DigiPen
INSTITUTE OF TECHNOLOGY

Chapter 13 | Bitwise Operators

Java AP

We've been using a lot of operators throughout this course but all worked on the whole of a variable.

Example	`public static void main(String[] args)`
	`{`
	`  int a = 4;`
	`  a += 5;`
	
	`  System.out.println(a);`
	`}`

Output	**9**

However, we can also represent the decimal value 4_{dec} as a binary number 00000100_{bin}. The binary representation shows the values of the individual bits of a number.
Operator that modify a number's bits are called ***bitwise operators***. For instance, we may want to set the first bit from **0** to **1** which leads to the following binary number $000001001_{bin}=5_{dec}$.

Note: Bitwise operators can be used with integral types only (I.E. int, long, short, etc...)

Signed binary numbers

Before going through the bitwise operators, let us learn a little about signed binary numbers and their history since we will be using them in a lot of the chapter's examples.

When counting, positive numbers can grow indefinitely towards +?. In a similar way negative numbers can grow indefinitely towards -?.

-?, ..., -2 , -1 , 0 , +1 , +2 , ..., +?

The '-' symbol is used to indicate negative numbers

The '+' symbol is used to indicate positive numbers. In many cases the '+' symbol is omitted.
Signed numbers has a single '0' (zero); having no sign.

The '+' symbol could be eliminated as it could be assumed, but the '-' symbol must still be represented somehow. Since computers do not understand anything but numbers, the '-' and the '+' symbols must be represented using two specific coded numbers. These codes could be represented using any two numbers as long as we understand them as the '-' and '+' signs when they occur at the beginning of any number. This means that the two above codes must be eliminated from the list of all possible numbers!

Over time many schemes to represent negative and positive numbers in computers were proposed. Some schemes seem to have been more successful than others. In all schemes a fixed number of bits is adopted when representing numbers.

There are several commonly known negative number representations. Since the start of computer development in the early 1950's these methods have been adopted and used at various times.

Signed magnitude representation

This early representation is intuitive since the sign and magnitude (absolute value) of a number are represented separately.

A 4 bit binary system could represent 16 ($16 = 2^4$) different numbers ranging from 0 (0000_2) to 15 (1111_2):

	Bits	*3 (MSB)*	*2*	*1*	*0 (LSB)*
0	***=***	***0***	***0***	***0***	***0***
1	***=***	***0***	***0***	***0***	***1***
...	***=***	***...***	***...***	***...***	***...***
15	***=***	***1***	***1***	***1***	***1***

PS: MSB is short for Most Significant Bit which in our case is the bit the far left. LSB is short for Least Significant Bit which in our case is the bit the far right.

Bit 3 is called the sign bit:

- If bit 3 = 0 => number is positive.
- If bit 3 = 1 => number is negative.

Since the MSB is reserved for the sign, 3 bits (bits 0 to 2) are left to represent the number.

The range of binary numbers represented in sign magnitude is: $-2^{n-1} + 1$ to $+ 2^{n-1} - 1$. For example, for 4 bits, the range is: -7 to + 7

The above table would become:

	Bits	*Sign bit*	*2*	*1*	*0*
+7	***=***	***0***	***1***	***1***	***1***
+2	***=***	***0***	***0***	***1***	***0***
+1	***=***	***0***	***0***	***0***	***1***
+0	***=***	***0***	***0***	***0***	***0***
-0	***=***	***1***	***0***	***0***	***0***
-1	***=***	***1***	***0***	***0***	***1***
-2	***=***	***1***	***0***	***1***	***0***
-7	***=***	***1***	***1***	***1***	***1***

Addition and subtraction with signed magnitude representation

Addition and subtraction in signed binary numbers is not straight forward, why?

Check the sign

if (Signs are Equal)
    Add magnitudes
    Append the sign to the result

else //Signs are different
    Compare the magnitudes
    Subtract the smaller from the larger
    Append sign of the greater to result

Example 1: (-3 – 2 = -5) or 1011 + 1010
(010 + 011 = 101)
append sign (1) = 1101

Example 2: (2 - 5 = -3) or 0010 + 1101
(101 – 010 = 011)
append sign (1) = 1011

Advantage and Disadvantage of the signed magnitude representation

Advantages: The method is simple and intuitive for humans to represent as many negative numbers as positive numbers.

Disadvantages: The method requires several tests and decisions (such as switching the order of operands) when performing arithmetic. It is harder to implement in computers, costing additional circuitry and execution time. In addition, dealing with the concept of borrowing is hard to implement with hardware. Finally, it has 2 representations for zero: +0 and -0! Adding +1 to -1 leads to either 1000 or 0000 both representing 0. Consequently it could be a poor choice for a computer system.

Complement representation

In order to simplify computer arithmetic circuits it would be nice if subtraction is handled the same way as addition without the need to deal with borrowing.

- 5 - 3 = 5 + (-3)
- (-3) should be represented in a certain format.

In mathematics, the "method of complements" is a technique used to subtract one number from another using only addition of positive numbers. In brief, the number to be subtracted is first converted into its "complement", and then added to the other number.

The complement of a number M is a value that together with M makes the whole number. It is determined by the base and number of digits.

	Bits	***Sign bit***	***2***	***1***	***0***
+7	***=***	***0***	***1***	***1***	***1***
+2	***=***	***0***	***0***	***1***	***0***
+1	***=***	***0***	***0***	***0***	***1***
+0	***=***	***0***	***0***	***0***	***0***
-0	***=***	***1***	***1***	***1***	***1***
-1	***=***	***1***	***1***	***1***	***0***
-2	***=***	***1***	***1***	***0***	***1***
-7	***=***	***1***	***0***	***0***	***0***

As you can see in the table above:

- Positive numbers are still represented as in the sign magnitude method.
- The negative representation of any positive number is done by flipping all its bits (0 becomes 1 and 1 becomes 0).
- This is called the ***1's complement*** representation

In 1964 a supercomputer (the CDC 6600) built by Semour Cray (while at CDC - Control Data Corporation) was based on 1's complement representation. Other computers such as the PDP-1 and UNIVAC 1100/2200 series also used the 1's complement representation for doing arithmetic.

Nowadays, 1's complement representation is not used in modern computer systems.

Disadvantages:

a) It has 2 representations for zero: +0 and -0! Even though they are the same algebraically. This causes problems when doing tests on arithmetic results.
b) Most computers now use a variation of 1's complement (called 2's complement) that eliminates the above problems

2's Complement

In Binary systems, a slight variation of the 1's complement is used to avoid the problem discussed in the 1's complement section above, thus using all ones (1111) to represent -1. In this case -1 + 1 = 1111 + 0001 = 0000 (1 discarded). Positive numbers are still represented as in the sign magnitude method. For negative numbers, all the bits are then complemented as follows:

In Math terms **the 2's complement of M is: $2^n - M$** where n is the number of bits representing M.
Another way to compute the 2's complement in a fast way: **2's Complement of M = 1's complement of M + 1**

Example:

Compute the 2's complement of M = 0100_2
M = $0100_{2\,=}$ > 1's of M = 1011_2
2's complement of M = $1011_2 + 1 = 1100_2$

An easier way to obtain the 2's complement of a binary number is to parse that number starting with the LSB until you find the rightmost digit that is equal to 1. Leave that digit and all other digits to the right of it unchanged, and then complement all digits to the left of that one digit.

Bits	8(MSB)	7	6	5	4	3	2	1(LSB)
+72	0	1	0	0	1	0	0	0
	Flip	Flip	Flip	Flip	No Flip	No Flip	No Flip	No Flip
-72	1	0	1	1	1	0	0	0

The 2's complement of a negative number represented in its 2's complement form is equal to its corresponding positive number.

Bits	8(MSB)	7	6	5	4	3	2	1(LSB)
-72	1	0	1	1	1	0	0	0
	Flip	Flip	Flip	Flip	No Flip	No Flip	No Flip	No Flip
+72	0	1	0	0	1	0	0	0

The range of numbers using n bits is $-(2^{n-1})$ to $(2^{n-1} - 1)$.
For 4 bits the range is: -8 to +7

	Bits	Sign bit	2	1	0
+7	=	0	1	1	1
+1	=	0	0	0	1
+0	=	0	0	0	0
-1	=	1	1	1	1
-8	=	1	0	0	0

Advantages:

It has a single representation of the zero: 0

Each positive number has a corresponding negative number that starts with a 1, except -8 which has no corresponding positive number.

Example:

5-5 is computed as follows in a 4-bit precision:
0101 + 1011 = 10000. The left most bit is lost since we are limited by 4 bit precision (corresponds to a 4 bit hardware inside the computer), so the answer is 0000 which is the only representation of zero.

The 2's complement simplifies the logic required for addition and subtraction, since can use the addition to add and subtract both negative and positive numbers the same way, hence reducing and optimizing the underlying computer circuitry.

Nowadays, almost all computer systems are based on the 2's complement representation.

Java's Bitwise Operators

Operator	Name
<<	Shift Left
>>	Shift Right
>>>	Shift Right with Zero Fill
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Compliment

Detailed explanation and examples on all the above bitwise operators will be provided in this chapter.

Shifting

Shift Left (<<)

The shift left operator **(<<)** shifts all the bits of a number **n** times to the left.

Example

```
public static void main(String[] args)
{
      int a = 10;
      a = a << 1;
      System.out.println(a);

      a = 10;
      a = a << 3;
      System.out.println(a);
}
```

Output

```
20
80
```

Explanation

```
     Binary:   Decimal:

     00001010       10

 <<         1

_____________________

     00010100       20
```

As you can see, all the bits moved one space to the left resulting in a different integer value. Every time we shift the bits, a zero value is placed at the least significant bit (the one on the far right).

```
     Binary:   Decimal:

     00001010       10

 <<          3

_____________________
     01010000       80
```

Shifting a number by 3 moves all its bits by 3 spaces.

Shifting to left by **n** is equivalent to multiplying the number by 2^n
10 << 1 = 20 and 10 * 2^1 = 20
10 << 3 = 80 and 10 * 2^3 = 80

For the next few example, we will be using the **byte** type in order to make things simpler to understand and experiment with.

The **byte,** like all integral type in java, is signed. The most significant bit will help us humans figure out very fast if the value is positive or negative. If the most significant bit (the one on the far left) is a **0** the number is **positive** otherwise the number is **negative.**

Value in Binary	Positive / Negative	Value in Decimal
10000001	Negative	-127
00000001	Positive	1
10100001	Negative	-95
01101111	Positive	111

Example

```
public static void main(String[] args)
{
	byte a = -10;
	a = (byte)(a << 3);
	System.out.println(a);

	a = (byte)(a << 1);
	System.out.println(a);

	a = (byte)(a << 1);
	System.out.println(a);
}
```

Output

```
-80
96
-64
```

Explanation

```
     Binary:   Decimal:

     11110110      -10

 <<         3
_____________________
     10110000      -80
```

We shifted the negative value -10 to the left by 3, we got -80 which we expected:
$-10 * 2^3 = -80$

Since we are restricted to 8 bits, every time we shift we lose the most significant bit.

Now let us see what happens if we shift -80 by 1 to the left.

```
     Binary:   Decimal:

     10110000      -80

 <<         1
_____________________
     01100000       96
```

The most significant bit is now a **0** and of course the value inside **a** is now **positive.** Developers have to be careful in these situations, they can assume that a -160 value will be in the variable **a** after executing the shift but because of our 8 bits restriction we ended up with a totally different value.

Shift Right (>>)

The shift right operator **(>>)** shifts all the bits of a number **n** times to the right.

Example

```
public static void main(String[] args)
{
	int a = 10;
	a = a >> 1;
	System.out.println(a);
}
```

Output

5

Explanation	Binary: Decimal: 00001010 10 >> 1 00000101 20 As you can see, all the bits moved one space to the right resulting in a different integer value. The most significant bit got replaced with a **0**. This illustrates that, like a left shift by **n** is equivalent to multiplying a number by 2^n, a right shift is equivalent to dividing a number by 2^n.

Let's test the **Shift Right** operator with negative values.

Example	`public static void main(String[] args)`
	`{`
	`  byte a = -10;`
	`  a = (byte)(a >> 1);`
	`  System.out.println(a);`
	`}`

Output	`-5`
Explanation	Binary: Decimal: 11110110 -10 >> 1 11111011 -5 All the bits moved one space to the right resulting in a different integer value. Unlike the example above, the most significant bit got replaced with a **1**. This is something special with the **Shift Right** operator, when shifting it replaces the most significant bit by **0** for **positive numbers** and by **1** for **negative numbers** in order to preserve the sign.

Shift Right with Zero Fill (>>>)

The shift right with zero fill operator **(>>)** shifts all the bits of a number **n** times to the right, but unlike the shift right operator (>>) it does not preserve the sign.

Example

```
public static void main(String[] args)
{
      int a = -10;
      System.out.println("a in decimal :" + a);
      System.out.println("a in binary: " + Integer.toBinaryString(a));
      a = a >> 1;
      System.out.println("a in decimal :" + a);
      System.out.println("a in binary: " + Integer.toBinaryString(a));

      a = -10;
      System.out.println("a in decimal :" + a);
      System.out.println("a in binary: " + Integer.toBinaryString(a));
      a = a >>> 1;
      System.out.println("a in decimal :" + a);
      System.out.println("a in binary: " + Integer.toBinaryString(a));
}
```

Output

```
The value of a in decimal :-10
The value of a in binary: 11111111111111111111111111110110
The value of a in decimal :-5
The value of a in binary: 11111111111111111111111111111011
The value of a in decimal :-10
The value of a in binary: 11111111111111111111111111110110
The value of a in decimal :2147483643
The value of a in binary: 11111111111111111111111111111011
```

Explanation

```
     Binary:                                  Decimal:

     11111111111111111111111111110110            -10

 >>                                1

_____________________________________________________
     11111111111111111111111111111011             -5
```

With the shift right operator (>>) all the bits moved one space to the right with a most significant bit value equal to 1 in order to preserve the sign.

```
      Binary:                                   Decimal:

      11111111111111111111111111110110            -10

>>>                                 1

-------------------------------------------------------

      01111111111111111111111111111011        2147483643
```

With the shift right with fill zero operator (>>>) all the bits moved one space to the right with a most significant bit value equal to 0 which will not preserve the sign hence the new value of 2147483643.

Note: The result of this operation will always be positive since the most significant bit will always be a zero.

AND(&) OR(|) XOR(^) NOT(~)

The bitwise operators allow us to run boolean logic on the bit level. What this means is that 0 is treated like false, and 1 like true. These operators only work on integral types i.e. int, long, short.

AND (&)

The bitwise **AND** operator (**&**) is a binary operator that works on the bit level. Given two bits, the AND operator will return a **1 if, and only if, both operands are equal to 1.**
In other words, the **AND** operator follows the following table:

&	0	1
0	0	0
1	0	1

Example

```
public static void main(String[] args)
{
      System.out.println("0 & 0 = " +  (0 & 0));
      System.out.println("0 & 1 = " +  (0 & 1));
      System.out.println("1 & 0 = " +  (1 & 0));
      System.out.println("1 & 1 = " +  (1 & 1));

      int one = 0b0000_1000;  // 8 in decimal
      int two = 0b1010_1000;  // 168 in decimal
      System.out.println("one & two = " + (one & two));
}
```

Output

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
one & two = 8
```

Explanation

```
     Binary:   Decimal:

     00001000         8

 &   10101000       168
_____________________
     00001000         8
```

Each bit in the **one** variable will be AND-ed with its respective bit in the **two** variable. The returned value will be a new integer.

OR (|)

The bitwise **OR** operator (**|**) is a binary operator that works on the bit level. Given two bits, the OR operator will return a **1 if either of the operands is equal to 1.**
In other words, the **OR** operator follows the following table:

|	**0**	**1**
0	**0**	**1**
1	**1**	**1**

Example

```
public static void main(String[] args)
{
      System.out.println("0 | 0 = " +  (0 | 0));
      System.out.println("0 | 1 = " +  (0 | 1));
      System.out.println("1 | 0 = " +  (1 | 0));
      System.out.println("1 | 1 = " +  (1 | 1));

      int one = 0b0100_1000;  // 72 in decimal
      int two = 0b1010_1000;  // 168 in decimal
      System.out.println("one | two = " + (one | two));
}
```

Output

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
one | two = 232
```

Explanation

```
     Binary:   Decimal:

     01001000        72

 |   10101000       168
_____________________
     11101000       232
```

Each bit in the **one** variable will be OR-ed with its respective bit in the **two** variable. The returned value will be a new integer.

XOR (^)

The bitwise **XOR** operator (**^**) is a binary operator that works on the bit level. Given two bits, the XOR operator will return a **1 if, and only if, one of the operands is equal to 1.**
In other words, the X**OR** operator follows the following table:

^	0	1
0	0	1
1	1	0

Note: 1 ^ 1 = 0 and not 1 (which is a very common mistake).

Example

```
public static void main(String[] args)
{
      System.out.println("0 ^ 0 = " +  (0 ^ 0));
      System.out.println("0 ^ 1 = " +  (0 ^ 1));
      System.out.println("1 ^ 0 = " +  (1 ^ 0));
      System.out.println("1 ^ 1 = " +  (1 ^ 1));

      int one = 0b0100_1000;  // 72 in decimal
      int two = 0b1010_1000;  // 168 in decimal
      System.out.println("one ^ two = " + (one ^ two));
}
```

Output

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
one ^ two = 224
```

Explanation

```
     Binary:   Decimal:

     01001000        72

 ^   10101000       168
_____________________
     11100000       224
```

Each bit in the **one** variable will be XOR-ed with its respective bit in the **two** variable. The returned value will be a new integer.

NOT (~)

The bitwise **NOT** operator (**~**) is a unary operator that works on the bit level. Given a bit, the NOT operator will return a **1 the bit is 0 and a 0 if the bit is 1.**
In other words, the **NOT** operator follows the following table:

~	0	1
	1	0

Example

```
public static void main(String[] args)
{
      System.out.println("~0 = " +  (~0));
      System.out.println("~1 = " +  (~1));

      int one = 0b0100_1000;  // 72 in decimal
      System.out.println("~one = " + (~one));
}
```

Output

```
~0 = -1
~1 = -2
~one = -73
```

Explanation

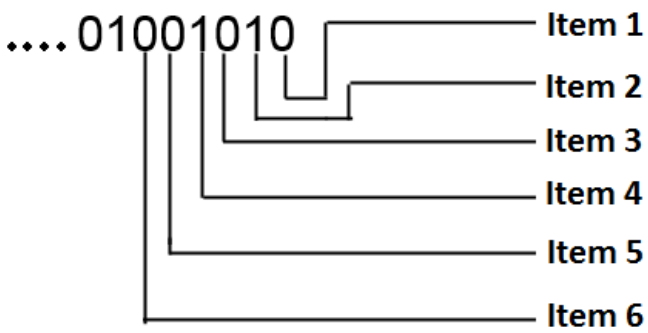
```
     Binary:   Decimal:

 ~   00000000         0
_____________________________
     11111111        -1


     Binary:   Decimal:

 ~   00000001         1
_____________________________
     11111110        -2


     Binary:   Decimal:

 ~   01001000        72
_____________________________
     10110111       -73
```

Each bit in the number will be NOT-ed which leads to a new integer value.

Using the bits of an integer as flags

Now that we know how all the bitwise operators work, let's see how we can benefit from them in code.
Assume we have an **integer** variable and the size of an integer is **32bits**. By using bitwise operators we can treat the bits inside that integer as 32 flags which can replace 32 booleans. By doing that, we are saving a lot of memory since the size of one integer is definitely less than the size of 32 booleans.

Say we are designing a video game where each player can collect a total of 32 items. One way of designing our code is to have an array of 32 booleans for each character. Each boolean will represent a specific item, and when the player collects that specific item we change the boolean to true.
Or, we can instead have one integer and use its bits as 32 flags, one for each item.



From the picture above we know that the player collected items 2, 4 and 7.
Let's say the game is running and for some reason we need to check if the player collected item 4, the only thing we need to do is AND (&) our items integer variable with the value 8 (which is called the item's mask) and if the result is anything other than 0 then the player has the item.

```
   01001010  Items
             Integer

&  00001000  Item4
             Mask
------------------
   00001000
```

Let's do the same test but with item 1:

```
   01001010  Items
             Integer

&  00000001  Item1
             Mask
------------------
   00000000
```

The result is 0 which indicates that the player hasn't collected item 1 yet.

Note: An item's mask is the integer value that has all the bits set to 0 except for the respective item's flag.

How about setting a bit flag on when the player collects an item. In order to do so, we OR (|) the items integer with the item's mask.

Assume the player just collected item 1:

```
   01001010  Items
             Integer

|  00000001  Item1
             Mask
-------------------
   01001011
```

As you can see, all the items integer's flags remained the same except for item 1's flag that got set.

Very Important Note	***When we want to set a bit flag to true, we need to change the items integer variable.*** ***ItemsInteger = ItemsInteger | ItemMask*** ***It is not the case when we check if an item is collected or not:*** ***if((ItemsInteger & ItemMask) != 0)*** ***{*** ***...*** ***}***

Last but not least, we can use the XOR (^) operator to toggle a bit ON and OFF. Assume that the 7th bit controls if a light is turned on or off:

```
   01001010  Items
             Integer

^  01000000  Bit7
             Mask
-------------------
   00001010
```

As you can see, XOR-ing the items integer variable with the "Bit7Mask" variable changed the 7th bit from 1 to 0. Let's XOR it again with the same mask:

```
   00001010  Items
             Integer

^  01000000  Bit7
             Mask
-------------------
   01001010
```

The 7th bit is changed from 0 to 1. Toggling is done with the following equation:

ItemsInteger = ItemsInteger ^ bitMask