# Integer Overflow Lab

> The version of binarysearch that I wrote for the JDK contained the same bug. It was reported to Sun recently when it broke someone's program, after lying in wait for nine years or so.
>
> *-Joshua Block, Java Sun http://www.wikibooks.org*

## 1 Introduction

While abstraction provides a way for computer scientists to think of computers as a black box (in order to write secure code you must), there comes a point in every programmer's life where they must unleash their inner hacker and become more aware of their hardware. This lab will begin you on that journey by supplying you with the tools necessary to detect integer overflow, a bug that lurked in the JDK for over nine years.

During this lab you will:

1. Learn to convert from base 2 to base 10
2. Get an introduction to bit shifting
3. Learn Binary Arithmetic
4. Learn what an integer overflow is and be able to rectify it
5. Understand the dangers contained in type errors

# 2 Lab Exercises

## 2.2 Changing Bases

The number system that people most commonly use is called decimal, or base 10. Let's take the number 106. It is composed in the following way:

$$6 * 10^0 = 6$$

$$0 * 10^1 = 0$$

$$1 * 10^2 = 100$$

For reasons that will be explained shortly, we will write this as:

| $10^7$ | $10^6$ | $10^5$ | $10^4$ | $10^3$ | $10^2$ | $10^1$ | $10^0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 6 |

The number system that computers use, however, is called binary, or base 2. Let's take the binary number 01101010 and convert it to decimal. Its internal representation is the following:

| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |

And so we have:

$$0 * 2^0 = 0$$

$$1 * 2^1 = 2$$

$$0 * 2^2 = 0$$

$$1 * 2^3 = 8$$

$$0 * 2^4 = 0$$

$$1 * 2^5 = 32$$

$$1 * 2^6 = 64$$

a. Write a function `int BinToDec(string bin)` that takes as an argument a string of binary digits and returns its value as an integer (*hint: You may find the constant* `INT_MAX` *defined in* `limits.h` *to be of use*).

```
int BinToDec(string bin) {

}
```

b. The bitshift operator "b ≪ n" takes a bitfield b and shifts it n bits. For instance, 01101010 ≪ 2 yields 10101000 (notice that the leftmost bits are shifted off the end and the rightmost bits are padded with zeros). Rewrite BinToDec to use the left bitshift operator (Hint: convert 1 and 2 to binary. What is the difference between them?).

```
int BinToDec(string bin) {

}
```

## 2.3 Binary Addition

a. Adding numbers in binary notation is very similar to adding numbers in base 10. Let's consider the following binary summands:

```
    10010100
+   00101011
------------
```

Finding the solution in this case would be very simple because there are no "carries": the sum of the individual digits from each summand is less than or equal to 1. Therefore, each digit of the solution is simply the sum of the digits in each summand:

```
    10010100
+   00101011
------------
    10111111
```

A binary addition problem that requires carries is also very similar to such a problem in base 10. Consider the following addition problem in base 10:

```
     15
+    28
-------
```

Since the sum of 5 and 8 is 13, we keep the 3 in the first digit of the solution and "carry" the 1 to be used in the addition of the next digit:

```
     1
     15
+    28
-------
     43
```

In binary addition, the sum of 1 and 1 is 10 (since 10 is the binary representation of the base-10 number 2), so in the case below, we would keep the 0 in the first digit of the solution and "carry" the 1 to be used in the addition of the next digit:

```
          1
    00010111
+   00101011
------------
           0
```

Using this method, complete the following binary addition problem (and write the "carried" numbers above the top summand):

```
    10010111
+   00101011
------------
```

b. So far, we have only described the binary representation of positive integers. To get the binary representation of a negative `int`, we need to use the "two's complement". This means:

1. start with the positive int
2. flip every bit
3. add 1 to the result (using the method described in section 2.3 above).

To tell if an `int` is negative, check the most significant (leftmost) bit: if it is 1, the `int` is negative, otherwise it is positive. For example, let's find the binary representation of -7.

| | |
|---|---|
| Step (1) is to take 7 in binary: | `00000111` |
| Step (2) is to flip every bit: | `11111000` |
| Step (3) is to add 1 to the result of step (2): | `11111001` |

So, the binary representation of -7 is `11111001`. As it turns out, switching signs from negative to positive in this binary representation is the same method! Let's show steps 1-3 again, but let's start with -10 to get 10:

| | |
|---|---|
| Step (1) is to take -10 in binary: | `11110110` |
| Step (2) is to flip every bit: | `00001001` |
| Step (3) is to add 1 to the result of step (2): | `00001010` |

The convenience of this representation of negative ints is that binary addition requires no special cases: it is done in the same way as described in section 2.3! To show this, compute -8 + 10 in binary (and write the "carried" 1s):

```
    11111000
+   00001010
------------
```

(Your result should have 9 bits, and the leftmost bit should be 1.)

c. Since we've decided to represent `int`s with only 8 bits, our 9-bit solution will simply not fit in the space we've allocated in memory. This is handled by dropping the extra leftmost bits to satisfy our 8-bit limit. What are the 8 rightmost bits of the solution to part (b) above?

d. With this binary representation, the solution to -8 + 10 turns out to be 2, which is correct! (If you did not get this answer for parts (b) and (c), you should retry those two sections until reaching the correct solution.) Again, with this representation of negative ints, processors do not need to perform any strange methods to handle addition with negative numbers, so addition can still be performed as fast as possible.

The range of `int`s that can be represented with 8 bits is -128 (10000000) to 127 (01111111). Again, the fastest way to tell if an `int` is positive or negative using its binary representation is to check the leftmost bit: if the leftmost bit is 1, the number is negative, otherwise it is positive.

What happens when adding 100 + 100?

```
    01100100
+   01100100
------------
```

e. According to our 8-bit int representation, is this number positive or negative?

f. What is the value of this result, and why do you think this has happened?

This problem is called integer overflow. There are several ways to "overflow" the memory representation of an int, and we'll go over three of them: addition, multiplication, and truncation.

## 2.4 Integer Overflow by Addition

a. Consider the following code snippet:

```
int a = getInt();
int b = getInt();
int c = a + b;
```

For each of the scenarios below, answer the following questions: Can `c` "overflow"? If so, what will be the sign of `c`? If not, why will `c` not overflow?

1) `a` and `b` are both positive.

2) `a` is positive and `b` is negative.

3) `a` and `b` are both negative.

b. Given your answers to part (a), how could a programmer detect if the sum of two `int`s will overflow?

c. Write a method that takes two `int`s and returns true if their sum will overflow, returning false otherwise.

```
bool sumWillOverflow(int a, int b) {









}
```

## 2.4 Integer Overflow by Multiplication

a. Consider the following code snippet:

```
int a = getInt();
int b = getInt();
int c = a * b;
```

Can the signs of `a` or `b` determine if `c` will overflow? Why or why not?

b. In `limits.h`, the constant `INT_MAX` is equal to the maximum value that can be represented by an `int`, and the constant `INT_MIN` is equal to the minimum value. Consider the following code snippet:

```
int a = getInt();
int b = getInt();
int c = a * b;
```

If `a` and `b` have the same sign (both are positive or both are negative) and `c` is overflowed, then theoretically `c > INT_MAX`. Also, `c = abs(a) * abs(b)`. Use these two equations to solve for `abs(a)`. (show all of your steps. there should only be 2 or 3).

c. Consider the code snippet from part (b). If `a` and `b` have different signs (one is positive and the other is negative) and `c` is overflowed, then theoretically `c < INT_MIN`. Also, `c = -abs(a) * abs(b)`. Use these two equations to solve for `abs(a)` (show all of your steps. there should only be 2 or 3).

d. Given your answers to parts (a), (b) and (c), how could a programmer detect if the product of two `int`s will overflow?

e. Write a method that takes two `int`s and returns true if their product will overflow an `int`'s memory representation, returning false otherwise.

```
bool productWillOverflow(int a, int b) {

}
```

## 2.4 Ariana 5 Software Failure: Integer Overflow by Truncation

On June 4, 1996, the first test flight of the Ariane 5 rocket malfunctioned and exploded approximately 37 seconds after takeoff. The software converts 64-bit floating point numbers to a 16-bit integer representation of the horizontal bias. The software, written in Ada, was also used with Ariane 4 rocket subsystems but was not apparent then. Ariane 5's faster engines caused the 64-bit numbers to be larger than in the Ariane 4 (larger than a 16-bit integer can represent), triggering an overflow condition that crashed the flight computer. Let's look into integer overflow and how to detect/rectify it.

a. Consider the following code:

```
#include  <stdio.h>
using  namespace  std;

int main(int argc, char argv[]){
    float f = 4294968296.0;
    int i = f;
    cout << "f: " << f << endl;
    cout << "i: " << i << endl;
}
```

Try running this code. What gets printed? Why do you think this is?

b. Let's look at the binary representation of these numbers. Add the following code to the end of the `main` function above and run it again.

```
// You don't need to worry about how this works.
// Basically, this prints the binary
// representations of f and i.
bitset<64> fBits(f);
bitset<64> iBits(i);
cout << "f in binary: " << fBits << endl;
cout << "i in binary: " << iBits << endl;
```

Note: you will have to `#include <bitset>` to make the code run. What gets printed?

c. Given the answer to part (b) above, what is the relationship between the values of `i` and `f`?