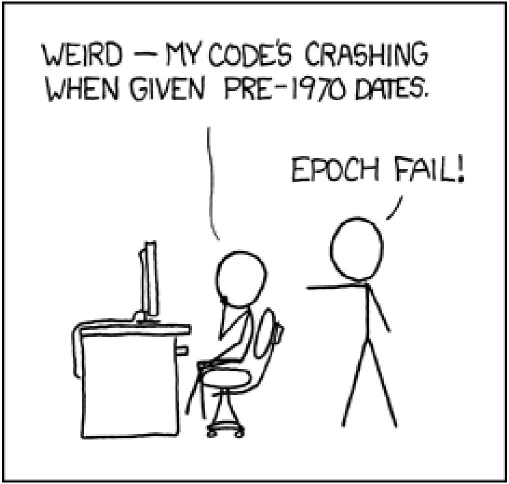
Debugging

Christopher Simpkins
chris.simpkins@gatech.edu

Bug



http://xkcd.com/376/

Program Errors

Three kinds of program errors, or bugs:

- Compile-time errors - compiler reports errors and does not produce a .class file
- Runtime errors - caught by the java runtime
- Semantic errors - program doesn't do what you expected it to do

BTW, why do we call program errors "bugs?"

Compile-time Errors

Java catches many kinds of errors at compile-time, including:

- Syntax errors - missing semicolons, parenthesis,
- File name conventions - file name doesn't match name of top-level class in file
- Type compatibility - a value is assigned to a variable that is not type compatible
- Name resolution - program refers to a name that is not in scope
- Method parameter matching - passing the wrong number or type of arguments to a method

Runtime Errors

A program can compile successfully and fail at runtime.

- Invalid casts - casting a value to an incompatible type
- Array index out of bounds - referencing an array element with a negative index or an index $\geq$ the array length
- `NullPointerException` - calling a method or accessing an instance member on an object reference that is `null`

Finding Errors

The process of finding errors is called **debugging**.

- Fixing compile-time errors is largely a matter of knowing the language and having a keen eye for typos
- Simple runtime errors, like array indexes out of bounds, are caught and reported with their precise location
- Semantic errors (which sometimes manifest as runtime errors) require a great deal of patience, detective work, and a toolbag of debugging techniques to find and fix

Finding semantic errors comprises the majority of debugging activity.

Debugging Techniques

Eliminating compile errors is usually called "getting the code to compile." When we talk about debugging we usually mean finding semantic errors. Some techniques:

- Tracing and watching - the biggest hammer in the debugging tool bag
- Assertions
- Explaining to a colleague, or "rubber ducking"[1]

[1] http://c2.com/cgi/wiki?RubberDucking, http://en.wikipedia.org/wiki/Rubber_duck_debugging

Tracing and Watching

Tracing the flow of execution of code can help to locate a bug.

- Manual: Print statements and logging give a play-by-play report of a program run
- Debugger: Breakpoints in a debugger allow you to step through a running program one statement at a time

Tracing is usually combined with watching the values of variables as the program runs

- Manual: Print statements and logging include values of variables of interest
- Debugger: variables can be designated as “watch” variables and be displayed separately while the program runs

Let’s try some of these techniques on Bugs.java

Assertions

Assertions are statements about program conditions that should be true at a given point of program execution.

```
assert condition;
```

- `condition` is any boolean expression
- Normally, assertions are not checked
- To hava `java` check assertions, run program with `enableassertions` switch
- A more robust version of `#IFDEF DEBUG` from the old C days

With `enableassertions` switch, when some `condition` in an assertion is `false`, the program terminates with a `java.lang.AssertionError`.

Assertions are useful for checking semantic errors in your programs, but require some effort.

Insertion Sort

The insertion sort algorithm in pseudocode (from CLRS Chapter 2):

```
for j = 2 to A.length // A[1 .. A.length] is an array
    key = A[j]
    // Insert A[j] into the sorted sequence A[i .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
   A[i + 1] = key
```

Insertion sort implemented in Java:

```
for (int j = 1; j < a.length; ++j) {
    int key = a[j];
    int i = j - 1;
    while(i >= 0 \&\& a[i] > key) {
        a[i + 1] = a[i];
        i = i - 1;
    }
    a[i + 1] = key;
}
```

Loop Invariants

A loop invariant expresses a formal property of an algorithm that:

- is true prior to the first iteration of the loop,
- if it is true before an iteration of the loop remains true before the next iteration, and
- upon loop termination gives a useful property that helps show that the algorithm is correct.

A Loop Invariant for Insertion Sort

```
1 for j = 2 to A.length
2     key = A[j]
3     // Insert A[j] into the sorted sequence A[i .. j - 1].
4     i = j - 1
5     while i > 0 and A[i] > key
6         A[i + 1] = A[i]
7         i = i + 1
8    A[i + 1] = key
```

At the start of each iteration of the for loop of lines 1-8, the subarray A[1 .. j - 1] consists of the elements originally in A[i .. j - 1], but in sorted order.

Expresing Loop Invariants as `assert`ions

Translating the assertion condition is easy. The trick is figuring out where to put it.

```
for (int j = 1; j < a.length; ++j) {
    assert isSorted(a, 0, j - 1);
    int key = a[j];
    int i = j - 1;
    while(i >= 0 \&\& a[i] > key) {
        a[i + 1] = a[i];
        i = i - 1;
    }
    a[i + 1] = key;
}
```

Note that we didn't express the entire invariant in Java. We could, but you must trade off implementation effort and benefit. Run the program with the `-ea` switch to enable assertions:

```
$ java -ea InsertionSort
```

See InsertionSort.java.

Avoiding Bugs

There are two ways to write error-free programs; only the third one works. – Alan Perlis

- Defensive programming - validate input, check array bounds, check for nulls, use checked expections
- Incremental development - develop program in small pieces, test peices individually before combining
- Code review/pair programming - have another set of eyes on your code

Let's add some defensive code to Bugs.java

Closing Thoughts

- Computers are compliant and finicky
- Debugging is an art and a science
- Think like a detective
- Code defensively