

# Java Program Structure

- In the Java programming language:
  - A program is made up of one or more *classes*
  - A class contains one or more *methods*
  - A method contains program *statements*
- These terms will be explored in detail throughout the course
- A Java application always contains a method called `main`

# Lincoln.java

```
public class Lincoln
{
    //-----
    // Prints a presidential quote.
    //-----
    public static void main (String[] args)
    {
        System.out.println ("A quote by Abraham Lincoln:");

        System.out.println ("Whatever you are, be a good one.");
    }
}
```

# Java Program Structure

```
// comments about the class
```

```
public class MyProgram  
{
```

class header



class body

Comments can be placed almost anywhere

```
}
```

# Java Program Structure

```
// comments about the class
```

```
public class MyProgram  
{
```

```
    // comments about the method
```

```
    public static void main (String[] args)
```

```
    {
```



method body

```
    }
```

method header

# Understanding Path structure

The path is required to be set for using tools such as javac, java, etc.

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

### **Temporary**

### **Permanent**

#### 1) How to set the Temporary Path of JDK in Windows

To set the temporary path of JDK, you need to follow the following steps:

Open the command prompt

Copy the path of the JDK/bin directory

Write in command prompt: set path=copied\_path

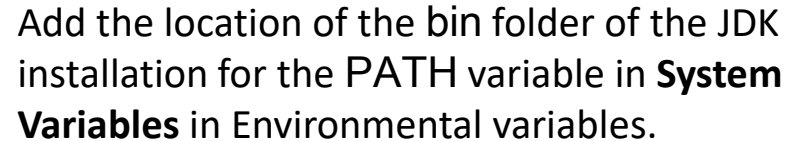
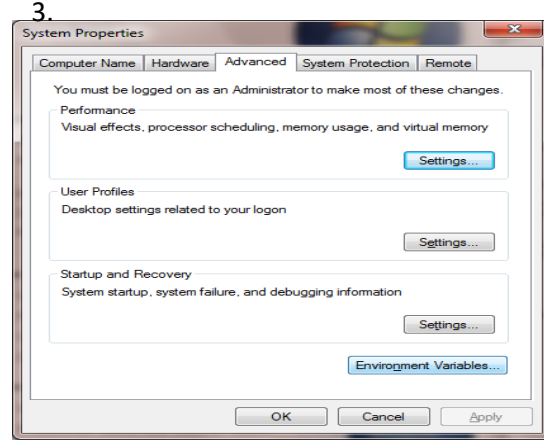
For Example:

set path=C:\Program Files\Java\jdk1.6.0\_23\bin

Let's see it in the figure given below:

# Understanding class path

- REF: <http://docs.oracle.com/javase/7/docs/webnotes/install/windows/jdk-installation-windows.html#Check>



C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.7.0\bin



# Java Data Types

---



# Data Types

Primitive

numeric

integer

byte

short

int

long

floating point

double

float

non - numeric

character

boolean

Non Primitive

strings

arrays

user defined  
classes

# Data Types

- Variables are nothing but **reserved memory locations** to store values. This means that when you create a variable you **reserve some space in memory**.
- All variables must first be declared before they can be used.
- Based on the **data type** of a variable, **the operating system allocates memory** and **decides what can be stored in the reserved memory**.
- Therefore, by assigning different data types to variables, you can store **integers**, **decimals**, or **characters** in these variables.
- The Java programming language is **statically-typed**.

# Data Types (Cont.)

- This involves stating the variable's type and name
  - Foreg: `int a=1;`
- Doing so tells your program that
  - **a field named "a" exists, holds numerical data, and**
  - **has an initial value of "1".**
- A variable's data type determines the values it may contain, plus the operations that may be performed on it.
- There are two categories of data types available in Java:
  - **Primitive Data Types**
  - **Reference/Object Data Types**

# Primitive Data Types (Cont.)

Data Type	Size (in Bytes)	Description	Example
float (Floating point Type)	4	<ul style="list-style-type: none"><li>❑ Store single-precision floating point numbers.</li><li>❑ Default value is 0.0f</li></ul>	float pi=3.14f;
double (Floating point Type)	8	<ul style="list-style-type: none"><li>❑ Store double-precision floating point numbers.</li><li>❑ Default value is 0.0d.</li></ul>	double pi=3.14; double pi=3.14d;
boolean	Not defined	<ul style="list-style-type: none"><li>❑ Represents one bit of information.</li><li>❑ Only two possible values: true and false.</li><li>❑ Used for simple flags that track true/false conditions.</li><li>❑ Default value is false.</li></ul>	boolean flag = true; boolean flg = false;
char	2	Single character	char ch='a';



# Primitive Data Types

- There are 8 primitive data types supported by Java.
- Primitive data types are **predefined by the language and named by a key word.**

Data Type	Size (in Bytes)	Description	Example
byte	1	<ul style="list-style-type: none"><li>❑ Store signed integer.</li><li>❑ Save memory.</li><li>❑ Default value is 0</li></ul>	byte a = -128; byte b = 127;
short	2	<ul style="list-style-type: none"><li>❑ Store signed integer.</li><li>❑ Default value is 0.</li></ul>	short s = -32768; short r = 32767;
int	4	<ul style="list-style-type: none"><li>❑ Large enough for the numbers</li><li>❑ Default value is 0.</li></ul>	int a = - 2147483648; int b = 2147483647;
long	8	<ul style="list-style-type: none"><li>❑ Range of values wider than those provided by int.</li><li>❑ Default value is 0L.</li></ul>	long a = 123L; long b = -123L;

# Default Values

- **Fields** that are declared but not initialized will be set to a reasonable default value by the compiler but not to the local variables.
- The following chart summarizes the default values for the different data types.

<b>Data Type</b>	<b>Default Value (for fields)</b>
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

# Reference Data Types

- Reference variables are created using **defined constructors of the classes**.
- They are used to **access objects**.
- These variables are declared to be of a **specific type** that cannot be changed.
  - ❖ **For example:** Employee, Student etc.
- It includes:
  - ❖ **Class objects**
  - ❖ **Various type of array variables**
- Default value of any reference variable is **null**.
- Used to refer to **any object of the declared type**.  
E.g. : **Animal** animal = new **Animal**("giraffe");



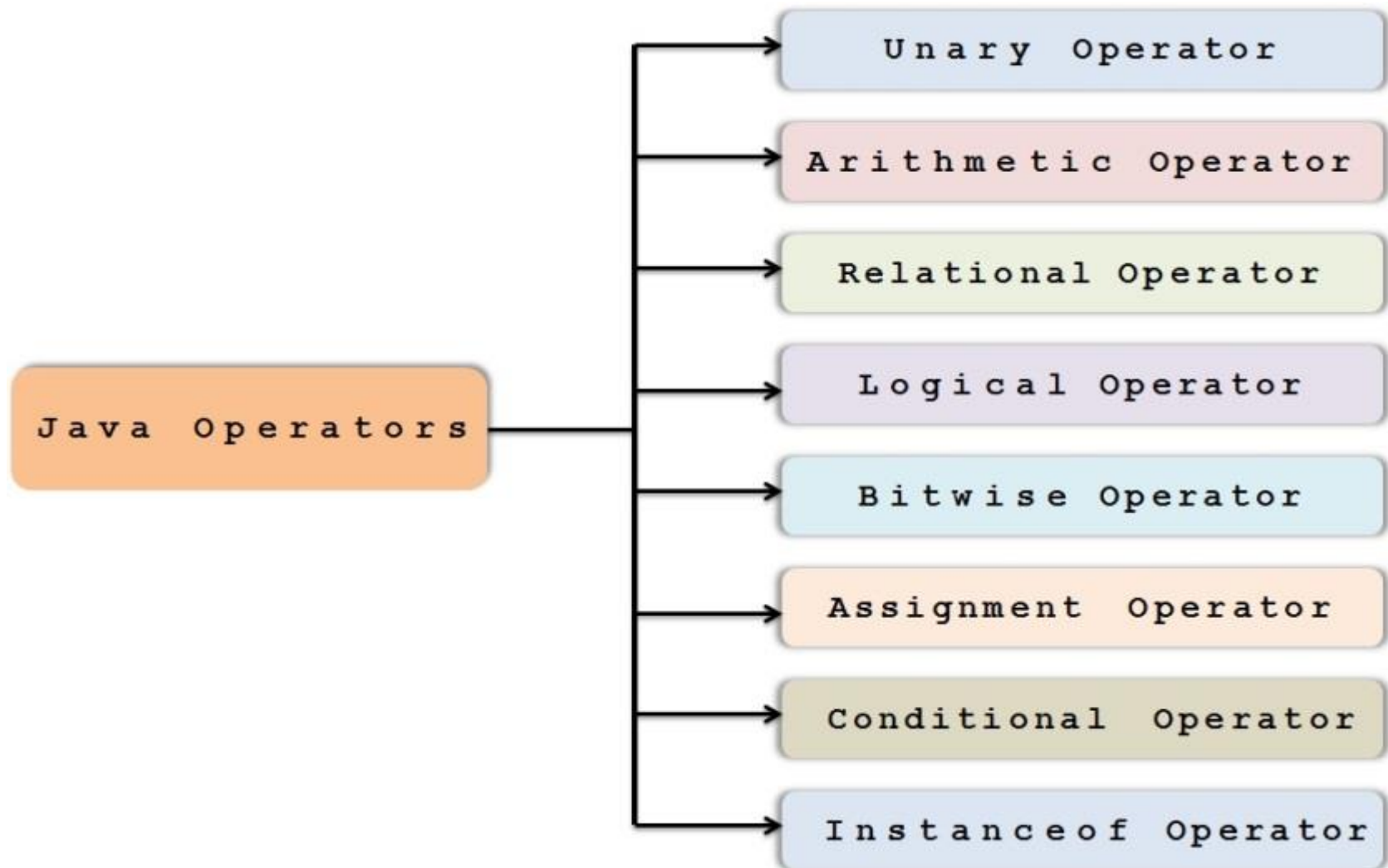
# OPERATOR

- An operator is a symbol that operates on one or more arguments to produce a result.
- Java provides a rich set of operators to manipulate variables.



# Operators in Java





# Arithmetic Operator

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

The following table lists the arithmetic operators

# Arithmetic Operator

Operator	Meaning	Example	Result
+	Addition	10 + 2	12
-	Subtraction	10 - 2	8
*	Multiplication	10 * 2	20
/	Division	10 / 2	5
%	Modulus (remainder)	10 % 2	0
++	Increment	a++ (consider a = 10)	11
--	Decrement	a-- (consider a = 10)	9
+=	Addition Assignment	a += 10 (consider a = 10)	20
-=	Subtraction assignment	a -= 10 (consider a = 10)	0
*=	Multiplication assignment	a *= 10 (consider a = 10)	100
/=	Division assignment	a /= 10 (consider a = 10)	1
%=	Modulus assignment	a %= 10 (consider a = 10)	0

Assume integer variable A holds 10 and variable B holds 20, then –

Show Examples [↗](#)

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

# Increment & Decrement Operator

The **operator (++)** and the **operator (--)** are Java's increment and decrement operators.

The **increment (++)** and **decrement operator (--)** are simply used to increase and decrease the value by one.

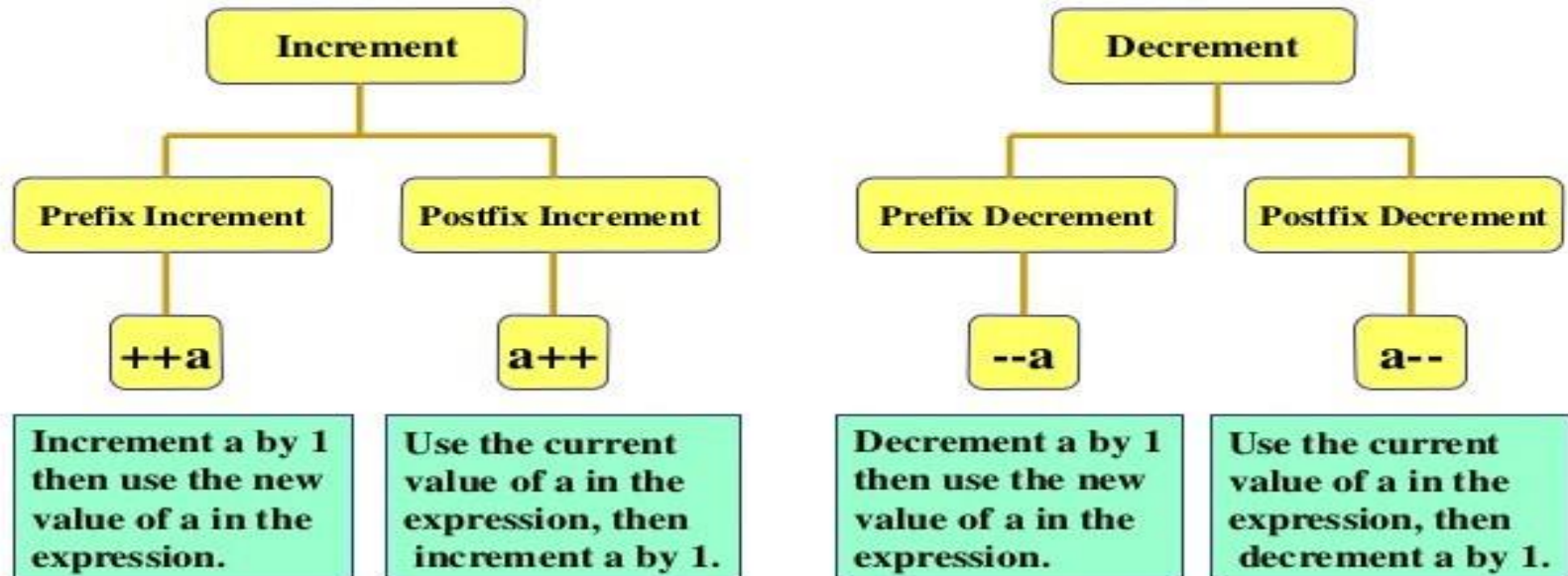
The increment operator increases its operand by one and the decrement operator simply decreases its operand by one. For instance, the statement :

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

can be rewritten like this by use of the increment operator i.e., ++:

```
x++;
```

# Increment and Decrement Operator





```
// Java program to illustrate
// Increment and Decrement operators
// Can be applied to variables only

public class Test {
    public static void main(String[] args)
    {
        int a = 10;
        int b = ++a;

        // uncomment below line to see error
        // b = 10++;

        System.out.println(b);
    }
}
```

Output:

# Relational Operator

**Relational operators** also known as **comparison operators** are used to check the relation between two operands.

In other words, the operators used to make comparisons between two operands are called ***relational operators***.

**Relational operators** are **binary operators** and hence require **two operands**.

The result of a **relational operation** is a **Boolean value** that can only be **true** or **false** according to the result of the comparison.

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is 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.	(A >= B) is not 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.	(A <= B) is true.

## //WRITE A PROGRAM TO IMPLEMENT ARITHMETIC OPERATOR

```
public class Test
{
    public static void main(String args[])
    {
        int a = 10;
        int b = 20;

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

# Logical Operator

**Logical operators** are used to combine one or more **relational expressions** that results in formation of complex expressions known as **logical expressions**.

**Like relational operators**, the **logical operators** evaluate the result of logical expression in terms of **Boolean values** that can only be **true or false** according to the result of the logical expression.

## Logical Operators

Operators	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT

The logical operators && and || are used when we want to form compound conditions by combining two or more relations. Logical operators return results indicated in the following table.

X	Y	X && Y	X    Y
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

## //WRITE A PROGRAM TO IMPLEMENT LOGICAL OPERATOR

```
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));
    }
}
```

# Assignment Operators

The **assignment operator (=)** is the most commonly used **binary operator** in Java.

It evaluates the **operand** on the **right hand** side and then assigns the resulting value to a variable on the **left hand side**.

The right operand can be a variable, constant, function call or expression.

The type of right operand must be type compatible with the left operand. The general form of representing assignment operator is

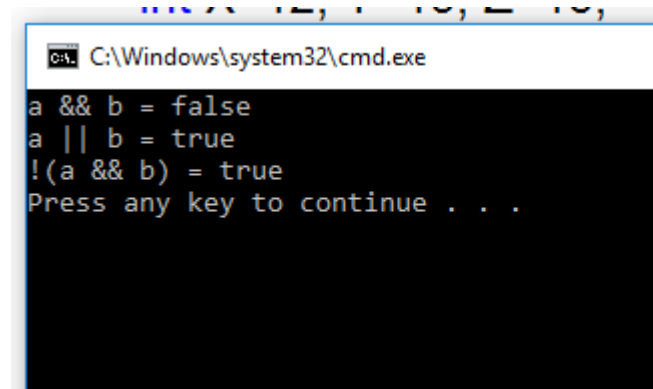


# Shorthand assignment operator

Operator	Usage	Effect
+=	a+=b;	a=a+b;
-=	a-=b;	a=a-b;
*=	a*=b;	a=a*b;
/=	a/=b;	a=a/b;
%=	a%=b;	a=a%b;
&=	a&=b;	a=a&b;
=	a =b;	a=a b;
^=	a^=b;	a=a^b;
<<=	a<<=b;	a=a<<b;
>>=	a>>=b;	a=a>>b;
>>>=	a>>>=b;	a=a>>>b;

//Java Example to implement assignment operations

```
class AssignmentOperator
{
    public static void main(String args[])
    {
        int X=12, Y=13, Z=16;
        System.out.println("The Assignment Value is : ");
        X+=2;
        Y-=2;
        Z*=2;
        System.out.println("The Value of X is : " +X);
        System.out.println("The Value of Y is : " +Y);
        System.out.println("The Value of Z is : " +Z);
    }
}
```



The screenshot shows a Windows command prompt window with the title bar "C:\Windows\system32\cmd.exe". The window contains the following text:

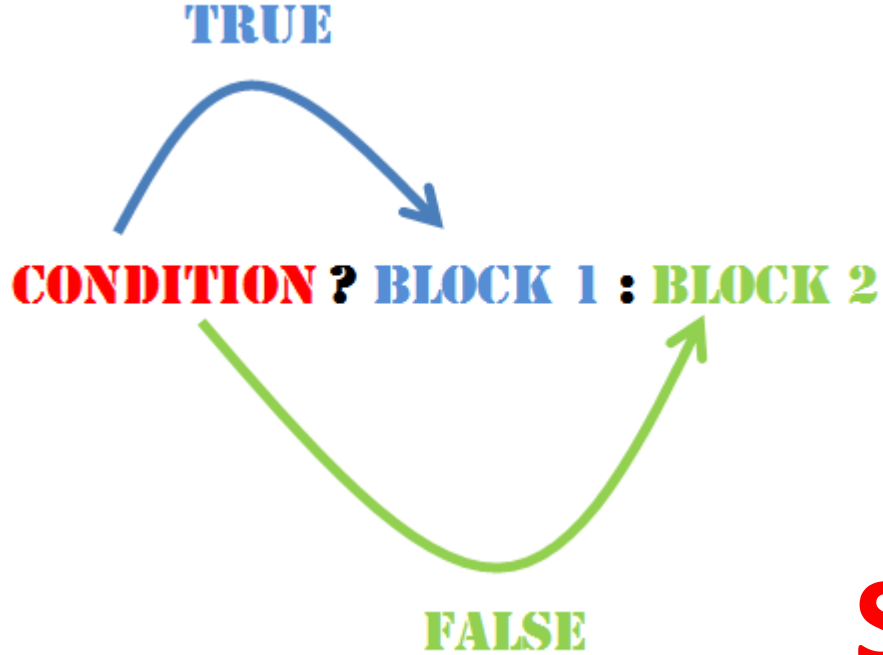
```
a && b = false
a || b = true
!(a && b) = true
Press any key to continue . . .
```

# Ternary Operators

The **Java ternary operator** functions like a simplified [Java if](#) statement.

The **ternary operator** consists of a condition that evaluates to either **true or false**, plus a value that is returned **if the condition is true** and another value that is returned if the condition is **false**.

**Here is a simple Java ternary operator example:**



**SYNTAX**

C= a>b ? A : B

## //Java Example to implement TERNARY operations

```
class AssignmentOperator
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        int a=2;
```

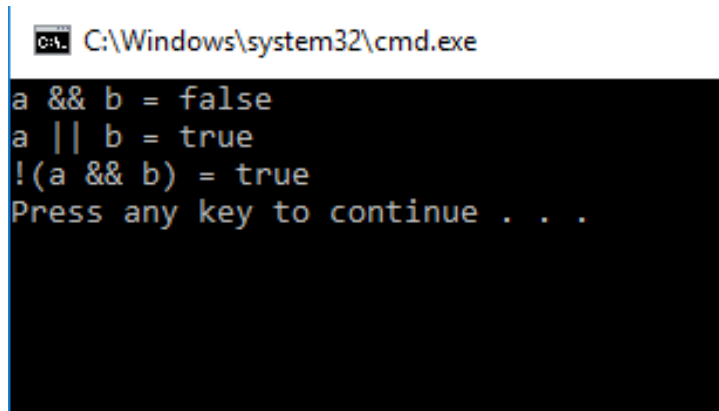
```
        int b=5;
```

```
        int min=(a<b)?a:b;
```

```
        System.out.println(min);
```

```
    }
```

```
}
```



```
C:\Windows\system32\cmd.exe
a && b = false
a || b = true
!(a && b) = true
Press any key to continue . . .
```

# BITWISE OPERATORS

- Java's *bitwise* operators operate on individual bits of integer (int and long) values.
- If an operand is shorter than an int, it is promoted to int before doing the operations.



# BITWISE OPERATOR

Bitwise operators are used to perform manipulation of individual bits of a number. They can be used with any of the integral types (char, short, int, etc). They are used when performing update and query operations of Binary indexed tree.

# BITWISE OPERATORS

Operator	Name	Description
<code>a&amp;b</code>	and	1 if both bits are 1.
<code>a b</code>	or	1 if either bit is 1.
<code>a^b</code>	xor	1 if both bits are different.
<code>~a</code>	not	Inverts the bits.
<code>n &lt;&lt; p</code>	left shift	Shifts the bits of n left p positions. Zero bits are shifted into the low-order positions.
<code>n &gt;&gt; p</code>	right shift	Shifts the bits of n right p positions. If n is a 2's complement signed number, the sign bit is shifted into the high-order positions.
<code>n &gt;&gt;&gt; p</code>	right shift	Shifts the bits of n right p positions. Zeros are shifted into the high-order positions.



# BITWISE OPERATOR

Bitwise operators are used to perform manipulation of individual bits of a number. They can be used with any of the integral types (char, short, int, etc). They are used when performing update and query operations of Binary indexed tree.

## 1. Bitwise OR (|) –

This operator is binary operator, denoted by '|'. It returns bit by bit OR of input values, i.e, if either of the bits is 1, it gives 1, else it gives 0.

For example,

```
a = 5 = 0101 (In Binary)
```

```
b = 7 = 0111 (In Binary)
```

```
Bitwise OR Operation of 5 and 7
```

```
  0101
```

```
| 0111
```

---

```
0111 = 7 (In decimal)
```

# BITWISE OPERATOR

## 2. Bitwise AND (&) –

This operator is binary operator, denoted by '&'. It returns bit by bit AND of input values, i.e, if both bits are 1, it gives 1, else it gives 0.

For example,

```
a = 5 = 0101 (In Binary)
```

```
b = 7 = 0111 (In Binary)
```

Bitwise AND Operation of 5 and 7

```
0101
```

```
& 0111
```

---

```
0101 = 5 (In decimal)
```

# BITWISE OPERATOR

## 3. Bitwise XOR (^) –

This operator is binary operator, denoted by '^'. It returns bit by bit XOR of input values, i.e, if corresponding bits are different, it gives 1, else it gives 0.

For example,

```
a = 5 = 0101 (In Binary)
```

```
b = 7 = 0111 (In Binary)
```

```
Bitwise XOR Operation of 5 and 7
```

```
0101
```

```
^ 0111
```

---

```
0010 = 2 (In decimal)
```

# BITWISE OPERATOR

## 4. Bitwise Complement (~) –

This operator is unary operator, denoted by '~'. It returns the one's complement representation of the input value, i.e, with all bits inversed, means it makes every 0 to 1, and every 1 to 0.

For example,

```
a = 5 = 0101 (In Binary)
```

```
Bitwise Complement Operation of 5
```

```
~ 0101
```

```
1010 = 10 (In decimal)
```

## // Java program to illustrate bitwise operators

```
public class operators
{
    public static void main(String[] args)
    {
        //Initial values
        int a = 5;
        int b = 7;

        // bitwise and
        // 0101 & 0111=0101 = 5
        System.out.println("a&b = " + (a & b));

        // bitwise or
        // 0101 | 0111=0111 = 7
        System.out.println("a | b = " + (a | b));

        // bitwise xor
        // 0101 ^ 0111=0010 = 2
        System.out.println("a^b = " + (a ^ b));
    }
}
```

```
// bitwise and
// ~0101=1010
// will give 2's complement of 1010 = -6

System.out.println("~a = " + ~a);

// can also be combined with
// assignment operator to provide shorthand
// assignment
// a=a&b

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

# instanceOf Operator

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The **instanceof** in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false.

If we apply the **instanceof** operator with any variable that has null value, it returns false.

```
class Simple1
{
    public static void main(String args[])
    {
        Simple1 s=new Simple1();
        System.out.println(s instanceof Simple1);//true
    }
}
```

---

# **Java bitwise Shift Operators**

---



## Bitwise Shift Operators:

### >> (Signed right shift):

In Java, the operator ">>" is signed right shift operator. All integers are signed in Java, and it is fine to use >> for negative numbers. The operator ">>" uses the sign bit (left most bit) to fill the trailing positions after shift. If the number is negative, then 1 is used as a filler and if the number is positive, then 0 is used as a filler. In simple terms it will divide the number by 2 to power number of shifted bit

```
int a =8; //binary representation is 1000
System.out.println(a>>1); //This will print 4 (binary representation 0100)
```

### << (Signed left shift):

This operator moves all bits to the left side (simply multiply the number by two to power number of bits shifted).

```
int a =8; //binary representation is 1000
System.out.println(a<<2); //This will print 32 (binary representation 100000)
```

### >>> (Unsigned right shift) :

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. First bit represents a sign of the integer.

```
int a =-2; // This is represented in binary as 10000000 0000000000000000 000000010
System.out.println(a>>>1);
//01000000 00000000 00000000 00000001 in decimal=2147483647
```

Below program demonstrate shift operators.

```

1 public class ShiftOperatorDemo {
2     public static void main(String[] args) {
3         int a = 34; //binary representation 00000000 00000000 00000000 001000
4         int b = -20; //binary representation 10000000 00000000 00000000 000101
5         System.out.println("Signed Right Shift 34 devide by 2= " + (a>>1) );
6         System.out.println("Signed Right Shift -20 devide by 2= " + (b>>1) );
7         System.out.println("Signed Left Shift 34 multiply by 2= " + (a<<1) );
8         System.out.println("Signed Letf Shift -20 multiply by 2= " + (b<<1) );
9         System.out.println("unsigned Right Shift of 34 = " + (a>>>1) );
10        System.out.println("unsigned Right Shift of -20 = " + (b>>>1) );
11    }
12 }

```

```

Signed Right Shift 34 devide by 2= 17
Signed Right Shift -20 devide by 2= -10
Signed Left Shift 34 multiply by 2= 68
Signed Letf Shift -20 multiply by 2= -40
unsigned Right Shift of 34 = 17
unsigned Right Shift of -20 = 2147483638

```

# Precedence of Java Operators

**Certain operators** have **higher precedence** than others; for example, the **multiplication** operator has **higher precedence** than the **addition operator**.

For example,  $x = 7 + 3 * 2$ ; here  $x$  is assigned **13**, not **20** because operator  $*$  has higher precedence than  $+$ , so it first gets multiplied with  $3 * 2$  and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Within an expression, higher precedence operators will be evaluated first.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression --expression +expression - expression ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^=  = <<= >>= >>>=	Right to left