Running tests

- It can be very tedious to run tests by hand
  - Need to have a test harness that will construct and pass in the right inputs
  - Need to look at the output, and compare it to the expected output
  - Need to handle exceptions, too
- So, let's make tools!

Programming unit tests

- In Java, a simple strategy for unit testing is to define *self-testing* classes
- Each class can define a static main method that runs some set of unit tests of the class
  - The main method builds arguments, invokes operations, checks results, handles exceptions
  - To run, just invoke the class as if it were the main application
    - java MyDataStructure
- Still pretty tedious...

Making unit tests easier

- There exist tools to help in constructing unit test harnesses
- E.g. Junit, a unit test framework for Java
  - Constructs a nice report of successes and failures
  - Provides some convenient helper functions

Defining a JUnit test case

- Import junit.framework.*
- Define a subclass of TestCase
- Implement any number of void test*XXX*() methods
  - Can invoke:
    - assertTrue("*msg*", *testExpr*);
    - assertEquals("*msg*", *expr1*, *expr2*);
    - fail("*msg*");
  - Can throw exceptions
- Can implement void setUp() to initialize some state used by each text*XXX* method

Making test cases runnable

- Add to your TestCase subclass *myTests*:

```
public static Test suite() {
  return new TestSuite(myTests.class);
}
public static void main(String[] args) {
  junit.textui.TestRunner.run(suite());
}
```

- Now can run it:

```
% java myTests
```


More on JUnit

- Can collect multiple TestCase subclasses into a larger TestSuite
  - TestCase and TestSuite implement Test
- Can use a GUI interface to run tests

```
% java junit.swingui.TestRunner myTests
```

- For more info, see http://junit.org
  - "Test Infected: Programmers Love Writing Tests"

Regression test suites

- Goal: accumulate a lot of good unit tests
  - Run them frequently after changes
- A good *regression test suite* gives confidence in development
  - Confidence to try big clean-ups without introducing uncaught bugs
  - Confidence to commit changes to rest of team

Beyond unit tests

- Unit tests aren't enough!
- Need to test that the units work together: *integration testing*
- [Why might errors crop up when testing groups of units that weren't caught when unit testing?]

Defensive programming

- The best programmers are defensive
  - They design & implement code that is unlikely to break
  - If there is a problem, the code breaks quickly and clearly
- Some strategies:
  - Minimize preconditions
  - Insert an assertion whenever they mentally expect and rely on something being true

Programming for change

- Expect change:
  - To software's design & requirements
  - To interfaces
  - To data structures
  - To people on the project
- Write code that minimizes reliance on things that might change, & is flexible in face of future changes
  - Fewer bugs introduced when these things change

Other tools

- Programming language choice(s) influence how likely programs are to be correct, how easy programs are to debug
  - E.g. array bounds checking, static type checking
- Programming environment tools can help mechanize much of testing
  - JUnit is a simple example
  - Some advanced static analysis tools can help to find bugs