Theory: Bitwise and bit-shift operations

As you know, in most cases, computer numbers are represented in binary format. In this format, each digit of a number can be either `0` or `1`. For example, the decimal value `15` is `1111` in the binary format. We've already learned how to convert integer numbers to the binary format and do some binary arithmetics. Now let's learn what else can be done with binary numbers.

The Java language provides several operators for manipulating individual bits of integer numbers. These operations are fast and simple actions, directly supported by the processor. They are particularly important in lower-level programming such as device drivers, low-level graphics, network communication, encryption, and compression.

Although computers often have efficient built-in instructions for performing arithmetic and logical operations, all these operations can be performed by combining the **bitwise** and **bitshift** operators and zero-testing in various ways.

§1. Bitwise operators

There are four bitwise operators: `~` **(bitwise NOT, inversion, complement)**, `|` **(bitwise OR)**, `&` **(bitwise AND)** and `^` **(bitwise XOR)**. Each of these operators goes through all bits of both operands (numbers) one by one (i.e. bitwise) and produces a new number as a result.

- `~` is a unary operator that **inverses** bits in the binary format of the number making every `0` a `1` and every `1` a `0`. It also changes the sign bit of the value.
- `|` is a binary operator that performs **bitwise OR**: the result digit is `1` if at least one operand digit is `1`, otherwise, it is `0`;
- `&` is a binary operator that performs **bitwise AND**: the result digit is `1` if both operand digits are `1`, otherwise, it is `0`;
- `^` is a binary operator that performs **bitwise XOR**: the result digit is `1` if exactly one operand is `1`, otherwise, it is `0`

> The listed operators can be applied to integer and boolean operands. If both operands are integers then bitwise operations will be performed. If both operands are booleans, they perform the corresponding logical operations (except `~`).

Let's assume we have two integer numbers: `15` and `10`. The first number has a binary representation `1111`, the second one is `1010`.

```
int first = 15;  // binary format 1111
int second = 10; // binary format 1010

int bitwiseAnd = first & second; // 1111 & 1010 = 1010, the result is 10
int bitwiseOr = first | second;  // 1111 | 1010 = 1111, the result is 15
int bitwiseXor = first ^ second; // 1111 ^ 1010 = 0101, the result is 5
```

We hope, now you understand how these operators work. We will demonstrate to you how to apply them in practice in the next topics and as well as in our projects.

§2. Bit-shift operators

In addition to the bitwise operators, Java also provides **bit-shift** operators that can be used to shift bits of an integer number from one position to another.

Current topic:

✓ Bitwise and bit-shift operations

Topic depends on:

✓ Stage 1 Types and variables

✗ 2s Complement

Table of contents:

↑ Bitwise and bit-shift operations

§1. Bitwise operators

§2. Bit-shift operators

§3. Precedence of bitwise and bit-shift operations

§4. Conclusion

Feedback & Comments

There are three bit-shift operators:

- `<<` is a signed bit-shift operator that shifts a bit pattern to the left by the distance specified in the right operand. It fills the empty place with zeros;
- `>>` is a signed bit-shift operator that shifts a bit pattern to the right by the distance specified in the right operand. It fills the empty place with the values of the sign bit.
- `>>>` is an unsigned bit-shift operator that shifts a bit pattern to the right by the distance specified in the right operand. It is almost like `>>`, but shifted values are filled up with zeros. The result of the `>>>` operator is always positive.

The following example illustrates how to perform **fast multiplication and division** by two using bit-shift operators.

```
int val = 25;   // binary: 0001 1001, decimal: 25

val = val << 1; // binary: 0011 0010, decimal: 50
val = val << 2; // binary: 1100 1000, decimal: 200

int anotherVal = 14;   // binary: 1110, decimal: 14
anotherVal = anotherVal >> 1; // binary: 0111, decimal: 7
```

As you can see, the result of the left-shift operator `<<` is equivalent to the multiplication by two, and the result of the right-shift operator is equivalent to the division by two. Let's generalize: when we use signed bit-shift operators we perform the multiplication or division of the left operand by the 2 in the power of right operand.

```
int newVal = 25;

newVal = newVal << 1; // 25 * 2^1 = 50
newVal = newVal << 3; // 50 * 2^3 = 400
newVal = newVal >> 2; // 400 / 2^2 = 100
```

Another example is the **calculation of the middle** of an integer positive interval.

```
int left = 10;
int right = 20;

int mid = left + right >> 1; // this is 15!
```

Of course, this magic produces the same result as `(left + right) / 2`, but the bit-shift version is often considered as a faster way to do that. Some algorithms in the Java standard library use this approach.

Unlike the signed right shift (`>>`), the unsigned (`>>>`) one does not take sign bits into consideration, it just shifts all the bits to the right and pads the result with zeros from the left. That means that for negative numbers, the result is always positive. Signed and unsigned right shifts have the same result for positive numbers.

§3. Precedence of bitwise and bit-shift operations

As well as arithmetic, bitwise and bit-shift operators have so-called precedence that determines the order of performing and grouping operations in the expression. Operations with higher precedence are performed before those with lower precedence. In the table below you will find all the operators we've learned so far ordered by their precedence in decreasing order.

Operators	Precedence
unary	+expr -expr ~
multiplicative	* / %

additive	+ -
shift	<< >> >>>
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|

When the operators have equal precedence, another rule is used that determines whether the evaluation should be performed from left to right or vice versa. It is called associativity.

All operators we have considered are evaluated from left to right, just like you're used to doing.

This means, that in two expressions `first | second & third` and `(first | second) & third` operations will be executed in a different order and hence results may vary. If you go back to the example above `left + right >> 1` in the code snippet, you can see that here we don't have to use brackets like in the equal `(left + right) / 2` because addition has higher precedence than all bit-shift operations. Remember these priorities when you combine arithmetic operators with the operations on bits.

§4. Conclusion

In this topic, we've learned how to perform certain operations on the bits of integer numbers and how these changes correspond to some of the arithmetic operations. Bitwise operators affect bits one by one according to the logical operations, while bit-shift ones allow us to move the whole bit pattern left or right. The theory may seem a bit complicated, however, it is essential to understand the basics. Now some practice will certainly help you to process new information!

Report a typo

303 users liked this theory. **22** didn't like it. **What about you?**

Start practicing

This content was created over 3 years ago and updated 12 days ago. Share your feedback below in comments to help us improve it!

Comments (20) Hints (0) Useful links (1) Show discussion