



Basics of programming 3

Unit tests in Java: JUnit



Unit tests

- Verification and validation has many levels
 - system tests
 - integration tests
 - unit tests
 - etc
- Testing a single unit is *unit test*
 - units in OO are classes and objects
- Automatism and repeatability are important
 - regression tests



Unit testing

- Small part of the software is tested
 - Single class or method
 - Each and every non-trivial method
- Tests are independent
 - Tests are stateless
- Developer and tester should be different persons



Unit testing – classical approach

- Code review
 - Useful if rules are observed
 - Not enough
- Manual testing
 - Develop tester applications
 - Simple
 - Becomes unmaintainable with time
 - Test are not organised
 - Results are not coherent



Unit testing – manual approach


- `System.out.println()`
 - Continuous diagnostic messages
 - Simple
 - Code is full with `println`-s
 - how to turn off?
 - Output tends to be unreadable
 - Manual control is needed



Unit testing – manual approach

■ Debugger

- IDE support for observing variables
- Slow
- Cumbersome for complex (multithreaded) applications
- Has to be done after each change
- Still manual



Unit testing – frameworks

- XUnit for many languages and environments
 - CppUnit (C++)
 - unittest (python)
 - etc.
- JUnit
 - open source Java testing framework
 - available as a JAR file
 - tests are written in Java
 - IDE-s provide built-in support
 - separate windows, perspectives, etc



JUnit features

- Assertions for testing expected results
 - standard result checks
- Test fixtures for sharing common test data
 - common functionality written once
- Test runners for running tests
 - automated testing
 - regression is easy



JUnit example

- Simple *integer* implementation

```
public class MyInt {
    private int value;
    public MyInt(int aValue) {
        value = aValue;
    }
    public void add(MyInt anInt) {
        value += anInt.getValue();
    }
    public int getValue() {
        return value;
    }
}
```

Example test

■ Simple test – naïve

- Create some objects – testing context, fixture
- Send messages to those objects
- Verify some assertions

```
public class MyTest {  
    public static void main(String[] args) {  
        MyInt m1 = new MyInt(5);  
        MyInt m2 = new MyInt(30);  
        m1.add(m2);  
        if (m1.getValue() != 35)  
            System.out.println("Sum failed");  
        if (m2.getValue() != 30)  
            System.out.println("m2 failed");  
    }  
}
```

Test run

Initialization

Check

Example JUnit test

```
public class MyIntTest1 {  
    MyInt m1, m2;  
    @Before  
    public void setUp() {  
        m1 = new MyInt(5);  
        m2 = new MyInt(30);  
    }  
    @Test  
    public void testAddInt() {  
        m1.add(m2);  
        assertEquals("sum Test", 35, m1.getValue());  
        assertEquals("m2 Test", 30, m2.getValue());  
    }  
}
```

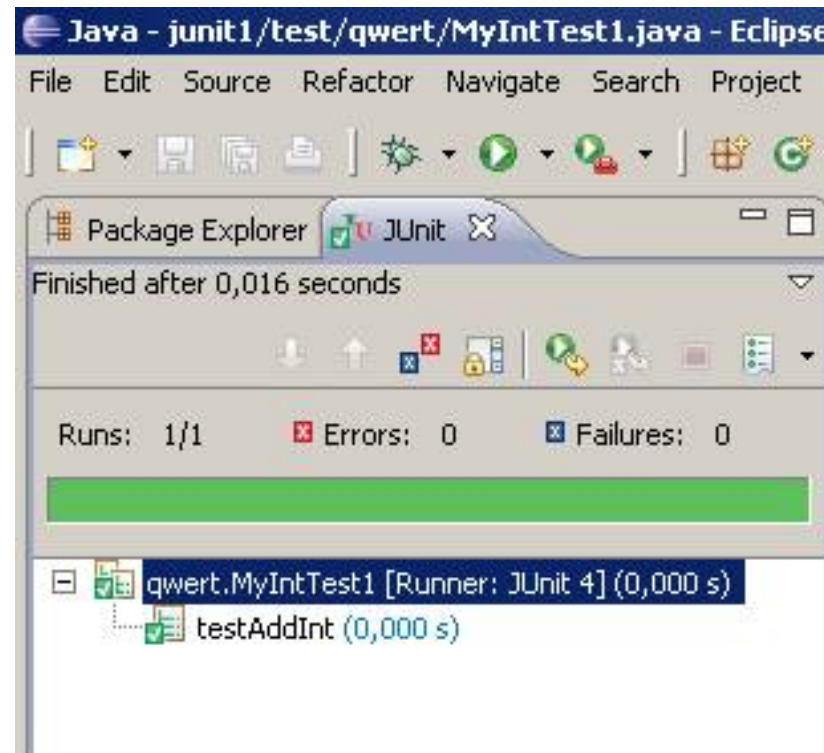
Initialization

Check

Test run

JUnit in Eclipse

- Java Build Path/Libraries/Add Library/JUnit 4
- Run As/JUnit Test





Test method

■ Constraints

- Each test is implemented as a method
- It takes no parameters and returns no value
- Test methods must be public
- Annotated by **@Test**
- Default test order is undefined but deterministic
 - order not known, but always the same
 - class annotation for lexicographic ordering (v4.11):
@FixMethodOrder(MethodSorters.NAME_ASCENDING)



Fixtures

■ Intro

- combine tests for a common set of objects
- e.g. initialization, clean-up etc
- tests don't share the objects
 - each test separately tests its own set of objects
- common objects are instance variables



Fixtures 2

- Types

- **@Before**

- called before each test: builds the context

- **@After**

- called after each test: tears down the context

- **@BeforeClass / @Afterclass**

- called before first test / after last test
 - for resource-intensive objects and initialization

Example JUnit test

```
public class MyIntTest1 {
    MyInt m1, m2;
    @Before
    public void setUp() {
        m1 = new MyInt(5);
        m2 = new MyInt(30);
    }
    @Test
    public void testAddInt() {
        m1.add(m2);
        assertEquals("sum Test", 35, m1.getValue());
        assertEquals("m2 Test", 30, m2.getValue());
    }
}
```

Initialization

Check

Test run



Fixtures and tests

- Execution order for two tests:

- @BeforeClass methods*

- @Before methods*

- @Test method #1*

- @After methods*

- @Before methods*

- @Test method #2*

- @After methods*

- @AfterClass methods*

Testing results

■ How to check if result is correct?

msg for exception
when fail

- static void **assertTrue**([String msg,] boolean condition)
- static void **assertFalse**([String msg,] boolean condition)
- static void **assertNull**([String msg,] Object object)
- static void **assertNotNull**([String msg,] Object object)
- static void **assertSame**([String msg,] Object exp, Object act)
- static void **assertNotSame**([String msg,] Object unexp, Object act)
- static void **assertEquals**([String msg,] X exp, X act)
- static void **assertArrayEquals**([String msg,] X exp, X act)
- static void **fail**([String msg])



Running tests

- Command line

- `java org.junit.runner.JUnitCore TestClass1
[...other test classes...]`

- Inside application

- `org.junit.runner.JUnitCore.
runClasses(TestClass1.class, ...);`

- Inside IDE

- click on *run tests...*



Test results

- Success
 - OK
- Failure
 - result is different from expected
 - tests fail if any assertion fails
- Error
 - unexpected exception was thrown
- Ignore
 - test was ignored (*assume* or *@Ignore*)



Test results 2

- Expecting exceptions

 - `@Test (expected=NumberFormatException.class) ...`

- Setting timeout

 - `@Test (timeut=100) ...`

- Ignoring test

 - `@Ignore ("some message") @Test ...`

 - using `Assume.assumeXXX` method changes fails into ignores

 - `assertNotNull(obj) → assumeNotNull(obj)`
if `obj` is null, test is ignored (instead of fail)



Rules

- Same init for different test classes

- put code into subclass of *ExternalResource*

- *public void before()*: runs before each test

- *public void after()*: runs after each test

- newly constructed for each test

- add resource class to test

- ```
@Rule public ExternalResource resource =
 new MyExternalResource();
```

- class level rules (like *BeforeClass*, etc)

- ```
@ClassRule ...
```



Subclassing test classes

- Test execution for subclass tests
 - bottom-up in inheritance hierarchy
- Before execution
 - top-down in inheritance hierarchy
- After execution
 - bottom-up in inheritance hierarchy



Parameterized testing

- For same test with different parameters
 - instances are created for the cross-product of the test methods and the test data elements

```
@RunWith(value = Parameterized.class)
public class ParamTest {
    private int a, b;
    public ParamTest(int a1, int b1) {a = a1; b = b1;}
    @Parameters public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][]{{1,5},{4,9},{2,7}});
    }
    @Test public void runTest() {Assert.assertTrue(a < b);}
}
```




Creating test suites

■ Grouping tests together

```
@RunWith (Suite.class)
@Suite.SuiteClasses ({
    TestFeatureLogin.class,
    TestFeatureLogout.class,
    TestFeatureNavigate.class,
    TestFeatureUpdate.class
})
public class FeatureTestSuite {
    // empty class holding the annotations
}
```



Categories of tests

- Tests can be annotated with categories
- Categories are simple annotations
- Tests can be category-annotated both on method and class level

```
@Category(categoryType1.class)
```

```
@Category({categoryType1.class,  
           categoryType2.class})
```

Category example

```
public interface FastTests {}
public interface SlowTests {}

public class A {
    @Test public void a() { ... }

    @Category(SlowTests.class)
    @Test public void b() { ... }
}

@Category({SlowTests.class, FastTests.class})
public class B {
    @Test public void c() { ... }
}
```

category markers



Categories used in suites

- In test suites one can select a set of categories

```
@RunWith( Categories.class )
@IncludeCategory( SlowTests.class )
@ExcludeCategory( FastTests.class )
@SuiteClasses( { A.class, B.class } )
public class SlowTestSuite {
    // Will run only test annotated
    // with SlowTests in test cases A and B
}
```



JUnit conventions

- Separate tests from sources
 - Usually separate directories (**src** vs. **test**)
 - Final application doesn't contain tests
- Test classes in same package as tested classes
 - Allows tests to access package, protected members
- For each tested class a single test class
 - Not a strict rule 😊



Concurrent testing

- Testing concurrent classes is hard

- help: *ConcurrentUnit*

1. Create a *Waiter*
2. Use *Waiter.await* to block the main test thread.
3. Use the *Waiter.assert* calls from any thread to perform assertions.
4. Once expected assertions are completed, use *Waiter.resume* call to unblock the main thread.

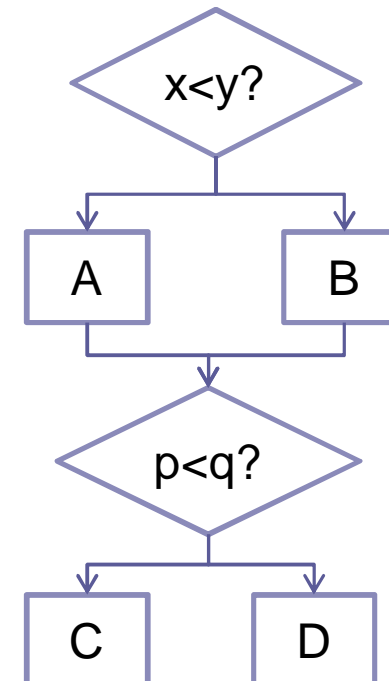
Test coverage

- How much of the code is tested?

- coverage = tested / total
- including exceptions
- static vs dynamic check
 - are all paths tested?

- Goal: 100% coverage

- bugs might still remain ☹️





Runtime checks

- Not considered testing
- Design by contract
 - precondition
 - method expects this from callers
 - invariant
 - what does not change during method call
 - postcondition
 - method assures this condition
 - Supported by some languages (e.g. Eiffel)



Design by contract example

```
public interface Stack<T> {  
    /** Pushes t on top of Stack */  
    void push(T t);  
    /** removes topmost element */  
    T pop();  
    /** returns topmost element */  
    T top();  
    /** returns number of stored elements */  
    int size();  
}
```



Design by contract example

■ `void push(T t)`

- *invariant*: all previously pushed items retained
order does not change
- *post*: `t` is on top

■ `void pop()`

- *pre*: stack is not empty
- *invariant*: all but top previously pushed items retained
order does not change
- *post*: topmost element is removed



Design by contract example

■ **T top()**

- *pre:* stack is not empty
- *invariant:* all pushed items retained
order does not change
- *post:* returned element is topmost element

■ **int size()**

- *invariant:* all pushed items retained
order does not change
- *post:* returned value equals number of items



Tests vs Design by contract

- Tests can be generated from DbC
 - pre, inv, post must be formally specified
 - conditions can be turned into test cases
- Tests during development
 - tests check software before regular use
- DbC during regular use
 - what to do if something happens?
 - mostly for prototyping



Java: assert keyword

- Runtime check of condition

- *assert expression;*

- e.g. `assert (stack.size() > 0);`

- *assert expression1 : expression2;*

- e.g. `assert (i % 5 == 0 : i);`

- assert cause is set to expression2

- When assertion fails

- *AssertionError* is thrown

- its an error -> should not be handled, may not be indicated



Java assert rules

- No public method parameter checking
 - use *IllegalArgumentException*, *NullPointerException*, etc instead
- No regular work should be done
 - e.g. `assert (stack.pop() == x); // NO!`
`Object o = stack.pop();`
`assert (o==x); // OK`
- Allowing assertions
 - `javac -source 1.4 MyClass.java`
 - otherwise *assert* is not a keyword



Java assert runtime enabling

■ For packages and classes

- *-enableassertions* Or *-ea*
- *-disableassertions* Or *-da*
 - arguments (like *-ea:hu.bme.iit...*)
 - *none*: for whole application
 - `packageName . . .`: for package and subpackages
 - `. . .`: for the unnamed package
 - `classname`: for the given class

■ For system classes

- *-enablesystemassertions* or *-esa*
- *-disablesystemassertions* or *-dsa*