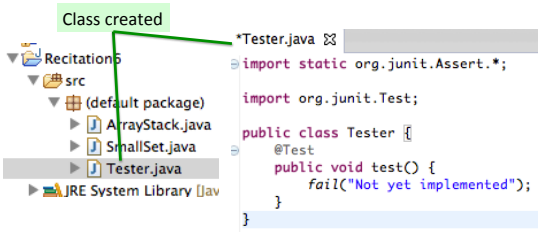
JUnit Testing / LinkedList

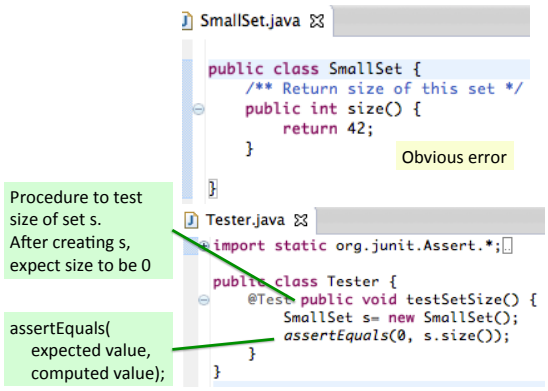
- Isaac Sheff and David Gries

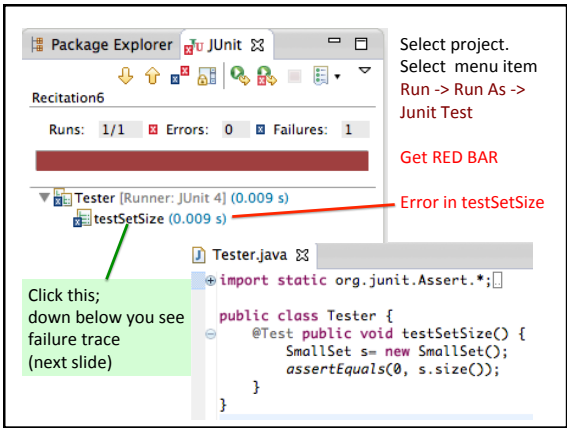
- Systematically and automatically test each piece —or unit— of your program with example inputs.
- A unit is generally a method —constructor, function, or procedure
- Save the tests of a unit and have them automatically executed whenever you test. Testing already tested methods is a good idea when a program is being written and changed.
- Build up to larger tests

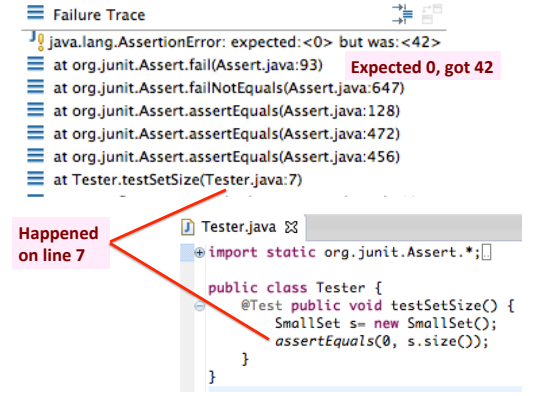
Making a Junit Testing Class

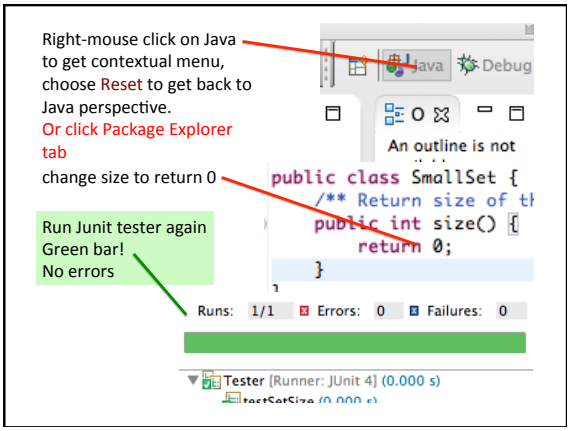
1. Select project or its src in Package Explorer
2. Select menu item File -> New -> Junit Test Case
3. Give class the name Tester and click Finish
4. If asked to add Junit 4 library to build path, say OK











Some JUnit asserts

- assertEquals(expected value, computed value)
- assertTrue(boolean exp that should be true)
- assertFalse(boolean exp that should be false)
- assertNull()
- assertNotNull()
- fail(message) --aborts test of procedure, printing message
- There are a lot more at http://junit.sourceforge.net/javadoc/

Can have several procedures

Each no-parameter procedure preceded by @Test is tested independently when running Junit test

DrJava does it slightly differently!

```
public class Tester {
    @Test public void testSetSize() {
        SmallSet s= new SmallSet();
        assertEquals(0, s.size());
        s.add("bc");
        assertEquals(0, s.size());
    }

    @Test public void testSetAdd() {
        // ...
    }

    @Test public void SetContains() {
        // ...
    }
```

We inserted procedure add in SmallSet. We test whether it increases set size

First test passed, second didn't

```
public class Tester {
    @Test public void testSetSize() {
        SmallSet s= new SmallSet();
        assertEquals(0, s.size());
    }

    @Test public void testSetAdd() {
        SmallSet s= new SmallSet();
        s.add("abc");
        assertEquals(1, s.size());
    }
```

Tester [Runner: JUnit 4] (
testSetSize (0.000 s)
testSetAdd (0.002 s)

Use to filter trace

Failure Trace
java.lang.AssertionError: expected:<1> but was:<0>
at Tester.testSetAdd(Tester.java:13)

```
/** max size of set */
public static final int MAX= 10;
private int n= 0; // size of set
// set elements in items[0..n-1]
private Object items[]= new Object[MAX];

/** Return size of this set */
public int size() {
    return n;
}

/** Add ob to set (if not there) */
public void add(Object ob) {
    items[n]= ob;
    n= n+1;
}
```

Look at add to see what's wrong

Statement missing

Run Junit test again and it works

Adding an element twice shouldn't increase set size

But it does!

```
@Test public void testSetAdd() {
    SmallSet s= new SmallSet();
    s.add("abc");
    assertEquals(1, s.size());
    s.add("abc");
    assertEquals(1, s.size());
}
```

Tester [Runner: JUnit 4] (0
testSetSize (0.000 s)
testSetAdd (0.000 s)

Failure Trace
java.lang.AssertionError: expected:<1> but was:<2>
at Tester.testSetAdd(Tester.java:15)

Fix add in SmallSet so that it does not add ob if it is already there.

```
/** Add ob to set (if not there) */
public void add(Object ob) {
    for (int k= 0; k < n; k++) {
        if (items[k].equals(ob))
            return;
    }
    items[n]= ob;
    n= n+1;
}
```

It passes the test!

Runs: 2/2 Errors: 0 Failures: 0

Tester [Runner: JUnit 4] (0.000 s)

Don't forget to change specification!

```
/** Add ob to set (if not there)
  * and return true iff it was added */
public boolean add(Object ob) {
    for (int k= 0; k < n; k++) {
        if (items[k].equals(ob))
            return false;
    }
    items[n]= ob;
    n= n+1;
    return true;
}
```

Refactor add: return whether or not it was added

It still works

Runs: 2/2 Errors: 0 Failures: 0

Tester [Runner: JUnit 4] (0.000 s)

Test Boundary Conditions: What if too many added?

```
@Test public void testAddTooMany() {
    SmallSet s= new SmallSet();
    for (int k= 0; k <= SmallSet.MAX; k++) {
        s.add(k);
    }
}
```

Test Fails with an Exception But is that correct?

Tester [Runner: JUnit 4] (0.000 s)
testSetSize (0.000 s)
testSetAdd (0.000 s)
testAddTooMany (0.000 s)

Failure Trace

java.lang.ArrayIndexOutOfBoundsException: 10
at SmallSet.add(SmallSet.java:21)
at Tester.testAddTooMany(Tester.java:21)

Make a new kind of Exception specific to this situation.

```
public class SmallSetException
        extends RuntimeException {

    public SmallSetException() {
        super();
    }

    public SmallSetException(String m) {
        super(m);
    }

}
```

Don't forget to change spec!

```
/** Add ob to set (if not there)
  * and return true iff it was added.
  * Throw SmallSetException if no room */
public boolean add(Object ob) {
    for (int k= 0; k < n; k++) {
        if (items[k].equals(ob))
            return false;
    }
    if (n == MAX)
        throw(new SmallSetException
                ("No room"));
    items[n]= ob;
    n= n+1;
    return true;
}
```

Fix SmallSet.add to throw exception

How to test for an exception

```
@Test public void testAddTooMany() {
    SmallSet s= new SmallSet();
    for (int k= 0; k < SmallSet.MAX; k++) {
        s.add(k);
    }
    try {
        s.add(SmallSet.MAX);
        fail("SmallSetException expected");
    } catch (SmallSetException e) {
    }
}
```

We expect this to throw an exception

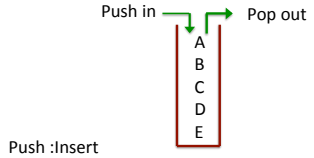
Execution of fail stops testing with given argument printed

Catch block catches exception and does nothing with it

This test passes, giving a green bar

Linked list implementation of a stack

Stack: a LIFO (Last-in-first-out) list



Push :Insert at beginning

head → A → B → C → D → E

Pop: Delete first value

Linked list implementation of a stack

```
/** Stack cell (a helper class; visible to other classes
      in same package). */
class StackCell<T> {
  public T datum; // Data for this cell
  public StackCell<T> next; // Next cell in this Stack

  /** Constructor: an instance with datum d and next cell next. */
  public StackCell(T d, StackCell<T> next) {
    datum= d;
    this.next= next;
  }
}
```

Linked list implementation of a stack

```
public class LinkedListStack<T> {
  // Head (first cell) of the List --null if list empty.
  // First element on the list is the top
  // Last element on the list is the bottom.
  private StackCell<T> head;

  /**Constructor: An empty stack. */
  public LinkedListStack() {
    head= null;
  }
```

Push and Pop

```
/** Push e on this stack. */
  public  void push(T e) {
    head= new StackCell<T>(e, head);
  }

  /** Pop top element of this stack and return it.
    * Throw RuntimeException if the stack is empty */
  public  T pop(T e) {
    if (head == null)
      throw new
          RuntimeException("Can't pop empty stack");
    T   h= head.datum;
    head= head.next;
    return h;
  }
```

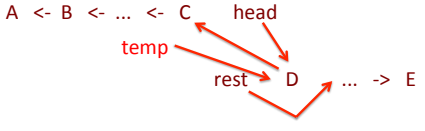
```
/** Reverse list in place. */
public void ReverseInPlace() {
  // Initial list: head -> A -> B -> ... -> C -> D -> ... -> E
  // Final   list:      A <- B <- ... <- C <- D <- ... <- E  <-  head

  StackCell<T> rest= head;
  head= null;
  // invariant:       A <- B <- ... <- C <-  head
  //                                  rest ->   D -> ... -> E
  // Each iteration pops first node of rest, puts it on head
  while (rest != null) {
    StackCell<T> temp= rest;
    rest= rest.next;
    temp.next= head;
    head= temp;
  }
}
```

Invariant: shows what head and rest look like just before each test of loop condition

Watch execution of repetend

```
// invariant:       A <- B <- ... <- C <-  head
//                                  rest ->   D -> ... -> E
// Each iteration pops first node of rest, puts it on head
while (rest != null) {
    StackCell<T> temp= rest;     Do this next
    rest= rest.next;             Do this next
     temp.next= head;            Do this next
    head= temp;                  Do this next
}
```



Recursive reverse

```
/** Push e on this stack. */
  /** Reverse this list in place. */
  public void ReverseInPlaceRecursive() {
    if (head == null) return;
    StackCell<T> c2= head.next;
    head.next= null;
    prependInPlace(head, c2);
  }
  /** Prepend the elements of list c2 to list c1.
    Precondition: c1 is not null. */
  public void prependInPlace(StackCell<T> c1, StackCell<T> c2)
```

Just prepend (put at beginning) to the list containing the first element all the other elements, one at a time!

A <- c1
c2 -> B -> C -> D -> ... -> E

Prepend elements of c2 to c1