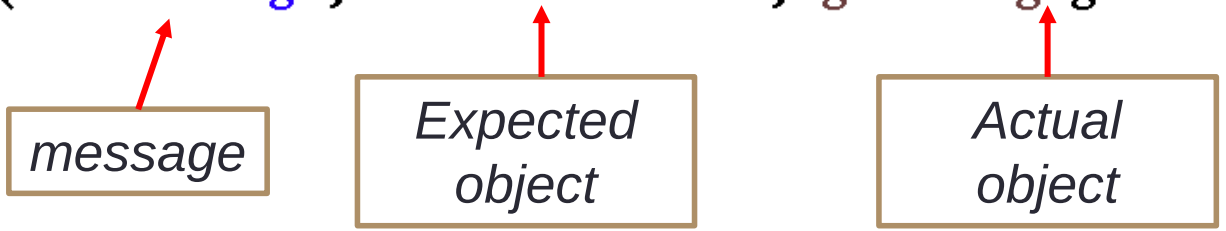


TDD Example

- Requirement: write a program that prints **'Hello World!'** in console.
- The first cycle (red):
  - Write a test to validate this behaviour of the program
  - Create an unit test and start with an assert

GreetingTest.java | Greeting.java

```
1  import static org.junit.Assert.*;
4
5  public class GreetingTest {
6
7      @Test
8      public void test() {
9          Greeting greeting = new Greeting();
10         assertEquals("Sth wrong", "Hello World!", greeting.getMessage());
11     }
12
13 }
```



TDD Example

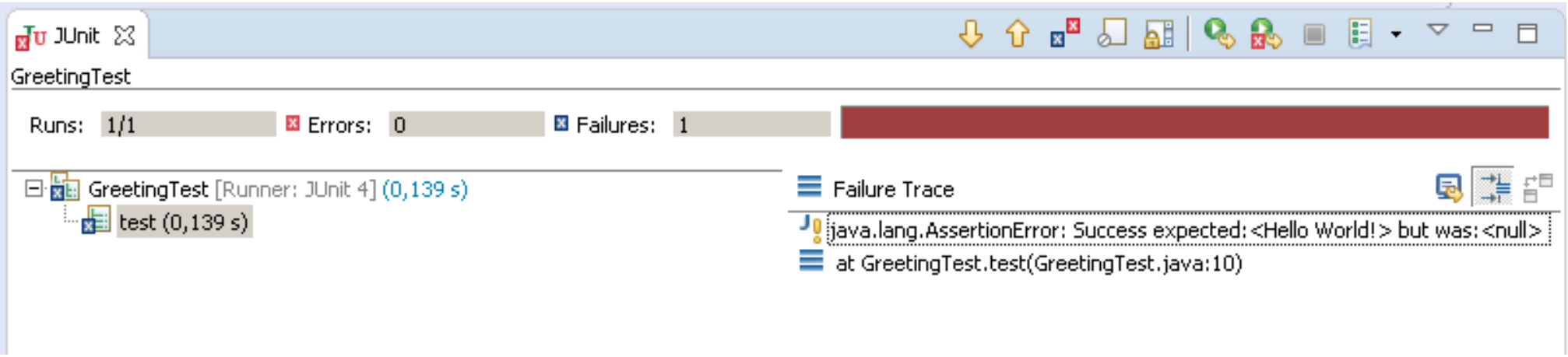
- The first cycle (red):
  - In this example there is a class with a method in it. When we want to call the method, we create an object of that class and call the method on that object.
  - Now there's neither a class nor a method → To get this test to compile, we'll have to create a class **Greeting** and add the method getMessage() in it.
  - **But make sure not to write anything more than what is needed to get the test compiling.**

GreetingTest.java | Greeting.java

```
1 public class Greeting {
2     public Object getMessage() {
3         return null;
4     }
5 }
```

TDD Example

- The first cycle (red):
  - Now we can compile code then run the test and see that it **fails**.



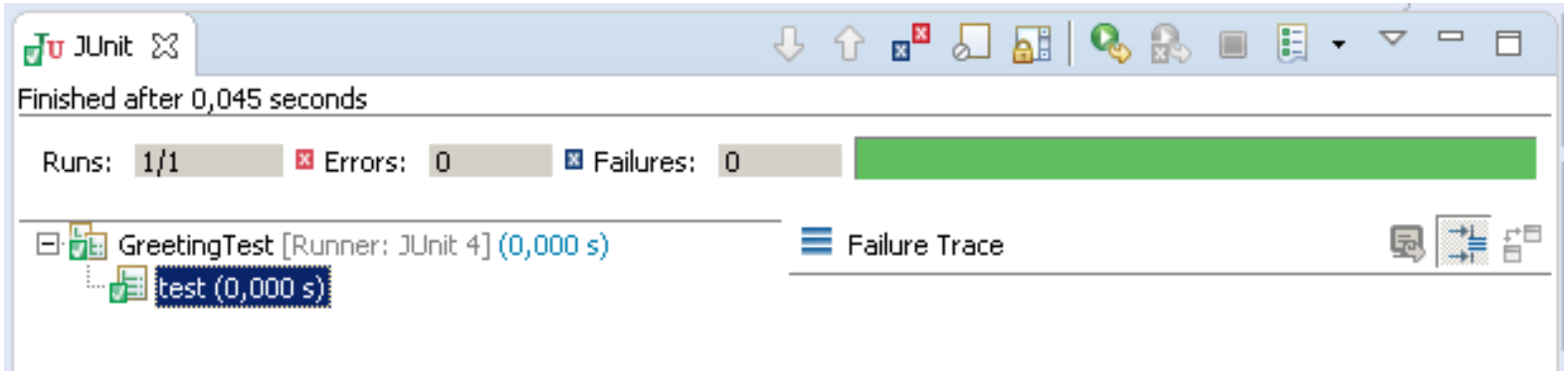
TDD Example

- The second cycle (green):
  - Write the minimum code to pass the test: The very minimum we have to do to pass the test is to return **“Hello world!”** instead of null.

GreetingTest.java | Greeting.java

```
1 public class Greeting {
2     public Object getMessage() {
3         return "Hello World!";
4     }
5 }
```

  - Run the test and see it pass!



TDD Example

- Refactor:
  - Refactor the IMPLEMENTATION:
  1. Remove the auto-generated comment.
  2. Change the return value from Object to String.
  3. Extract the String “Hello World!” into a field.
  4. Initialize the field in the constructor - that refactoring the implementation may result in changes to the test as well.
  - Refactor the TEST:
    - **Make sure to run the test after each change.**
  - After each and every change, run the test to make sure it passes.

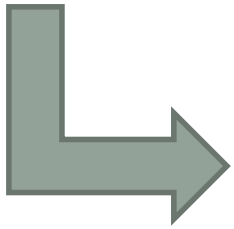
TDD Example

- Refactor:
  - Refactor the IMPLEMENTATION

Before

GreetingTest.java | Greeting.java

```
1 public class Greeting {
2     public Object getMessage() {
3         return "Hello World!";
4     }
5 }
```



After

GreetingTest.java | Greeting.java

```
1  public class Greeting {
2      private String message;
3
4      public Greeting(String message) {
5        this.message = message;
6      }
7
8      public String getMessage() {
9        return message;
10     }
11 }
```

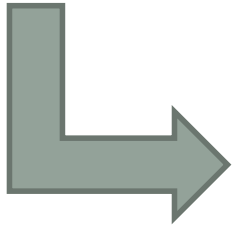
TDD Example

- Refactor:
  - Refactor the TEST

Before

GreetingTest.java | Greeting.java

```
1 import static org.junit.Assert.*;
4
5 public class GreetingTest {
6
7     @Test
8     public void test() {
9         Greeting greeting = new Greeting();
10        assertEquals("Sth wrong", "Hello World!", greeting.getMessage());
11    }
12
13 }
```



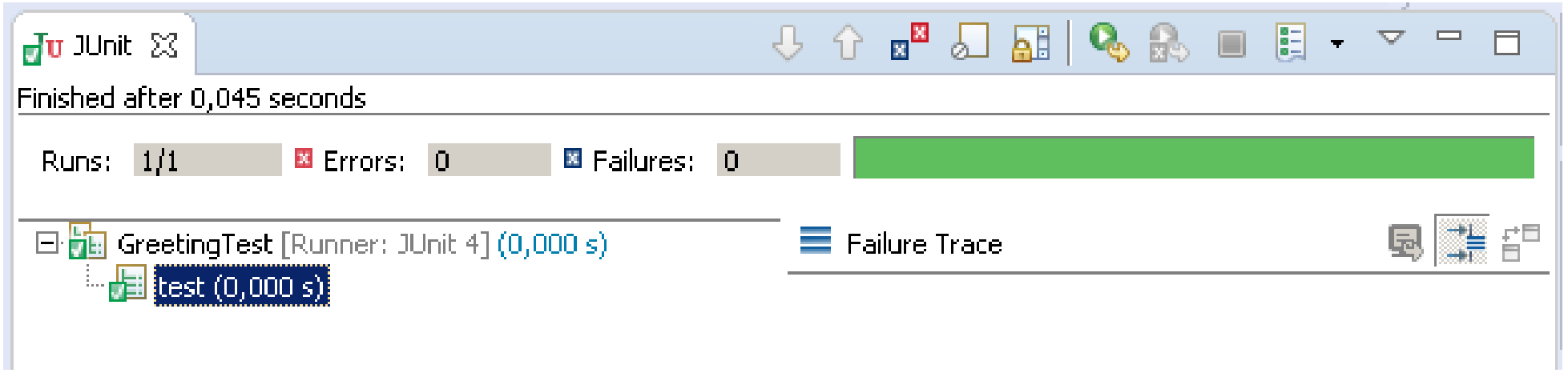
After

GreetingTest.java | Greeting.java

```
1 import static org.junit.Assert.*;
4
5 public class GreetingTest {
6
7     @Test
8     public void test() {
9         Greeting greeting = new Greeting("Hello World!");
10        assertEquals("Sth wrong", "Hello World!", greeting.getMessage());
11    }
12
13 }
```

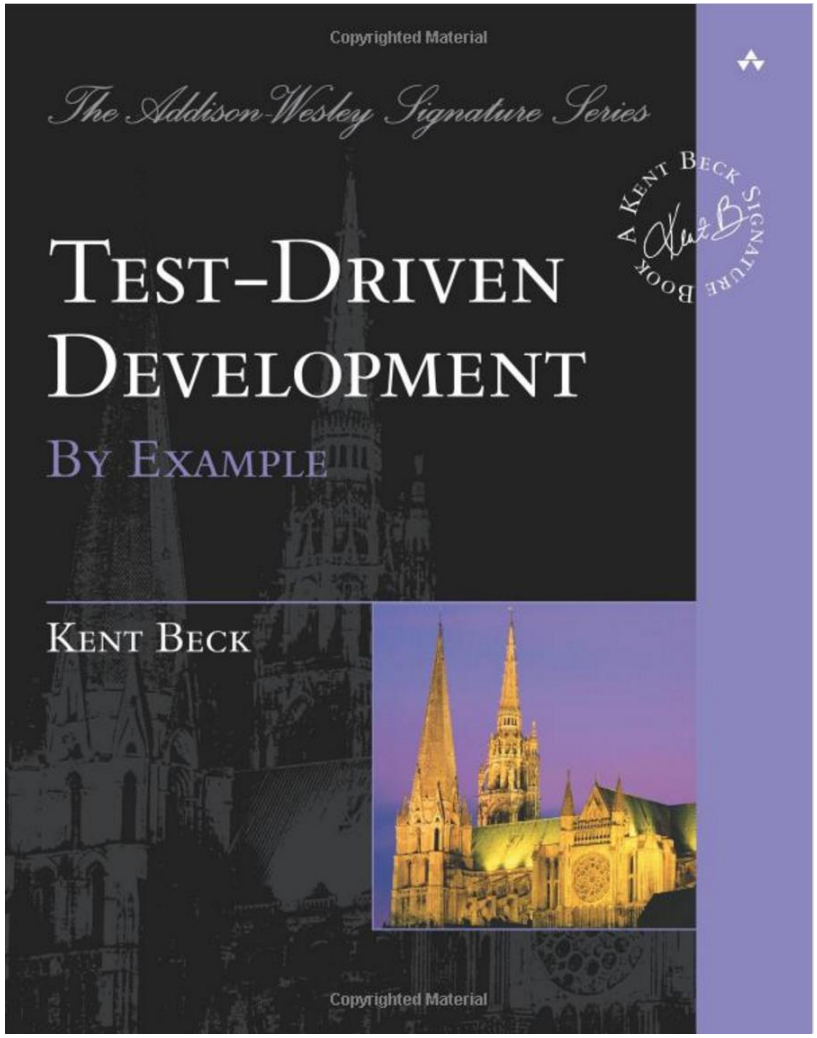
TDD Example

- Refactor:
  - Re-run the test to make sure it pass.



TDD Book

Test-Driven Development By Example by Kent Beck



https://www.amazon.com/Test-Driven-Development-Kent-Beck/dp/0321146530

Appendix: JUnit 5 vs JUnit 4

- Required JDK Version
  - Junit 4 requires Java 5 or higher.
  - Junit 5 requires Java 8 or higher.
- Assertions
  - In Junit 4, **org.junit.Assert** has all assert methods to validate expected and resulted outcomes.
    - They accept extra parameter for error message as FIRST argument in method signature
  - In JUnit 5, **org.junit.jupiter.Assertions** contains most of assert methods including additional assertThrows() and assertAll() methods.
    - assertAll() is in experimental state as of today, and is used for grouped assertions.

https://howtodoinjava.com/junit5/junit-5-vs-junit-4/

Appendix: JUnit 5 vs JUnit 4 (2)

- Architecture
  - JUnit 4 has everything bundled into single jar file.
  - Junit 5 is composed of 3 sub-projects i.e. JUnit Platform, JUnit Jupiter and JUnit Vintage.

1. **JUnit Platform**
   - It defines the TestEngine API for developing new testing frameworks that runs on the platform.

2. **JUnit Jupiter**
   - It has all new junit annotations and TestEngine implementation to run tests written with these annotations.

3. **JUnit Vintage**
   - To support running JUnit 3 and JUnit 4 written tests on the JUnit 5 platform.

https://howtodoinjava.com/junit5/junit-5-vs-junit-4/

Appendix: JUnit 5 vs JUnit 4 (3)

- Tagging and Filtering
  - In Junit 4, @category annotation is used.
  - In Junit 5, @tag annotation is used.
- Test Suites
  - In Junit 4, @RunWith and @Suite annotation.
  - In Junit 5, @RunWith, @SelectPackages and @SelectClasses.
- 3rd Party Integration
  - In Junit 4, there is no integration support for 3rd party plugins and IDEs. They have to rely on reflection.
  - JUnit 5 has dedicated sub-project for this purpose i.e. JUnit Platform. It defines the TestEngine API for developing a testing framework that runs on the platform.

https://howtodoinjava.com/junit5/junit-5-vs-junit-4/

Appendix: JUnit 5 vs JUnit 4 (4)

- Annotations

FEATURE	JUNIT 4	JUNIT 5
Declare a test method	`@Test`	`@Test`
Execute before all test methods in the current class	`@BeforeClass`	`@BeforeAll`
Execute after all test methods in the current class	`@AfterClass`	`@AfterAll`
Execute before each test method	`@Before`	`@BeforeEach`
Execute after each test method	`@After`	`@AfterEach`
Disable a test method / class	`@Ignore`	`@Disabled`
Test factory for dynamic tests	NA	`@TestFactory`
Nested tests	NA	`@Nested`
Tagging and filtering	`@Category`	`@Tag`
Register custom extensions	NA	`@ExtendWith`

https://howtodoinjava.com/junit5/junit-5-vs-junit-4/