Assertions, Testing and Debugging

Csc 115 Fall 2002

Assertions

Review -- Assertions

- An *assertion* is the statement of a fact that should be true at a given point in the execution of a program
  - assertions can be written as comments, to document the code:

```
count--;
// assert:  count >= 0
```

  - they can be written as code, to verify assumptions at runtime:

```
count--;
if (!(count >= 0)) throw new AssertionFailure();
```

- An assertion at the beginning of a method is called a *precondition*
  - it will often validate the method's arguments
- An assertion at the end of a method is called a *postcondition*
  - it will often validate the method's work and/or result
- When assertions are stated using a formal logical language, it's sometimes possible to prove a program's correctness; this is called *verification*

Assertions -- 2

- We can have executable and non-executable assertions:
- Non-executable assertions (quantifiers):
  - For all i in 0..len-1, A[I] > 0
  - There exists some element in A that is equal to x
  - Note: Quantifiers can't be directly specified in Java

Non-executable assertions as Specifications

- Specifications state what a program should do (not how it should do it)
- Especially needed when splitting up work between multiple programmers
- A specification can say at a high level what a program, down to the low level details in the program
- Can be formal or informal…. Informal ok for some applications, but formal specifications are required for safety critical applications
- Assertions can be used for writing specifications
- Assertions inside the code can also be used for verification

Executable assertions in Java

- Each assertion contains a boolean expression that you believe will be true when the assertion executes
- If it is not true, the system will throw an error
- By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behaviour of your program, increasing your confidence that the program is free of errors
- "assert" some boolean expression

  e.g. `assert n > 0;`

  e.g. `assert n > 0 : n; //` prints "n"

  same as: if (n<=0) throw new AssertionError(n);
- If the expression is false, an "AssertionError" will be thrown

Assertions in Java -- 2

- Use “assert” for:
  - *Internal variants* (example within an else statement, switch statements etc)
  - *Control-flow invariants* (when you think control shouldn’t reach a certain point in the program – but note you will get a compile time error if you do an assert at a spot that cannot be reached)
  - *Preconditions* – use for private methods to test arguments only but not for public methods!
  - *Postconditions* – use for both public and private methods after a computation for example

When should you not use Assertions in Java

- For argument checking of published specifications (public methods) – these specifications must be obeyed whether assertion checking is enabled or not (exceptions should be defined in this case)
- Do not use assertions to do any work that your application requires for correct operation, e.g.:

```
// Broken! - action is contained in assertion
assert names.remove(null);
```

Enabling and Disabling Assertion Checking in Java

- By default, assertions are disabled at runtime. Two command-line switches allow you to selectively enable or disable assertions.
- To enable assertions at various granularities, use the -enableassertions, or -ea, switch. To disable assertions at various granularities, use the -disableassertions, or -da, switch. You can specify the granularity with the arguments that you provide to the switch.
- You should turn them off when you are done debugging for efficiency reasons.

- Notes:
  - Assert is only available in Java 1.4 as part of the language specification
  - See this link for more information: http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html

Debugging and Testing

Testing & Debugging (review….)

- Testing: Process of verifying the correctness of a program and for identifying bugs

- Debugging: Tracking the execution of the program and discovering errors in it (can use a debugger or print statements)

- Note: it is impossible to test all possible inputs! But we can test a representative subset of inputs, and can make sure we test each method (or statement) at least once

Strategies for Unit Testing

- *Unit testing*
  - The practice of testing a single method or class, separately from the overall program in which it is used
- Important things to test for
  - API of a class (methods, parameters)
  - Proper initialization of fields
  - Boundary conditions (e.g., array bounds, off by one)
  - Error conditions
  - Execution paths (*statement coverage*)
- Using **`println()`** and **`toString()`** for debugging purposes
  - Design this ability in from the start like other requirements
  - Write/override the **`toString()`** method for each class
  - You can then print before-after pictures in your test code
- Code inspection and code walk-throughs

Bottom-Up Testing

- Why is bottom-up testing a useful approach?
  - the smaller the piece of code being tested, the easier it is to locate and fix bugs
  - if the code being tested has dependencies, those dependencies are also tested
  - so start at the bottom, with the smallest possible modules and fewest dependencies
- Classes are tested from the bottom to the top of the class hierarchy
- If a group of modules forms a dependency cycle, you can only test the cluster as a whole—so avoid creating dependency cycles!

> Bottom-Up Testing Order Principle:
> Whenever possible, before testing a given method X, test all methods that X calls or that prepare data that X uses.

Top-Down Testing

- Why would you want to do this?
  - when working in a team, layers are often implemented in parallel
  - A component may depend on others that aren't available yet
  - Don't wait for others before starting testing; use stubs
- How to do it?
  - *stub out* any dependencies of your component: fake realistic results with a minimum of effort
  - the stub might:
    - return a very small number of hard-coded items
    - only be able to deal with your specific test data

- Stubbing out components can also be useful in breaking dependency cycles, allowing the co-dependent components to be tested individually

Integration, Acceptance and Regression Testing

- When unit testing is complete, you must test the interactions of the classes with *integration testing*

- When the project is complete, you often have to run a final *acceptance test* before the customer officially accepts your work

- Once a system is released into service, it enters the *maintenance phase* of its lifecycle
  - in this phase, more bugs are discovered and fixed, and new features added
  - as things change, you want to do *regression testing* to make sure that the changes don't break previously working code
  - it's useful to have a suite of test drivers that can automatically run all unit and integration tests, and report on the results
  - note: it's very common for fixes or upgrades to interfere with seemingly unrelated code

White box and Black box Testing

- Black box testing:
  - Think of method or program as a "black box" that does a job.
  - You don't know how it works, just have the specification which specifies the inputs and expected outputs (says nothing about how it does what it does)
  - Pick test cases to see if performance matches specification – pick boundary cases and wide range of inputs

- White box testing:
  - In this approach, look at the actual code and consider how it works
  - Make sure each part of the code is exercised by your test cases.
  - Care about test case “coverage”

Debugging

- 3 main types of errors:
  - Compile time errors (program won't run)
  - Run time errors (program partially runs, but crashes)
  - Logical errors (program runs, but the result is incorrect)

Compile time errors

- Sometimes the problem is on the current line that the compiler indicates, but often it is on a previous line….
- Two possible kinds of compile time errors are:
  - Syntax errors (error in format, such as unbalanced parenthesis, missing semicolon)
  - Semantic errors (undeclared variable, wrong parameter type for method)

- Advice:
  - Just fix first syntax error caught and recompile
  - For semantic errors, carefully read message; use better tools
  - Carefully write code, understand what you are doing

CompileErrors.java

Run time errors

- It can be hard to determine where a run time error occurs....
- Can be found by using println's or a debugger to see when the crash happens
- Often have a hypothesis about what caused the error, start there
- Presentation of output is very important if you are to make sense of it
- Makes sense to have such output be reusable for when you need to make further modifications

Permute.java

- Common causes of run time errors:
  - Null pointer
  - Index array out of bounds
  - Divide by zero

RuntimeErrors.java

Logical errors

- Finding logical errors is a similar process to finding runtime error…
- Narrow down search, do lots of inspection, try different input values to see which ones pass/fail
- More strategic inspection of intermediate values
- Some things to watch out for:
  - Copying references instead of objects
  - Creating references but not objects
  - '==' on object references
  - putting a return type on a constructor (compile time error)

General Debugging Advice

- Test as you write your program
- Revisit previous assumptions and assume nothing….
- Walk through some examples – the devil is often in the details
- Explain the program to a friend (or your teddybear!)
- Use good style and comments
- Try adding debug output (println, toString calls)
  - ✍ Try narrowing down your search
  - ✍ Print intermediate results, more and more detail as you go
- Consider creating logfiles to help record errors during active use

How to avoid or reduce errors

- Simple interfaces – allows for easier isolation of errors
- Unit test, bottom up testing
- Make it easier to test when the program is changed
- Make use of exceptions and provide appropriate error messages
- Make use of assertions which will help detect errors before the program crashes or the results are not what are expected
- When you do discover a bug, look for similar bugs elsewhere in the program
- Always keep a “clean copy” of your program (use version control)

Debugging tools

- Help you quickly locate crashes
- Procedure tracing facilities
- Step through complicated logic
- No need to change source code

But:

- Tracing can be slow and tedious
- Data structures are typically hard to display with debuggers
- Debugger may alter program execution (bugs sometimes hard to reproduce when you use a debugger, subtle!)
- Not available in all environments….
- Let's look at the debugger in Eclipse….