JUnit
A regression unit testing framework for Java

Overview

- JUnit provides a framework for programming and managing unit tests in Java
- It provides
  - Support to conveniently write and manage tests
  - And tools to automatically run the defined tests
- It supports *test driven development*
- Re-running tests is easy – *regression testing*
- Integrated into Together/Eclipse, also recent BlueJ, …
- Also available at
  - http://www.junit.org/
  - http://junit.org/junit4/javadoc/latest/
- Currently at version 4.x
  - Different versions have different constructs
  - Version 5 is currently under development

- JUnit was originally written by Erich Gamma and Kent Beck
  - Also associated with *agile approaches*
  - In particular Extreme Programming – test driven
- History (rough):
  - About 1990: Kent Beck: Unit testing for Smalltalk
  - Late 1990s: SUnit
  - Later: Framework ported to Java as JUnit
  - And has spread to many languages: The "xUnit" family
- See http://junit.org/junit4/faq.html
- Note: Although this is *unit testing*, it is easy to use it to organize bottom-up integration testing

Simple example

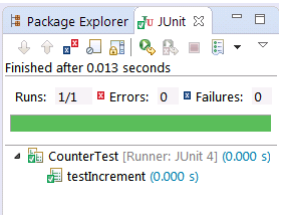
- A simple class to be tested:

```
public class Counter {
  private int count = 0;
  public int getCount() { return count; }
  public void increment() { count++; }
}
```

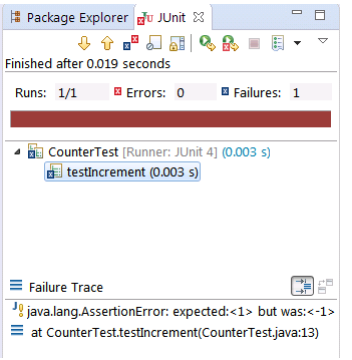
- A simple JUnit test class:

```
public class TestCounter {
  @Test
  public void testIncrement() {
    Counter c = new Counter();
    c.increment();
    assertEquals(1, c.getCount());
  }
}
```

- Running the test class in Eclipse gives:



- But if `counter++` is replaced by `counter--`:



JUnit – general idea

- JUnit allows us to test Java classes
- We make *no change* to the Java class to be tested
- For each *class under test*, we write a *test class* (*driver*)
  - …containing the tests we want to run
  - …each packaged as a *test method* (or *test case*) *that* calls one or more operations offered by the class under test
  - …and checks the results using JUnit *asserts*
- JUnit provides Java *annotations* to tag the parts: e.g. `@Test`
- JUnit provides *test class runners* that execute the set of tests in a test class and generate a report
  - *Asserts succeed silently*
  - *Asserts that fail* generate a report message and abandon that test method
  - The focus is on highlighting the *failures!*

Fixtures

- The test methods may instantiate specific objects to test
- Or the test methods in a test class may need a common set of objects initialized in the same way:
  - Known as a *fixture*
  - This is *set up* before each test method is executed
  - And *torn down* after each test method
  - Set up and tear down are in annotated methods: `@Before`, `@After`
  - The test runner organizes the test sequence:

```
Set up fixture
Execute test method 1
Tear down
Set up fixture
Execute test method 2
Tear down
etc
```


Test classes – other points

- The roles of the various parts of a test class are indicated by Java *annotations*
  - The test runner uses Java "reflection" to identify the roles of the parts
- Test classes may also contain
  - "Global" variables – useful for fixtures
  - Private helper methods
- It is also possible to specify set up/tear down code to run *exactly once* before/after running the whole test class
  - Methods annotated with `@BeforeClass`, `@AfterClass`
- Test classes may be grouped into *test suites*
  - Facilitate easy "one click" regression testing

A wide range of assert methods

- All the `assert…` methods are:
  - Imported from JUnit library `org.junit.Assert`
  - `public static void assert…(<type> expected, <type> actual)`
- All assert… methods have an optional first parameter:
  - `… assert…(String message, <type> exp, <type> act)`
  - The message is included in a *failure* report
- Some of the range:
  - `assertEquals(int expected, int actual)`
  - also other standard types
  - `assertTrue(boolean condition)`
  - `assertFalse(boolean condition)`
  - `assertNull(Object reference)`
  - `assertNotNull(Object reference)`
  - `fail()`

Testing example: Theatre seat booking

- Here is the theatre seat class (state diagram example):

```
public class Seat {
  public static final int NOTBOOKED = 0,
                          RESERVED = 1, BOOKED = 2;
  private int state = NOTBOOKED;
  public void reserve() {
    if (state != NOTBOOKED) return;
    state = RESERVED;
  }
  public void cancel() {
    if (state != RESERVED) return;
    state = NOTBOOKED;
  }
  public void pay() {
    if (state == BOOKED) return;
    state = BOOKED;
  }
  public int getState() { return state; }
}
```

- Test plan:
  - Fixture: one `Seat` object
  - Test *construction*: successful and in `NOTBOOKED` state
  - Test `reserve`: in all three states: check the resulting state:

    `NOTBOOKED -> RESERVED`
    `RESERVED -> RESERVED`
    `BOOKED -> BOOKED`

    Could be one test method or three
  - Test `pay` similarly
  - Test `cancel` similarly
  - Need to test `getState`?

- A possible `SeatTest` class:

```
public class SeatTest {
  private Seat aSeat;   // The fixture

  @Before
  public void setUp() throws Exception {
    aSeat = new Seat();
  }

  @After
  public void tearDown() throws Exception {
    aSeat = null;
  }

  @Test
  public void testCreate() {
    assertNotNull("Seat not created properly", aSeat);
    assertEquals("Initial Seat state is wrong",
                 Seat.NOTBOOKED, aSeat.getState());
  }
```

- `SeatTest` continues:

```
@Test
public void testReserve() {  // All in one method
  // Initially not booked, attempt reserve
  aSeat.reserve();
  assertEquals("Reserving a not booked seat fails",
               Seat.RESERVED, aSeat.getState());
  // State is now reserved
  aSeat.reserve();
  assertEquals("Reserving a reserved seat fails",
              Seat.RESERVED, aSeat.getState());
  // Should still be reserved
  aSeat.pay();
  // Should now be booked
  aSeat.reserve();
  assertEquals("Reserving a booked seat fails",
              Seat.BOOKED, aSeat.getState());
}
```


- `SeatTest` continues:

```
@Test
public void testCancel() {
  fail("Not yet implemented");  // To be done
}

@Test
public void testPay() {
  fail("Not yet implemented");  // To be done
}
```

Some final remarks

- Must consider *negative* as well as *positive* tests
- JUnit can also handle testing whether
  - Exceptions are thrown when expected
    (that is: an exception is to be a success, not a failure!)
  - Results are produced within timeout periods
- A bug slips through? The inspiration to add a new test!
- Test driven development:
  - Given a specification, write the tests (black box)
  - Then code and keep testing until the tests all succeed
  - The code is ready! (or the tests are poor)
- Writing tests can *reduce* overall development time
- "Run all your unit tests as often as possible, ideally every time the code is changed." (JUnit FAQ)
- "Test until fear turns to boredom." (JUnit FAQ)

End of lecture