

# Bonus Chapter 2: Twiddling Your Bits

---

## *In This Chapter*

- ✓ Getting into binary
- ✓ Using hex and octal literals
- ✓ Using bitwise operators
- ✓ Using shift operators

**B**it manipulation refers to the fine art of playing with the individual bits that make up data in a Java program. Java provides many features for playing with individual bits. Most of them apply to integer types, and they're used mostly with `int` values.

In this chapter, you find out about the binary counting system, hexadecimal, and octal, as well as some specific Java features for working with bits.

## *Counting by Ones*

*Binary* is a counting system that uses only two numerals: 0 and 1. In the decimal system to which most people are accustomed, you use ten numerals: 0 through 9. In an ordinary decimal number, such as 3,482, the rightmost digit represents ones; the next digit to the left, tens; the next, hundreds; the next, thousands; and so on. These digits represent powers of ten: first  $10^0$  (which is 1); next,  $10^1$  (10); then  $10^2$  (100); then  $10^3$  (1,000); and so on.

In binary, you have only two numerals rather than ten, which is why binary numbers look somewhat monotonous, as in 110011, 101111, and 100001.

The positions in a binary number (called *bits* rather than *digits*) represent powers of two rather than powers of ten: 1, 2, 4, 8, 16, 32, and so on. To figure the decimal value of a binary number, you multiply each bit by its corresponding power of two and then add the results. The decimal value of binary 10101, for example, is calculated as follows:

$$\begin{array}{r} 1 \times 2^0 = 1 \times 1 = 1 \\ + 0 \times 2^1 = 0 \times 2 = 0 \\ + 1 \times 2^2 = 1 \times 4 = 4 \\ + 0 \times 2^3 = 0 \times 8 = 0 \\ + 1 \times 2^4 = 1 \times 16 = +16 \\ \hline 21 \end{array}$$

Here are some of the more interesting characteristics of binary and how the system is similar to and differs from the decimal system:

- ◆ In decimal, the number of decimal places allotted for a number determines how large the number can be. If you allot six digits, for example, the largest number possible is 999,999. Because 0 is itself a number, however, a six-digit number can have any of 1 million different values.

Similarly, the number of bits allotted for a binary number determines how large that number can be. If you allot eight bits, the largest value that number can store is 1111111, which happens to be 255 in decimal.



- ◆ To quickly figure how many different values you can store in a binary number of a given length, use the number of bits as an exponent of two. An eight-bit binary number, for example, can hold  $2^8$  values. Because  $2^8$  is 256, an eight-bit number can have any of 256 different values — which is why a byte, which is 8 bits, can have 256 different values. A 32-bit number can hold 4,294,967,296 different values.
- ◆ This “powers of two” thing is why computers don’t use nice, even, round numbers in measuring such values as memory or disk space. A value of 1K, for example, is not an even 1,000 bytes — it’s 1,024 bytes because 1,024 is  $2^{10}$ . Similarly, 1MB is not an even 1,000,000 bytes, but rather is 1,048,576 bytes, which happens to be  $2^{20}$ .
- ◆ One basic test of computer nerddom is knowing your powers of two because they play such an important role in binary numbers. Just for the fun of it, but not because you really need to know, Table BC2-1 lists the powers of two up to 32.
- ◆ Table BC2-1 also shows the common shorthand notation for various powers of two. The abbreviation *K* represents  $2^{10}$  (1,024). The *M* in *MB* stands for  $2^{20}$ , or 1,024K, and the *G* in *GB* represents  $2^{30}$ , which is 1,024M. These shorthand notations don’t have anything to do with TCP/IP, but they’re commonly used for measuring computer disk and memory capacities, so I thought I’d throw them in at no charge because the table had extra room.

**Table BC2-1**

**Powers of Two**

<i>Power</i>	<i>Bytes</i>	<i>Kilobytes</i>	<i>Power</i>	<i>Bytes</i>	<i>K, MB, or GB</i>
$2^1$	2		$2^{17}$	131,072	128K
$2^2$	4		$2^{18}$	262,144	256K
$2^3$	8		$2^{19}$	524,288	512K

<i>Power</i>	<i>Bytes</i>	<i>Kilobytes</i>	<i>Power</i>	<i>Bytes</i>	<i>K, MB, or GB</i>
$2^4$	16		$2^{20}$	1,048,576	1MB
$2^5$	32		$2^{21}$	2,097,152	2MB
$2^6$	64		$2^{22}$	4,194,304	4MB
$2^7$	128		$2^{23}$	8,388,608	8MB
$2^8$	256		$2^{24}$	16,777,216	16MB
$2^9$	512		$2^{25}$	33,554,432	32MB
$2^{10}$	1,024	1K	$2^{26}$	67,108,864	64MB
$2^{11}$	2,048	2K	$2^{27}$	134,217,728	128MB
$2^{12}$	4,096	4K	$2^{28}$	268,435,456	256MB
$2^{13}$	8,192	8K	$2^{29}$	536,870,912	512MB
$2^{14}$	16,384	16K	$2^{30}$	1,073,741,824	1GB
$2^{15}$	32,768	32K	$2^{31}$	2,147,483,648	2GB
$2^{16}$	65,536	64K	$2^{32}$	4,294,967,296	4GB

## Complementing Your Twos



In Java, the first bit of any integer value is a *sign bit*, which determines whether the number is positive. If the sign bit is zero, the number is positive. If the sign bit is 1, the number is negative. For this reason, the largest value you can store in an integer variable is half the number of different values that can be stored. For example, Table BC2-1 shows that a 32-bit value can store values up to 4,293,967,296. But to allow for negative numbers, the maximum value of an `int` (which is a 32-bit number) is 2,147,483,647.

Java uses a technique called *two's complement* to efficiently handle negative values. Two's complement was a technique developed decades ago, in the early years of computers, because it drastically simplifies the work that needs to be done by the hardware for addition and subtraction.

The basic idea of two's complement is that to get the negative of any number, you invert all the bits that make up the number, and then add 1. For example, to represent  $-30$  in two's complement, you first need the binary equivalent for 30 (to keep the example simple, I use just eight bits):

```
00011101
```

Then you invert all the bits. The ones become zeros, and the zeros become ones:

```
11100010
```

Finally, you add 1:

```
  11100010
+   _____ 1
  11100011
```

If two's complement really works, you take that 11100011 result and apply two's complement to it again and get the original 00011101 value back:

```
  11100011    // -30
  00011100    // invert it
+   _____ 1    // add 1
  00011101    // it works!
```

It's a darn good thing the computer does all this for you. Personally, I think it's voodoo. But it works.

## So What the Hex Is Hex?

*Hex*, which is short for *hexadecimal*, is another counting system. Hex is *base 16*, which means that it uses 16 numerals rather than decimal's 10 numerals or binary's 2. In hexadecimal, the numerals are 0 through 9 and the letters A, B, C, D, E, and F. Hex numbers, therefore, often look like random conglomerates of letters and numerals, such as 1F00 and B23C.



The reason we humans count in base 10 is that we have ten fingers. If we had 16 fingers, you'd already be able to count in hex. And just think how much faster you could type. (They'd probably just get in the way if you play the piano, however.)

The beauty of hex is that, because 16 happens to be a power of two, hex turns out to be a form of shorthand for binary: Every hexadecimal digit represents four binary bits. Any binary number can therefore be written as a hexadecimal number with one-fourth as many digits. The binary number 1011010111011000, for example, can be written in hex as B5D0. Because hex is much more concise than binary, hex is more frequently used.

Converting between hex and binary is easy because each hex digit represents a group of four binary bits. Table BC2-2 shows you how to convert binary values to hex values.

<i>Hex</i>	<i>Binary</i>	<i>Hex</i>	<i>Binary</i>
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

You can create hexadecimal literals in Java by starting the literal with 0x, and then following it with one or more hexadecimal digits, as in this example:

```
int i = 0xFF00;
```

**Note:** You can write the digits A through F with upper- or lowercase letters.

## Using Octal

*Octal* — base 8 — is also commonly used as a shorthand for representing binary values. In octal, the digits can be from 0 to 7, and each group of three bits represents one digit. Table BC2-3 shows the conversion between binary and octal digits.

<i>Hex</i>	<i>Binary</i>
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

To write an octal literal in Java, just start the number with 0 and don't use any digits larger than 7. Here's an example:

```
int i = 062;
```

Here 062 is interpreted as an octal constant because it begins with a zero.



If you want to create an octal literal with a value of zero, you must use two zeros:

```
int i = 00;
```

This is the kind of question that's likely to show up on some Java certification test, even though it's completely insignificant because zero is zero no matter what counting system you're working in. Thus the following statements all assign exactly the same value to *i*:

```
i = 0;      // decimal zero
i = 0x0;    // hexadecimal zero
i = 00;     // octal zero
```

## *Printing Binary Data*

The `Integer` class has the following methods that convert an `int` value to a string that contains the binary, hexadecimal, and octal values of the `int`:

- ◆ `static String toBinaryString(int i)`: Converts the integer to a binary string.
- ◆ `static String toHexString(int i)`: Converts the integer to a hexadecimal string.
- ◆ `static String toOctalString(int i)`: Converts the integer to an octal string.

These methods are usually used along with `System.out.println` to print `int` values in binary, hex, or octal to help you test and debug bit-twiddling routines. Here's a snippet of code that shows how these methods work:

```
int i = 75;
System.out.println("Decimal value:\t"
    + Integer.toString(i));
System.out.println("Binary value:\t"
    + Integer.toBinaryString(i));
System.out.println("Hex value:\t"
    + Integer.toHexString(i));
System.out.println("Octal value:\t"
    + Integer.toOctalString(i));
```

Run these statements and the following output is displayed on the console:

```
Decimal value: 75
Binary value: 1001011
Hex value: 4b
Octal value: 113
```



These methods are especially useful when you're trying to debug a program that manipulates individual bits.

## Using Bitwise Operators

Java provides for special operators called *bitwise* operators that are designed to play with the individual bits of an integer. Table BC2-4 summarizes these operators and I describe them in the following sections.



Notice that like arithmetic operators, these operators also have assignment versions that perform the operation and assign the result back to the first operand.

Table BC2-4		Bitwise Operators	
<i>Operator</i>	<i>Assignment</i>	<i>Name</i>	<i>Description</i>
&	&=	AND	Compares the corresponding bits in the two operands and sets the corresponding bit in the result to 1 if both operand bits are 1. If one or both of the operand bits are 0, the result bit is set to 0.
	=	OR	Compares the corresponding bits in the two operands and sets the corresponding bit in the result to 1 if either of the operand bits are 1. If both of the operand bits are 0, the result bit is set to 0.
^	^=	Exclusive OR	Compares the corresponding bits in the two operands and sets the corresponding bit in the result to 1 if exactly one of the operand bits is 1. If both of the operand bits are 1 or both are 0, the result bit is set to 0.
~	~=	Complement	Inverts all the bits in the operand. Note that this is a unary operator.
<<	<<=	Shift left	Shifts each bit in the first operand left by the number of digits indicated by the second operand. Bits are lost from the left of the value, and any vacated bits on the right are filled in with zeros.

(continued)

Table BC2-4 (continued)

<i>Operator</i>	<i>Assignment</i>	<i>Name</i>	<i>Description</i>
>>	>>=	Shift right	Shifts each bit in the first operand right by the number of digits indicated by the second operand. Bits are lost from the right of the value, and any vacated bits on the left are filled in with the sign bit (the first bit of the original value).
>>>	>>>=	Shift right	Shifts each bit in the first operand right by (zero extension) the number of digits indicated by the second operand. Bits are lost from the right of the value, and any vacated bits on the left are filled in with zeros.

## Using AND

The AND operator (&) examines each corresponding bit in its two operands and sets that bit to 1 in the result value only if both of those bits in the operand values are 1. Here's a little snippet of code that demonstrates how this operator works in Java:

```
int i = 0x94;    // decimal 148
int j = 0xCD;   // decimal 205
int k = i & j;
System.out.println(Integer.toBinaryString(i));
System.out.println(Integer.toBinaryString(j));
System.out.println("-----");
System.out.println(Integer.toBinaryString(k));
```

Here's the console output from these statements:

```
10010100
11001101
-----
10000100
```

As you can see, only those bits that are 1 in both operands are set to 1 in the result.

One of the most common uses for AND is to find out whether a particular bit in an `int` variable is set. To do that, make the second operand be a value in which only the bit you're interested in is set to 1; all other bits are set to zero. This second operand is called a *mask* because it covers the values you aren't interested in.

For example, suppose you're writing a robot system that reads a 32-bit value from a sensor sub-system, and the third bit from the right in this value indicates whether the robot's eyes are closed. To find out if the robot's eyes are indeed closed, you could use a mask with the third bit from the right turned

on. The hex value for all zeros except that bit happens to be 0x04, so this code does the trick:

```
int robotSensor = readSensors();
int eyeMask = 0x00000004;
int eyeClosed = robotSensor & eyeMask;
if (eyeClosed > 0)
    System.out.println("The eyes are closed.");
else
    System.out.println("The eyes are open.");
```

Here I assume the method `readSensors` somehow gets the `int` value from the sensor sub-system. I then set up a mask in which the only bit that's 1 is bit 3, and use the `&` operator to AND this mask with the sensor values. The result is stored in `eyeClosed`, which is greater than zero if the third bit was set to 1.

The AND operator can also be used to set a specific bit in a value to 0. To do that, you set up a mask in which every bit is 1 except the one you want to set to 0. For example, the following statements set the robot sensor's eye bit to zero:

```
int eyeMask = 0xffffffffb;
robotSensor &= eyeMask;
```

Here the hex literal `0xffffffffb` sets every bit to 1 except the third from the right. Then the AND expression turns this bit off `robotSensor` value.

## Working with the binary Windows Calculator

The Calculator program that comes with all versions of Windows has a special Scientific mode that many users don't know about. When you flip the Calculator into this mode, you can do instant binary and decimal conversions, which can occasionally come in handy when you're working with bitwise operations.

To use the Windows Calculator in Scientific mode, launch the Calculator by choosing Start→All Programs→Accessories→Calculator. Then choose View→Scientific from the Calculator's menu. The Calculator changes to a fancy scientific model — the kind I paid \$200 for when I was in college. All kinds of buttons appear, enabling you to do far more than just

add, subtract, multiply, and divide. The scientific calculator also happens to be very useful for binary calculations.

You can click the Bin and Dec radio buttons to convert values between decimal and binary. For example, to find the binary equivalent of decimal 155, enter 155 and click the Bin radio button. The value in the display changes to 10111011.

The Scientific Calculator has several features that are designed specifically for binary calculations, including AND, OR, and XOR.

The Scientific Calculator can also handle hexadecimal conversions, which you'll find useful from time to time.

## *Using OR*

The OR operator (`|`) works like the AND operator, but sets a bit in the result value to 1 if either of the bits in the operands are 1. Here's a snippet that demonstrates how it works:

```
int i = 0x94;
int j = 0xCD;
int k = i | j;

System.out.println(Integer.toBinaryString(i));
System.out.println(Integer.toBinaryString(j));
System.out.println("-----");
System.out.println(Integer.toBinaryString(k));
```

Here's the output from this code:

```
10010100
11001101
-----
11011101
```

As you can see, any bit that's 1 in either operand is set to 1 in the result.

Like AND, OR is also often used with masks. But instead of finding out if a particular bit is set, OR lets you set a particular bit. For example, here's some code that sets the third bit of the robot sensor value to 1 so the robot closes his eyes:

```
int robotSensor = 0x00000000;
int eyeMask = 0x00000004;
robotSensor |= eyeMask;
```

This code starts by initializing the `robotSensor` value to all zeros. Then it creates a mask for the third bit from the right and uses the OR operator to set that bit in the `robotSensor` value.

## *Using XOR*

The exclusive-OR operator (`^`) sets a bit in the result to 1 if only one of the operands has that bit set to 1. If both operands have that bit set to 1 or 0, the corresponding result bit is set to 0. For example, consider these statements:

```
int i = 0x94;
int j = 0xCD;
int k = i ^ j;
```

Console output that displayed the binary values of these variables looks like this:

```
10010100
11001101
-----
01011001
```

As you can see, bits in the result are 1 only if the corresponding bits in the operands are different. If they are the same (both 0 or both 1), the bit is off. (Note that the `toBinaryString` method omits leading zeros, so the leading zero in the result isn't actually displayed. I put it back in so the bits line up.)

The XOR operator has an unusual capability: You can use it to swap two integer values without having to create a third integer. For example, consider this code:

```
i ^= j;
j ^= i;
i ^= j;
```

Here the values of `i` and `j` are exchanged. This way is a tad bit more efficient than the normal way to swap integers:

```
int temp = i;
i = j;
j = temp;
```

However, someone who isn't familiar with bitwise operators might not understand what these statements do. So I'd avoid this technique. (If you want a little mind-puzzle to occupy your spare time, see if you can figure out why this works.)

## The complement operator

The *complement operator* (`~`) doesn't just live to say nice things about you. (That would be the *compliment operator*.) Instead, it simply reverses all the bits in an integer. The 1s become 0s, and the 0s become 1s. For example:

```
int j = ~i;
```

If you're into technical stuff, you can use the complement operator to prove to yourself that the two's-complement thing (described earlier in "Complementing Your Twos") really does work. Because Java represents negative numbers by inverting the bits and adding 1, you can determine the negative of any number by using the complement operator, and then incrementing. Here's an example:

```
int num = 156;
System.out.println(num);
num = ~num;
num++;
System.out.println(num);
```



## 12 *Using Bitwise Operators*

---

Just to show that two's-complement really does work, here's the console output for these statements:

```
156
-156
```

I'll be a monkey's uncle.

### *Shifty operations*

The last set of bitwise operators are the *shift operators* (<<, >>, and >>>). They slide the bits of an integer left or right by a specified amount. For example:

```
int i = 0x94;
int j = i << 3;
System.out.println(Integer.toBinaryString(i));
System.out.println(Integer.toBinaryString(j));
```

Here the value of *i* is shifted to the left three bits. Here's the console output:

```
10010100
10010100000
```

Unfortunately, these numbers are left-aligned instead of right-aligned, but you can see that the three zeros have been added to the right of the number.



The interesting thing about shifting is that the value of an integer doubles each time you shift it one place to the left. For example, consider this code:

```
int a = 3;
for (int b = 0; b < 10; b++)
    System.out.println(a << b);
```

The console output for these statements is this:

```
3
6
12
24
48
96
192
384
768
1536
```

Similarly, each time you shift right, the value is cut in half:

```
int c = 512;
for (int b = 0; b < 10; b++)
    System.out.println(c >> b);
```

Here's the output from this code:

```
500
250
125
62
31
15
7
3
1
0
```

Note that integers aren't very good at division; the results are rounded down to integer values. Thus half of 125 is 62, and half of 31 is 15.



If you're an efficiency junkie, shift operations are a little more efficient than multiplications. Thus  $i = j \ll 2$  is more efficient than  $i = j * 2$ .



You can use a shift operator to build a method that replaces the `toBinaryString` method of the `Integer` class. The problem with `toBinaryString` is that it omits leading zeros. That results in binary values that don't line up on successive lines, as shown in several previous sections of this chapter. The following method solves that problem:

```
public static void printBinary(int val, int count)
{
    String s = "";
    int mask = 0x01;
    int bit = 0x00;

    for (int i = 1; i <= count; i++)
    {
        bit = val & mask;
        if (bit > 0)
            s = "1" + s;
        else
            s = "0" + s;
        mask = mask << 1;
    }
    System.out.println(s);
}
```

## 14 *Using Bitwise Operators*

---

This method accepts two parameters: an `int` value to be printed in binary, and another `int` that specifies how many bits you want to print. The method works by using a mask in a `for` loop to examine the value one bit at a time, starting with the rightmost bit. If that bit is one, then 1 is added to the start of the string to be printed. If the bit is zero, then 0 is added. The mask is shifted to the left by one bit each time through the loop.

Here's another version of the first example that was presented in this section, this time using the `printBinary` method:

```
int i = 0x94;
int j = i << 3;
printBinary(i, 16);
printBinary(j, 16);
```

And here's the resulting output:

```
0000000010010100
0000010010100000
```

This time, you can clearly see that the first value has been shifted left three bits in the second value.