if (justMetYou) {
    crazy = true;
    cout << number << endl;
    int x = rand() % 100;
    if (x >= 50)
        callMe();
}

"Call Me Maybe"

private function bad() {
    break;
}

"Breaking Bad"

class StarWars(int episode) {
    if (episode == 6)
        return Jedi;
}

"Star Wars: Episode VI – Return of the Jedi"
Agenda

• Graphs
• Internal vs. external testing
• How to use JUnit
• How to use test script
Graphs
Graphs

Nodes

Graph representation with nodes A, B, C, D, and E connected with directed edges.
Graphs

Edges

A → B
A → C
A → D
A → E
B → A
B → D
C → A
C → B
C → D
D → A
D → B
D → C
D → E
E → A
E → B
E → C
Graphs

Children of A
Graphs

Parents of D
Graphs

Path from A to C
Graphs

Shortest path from A to C?
Graphs

Shortest path from A to B?
Internal vs. External Testing

• Internal : JUnit
  o How you decide to abstract the object
  o Checked with implementation tests

• External: test script
  o Client expects to see concrete object
  o Checked with specification tests
A JUnit Test Class

- A method with @Test is flagged as a JUnit test
- All @Test methods run when JUnit runs

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

public class TestSuite {
    ...

    @Test
    public void TestName1() {
        ...
    }
}
```
Using Assertions

- Verifies that a value matches expectations
  - `assertEquals(42, meaningOfLife());`
  - `assertTrue(list.isEmpty());`
- If the value isn’t what it should be, the test fails
  - Test immediately terminates
  - Other tests in the test class are still run as normal
  - Results show details of failed tests
# Using JUnit Assertions

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Case for failure</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assertTrue(test)</code></td>
<td>the boolean test is false</td>
</tr>
<tr>
<td><code>assertFalse(test)</code></td>
<td>the boolean test is true</td>
</tr>
<tr>
<td><code>assertEquals(expected, actual)</code></td>
<td>the values are not equal</td>
</tr>
<tr>
<td><code>assertSame(expected, actual)</code></td>
<td>the values are not the same (by ==)</td>
</tr>
<tr>
<td><code>assertNotSame(expected, actual)</code></td>
<td>the values are the same (by ==)</td>
</tr>
<tr>
<td><code>assertNull(value)</code></td>
<td>the given value is not null</td>
</tr>
<tr>
<td><code>assertNotNull(value)</code></td>
<td>the given value is null</td>
</tr>
</tbody>
</table>

- And others: [http://www.junit.org/apidocs/org/junit/Assert.html](http://www.junit.org/apidocs/org/junit/Assert.html)
- Each method can also be passed a string to display if it fails:
  - `assertEquals("message", expected, actual)"`
Checking for Exceptions

• Verify that a method throws an exception when it should.
• Test passes if specified exception is thrown, fails otherwise.
• Only time it’s OK to write a test without a form of asserts.

```java
@Test(expected=IndexOutOfBoundsException.class)
public void testGetEmptyList() {
    List<String> list = new ArrayList<String>();
    list.get(0);
}
```
Setup and Teardown

• Methods to run before/after each test case method is called:

```java
@Before
public void name() { ... }
@After
public void name() { ... }
```

• Methods to run once before/after the entire test class runs:

```java
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```
public class Example {
    List empty;

    @Before
    public void initialize() {
        empty = new ArrayList();
    }

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

    @Test
    public void remove() {
        ...
    }
}
Don’t Repeat Yourself

• Can declare fields for frequently-used values or constants
  o private static final String DEFAULT_NAME = “MickeyMouse”;
  o private static final User DEFAULT_USER = new User(“lazowska”, “Ed”, “Lazowska”);

• Can write helper methods, etc.
  o private void eq(RatNum ratNum, String rep) {
      assertEquals(rep, ratNum.toString());
  }
  o private BinaryTree getTree(int[] items) {
      // construct BinaryTree and add each element in items
  }
#1: Be descriptive

- When a test fails, JUnit tells you:
  - Name of test method
  - Message passed into failed assertion
  - Expected and actual values of failed assertion
- The more descriptive this information is, the easier it is to diagnose failures

<table>
<thead>
<tr>
<th>Level of goodness</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good</td>
<td>testAddDaysWithinMonth()</td>
</tr>
<tr>
<td>Not so good</td>
<td>testAddDays1(), testAddDays2()</td>
</tr>
<tr>
<td>Bad</td>
<td>test1(), test2()</td>
</tr>
<tr>
<td>Overkill</td>
<td>TestAddDaysOneDayAndThenFiveDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFebruary()</td>
</tr>
</tbody>
</table>
#1: Be descriptive

- Take advantage of message, expected, and actual values
- No need to repeat expected/actual values or info in test name
- Use the right assert for the occasion:
  - `assertEquals(expected, actual)` instead of `assertTrue(expected.equals(actual))`
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
public class DateTest {

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }

public class DateTest {

    ... 

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
}
public class DateTest {

    ...

    // Test addDays when it causes a rollover between months
    @Test
    public void testAddDaysWrapToNextMonth() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }

    ...

    Use assertion to check expected results
public class DateTest {

...

// Test addDays when it causes a rollover between months

@Test

public void testAddDaysWrapToNextMonth() {
    Date actual = new Date(2050, 2, 15);
    actual.addDays(14);
    Date expected = new Date(2050, 3, 1);
    assertEquals("date after +14 days", expected, actual);
}

Message gives details about the test in case of failure
#2: Keep tests small

• Ideally, test one thing at a time
  o “Thing” usually means one method under one input condition
  o Not always possible – but if you test $x()$ using $y()$, try to test $y()$ in isolation in another test

• Low-granularity tests help you isolate bugs
  o Tell you exactly what failed and what didn’t

• Only a few (likely one) assert statements per test
  o Test halts after first failed assertion
  o Don’t know whether later assertions would have failed
#3: Be thorough

- Consider each equivalence class
  - Items in a collection: none, one, many
- Consider common input categories
  - `Math.abs()`: negative, zero, positive values
- Consider boundary cases
  - Inputs on the boundary between equivalence classes
    - `Person.isMinor()`: age < 18, age == 18, age > 18
- Consider edge cases
  - -1, 0, 1, empty list, `arr.length`, `arr.length-1`
- Consider error cases
  - Empty list, null object
Other Guidelines

• Test all methods
  o Constructors are exception to the rule

• Keep tests simple
  o Minimize if/else, loops, switch, etc.
  o Don’t want to debug your tests!

• Tests should always have at least one assert
  o Unless testing that an exception is thrown
  o Testing that an exception is not thrown is unnecessary
  o `assertTrue(true)` doesn’t count!

• Tests should be isolated
  o Not dependent on side effects of other tests
  o Should be able to run in any order
JUnit Summary

• Tests need *failure atomicity* so we know exactly what failed
  o Each test should have a descriptive name
  o Assertions should have clear messages to know what failed
  o Write many small tests, not one big test
• Test for expected errors / exceptions
• Choose a descriptive assert method, not always `assertTrue`
• Choose representative test cases from equivalent input classes
• Avoid complex logic in test methods if possible
• Use helpers, `@Before` to reduce redundancy between tests
Java Asserts vs. JUnit Asserts

• We’ve just been discussing JUnit assertions so far
• Java itself has assertions
  o However, in order to use them, you need to enable a special flag in the Java Virtual Machine (JVM)

```java
public class LitterBox {
    ArrayList<Kitten> kittens;

    public Kitten getKitten(int n) {
        assert (n >= 0);
        return kittens(n);
    }
}
```
Enabling Java Asserts

- Right click the .java file you are running
- Go to “Run As” → “Run Configurations”
- Click on the “Arguments” tab
- Enter "-ea" under “VM arguments”
Assertions vs. Exceptions

- Assertions should check for things that should **never** happen
- Exceptions should check for things that **might** happen
- “Exceptions address the robustness of your code, while assertions address its correctness”
Test Script Language

- Text file with one command listed per line
- First word is always the command name
- Remaining words are arguments
- Commands will correspond to methods in your code
# Create a graph
CreateGraph graph1

# Add a pair of nodes
AddNode graph1 n1
AddNode graph1 n2

# Add an edge
AddEdge graph1 n1 n2 e1

# Print the nodes in the graph and the outgoing edges from n1
ListNodes graph1
ListChildren graph1 n1
Test Script Language

CreateGraph A
AddNode A n1
AddNode A n2

CreateGraph B
ListNodes B
AddNode A n3
AddEdge A n3 n1 e31
AddNode B n1
AddNode B n2
AddEdge B n2 n1 e21
AddEdge A n1 n3 e13
AddEdge A n1 n2 e12

ListNodes A
ListChildren A n1
ListChildren B n2