JUnit

Lecture 9

Testing

- Testing helps increase our confidence in our code
  - "If it isn't tested, assume it doesn't work"
- Testing is a comparison:
  - Expected behavior of the component
    - See Javadoc description
  - Actual behavior of the component
    - Run the code
- Three parts:
  - Implementation, specification, test cases
- Some believe in test-driven development
  - Write tests first!
  - Then write code so that all tests compile
  - Then refine code so that all tests pass
  - Repeat: write more tests, refine code so they pass

Writing Good Tests

- Goal: to expose problems!
  - Assume role of an adversary
  - Failure == success
- Test boundary conditions
  - eg 0, Integer.MAX_VALUE, empty array
- Test different categories of input
  - eg positive, negative, and zero
- Test different categories of behavior
  - eg each menu option, each error message
- Test "unexpected" input
  - eg null pointer, last name includes a space
- Test representative "normal" input
  - eg random, reasonable values

Primitive Testing: println

- Console IO to observe actual behavior
- Compare IO with expected output
- See TestRandom example
- Advantages:
  - Testing code is simple, easy, intuitive
- Problems:
  - Exhaustive testing means lots of output
  - Comparison is tiresome and error-prone
  - Difficult to automate

More Serious Testing: JUnit

- A "framework" for testing Java code
  - Frameworks are libraries with gaps
  - Programmer writes classes following particular conventions to fill in gaps
  - Result is the complete product
- Current version of JUnit: 4 (4.4)
  - JUnit 4.3.1 is bundled with Eclipse 3.4
  - Big changes from JUnit 3.8
  - Beware: most information available online is about 3.8

Example: RandomWithParityTest


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

public class RandomWithParityTest {
  private RandomWithParity p; //coding to the interface

  @Test public void minRange() {
    p = new UnfilteredRandom();
    int actual = p.generateNumber(1);
    assertEquals ("Smallest range", 0, actual % 2);
  }
  @Test public void maxRange() {
    p = new UnfilteredRandom();
    int actual = p.generateNumber(Integer.MAX_VALUE);
    assertEquals ("Largest range", 0, actual % 2);
  }
}
```

Running from the command line (with JUnit in the classpath):

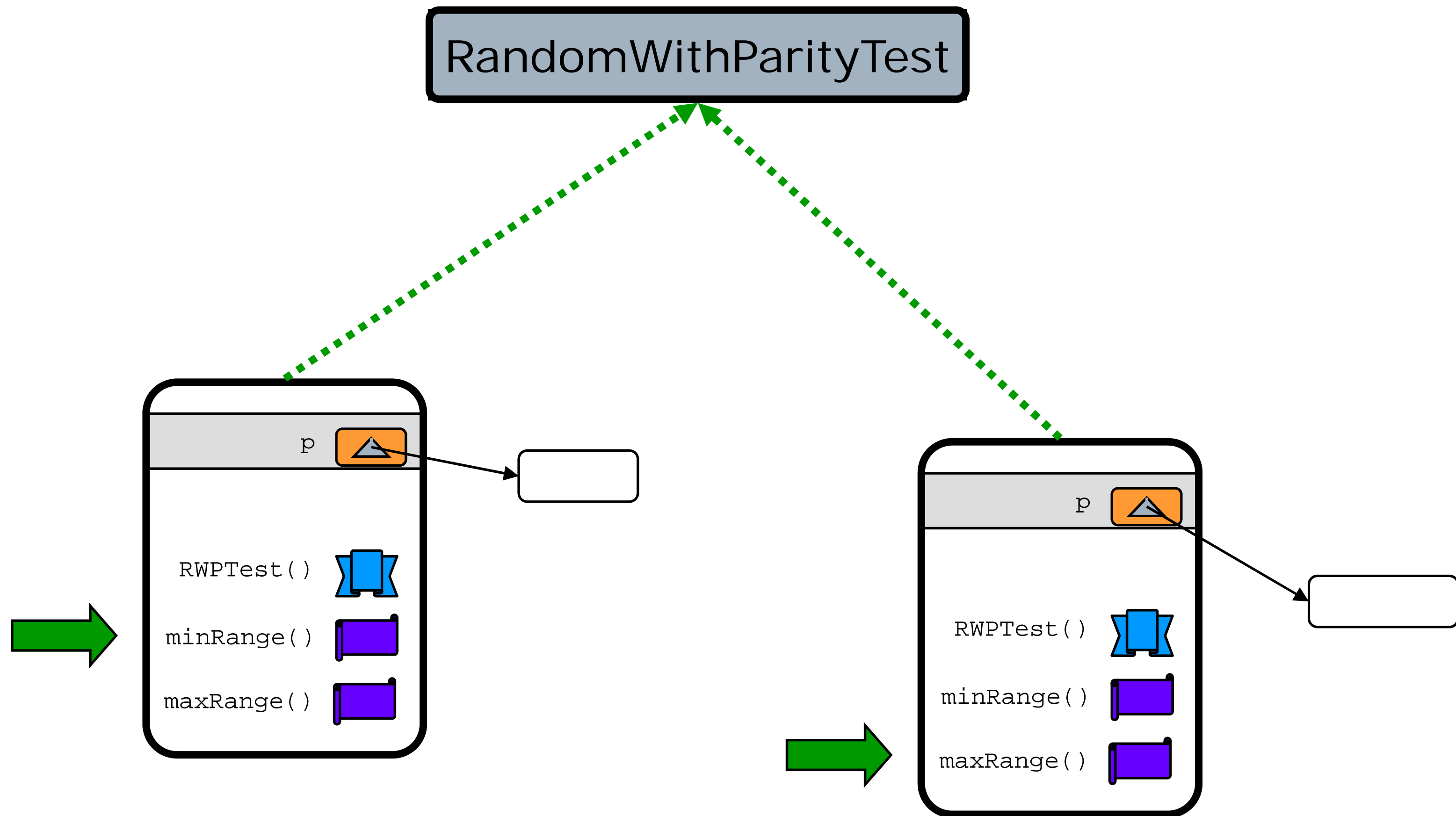
java org.junit.runner.JUnitCore RandomWithParityTest

But we will use Eclipse instead (more later …)

Vocabulary

- Test case
  - Exercises a single unit of code / behavior / functionality
  - Test cases should be *small* (ie test one thing)
  - Test cases should be *independent*
  - In JUnit: A public method marked with @Test
- Test fixture
  - Exercises a single class
  - A collection of *test cases*
  - In JUnit: A class containing @Test methods
- Test suite
  - Exercises all (or most) classes in a program
  - A collection of *test fixtures*
  - In JUnit: A class marked with @Suite

Execution Model: Multiple Instances



Execution Model: Implications

- Separate instances of test class created
  - One instance / test method
- Do not use test cases with side effects
  - Passing or failing one test case should not affect the others
- Do not rely on order of tests
  - Method listed first not guaranteed to be executed first
- Fixture: common set-up to all test cases
  - Field for instance of class being tested
  - Factor initialization code into its own method
  - Mark this method(s) with @Before

Good Practice: @Before

- Initialize a fixture with a setup method (ie marked with @Before) rather than the constructor
- Reasons:
  - If the code being tested throws an exception *during the setup*, the output is much more meaningful
  - Symmetry with @After method for cleaning up after a test case

Example: RandomWithParityTest

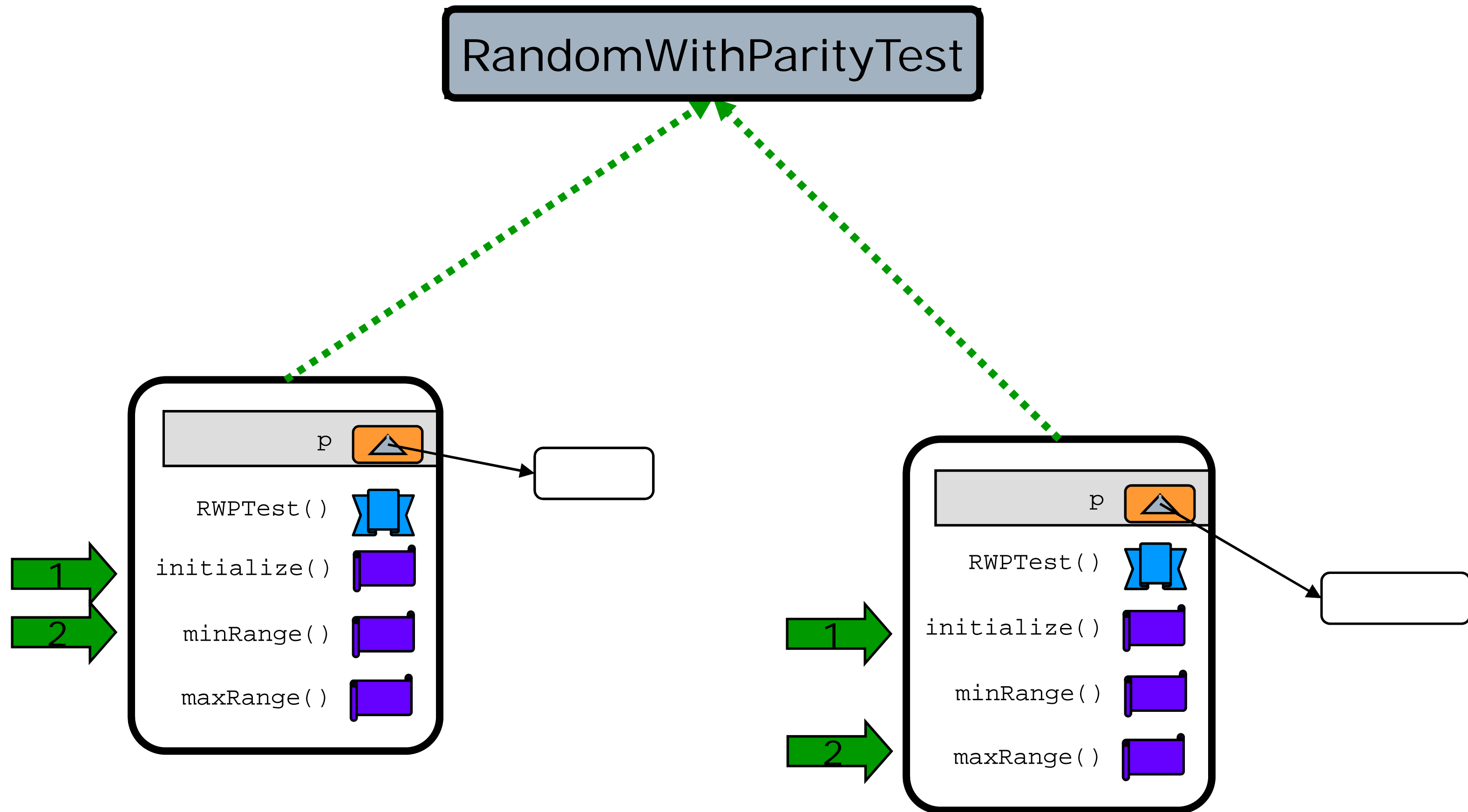

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;
public class RandomWithParityTest {
  private RandomWithParity p;

  @Before public void initialize() {
    p = new UnfilteredRandom();
  }

  @Test public void minRange() {
    int actual = p.generateNumber(1);
    assertEquals ("Smallest range", 0, actual % 2);
  }

  @Test public void maxRange() {
    int actual = p.generateNumber(Integer.MAX_VALUE);
    assertEquals ("Largest range", 0, actual % 2);
  }
}
```

Execution Model



Practice: Anachronisms

- Common, but out-dated, idioms (to avoid)
- Test method names start with "test"
  - This used to be a requirement (prior to JUnit 4)
  - Now use @Test annotation and name method something appropriate
- Set up (tear down) method named setUp (tearDown)
  - This used to be a requirement (prior to JUnit 4)
  - Now use @Before (@After) annotation and name method something appropriate
- A static method called suite()

```
public static junit.framework.Test suite() {
  return new JUnit4TestAdapter(ThingTest.class);
}
```

  - Allows JUnit 4 tests to be run by older JUnit frameworks and tools

Assertions

- Different kinds of tests
  - Static methods of org.junit.Assert

```
assertEquals (message, expected, actual);
assertTrue (message, condition);
assertFalse (message, condition);
assertNull (message, object);
assertNotNull (message, object);
```

- Timed tests
  - Parameterize @Test with timeout
  - Long argument is number of ms allowed for

```
@Test(timeout=100) public void maxRange() {
  int actual = p.generateNumber(1);
  assertTrue ("Largest range", actual%2==0);
}
```

Good Practice: assertEquals

- Prefer assertEquals to assertTrue
  - assertEquals is overloaded
    - Expected and actual can be primitives or references
  - Failed test case produces useful output

```
org.junit.ComparisonFailure: Age at birth expected:
    <0> but was: <1>
```

  - Compare with assertTrue

```
java.lang.AssertionError: Age at birth
```

- Use 3-argument version
  - 1st argument: String to display on failure

```
assertEquals(String msg, int expected, int actual)
```

- For now, avoid using assertEquals to directly compare instances of your own classes
  - assertEquals on Java classes (Integer, String…)? OK
  - assertEquals on your classes (Pencil…)? later

Good Practice: Comparing Floats

- Never compare floating point numbers directly for equality

```
assertEquals("Low-density experiment",
   1.456, calculated);
```

  - Numeric instabilities make exact equality problematic
- Better approach: Equality with tolerance

```
assertEquals("Low-density experiment",
   1.456, calculated, 0.001);
```

Eclipse Demo

- New > JUnit Test Case
- First screen of wizard:
  - Checkbox "New JUnit 4 Test"
  - Enter name of test class (eg ThingTest)
  - Enter name of "class under test" (eg Thing)
  - If warning "JUnit 4 not on build path" appears, click link to add it to build path
- Second screen of wizard:
  - Select methods to test
  - Generates one test case / selected method
  - But you will need many more than that
- To run, Run As… > JUnit Test Case

Specification vs Implementation

- Tests can be written for either
  - Specification tests test only behavior promised in Javadoc of *interface*
  - Implementation tests test all behavior documented in Javadoc of *class*
- Examples:
  - Interface does not guarantee order of elements in a returned array, but implementation always has them in sorted order
  - RandomWithParity guarantees only even/odd values, AlternatingCoin gives 0,1,0,…
- Specification tests work for all (correct) classes implementing the given interface
  - See RandomWithParityTest

Test Suite

- To run multiple test classes, they can be bundled together into a test suite

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

@RunWith(Suite.class)
@Suite.SuiteClasses({
  RandomWithParityTest.class,
  CoinAlternatingTest.class,
  UnfilteredRandomTest.class,
})
public class VegasSuite {
  //the class remains completely empty,
  //used only as holder for above annotations
}
```

- Eclipse also allows running "all JUnit tests in package"
  - Preferred because no extra book-keeping, but Eclipse-specific

Good Practice: Organization

- Keep test classes in the same project as the code
  - They are part of the build
  - Helps to keep tests current
- Name test classes consistently
  - eg WritingStickTest tests WritingStick
- Group tests in same package, but different source folder as the code
  - Eg project X9, package osu.cse:
    - Code: X9/src/osu/cse/WritingStick.java
    - Tests: X9/test/osu/cse/WritingStickTest.java
  - Tests can see public and package-visible stuff

Supplemental Reading

- JUnit web site
  - http://www.junit.org
  - See "Getting Started"
- JUnit FAQ
  - http://junit.sourceforge.net/doc/faq/faq.htm
- JUnit cookbook
  - http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- IBM developerWorks
  - "An Early Look at JUnit 4"
  - http://www-128.ibm.com/developerworks/java/library/j-junit4.html
  - Assumes JUnit 3.8 background

Summary

- Nature of testing
  - Specification, implementation, test cases
- JUnit overview
  - Test case: method marked with @Test
  - Test fixture: class collecting common tests
  - Test suite: set of fixtures
  - Assertions
- Execution model
  - Multiple instantiation of test class
  - Independence of test cases
  - No ordering guarantee