UNIT I

Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Classes
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Objects And Classes

Object Objects have states and behaviours. Example: A dog has states - colour, name, breed as well as behaviours – wagging the tail, barking, eating. An object is an instance of a class.

Class

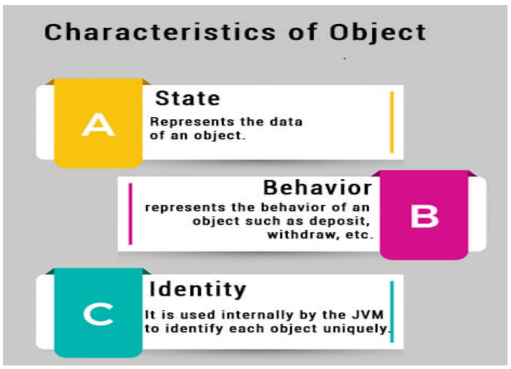
A class can be defined as a template/blueprint that describes the behaviour/state that the object of its type support.

Objects in Java

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.


Characteristics of Object
A
State
Represents the data of an object.
Behavior
represents the behavior of an object such as deposit, withdraw, etc.
B
C
Identity
It is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

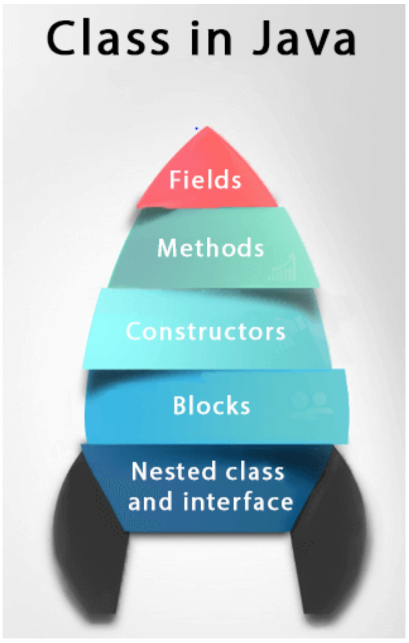
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

Classes in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**



Following is an example of a class.

```
public class Dog
{
String breed;
int age;
String color;
void barking()
{ }
 }
```

A class can contain any of the following variable types.

• **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

• **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

• **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword. A class can have any number of methods to access the value of various kinds of methods. In the above example, barking (), hungry () and sleeping () are methods.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

- In Java, the basis of encapsulation is the class. There are mechanisms for hiding the complexity of the implementation inside the class.
- Each method or variable in a class may be marked private or public.
- The public interface of a class represents everything that external users of the class need to know, or may know.
- The private methods and data can only be accessed by code that is a member of the class.
- Therefore, any other code that is not a member of the class cannot access a private method or variable.
- Since the private members of a class may only be accessed by other parts of program through the class' public methods, we can ensure that no improper actions take place.

Inheritance

Inheritance is the process by which one object acquires the properties of another object. Inheritance is an important pillar of OOP (Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class.

In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.

Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

For eg., a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose.

Consider a stack (which is a last-in, first-out LIFO list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.

Platform independent: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

Features of JAVA

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as Java buzzwords.

A list of the most important features of the Java language is given below.

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:

Java syntax is based on C++ (so easier for programmers to learn it after C++).

Object-Oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

Secure

Java program cannot harm other system thus making in secure. Java provides a secure means of creating Internet application.

With Java's secure feature, it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Robust

Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Interpreted

Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and lightweight process.

High Performance

With the use of Just-In-Time compilers, Java enables high performance. Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.

It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Multithreaded

With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

Distributed:

Java is designed for the distributed environment of the internet. Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications.

This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

Basic of Java Programming using Class

```
class Student{
//defining fields
int id; //field or data member or instance variable
String name;
//creating main method inside the Student class
public static void main(String args[]){
//Creating an object or instance
Student s1=new Student(); //creating an object of Student
//Printing values of the object
System.out.println(s1.id); //accessing member through reference variable
System.out.println(s1.name);
}
}
```

Constructor

Every class has a constructor. If the constructor is not defined in the class, the Java compiler builds a default constructor for that class. While a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behaviour of the class and its objects.

Rules for writing Constructor

• Constructor(s) of a class must have same name as the class name in which it resides.
• A constructor in Java cannot be abstract, final, static and synchronized.
• Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor

Following is an example of a constructor − Example

```
 public class myclass
{
public myclass()
 {
// Constructor
}
 public myclass(String name)
 { // This constructor has one parameter, name.
}
}
```

Types of Constructors

There are two type of constructor in Java:

1. **No-argument constructor**: A constructor that has no parameter is known as default constructor. If the constructor is not defined in a class, then compiler creates default constructor (with no arguments) for the class.

   If we write a constructor with arguments or no-argument then compiler does not create default constructor. Default constructor provides the default values to the object like 0, null etc. depending on the type.

```
// Java Program to illustrate calling a no-argument constructor
 import java.io.*;
class myclass
 {
int num;
String name;
 // this would be invoked while object of that class created.
myclass()
{
System.out.println("Constructor called");
 }
}
class myclassmain
{
public static void main (String[] args)
  {        // this would invoke default constructor.
 myclass m1 = new myclass(); // Default constructor provides the default values to the
object like 0, null
System.out.println(m1.num);
System.out.println(m1.name);
}
}
```

2. **Parameterized Constructor**

   A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use parameterized constructor.

```
// Java Program to illustrate calling of parameterized constructor.
 import java.io.*;
class myclass
{     // data members of the class.
String name;
int num;
// contructor with arguments.
myclass(String name, int n)
{
this.name = name;
this.num = n;
}
}
class myclassmain
{
public static void main (String[] args)
{        // this would invoke parameterized constructor.
myclass m1 = new myclass("Java", 2017);
System.out.println("Name :" + m1.name + " num :" + m1.num);
}
}
```

There are no "return value" statements in constructor, but constructor returns current class instance. We can write 'return' inside a constructor.

Methods in Java

A method is a collection of statement that performs specific task. In Java, each method is a part of a class and they define the behaviour of that class. In Java, method is a jargon used for method.

Advantages of methods

- Program development and debugging are easier
- Increases code sharing and code reusability
- Increases program readability
- It makes program modular and easy to understanding
- It shortens the program length by reducing code redundancy

Types of methods

There are two types of methods in Java programming:

- Standard library methods (built-in methods or predefined methods)
- User defined methods

Packages	**Library Methods**	**Descriptions**

java.lang.Math
All maths related methods are defined in this class
	acos()
	exp()
	abs()
	log()
	sqrt()
	pow()
		Computes arc cosine of the argument
		Computes the e raised to given power
		Computes absolute value of argument
		Computes natural logarithm Computes square root of the argument
		Computes the number raised to given power

java.lang.String
All string related methods are defined in this class
	charAt()
	concat()
	compareTo()
	indexOf()
	
	toUpperCase()
		Returns the char value at the specified index. Concatenates two string
		Compares two string
		Returns the index of the first occurrence of the given character
		converts all of the characters in the String to upper case

java.awt
contains classes for graphics
	add()
	setSize()
	setLayout()
	setVisible()
		inserts a component
		set the size of the component
		defines the layout manager
		changes the visibility of the component

Example:

Program to compute square root of a given number using built-in method.

```
public class MathEx

{

public static void main(String[] args)
{
        System.out.print("Square root of 14 is: " + Math.sqrt(14));
}
}
```

Sample Output: Square root of 14 is: 3.7416573867739413

User-defined methods

The methods created by user are called user defined methods.

Every method has the following.

- Method declaration (also called as method signature or method prototype)
- Method definition (body of the method)
- Method call (invoke/activate the method)

Method Declaration

The syntax of method declaration is:

Syntax:

```
return_type method_name(parameter_list);
```

Here, the return_type specifies the data type of the value returned by method. It will be void if the method returns nothing. method_name indicates the unique name assigned to the method. parameter_list specifies the list of values accepted by the method.

Method Definition

Method definition provides the actual body of the method. The instructions to complete a specific task are written in method definition. The syntax of method is as follows:

Syntax:

```
modifier return_type method_name(parameter_list){
        // body of the method
      }
```

Here,

Modifier – Defines the access type of the method i.e accessibility region of method in the application

return_type – Data type of the value returned by the method or void if method returns nothing

method_name – Unique name to identify the method. The name must follow the rules of identifier

parameter_list – List of input parameters separated by comma. It must be like

datatype parameter1,datatype parameter2,……

List will be empty () in case of no input parameters.

method body – block of code enclosed within { and } braces to perform specific task

Types of User-defined methods

The methods in C are classified based on data flow between calling method and called method. They are:

- Method with no arguments and no return value
- Method with no arguments and a return value
- Method with arguments and no return value
- Method with arguments and a return value.

Access Specifiers

Access specifiers or access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class. It determines whether a data or method in a class can be used or invoked by another class or subclass.

Types of Access Specifiers

There are 4 types of java access specifiers:

1. Private
2. Default (no speciifer)
3. Protected
4. Public

Access Modifiers	Default	Private	Protected	Public
Accessible inside the class	Yes	Yes	Yes	Yes
Accessible within the subclass inside the same package	Yes	No	Yes	Yes
Accessible outside the package	No	No	No	Yes
Accessible within the subclass outside the package	No	No	Yes	Yes

Private access modifier

Private data fields and methods are accessible only inside the class where it is declared i.e accessible only by same class members. It provides a low level of accessibility. Encapsulation and data hiding can be achieved using private specifiers

Example:

Role of private specifier

```
class PrivateEx{
 private int x; // private data
 public int y; // public data
 private PrivateEx(){} // private constructor
 public PrivateEx(int a,int b){ // public constructor
 x=a;
 y=b;
 }
}
public class Main {
 public static void main(String[] args) {
```

```
                    PrivateEx obj1=new PrivateEx(); // Error: private constructor cannot be applied
                  PrivateEx obj2=new PrivateEx(10,20); // public constructor can be applied to obj2
                       System.out.println(obj2.y); // public data y is accessible by a non-member
      System.out.println(obj2.x); //Error: x has private access in PrivateEx
      }
      }
```

Default access modifier

If the specifier is mentioned, then it is treated as default. There is no default specifier keyword. Using the default specifier we can access class, method, or field which belongs to the same package, but not from outside this package.

Example:

```
   Role of default specifier
   class DefaultEx{
           int y=10;// default data
   }
   public class Main {
    public static void main(String[] args) {
    DefaultEx obj=new DefaultEx();
    System.out.println(obj.y); // default data y is accessible outside the class
    }
   }
```

Sample Output:

```
    10
```

In the above example, the scope of class DefaultEx and its data *y* is default. So it can be accessible within the same package and cannot be accessed from outside the package.

Protected access modifier

Protected methods and fields are accessible within same class, subclass inside same pack age and subclass in other package (through inheritance). It cannot be applicable to class and interfaces.

Example:

```
   Role of protected specifier
   class Base{
    protected void show(){
        System.out.println("In Base");
        }
       }
      public class Main extends Base{
        public static void main(String[] args) {
        Main obj=new Main();
         Obj.show();
         }
}
```

Sample Output:

```
    In Base
```

In this example, *show()* of class Base is declared as protected, so it can be accessed from outside the class only through Inheritance.

Public access modifier

The public access specifier has highest level of accessibility. Methods, class, and fields declared as public are accessible by any class in the same package or in other package.

Example:

Role of public specifier

```
class PublicEx{
public int no=10;
}
public class Main{
public static void main(String[] args) {
PublicEx obj=new PublicEx();
System.out.println(obj.no);
}
}
```

Sample Output:

```
    10
```

In this example, public data no is accessible both by member and non-member of the class.

Static Member

In Java, static members are those which belongs to the class and you can access these members without instantiating the class.

The static keyword can be used with methods, fields, classes (inner/nested), blocks. The static keyword can be used with:

- Variable (static variable or class variable)
- Method (static method or class method)
- Block (static block)
- Nested class (static class)
- import (static import)

Static variable

Variable declared with keyword static is a static variable. It is a class level variable commonly shared by all objects of the class.

- Memory allocation for such variables only happens once when the class is loaded in the memory. scope of the static variable is class scope (accessible only inside the class)
- lifetime is global (memory is assigned till the class is removed by JVM).
- Automatically initialized to 0.
- It is accessible using ClassName.variablename
- Static variables can be accessed directly in static and non-static methods.

```
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Sample Output:

```
111 Karan ITS
222 Aryan ITS
```

Static Methods

You can create a static method by using the keyword *static*. Static methods can access only static fields, methods. To access static methods there is no need to instantiate the class, you can do it just using the class name as

```
public class MyClass {
   public static void sample(){
    System.out.println("Hello");
  }
  public static void main(String args[]){
    MyClass.sample();
  }
}
```

Sample Output

```
          Hello
```

Static Blocks

These are a block of codes with a static keyword. In general, these are used to initialize the static members. JVM executes static blocks before the main method at the time of class loading.

```
public class MyClass {
  static{
    System.out.println("Hello this is a static block");
  }
  public static void main(String args[]){
    System.out.println("This is main method");
  }
}
```

Nested class (static class)

Java inner class or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Syntax

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Data Types

Java is a statically typed and also a strongly typed language. In Java, each type of data (such as integer, character, hexadecimal, etc.) is predefined as part of the programming language and all constants or variables defined within a given program must be described with one of the data types.

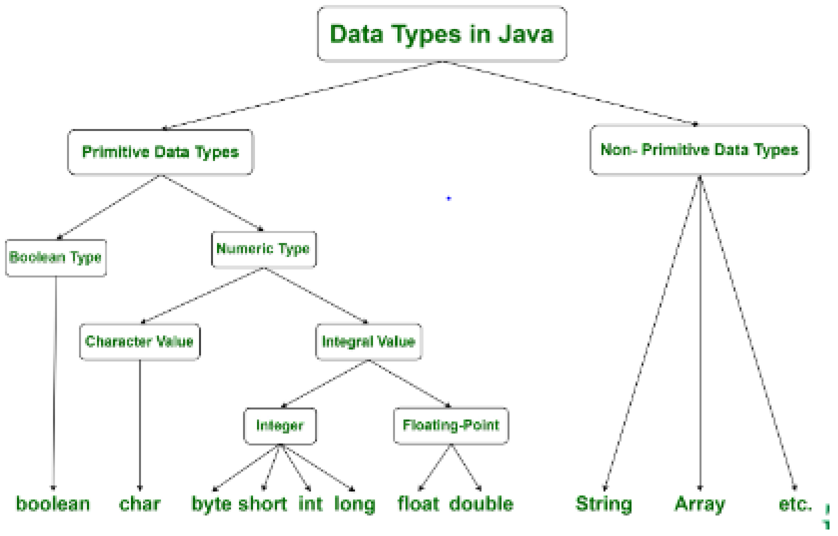
Data types represent the different values to be stored in the variable. In java, there are two categories of data types:

- Primitive data types
- Non-primitive data types

There are 8 types of primitive data types:

- boolean data type

- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type



Data Type	Default Value	Default Size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.

The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example

Boolean one = **false**

Byte Data Type

The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.

The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example

byte a = 10, **byte** b = -20

Short Data Type

The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.

The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example

short s = 10000, **short** r = -5000

Int Data Type

The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.

The int data type is generally used as a default data type for integral values unless if there is no problem about memory.

Example

int a = 100000, **int** b = -200000

Long Data Type

The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and maximum value is 9,223,372,036,854,775,807.

Its default value is 0. The long data type is used when you need a range of values more than those provided by int.

Example

long a = 100000L, **long** b = -200000L

Float Data Type

The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers.

The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example

float f1 = 234.5f

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example

double d1 = 12.3

Char Data Type

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters.

Example:

char letterA = 'A'

Variables

A variable is the holder that can hold the value while the java program is executed. A variable is assigned with a datatype. It is the name of a reserved area allocated in memory. In other words, it is the name of a memory location. There are three types of variables in java: local, instance and static.

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Before using any variable, it must be declared. The following statement expresses the basic form of a variable declaration

Syntax

datatype variable [= value][, variable [= value] ...] ;

Example

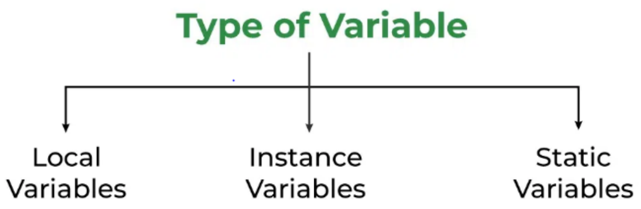
int a, b, c; // Declaration of variables a, b, and c.

int a = 20, b = 30; // initialization

byte B = 22; // Declaration initializes a byte type variable B.

Types of Variable There are three types of variables in java:

- local variable
- instance variable
- static variable



Local variable

- Local variables are declared inside the methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered
- Local variable will be destroyed once it exits the method, constructor, or block.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally
- There is no default value for local variables, so local variables should be declared and

  an initial value should be assigned before the first use.
- Access specifiers cannot be used for local variables.

Instance Variable

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as static.

It is called an instance variable because its value is instance-specific and is not shared among instances.

Static variable

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

public class A

```
{
    static int m=100;//static variable
    void method()
    {
        int n=90;//local variable
    }
    public static void main(String args[])
    {
        int data=50;//instance variable
    }
}
```

Operators

Operator in java is a symbol that is used to perform operations. Java provides a rich set of operators to manipulate variables.For example: +, -, *, / etc.

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators
- Unary Operator
- Ternary Operator

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the assignment operator (=) to assign the value 10 to a variable called x:

Example:

```
int X=10;
```

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Java Comparison Operators

Comparison operators are used to compare two values (or variables). This is important in programming, because it helps us to find answers and make decisions.

The return value of a comparison is either true or false. These values are known as *Boolean values*, and you will learn more about them in the Booleans and If..Else chapter.

In the following example, we use the greater than operator (>) to find out if 5 is greater than 3:

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Java Logical Operators

You can also test for true or false values with logical operators.

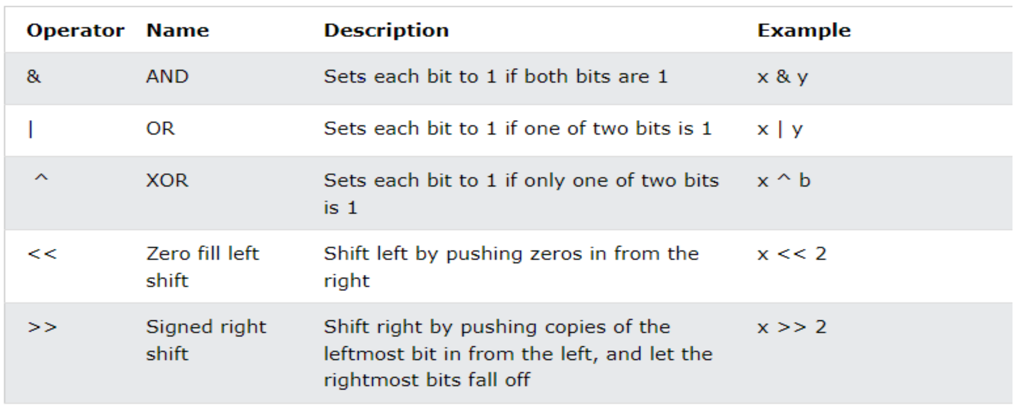
Logical operators are used to determine the logic between variables or values:

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Bitwise Operators

Bitwise operators are used on (binary) numbers:

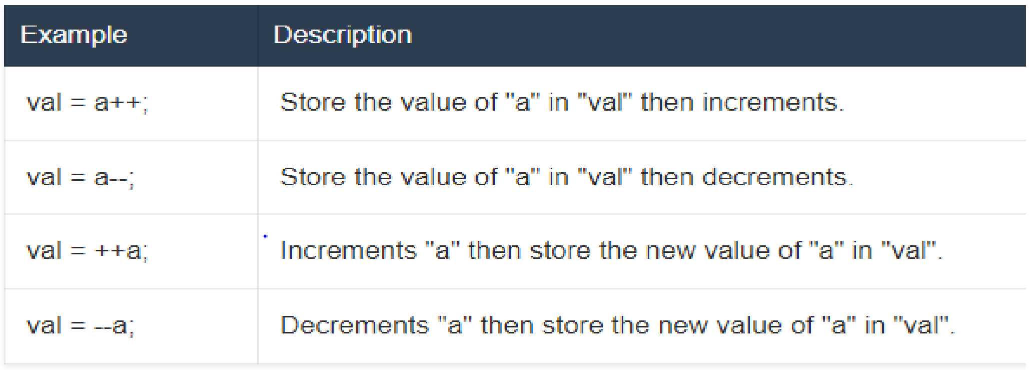
In Java, an **operator** is a symbol that performs the specified operations. In this section, we will discuss only the **bitwise operator and its types** with proper examples.



Operator	Name	Description	Example
&	AND	Sets each bit to 1 if both bits are 1	x & y
|	OR	Sets each bit to 1 if one of two bits is 1	x | y
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ b
<<	Zero fill left shift	Shift left by pushing zeros in from the right	x << 2
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2

Unary Operator

In Java, unary arithmetic operators are used to increasing or decreasing the value of an operand. Increment operator adds 1 to the value of a variable, whereas the decrement operator decreases a value.



Example	Description
val = a++;	Store the value of "a" in "val" then increments.
val = a--;	Store the value of "a" in "val" then decrements.
val = ++a;	Increments "a" then store the new value of "a" in "val".
val = --a;	Decrements "a" then store the new value of "a" in "val".

Ternary Operators

There is also a short-hand if else, which is known as the ternary operator because it consists of three operands.

It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:

Syntax

variable = (condition) ? expressionTrue : expressionFalse;

Example

```
int time = 20;

String result = (time < 18) ? "Good day." : "Good evening.";

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