C Finding and Fixing Bugs

C.1 Introduction

As you will quickly find the BUG is the pain of all programmers existence. This section looks at the most common types of BUGS and some of the strategies for finding and fixing them.

Remember: Computer are inanimate pieces of electronics with no inherent intelligence or malice; they do "exactly" what *you* tell them to do, *wrong though it may be*! Both JAVA and UNIX operate on the YAFI-YOGI (**Y**ou **a**sked **f**or **i**t - **Yo**u **g**ot **i**t) principle, which means that a non-working program means *you* have put a mistake into it.

C.2 Types of "Bugs"

Basically there are three types of BUGS, these being:

Syntax Errors: You have a mistake in the syntax of a statement in the JAVA code. Such errors usually mean that no `class` file is produced, or if it is produced it is very unlikely to work correctly. Then you compile your program, with `javac`, you will get error messages and the "line number" where the compiler first noticed the errors. The most common errors are:

1. Forgetting to declare a variable, or miss-spelling the name of the variable either in the declaration or when used in the code. You will get an error message of the form:

   ```
   CheckPoint_4.java:10: cannot resolve symbol
   symbol  : variable xValeu
   location: class IfTry
         if (xValeu > 5) {
             ^
   ```

   This means at line 10 of file `CheckPoint_4.java` the compiler found a variable called `xValeu` which it was not expecting to find. It also shows you the line where the error occured.

2. Getting the name or calling parameter of a *method* wrong will result

   ```
   CheckPoint_4.java:10: cannot resolve symbol
   symbol  : method setColour  (int)
   location: class cplab.DataSet
   data.setColour(3);
         ^
   ```

   Remember correct spelling is **vital**, also if you supply the wrong parameters the compiler will tell you that it `cannot resolve symbol` since the parameter type list is part of the definition of the method.

3. Forgetting the "`;`" at the end of a statement. You will typically get as error message of the form:

```
IfTry.java:17: ';' expected
System.out.println("Value of x is :" + xValue)
                                              ^
```

with the ^ "showing" you where the compiler *thinks* the `;` *should* go. Remember it is the compilers "guess" and it may not be correct!

4. Forgetting to close the "`" "`" round a string, the "`{ }`" round a group of statements, or the "`/* */`" round comments. You will get a variety of odd errors, most of which make sensible suggestions as to what is wrong. Most of these problem are picked up by the colour highlighting in `emacs`.

5. Forgetting an `import` file at the top of the program. For example if you try and use `Graph` object without including `gov.noaa.pmel.sgt.cplab.*` you will get:

```
CheckPoint_4.java:8: cannot resolve symbol
symbol  : class SimpleGraph
location: class CheckPoint_4
    SimpleGraph graph = new SimpleGraph();
    ^
```

which means the compiler was not expecting `SimpleGraph` since you forgot to tell it to look in `...cplab.*`.

6. General syntax errors in statements, for example in arithmetic statements, assignments or loops. Again the compiler will give you the line number where the error occurs.

Fix syntax errors **one at a time**, starting with the *first* error detected. Remember that one *simple* error, for example a missed "}" can result in dozens, sometimes hundreds, of totally spurious, errors message from other parts of the program that are actually perfectly correct!

Runtime Errors: These occur when the program *compile* successfully but when you try and run it, it either *crashes* with an *Exception* or in the worst case does not appear to do anything at all!

There are a vast range of possible *Expections*, however at this level of programming the most likely are:

1. `NumberFormatException`, normally combined with

```
at java.lang.FloatingDecimal.readJavaFormatString...
```

means you have tried to read a number from a `String` that contained something other that a sensible number. When using the `Display` class, this means that you types *rubbish* into a prompt field that was expecting an `int` or a `double`.

2. `ArrayIndexOutOfBoundsException`: Your program has tried to access "off the beginning or end" or an array. Again the line number will tell you where this happened.

3. *Floating Point Errors*: Your program has tried to created a `double` number greater than 10^{308}. This usually means you have tried to divide by *zero* (or a very small number), *or* you have gone round a loop far more times than you wanted and some number has got "very big". The `double` gets set the `NaN` or `Infinity` but the program **continues**, producing rubbish.

4. *Nothing Happens*: Your program is either:

   (a) Waiting for input, Look for `.waitForButtonPress()` statements in the wrong place, or multiple `Display` panels when you only wanted one.

   (b) Charging aimlessly, and non-productively round a infinite loop without ever getting to its termination condition.

   To stop the program type `Ctrl-C` (both keys at once).

   The debugging strategy depends on the complexity of the program. For the type of programs you are writing at the moment usually "reading through" the code is enough to spot the mistake. The most common ones are:

   (a) Forgetting to assign a value to a variable, which defaults to zero, then using it assuming it has a non-zero value. *Overflows* and *Infinity loops*.

   (b) Miss-placed "`;`" in loop, for example

```
int x = 1;
while ( x < 100) ;
   {
      < rest of loop>;
   }
```

   will loop *infinitely* since the miss-placed "`;`" means the `while` loop has a single null instruction as the loop body, and *not* the section in "{ }" as you expect!

   (c) Forgetting to update the loop variable.

   (d) Using "==" test on `float` numbers which, due to rounding errors, is *always* `false`. Note:

```
boolean testValue = Math.sqrt(2.0)*Math.sqrt(2.0) == 2.0;
```

   will set `testValue` to `false` since not all 64 bits of `Math.sqrt(2.0)*Math.sqrt(2.0)` will be *identical* to `2.0`.

   If "reading the code" fails, then there are two strategies, these being:

   (a) Insert `println()` statement to check the value of key variables in the program. This is the "old-way" but is simple and valuable for finding errors in simple programs.

   (b) Use an *interactive debugger* or profiler. At present line debuggers for JAVA are very clumberson and complex, not a sensible strategy for programs at this level (yet)?

Working Program – Wrong Results: This is the real *fun* one! If the program produces:

1. UTTER RUBBISH: For example a CONSTANT no matter what input you supply. The most likely problem is a program bug as discussed above. For example forgetting to assign a value to a variable, mistake in a loop which means that it never get executed etc. Search for bugs as described in the *Runtime Errors* above.

2. ALMOST RIGHT RESULTS: For example the right results with some data, wrong for others. This is not likely to be simple coding error, much more likely to be an error in the logic of the code, for example some conditional statement is wrong. This is tough to find.

   There is also the worse-case synario, you have the underlying mathematics and/or physics of the calculation wrong. Then no matter how much you "play" with the program it will still *always* produce wrong results!

Finally when you do make changes to the source code with `emacs` remember to use SAVE BUFFER to update the source file on disk *and compile* the modified code with `javac`. I would prefer not to think about how many times I have failed to do one or other of these!

C.3 Problems with the Systems

Big computer system do sometimes "go wrong". This usual symptom is they become *very slow* to respond or *stop* responding all together. This can be caused by anything from too many people trying to run big programs to a hardware fault on the server. If this happens:

1. Stop trying to do anything. Trying to open a new window or "playing" with the mouse will only make things worse. If however you are running a program that may have "looped" it may be *you* causing the problem. Try stopping it with `Ctrl-C` as discussed above or using the *Kill* option to stop the terminal window.

2. If nothing improves within about 30 secs:

   (a) Call a demonstrator or see Mrs McIvor in the computing office for help.

   (b) If there is nobody else the laboratory try and *Exit* the Window Manager, (try and log-off). If this also fails, leave the system alone, **do not** switch-off but leave a note on the terminal and/or see Mrs McIvor as soon as possible.

C.4 Myths, General Miss-conceptions and Classic Excuses

Round computing there are a whole series of *myths*, *miss-conceptions* and *total rubbish*, a few of which are:

1. The computer does "*what you tell it*" it is a deterministic machine *without* intelligence. It does *not* have a personal vendetta against you, despite what you may feel when all the other students programs run perfectly and yours completely fails! Comments like "*it does not like me*" will be treated with scorn and ridicule!

2. Repeating the same *compile* or *execute* many times "*in case the computer made a mistake*" just wastes your time and add to your frustration and stress levels. You *will* get the same answer each time. Your program will *not* magically start working.

3. The chance of the "*computer being wrong*", that is *you* finding a mistake in the *compiler* or system is about the same as you been struck by lightening on the way home! All non-trivial pieces of software do have "bugs" but the bugs in the compiler are likely to be so obscure that none of the simple program that you will write will show them up.

4. The *PC urban myth* of "*the program getting corrupted on disk*" would result on the whole computer system crashing with lots of "panic" messages. It would *not* just effect your program. This problem does occur with Windows-95/98/2000 but not on UNIX.

5. The "*the computer lost my program*" is just possible, but it would not loose *just* yours. Lost files mean a computer hard disc fault which typically looses many files with lots of "panic" message. In this unlikely event we are able to "restore" your files to the state they were in "yesterday evening". This has only happened once in 8 years of running the CP-Lab, so it is very unlikely. If a file vanishes from the system it means *you deleted it*.

6. The "*somebody hacked into my account and changed my files*" is very unlikely *unless* you gave them your password; in which case tough! There is a finite chance of a machine being "hacked into" but if somebody does succeed in doing this is is rather unlikely they would modify one persons file when there is a whole system to muck up!

Experimental observation of the probability of "*occurances*" 4,5, & 6 suggest a relation of the form $\exp(-\Delta T^2)$ where ΔT is the time (in days) left before a programming project deadline is due!