Topic Covered:
The Bitwise Logical Operators, Left shift operator (<<), Right shift Operator (>>), Unsigned Right shift Operator (>>>), The Relational Operators, Boolean Logical Operators, The ? Operator and Operator Precedence.

THE BITWISE OPERATORS

Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands.

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

THE BITWISE LOGICAL OPERATORS

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation. Bitwise operators are applied to each individual bit within each operand.

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

THE BITWISE NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:
`00101010` becomes `11010101` after the NOT operator is applied.

THE BITWISE AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example:

```
   00101010   42
 & 00001111   15
   00001010   10
```

THE BITWISE OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
   00101010 42
 | 00001111 15
   00101111 47
```

THE BITWISE XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^.

```
  00101010 42
^ 00001111 15
  00100101 37
```

PROGRAM 1: Demonstrate the bitwise logical operators.

```
class Bitwise {
public static void main(String args[]) {
byte a = 42;
byte b = 15;
System.out.println("Not a = " +  ~a );
System.out.println("a AND b = " + (a&b) );
System.out.println("a OR b = " + (a|b) );
System.out.println("a XOR b = " + (a^b) );
}
}
```

LEFT SHIFT OPERATOR (<<)

Shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two. For example,

```
a = 5          → (00000101)
b = -10        → (11110110)

a << 1 = 10    → (00001010)
a << 2 = 20    → (00010100)

b << 1 = -20   → (11101100)
b << 2 = -40   → (11011000)
```

PROGRAM 2: Left shifting a byte value.

```
class ByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
}
}
```

RIGHT SHIFT OPERATOR (>>)

Shifts the bits of the number to the right and fills on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
Example 1:

```
a = 10          → (00001010)
a >> 1 = 5      → (00000101)
```

Example 2:

```
a = -10         →(11110110)
a >> 1 = -5     →(11111011)
```

We preserve the sign bit.

PROGRAM 3: Right shift Operator

```
class Test {
public static void main(String args[])  {
int x = -4;
System.out.println(x>>1);
int y = 4;
System.out.println(y>>1);
}
}
```

UNSIGNED RIGHT SHIFT OPERATOR (>>>)

Shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0. (>>>) is unsigned-shift; it'll insert 0. (>>) is signed, and will extend the sign bit.
Example 1:

```
a = 10          → (00001010)
a >>> 1 = 5     → ((00000101)
```

Example 2:

```
a = -10         →(11110110)
a>>>1 = 123      →(01111011)
```

DOES NOT preserve the sign bit.

PROGRAM 4: Unsigned Right shift Operator

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

// x is stored using 32 bit 2's complement form.
// Binary representation of -1 is all 1s (111..1)
int x = -1;
System.out.println(x>>>29);  // The value of 'x>>>29' is 00...0111
System.out.println(x>>>30);  // The value of 'x>>>30' is 00...0011
System.out.println(x>>>31);  // The value of 'x>>>31' is 00...0001
}
}
```

PROGRAM 5: Compound bitwise operator assignments to manipulate the variables

```
class OpBitEquals {
public static void main(String args[]) {
int a = 1, b = 2, c = 3;
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

THE RELATIONAL OPERATORS

OPERATOR	DESCRIPTION
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes 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	Operator	Result
&	Logical AND	&=	AND assignment
|	Logical OR	|=	OR assignment
^	Logical XOR (exclusive OR)	^=	XOR assignment
||	Short-circuit OR	==	Equal to
&&	Short-circuit AND	!=	Not equal to
!	Logical unary NOT	?:	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

PROGRAM 6: Demonstrate relational operators and Boolean logical operators.

```
class BoolLogic {
public static void main(String args[]) {
int x = 7, y = 9;
boolean a = x<y;
boolean b = x==y;
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);
}
}
```

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* logical operators. 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 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. It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
if(c==1 & 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 ? OPERATOR

The ternary operator ? can replace certain types of if-then-else statements. It takes three expressions and returns a value:

```
(expression1) ? (expression2) : (expression3)
```

It replaces the following statements of if else structure

```
if(a>b)
c=a;
else
c=b;
```

can be replaced by

```
c=(a>b)?a:b
```

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

PROGRAM 7: Demonstrate ?.

```
class Ternary {
public static void main(String args[]) {
int k, i = 10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
i = -10;
k = i < 0 ? -i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
}
}
```

OPERATOR PRECEDENCE

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			