Programming Language II
CSE-215

Prof. Dr. Mohammad Abu Yousuf

yousuf@juniv.edu

Operators

Introduction

- Most of its operators can be divided into the following four groups:

  arithmetic,

  bitwise,

  relational, and

  logical.

Arithmetic Operators

Operator	Result
+	Addition (also unary plus)
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

Arithmetic Operators

- The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean types, but you can use them on char types, since the char type in** Java is, essentially, a subset of **int.**

Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The Bitwise Operators

- These operators act upon the individual bits of their operands

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

- The bitwise logical operators are **&, |, ^, and ~.**

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Bitwise OR

Bitwise OR is a binary operator (operates on two operands). It's denoted by |.

The | operator compares corresponding bits of two operands. If either of the bits is 1, it gives 1. If not, it gives 0. For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  ________
  00011101 = 29 (In decimal)
```

Example 1: Bitwise OR

```
class BitwiseOR {
    public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        result = number1 | number2;
        System.out.println(result);
    }
}
```

When you run the program, the output will be:

```
29
```

Bitwise AND

Bitwise AND is a binary operator (operates on two operands). It's denoted by &.
The & operator compares corresponding bits of two operands. If both bits are 1, it gives 1. If either of the bits is not 1, it gives 0.

For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25

  00001100
& 00011001
  ________
  00001000 = 8 (In decimal)
```

Example 2: Bitwise AND

```
class BitwiseAND {
     public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        result = number1 & number2;
        System.out.println(result);
     }
}
```

When you run the program, the output will be:

```
8
```

Bitwise Complement

Bitwise complement is an unary operator (works on only one operand). It is denoted by ~.

The ~ operator inverts the bit pattern. It makes every 0 to 1, and every 1 to 0.

```
35 = 00100011 (In Binary)
Bitwise complement Operation of 35

~ 00100011
  ________
  11011100 = 220 (In decimal)
```

Example 3: Bitwise Complement

```
class Complement {
    public static void main(String[] args) {

        int number = 35, result;

        result = ~number;
        System.out.println(result);
    }
}
```

When you run the program, the output will be:

```
-36
```

Why are we getting output -36 instead of 220?
It's because the compiler is showing 2's complement of that number; negative notation of the binary number.

For any integer n, 2's complement of n will be -(n+1).

Decimal	Binary	2's complement
0	00000000	-(11111111+1) = -00000000 = -0(decimal)
1	00000001	-(11111110+1) = -11111111 = -256(decimal)
12	00001100	-(11110011+1) = -11110100 = -244(decimal)
220	11011100	-(00100011+1) = -00100100 = -36(decimal)

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

Bitwise XOR

Bitwise XOR is a binary operator (operates on two operands). It's denoted by ^.

The ^ operator compares corresponding bits of two operands. If corresponding bits are different, it gives 1. If corresponding bits are same, it gives 0.

For example,

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)
Bitwise XOR Operation of 12 and 25
  00001100
| 00011001

  ________
  00010101 = 21 (In decimal)
```

Example 4: Bitwise XOR

```
class Xor {
     public static void main(String[] args) {

        int number1 = 12, number2 = 25, result;

        result = number1 ^ number2;
        System.out.println(result);
     }
}
```

When you run the program, the output will be:

```
21
```

left shift, right shift

A= 00111100

<< (left shift)

- Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand

  Example: A << 2 will give 240 which is 1111 0000.

>> (right shift)

- Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

  Example: A >> 2 will give 15 which is 1111

zero fill right shift

- **>>> (zero fill right shift)**Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

  Example: A >>>2 will give 15 which is 0000 1111

Signed Left Shift

The left shift operator << shifts a bit pattern to the left by certain number of specified bits, and zero bits are shifted into the low-order positions.

```
212 (In binary: 11010100)

212 << 1 evaluates to 424 (In binary: 110101000)
212 << 0 evaluates to 212 (In binary: 11010100)
212 << 4 evaluates to 3392 (In binary: 110101000000)
```

Example: Signed Left Shift

```
class LeftShift {
    public static void main(String[] args) {

        int number = 212, result;

        System.out.println(number << 1);
        System.out.println(number << 0);
        System.out.println(number << 4);
    }
}
```

When you run the program, the output will be:

```
424
212
3392
```

Signed Right Shift

The right shift operator >> shifts a bit pattern to the right by certain number of specified bits.

```
212 (In binary: 11010100)

212 >> 1 evaluates to 106 (In binary: 01101010)
212 >> 0 evaluates to 212 (In binary: 11010100)
212 >> 8 evaluates to 0 (In binary: 00000000)
```

If the number is a 2's complement signed number, the sign bit is shifted into the high-order positions.

Example: Signed Right Shift

```
class RightShift {
    public static void main(String[] args) {

        int number = 212, result;

        System.out.println(number >> 1);
        System.out.println(number >> 0);
        System.out.println(number >> 8);
    }
}
```

When you run the program, the output will be:

```
106
212
0
```

Unsigned Right Shift

The unsigned right shift operator << shifts zero into the leftmost position.

```
class RightShift {
    public static void main(String[] args) {

        int number1 = 5, number2 = -5;

        // Signed right shift
        System.out.println(number1 >> 1);

        // Unsigned right shift
        System.out.println(number1 >>> 1);

        // Signed right shift
        System.out.println(number2 >> 1);

        // Unsigned right shift
        System.out.println(number2 >>> 1);
    }
}
```

When you run the program, the output will be:

```
2
2
-3
2147483645
```

Notice, how signed and unsigned right shift works differently for 2's complement.

The 2's complement of 2147483645 is 3.

```java
public class Test {

  public static void main(String args[]) {
     int a = 60;         /* 60 = 0011 1100 */
     int b = 13;         /* 13 = 0000 1101 */
     int c = 0;

     c = a & b;        /* 12 = 0000 1100 */
     System.out.println("a & b = " + c );

     c = a | b;        /* 61 = 0011 1101 */
     System.out.println("a | b = " + c );

     c = a ^ b;        /* 49 = 0011 0001 */
     System.out.println("a ^ b = " + c );

     c = ~a;           /*-61 = 1100 0011 */
     System.out.println("~a = " + c );

     c = a << 2;     /* 240 = 1111 0000 */
     System.out.println("a << 2 = " + c );

     c = a >> 2;     /* 15 = 1111 */
     System.out.println("a >> 2  = " + c );

     c = a >>> 2;     /* 15 = 0000 1111 */
     System.out.println("a >>> 2 = " + c );
  }
}
```

```
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 15
a >>> 15
```

Boolean Logical Operators

- The Boolean logical operators shown here operate only on **boolean operands. All of the** binary logical operators combine two **boolean values to form a resultant boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Boolean Logical Operators

- The logical Boolean operators, **&, |, and ^, operate on boolean values in the same** way that they operate on the bits of an integer.

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Boolean Logical Operators

```
// Demonstrate the boolean logical operators
class BoolLogic {
  public static void main(String args[]) {
    boolean a = true;
    boolean b = false;
    boolean c = a | b;
    boolean d = a & b;

    boolean e = a ^ b;
    boolean f = (!a & b) | (a & !b);
```

```
          a = true
          b = false
        a|b = true
        a&b = false
        a^b = true
!a&b|a&!b = true
         !a = false
```

```
    boolean g = !a;
    System.out.println("          a = " + a);
    System.out.println("          b = " + b);
    System.out.println("        a|b = " + c);
    System.out.println("        a&b = " + d);
    System.out.println("        a^b = " + e);
    System.out.println("!a&b|a&!b = " + f);
    System.out.println("         !a = " + g);
  }
}
```

the string representation of a Java boolean value is one of the literal values true or false

Java AND Operator Example: Logical && and Bitwise &

- The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.
- The bitwise & operator always checks both conditions whether first condition is true or false.

```
1.class OperatorExample{
2.public static void main(String args[]){
3.int a=10;
4.int b=5;
5.int c=20;
6.System.out.println(a<b&&a++<c);//false && true = false
7.System.out.println(a);//10 because second condition is not checked
8.System.out.println(a<b&a++<c);//false && true = false
9.System.out.println(a);//11 because second condition is checked
10.}}
```

Output:
false
10
false
11

Java AND Operator Example: Logical && and Bitwise &

- This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

- For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

- Since the short-circuit form of AND (**&&) is used, there is no risk of causing a run-time** exception when **denom is zero. If this line of code were written using the single &** version of AND, both sides would be evaluated, causing a run-time exception when **denom is zero.**

Java OR Operator Example: Logical || and Bitwise |

- The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.
- The bitwise | operator always checks both conditions whether first condition is true or false.

```
1.class OperatorExample{
2.public static void main(String args[]){
3.int a=10;
4.int b=5;
5.int c=20;
6.System.out.println(a>b||a<c);//true || true = true
7.System.out.println(a>b|a<c);//true | true = true
8.//|| vs |
9.System.out.println(a>b||a++<c);//true || true = true
10.System.out.println(a);//10 because second condition is not checked
11.System.out.println(a>b|a++<c);//true | true = true
12.System.out.println(a);//11 because second condition is checked
13.}}
```

Java OR Operator Example: Logical || and Bitwise |

```
Output: Previous program
true
true
true
10
true
11
```

```java
public class Test {

   public static void main(String args[]) {
      boolean a = true;
      boolean b = false;

      System.out.println("a && b = " + (a&&b));

      System.out.println("a || b = " + (a||b) );

      System.out.println("!(a && b) = " + !(a && b));
   }
}
```

```
a && b = false
a || b = true
!(a && b) = true
```

Conditional Operator (? :)

- Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

```
variable x = (expression) ? value if true : value if false
```

Conditional Operator (? :)

```
public class Test {

   public static void main(String args[]){
      int a, b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

```
Value of b is : 30

Value of b is : 20
```

instance of Operator

- This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). **instanceof** operator is written as:

```
( Object reference variable ) instanceof  (class/interface type)
```

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

Output

```
true
```

Operator Precedence

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
=	op=					
Lowest						

Thank you