The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.

Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."

Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"

Anger

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.

Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.

Acceptance

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.

Section 4:
HW5, Graphs, and Testing

Slides by Vinod Rathnam

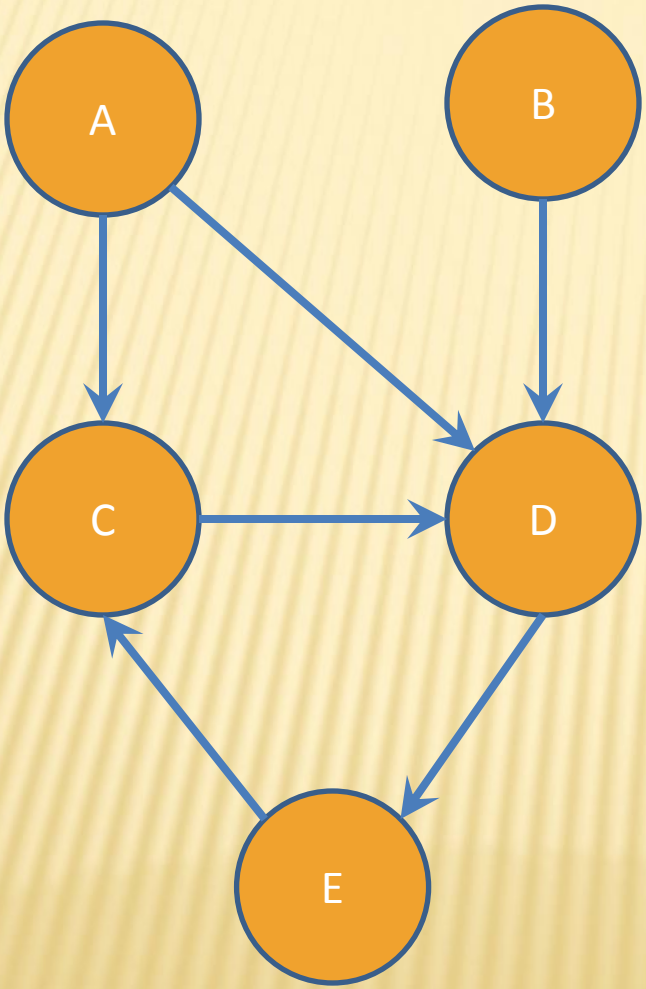
with material from Alex Mariakakis, Krysta Yousoufian, Mike Ernst, Kellen Donohue

AGENDA

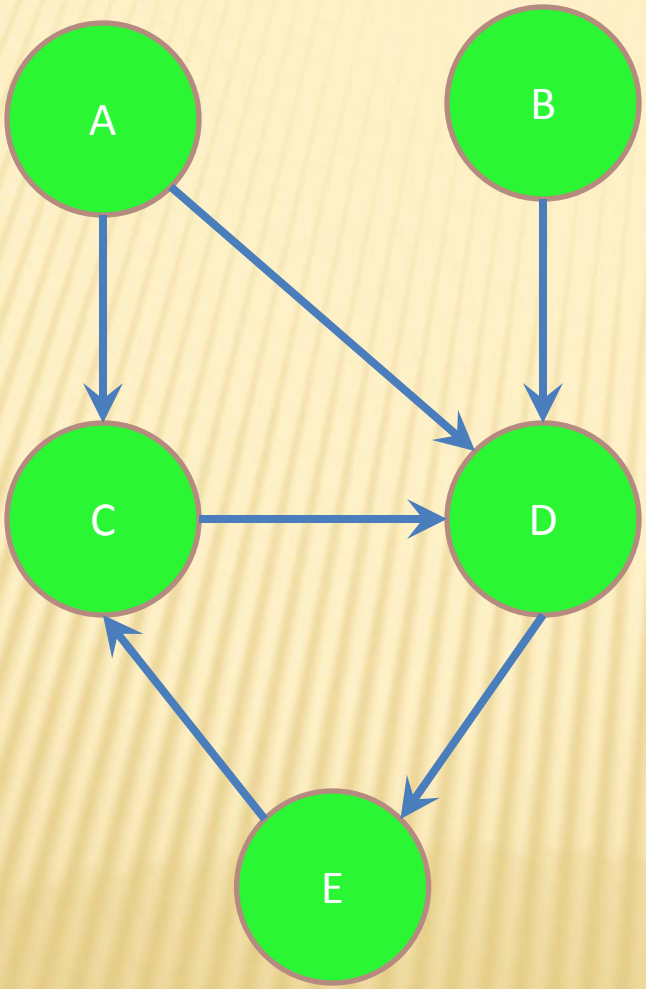
- HW5
- Graphs
- JUnit Testing
- Test Script Language (Demo)
- JavaDoc (Demo)

DEMO: HW 5 STARTER FILES

GRAPHS

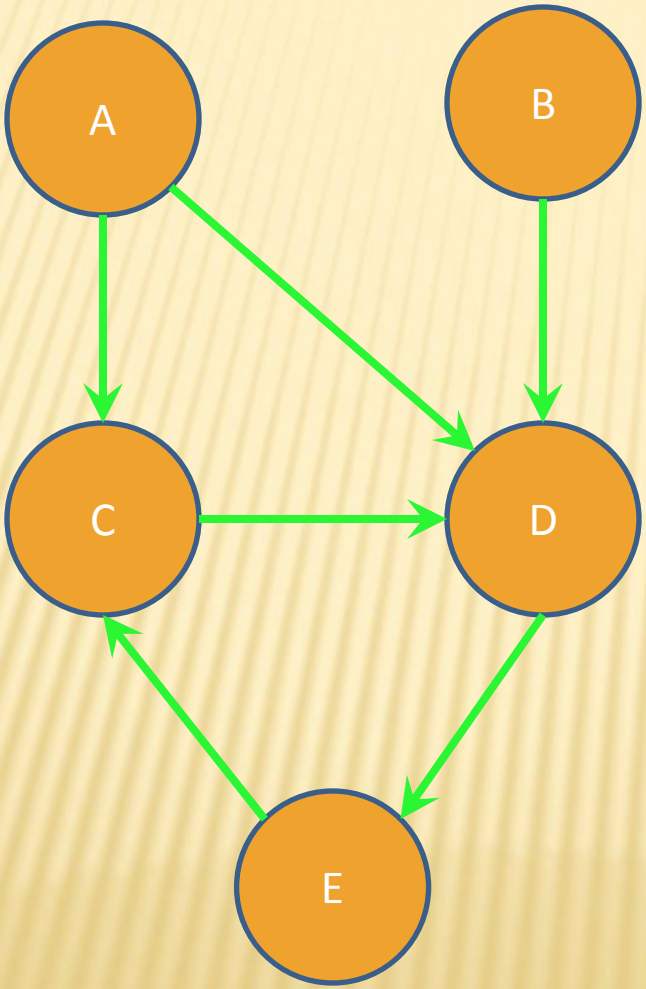


GRAPHS



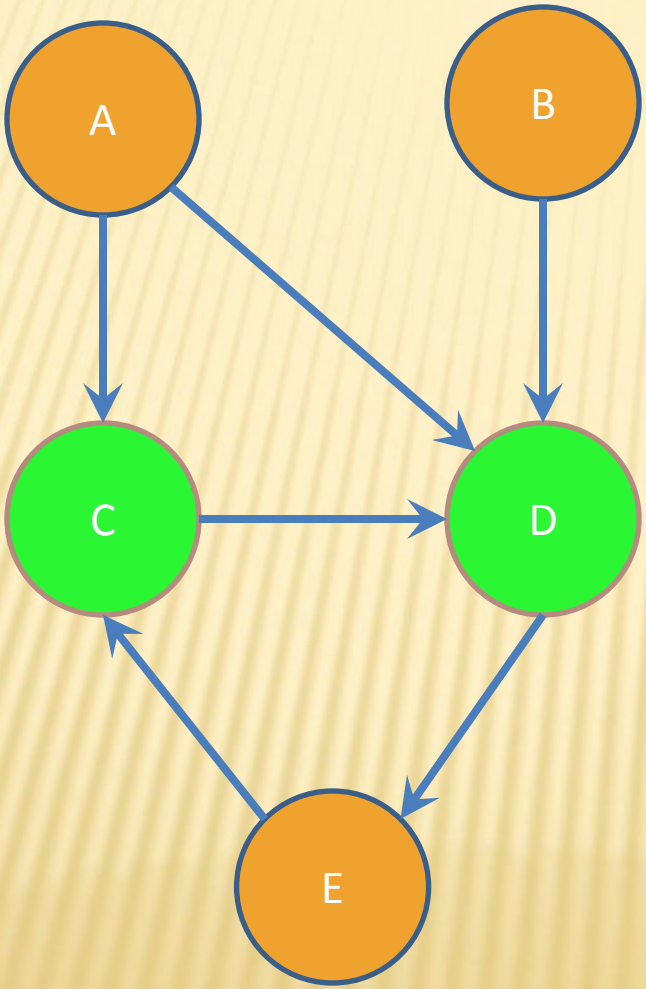
Nodes

GRAPHS



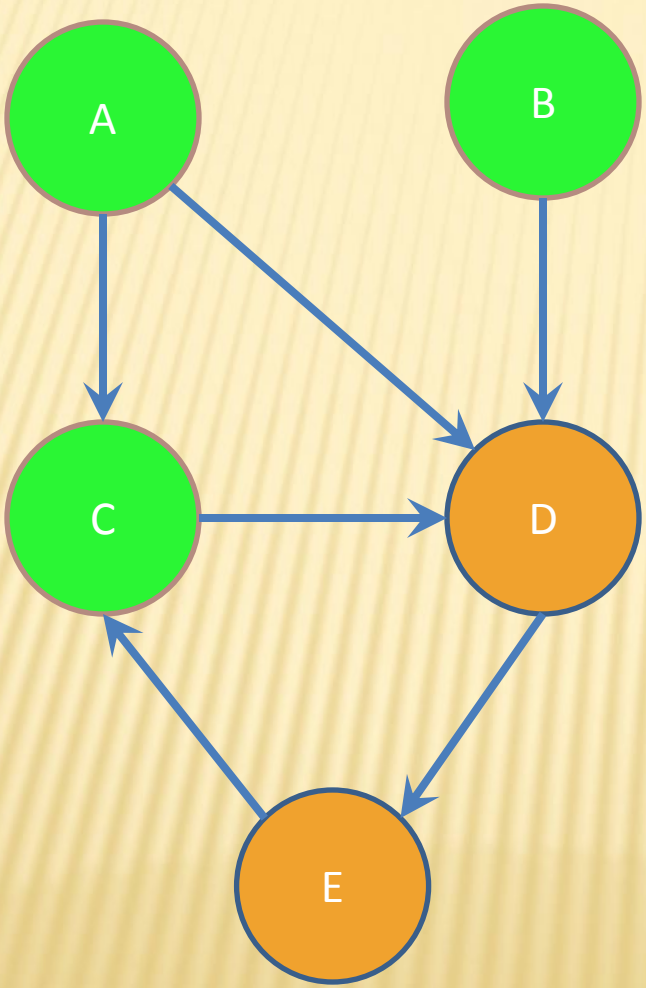
Edges

GRAPHS



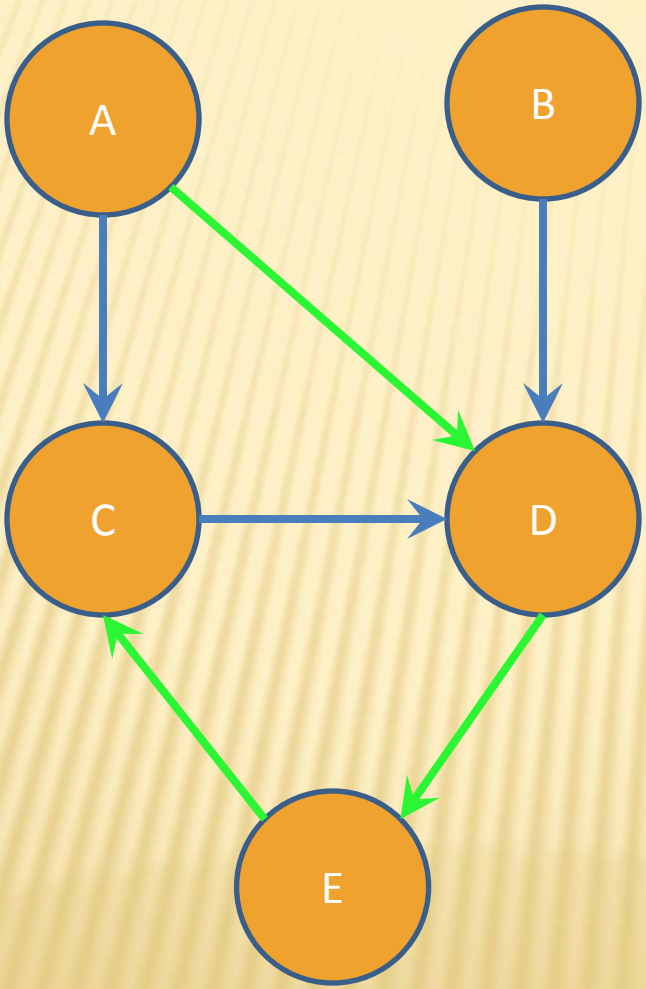
Children of A

GRAPHS



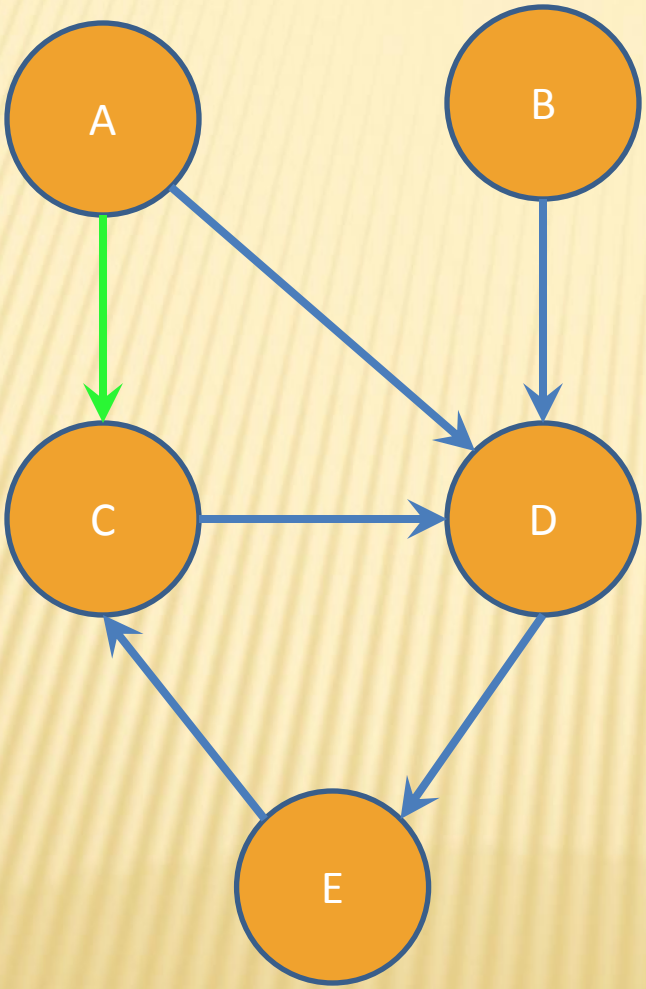
Parents of D

GRAPHS



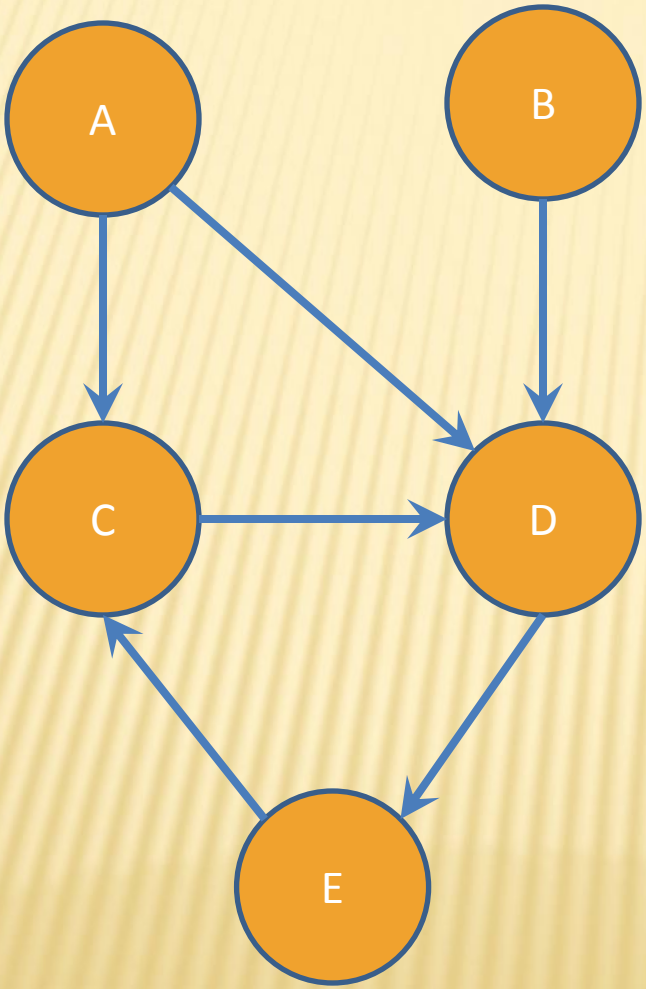
Path from A to C

GRAPHS



Shortest path from A to C?

GRAPHS



Shortest path from A to B?

INTERNAL VS. EXTERNAL TESTING

- Internal : JUnit
  - How you decide to implement the object
  - Checked with implementation tests
- External: test script
  - Your API and specifications
  - Testing against the specification
  - 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 JUNIT 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

JUNIT ASSERTS VS. JAVA ASSERTS

- We've just been discussing JUnit assertions so far
- Java itself has assertions

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

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

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”

EXTERNAL TESTS: TEST SCRIPT LANGUAGE

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

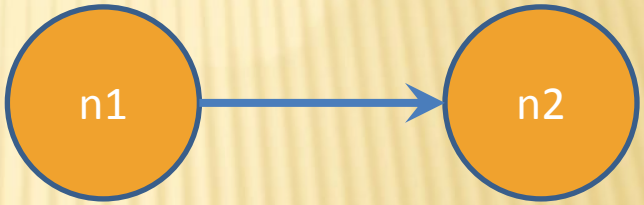
TEST SCRIPT LANGUAGE

```
# 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
```

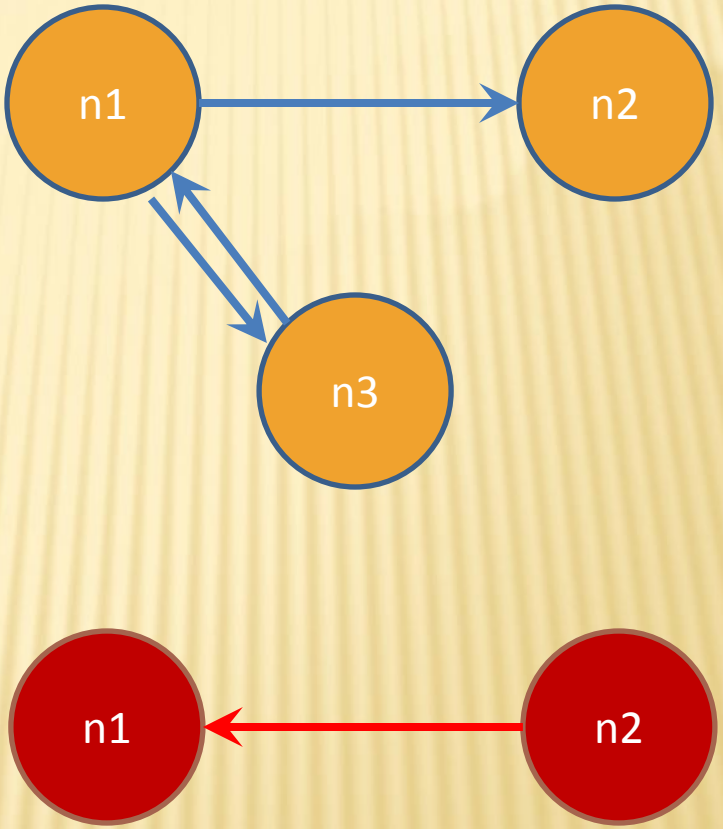


TEST SCRIPT LANGUAGE

```
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
```



DEMO: TEST SCRIPT LANGUAGE

JAVADOC API

- Now you can generate the JavaDoc API for your code
- Instructions online: http://courses.cs.washington.edu/courses/cse331/15wi/tools/editing-compiling.html#javadoc
- Demo: Generate JavaDocs