```
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”

Section 4:
Graphs and Testing

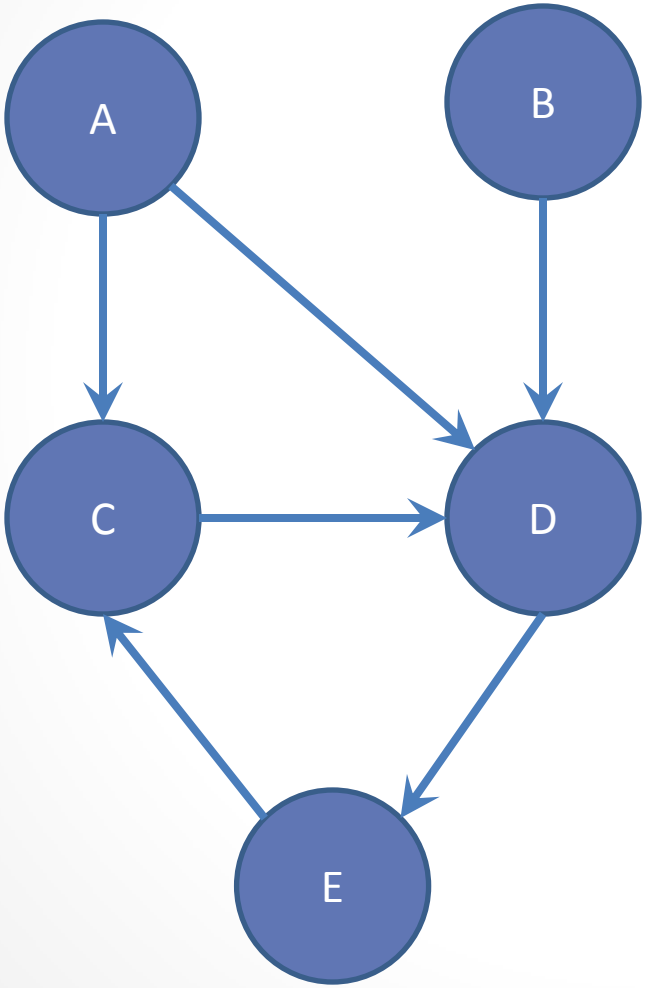
Slides by Alex Mariakakis

with material from Krysta Yousoufian, Mike Ernst, Kellen Donohue

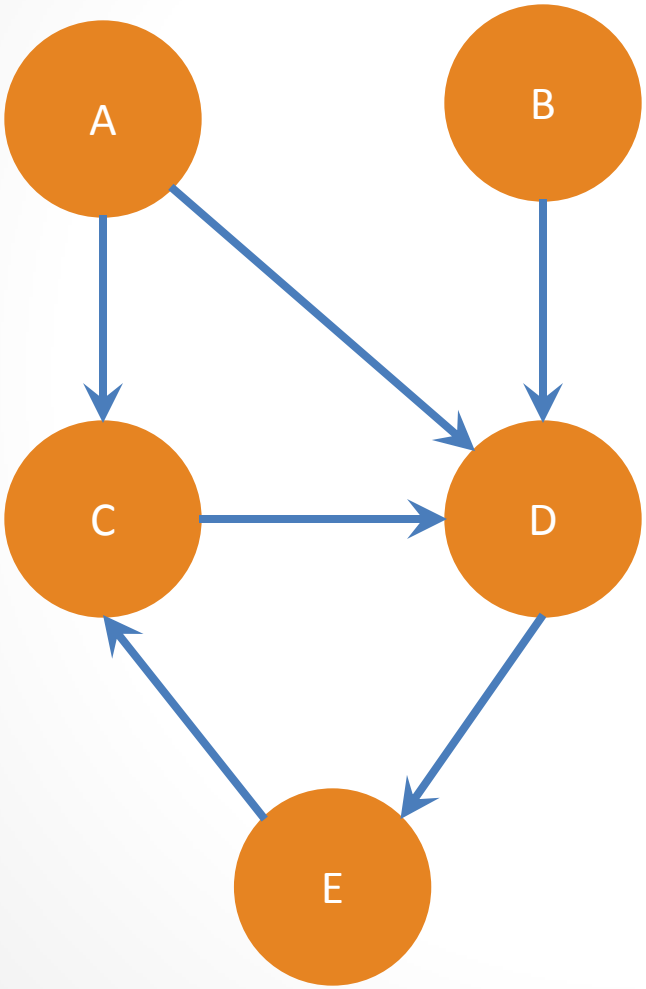
Agenda

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

Graphs

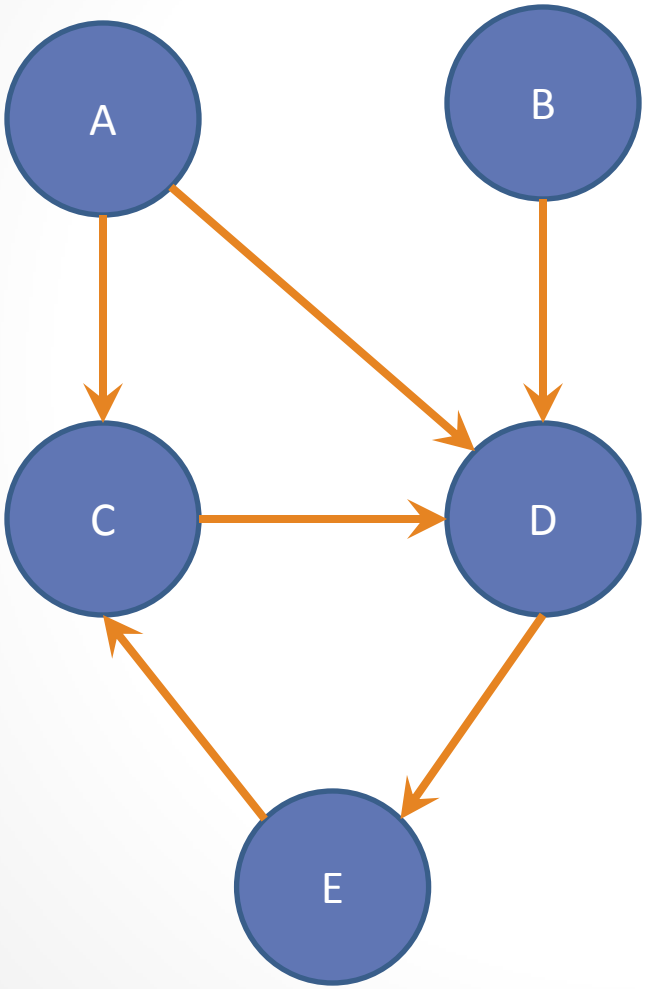


Graphs



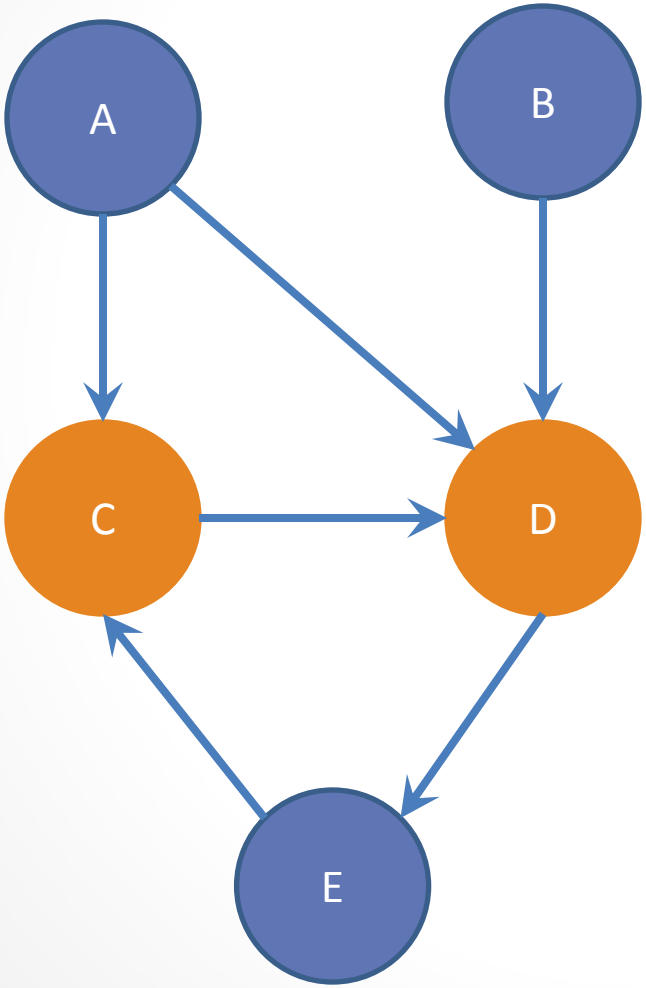
Nodes

Graphs



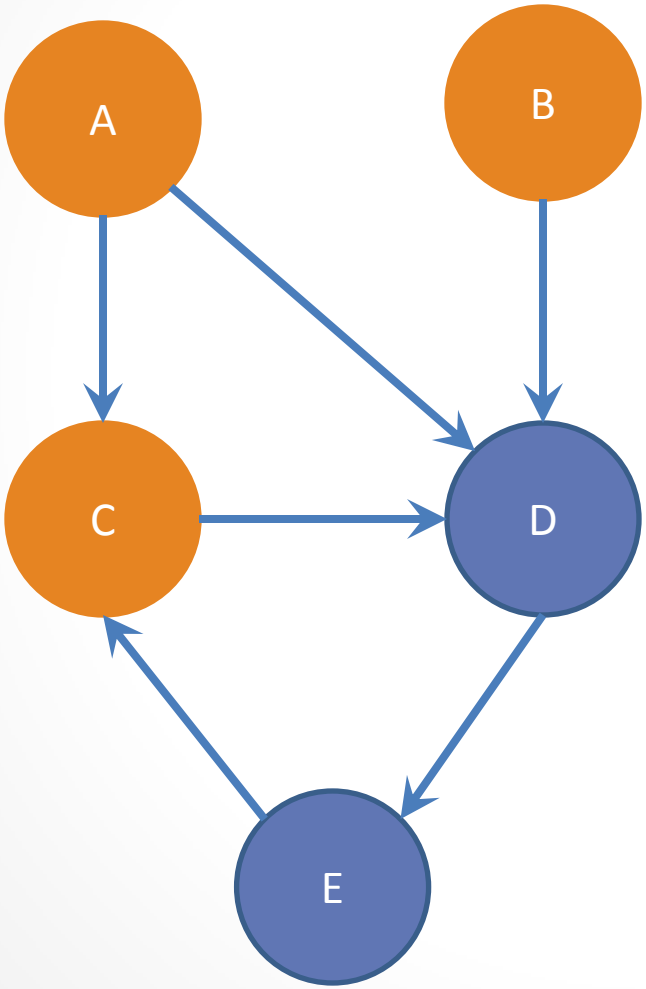
Edges

Graphs



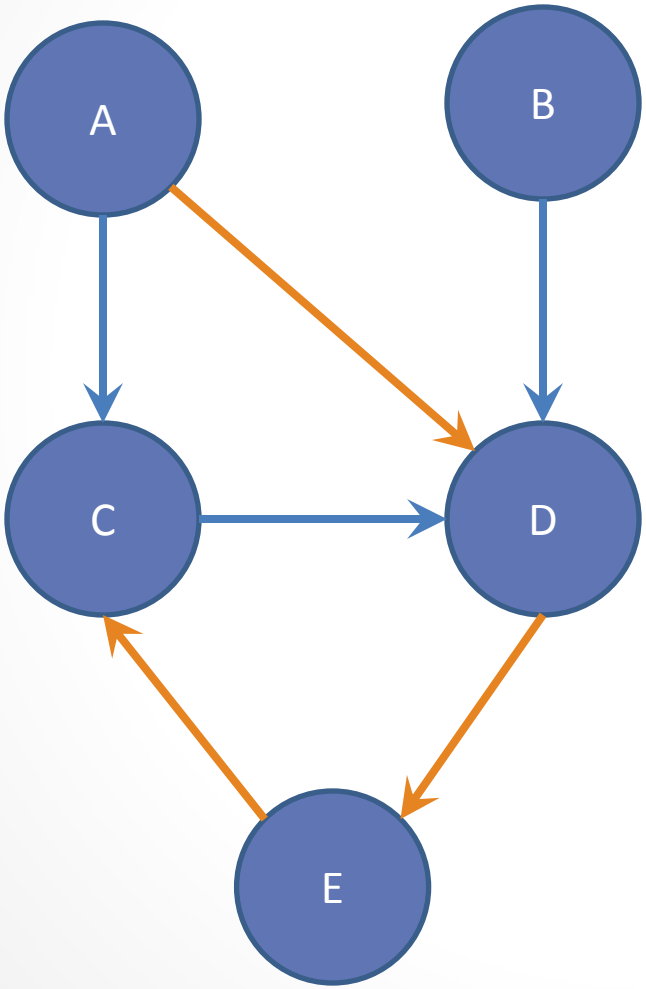
Children of A

Graphs



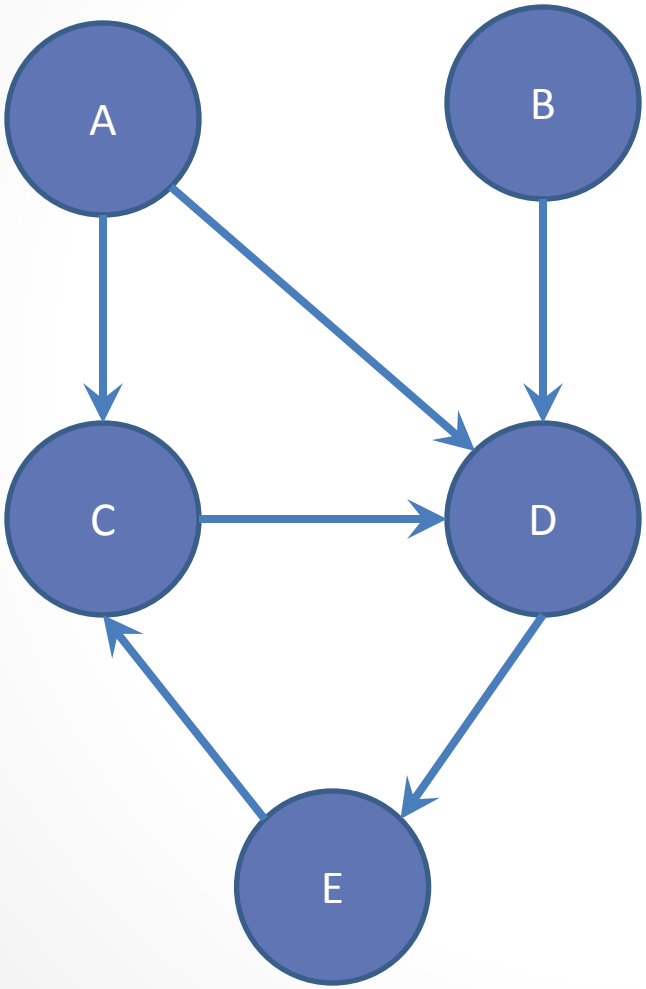
Parents of D

Graphs



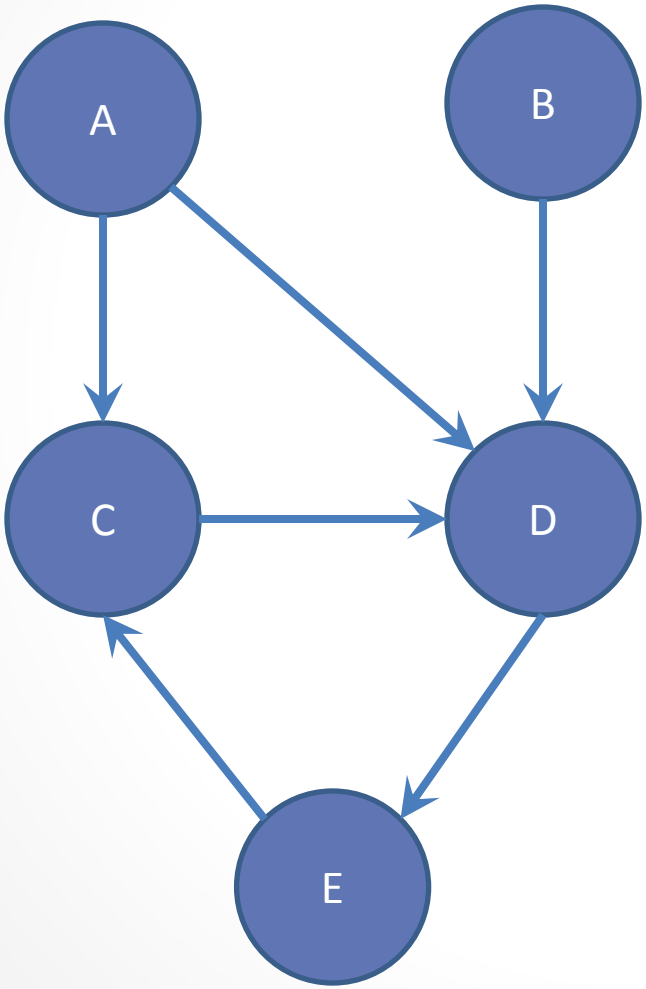
Path from A to C

Graphs



Shortest path from A to C?

Graphs



Shortest path from A to B?

Internal vs. External Testing

- Internal : JUnit
  - How you decide to abstract the object
  - Checked with implementation tests
- External: test script
  - Client expects to see concrete object
  - 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

```
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

Assertion	Case for failure
`assertTrue(test)`	the boolean test is false
`assertFalse(test)`	the boolean test is true
`assertEquals(expected, actual)`	the values are not equal
`assertSame(expected, actual)`	the values are not the same (by ==)
`assertNotSame(expected, actual)`	the values are the same (by ==)
`assertNull(value)`	the given value is not null
`assertNotNull(value)`	the given value is null

- And others: 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`

```
@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:

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

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

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

Setup and Teardown

```
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
  - `private static final String DEFAULT_NAME = "MickeyMouse";`
  - `private static final User DEFAULT_USER = new User("lazowska", "Ed", "Lazowska");`
- Can write helper methods, etc.
  - ```
    private void eq(RatNum ratNum, String rep) {
        assertEquals(rep, ratNum.toString());
    }
    ```
  - ```
    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

Level of goodness	Example
Good	`testAddDaysWithinMonth()`
Not so good	`testAddDays1(), testAddDays2()`
Bad	`test1(), test2()`
Overkill	`TestAddDaysOneDayAndThenFiveDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFebruary()`

#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))`

Let's put it all together!

```
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);
    }
```

Let's put it all together!

```
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);
    }
```

Tells JUnit that this method is a test to run

Let's put it all together!

```
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);
    }
```

Descriptive method name

Let's put it all together!

```
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

Let's put it all together!

```
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
  - "Thing" usually means one method under one input condition
  - 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
  - Tell you exactly what failed and what didn't
- Only a few (likely one) assert statements per test
  - Test halts after first failed assertion
  - 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
  - Constructors are exception to the rule
- Keep tests simple
  - Minimize `if/else`, `loops`, `switch`, etc.
  - Don't want to debug your tests!
- Tests should always have at least one assert
  - Unless testing that an exception is thrown
  - Testing that an exception is not thrown is unnecessary
  - `assertTrue(true)` doesn't count!
- Tests should be isolated
  - Not dependent on side effects of other tests
  - Should be able to run in any order

JUnit Summary

- Tests need *failure atomicity* so we know exactly what failed
  - Each test should have a descriptive name
  - Assertions should have clear messages to know what failed
  - 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

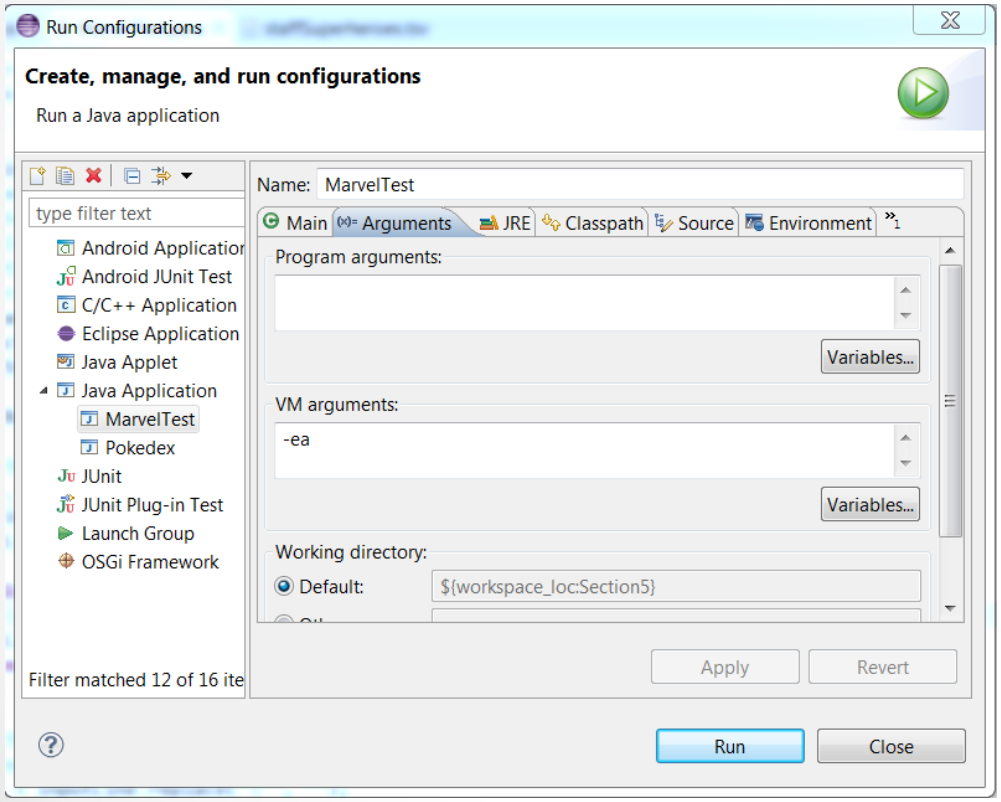
Java Asserts vs. JUnit Asserts

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

```
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

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

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

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

   public Kitten getKitten(int n) {
      try {
         return kittens(n);
      } catch(Exception e) {
      }
   }
}
```

- 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

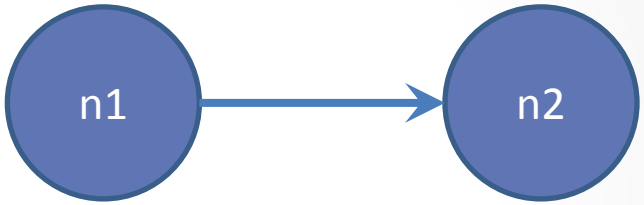
Test Script Language

```
# 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
```