JUnit

Primitive Testing

- Write `main` as a ***command interpreter*** with console input/output, so user (tester) provides inputs and observes actual results (as in some recent lab skeletons)
- Tester compares actual results with allowed/expected results by ***inspection***
- Pros/cons:
  - Simple, easy, intuitive
  - Tedious, error-prone, not automated

Example

```
String command = getCommand(in, out);
while (!command.equals("q")) {
  if (command.equals("i")) {
    out.print("Enter a natural number: ");
    NaturalNumber n =
        new NaturalNumber2(in.nextLine());
    out.println("Before increment: n = " + n);
    increment(n);
    out.println("After increment:  n = " + n);
  } else if (command.equals("d")) {...}
  command = getCommand(in, out);
}
```

More Automated Testing

- Write `main` to contain sets of inputs and expected results in “parallel arrays” of argument values and expected results (as in some other recent lab skeletons)
- Simple loop in `main` compares actual results with allowed/expected results
- Pros/cons:
  - Better, primarily because the process is now far more automatic

Example

```
final int[] numbers = { 0, 0, 1, 82, 3, 9, 27, 81, 243 };
final int[] roots = { 1, 2, 3, 2, 17, 2, 3, 4, 5};
final int[] results = { 0, 0, 1, 9, 1, 3, 3, 3, 3 };
for (int i = 0; i < numbers.length; i++) {
  int x = root(numbers[i], roots[i]);
  if (x == results[i]) {
    out.println("Test passed: root(" + numbers[i]
        + ", " + roots[i] + ") = " + x);
  } else {
    out.println("*** Test failed: root(" + numbers[i]
        + ", " + roots[i] + ") expected " + results[i]
        + " but was " + x);
  }
}
```

Remaining Problems

- One new drawback of this approach is that you need to be able to write the values of the arguments and expected results using Java literals in the array initializations
  - This does not work for some types, where each set of input values and/or expected results must be created by performing a series of method calls

Remaining Problems

- Another drawback of this approach is that, if there are multiple allowed results for the given arguments, mere equality checking with actual results *does not work*
  - Recall the `aFactor` method; what happens if we write in the `results` array that *the* "expected" result is 6, when any of 1, 2, 3, or 6 (and maybe other results) are *also* allowed?

Serious Testing: JUnit

- ***JUnit*** is an industry-standard “framework” for testing Java code
  - A ***framework*** is one or more components with “holes” in them, i.e., some missing code
  - Programmer writes classes following particular conventions to fill in the missing code
  - Result of combining the framework code with the programmer’s code is a complete product

Example

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

public class NaturalNumberRootTest {

  @Test
  public void test1327Root3() {
    ...
  }


  ...


}
```

Example

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

public class Natura      erRootTest {

   @Test
   public voi
      ...
   }

   ...

}
```

These imports let you use JUnit features. The use of a ***static import*** allows you to call the static methods of `org.junit.Assert` without qualifying their names (see, e.g., `assertEquals` in upcoming code).

Example

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

public class NaturalNumberRootTest {

  @Test
  public void test1327Root3()
    ...
  }

  ...

}
```

A test plan or ***test fixture*** is a `public class` with one method per test case.

Example

```
@Test
public void test1327Root3() {
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(1327);
  NaturalNumber r =      NaturalNumber2(3);
  NaturalNumber rExpecte     new NaturalNumber2(3);
  NaturalNumber rt = Natura       rRoot.root(n, r);
  NaturalNumber rtExpected = n        ralNumber2(10);
  assertEquals(nExpected,
  assertEquals(rExpected,
  assertEquals(rtExpected
}
```

Each ***test case*** is a **`public void`** method with no parameters.

Example

```
@Test
public v id test1327Root3() {
  NaturalNum   r n = new NaturalNumber2(1327);
  NaturalNumber   xpected = new NaturalNumber2(1327);
  NaturalNumber r =     NaturalNumber2(3);
  NaturalNumber rExpect     new NaturalNumber2(3);
  NaturalNumber rt = Natur     rRoot.root(n, r);
  NaturalNumber rtExpected =     ralNumber2(10);
  assertEquals(nExpected,
  assertEquals(rExpected,
  assertEquals(rtExpected
}
```

Each test case has an `@Test` ***annotation*** just before it.

Example

There is an easy way to make a new test case: copy/paste another and then edit slightly.

```
@Test
public void test1327Root3(
  NaturalNumber n = new Na
  NaturalNumber nExpected =
  NaturalNumber r = new Natur      umber2(3);
  NaturalNumber rExpected = new NaturalNumber2(3);
  NaturalNumber rt = NaturalNumberRoot.root(n, r);
  NaturalNumber rtExpected = new NaturalNumber2(10);
  assertEquals(nExpected, n);
  assertEquals(rExpected, r);
  assertEquals(rtExpected, rt);
}
```

Vocabulary Review

- ***Test case***
  - Exercises a single unit of code, normally a method (and a test case normally makes one call to that method)
  - Test cases should be *small* (i.e., should test one thing)
  - Test cases should be *independent* of each other
  - In JUnit: a public method that is annotated with `@Test`
- ***Test fixture***
  - Exercises a single class
  - Is a collection of *test cases*
  - In JUnit: a class that contains `@Test` methods
- Note: In Eclipse, select "New > JUnit Test *Case*" to create a new JUnit test *fixture*!

New Vocabulary

- ***(JUnit) Assertion***
  - A claim that some boolean-valued expression is true; normally, a comparison between expected and actual results (i.e., the `equals` method says they are equal)
- ***Passing a test case***
  - All JUnit assertions in the test case are *true* when the test case is executed (and no error occurred to stop program execution)
- ***Failing a test case***
  - Some JUnit assertion in the test case is *false* when the test case is executed

Execution Model

- Separate instances (objects) are created from the JUnit test fixture
  - JUnit creates one instance per test case (!)
- Implication:
  - Do not rely on order of test cases
    - Test case listed first in JUnit test fixture is not guaranteed to be executed first

JUnit Assertions

- Two most useful static methods in `org.junit.Assert` to check actual results against allowed results:

```
assertEquals (expected, actual);
assertTrue(expression);
```

- There is rarely a reason to use any of the dozens of other assertion static methods in `org.junit.Assert`

Timed Tests

- What if you’re worried about an infinite loop?
  - Parameterize `@Test` with a ***timeout***: number of milliseconds before the test case is terminated for running too long

    ```
    @Test(timeout=100)
    ```

  - Problem: How do you know what is long enough for a test case to run?

Best Practices

- Some ***best practices***:
  - Keep JUnit test fixtures in the same Eclipse project as the code, but in a separate source folder (for this course: regular code in “src”, test classes/fixtures in “test”)
    - Tests are then included when project is “built”
    - Helps keep test fixtures consistent with other code

Best Practices

- Name test fixtures consistently
  - Example: class `NaturalNumberRootTest` tests class `NaturalNumberRoot`
- Name test cases consistently
  - Example: method `testFoo13` tests method `foo` with input 13

Recommended Test Case Style

```java
public void test1327Root3() {
  /*
   * Set up variables and call method under test
   */
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(1327);
  NaturalNumber r = new NaturalNumber2(3);
  NaturalNumber rExpected = new NaturalNumber2(3);
  NaturalNumber rt = NaturalNumberRoot.root(n, r);
  NaturalNumber rtExpected = new NaturalNumber2(10);
  /*
   * Assert that values of variables match expectations
   */
  assertEquals(nExpected, n);
  assertEquals(rExpected, r);
  assertEquals(rtExpected, rt);
}
```

Recommended Test Case Style

```
public void testDivideBy10NonZero() {
  /*
   * Set up variables and call method under test
   */
  NaturalNumber n = new NaturalNumber2(1327);
  NaturalNumber nExpected = new NaturalNumber2(132);
  int k = n.divideBy10();
  /*
   * Assert that values of variables match expectations
   */
  assertEquals(nExpected, n);
  assertEquals(7, k);
}
```

Sometimes, you can write the expected value directly.

Alternative Test Case Style

```
public void testDivideBy10NonZero() {
  /*
   * Set up variables and call method under test
   */
  NaturalNumber n = new NaturalNumber2(1327);
  int k = n.divideBy10();
  /*
   * Assert that values of variables match expectations
   */
  assertEquals("132", n.toString());
  assertEquals(7, k);
}
```

Use `toString`? May be OK, but `equals` is better.

Resources

- *JUnit in Action, Second Edition* (Petar Tahchiev, *et al*., 2010)
  - – https://library.ohio-state.edu/record=b8534108~S7