ShreeSoft INFORMATICS

Address: Tirumala Plaza, Upendra Nagar, Cidco, Nashik.

Join us on Telegram:
https://t.me/ShreeSoftinformatics

Join us on WhatsApp:
https://chat.whatsapp.com/BrLM0ZoWMbpA8nXi9Mrocl

Contact: **+91 8308341531**

Language Fundamental of Java

Identifier

A name in java program element is called as identifier which can be used for identification purpose. It can be method name, variable name, class name or label name.

Example

```
class Demo
{
        Public static void main (String [] args)
    {
        int a=7;
    }
}
```

Rules for naming Identifiers

1. The only allowed characters in java identifiers are a-z, A-Z, 0 -9, $ and _.
   Ex - core_java → ✓
   core# → ×
   If we are using any other character we will get compile time error.

2. Identifers can't start with digit.
   Ex - Demo123 ✓
   123demo ✗

3. Java identifiers are case sensitive of course java language itself treated as case sensitive programming language.
   Ex
   ```
   Class Demo
   {
           int number=7;
           int Number=7;
           int NUMBER=7;
   }
   ```
   } we can differentiate with respect to case.

4. There is no length limit for java identifier but it is not recommended to take to lengthy identifiers.

5. We can't use reserved words as identifiers.

Ex - int x =7;
int if = 7;

6. All predefined java class name and interface names we can use as identifiers.

Ex:

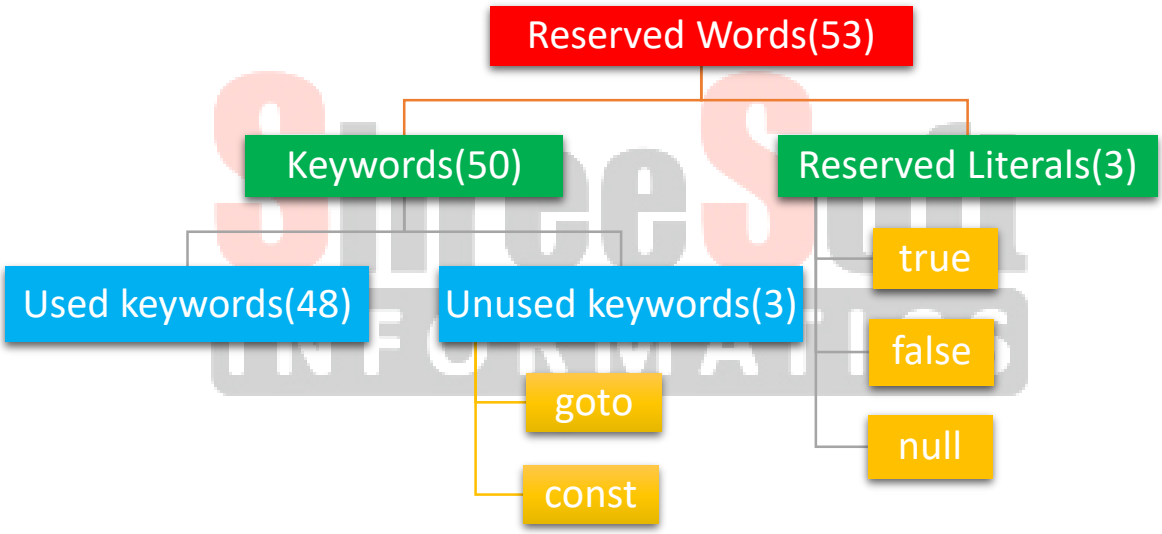
```
class Test
{
    Public static void main(String[] args)
    {
        int String = 11;              |    int Runnable = 77;
        System.out.println(String);   |    System.out.println(Runnable);
    }
}
```

Output: 11 or 77

Even though it is valid but it is not a programming practice because it reduces readability and creates confusion.

Keywords or Reserved words:

In java some identifiers are reserved to associate some functionality or meaning such type of reserved identifiers are called reserved words.



Reserved words for data types: (8)	Reserved words for flow control: (11)
1) byte 2) short 3) int 4) long 5) float 6) double 7) char 8) boolean	1) if 2) else 3) switch 4) case 5) default 6) for 7) do 8) while 9) break 10) continue 11) return

Keywords for modifiers (11)	Keywords for Exception Handling (6)	Keywords related class (6)	Keywords related object (4)
1) public 2) private 3) protected 4) static 5) final 6) abstract 7) synchronized 8) native 9) strictfp (1.2 version) 10) transient 11) volatile	1) try 2) catch 3) finally 4) throw 5) throws 6) assert (1.4 version)	1) class 2) package 3) import 4) extends 5) implements 6) interface	1) new 2) instanceof 3) super 4) this

void return type keyword:
If a method won't return anything compulsory that method should be declared with the void return type in java but it is optional in C++.

Unused keywords:
goto: Create several problems in old languages and hence it is banned in java.
const: Use final instead of this. By mistake if we are using these keywords in our program we will get compile time error.

Reserved literals:
1) true
2) false
} Values for boolean data type.
3) null - default value for object reference.

Enum: This keyword introduced in 1.5v to define a group of named constants
Example: enum Beer { KF, RC, KO, FO };

Separators in Java

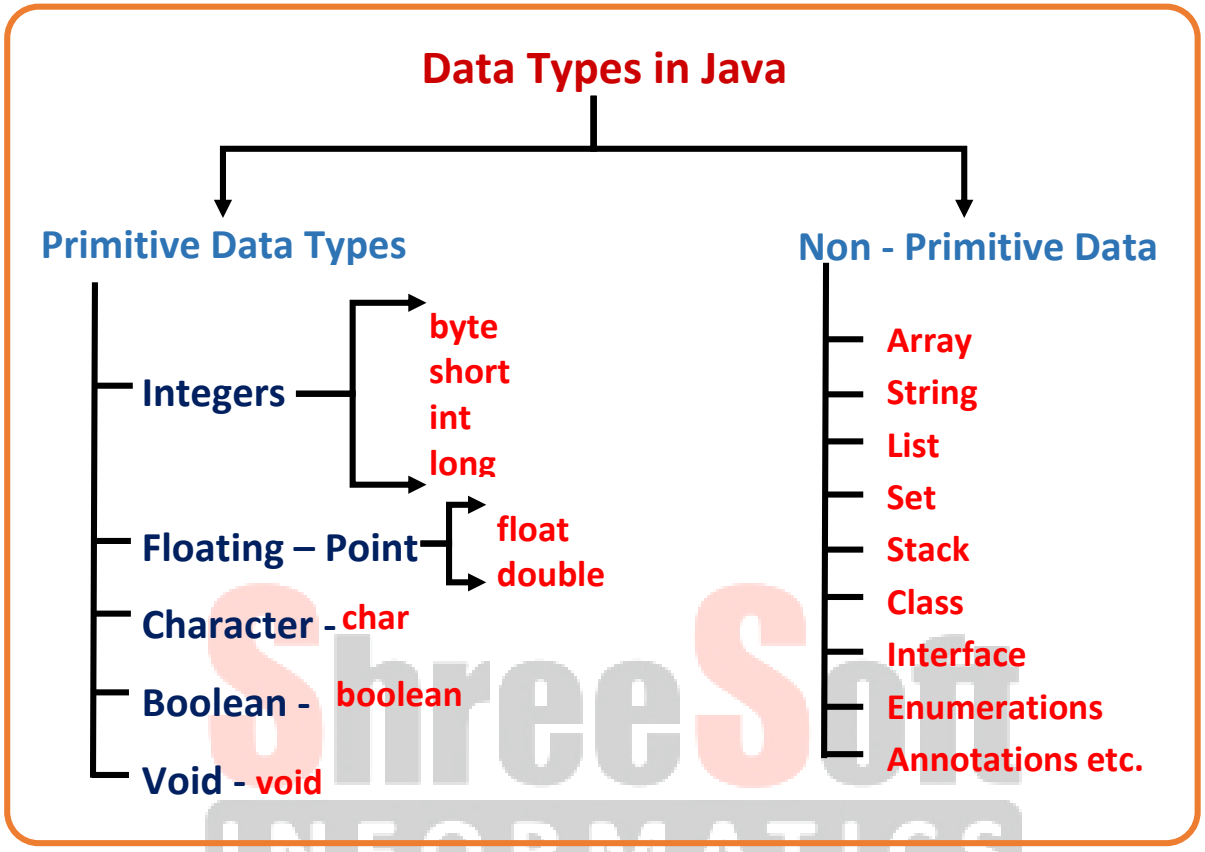
Separators help us defining the structure of a program. In Java, There are few characters used as separators. The most commonly used separator in java is a semicolon (;).

Separator	Name	Usage
.	Period	It is used to separate the package name from sub-package name & class name. It is also used to separate variable or method from its object or instance.
,	Comma	It is used to separate the consecutive parameters in the method definition. It is also used to separate the consecutive variables of same type while declaration.
;	Semicolon	It is used to terminate the statement in Java.
()	Parenthesis	This holds the list of parameters in method definition. Also used in control statements & type casting.
{}	Braces	This is used to define the block/scope of code, class, methods.
[]	Brackets	It is used in array declaration.

Data types:

In Java, Each and Every variable has a type, every expression has a type and all types are strictly define more over every assignment should be checked the type compatibility by the compiler. Hence java language is considered as strongly typed programming language.

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used.



Type Name	Size	Range	Default value
byte	1 byte	-128 to 127 (2^{-7} to 2^{7}-1)	0
short	2 byte	-32,768 to 32,767 (2^{-15} to 2^{15}-1)	0
int	4 byte	-2,147,483,648 to 2,147,483,647 (2^{-31} to 2^{31}-1)	0
long	8 byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (2^{-63} to 2^{63}-1)	0
float	4 byte	1.4E-45 to 3.4028235E38	0.0
double	8 byte	4.9E-324 to 1.7976931348623157E308	0.0
char	2 byte	0 to 65,536	'\u0000'
boolean	1 bit	true, false	false
void	0 byte	--	--

Note: Except boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and"-ve" numbers.

Write a program to find the memory size and range of datatype.

```
package PrimitiveDatatypes;
public class DatatypeRange {

	public static void main(String[] args) {
		System.out.println("Datatype\tSize\tRange");
		System.out.println("byte\t\t"+Byte.SIZE/8+"\t"+Byte.MIN_VALUE+"to"+Byte.MAX_VALUE);
		System.out.println("short\t\t"+Short.SIZE/8+"\t"+Short.MIN_VALUE+"to"+Short.MAX_VALUE);
		System.out.println("Integer\t\t"+Integer.SIZE/8+"\t"+Integer.MIN_VALUE+" to "+Integer.MAX_VALUE);
		System.out.println("Long\t\t"+Long.SIZE/8+"\t"+Long.MIN_VALUE+" to "+Long.MAX_VALUE);
		System.out.println("Float\t\t"+Float.SIZE/8+"\t"+Float.MIN_VALUE+" to "+Float.MAX_VALUE);
		System.out.println("Double\t\t"+Double.SIZE/8+"\t"+Double.MIN_VALUE+" to "+Double.MAX_VALUE);
	}
}
```

Integral Data type:

1) **byte:** Stores whole number from -128 to +127. The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.

   Example:

```
byte b=7;

byte b2=135; // C.E:possible loss of precision found : int required : byte

byte b=13.7; // C.E:possible loss of precision

byte b=true; // C.E:incompatible types

byte b="shree"; // C.E:incompatible types found : java.lang.String required : byte
```

> **Note:** byte data type is also best suitable if we are handling data in terms of streams either from the file or from the network.

2) **short:** As with byte, the same guidelines apply we can use a short to save memory in large arrays, in situations where the memory savings actually matters.

   Example:

```
short s=130;
short s=32768; // C.E:possible loss of precision
short s=true; // C.E:incompatible types
```

3) **int:**
By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of $2^{31}-1$. In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of $2^{32}-1$. Use the Integer class to use int data type as an unsigned integer.

We will see more information about the Number Class in the chapter Wrapper Classes. Static methods like compareUnsigned, divideUnsigned etc have been added to the Integer class to support the arithmetic operations for unsigned integers.

4) **Example:**

```
int x=130;
int x=10.5; // C.E:possible loss of precision
int x=true; // C.E:incompatible types
```

5) **long:** The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -2^{63} and a maximum value of $2^{63}-1$.
In Java SE 8 and later, we can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$. Whenever int is not enough to hold big values then we should go for long data type.
The Long class also contains methods like compareUnsigned, divideUnsigned etc. to support arithmetic operations for unsigned long. We will see the usage of this method in wrapper class.

Example 1: When we calculate factorial of more than 14 -15 value that time int data type is not enough. Hence we can take factorial result variable of type long as

```
long n=15,f=1;
while(n!=0)
{
    f=f*n;
    n--;
}
System.out.println("Factorial:"+f);
```

Example 2: To hold the number of characters present in a big file. int may not enough hence the return type of length() method is long.

```
long len=f.length(); //f is a file
```

Note: All the above data types (byte, short, int and long) can be used to represent whole numbers. If we want to represent real numbers then we should go for floating point data types.

Floating Point Datatypes:

1) **float:** If we want to 5 to 6 decimal places of accuracy then we should go for float. float follows single precision.
2) **double:** If we want to 14 to 15 decimal places of accuracy then we should go for double. double follows double precision.

Char data type:

1) C & C++ languages follows ASCII code based language which contains 256 (0 to 255) characters. To store character in c & c++ language 1 byte (8 Bits) memory size is enough.
2) But java allowed to use any worldwide character set follows by Unicode based character set which contains between 65536 (0 to 65535) characters. To store Unicode character we need 2 bytes memory size.

Example:

```
char ch1=78; // ch1='M'
char ch2=65536;//C.E: possible loss of precision
```

boolean data type:

The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

Example 1:

```
boolean b=true;
Boolean b=True;        //C.E:cannot find symbol
boolean b="True";      //C.E:incompatible types
boolean b=0;           //C.E:incompatible types
```

Example 2:

```
short flag=0;
if(flag) {
    System.out.println("Welcome");
}
else {
    System.out.println("Good Bye");
}
```

Compile time Error:
Incompatible types
Found: int
Required: boolean

```
While(1) {
    System.out.println("ShreeSoft");
}
```

Java is pure object oriented programming or not?

Java is not considered as pure object oriented programming language because several oops features (like multiple inheritance, operator overloading) are not supported by java moreover we are depending on primitive data types which are non-objects. So java is not pure object oriented programming language.

Variable:

A variable is a named memory location used to store a data value. A variable can be defined as a container that holds the value while the Java program is executed. A variable is assigned with a datatype. It is the basic unit of storage in a program.

1. The value can be access using variable name.
2. The value stored in a variable can be changed during program execution.
3. A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.

Syntax to declare a variable:

datatype var1 [= value][, var2 [= value] …];

Example:

```
byte b; // declare one variable b of type byte.
int x=13, y=7; // declare 2 variable x and y of type int with initializing value 13 and 7.
float pi=3.14; // declare variable pi of type float having value 3.14
char ch='M'; // declare variable ch of type char having value 'M'.
```

Types of variable in Java:

1) Instance variable (Non Static Field)
2) Class variable (Static Field)
3) Local variable
4) Parameters

1) Instance Variables (Non-Static Fields):

Instance variables are those variables that are declared inside a class. These variables cannot be declared within a block, method, or constructor. Non-static fields are also known as instance variables because their values are unique to each instance of a class (to each object, in other words) i.e. Each instance (objects) of a class has its own copy of the instance variable.

Properties of the Instance variable:

1. They are declared within a class but outside a block, method or constructor.
2. It cannot be defined by a static keyword.
3. Unlike Local variables, these variables have a default value.
4. The integer type has a default value '0' and the boolean type has the default value 'false'.
5. Unlike Local variables, we have access modifiers for Instance variables.
6. The memory allocated to store instance variables is Heap memory.

Example 1:

```
public class InstanceVariable1 {
    // Declaring instance variables
    public int rollNum;
    public String name;
    public int totalMarks;
    public int number;

    public static void main(String[] args) {
        // created object
        InstanceVariable1 in = new InstanceVariable1 ();
        in.rollNum = 95;
```

```
                in.name = "Saket";
                in.totalMarks = 480;

                // printing the created objects
                System.out.println(in.rollNum);
                System.out.println(in.name);
                System.out.println(in.totalMarks);
                /* we did not assign the value to number so it will print '0' by default */
                System.out.println(in.number);
        }
    }
```

Example 2:

```
public class InstanceVariable2 {

        public static void main(String[] args) {

                Employee employee = new Employee();

                // Before assigning values to employee object
                System.out.println(employee.empName);
                System.out.println(employee.id);
                System.out.println(employee.age);

                employee.empName = "Ramesh";
                employee.id = 100;
                employee.age = 28;

                // After assigning values to employee object
                System.out.println(employee.empName);
                System.out.println(employee.id);
                System.out.println(employee.age);
    }
}
class Employee {

        // instance variable employee id
        public int id;

        // instance variable employee name
        public String empName;

        // instance variable employee age
        public int age;
}
```

2) Class Variables (Static Fields):

The variable which is declared inside class with static modifier is called as Static fields or class variable.

Sometimes, we want to have variables that are common to all objects. This is accomplished with the static modifier. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

Properties of the Class variable:

1. These variables cannot be local variable.
2. Static variables are shared among all the instances of a class.
3. The default values of Static/Class variables are the same as the Instance variables.
4. Static variables can be used within a program by calling the **className.variableName**
5. The memory allocated to store Static variables is Static memory.

Example 1:

```
public class StaticVariable {
        // radius is declared as private static
        private static int radius;

        // pi is a constant of type double declared as static
        private static final double pi = 3.14;

        public static void main(String[] args) {
                // assigning value of radius
                radius = 7;

                // calculating and printing circumference
                System.out.println("Circumference of a circle is: " + 2*pi*radius);
    }
}
```

Example 2:

```
public class StaticVariableExample {

        public static void main(String[] args) {
                Student s1 = new Student(100, "Manoj");
                Student s2 = new Student(101, "Suhas");
                Student s3 = new Student(102, "Tushar");
                Student s4 = new Student(103, "Atul");

                System.out.println("******Student Details **********");
                System.out.println(s1.toString());
                System.out.println(s2.toString());
                System.out.println(s3.toString());
                System.out.println(s4.toString());
        }
}
class Student {
```

```
        private int rollNo;
        private String name;
        private static String college = "ShreeSoft Informatics"; // static variable
        public Student(int rollNo, String name) {
                super();
                this.rollNo = rollNo;
                this.name = name;
        }
        @Override
        public String toString() {
                return "Student [rollNo=" + rollNo + ", name=" + name + ", college=" + college +
        "]";
        }
}
```

3) Local Variables:

A variable which is declared within any method, constructor or block is called as local variable.

Properties of the Class variable:

1. No access modifiers for local variables.
2. These can be used only within the same block, method, or constructor where it is initialized.
3. No default value after you have declared your local variable. You need to initialize your declared local variable.
4. It can't be defined by a static keyword.

Are static local variables allowed in Java?

Static local variables are not allowed in java because the scope of local variable is limited to function, it violets the purpose of static modifier. Hence compiler does not allow static local variable.

Example:

```
public class Local {
        public void calculate() {
                // initialized a local variable radius
                int radius = 7;

                // calculating area of circle
                Area = 3.14 * radius * radius;
                System.out.println("Area of Circle: " +area);
        }
        public static void main(String args[]) {
                // a is a reference used to call calculate() method
                local a = new local();
                a.calculate();
        }
}
```

4) Parameters:

A variable which is declared within any function header is called as Parameter

Example:

```
public class Local {
        public void calculate(int radius) {
                Area = 3.14 * radius * radius;
                System.out.println("Area of Circle: " +area);
        }
        public static void main(String args[]) {
                int r=7;
                local a = new local();
                a.calculate(r);
        }
}
```

Scope and Life of Variables in Java:

1) Instance Variables:

A variable which is declared inside a class and outside all the methods and blocks is an instance variable. The general scope of an instance variable is throughout the class except in static methods. The lifetime of an instance variable is until the object stays in memory.

2) Class Variables:

A variable which is declared inside a class, outside all the blocks and is marked static is known as a class variable. The general scope of a class variable is throughout the class and the lifetime of a class variable is until the end of the program or as long as the class is loaded in memory.

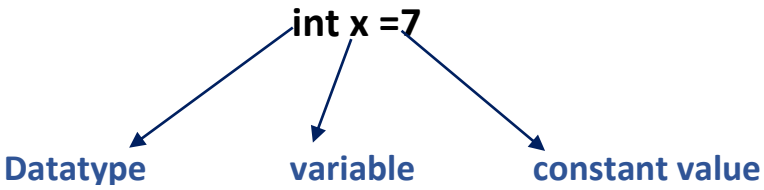
3) Local Variables:

All other variables which are not instance and class variables are treated as local variables including the parameters in a method. Scope of a local variable is within the block in which it is declared and the lifetime of a local variable is until the control leaves the block in which it is declared.

Literals in Java:

Any constant value which can be assigned to the variable is called literal.

Example:



Integral Literals:

Case 1: For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

1) **Decimal literals:** In this allowed digits are 0 to 9.
   Example:
   ```
   int x=7;
   ```

2) **Octal literals:** Allowed digits are 0 to 7. Literal value should be prefixed with zero.
   Example:
   ```
   int x=010;
   ```

3) **Hexa Decimal literals:**
   - The allowed digits are 0 to 9, A to Z. For the
   - Extra digits we can use both upper case and lower case characters.
   - This is one of very few areas where java is not case sensitive.
   - Literal value should be prefixed with 0x (or) 0X.

   Example:
   ```
   int x=0x10;
   ```

   Example 1:
   ```
   public class Integer_Literals {

       public static void main(String args[]) {
           int decimal=7;
           int octal=011;
           int hexa=0X10;
           System.out.println("Decial:"+decimal);
           System.out.println("Octal:"+octal);
           System.out.println("HexaDecial:"+hexa);
       }
   }
   ```

Case 2: By default every integral literal is int type but we can specify explicitly as long type by suffixing with small "l" (or) capital "L".

Example:

int x=13; → valid
long l=13L; → valid
long l=13; → valid
int x=13l; → //C.E:possible loss of precision(invalid) found : long required : int

Case 3: There is no direct way to specify byte and short literals explicitly. But whenever we are assigning integral literal to the byte variables and its value within the range of byte compiler automatically treats as byte literal. Similarly short literal also.

Example:

byte b=127; → valid
byte b=130; → // C.E:possible loss of precision(invalid)
short s=32767; → valid
short s=32768; → // C.E:possible loss of precision(invalid)

Floating Point Literals:

Case 1: Floating point literal is by default double type but we can specify explicitly as float type by suffixing with f or F.

Example:

```
float f=123.456;   → //C.E: Possible loss of precision
float f=123.456f;  → valid
double d=123.456; → valid
```

Case 2: We can specify explicitly floating point literal as double type by suffixing with d or D.

Example:

```
double d=789.654D;
```

Case 3: We can specify floating point literal only in decimal form and we can't specify in octal and hexadecimal forms.

Example:

```
double d=123.456;    → valid
double d=0123.456;   → valid //it is treated as decimal value but not octal
double d=0x123.456; → Invalid //C.E:malformed floating point literal
```

Case 4: We can assign integral literal directly to the floating point data types and that integral literal can be specified in decimal, octal and Hexadecimal form also.

Example:

```
double d=0xBeef;
System.out.println(d); //48879.0
```

Note: But we can't assign floating point literal directly to the integral types.

Case 5: We can specify floating point literal even in exponential form also (significant notation).

Example:

```
double d=10e2;  → valid //10*102
System.out.println(d);   → //1000.0
float f=10e2;  → Invalid // C.E: Possible loss of precision
float f=10e2F;   → valid
```

Boolean literals:

The only allowed values for the boolean type are true or false where case is important i.e. lower case

Example:

```
boolean b=true;   → valid
boolean b=0;      → //C.E:incompatible types
boolean b=True;   → //C.E:cannot find symbol
boolean b="true";   → //C.E:incompatible types
```

Char literals:

Case 1: A char literal can be represented as single character within single quotes.

Example:

```
char ch='a';      → valid
char ch=a;        → //C.E:cannot find symbol
char ch="a";    → //C.E:incompatible types
char ch='ab';   → //C.E:unclosed character literal(invalid)
```

Case 2: We can specify a char literal as integral literal which represents Unicode of that character. We can specify that integral literal either in decimal or octal or hexadecimal form but allowed values range is 0 to 65535.

Example:

```
char ch=97;   → valid
char ch=0xFace;   → valid
System.out.println(ch); →  //
char ch=65536;    → //C.E: possible loss of precision
```

Case 3: We can represent a char literal by Unicode representation which is nothing but '\uxxxx'. (4 digit hexa-decimal number).

Example:

```
char ch='\ubeef';
char ch1='\u0061';
System.out.println(ch1);   →  //a
char ch2=\u0062;    → //C.E:cannot find symbol
char ch3='\iface';     →  //C.E:illegal escape character
```

Case 4: Every escape sequence character in java acts as a char literal.

Example:

```
char ch='\n';    → // valid
char ch='\l';    →  //C.E:illegal escape character
```

String literals:

Any sequence of characters with in double quotes is treated as String literal.

Example:

```
String s="Ashok";   → valid
```

null Literals:

- ❒ null is a reserved word (keyword) in Java for literal values.
- ❒ It is a literal similar to the true and false.
- ❒ It is just a value that shows that the object is referring to nothing. The invention of the word "null" originated to denote the absence of something.
- ❒ For example, the absence of the user, a resource, or anything. But, over the years it puts Java programmers in trouble due to the disturbing null pointer exception. When you declare a boolean variable, it gets its default value as false.

Case 1: The reserved word null is case sensitive and cannot be written as Null or NULL as the compiler will not recognize them and will certainly give an error.

Example:

```
public class Test
{
   public static void main (String[] args) throws java.lang.Exception {
      // compile-time error : can't find symbol 'NULL'
      Object obj = NULL;

      //runs successfully
      Object obj1 = null;
   }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Null cannot be resolved to a variable
at Test.main(Test.java:6)
```

Case 2: Any reference variable in Java has a default value **null**.

```
public class Test
{
   private static Object obj;
   public static void main(String args[])
   {
      // it will print null;
      System.out.println("Value of object obj is : " + obj);
   }
}
```

Output:

```
Value of object obj is : null
```

Case 3: Unlike the common misconception, null is not Object or neither a type. It's just a special value, which can be assigned to any reference type and you can type cast null to any type

Examples:

```
   String s1 = null;
   Integer i1 = null;
   Double d1 = null;

   // null can be type cast to String
```

```
String s1 = (String) null;

// it can also be type casted to Integer
Integer i1 = (Integer) null;

// yes it's possible, no error
Double d1 = (Double) null;
```

Case 4: Autoboxing and unboxing → During auto-boxing and unboxing operations, compiler simply throws Nullpointer exception error if a null value is assigned to primitive boxed data type.
Example:

```
public class Test {
    public static void main(String[] args) throws java.lang.Exception {
        Integer i = null;

        // Unboxing null to integer throws NullpointerException
        int a = i;
    }
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "i" is null
at Test.main(Test.java:6)
```

Case 5: 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). At run time, the result of the instanceof operator is true if the value of the Expression is not null. This is an important property of instanceof operation which makes it useful for type casting checks.
Example:

```
public class Test {
    public static void main(String[] args) throws java.lang.Exception {
        Integer i = null;
        Integer j = 10;

        // prints false
        System.out.println(i instanceof Integer);

        // Compiles successfully and print true
        System.out.println(j instanceof Integer);
    }
}
```

Output:

```
false
true
```

Case 6: Static vs Non static Methods → We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with

reference variables with null values. Since static methods are bonded using static binding, they won't throw Null pointer Exception.
Example:

```
public class Test {
   public static void main(String args[]) {
      Test obj = null;
      obj.staticMethod();
      obj.nonStaticMethod();
   }

   private static void staticMethod() {
      // Can be called by null reference
      System.out.println("static method, can be called by null reference");
   }

   private void nonStaticMethod() {
      // Can not be called by null reference
      System.out.print("Non - static method -cannot be called by null reference");
   }
}
```

Output:

```
static method, can be called by null reference
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"Test.nonStaticMethod()" because "obj" is null
at Test.main(Test.java:5)
```

Case 7: == and != ➔ The comparison and not equal to operators are allowed with null in Java. This can made useful in checking of null with objects in java.
Example:

```
public class Test {
   public static void main(String args[]) {
       // return true;
      System.out.println(null == null);

      // return false;
      System.out.println(null != null);
   }
}
```

Output:

```
true
false
```

Case 8: **"null" can be passed as an argument in the method :**
We can pass the null as an argument in java and we can print the same. The data type of argument should be Reference Type. But the return type of method could be any type as void, int, double or any other reference type depending upon the logic of program.
Here, the method "print_null" will simply print the argument which is passed from the main method.

Example:

```
import java.io.*;
class Test {
   public static void print_msg(String str) {
      System.out.println("Hey, I am: " + str);
   }
   public static void main(String[] args) {
      Test.print_msg(null);
   }
}
```

Output :

Hey, I am: null

Case 9: '+' operator on null

We can concatenate the null value with String variables in java. It is considered a concatenation in java.

Here, the null will only be concatenated with the String variable. If we use "+" operator with null and any other type (Integer, Double, etc.,) other than String, it will throw an error message.

Integer a=null+7 will throw an error message as "**bad operand types for binary operator '+' "**

```
import java.io.*;
class Test {
   public static void main(String[] args)
   {
      String str1 = null;
      String str2 = "_value";
      String str3 = str1 + str2;
      System.out.println("Concatenated value : "+ str3);
   }
}
```

Output

```
Concatenated value : null_value
```

Java 1.7 Version Enhancements with respect to Literals:

There are 2 Enhancement as

1) Binary Literals
2) Usage of '_' in Numeric Literals

Binary Literals:

For the integral data types untill 1.6 version we can specified literal value in the form of Decimal, Octal, Hexadecimal. But from 1.7 version onwards we can specified literal value in binary form also. The allowed digits 0 and 1. Binary Literal should be prefixed with 0b or 0B.

Example:

```
int x = 0b111;
System.out.println(x); // 7
```

Usage of _ symbol in numeric literals:

Case 1: From 1.7v onwards we can use underscore(_) symbol in numeric literals.

Example:
double d = 123456.789; → //valid
double d = 1_23_456.7_8_9; → //valid
double d = 123_456.7_8_9; → //valid

The main advantage of this approach is readability of the code will be improved at the time of compilation ' _ ' symbols will be removed automatically, hence after compilation the above lines will become double d = 123456.789

Case 2: We can use more than one underscore symbol also between the digits.
Example :
double d = 1_23_ _456.789;

Case 3: We should use underscore symbol only between the digits.
Example:
double d=_1_23_456.7_8_9; → //invalid
double d=1_23_456.7_8_9_; → //invalid
double d=1_23_456_.7_8_9; → //invalid

Constant:

- ❒ A value which is fixed and does not change during the execution of a program is called constants in java. In other words, Java constants are fixed (known as immutable) data values that cannot be changed.
- ❒ Constants are not supported directly in Java. Instead, there is an alternative way to define a constant in Java by using the static and final keywords / Modifier.

Syntax to declare in Constant

```
static final datatype variable_name = constantValue;
```

static modifier: It allows the constant to be available without loading an instance/object of its declaration class. A static modifier is used for memory management as well. It creates only one copy of variable or instance on memory which is share by all instances of class.

final modifier: It makes the variable immutable. This means the value of the variable cannot be changed.

Example:

```
static final float PI = 3.14f;
```

```
public class Constant {
  static final double PI = 3.14;
  public static void main(String[] args) {
        System.out.println("Value of PI:  " + PI);
  }
}
```

Note: According to Java naming convention, constant names in Java must be capitalized.

The static non-access modifier has no role in making the variable constant. Then why is it used?

The static keyword helps in achieving memory efficiency. Since the value of the final variable cannot be changed, there is no point in keeping a copy of the variable for each instance of the class where it is declared. Hence, the variable is associated with the class rather than an instance.

Why do we use Constants in Java?

The constants are used in Java to make the program easy to understand and readable. Since constant variables are cached by application and JVM, it improves the performance of an application by making it efficient.

Declaring Constant as Private:

Using the private access modifier before the constant name makes it inaccessible from other classes. It means the value of the constant can be accessed from within the class only.

```
class ClassA {
  private static final double PI = 3.14;
}

public class PrivateConstant {
  public static void main(String[] args) {
    System.out.println("Value of PI: " + ClassA.PI);
  }
}
```


Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The field ClassA.PI is not visible
at PrivateConstant.main(PrivateConstant.java:7)
```

Declaring Constant as Public

Using the public access modifier before the constant name makes it available anywhere in the program. It means the value of the constant can be accessed from any class or package.

```
class ClassA {
        public static final double PI = 3.14;
}
public class PublicConstant {
        public static void main(String[] args) {
            System.out.println("Value of PI: " + ClassA.PI);
        }
}
```

Output:

Value of PI: 3.14

Type Conversions

- The process of changing one type of value to another type of value is called as Type Conversion. Type conversion is also called Type Casting.
- If two data types are compatible with each other, Java will perform such conversion automatically or implicitly for you.
- We can do type conversion in java by assigning a value of one variable to a variable of another type.
- We can easily convert a primitive data type into another primitive data type by using type casting.
- Similarly, it is also possible to convert a non-primitive data type (referenced data type) into another non-primitive data type by using type casting.
- But we cannot convert a primitive data type into an advanced (referenced) data type by using type casting. For this case, we will have to use methods of Java Wrapper classes.

Types of Type of Conversion:

There are two types of conversion in java as

1) Implicit / Automatic / Widening Conversion
2) Explicit / Casting / Narrowing Conversion

Implicit / Automatic / Widening Conversion

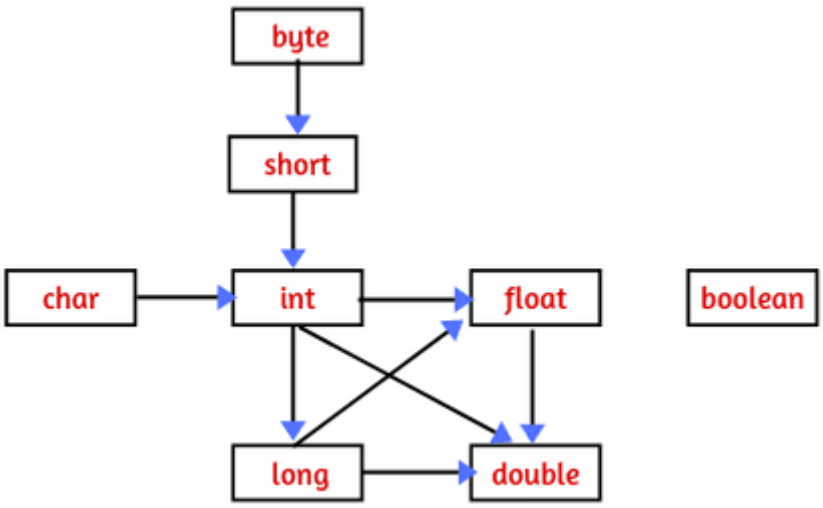
- Automatic conversion (casting) done by Java compiler internally is called implicit conversion or implicit type casting in java.
- Implicit casting is performed to convert a lower data type into a higher data type. It is also known as automatic type promotion in Java.

  Example 1:

  int x = 7;

  float y = x; // y=7.0, assigning int value to float type of variable which is automatic conversion.

  byte z = x; // Type mismatch: cannot convert from int to byte.



Remember Some Rules for Automatic Type Conversion

1) If byte, short, and int are used in a mathematical expression, Java always converts the result into an int.
2) If a single long is used in the expression, the whole expression is converted to long.
3) If a float operand is used in an expression, the whole expression is converted to float.

4) If any operand is double, the result is promoted to double.
5) Boolean values cannot be converted to another type.
6) Conversion from float to int causes truncation of the fractional part which represents the loss of precision. Java does not allow this.
7) Conversion from double to float causes rounding of digits that may cause some of the value's precision to be lost.
8) Conversion from long to int is also not possible. It causes the dropping of the excess higher order bits.

Explicit / Casting / Narrowing Conversion

- As we know that implicit type conversion is helpful and safe but it will not fulfilled all required needs.
- Suppose we will assign a double value to an int variable then conversion cannot performed automatically because int is smaller type than a double type.
- In this case we must do explicit type casting to create conversion between two incompatible types. So this type of conversion called as Narrowing conversion.
- The process of conversion from Higher datatype to Lower datatype is called as Explicit type casting
- Explicit conversion performed by Programmer not by complier.

Syntax for type casting

(Datatype name)expression;

Example:

```
int x;
double y = 9.99;
x = (int) y;   // It will compile in Java and the resulting value will simply be 9.
```


Disadvantage of Explicit Type casting

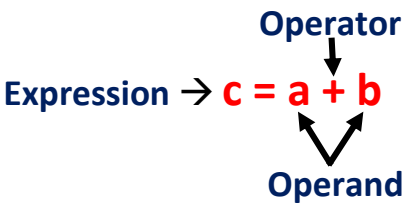
- When you will use type casting in Java, you can lose some information or data.
- Accuracy can be lost while using type casting.
- When a double is cast to an int, the fractional part of a double is discarded which causes the loss of fractional part of data.

Operators and Expressions in Java

What is Operator: A symbol that represent to operation certain operation is called as Operator.

What is Operand: The variables or constants that operators act upon are called **operands**. Operand is also called as Variable.

What is Expression: The alternate combination of operand and operator is called as Expression.



Types of Operator:

There are three types of operators in java based on the number of operands used to perform an operation.

Unary Operator: The operator which required one operand to perform operation is called as unary operator.

Binary Operator: The operator which required two operand to perform operation is called as Binary operator.

Ternary Operator: The operator which required three operand to perform operation is called as Ternary operator.

Classification of Operators based on Symbols and Named in Java

The Java operators can be classified on the basis of symbols and named.

1. Symbolic operator in Java

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bitwise operators
- Bitwise shift operators
- String concatenation Operator
- new operator
- [ ] operator

2. Named operators in Java

- instanceof operator

Arithmetic Operators:

The +, -, *, /, % operators are called as Arithmetic Operator. Arithmetic operators are used to performing fundamental arithmetic operations such as addition, subtraction, multiplication, and division and modulus division (Remainder) on numeric data types.

Example:

```
public class ArithmeticOperations {

	public static void main(String args[]) {

		int a = 20, b = 10;
		 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("a % b = " +(a % b));
	 }
}
```

Case 1: If we perform arithmetic operation on any type of value then we will get the result of higher datatype value.

byte + byte = int
byte + short = int
short + short = int
short + long = long
double + float = double
int + double = double
char + char = int
char + int = int
char + double = double

Example:
```
byte b=13;
char ch = 'a';
short s = 7;
System.out.println(ch + b);    // 110
System.out.println(b + s);   //20
```

Case 2: In integral arithmetic (byte, short, int, long) there is no way to represents infinity, if infinity is the result, we will get the ArithmeticException / by zero.

```
System.out.println(10/0);      // RE: ArithmeticException / by zero
```

But in floating point arithmetic (float , double) there is a way represents infinity.

```
System.out.println(10/0.0); // output : infinity
```

For the Float & Double classes contains the following constants.
1. POSITIVE_INFINITY
2. NEGATIVE_INFINITY

Hence , if infinity is the result we won't get any ArithmeticException in floating point arithmetics
Example :

```
System.out.println(10/0.0);     // Infinity
System.out.println(-10/0.0);   //  Infinity
```

Case 3: NaN (Not a Number) in integral arithmetic (byte, short, int, long) there is no way to represent undefined the results.
Hence the result is undefined we will get ArithmericException in integral arithmetic

```
System.out.println (0/0); // RE: ArithmeticException / by zero
```

But In floating point arithmetic (float, double) there is a way to represents undefined the results.
For the Float, Double classes contains a constant NaN, Hence the result is undefined we won't get ArithmeticException in floating point arithmetics.

```
System.out.println(0.0/0.0);       //  NaN
System.out.println(-0.0/0.0);     //  NaN
```

Relational Operator:

- The **>, <, >=, <=, ==, !=** operators are called as Relational Operators**.**
- Relational operators in Java are those operators that are used to perform the comparison between two numeric values or two variables.
- These operators determine the relationship between them by comparing operands. Therefore, relational operators are also called comparison operators.
- The expression which contain relational operator is called as Conditional Expressions. In java conditional expression gives result in the form boolean values true or false.

Example 1:

```
public class RelationalOperations {

        public static void main(String args[]) {
                int a = 20, b = 10;
                System.out.println(a>b);
                System.out.println(13<7);
                System.out.println('a' == 97);
        }
}
```

Example 2: we cannot apply relational operator on boolean values.

System.out.println(true > false); →Compile time error

Example 3: We cannot apply relational operators for object types.

System.out.println("ShreeSoft" > "ShreeSoft"); → Compile time error

Example 4: Nesting of relational operator is not allowed

System.out.println(7>13>16);

Example 5: We can apply on boolean only equality operator.

System.out.println(true == true);

Example 6: We can apply on object only equality operator.

```
String s1 = new String();
String s2 = new String();
String s3 = s1;

System.out.println(s1 == s2); → false
System.out.println(s1 == s3); → true
```

Logical Operator:

- The **&& (Logical AND), || (Logical OR), ! (Logical NOT)** operator is called as Logical operator.

- Java logical operators combine the more than one comparison into one condition group. It is useful when we want to check more than one condition at a time.

Logical AND (&&):

The AND operator combines two expressions (or conditions) together into one condition group. Both expressions are tested separately by JVM and then && operator compares the result of both.

If the conditions on both sides of && operator are true, the logical && operator returns true. If one or both conditions on either side of the operator are false, then the operator returns false.

Example:

```
int x = 7,y=13;
System.out.println(x<=13 && 'a'==97);   //true
System.out.println(x!=7 && y==13);  // false
```

Logical OR (||):

The OR operator returns true if either one or both of the conditions returns true. If the conditions on both sides of the operator are false, the logical OR operator returns false.

Example:

```
int x = 7,y=13;
System.out.println(x<=3 || 'a'==97);   //true
System.out.println(x!=7 || y!=13);  // false
```

Logical NOT (!):

The NOT operator is used to reverse the logic state of its operand. If the condition is correct, the logical NOT operator returns false. If the condition is false, the operator returns true.

Example:

```
int x = 7;
System.out.println(!(x==7));   //false
System.out.println(!(x!=7));  // true
```

Assignment Operator:

An operator which is used to store a value into a particular variable is called **Assignment Operator in java**.

We can assign a value or value of another variable or result of expression or a value return by a methods.

Example:

```
int x, y=13;
x=7; //assign value i.e. the value of x is 7
x=y; // assign value of y to x i.e. x=13
x=y*9+x; // assigning result of 'y*9+x' expression to x i.e. x=124
x = Math.sqrt(169); // assign value return by sqrt() method i.e. x=13
```

Increment and Decrement Operator:

Increment Operator: The ++ symbol is called as increment operator. This operator increase value by 1 of given operand. We can use two different way increment operator.

Post Increment	Pre Increment
1) If we placed increment operator after operand is called as post increment operator. 2) In this first assign then increment. Ex - int m=1,n; n=m++; output: m=2, n=1	1) If we placed increment operator before operand is called as pre increment operator. 2) In this first increment then assign. Ex - int m=1,n; n=++m; output: m=2, n=2

Decrement Operator: The -- symbol is called as decrement operator. This operator decrease value by 1 of given operand. We can use two different way decrement operator.

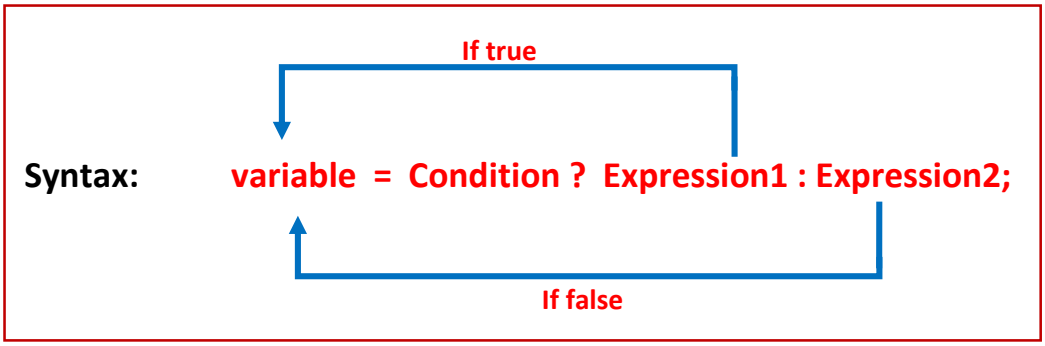
Post Decrement	Pre Decrement
1) If we placed decrement operator after operand is called as post decrement operator. 2) In this first assign then decrement. Ex - int m=1,n; n=m--; output: m=0, n=1	1) If we placed decrement operator before operand is called as pre decrement operator. 3) In this first decrement then assign. Ex - int m=1,n; n=--m; output: m=0, n=0

Conditional Operators:

Conditional operator in Java provides a one line approach for creating a simple conditional statement.

It is often used as a shorthand method for if-else statement. It makes the code much simpler, shorter and readable.

The conditional operator (? :) is also known as ternary operator in java because it takes three operands and perform a conditional test.



Example:

```
int x=8;
String result = x%2==0 ? "Even" : "Odd";
System.out.println(result);
```

Bitwise Operator:

- An operator that performed operations on individual bits (0 or 1) of the variable is called **Bitwise operator in java**.
- It acts only integer data types such as byte, short, int, and long. Bitwise operators in java cannot be applied to float and double data types.
- The internal representation of numbers in the case of bitwise operators is represented by the binary number system. Binary number is represented by two digits 0 or 1.
- Therefore, these operators are mainly used to modify bit patterns (binary representation).

Operator	Meaning
&	bitwise AND (Binary)
|	bitwise OR (Binary)
^	bitwise exclusive OR (Binary)
~	bitwise NOT (Unary)
<<	shift left
>>	shift right
>>>	unsigned right shift

Bitwise AND (&) Operator:

It is a binary operator denoted by the symbol **&**. It returns 1 if and only if both bits are 1, else returns 0.

Bitwise OR (|) Operator:

It is a binary operator denoted by the symbol **|**. It returns 1 if at least one bits are 1, else returns 0.

Bitwise XOR (^) Operator:

It is a binary operator denoted by the symbol **^**. It returns 0 if and only if both bits are 1, else returns 1.

Examples:

```
int x = 13, y = 7;

     x = 0000 0000 0000 0000 0000 0000 0000 1101
     y = 0000 0000 0000 0000 0000 0000 0000 0111
-----------------------------------------------------------------
x & y = 0000 0000 0000 0000 0000 0000 0000 0101 = 5

     x = 0000 0000 0000 0000 0000 0000 0000 1101
     y = 0000 0000 0000 0000 0000 0000 0000 0111
-----------------------------------------------------------------
x | y =  0000 0000 0000 0000 0000 0000 0000 1111 = 15

     x = 0000 0000 0000 0000 0000 0000 0000 1101
     y = 0000 0000 0000 0000 0000 0000 0000 0111
-----------------------------------------------------------------
x ^ y =  0000 0000 0000 0000 0000 0000 0000 1010 = 10
```

Bitwise left shift (<<) Operator:

The operator that shifts the bits of number towards left by n number of bit positions is called **left shift operator in Java**.

Example:

```
int x = 7;
x = x << 2;
x = 0000 0000 0000 0111
```

Remove 2 Zero's

x = 0000 0000 0001 1100 = **28**

Add 2 Zero's

Bitwise right shift Operator (>>)

Example:

```
int x = 7;
x = x >> 2;
x = 0000 0000 0000 0111
```

Remove 2 Zero's

x = 0000 0000 0000 0001 = **1**

Add 2 Zero's

Unsigned Right Shift Operator

The unsigned right shift operator in java performs nearly the same operation as the signed right shift operator in java. The unsigned right shift operator is represented by a symbol >>>, read as triple greater than.

The unsigned right shift operator always fills the leftmost position with 0s because the value is not signed. Since it always stores 0 in the sign bit, it is also called zero fill right shift operator in java.

If you will apply the unsigned right shift operator >>> on a positive number, it will give the same result as that of >>. But in the case of negative numbers, the result will be positive because the signed bit is replaced by 0.

Example:

```
int x = 10;
int y = -10;
System.out.println("x >> 1 = " + (x >> 1));
System.out.println("x >>> 1 = " + (x >>> 1));
System.out.println("y >> 2 = " + (y >> 2));
System.out.println("y >>> 2 = " + (y >>> 2));
```

Bitwise NOT (~) Operator:

It is a unary operator denoted by the symbol **~** (pronounced as the tilde). It returns the inverse or complement of the bit. It makes every 0 a 1 and every 1 a 0.

Example:

```
int x= -13, y=7;
System.out.println(~x);
System.out.println(~y);
```

String concatenation Operator

Concatenation is the process of combining two or more strings to form a new string by subsequently appending the next string to the end of the previous strings. In java we can concatenate two string using '+' or '+=' and concate() method which is defined in the java.lang.String class.

Example:

```
String str1 = "Hello";
String str2 = "World";
String str3 = str1 + str2;
System.out.println(str3);
```

new operator:

1. The new operator is used in Java to create new objects. It can also be used to create an array object.
2. There is no "delete" operator in java because destruction of useless objects is the responsibility of garbage collector

[ ] operator:

This operator will be used to create array.

instanceof operator:

We can use the instanceof operator to check whether the given an object is particular type or not.

Example:

```
class Simple {
  public static void main(String args[]){
  String str1 = "ShreeSoft-Informatics";

  if (str1 instanceof String)
    System.out.println("str1 is instance of String");
  else
    System.out.println("str1 is NOT instance of String");
 }
}
```