

# Object-Oriented Testing & JUnit

## Acknowledgement:

Materials adapted from Prof. M. Stepp, M. Ernst, S. Reges,  
D. Notkin, R. Mercer, Wikipedia

# Object-Oriented Testing

- *When should testing begin?*
- Analysis and Design:
  - Testing begins by evaluating the OOA and OOD models
  - *How do we test OOA models (requirements and use cases)?*
  - *How do we test OOD models (class and sequence diagrams)?*
  - Structured walk-throughs, prototypes
  - Formal reviews of correctness, completeness and consistency
- Programming:
  - *How does OO make testing different from procedural programming?*
  - Concept of a 'unit' broadens due to class encapsulation
  - Integration focuses on *classes* and their context of a use case scenario or their execution across a thread
  - Validation may still use conventional black box methods

# Bugs and testing

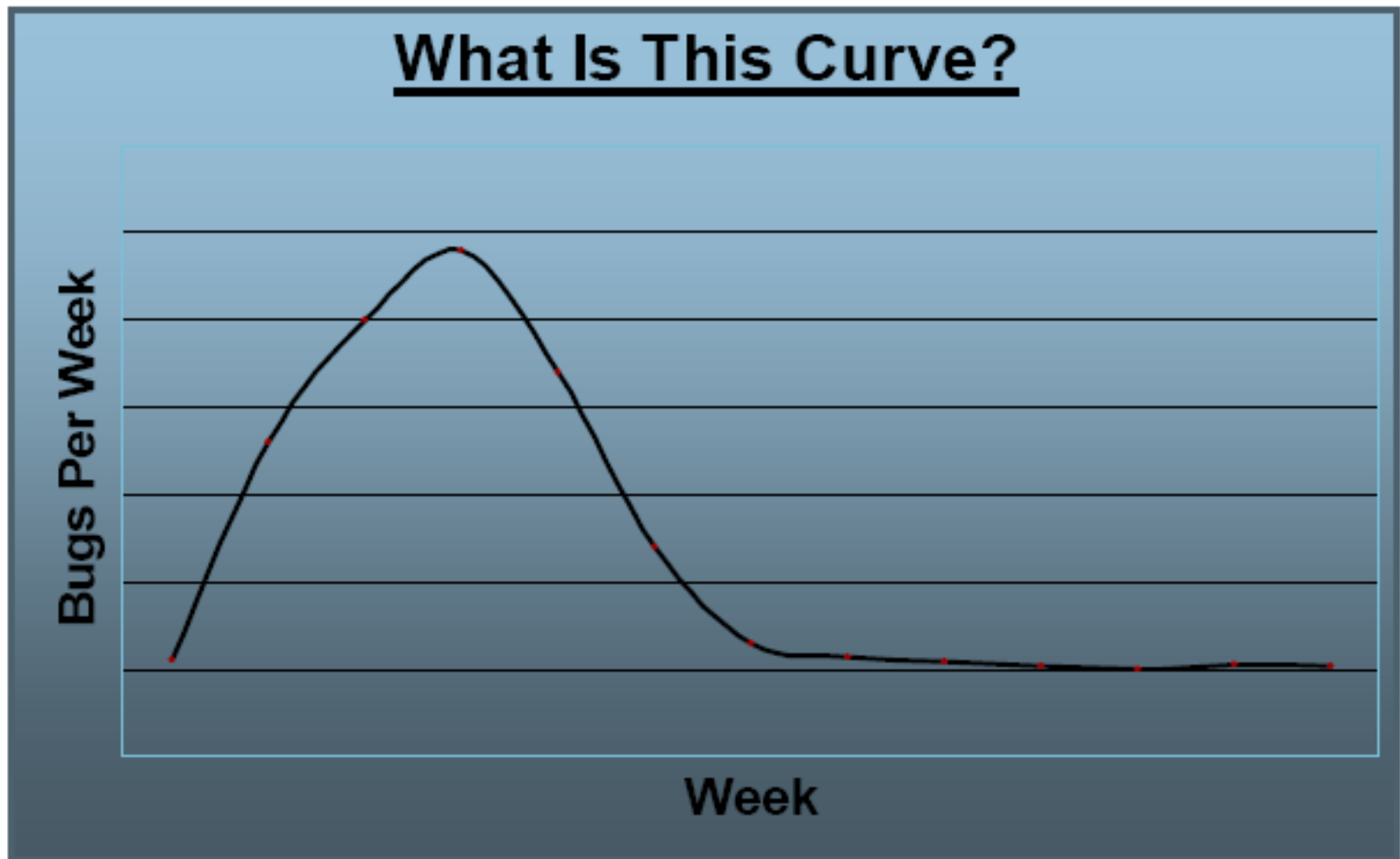


- **Software reliability:** Probability that a software system will not cause failure under specified conditions.
  - Measured by uptime, MTTF (mean time till failure), crash data.
- **Bugs** are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.
- **Testing:** A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation).

# Difficulties of testing

- Perception by some developers and managers:
  - Testing is seen as 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 you:
  - It is impossible to completely test a system.
  - Testing does not always directly reveal the actual bugs in the code.
  - Testing does not prove the absence of errors in software.

# The Bug Curve



# Criteria for Completion of Testing

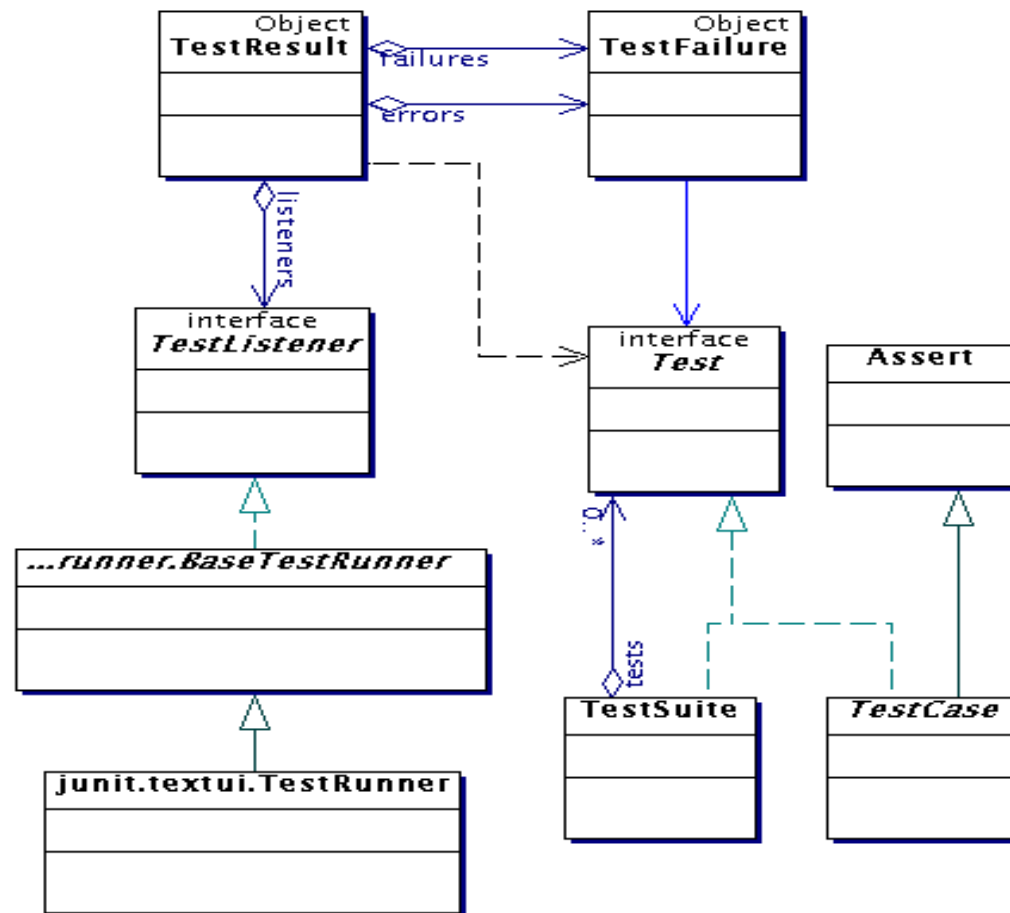
- When are we done testing? (Are we there yet?)
- How to answer this question is still a research question
  - One view: testing is never done... the burden simply shifts from the developer to the customer
  - Or testing is done when you run out of time or money
  - Or use a statistical model:
    - Assume that errors decay logarithmically with testing time
    - Measure the number of errors in a unit period
    - Fit these measurements to a logarithmic curve
    - Can then say: “with our experimentally valid statistical model we have done sufficient testing to say that with 95% confidence the probability of 1000 CPU hours of failure free operation is at least 0.995”
  - Or machine learning-based:
    - iSENSE: Completion-Aware Crowdfunding Management.
      - URL: <https://dl.acm.org/citation.cfm?id=3339619>

# Unit testing



- **Unit testing:** Looking for errors in a subsystem in isolation.
  - Generally a "subsystem" means a particular class or object.
  - The Java library **JUnit** helps us to easily perform unit testing.
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail.

# JUnit Mechanics

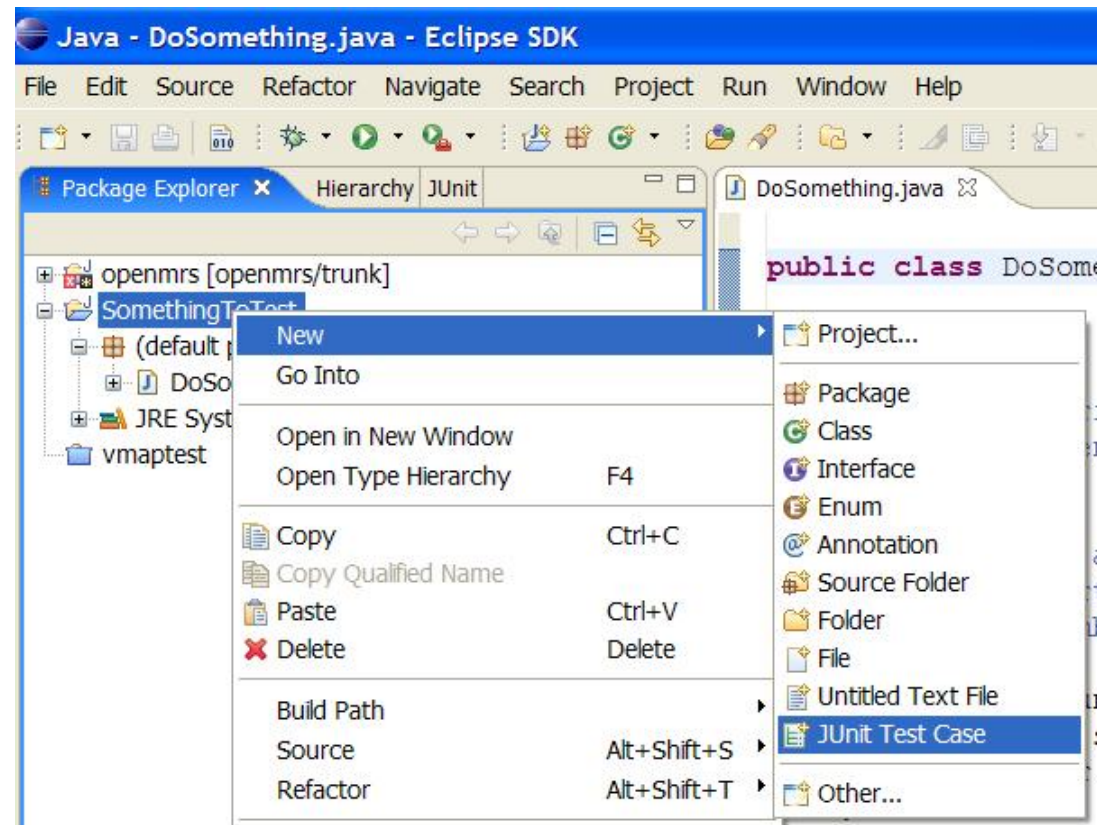




# JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project** → **Properties** → **Build Path** → **Libraries** → **Add Library...** → **JUnit** → **JUnit 4** → **Finish**

- To create a test case:
  - right-click a file and choose **New** → **Test Case**
  - or click **File** → **New** → **JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



# A JUnit test class

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

public class name {
    ...

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

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

# JUnit assertion methods

<code>assertTrue(<b>test</b>)</code>	fails if the boolean test is <code>false</code>
<code>assertFalse(<b>test</b>)</code>	fails if the boolean test is <code>true</code>
<code>assertEquals(<b>expected</b>, <b>actual</b>)</code>	fails if the values are not equal
<code>assertSame(<b>expected</b>, <b>actual</b>)</code>	fails if the values are not the same (by <code>==</code> )
<code>assertNotSame(<b>expected</b>, <b>actual</b>)</code>	fails if the values <i>are</i> the same (by <code>==</code> )
<code>assertNull(<b>value</b>)</code>	fails if the given value is <i>not</i> <code>null</code>
<code>assertNotNull(<b>value</b>)</code>	fails if the given value is <code>null</code>
<code>fail()</code>	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?

org.junit.Assert: <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

# ArrayIntList JUnit test

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

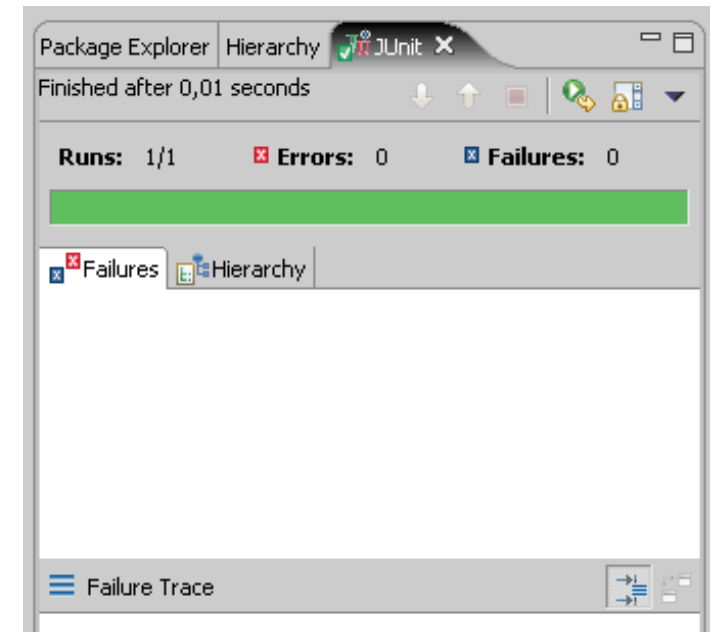
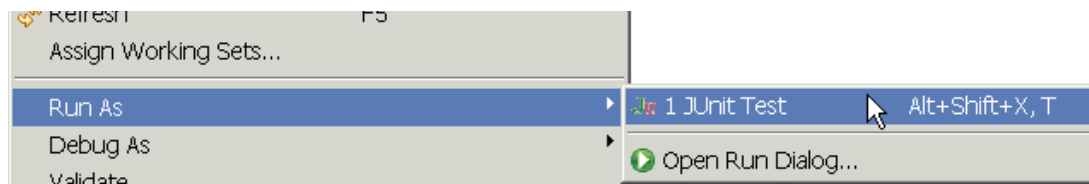
public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));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

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

    ...
}
```

# Running a test

- Right click it in the Eclipse Package Explorer at left; choose:  
**Run As → JUnit Test**
- The JUnit bar will show **green** if all tests pass, **red** if any fail.
- The Failure Trace shows which tests failed, if any, and why.



# 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.
    - It should be efficient enough to add 1,000,000 days in a call.

# What's wrong with this?

```
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
}
```



# 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
                                        // and equals methods)
    @Test
    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
}
```

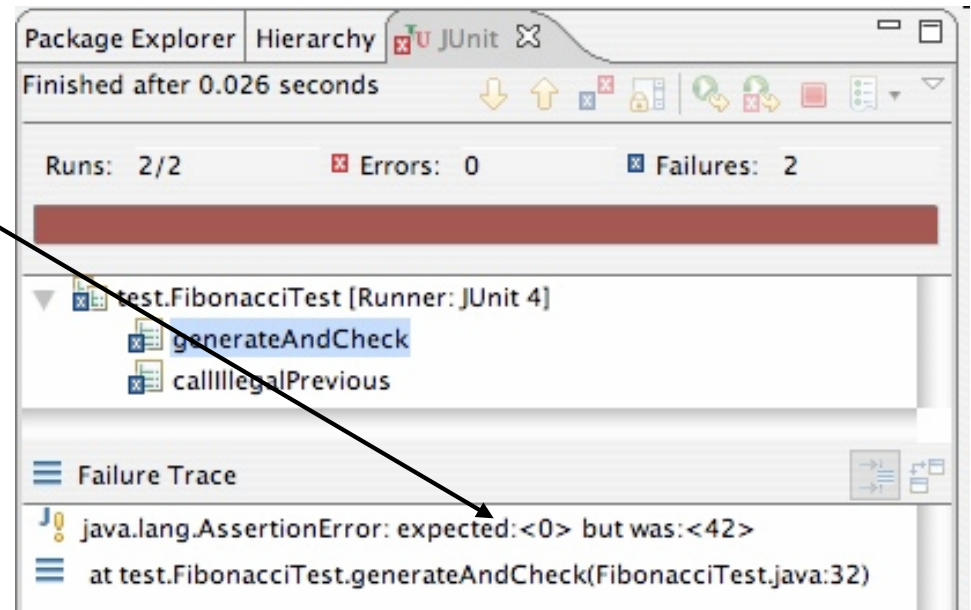
# What's wrong with this?

```
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);  
    }  
    ...  
}
```

# Good assertion messages

```
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("adding one day to 2050/2/15",  
            expected, actual);  
    }  
    ...  
}
```

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



# Tests with a timeout

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

- The above method will be considered a failure if it doesn't finish running within 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 the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

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

# Setup and teardown

## **@Before**

```
public void name() { ... }
```

## **@After**

```
public void name() { ... }
```

- methods to run before/after each test case method is called

## **@BeforeClass**

```
public static void name() { ... }
```

## **@AfterClass**

```
public static void name() { ... }
```

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



# 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

# What's wrong with this?

```
public class DateTest {
    // test every day of the year
    @Test(timeout = 10000)
    public void tortureTest() {
        Date date = new Date(2050, 1, 1);
        int month = 1;
        int day = 1;
        for (int i = 1; i < 365; i++) {
            date.addDays(1);
            if (day < DAYS_PER_MONTH[month]) {day++;}
            else {month++; day=1;}
            assertEquals(new Date(2050, month, day), date);
        }
    }

    private static final int[] DAYS_PER_MONTH = {
        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    }; // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
}
```

# 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.

# 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 y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date act = new Date(y, m, d);
        actual.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(y, m, d);
        addHelper(date, add, y2, m2, d2);
        return d;
    }

    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, d);
    }

    ...
}
```

# Regression testing

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

# 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));
```



# What's wrong with this?

```
public class DateTest {  
    // shared Date object to test with (saves memory!!1)  
    private static Date DATE;  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_sameMonth() {  
        DATE = new Date(2050, 2, 15); // first test;  
        addhelper(DATE, +4, 2050, 2, 19); // DATE = 2/15 here  
    }  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_nextMonthWrap() { // second test;  
        addhelper(DATE, +10, 2050, 3, 1); // DATE = 2/19 here  
    }  
  
    @Test(timeout = DEFAULT_TIMEOUT)  
    public void addDays_multipleCalls() { // third test;  
        addDays_sameMonth(); // go back to 2/19;  
        addhelper(DATE, +1, 2050, 2, 20); // test two calls  
        addhelper(DATE, +1, 2050, 2, 21);  
    }  
    ...  
}
```

# 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 HW2Tests {}
```

# 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.

# Resources

- <http://www.junit.org>
  - Download of JUnit
  - Lots of information on using JUnit
- <http://junit.sourceforge.net/>
- <http://code.google.com/p/t2framework/wiki/JUnitQuickTutorial>
- <http://www.thecoadletter.com>
  - Information on Test-Driven Development