By anyone's account the JVM and the class file format have been a raving success. Considering the number of languages that have targeted to this platform beyond Java shows how resilient and powerful it is.

The first and foremost job of a JVM and Class file format of course is execution, and certainly the designers of Java got a lot right. But execution isn't the whole story. As time has past it has become apparent that other criteria are important to a pairing like this.

One such item is class file inspection. Whether it be for AOP, static code analysis for bugs, or dynamic analysis for performance, the ability for automated processes to inspect the code is important as well.

For the most part, the class file format is easy traversed. Tools like ASM, BCEL have done a lot of work doing just that. But there are a few areas that make code inspection difficult. This blog will show some of these problem areas.

First is the **switch** statement.

There are two opcodes for switch statements in the class file format:

LOOKUPSWITCH - used when the various case's are not particularly contiguous.
TABLESWITCH - for when the cases are contiguous and a jump table can be used.

Each suffer the same flaw however, that being there's no quaranteed way to determine where the end of the last case statement is, or whether or not there is a default case. In some cases you can determine this empirically, by looking for absolute downward branching GOTO statements in other case blocks, and if they exist, that is the first statement after the switch. However if all other case statements either return or throw exceptions, you have no way of knowing where that end is.

Second is the **trinary** statement

Often you try to emulate the state of the operand stack as you navigate a method. In most cases, you can follow the flow of instructions top-down through a method call, and process statements as they arrive without much concern for branching. This is because the compiler behaves respectfully most of the time with regards to the stack and branching. In most cases, the class file format will not process a branch with anything on the stack. The only deviation from this is the trinary. With it you will see:

```
iload_1

ifeq8

iconst_0

goto9

iconst_1

istore_2
```

for the following code
```
int a = b ? 0 : 1;
```

Here there is a goto 9 with the stack still populated from the iconst 0. In my experience this is the only case where branches occur with a populated stack. The 'simple' solution is to assume you are in a trinary when this occurs, but really that's just a hack, and likely to break sometime down the road.

Third is the **try/catch/finally** mechanism.

There is no opcode support for try/catch/finally. This is fine, because try/catch/finally blocks are documented in code attributes of the class file format that defines the various instruction offsets where these blocks occur.

The attributes, as viewed by javap, will look like this:

```
  Exception table:
   from   to  target type
     2     6     9   Class java/lang/Exception
     2    13    18   any
```

The first line documents a try/catch block that catches an Exception object, the second line documents a finally block for the above try block, as the 'type' value is 'any'.

The 'from' is the instruction offset where the try starts, the 'to' documents the end of the try block, and the 'target' documents where the exception handler occurs. Typically this address is where the exception object is stored in a local variable.

This works perfectly fine for executing a code block. But what is missing is critical for inspection. There is no documentation of where the catch block ends.

Again you can sometimes infer this. If there is a branch at the end of the try section (or a catch section), you can assume that this is the first address after the try/catch/finally block, but like the switch statement this is not always the case. With inspection you often want to know that all appropriate clean up happens in a finally block, but if you don't know where that ends, that makes that determination difficult.

Speaking of **try/catch blocks**, this is the fourth problem area.

Once upon a time, the java compiler relied on two opcodes to handle exception handling. the JSR/RET pair. This allowed code to implement mini procedures within a method. Catch blocks would issue JSR calls to switch to a catch handler, and the RET would return once that catch handler was done.

For a long time, now, these two operators have gone unused however, as 'modern' (post 1.3 ?) compilers no longer use them. Instead they inject catch blocks in normal code flow. But the compiler plays fast and loose with try/catch/finally blocks. Often the compiler will emit duplicate blocks of code in various parts of your methods to handle catch or finally blocks, not just where you defined them. Based on rules that i have not been able to determine, the same catch block you write, may show up two-three-four times in the same method. You will see this, if you compile your code with debugging options and look at the line number code attribute. The attribute documents the mapping between source lines and code byte offset. You would think there would be a one to one relationship, but this is not the case. You will see something like this:

```
LineNumberTable:
   line 13: 0
   line 15: 2
   line 16: 9
   line 17: 11
   line 19: 13
   line 18: 18
   line 19: 20
   line 20: 22
   line 19: 25
   line 21: 27
```

Notice that lines 19 is represented twice in the same byte code. That is because the compiler pushed a catch block up near where it would happen in the try block, and left the original one where it was.

I have seen the same line show up 3, 4, 5 times in different byte offsets. This makes inspecting code very difficult.

The fifth area is the handling of **longs and doubles**.

As you know, these types are 64 bits, and thus bigger than a 32 bit register. Of course the JVM works on 32 bit registers, and so special sauce has to be used to handle these 64 bit beasties.

If you define a method like

public void generate(int code, long value, String description)

the registers will be assigned as follows

```
R0 : 'this'
R1 : code
R2 and R3: value
R4: description
```

yes that's right, longs and doubles occupy pairs of registers. Therefore all operations dealing with these types push and pull 64 bits from the stack and into registers and back again.

something like this:

```
LLOAD 2
```

actually pulls the value out of register 2 and register 3 and treats them like a long, and puts them in succeeding spots on the stack.

There are all kinds of rules about splitting 64 bit values, so doing

```
LLOAD 2

ISTORE 1
```

will cause the JVM to puke in it's soup, because you are pushing a long from R2 and R3, and then trying to pull an int from the top of the stack into R1. But that int is half of a long, and a nono.

While i don't know how you would make this easier, it causes pain when parsing class files.

Which leads to some nasty opcodes available in the JVM,

```
DUP2_X1

DUP2_X2
```

These bad boys attempt to duplicate items on the stack and push them down several values into the stack. They are massively complicated to handle cases where longs and doubles are on the stack and handle them correctly. Fortunately they are so complicated that I have never seen the java compiler ever generate these opcodes, and hopefully we can just ignore them.

The six area isn't really a problem, just an **oddity**

```
BIPUSH

SIPUSH
```

push constant byte and short values on the stack. But there is no `CIPUSH`

or pushes for ints, floats, doubles, longs, etc, etc.

Seems odd? well, there's just another instruction for that

```
LDC
```

Use LDC for int, long, float, double

Seventh and finally is the **internal format of class names**.

All of the primitive types have single letter names for their types, so

float -> F
double -> D
int -> I

etc.

Classes on the other hand, have a fancier format

Ljava/lang/String;

That's an L followed by the fully qualified class name with slashes replacing periods, followed by a semicolon.

While it's a funny format, it's perfectly fine it would seem, as you just need a starting letter that doesn't conflict with the primitive types.

Hmmmmmm

long is ??????

```
long -> J
```

?????

Why?

Given that the choice was relatively arbitrary, why did they choose to use L as the start for a class? Given that long -> L would be an obvious mapping and given that these things are Classes, one would have thought that the class internal format would have been

```
Ojava/lang/String;
```

since String is an Object vs. a primitive (or perhaps T for type)

```
long -> L
```
and

But alas it's not. Doesn't cause any real problems, just is a real annoyance.

At this point, the chances of any of these items being fixed is next to nil. And all in all, the Class file format has been a big success, but it's just the small things that drive you crazy... well me anyway.