Java provides a rich operator environment. Most of its operators can be dividend into the following four categories.

1. Arithmetic Operator
2. Bitwise Operator
3. Relational Operator
4. Logical Operator

❖ Arithmetic Operators

Arithmetic Operators are used in mathematical expressions in the same way that they are used in algebra. The following tables lists the arithmetic operators.

Operator	Result
+	Addition
-	Substraction (unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment
-=	Substraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

The operands of the arithmetic operators must be of a numeric type. You cannot use them on Boolean types.

➢ The Modulus Operator:

The modulus operator, % returns the reminder of a division operation. It can be applied to floating-point types as well as integer types. (This differs from C/C++, in which the % can only be applied to integer types). The following program demonstrates the %.

```
// Demonstrate the % operator.
class Modulus {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;
    System.out.println("x mod 10 = " + x % 10);
    System.out.println("y mod 10 = " + y % 10);
  }}
```

Output:

```
x mod 10=2
y mod 10=2.25
```

- **Arithmetic Assignment Operators:**

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming.

```
a=a+3;
```

In Java, you can rewrite this statement as shown here.

```
a+=3;
```

Here is another example,

```
a=a%2;
```

Which can be expressed as

```
a%=2;
```

There are assignment operators for all of the arithmetic, binary operators. Thus any statement of the form

```
var=var op expression;
```

can be rewritten as

```
var op=expression;
```

- **Benefit of assignment operators:**

1. They save you a bit of typing, because they are 'shorthand" for their equivalent long forms.
2. They are implemented more efficiently by the Java run time system than are their equivalent long forms.

- **Increment and Decrement Operators**

The ++ and – are Java's increment and decrement operators. They have some special properties that make them quite interesting. The increment operator increases its operand by one. The decrement operator decreases its
its operand by one.

For example,

```
x=x+1;
y=y-1
```

can be rewritten like this by use of the increment operator:

```
x++; y--;
```

In this operator there are two differences between prefix form and postfix form. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
In the postfix form, the previous value is obtained for use in the expression and then the operand is modified.

```
x=42;
y=++x;
```

In this case, y is set to 43 as you would expect, because the increment occurs before x is assigned to y. thus, the line y=++x; is the equivalent of these two statements.

```
x=x+1;
y=x;
```

However, when written like this

```
x=42;
y=x++;
```

The value of x is obtained before the increment operator is executed, so the value of y is 42. Of course, in both cases x is set to 43. Here, the line y=x++; is the equivalent of these two statements.

```
y=x;
x=x+1;
```

The following program demonstrates the increment operator.

```
// Demonstrate ++ and --.
class incdec {
    public static void main(String args[]) {
      int a = 1;
      int b = 2;
      int c;
      int d;

      c = ++b;
      d = a++;
      c++;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      System.out.println("c = " + c);
      System.out.println("d = " + d);
  }
}
```

Bitwise Operators

- Java support special operators known bitwise operators for manipulation of data at bit level either 0 or 1.
- Bitwise operator can be applied to the integer types, long, int, short, char and byte. These operators can't be applied to float or double data type.
- Bitwise operators are use for testing bits, shifting bits to left, shifting bits to right & shifting bits to right zero fill. They are summarized in the following tables.

Operator	Result
~	Bitwise Unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
>>	Bitwise Shift Right
<<	Bitwise Shift Left
>>>	Bitwise Right Zero Fill
&=	Bitwise AND Assignment
|=	Bitwise OR Assignment
^=	Bitwise Exclusive OR Assignment
>>=	Bitwise Shift Right Assignment
<<=	Bitwise Shift Left Assignment
>>>=	Bitwise Right Zero Fill Assignment

The Bitwise Logical Operators:

- The Bitwise Logical Operators are &,|, ^ and -. The following table shows the outcome of each operation. It is remembered that the bitwise operator are applied to each individual bit within each operand.

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

The Bitwise & (AND)

Bitwise AND operator represented by & or AND. The result of AND operation is 1 if both bits have values **1**(One) otherwise **0 (Zero).**

For Ex. If the value of x is 17 and value of y is 28

X=17	-→	0001	0001	**8 bit**
Y=28	-→	0001	1100	**8 bit**
X & Y	-→	0001	0000	**8 bit**

Assignment: Write a program to find out even and odd number by using Bitwise AND Operator.

The Bitwise | Operator (OR)

The Bitwise OR Operator is represented by |. The result of OR operation is **1 (One)** at least one bit has a value **1 (One)** otherwise **0 (Zero)**.

For Ex. If the value of x is 17 and value of y is 28

X=17	-→	0001	0001	**8 bit**
Y=28	-→	0001	1100	**8 bit**
X | Y	-→	0001	1101	**8 bit**

Assignment: Write a program to which use Bitwise OR Operator.

The Bitwise Exclusive ^ Operator (Exclusive OR)

This operator is represented by ^. The result of exclusive or is 1 (One) if only one of the bits is **1 (One)** otherwise **0 (Zero).**

For Ex. If the value of x is 17 and value of y is 28.

X=17	-→	0001	0001	**8 bit**
Y=28	-→	0001	1100	**8 bit**
X ^ Y	-→	0000	1101	**8 bit**

Assignment: Write a program to which use Bitwise OR Operator.

The Bitwise ~ Operator (NOT)

This operator is represented by ~. This operator is also known as bitwise complement or unary NOT operator. The result of NOT operator is **1 (one)** if bits is **0 (Zero)** Otherwise **0 (Zero)** if bits is **1 (One)**.

For Ex. If the value of x is 17.

X=17	-→	0001	0001	**8 bit**
~X	-→	1110	1110	**8 bit**

Assignment: Write a program to which use Bitwise OR Operator.

Using the Bitwise Logical Operators:

```
// Demonstrate the bitwise logical operators.
class BitLogic {
  public static void main(String args[]) {
    String binary[] = {
      "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
      "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
    };

    int a = 3; // 0 + 2 + 1 or 0011 in binary
    int b = 6; // 4 + 2 + 0 or 0110 in binary
    int c = a | b;
    int d = a & b;
    int e = a ^ b;
    int f = (~a & b) | (a & ~b);
    int g = ~a & 0x0f;

    System.out.println("        a = " + binary[a]);
    System.out.println("        b = " + binary[b]);
    System.out.println("      a|b = " + binary[c]);
    System.out.println("      a&b = " + binary[d]);
    System.out.println("      a^b = " + binary[e]);
    System.out.println("~a&b|a&~b = " + binary[f]);
    System.out.println("       ~a = " + binary[g]);
  }
}
```

❖ The Bitwise Left Shift Operator (<<)

This operator is represented by <<. The left shift operator <<, shifts all of the bits in a value to the left a specified number times.

Each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2. But you need to watch out. If you shift a 1 bit into high-order position (bit 31 or 63), the value will become negative.

It has the following general form:

Syntax: Value<<num

Where **Value=** It specifies the value that we want to shifts as bits to the left direction.
Num= How many bits shifted in the left direction.

For Ex. We assume 8 bit value in this example. The value of x is 17 and shifts that value by 3 bit at left direction.

X=17	-→	0001	0001	**8 bit**
X<<3	**-→**	***		

X=X<<3	-→	1000	1000	**8 bit**

Note: (* means 3 bits shifts from the left. So these 3 bits will be lost at left direction and the next all bits will be replaced with that lost bits and remaining bits will be filled with 0.)

Assignment: Write a program to which use Bitwise Left Shift Operator.

❖ The Bitwise Right Shift Operator (>>)

This operator is represented by >>. The right shift operator >>, shifts all of the bits in a value to the right a specified number times.

Each right shift has the effect to divide that value by two and discards any remainder. You can take an advantage of this for high-performance integer division by 2. Of course, you must be sure that you are not shifting any bits off the right end.

It has the following general form:

Syntax: Value>>num

Where **Value=** It specifies the value that we want to shifts as bits to the right direction.
Num= How many bits shifted in the right direction.

For Ex. Here we assume **8 bit** in this example. The value of x is 35 and shifts that value by 2 bit at right direction.

X=35	-→	0010	0011	**8 bit**
X>>2	**-→**		**	
X=X>>2	-→	0000	1000	**8 bit**

Note: (** means 2 bits shifts from the right. So these bits will be lost at right direction and the next all bits will be replaced with that lost bits and remaining bits will be filled with 0.)

Assignment: Write a program to which use Bitwise right Shift perator.

- **How java represents negative numbers?**

All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is **00101010,** where each position represents a power of two, starting with 2^0 at the rightmost bit. The next bit position to the left would be 2^1 or 2, continuing toward the left with 2^2 or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3 and 5 (counting from 0 at the right); thus 42 is the sum of $2^1 + 2^3 + 2^5$, which is 2+8+32.

42 (Binary) = 00101010
42 (Decimal) =
76543210
$2^5\ 2^3\ 2^1 = 42$

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two's complement.

2's complement means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result. For example , -42 is represented by inverting all of the bits in 42, or 00101010, which holds 11010101, then adding 1, which results in 11010110 or -42. To decode a negative number, first invert all of the bits, and then add 1. -42 or 11010110 inverted holds 00101001 or 41, so when you add 1 you get 42.

```
Binary of 42    =    00101010
Binary of -42   =    11010101 (By inverting all of the bits in a value)
                          +1 (add 1)
                     -----------
                     11010110
```

Sign Extension: When you are shifting right, the top (leftmost) bits exposed by the right shifts are filled in with the previous contents of the top bit. This is called sign extension and serves to preserve the sign of negative numbers when you shift them to right. For example, -42>>3 is -6

For Ex. Here we assume **8 bit** in this example. The value of x is -42 and shifts that value by 3 bit at right direction.

```
X=-42    -→          1101      0110      8 bit
X>>3     -→                     ***
-------------------------------------------------------
X=X>>3   -→          1111      1010      8 bit
```

Note: (* means 3 bits shifts from the right. So these bits will be lost at right direction and the next all bits will be replaced with that lost bits and remaining bits will be filled with 1 to preserve the negative sign.)

Assignment: Write a program to which use Bitwise right Shift Operator for negative value.

The Unsigned Right Shift Operator (>>>)
Or
Shifts Right Zero Fill Assignment Operator (>>>)

This operator is represented by >>>. The unsigned right shift operator >>>, shifts all of the bits in a value with 0 (zero) to the right a specified number times.

As you have just seen, the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However sometimes this is undesirable.

For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an **unsigned shift.**

To accomplish this, you will use java's unsigned, shift-right operator, **>>>** which always shifts zeros into the high-order bit.

If a is set to -1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255.

```
a=-1          11111111  11111111  11111111  11111111  =-1
a=a>>>24      00000000  00000000  00000000  11111111  =255
```

The **>>>** operator is often not as useful as you might like, since it is only meaningful for 32 or 64-bit values. Remember, smaller values are automatically promoted to **int** in expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8 or 16 bit value.

Assignment: Write a program to which use Bitwise unsigned right Shift Operator for negative value of integer data type and shift the bit from 20 to 24 bit and 25 to 32 bits.

Q 1: What do you mean by unsigned shift.
Q 2: What is the actual purpose of unsigned right shift operator?

SSCCS

❖ Bitwise Operator Assignments:

All of the binary bitwise operators have a shorthand form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

For example the following two statements, which shifts the value in a right by four bits are equivalent.

```
a=a>>4;
a>>=4;
```

Likewise, the following two statements, which result in **a** being assigned the bitwise expression **a** OR **b**, are equivalent:

```
a = a | b;
a |= b;
```

The following program creates a few integer variables and then uses the shorthand form of bitwise operator assignments to manipulate the variables:

```
class opbitEquals {
  public static void main(String args[]) {
    int a = 1;
    int b = 2;
    int c = 3;

    a |= 4;          //a=a|4 = 1 | 4 = 0001 | 0100 = 0101 = 5
    b >>= 1;         // 1
    c <<= 1;         // 6
    a ^= c;          // a ^ c = 5 ^ 6 = 0101 ^ 0110= 0011= 3
    System.out.println("a = " + a);
    System.out.println("b = " + b);
    System.out.println("c = " + c);
  }
}
```

//0011= $2^0 + 2^1 = 1 + 2 = 3$

SSCCS

❖ Relational Operators:

The relational operators determine the relationship that one operand has to other. They determine equality and ordering. The relational operators are shown here:

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

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expression that control the **if** statement and the various **loop statements**.
As stated, the result produced by a relational operator is a **boolean** value.

For example, the following code fragment is perfectly valid.

```
int a=4;
int b=1;
boolean c=a<b;
```

In this case the result of **a<b** is **false** stored in **c**.

If you are coming from a C/C++ background, please not the following.

```
int done;
//……
if(!done) …..// valid in C/C++
if(done)  …..// but not in Java
```

In Java, these statements must be written like this:

```
if(done==0) …    // This is Java-style.
if(done!=0) …    //
```

The reason is that Java does not define **true and false** in the same way as C/C++. In C/C++, **true** is any **nonzero value** and **false is zero**. **In Java**, **true and false** are nonnumeric values which do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators.

For example,

```
boolean b;
b=true;

if(b){          //valid in Java, condition will become true
…..
…..
}
```

❖ 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

The following table shows the effect of each logical operation:

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

The following program demonstrates the 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);
    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);
  }
}
```

Output:

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

SSCCS

❖ Short-Circuit Logical Operators:

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the **Boolean AND and OR operators**, and are known as short-circuit operators.

You know very well that the **OR operator** results in **true** when **A is true**, no matter what **B is**. Similarly, the **AND operator** results in **false** when **A is false**, no matter **what B** is.

If you use the **|| and &&** forms, rather than the **| and &** forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the **left operand** alone.

This is very useful when the **right-hand operand** depends on the left one being true or false 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:

```
int denom=0,num=10
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 have to be evaluated, causing a run-time exception when **denom is ZERO**.

For example,

```
int c=1,e=99;
if (c==2 || e++ <100)
d=100;
```

Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

❖ The Assignment Operator:

The Assignment Operator is the single equal sign=. The assignment operator works in Java much as it does in any other computer language. It has this general form:

Var=expression;

Here, the type of **var** must be compatible with the type of expression.
The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x,y,z;
x=y=z=100;       //set x, y, and z to 100
```

This fragment sets the variables **x, y and z** to 100 using a single statement.
This works because the = is an operator that holds the value of the right-hand expression.

Therefore the value of z=100 is 100, which is then assigned to y, which in turn is assigned to x. using a "chain of assignment" is an easy way to set a group of variables to a common value.

❖ The ? Operator:

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operators is the **?** and it works in Java much like it does in **C, C++ and C#**. The **?** has this general form:

expression1 ? expression2:expression3

Here, expression1 can be any expression that evaluates to a **Boolean value**. If expression1 is true, then expression2 is evaluated;otherwise, expression3 is evaluated.

The result of the **?** Operation is that of the expression evaluated. Both **expression2 and expression3** are required to return the same type, which can't be **void.**

For example,

```
int  i, j,ans;
i=10;
j=20;
ans=(i>j) ? i : j;
```

When Java evaluates this assignment expression, it first looks at the expression to the **left** of the question mark. If i is greater than j then the expression between **question mark** and **colon** is evaluated and used as the value of the entire **?** expression. If **I** is not greater than **j** then the expression after the **colon** is evaluated and used for the value of the

entire **? expression.** The result produced by the **?** operator is then assigned to **ans.**

Here , is a program that demonstrate the ? operator. It uses it to obtain the largest value between two variables.

For example,

```
// Demonstrate ?.
class largest {
  public static void main(String args[]) {
    int i, j,k;

    i = 10,j=20;
    k = i > j ? i : j;        // get largest value
   System.out.println(i + " is " + k);
  }}
```

❖ Using Parentheses:

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a>>b+3;
```

This expression first adds 3 to b and then shifts a right by that result. That is this expression can be rewritten using redundant parentheses like this:

```
a>> (b+3)
```

However, if you want to first shift **a right by b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a>>b) +3
```