

Booleans and Bitmasks and Return Values

by Julian Robichaux, panagenda

originally published on socialbizug.org, March 2014

I usually try to steer clear of discussions over coding style, mostly because they tend to gear themselves around The One True Way to do a thing, and these tend to be generalizations at best, if not pure opinion and personal preference. Some popular subjects of rather heated debate include:

- No class method should ever have more than ## lines of code!
- I don't like the way you write loops!
- The RIGHT way to format code using brackets, linefeeds, and indents.
- My font is better than your font (neener neener).

Whatever. It's all part of the rich tapestry of life, I say.

However (you knew there would be a "however", didn't you?), in this article I will make the most gentle of suggestions for a possible way to structure your function/method calls. It's not a mandate, not even a "you're doing it wrong"... just another way of doing things that might be different from how you are currently doing it.

NOTE: I'll be doing the code examples in Java, but it's a short step from Java to whatever language you choose to use.

A Typical Method Signature

Let's use this as our example method signature:

```
public boolean doSomething(String s, boolean option1) {  
}
```

You pass the method a String and some sort of true/false flag for an option parameter, and it returns true or false (assumedly to indicate whether or not the method was successful or not... we will get that later).

So what happens if you decide that you need another option flag? Or the option can actually be one of three things instead of two? You might overload the method or change the signature to something like:

```
public boolean doSomething(String s, boolean option1, boolean option2)  
{  
}
```

Over time, you might end up with a few more options, and after a while you could end up calling the method like this:

```
if ( doSomething("foo", true, true, false, true, false) ) {  
    // I guess that's okay...
```

```
}
```

But then, well... what does it mean? It's really hard to keep up with what all those different boolean parameters and what they're supposed to stand for, and you get to a point where your code isn't really very readable. Where code isn't readable, bugs find cracks in the walls where they can sneak in without you knowing it.

UNTIL IT'S TOO LATE!!! (cue thunderstorm and dramatic music)

So, is there a better way to do it? Or at least a different way?

Integers Instead of Booleans

What if you write the method like this:

```
public boolean doSomething(String s, int options) {  
}
```

This way you could have as many options as you want, right? Or at least as many options as there are int values. (DANGER: if you think you might have more options than there are int values, you need to seriously rethink your code.)

When you call the code this way, you end up with:

```
if ( doSomething("foo", 3) ) {  
    // all right, option 3 is good...  
}
```

That looks nicer, but from a readability standpoint it's still kind of hard to understand because, well, what is option 3 anyway? Which brings us to:

Using Constants for Integer Value Flags

You can make your code even more readable like this:

```
public static final int OPTION_PLAIN = 0;  
public static final int OPTION_BOLD = 1;  
public static final int OPTION_ITALIC = 2;  
public static final int OPTION_BLINKING = 3;  
  
public boolean doSomething(String s, int options) {  
}
```

So that your method call becomes:

```
if ( doSomething("foo", OPTION_BLINKING) ) {  
    // make it blink like a boss  
}
```

That's exactly the same call as we made before, but now we know exactly what the option means (as long as the names are meaningful, of course). The code is readable! And flexible, because you can always add more options. It also has the added advantage of:

- Fewer comments are required to explain what the heck you're doing
- If it's a public method, the API makes more sense

It's all looking good. Some might even say "better", but I'm not going to judge.

One other thought, though. What if you want to use OPTION_BOLD and OPTION_BLINKING together? Darn it, you're back to the problem of adding more and more parameters. Unless...

Using Bitmasks for Integer Value Flags

Let's try this on for size:

```
public static final int OPTION_PLAIN = 1 << 0;           // 0001
public static final int OPTION_BOLD = 1 << 1;             // 0010
public static final int OPTION_ITALIC = 1 << 2;            // 0100
public static final int OPTION_BLINKING = 1 << 3;          // 1000

public boolean doSomething(String s, int options) { }
```

Look at that, we're working in binary! Well, technically we're always working in binary on a computer whether we know it or not, but this time we're trying to.

In Java, `<<` is the left-shift operator, which means it shifts all the bits over a certain number of places to the left. `1 << 2` takes the number 1 and moves it 2 bits to the left, so that `0001` becomes `0100`.

If that makes you a little cross-eyed, you can either do some searches for "bit shift" or you can just trust what you see and keep on going. Don't worry too much about it, as long as you stick to the `1 << y` notation you shouldn't get in too much trouble.

So what's the advantage of that? Using this format you can easily combine options together using a logical "or", like this:

```
if ( doSomething("foo", OPTION_BOLD | OPTION_BLINKING) ) {
    // make it blink like a boss
}
```

Inside your method code, you can determine which options were passed like this:

```
public boolean doSomething(String s, int options) {
    if ( (options & OPTION_PLAIN) != 0 ) {
        // one of the options was plain
    }
    if ( (options & OPTION_BOLD) != 0 ) {
        // one of the options was bold
    }
}
```

```
// etc.  
}
```

Using binary flags, you can pass in multiple options using a logical “or” and read multiple options using a logical “and”. This is called bitmasking, and you’ve probably seen and used it lots of times in other people’s code. It’s very easy to implement in your own code too.

A few things to watch out for:

- You shouldn’t normally use zero as a possible value, because you can’t mask off a zero value when combined with other values
- You are limited to a relatively small number of options for a given situation, depending on your data type (31 options for a Java int, 63 options for a Java long, etc.). Normally that should be plenty of options though.
- Be careful of how the language you work in casts numbers if you’re going to shift beyond the maximum integer value. For example, in Java the expressions `(1 << 42)` and `(1L << 42)` will give very different results. Try it!

Boolean Return Values

I mentioned earlier that I would discuss the use of a boolean return value. Here’s something to chew on as far as that goes — again, I’m not making up rules or anything, just offering up some thoughts.

It’s pretty common to have a method return “true” if the code runs to completion properly, and “false” if it failed. In the case of big problems, Java code generally throws Exceptions instead (another subject of lengthy debate for some people, and one that I’ll stay out of).

But what if you want to be a little more granular about what happened when a method returns? Perhaps your method presents the user with yes/no/cancel options? Or it creates a new file and you want to indicate if it created a brand new file, overwrote an existing file, or didn’t create a file at all (which may or may not be an Exception depending on what the method does).

In that case, you could use either an int or an Object as a return value:

```
public static final int RESULT_YES = 0;  
public static final int RESULT_NO = 1;  
public static final int RESULT_CANCEL = 2;  
  
public int doSomething(String s, int options) {  
}
```

Or:

```
public static final ResultObject RESULT_YES = new ResultObject("Yes",  
0);  
public static final ResultObject RESULT_NO = new ResultObject("No",  
1);
```

```
public static final ResultObject RESULT_CANCEL = new  
ResultObject("Cancel", 2);  
  
public ResultObject doSomething(String s, int options) {  
}
```

If you really want to be thorough, you could also make the ResultObject int values into constants, and turn the ResultObject Strings into localized Strings using properties files. And don't forget to write a good equals() method in the ResultObject class for comparisons.

In any case, you've probably seen and used this technique plenty of times before too, and maybe it can be useful in your code.

Conclusion(?)

Again, I'm only offering suggestions in this article, not hard-and-fast rules by any means. I certainly still use plenty of boolean values in my own code, and I don't see myself stopping any time soon. But it's good to step back sometimes and think about other ways to structure your code.

I will also spare you my thoughts on naming conventions for constant values, because I know that everyone has their own opinion on that too.

:-)