EE461L
Software Engineering Lab

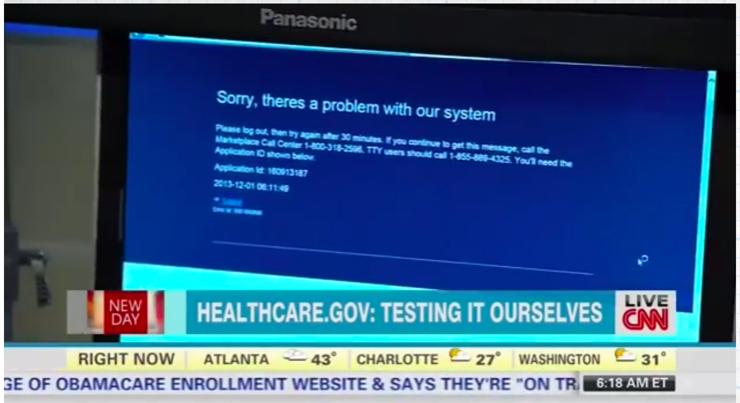
"To err is human, but to really foul things up you need a computer."
-- Paul Ehrlich

"I guess you could call it a "failure", but I prefer the term "learning experience"."
--The Martian

Testing

Resources:

https://courses.cs.washington.edu/courses/cse331/11sp/lectures/slides/09-junit.ppt



Sorry, theres a problem with our system

Please log out, then try again after 30 minutes. If you continue to get this message, call the Marketplace Call Center 1-800-318-2596. TTY users should call 1-855-889-4325. You'll need the Application ID shown below.

Application Id: 160913187

2013-12-01 06:11:49

Panasonic
NEW DAY
HEALTHCARE.GOV: TESTING IT OURSELVES
LIVE CNN
RIGHT NOW
ATLANTA 43°
CHARLOTTE 27°
WASHINGTON 31°
GE OF OBAMACARE ENROLLMENT WEBSITE & SAYS THEY'RE "ON TR
6:18 AM ET

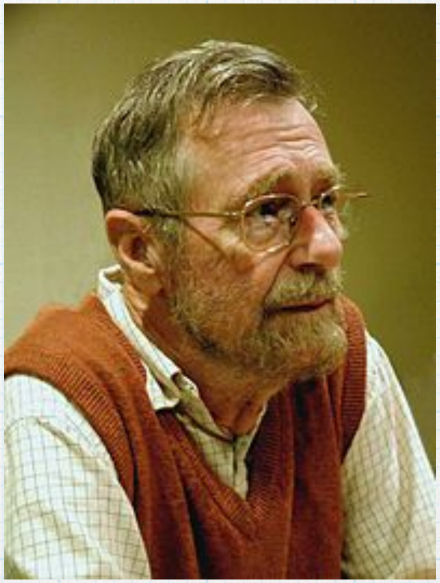
Testing

- It's a fundamental and absolute necessity
- Historical Examples
  - Mars Climate Orbiter – metric/Imperial units for force
  - Therac-25 – concurrent programming errors (mishandling of a race condition) led to massive radiation overdoses
  - ARIANE 5 – floating point to integer overflow
  - Healthcare.gov – failed to perform integration testing
- Anecdotal data
  - One to five errors per KLOC in mature software
  - More than 10 bugs for KLOC in prototype
  - Largely independent of programming language

What can you learn from testing?

"Program testing can be used to show the presence of bugs, but never to show their absence!"

--Edsgar Dijkstra, *Notes on Structured Programming,* 1970



Nevertheless testing is essential. Why?

Difficulties of Testing

- Perception by some developers and managers:
  - Testing is a novice’s job
  - Assigned to the least experienced team members
  - Done as an afterthought (if at all)
    - “My code is good; it won’t have bugs. I don’t need to test it.”
    - “I’ll just find the bugs by running the client program”
- Limitations of what testing can show
  - It’s impossible to completely test a system
  - Testing does not always directly reveal the actual bugs
  - Testing does not prove the absence of errors

Approaches to Verification

- Testing: exercise program to try and generate failures
  - Purpose: reveal failures by running program on test cases
    - generated by hand or randomly
  - Limits: small subset of use-cases
  - "dynamic verification"
- Static verification: identify (specific) problems by looking at source code; considering all execution paths
  - Positive: no test writing (which is hugely time consuming)
  - Negatives: very limited capacity of automated techniques; difficult to formalize properties; false positives

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
-Donald Knuth, 1977

Approaches to Verification

- Code inspection/review/walkthrough: manual review of program text to detect faults
  - Limits: informal, uneven, expensive (?)
- Formal proof: prove, starting from program source, that program text implements the program specification
  - Limits: limited automation, requires PhD, extremely time consuming

Testing

- A practice supported by a wealth of industrial and academic research and by commercial experience
- Testing and code review/inspection are the most common quality assurance methods
- Testing is a means of detecting/revealing errors
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected

Testing Details

- Testing: execute code (program/library/class) with a sample of the input data
  - Dynamic: program must be executed
  - Optimistic: exercise a (exponentially small) fraction of all possible input data
    - Other inputs are consistent with the behavior exhibited on the subset that is used

Levels of Testing

- **Unit testing**: the execution of a complete class, routine, or small program
- **Component testing**: the execution of a class, package, small program, or other program element
- **Integration testing**: the combined execution of two or more classes, packages, components, or subsystems
- **System testing**: the execution of the software in its final configuration, including integration with other software and hardware systems
- **Regression testing**: the repetition of previously executed test cases for the purpose of finding defects

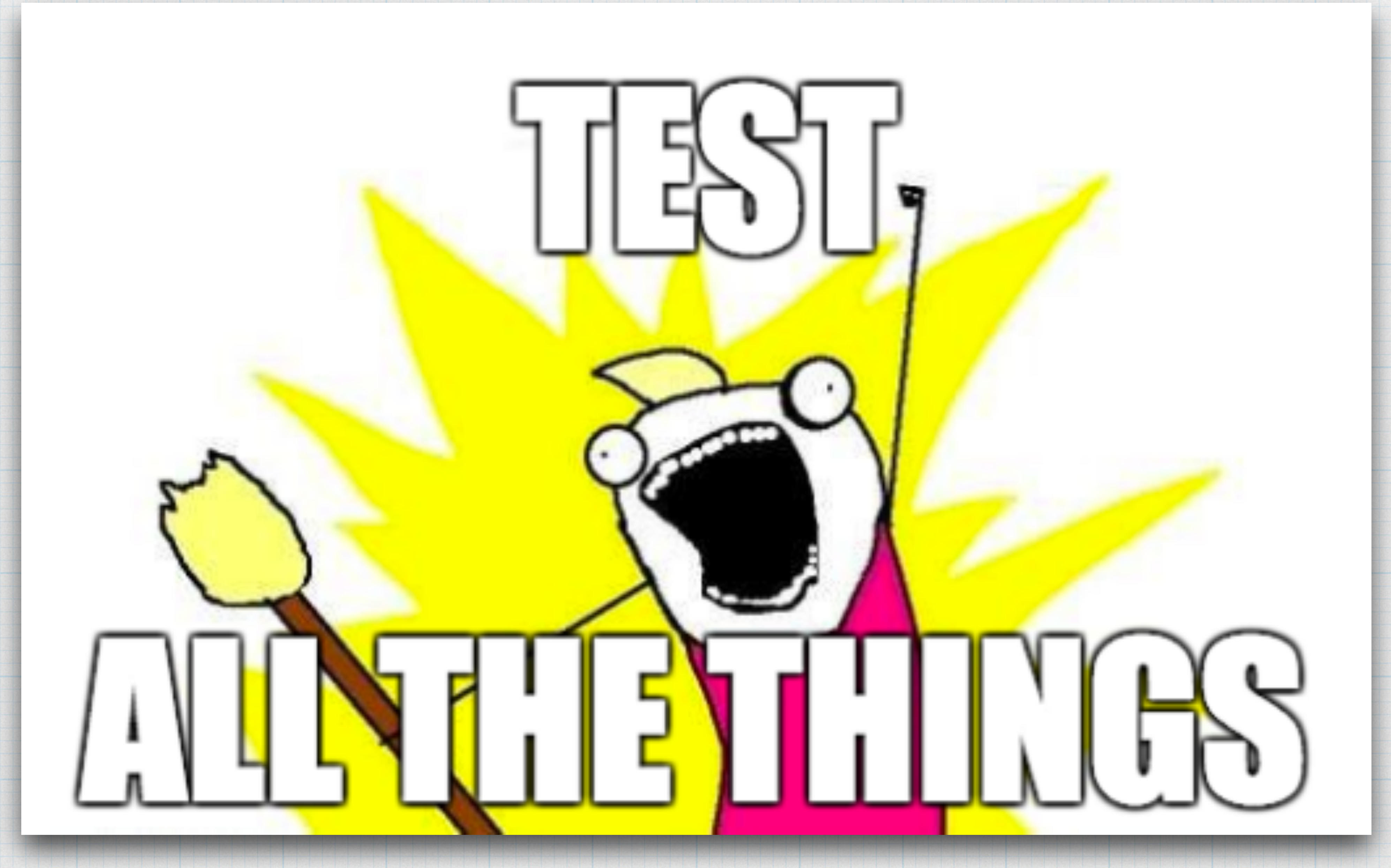
Levels of Testing

- **Black-box testing**: tests in which the test cannot see the inner workings of the item being executed
- **White-box testing**: tests in which the tester is aware of the inner workings of the item being tested

Unit Testing

- Focus on a smallest unit of design (e.g., method, class)
- Test the following:
  - local data structures
  - basic algorithms
  - boundary conditions
  - error handling

What should I test?


TEST
ALL THE THINGS

Some Sound Advice (from StackExchange)

- Test the common case of everything you can
  - This will tell you what code breaks after you make some change
- Test the edge cases of a few unusually complex pieces of code that you think probably have errors
- Whenever you find a bug, write a test case to cover it before fixing it
- Add edge-case tests to less critical code whenever someone has time to kill

https://softwareengineering.stackexchange.com/questions/750/what-should-you-test-with-unit-tests

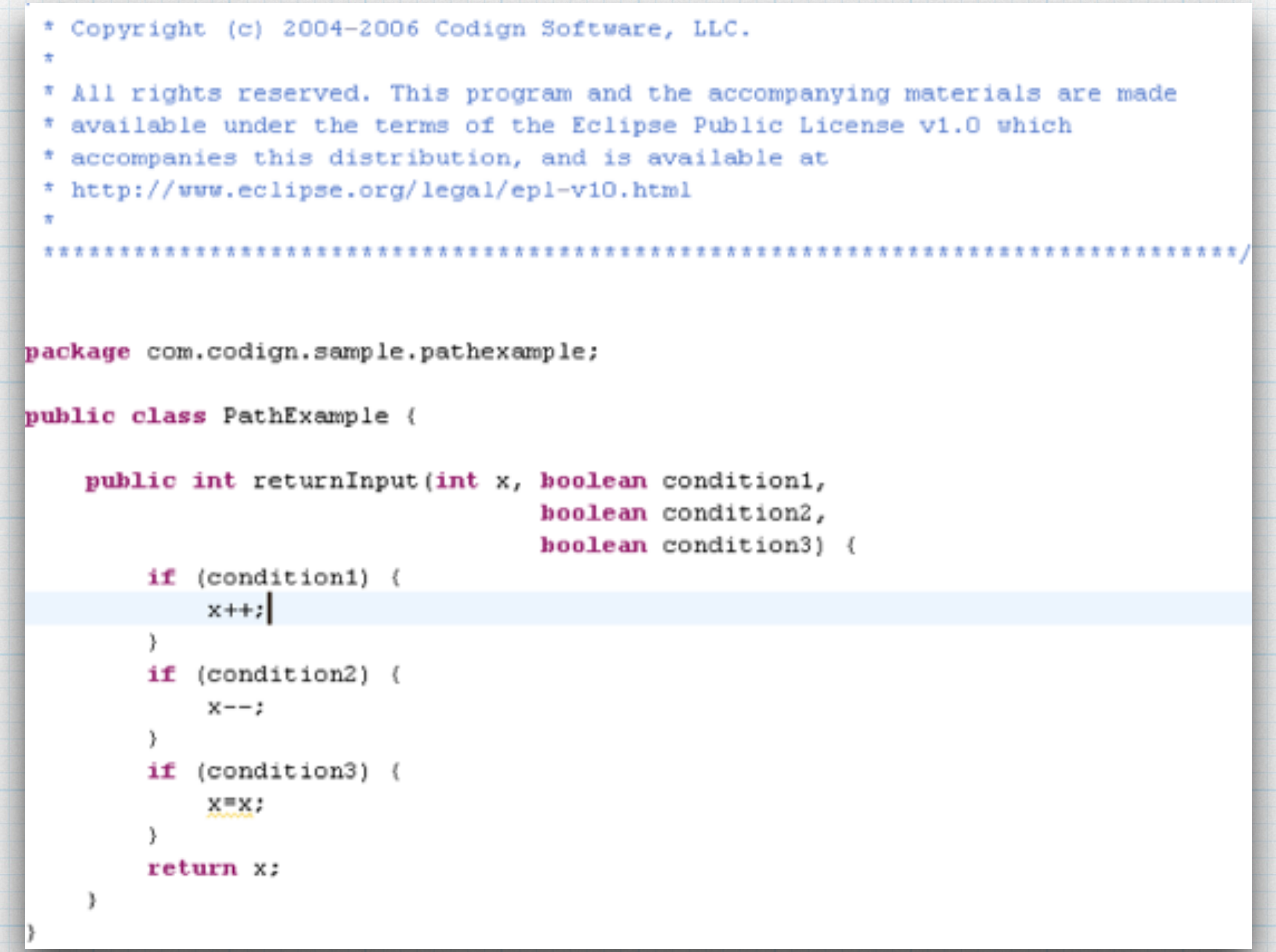
Test Adequacy Criteria

- **Problem 1:** sometimes developers do not write enough tests
- **Problem 2:** sometimes developers write too many redundant tests
- **Problem 3:** during software evolution, we do not have time to (re)run all of the tests; identifying the most relevant tests is hard

Test Coverage

- **Statement coverage**: has each statement been executed by at least one test?
- **Branch coverage**: has each control structure evaluated to both true and false?
- **Path coverage**: has every possible route through the program been executed?

Branch and Path Coverage Example



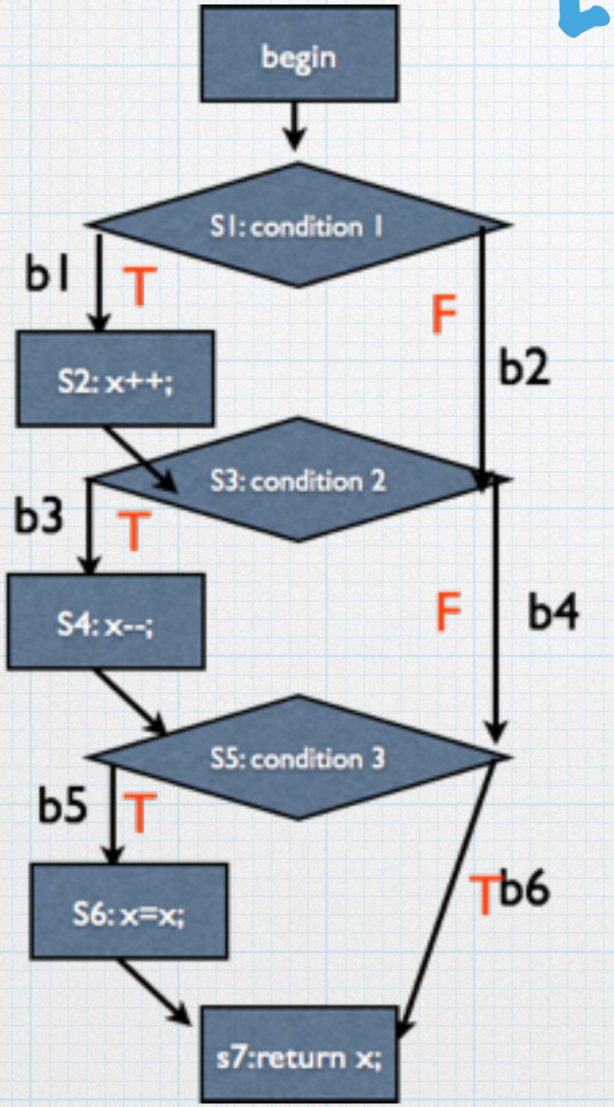
```
 * Copyright (c) 2004-2006 Codign Software, LLC.
 *
 * All rights reserved. This program and the accompanying materials are made
 * available under the terms of the Eclipse Public License v1.0 which
 * accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 *******************************************************************************/


package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1,
                                  boolean condition2,
                                  boolean condition3) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        if (condition3) {
            x=x;
        }
        return x;
    }
}
```

Branch and Path Coverage Example



input	exercised statements	exercised branches	exercised paths
(cond1=true, cond2=true, cond3=true)	s1, s2, s3, s4, s5, s6, s7	b1, b3, b5	[b1, b3, b5]
Coverage			
(cond1=false, cond2=false, cond3=false)			
Coverage			
(cond1=false, cond2=true, cond3=true)			
Coverage			

This Can Get CRAZY

```
public static int fun1(int N) {
    int sum = 0;
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= Math.pow(3, i); j++) {
            System.out.println("HelloWorld");
            if (new Random().nextInt() % 2 == 0)
                sum++;
        }
    }
    return sum;
}
```

How many paths?

Exponential in N!

Test Automation

- Tests must be runnable by script
- Otherwise:
  - Will not be run often
  - You'll forget how to set them up and run them
- Tests should verify their own results without human intervention

Goals of Test Automation

- Automated tests should be repeatable
  - on your machine **and** on someone else's
- Automated tests should be robust
  - a failure should point to a bug (in the system under test (SUT) or in the test)
  - what's a bug? something that leads to code being changed
- Tests should be fast: want timely feedback

Record-Replay Automated Testing

- Idea: record a manual test
    - play back on demand
    - widely used for GUI testing
- Extremely fragile
    - breaks if environment changes anything
    - synchronize with the UI –> SLOW
- Brittle: cannot generalize
    - it’s a literal record: if anything changes, it breaks the test
- Alternative: manual testing: people can adapt to slight modifications

Regression Testing

- Idea: when you find a bug, write a test that exhibits the bug
  - Run that test when the code changes to ensure that the bug doesn't come back
- Fact: without regression testing, old bugs recur frequently
- Regression testing ensures forward progress

Regression Testing (cont.)

- Regression testing should be automatic
  - Ideally run regressions after each change
  - Detect problems as quickly as possible
- But testing is expensive
  - Limits how often it can be run in practice
  - Reducing cost of regression testing is a long standing research problem
    - Example: prioritized testing (run tests that exercise changed code first)

Regression Testing (cont.)

- Regression testing is not just about bug tests
  - requirements/acceptance tests
  - performance tests
- Run entire suite of tests on a regular basis to ensure old tests still work
  - every commit or nightly
  - much easier to fix problems sooner rather than later
  - avoids having new code built on buggy code
- “Smoke test” – subset of full regression test
  - just to make sure nothing is horribly horribly wrong

JUnit

JUnit

- Automated unit testing framework
  - Provides the required environment for the component
  - Executes the individual services of the component
  - Compared the observed program state with the expected program state
  - Reports any deviation from the expectations
  - Does all of this automatically

Sidebar: Assertions

- The main tool of component test is the comparison of the observed state with the expected state using assertions
    - `assert` in Java (JDK 1.4 and later)
- `assert(b)`
    - If `b` is true, nothing happens—the assertion passes
    - If `b` is false, a runtime error occurs
    - C and C++, similar to executing abort()
    - Java, raise `AssertionError` exception

Writing Tests in JUnit

- Each test case is realized by its own class
- Each test of the test case is realized by its own method, annotated with `@Test`
- Statically imported assertion methods from `org.junit.Assert` (or `org.junit.jupiter.Assertions` in JUnit 5)
  - `assertTrue(`*boolean expression*`)`
  - `assertEquals(`*two values*`)`
  - `...`

A JUnit Test Class

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

public class name {
    ...

    @Test
    public void name() {  // a test case method
        ...
    }
}
```

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

JUnit Assertion Methods

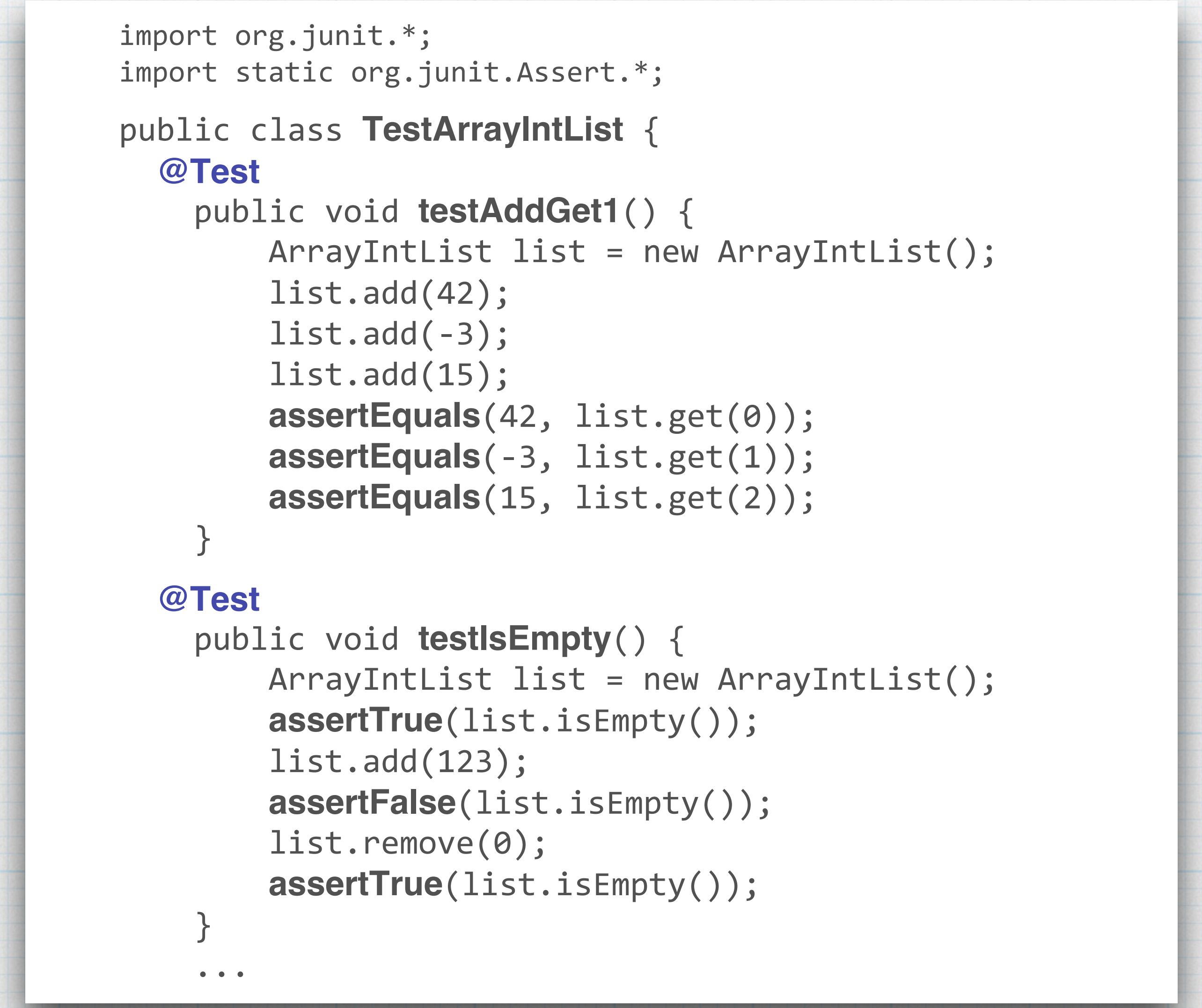
assertTrue(**test**)	fails if the boolean test is false
assertFalse(**test**)	fails if the boolean test is true
assertEquals(**expected**, **actual**)	fails if the values are not equal
assertSame(**expected**, **actual**)	fails if the values are not the same (by ==)
assertNotSame(**expected**, **actual**)	fails if the values *are* the same (by ==)
assertNull(**value**)	fails if the given value is *not* null
assertNotNull(**value**)	fails if the given value is null
fail()	causes current test to immediately fail

Each method can also be passed a string to display if it fails:

e.g. assertEquals("**message**", **expected**, **actual**)

Why is there no pass method?

ArrayIntList JUnit Test



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

public class TestArrayIntList {
  @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

  @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
    ...
```

JUnit Example

Given a `Date` class with the following methods:

```
public Date(int year, int month, int day)
public Date()                   // today
public int getDay(), getMonth(), getYear()
public void addDays(int days)   // advances by days
public int daysInMonth()
public String dayOfWeek()       // e.g. "Sunday"
public boolean equals(Object o)
public boolean isLeapYear()
public void nextDay()           // advances by 1 day
public String toString()
```

What's Wrong?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
}
```

Well-Structured Assertions

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());   // expected
        assertEquals(2, d.getMonth());     // value should
        assertEquals(19, d.getDay());      // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

In JUnit 5, the optional assertion message is now the LAST parameter

Expected Answer Objects

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d);  // use an expected answer
    }                               // object to minimize tests

                                    // (Date must have toString
    @Test                           //  and equals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

Naming Test Cases

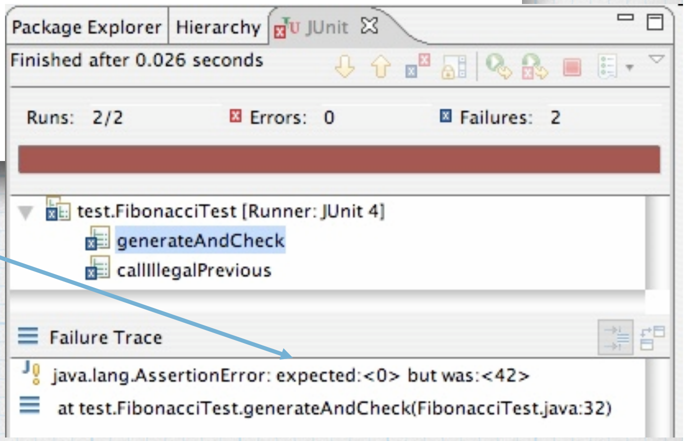
```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
  // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

What's Wrong With This?

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

JUnit will already show the expected and actual values in its output; don't need to repeat them in the assertion message



Finished after 0.026 seconds

Runs: 2/2 Errors: 0 Failures: 2

test.FibonacciTest [Runner: JUnit 4]
generateAndCheck
callIllegalPrevious

Failure Trace

```
java.lang.AssertionError: expected:<0> but was:<42>
at test.FibonacciTest.generateAndCheck(FibonacciTest.java:32)
```

Tests With a Timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

considered a failure if it doesn't finish running in 5000 ms

```
private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() { ... }
```

times out/fails after 2000 ms

Pervasive Timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, d);
    }


    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // almost every test should have a timeout so it can't
  // lead to an infinite loop; good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Testing for Exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

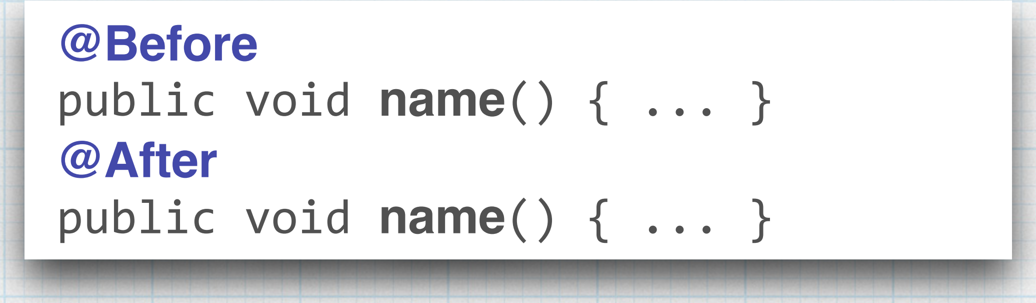
will pass if it does throw given exception (fails if the exception is not thrown)

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);    // should fail
}
```

replaced with assertThrows in Junit 5

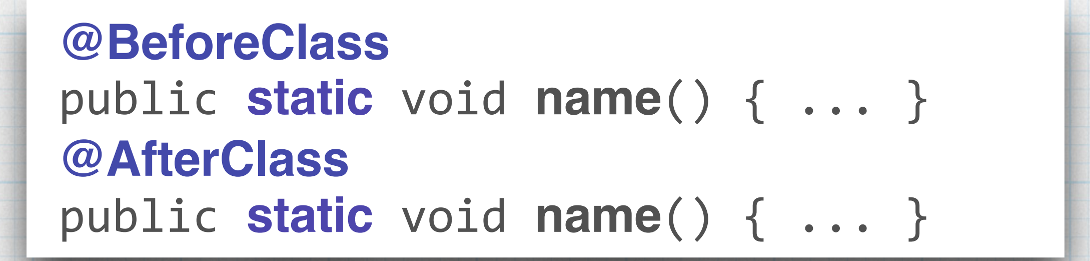
Setup and Teardown

- Methods to run before/after each test cast 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() { ... }
```

replaced with @BeforeEach and @BeforeAll in Junit 5

More Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

Trustworthy Tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.
- Tests should avoid logic.
  - minimize `if/else`, **loops**, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.
- Torture tests are okay, but only *in addition* to simple tests.

JUnit Exercise

- Given a Date class with the following methods:
  - `public Date(int year, int month, int day)`
  - `public Date()                     // today`
  - `public int getDay(), getMonth(), getYear()`
  - `public void addDays(int days)    // advances by days`
  - `public int daysInMonth()`
  - `public String dayOfWeek()        // e.g. "Sunday"`
  - `public boolean equals(Object o)`
  - `public boolean isLeapYear()`
  - `public void nextDay()            // advances by 1 day`
  - `public String toString()`
- Come up with unit tests to check the following:
  - That no **`Date`** object can ever get into an invalid state.
  - That the **`addDays`** method works properly.

Squashing Redundancy

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_withinSameMonth_1() {
    addHelper(2050, 2, 15, +4, 2050, 2, 19);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_wrapToNextMonth_2() {
    addHelper(2050, 2, 15, +14, 2050, 3, 1);
    }

  // use lots of helpers to make actual tests extremely short
    private void addHelper(int y, int m, int d, int add,
                           int y2, int m2, int d2) {
        Date act = new Date(y, m, d);
        act.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", exp, act);
    }

  // can also use "parameterized tests" in some frameworks
    …
```

Flexible Helpers

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
    addhelper(d, +32, 2050, 4, 2);
    addhelper(d, +98, 2050, 7, 9);
    }

  // Helpers can box you in; hard to test many calls/combine.
  // Create variations that allow better flexibility
    private Date addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date date = new Date(y1, m1, d1);
        addHelper(date, add, y2, m2, d2);
        return date;
    }

    private void addHelper(Date date, int add,
                           int y2, int m2, int d2) {
        date.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("date after +" + add + " days", expect, date);
    }
    ...
```

Regression Testing

- **regression**: a feature that worked previously no longer works
  - Likely when code changes/grows over time
  - A new feature/fix can cause new bug or reintroduce a bug
- **regression testing**: re-executing prior unit tests after a change
  - Often done by scripts during automated testing
  - Used to ensure old fixed bugs are still fixed
  - Gives your app a minimum level of working functionality
- Many products have set of mandatory check-in tests that must pass before code can be added to a repo

Test Driven Development

- Unit tests can be written after, during, or even **before** coding.
  - **test-driven development**: Write tests, *then* write code to pass them.
- Imagine that we'd like to add a method `subtractWeeks` to our `Date` class, that shifts this `Date` backward in time by the given number of weeks.
- Write code to test this method *before* it has been written.
  - Then once we do implement the method, we'll know if it works.

Tests and Data Structures

- Need to pass lots of arrays? Use array literals

```
public void exampleMethod(int[] values) { ... }
...
exampleMethod(new int[] {1, 2, 3, 4});
exampleMethod(new int[] {5, 6, 7});
```

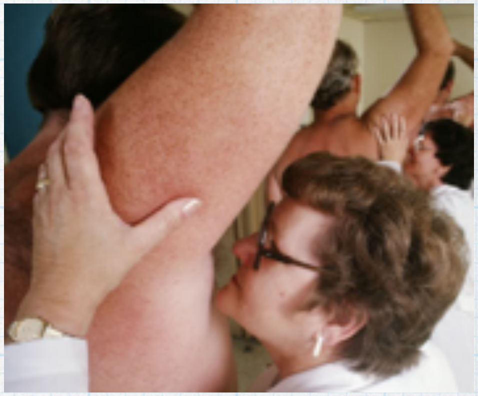
- Need a quick ArrayList? Try `Arrays.asList`

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Need a quick set, queue, etc.? Many collections can take a list

```
Set<Integer> list = new HashSet<Integer>(
                        Arrays.asList(7, 4, -2, 9));
```

Test Case "Smells"



- Tests should be self-contained and not care about each other.
- "Smells" (bad things to avoid) in tests:
  - Constrained test order (Test A must run before Test B) (usually a misguided attempt to test order/flow)
  - Tests call each other (Test A calls Test B's method) (calling a shared helper is OK, though)
  - Mutable shared state (Tests A/B both use a shared object) (If A breaks it, what happens to B?)

Test Suites

- test suite: one class that runs many JUnit tests.
  - an easy way to run all of your app's tests at once.

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestCaseName.class,
    TestCaseName.class,
    ...
    TestCaseName.class,
})
public class name {}
```

Test Suite Example

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    WeekdayTest.class,
    TimeTest.class,
    CourseTest.class,
    ScheduleTest.class,
    CourseComparatorsTest.class
})
public class HWTests {}
```

JUnit Summary

- Tests need failure atomicity (ability to know exactly what failed)
- Each test should have a clear, long, descriptive name
  - Assertions should always have clear messages to know what failed
  - Write many small tests, not one big test
    - Each test should have roughly just 1 assertion at its end
- Always use a `timeout` parameter to every 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

Questions?

Additional Resources

- Selenium
- Emma