CS11 – Advanced Java

Winter 2011-2012
Lecture 2
Today’s Topics

- Assertions
- Java 1.5 Annotations
- Classpaths
- Unit Testing!
- Lab 2 hints 😊
Assertions!

- Assertions are a very useful language feature
- Provide two major benefits
- Can **test** the assumptions that your code makes
  - Add statements to your code that test your assumptions
  - These assertions are tested at runtime
  - If assertion is violated then program is halted with an error
- Assertions also **document** your assumptions
  - Like javadoc, the code specifies its own assumptions
  - Other developers can read your code and see exactly what you think should be true
Assertions in Java

- Java 1.4 added an `assert` keyword
  ```java
  assert result >= 0;
  ```
  - Condition must evaluate to a `boolean` value
  - No parentheses required around the condition

- If the condition is false at runtime, a `java.lang.AssertionError` is thrown
  - `AssertionError` is in the `Error` subtree of the Java exception hierarchy
    - Since it’s an exception, it includes a stack-trace for where the assertion-failure occurred
  - From Java API for `java.lang.Error`:
    - An `Error` is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch.
Simple assert syntax:
```java
assert cond;
```
- `cond` must evaluate to a `boolean` value

Can also specify details for when failure occurs:
- ```java
  assert cond : expr;
  ```
- `expr` must evaluate to something
  - e.g. it cannot be a call to a `void` function
  - `expr` is only evaluated if `cond` is false

Error details should indicate what went wrong
- Make it easy to debug your software!
- Example:
  ```java
  assert result >= 0 : "Bad result " + result;
  ```
Disabling Assertions

- Assertions are sometimes expensive to test
  - Example: a class that can sort its contents
    ```java
    public class Sorter {
        public boolean inOrder() {
            ... // Iterate through contents to test
            order
        }

        public void sort() {
            ... // Do the sorting magic here
            assert inOrder() : "My sort is broken!";
        }
    }
    ```
- Java can enable/disable assertions at runtime
  - A class’ assertion behavior is enforced when it is loaded
  - Can’t turn on/off a class’ assertions after the class is loaded
Disabling Assertions (2)

- Java VM uses these arguments for assertions:
  - `-enableassertions` (or `-ea`)
    - Enables assertions in all classes except system classes
  - `-disableassertions` (or `-da`)
    - Disables assertions in all classes except system classes

- Example options:
  - `-ea package.ClassName` or `-da package.ClassName`
    - Enables/disables assertions in a specific class
  - `-ea package...` or `-da package...`
    - Enables/disables assertions in all classes in a package

- To enable/disable assertions in system classes:
  - `-enablesystemassertions` or `-esa`
  - `-disablesystemassertions` or `-dsa`
Java Assertion Guidelines

- **Do not** use Java assertions for verifying the arguments of public APIs!
- Standard Java approach is to use **exceptions** to flag invalid arguments
- A *small* set of examples from the Java API:
  - `NullPointerException`
    - null was specified for a required reference-argument
  - `IndexOutOfBoundsException`
    - an index argument was out of the required range
  - `NumberFormatException`
    - a string representation of a number is not the correct format
  - `IllegalArgumentException`
    - a general catch-all for bad arguments
Don’t put required code into assertion tests!

```java
assert set.remove(obj) : "obj not found: " + obj;
```

Problem?
- When this assertion is disabled, the remove operation won’t occur at all!

Many more guidelines for assertions in Java
- For more info, see “Programming with Assertions”

Java Naming Conventions

- A common usage pattern for classes in Java:
  - Create a class for use in a 3rd-party framework
  - Frequently, the class needs to adhere to certain naming conventions
    - Framework can look up methods and fields on the class
    - External dev tools can parse the code and find methods/fields

- Example: J2EE web-application frameworks
  - Enterprise JavaBeans (EJBs) encapsulate web-app logic
  - EJBs must implement certain interfaces, and EJB methods must follow certain naming conventions
  - When these rules are violated, J2EE application server gets very unhappy.
Java Annotations

- Java 1.5 introduces a simpler solution:
  - Attach annotations (i.e. metadata) to classes, and their fields and methods

- Annotations can be extracted by external tools
  - Instead of looking for methods with a particular name or signature, retrieve all methods with a specific annotation

- Annotations are also used by the Java compiler, VM
  - Examples:
    - “this method is deprecated”
    - “this method implements an interface method”
    - “this method overrides a parent-class method”
Annotations are like classes
- They have a specific type
- They can contain fields to store annotation details

Annotation specifications include:
- What they can appear on (e.g. only classes, or only methods)
- A retention policy: when and where they are made available
  - “Source” – only available at compile-time
  - “Class” – annotations included in compiled class file, but JVM may discard them at load-time
  - “Runtime” – annotations must be kept by the JVM at runtime, so that they can be extracted and read by other code
A Simple Example

- You need to write a 2D point class
  ```java
class Point2d {
    private double xCoord, yCoord;

    public boolean equals(Point2d obj) {
      ... // Implementation of equals
    }
}
```

- Problems?
  - This is not a correct declaration of `equals()`!
  - Must take an argument of type `Object`
- The compiler doesn’t tell us there is a problem!
  - Code just acts bizarrely when used with collections, etc.
Now with Annotations

- Java provides some annotations for you to use
  - `@Override` – A method overrides a parent-class method

Update our code:

```java
public class Point2d {
    private double xCoord, yCoord;

    @Override
    public boolean equals(Point2d obj) {
        ...
        // Implementation of equals
    }
}
```

- Since we didn’t declare `equals()` properly, it doesn’t actually override `Object.equals()`
  - The compiler reports an error, and now you can fix your bug.
More Annotation Details

- You can create your own annotations too!
  - Create your own Java class-processing tools
  - 3\textsuperscript{rd}-party tools and frameworks have their own annotations to use in your software

- Java annotation documentation

- Java Annotation Processing Tool
Java Classpaths

- When a Java program refers to a class, the class’ definition has to be available somewhere
  
  ```java
  import javax.vecmath.Vector3f;
  ...
  Vector3f v = new Vector3f(1.0f, 0.0f, 0.0f);
  ```

- When the code is compiled, `javac` has to find definition of `javax.vecmath.Vector3f`

- When the code is run, the JVM has to find this definition too

- The **classpath** tells Java where to look for class definitions

  - Default classpath is the current directory “.”
  - (Java system classes aren’t handled via this classpath…)
Specifying the Classpath

- When you are using external libraries, you need to specify the classpath
  - `javax.vecmath.Vector3f` is in Java3D library
  - Not in standard Java API, and not in our local directory!

- Two ways:
  - Use `-classpath` (or `-cp`) argument to `javac` and `java`
  - Specify the `CLASSPATH` environment-variable

- This value is a path expression
  - File-separators and path-separators depend on the OS!
  - Windows: `-cp C:\path\one;C:\path\two`
  - Linux/Mac: `-cp /path/one:/path/two`
  - If path contains spaces, enclose it with double-quotes
Specifying the Classpath (2)

- The classpath can include:
  - A path to a directory, if the directory contains `.class` files
  - A path to a specific JAR file
    - JAR files are archives of Java class files; JAR = Java ARchive
    - More on JAR files in a few weeks
    - (See docs and `jar` utility if you are curious)

- Classpaths cannot simply refer to the directory where JAR files reside!
  - Must actually specify the JAR files themselves in the classpath
Classpath Example

- If our `Vector3f` class lives in `vecmath.jar`
  - If `vecmath.jar` is in the same directory:
    - `javac -cp vecmath.jar MyClass.java`
  - If `vecmath.jar` lived somewhere else:
    - `javac -cp /path/to/vecmath.jar MyClass.java`
  - Running our code is similar:
    - `java -cp /path/to/vecmath.jar MyClass`

- Specifying the classpath eliminates the current directory from the path
  - May need to do this kind of thing in some circumstances:
    - `javac -cp .:/path/to/vecmath.jar MyClass.java`
    - `java -cp .:/path/to/vecmath.jar MyClass`
Testing the Word List

- Last week you created a word-list class
  - Wrote a very simple test for it
  - A lot of functionality went untested!
- Would like to create a series of test cases to exercise our class
  - Each test exercises a single feature of our class
  - If a test fails, should be simple to diagnose and solve
- Unit testing:
  - Tests for the smallest verifiable units of your program
  - In Java, the smallest testable units are methods on a class
Unit-Testing Goals

- Ideally, your test suite should exercise all your code
  - Every code-path through your program
  - Tests that verify normal behavior
  - Tests that verify error-handling behavior too!
    - Called “negative tests”
    - Make sure proper exceptions are thrown in error cases
    - Make sure program doesn’t end up in an invalid state
    - Make sure program releases any allocated resources

- Code-coverage tools measure how much code is exercised by a test suite
  - Several different measures for code coverage
  - Critical applications often require 100% coverage
Unit-Testing Goals (2)

- Unit-testing attempts to isolate each class, and ideally each method
  - Makes identification and resolution of bugs much easier
- Classes frequently reference other classes…
  - Often hard to test a single class in isolation
- Unit-testing motivates separation of interface from implementation
  - Classes interact with each other through well-defined interfaces
  - Test suite provides a dummy implementation for the class being tested to use
  - Can also use dummy impl. to simulate various cases
Unit-Testing Limitations

- Unit-testing is an easy way to improve software quality
  - *No excuse* to not employ unit-testing on your software
- Still only exercises individual units…
  - May be larger-scale design issues, incompatibilities, etc.
- Integration testing:
  - Individual components and modules are combined and tested as a group
  - Usually started after unit-testing has made good headway
- System testing:
  - Entire software system is tested and verified, as a whole
  - Follows after integration testing has made good progress
Regression Testing

- One other important testing methodology to know about: regression testing

Scenario:
- You are working on a software project that has a test suite
- You make some changes to the project...
- Suddenly there are new failures for tests that used to pass!

This is called a regression
- You broke a feature that used to work (more common)
- You added code that exposed a hidden bug (less common)

Extremely important to prevent regressions!
- Especially true when fixing bugs on released software
- Customer wants a bug-fix release that makes their life better, not worse.
Two main practices for finding and preventing regressions!

First practice:
- When you add a new feature or fix a bug, run the entire test suite against your software
- If your test suite is complete, will quickly identify any regressions that your changes have caused

Second practice:
- Whenever a new bug is discovered, write a specific test case to check for that specific bug

Good software companies employ both of these practices on their software products
Java Unit-Testing Frameworks

- Easiest to manage testing operations within a test framework
  - Each unit-test is implemented as a separate method
  - Can group tests into different categories
    - e.g. “smoke tests,” “regression tests,” “long tests”
  - Run groups of tests from a unified entry-point
  - View summary results in a clean and concise way

- Java has two very well-known testing frameworks
  - JUnit (http://www.junit.org)
    - Older and well-established, but with some big limitations
  - TestNG (http://testng.org)
    - New alternative created to solve JUnit’s deficiencies
JUnit vs. TestNG

- JUnit is focused primarily on unit-testing
  - Does a great job with simple unit-testing
  - Doesn’t do so well with integration testing, or other more advanced testing patterns
- TestNG is designed to handle many different kinds of testing
  - Unit testing and integration testing both supported
  - Can specify dependencies between tests
    - For integration tests, may need a series of steps
- We will use TestNG this term
Tests and Annotations

- **Old JUnit 3.x approach:**
  - Implement test methods on a test class
    - Method name must start with “test”
    - Method signature: no arguments, no return-value
    - Method must have public access, and cannot be static.

- **JUnit 4 and TestNG approach:**
  - Annotate test methods with a `@Test` annotation
  - No other real requirements on test methods

- Both test frameworks provide many other annotations for various uses
A simple test class for our word-list:

```java
import org.testng.annotations.*;

public class TestWordList {
    /** Test the WordList default constructor. */
    @Test
    public void testDefaultCtor() {
        WordList wl = new WordList();
        assert wl.size() == 0;
        // Make sure internal set was initialized.
        assert !wl.contains("random");
    }
}
```

Add more test methods, marked with `@Test` etc.
Compiling Your Tests

- Java compiler needs to know about TestNG JAR file
  - Contains the TestNG annotations, in particular

- Example *nix command-line:
  
```bash
javac -cp .:testng-5.8-java15.jar
   TestWordList.java
```
  
- ...assuming that all files, including TestNG JAR, are in current directory
- On Windows, use ; instead of : in the classpath
Running Your Tests

- TestNG takes an XML configuration file
  - testng.xml
  - Details are on TestNG website

- For this week, just specify the test classes on the command-line
  - `java -cp .:testng-5.8-java15.jar org.testng.TestNG \
    -testclasses TestWordList`
  - For multiple classes, separate names with spaces
    - `testclasses TestWordList TestBoggleBoard`
Grouping Tests

- Can specify one or more groups for each test
  ```java
  /** Test the WordList default constructor. */
  @Test(groups = {"basic"})
  public void testDefaultCtor() {
      WordList wl = new WordList();
      assert wl.size() == 0;
      // Make sure internal set was initialized.
      assert !wl.contains("random");
  }
  ```
  
  - `groups` is an array of `String` values
  - Can specify multiple groups:
    ```java
    @Test(groups = {"basic", "fileio"})
    ```

- To run tests in one or more groups:
  ```bash
  java ... org.testng.TestNG ... -groups basic fileio
  ```
Negative Tests

- Tests should also exercise error handlers
  - Java methods indicate errors by throwing an exception
- Create a test to verify that WordList constructor throws an exception when an invalid file is specified

```java
/** Verify behavior when a file is missing. */
@Test(groups={"fileio"},
    expectedExceptions={IOException.class})
public void testMissingFile() {
    File f = new File("missing.txt");
    assert !f.exists();
    WordList wl = new WordList(f);
}
```
- Test is marked as a failure if no exception is thrown, or if a non-matching exception is thrown
This Week’s Assignment

- Create a BoggleBoard class for storing the board state
  - Support N×N grids, not just 4×4
  - Populate board with strings containing A..Z, or Qu
- Question: How to generate random letters?
- Java has java.util.Random class for generating random numbers
  - Lots of different methods!
  - public int nextInt(int n)
    - Generates an integer value in range [0, n)
Generating Random Letters

- Can generate random numbers in range \([0, 26)\)
  - How do we turn these into letters of the alphabet?

Some ideas:

- Populate an `ArrayList<Character>` with all 26 character values
  - Use random numbers to index into the collection
- Compute the value directly:
  - `char ch = (char) (65 + rand.nextInt(26));`
    - What does the 65 mean?!
  - `char ch = (char) ('A' + rand.nextInt(26));`
    - Always use a character literal instead of the numeric code!
Generating Random Letters (2)

- Why is the char-cast outside the expression?
  ```java
  char ch = (char) ('A' + rand.nextInt(26));
  ```
  - What’s wrong with:
    ```java
    char ch = 'A' + (char) rand.nextInt(26);
    ```
- In Java, result of + is going to be one of:
  - `double, float, long, or int`
  - For our case: `char + char = int`
Java Arithmetic Casting Rules

- From the Java language spec, section 5.6.2:
  - If either operand is of type double, the other is converted to double.
  - Otherwise, if either operand is of type float, the other is converted to float.
  - Otherwise, if either operand is of type long, the other is converted to long.
  - Otherwise, both operands are converted to type int.

- Specifically, these rules are used for Java arithmetic operators
  - Keep this in mind when writing mixed-type expressions…
Besides creating the Boggle-board class, also need to create a test suite for your code

- For `WordList`, create `TestWordList`
- For `BoggleBoard`, create `TestBoggleBoard`

Use TestNG annotations and test-harness to run your tests

Make sure your test suite is complete!
Make sure your code passes all tests!