# MODULE 1
# OOPS CONCEPTS AND JAVA PROGRAMMING

## The Java Language:

• Java was initiated by **James Gosling** and others at **Sun Microsystems** in 1991.
• Team: Mike Sheridan **and** Patrick Naughton
• This language was initially called **"OAK".**
• Later it was renamed as **"JAVA"** in 1995.
• It is one of the world's most widely used computer language.
• From practical point of view, it is an excellent language to learn.
• Java was perfect for the Web.

## C++ vs Java

| C++ | Java |
|---|---|
| **C++ is platform-dependent.** | Java is platform-independent. |
| **C++ supports goto statement.** | Java doesn't support goto statement. |
| **C++ supports multiple inheritance.** | Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java. |
| **C++ supports operator overloading.** | Java doesn't support operator overloading. |
| **C++ supports pointers. You can write pointer program in C++.** | java has restricted pointer support in java. |
| **C++ supports both call by value and call by reference.** | Java supports call by value only. There is no call by reference in java. |
| **C++ doesn't support >>> operator.** | Java supports unsigned right shift >>> operator |

## Features of Java

The Java Features are:

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- 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++).

• Java has removed many complicated and rarelyused features, for example, explicit pointers, operator overloading, etc.

• There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

**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 behavior.

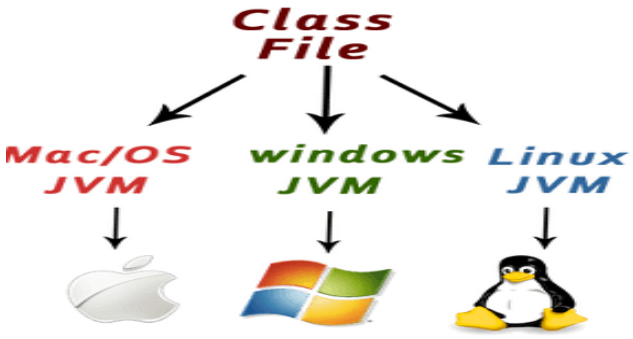
**Portable**

• Java is portable because it facilitates you to carry the Java bytecode to any platform.

• It doesn't require any implementation.

**Platform Independent:**

• Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.




**Secured**

• Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

• No explicit pointer

• Java Programs run inside a virtual machine sandbox

**Robust**

• 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.

**Compiled and Interpreted:**
• Java code is compiled to bytecode.
• Bytecode are interpreted on any platform by JVM.

**High Performance**
• Bytecode is highly optimised.
• JVM execute Bytecode much faster

**Multithreaded**
• Multithreading means handling more than one job at a time.
• The main advantage of multi-threading is that it shares the same memory.

**Distributed**
• Java programs can be shared over the internet

## The Key attributes of OOP's

**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:

**The key attributes of OOP's are:**
1. Inheritance
2. Polymorphism
3. Encapsulation

- **Inheritance:**
  - When one object acquires all the properties and behaviors of parent object i.e. known as inheritance.
- **Polymorphism:**
  - we can perform a *single action in different ways*.
  - Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
  - There are two types of polymorphism in Java**: compile-time polymorphism** and **runtime polymorphism**.
  - We can perform polymorphism in java by method overloading and method overriding.
- **Encapsulation:**
  - Binding (or wrapping) code and data together into a single unit is known as encapsulation.

### Advantages of Inheritance

**Minimizing duplicate code:** Key benefits of Inheritance include minimizing the identical code as it allows sharing of the common code among other subclasses.

**Flexibility:** Inheritance makes the code flexible to change, as you will adjust only in one place, and the rest of the code will work smoothly.

**Overriding:** With the help of Inheritance, you can override the methods of the base class.

**Data Hiding:** The base class in Inheritance decides which data to be kept private, such that the derived class will not be able to alter it.

**Data Abstraction**
Data abstraction is the process of hiding certain details and showing only essential information to the user.

## JVM

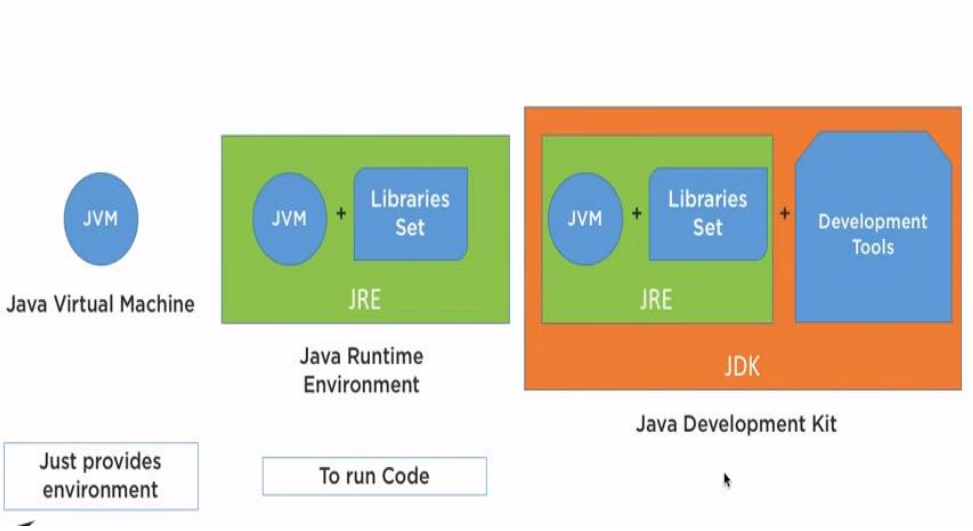
• JVM (Java Virtual Machine) is an abstract machine
• It is called a virtual machine because it doesn't physically exist.
• It is a specification that provides a runtime environment in which Java bytecode can be executed.
• It can also run those programs which are written in other languages and compiled to Java bytecode.
• JVMs are available for many hardware and software platforms.
• JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent.

## JRE

• JRE is an acronym for Java Runtime Environment.
• It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications.
• It is used to provide the runtime environment. It is the implementation of JVM.
• It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

## JDK

JDK (Java Development Kit) is a software development kit required to develop applications in Java.
• you download JDK, JRE is also downloaded with it.
• In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc).
How Java Code Runs?



## Introducing Classes:

• 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**

• An object is an instance of a class.

**The general form of a class:**
A class is created by using the keyword **class.**
The general form of a class is,

```
class classname{
        type instance-variable1;
        type instance-variable2;
        …….
        type instance-variableN;

        type method1(parameters){
                //body of the method
        }
        type method2(parameters){
                //body of the method
        }
        //…..
        type methodN(parameters){
                //body of the method
        }
}
```

## Objects

• A Java object is a member (also called an instance) of a Java class.
• Each object has an identity, a behavior and a state. The state of an object is stored in fields (variables), while methods (functions) display the object's behavior.

### Creating an Object

There are three steps when creating an object from a class
- **Declaration** - A variable declaration with a variable name with an object type.
- **Instantiation** - The '**new**' **keyword** is used to create the object.
- **Initialization** - The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

• To create a simple java program, you need to create a **class** that contains **main method**.
Ex1:

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

- save this file as **Simple.java**

- To compile:**javac Simple.java**
- To execute: **java Simple**

• **class** keyword is used to declare a class in java.
• **public** keyword is an access modifier which represents visibility, it means it is visible to all.
• **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. So it saves memory.
• **void** is the return type of the method, it means it doesn't return any value.
• **main** represents startup of the program.
• **String[] args** is used for command line argument.
• **System.out.println()** is used print statement.

**Example:**

```
class Vehicle {
        int passengers; // number of passengers
        int fuelCap; // fuel capacity in litres
        int mpg; // fuel consumption in miles per litres
}
```

• A class definition creates a new data type. In this case, the **new data type** is called **Vehicle.**
• You will use this name to declare objects of type **Vehicle.**
• To create a **Vehicle** object, you will use a statement such as the following:

```
Vehicle minivan= new Vehicle();
```

• After this statement executes, **minivan** will be an instance of **Vehicle.**
• The **dot operator(.)** links the name of an object with the name of a member.
• The general form of the **dot operator** is

**object.member**

**Ex:**

**minivan.fuelcap=16;**

• In general the dot(.)operator is used to access **both instance variables and methods.**

```
class Vehicle {
        int passengers;
         int fuelCap;
         int mpg;
}
class VehicleDemo {
        public static void main(String[] args) {
                Vehicle minivan = new Vehicle();
                int range;
                minivan.passengers = 7;
                minivan.fuelCap = 16;
                minivan.mpg = 21;
                range = minivan.fuelCap * minivan.mpg;
                System.out.println("Minivan can carry " + minivan.passengers + " with
                a range of " + range);
        }
}
```

## Java Comments

The java comments are statements that are not executed by the **compiler and interpreter.**

**Types of Java Comments**

There are 3 types of comments in java.

- Single Line Comment
- Multi Line Comment
- Documentation Comment
-

**Java Single Line Comment:**

The single line comment is used to comment only one line.

**Syntax:**

```
//This is single line comment
```

**Example:**

```
class CommentExample1 {
    public static void main(String[] args) {
        int i=10; //Here, i is a variable
        System.out.println(i);
    }
}
```

**Java Multi Line Comment:**

The multi line comment is used to comment multiple lines of code.

**Syntax:**

```
/*
    This is
    multi line
    comment
*/
```

**Example:**

```
class CommentExample2 {
    public static void main(String[] args) {
        /* Let's declare and
        print variable in java. */
        int i=10;
        System.out.println(i);
    }
}
```

**Java Documentation Comment:**

• The documentation comment is used to create documentation API.

• To create documentation API, you need to use **javadoc tool**.

**Syntax:**

```
/**
    This is
    documentation
    comment
*/
```

## Data Types in Java

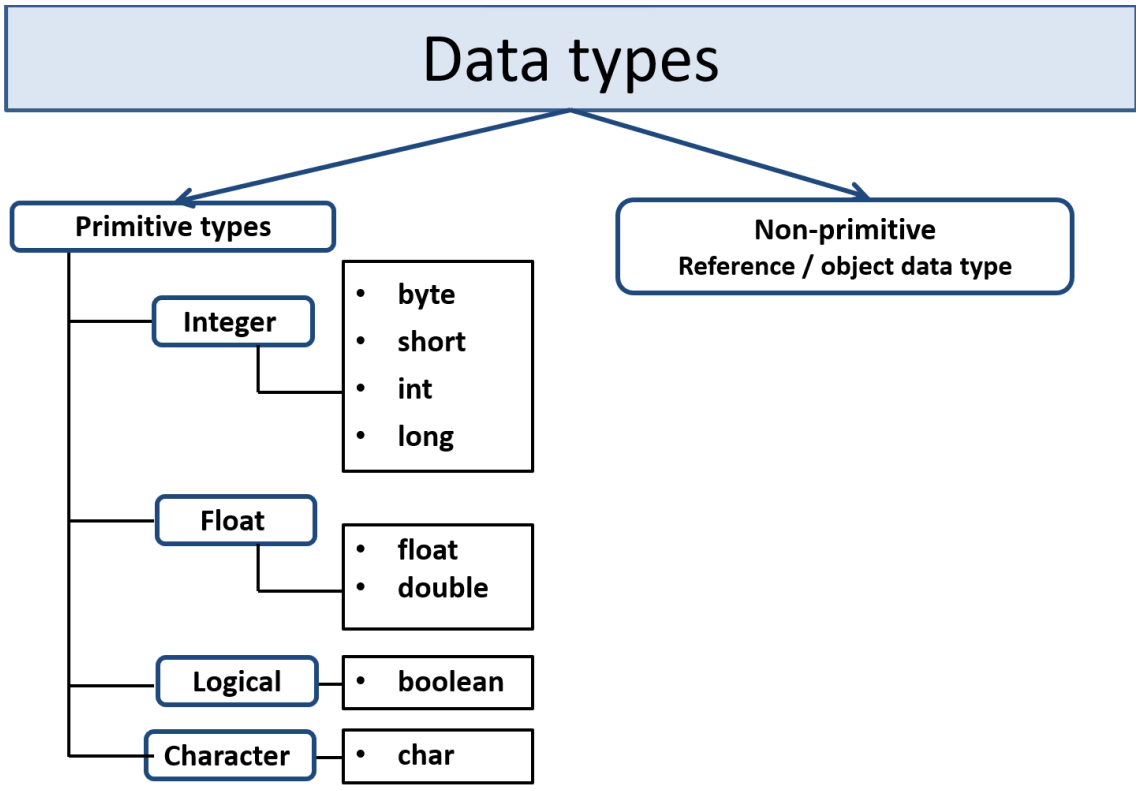
• Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

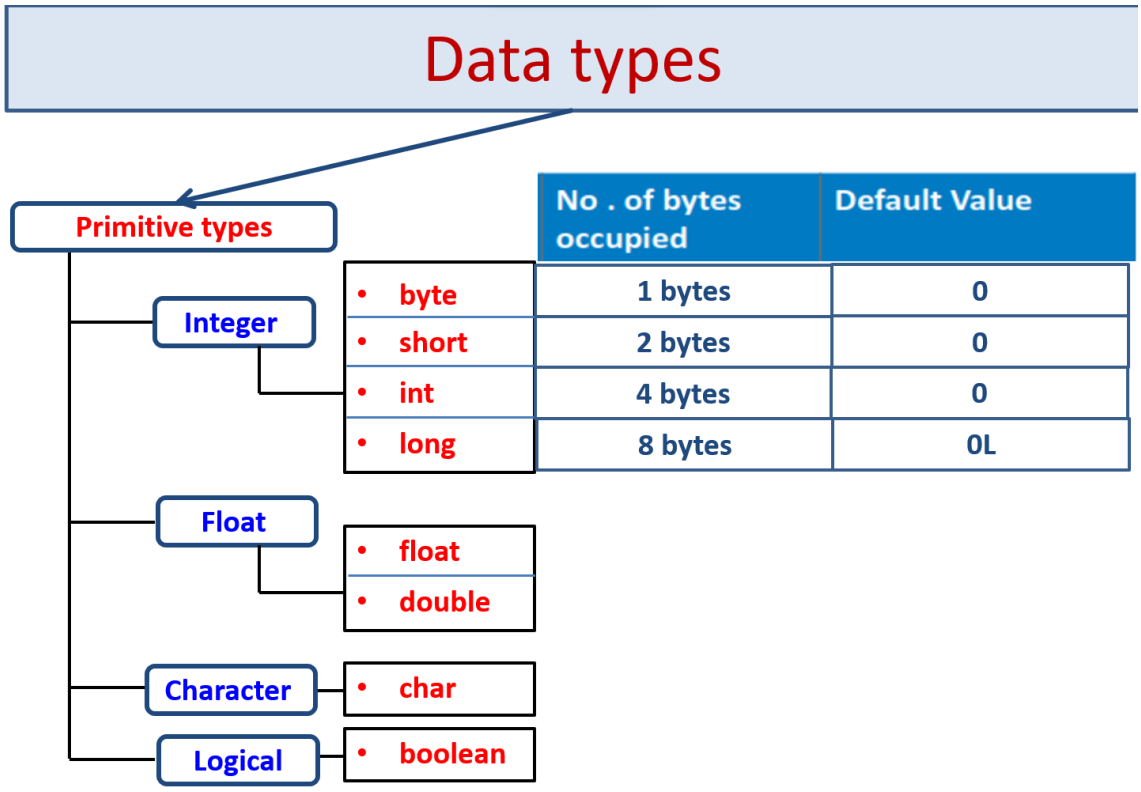
• **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.

• **Non-primitive data types:** The non-primitivedata types include Classes, Interfaces, and Arrays.

### Java's built in primitive data types

| Type | Meaning |
|---|---|
| **boolean** | Represents true/false values |
| **byte** | 8-bit integer |
| **char** | Character |
| **short** | Short Integer |
| **int** | Integer |
| **long** | Long Integer |
| **float** | Single-precision floating point |
| **double** | Double-precision floating point |






## JAVA KEYWORDS

• In the **Java** programming language, a **keyword** is any one of 55 reserved words that have a predefined meaning in the language; because of this, programmers cannot use **keywords** as names for variables, methods, classes, or as any other identifier.

### The Java Keywords

List of Java Keywords

| | | | | |
|---|---|---|---|---|
| abstract | default | if | package | synchronized |
| assert | do | implements | private | this |
| boolean | double | import | protected | throw |
| break | else | instanceof | public | throws |
| byte | extends | int | return | transient |
| case | false | interface | short | true |
| catch | final | long | static | try |
| char | finally | native | strictfp | void |
| class | float | new | super | volatile |
| const | for | null | switch | while |
| continue | goto | | | |

***Although const and goto are reserved words, they are not currently part of the Java language.***

## Java Variables

• A variable is a container which holds the value while the java program is executed.
• A variable is assigned with a datatype.
• There are three types of variables in java: **local, instance and static.**

### Local Variable:

• A variable declared inside the body of the method is called **local variable.**
• You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
• A local variable cannot be defined with "**static**" **keyword**.

### Instance Variable

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

### Static variable

• A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class.

```
class A {
        int data=50;//instance variable
        static int m=100;//static variable
        void method() {
                int n=90;//local variable }
        }//end of class

class Student8 {
        int rollno; String name; static String college =“RNSIT";
        Student8(int r,String n) {
                rollno = r; name = n;
        }
        void display () {
                System.out.println(rollno+" "+name+" "+college);
        }
        public static void main(String args[]) {
                Student8 s1 = new Student8(100,"Kiran");
                Student8 s2 = new Student8(222,"Arya");
                s1.display();
                s2.display();
        }
}
```

• Variables are declared using this form of statement.

**type var-name;**

• Here **type** is the date type of the variable and **varname** is its name.
• The type of the variable cannot be change during its lifetime.

**Ex:**

an **int** variable cannot turn into a **char** variable

• All the variables in java must be declared prior to their use.

**Initializing a variable:**

- follow the variable's name with an equal sign and the value being assigned.
- The general form of initialization is shown here

**type var=value;**

**Ex:**

```
int count=10;
char ch ='a';
float f=1.2;
```

• When declaring two or more variables of the same type using a comma-separated list.

**Ex:** int a,b=10,c,d=15;

### The scope and lifetime of variables

- Java allows variables to be declared within any block. A block is begin with an opening brace and ended by a closed brace.
- A block defines a **scope.**
- Each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of your program.
- It also determines the lifetime of those objects.

```
class Demo {
        public static void main ( String [ ] args )
        {
                int x =10; if (true)
                {
                        int y = 20;
                        System.out.println ( y ); // OK
                }
                System.out.println ( y ); // ERROR, not in scope
                System.out.println ( x ); // OK
        }
}
```

## Operators

- Operators in Java are the symbols used for performing specific operations in Java. Operators make tasks like addition, multiplication, etc which look easy although the implementation of these tasks is quite complex.

### Types of Operators in Java

• There are multiple types of operators in Java all are mentioned below:
• There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

### 1. Arithmetic Operators

- Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.
- The **operands** of the arithmetic operators must be of a **numeric type**.

- You cannot use them on **boolean types, but you can use them on char types, since the char type in Java is** a subset of **int.**
- The following table lists the arithmetic operators:

| Operator | Result |
|---|---|
| + | Addition |
| - | Substraction(Also unary minus) |
| * | Multiplication |
| / | Division |
| % | Modulus |

**The Basic Arithmetic Operators**

- The basic arithmetic operations—**addition, subtraction, multiplication, and division**— all behave as you would expect for all numeric types.

```
class BasicMath {
        public static void main(String args[]) {
                // arithmetic using integers
                System.out.println("Integer Arithmetic");
                int a = 1 + 1;
                int b = a * 3;
                int c = b / 4;
                int d = c - a;
                int e = -d;
                System.out.println("a = " + a); // 2
                System.out.println("b = " + b); // 6
                System.out.println("c = " + c); // 1
                System.out.println("d = " + d); // -1
                System.out.println("e = " + e); // 1
        }
}
```

**The Modulus Operator:**

- The modulus operator, **%, returns the remainder of a division operation. It can be applied to** floating-point types as well as integer types.
- The following example program demonstrates the **%:**

**EX:**

```
class Modulus {
        public static void main(String args[]) {
                int x = 42;
                double y = 42.25;
                System.out.println("x mod 10 = " + x % 10); // 2
                System.out.println("y mod 10 = " + y % 10); // 2.25
        }
}
```

**Arithmetic Compound Assignment Operators:**

- Java provides special operators that can be used to combine an arithmetic operation with an assignment.
- **EX:**
  - a = a + 4;
- In Java, you can rewrite this statement as shown here:
  - a += 4;
- This version uses the **+= *compound assignment operator. Both statements perform the same*** action.

- Here is another example,
  - a = a % 2;
- which can be expressed as
  - a %= 2;

## 2. Increment and Decrement or Unary Operator

- The ++ and the – – are Java's increment and decrement operators.
- The increment operator increases its operand by one. The decrement operator decreases its operand by one.

**EX1:**

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

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

```
x++;
```

- Similarly, this statement:

```
x = x - 1;
```

is equivalent to

```
x--;
```

**EX:2**

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

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

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

- However, when written like this,

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

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

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

## 3. The Bitwise Operators

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

| Opereator | Result |
|---|---|
| ~ | Bitwise unary NOT |
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR |

- Since the bitwise operators manipulate the bits within an integer, it is important to understand what effects such manipulations may have on a value.
- All of the integer types are represented by **binary numbers of varying bit widths.**

- each position represents a power of two, starting with 2 power 0 at the rightmost bit. The next bit position to the left would be 2 power 1 or 2 and so on..
- The bitwise operators are &, |, ^, and ~. The following table shows the outcome of each operation. • the bitwise operators are applied to each individual bit within each operand.

| A | B | A\|B | A&B | A^B | ~A |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

| | | |
|---|---|---|
| \| | Binary OR Operator | The "Binary OR operator" returns 1 if one of its operands evaluates as 1. if either or both operands evaluate to 1, the result is 1. |
| ^ | Binary XOR Operator | It stands for "exclusive OR" and means "one or the other", but not both. The "Binary XOR operator" returns 1 if and only if exactly one of its operands is 1. If both operands are 1, or both are 0, then the result is 0. |
| & | Binary AND Operator | The "Binary AND operator" returns 1 if both operands are equal to 1. |

**The Bitwise NOT:**

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

**The Bitwise AND:**

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

**Example:**

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

**The Bitwise OR:**

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

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

**The Bitwise XOR:**

- The XOR operator, **^, combines bits such that if exactly one operand is 1, then the result is 1.**
- Otherwise, the result is zero.

**EX:**

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

## 4. Shift Operators

**The Left Shift:**

- The left shift operator, **<<, shifts all of the bits in a value to the left a specified number of times.**
- It has this general form:

  *value* << *num*
- Here, ***num*** *specifies the number of positions to leftshift the value in* ***value.***
- *That is, the* **<<** moves all of the bits in the specified value to the left by the number of bit positions specified by *num.*

```
class OperatorExample {
    public static void main(String args[]) {
        System.out.println(10<<2);//10*2^2=10*4=40
        System.out.println(10<<3);//10*2^3=10*8=80
        System.out.println(20<<2);//20*2^2=20*4=80
        System.out.println(15<<4);//15*2^4=15*16=240
    }
}
```

**The Right Shift:**

- The right shift operator, **>>, shifts all of the bits in a value to the right a specified number of** times. Its general form is shown here:

  *value* >> *num*
- Here, ***num*** *specifies the number of positions to rightshift the value in value.*
- *That is, the* **>>** moves all of the bits in the specified value to the right the number of bit positions specified by ***num***.

**EX:**

```
int a = 32;
a = a >> 2; // a now contains 8
```

- When a value has bits that are "shifted off," those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two loworder bits to be lost, resulting again in **a being set to 8.**

```
int a = 35;
a = a >> 2; // a still contains 8
```

- Looking at the same operation in binary shows more clearly how this happens:

```
00100011 35
```

```
>> 2
00001000 8
class OperatorExample {
        public static void main(String args[]) {
                System.out.println(10>>2);//10/2^2=10/4=2
                System.out.println(20>>2);//20/2^2=20/4=5
                System.out.println(20>>3);//20/2^3=20/8=2
        }
}
```

**The Unsigned Right Shift(>>>)**

- you will generally want to shift **a zero** into the highorder bit **no matter what its initial value was**.
- This is known as an ***unsigned shift***. *To accomplish this, you will use Java's unsigned,* ***shift-right operator(>>>),*** which always shifts zeros into the high-order bit.
- The **>>> operator is often not as useful as you might like, since it is only meaningful** for 32- and 64-bit values.

**EX:**

```
int a = -1;
a = a >>> 24;
System.out.println(a);
```

- The above code fragment demonstrates the **>>>.**
- **Here, a is set to –1, which sets all** 32 bits to 1 in binary.
- This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension.
- This sets **a to 255.**
- int a = -1;
- a = a >>> 24;
- Here is the same operation in binary form to further illustrate what is happening:
- 11111111 11111111 11111111 11111111 –1 in binary as an int >>> 24
- 00000000 00000000 00000000 11111111 255 in binary as an int

**5. Relational Operators:**

- The *relational operators determine the relationship that one operand has to the other.*
- Specifically, they determine equality and ordering. The relational operators are shown here:

| Operator | Result |
|---|---|
| == | Equal to |
| != | Not Equal to |
| > | Greater than |
| < | Lessthan |
| >= | Greaterthan or equal to |
| <= | Lessthan or equal to |

**6. Logical Operators:**

- **The Boolean logical operators shown here operate only on boolean operands.**

| Operator | Result |
|---|---|
| & | Logical AND |
| \| | Logical OR |
| ! | Logical NOT |
| ^ | Logical XOR (exclusive OR) |
| \|\| | Short-circuit OR |
| && | Short-circuit AND |
| ! | Logical unary NOT |
| &= | AND assignment |
| \|= | OR assignment |
| ^= | XOR assignment |
| == | Equal to |
| != | Not equal to |
| ?: | Ternary if-then-else |

- The logical Boolean operators, &, |, and ^, operate on boolean values in the same way that they operate on the bits of an integer.
- The logical ! operator inverts **the Boolean state:**
  - **!true == false and !false == true.**

| A | B | A\|B | A&B | A^B | !A |
|---|---|---|---|---|---|
| False | False | False | False | False | True |
| True | False | True | False | True | False |
| False | True | True | False | True | True |
| True | True | True | True | False | False |

```
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a; int a=10; int b=5;
int c=20;
System.out.println(a<b&&a<c);
System.out.println(a<b&a<c);
```

**Short-Circuit Logical Operators (&& and ||):**

• The **logical && operator** doesn't check second condition if first condition is false.
• It checks second condition only if first one is true.
• The **bitwise & operator** always checks both conditions whether first condition is true or false.

```
class OperatorExample {
        public static void main(String args[]) {
                int a=10,b=5,c=20;
                System.out.println(a<b&&a<c);
                System.out.println(a<b&a<c);
        }
}
```

**EX:**

```
class OperatorExample {
        public static void main(String args[])
        {
                int a=10; int b=5;
                int c=20; System.out.println(a<b&&a++<c); System.out.println(a);
                System.out.println(a<b&a++<c); System.out.println(a);
        }
}
```

• The **logical || operator** doesn't check second condition if first condition is **true**.
• It checks second condition only if first one is false.
• The **bitwise | operator** always checks both conditions whether first condition is **true or false**.

**EX:**

```
class OperatorExample {
        public static void main(String args[]) {
                int a=10, b=5, c=20;
                System.out.println(a>b||a<c);
                System.out.println(a>b|a<c);
                System.out.println(a>b||a++<c);
                System.out.println(a);
                System.out.println(a>b|a++<c);
                System.out.println(a);
        }
}
```

**7. The ? Operator or ternary operator:**

• Java includes a special ***ternary (three-way) operator*** *that can replace certain types of if-then-else* statements.
• The ? has this general form:

***expression1 ? expression2 : expression3***

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

**EX:**

```
public class Test {
        public static void main(String args[]) {
        int a, b; a = 10;
        b = (a == 1) ? 20: 30;
        System.out.println( "Value of b is : " + b ); b = (a == 10) ? 20: 30;
        System.out.println( "Value of b is : " + b );
        }
}
```

**8. The Assignment Operator:**

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

***var = expression;***

• Here, the type of ***var*** *must be compatible with the type of* ***expression.***

***EX:***

```
        int x, y, z;
        x = y = z = 100;
```

• The value of **z = 100 is 100, which is then assigned to y, which in turn is assigned to x.**

## Expressions

- An **expression** is a syntactic construction that has a value.
- **Expressions** are formed by combining variables, constants, and method returned values using operators.

**The Type Promotion Rules:**

- With in expression, it is possible to mix two or more different types of data as long as they are compatible with each other.
- you can mix **short** and **long** within an expression because they are both numeric types.
- When different types of data are mixed an expression, they are all converted to the same type.
- All **char, byte** and **short** values are **promoted** to **int.**
- If one operand is a **long,** the whole expression is promoted to **long.**
- If one operand is a **float operand,** the entire is promoted to float.
- If one operand is **double**, the result is **double**

**Ex:**

```
class rnsit {
        public static void main(String[] args)
        {
                byte b; int i; b=10;
                char ch1='a',ch2='b';
                ch1=(char)(ch1 + ch2);
                i=b * b; // no cast needed
                System.out.println(" i and b:" +i+ " " +b);
                System.out.println(ch1);
        }
}
```

- No cast is needed when **i=b*b** because b is promoted to **int** when the expression is evaluated.
- The same sort of situation also occurs when performing operations on **char.**
- Without the cast, the resulting of adding ch1 and ch2 would be **int**, which can't be assigned to a **char.**

## Type conversion in assignments

**• Booleans cannot be converted to other types.**

• For the other primitive types (char, byte, short, int, long, float, and double), there are two kinds of conversion:

1. **Implicit or Widening or Automatic Type Conversion**

2. **Explicit or Narrowing or Type casting Implicit conversions:** An implicit conversion means that a value of one type is changed to a value of another type without any special directive from the programmer.

**• A data type of lower size (occupying less memory) is assigned to a data type of higher size**. This is done implicitly by the JVM.

• The lower size is widened to higher size. This is also named as **automatic type conversion**.

**EX:**

```
public class Test {
        public static void main(String[] args) {
                int i = 100; //occupies 4 bytes
                long l = i; // occupies 8 bytes
                double f = l; // occupies 8 bytes
                System.out.println("Int value "+i); System.out.println("Long value "+l);
                System.out.println("Float value "+f);
        }
}
```

**OUTPUT**

Int value 100
Long value 100 Float value 100.0

**Explicit conversions:**

• If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.
• This is useful for incompatible data types where automatic conversion cannot be done.
• The general form of cast is **(target-type)expression**
• Here, target-type specifies the desired type to convert the specified value to.
• The thumb rule is, on both sides, the same data type should exist.
• **EX:**

```
class Test
{
        public static void main(String[] args) {
                double d = 100.04;
                long l = (long)d; int i = (int)l;
                System.out.println("Double value "+d); //100.04
                System.out.println("Long value "+l); // 100
                System.out.println("Int value "+i); // 100
        }
}
```

## Control Statements

• There are three types of control statements:
  - Selection statements
  - Iteration statements
  - Jump statements

• ***Selection statements*** *allow your program to choose different paths of execution based* upon the outcome of an expression or the state of a variable.
• ***Iteration statements*** *enable* program execution to repeat one or more statements.
• ***Jump statements*** *allow your program to execute in a nonlinear fashion.*

### Selection statements:

### Java if Statement:

• The Java if statement tests the condition. It executes the *if block* if condition is true.

**Syntax:**

```
if(condition)
{
        //code to be executed
}
```

**EX:**

```
public class IfExample {
        public static void main(String[] args) {
                int age=20; if(age>18)
                {
                        System.out.print("Age is greater than 18");
                }
        }
}
```

### Java if-else Statement

• The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

**Syntax:**

```
        if(condition)
        {
                //code if condition is true
        }
        else {
                //code if condition is false
        }
```

**EX:**

```
public class IfElseExample {
        public static void main(String[] args) {
                int number=13; if(number%2==0)
                {
                        System.out.println(“MBA");
                }
                else{
                        System.out.println(“MCA");
                }
        }
}
```

### Java if-else-if ladder Statement

• The if-else-if ladder statement executes one condition from multiple statements.

**Syntax:**

```
        if(test_expression)
        {
                //execute your code
        }
        else if(test_expression n) {
                //execute your code }
        else {
                //execute your code
        }
```

**Ex:**

```
public class Sample {
public static void main(String args[]) {
        int a = 30, b = 30;
        if (b > a){
                System.out.println(“BANGALORE");
        }
        elseif(a>b){
                System.out.println(“DELHI");
        } else {
                System.out.println(“HYDERABAD");
        }
}
}
```

**Nested if statement:**

```
public class Test {
        public static void main(String args[]) {
                int x = 30; int y = 10;
                if( x == 30 )
                {
                        if( y == 10 )
                {
                        System.out.print(“BANGALORE"); }
                }
        }
}
```

**Switch Statement**

• The switch statement is Java’s multiway branch statement.
• It provides an easy way to dispatch execution to different parts of your code based on the **value of an expression.**
• There can be *one or N number of* ***case values*** for **a switch expression.**
• The case value must be of **switch expression** type only.
• The case value must be *literal or constant.*

**Here is the general form** of a **switch statement:**

```
switch (expression)
{
        case value1:
        // statement sequence break;
        case value2:
        // statement sequence break;
        . .
        case valueN:
        // statement sequence
        break; default:
        // default statement sequence
}
```

**EX:**

```
class SampleSwitch {
        public static void main(String args[]) {
                for(int i=0; i<5; i++) switch(i)
```

```
                {
                        case 0:
                                System.out.println(“JAVA"); break;
                        case 1:
                                System.out.println(“DBMS"); break;
                        case 2:
                                System.out.println(“WEB");
                        break; case 3:
                                System.out.println(“UID"); break;
                        default:
                                System.out.println(“SOFWARE");
                }
        }
}
```

**EX2:**

• The break statement is optional.
• If you omit the break, execution will continue on into the next case.
• It is sometimes desirable to have multiple cases without break statements between them.

```
class MissingBreak {
public static void main(String args[]) {
for(int i=0; i<12; i++) switch(i)
{
        case 0:
        case 1:
        case 2:
        case 3:
        case 4: System.out.println(“MOBILE"); break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
                System.out.println(“APPLICATIONS"); break;
        default:
                System.out.println(“ENGINEERING"); }
        }
}
```

**EX3:**

```
class Switch {
        public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month)
        {
                case 12:
                case 1:
                case 2:
                        season = "Winter";
                        break;
                case 3:
                case 4:
                case 5:
```

```
                season = "Spring"; break;
                case 6:
                case 7:
                case 8:
                        season = "Summer"; break;
                case 9:
                case 10:
                case 11:
                        season = "Autumn"; break;
                default:
                        season = "Bogus Month";
                }
                System.out.println("April is in the " + season + ".");
        }
}
```

**Nested Switch Statements**

```
public class NestedSwitchExample {
        public static void main(String[] args) {
                int i = 0; int j = 1;
                switch(i) {
                        case 0: switch(j) {
                                case 0: System.out.println(“KIRAN");
                                        break;
                                case 1: System.out.println(“SHIVA");
                                        break;
                                default: System.out.println(“BANGALORE");
                                }
                                break;
                        default:
                                System.out.println(“MCA");
                }
        }
}
```

## Iteration statements:

• Java’s iteration statements are **for, while, and do-while.**
• These statements create what we commonly call *loops.*

**while**

• The while loop is Java’s most fundamental loop statement.
• It repeats a statement or block while its controlling expression is true.
• Here is its general form:

```
while(condition)
{
        // body of loop
}
```

• The *condition can be any* ***Boolean expression.***
• *The body of the loop will be executed as long* as the conditional expression is **true.**
• When *condition becomes* ***false****, control passes to the*
• next line of code immediately following the loop.
• The curly braces are unnecessary if only a single statement is being repeated.

**EX:**

```
class While {
        public static void main(String args[]) {
                int n = 10;
                while(n > 0)
                {
                        System.out.println("tick " + n); n--;
                }
        }
}
```

• The body of the while (or any other of Java's loops) can be empty.
• This is because a *null statement (one that consists only of a* ***semicolon****) is syntactically valid in Java.*

**EX2 :**

```
        class NoBody {
                public static void main(String args[]) {
                        int i, j; i = 100; j = 200;
                        while(++i < --j) ; // no body in this loop
                        System.out.println("Midpoint is " + i);
                }
        }
```

**OUTPUT**
**It generates the following output:**
Midpoint is 150

## do-while:

• The do-while loop always executes **its body at least once, because its conditional expression is at the bottom of the loop**.
• Its general form is

```
        do
        {
                // body of loop
        } while (condition);
```

• Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
• If this expression is **true**, the loop will repeat. Otherwise, the loop terminates
**EX:**

```
class DoWhileExample {
        public static void main(String[] args) {
                int i=11;
                do
                {
                        System.out.println(i);
                        i++;
                }while(i<=10);
        }
}
```

## The for loop:

• The for loop will continue to execute as long as the condition tests **true**.

• Once the condition becomes **false,** the loop will exit and program execution will resume on the statement following of for.
• A for loop is most commonly used when you know that a loop will execute a predetermined number of times.
• The general form of the for loop for repeating a single statement is

**for(initialization; condition; increment/decrement )**
**statement**

**Ex:**

```
class example {
        public static void main(String[] args) {
                for(int i=1; i<5; i++) {
                        System.out.println("Count is: " + i);
                }
        }
}
```

**Using the Comma:**

Ex:

```
class mca{
        public static void main(String[] args) {
                int i,j; for(i=0,j=10;i<j ; i++,j--)
                {
                        System.out.println(“i and j: “ + i + “ “ + j);
                }
        }
}
```

• **Missing pieces:**
• Some interesting **for** loop variations are created by leaving pieces of the loop definition empty .
• In java, it is possible for any or all of the initialization, condition or iteration portions of the for loop to be blank.

**EX:**

```
int i; for(i=0,i<10;)
{
        System.out.println(“value of i:“ + i ); i++;
}
```

**Java Infinitive For Loop:**
• If you use two semicolons( ;;) in the for loop, it will be infinitive for loop.
EX:

```
class ForExample {
        public static void main(String[] args) {
                for(;;)
                {
                        System.out.println("infinitive loop");
                }
        }
}
```

## The for-each style for loop

• A **for-each** loop cycles through a collection of objects, such as an array, in strictly sequential fashion from **start to finish.**
• The general form of the **for-each** style is

**for(type itr-var:collection)**
**statement-block**

• Here, **type** specifies the type.
• **itr-val** specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
• The collection being cycled through is specified by **collection.**

**Ex:1**

```
class ForEach {
        public static void main(String[] args) {
                int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }; int sum = 0;
                for(int x : nums) {
                        System.out.println("Value is: " + x); sum += x;
                }
                System.out.println("Summation: " + sum);
        }
}
```

**Ex2:**

```
class Big {
        public static void main (String [] arg) {
        int [] myArr = { 45, 5, 34, 8 } ;
        int big = myArr[0];
        for( int num : myArr)
                if ( big<num )
                        big = num;
        System.out.println("Bigest = " + big) ;
        }
}
```

**EX:**

```
class ForEachExample {
        public static void main(String[] args)
        {
                int arr[]={12,23,44,56,78};
                for(int i:arr)
                {
                        System.out.println(i);
                }
        }
}
```

**Nested for Loops: EX:**

```
class Nested {
        public static void main(String args[]) {
                int i, j;
                for(i=0; i<5; i++) {
                        for(j=i; j<5; j++) {
                                System.out.print(".");
                        }
                        System.out.println();
                }
        }
}
```

## Jump Statements:

• Java supports three jump statements: **break, continue, and return.**
• These statements transfer control to another part of your program. Break:
• In Java, the break statement has **three uses**.
• First, as you have seen, it terminates a statement sequence in a **switch statement**.
• Second, it can be used to exit a loop.
• Third, it can be used as a "civilized" form of goto.

### Using break to Exit a Loop:

• When a **break statement is** encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

**Ex:**

```
class BreakLoop {
        public static void main(String args[]) {
                for(int i=0; i<100; i++) {
                        if(i == 10)
                                break;
                        System.out.println("i: " + i);
                }
                System.out.println("Loop complete.");
        }
}
```

**EX:**

```
class BreakLoop3 {
        public static void main(String args[]) {
        for(int i=0; i<3; i++) {
                System.out.print("Pass " + i + ": ");
                for(int j=0; j<100; j++)
                {
                        if(j == 10) break; System.out.print(j + " ");
                }
                System.out.println();
        }
        System.out.println("Loops complete.");
        }
}
```

### Use break as a form of goto

• Java **does not** have a **goto statement**.
• Java defines an expanded form of the **break statement.**
• By using this form of **break,** you can break **out of one or more blocks of code.**
• These blocks need not be part of a **loop or a switch.** They can be any block.
• You can specify precisely where execution will resume, because this form of break works with a label.
• The general form of the labeled **break statement** is:
• **break label;**
• Here **label** is the name of a label that identifies a **statement or a block of code**.
• When this form of **break** executes, control is transferred out of the labeled statement or block.

```
class Break {
        public static void main(String args[]) {
                boolean t = true;
                first: {
                        second: {
                                third: {
                                        System.out.println("Before the break.");
                                        if(t)
                                                break second;
                                        System.out.println("This won't execute");
                                }
                                System.out.println("This won't execute");
                        }
                        System.out.println("This is after second block.");
                }
        }
}
```

---

```
class BreakLoop4 {
        public static void main(String args[]) {
                outer:
                        for(int i=0; i<3; i++) {
                                System.out.print("Pass " + i + ": "); for(int j=0; j<100; j++)
                                {
                                        if(j == 4) break outer; System.out.print(j + " ");
                                }
                        System.out.println("This will not print");
                }
                System.out.println("Loops complete.");
        }
}
```

---

• Keep in mind that you cannot break to any label which is not defined for an enclosing block.

```
class BreakErr {
        public static void main(String args[]) {
                one: for(int i=0; i<3; i++) {
                        System.out.print("Pass " + i + ": "); }
                        for(int j=0; j<100; j++) {
                                if(j == 10)
                                        break one; // WRONG
                                System.out.print(j + " ");
                }
        }
}
```

**Using continue**

• The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately.
• It can be used with for loop or while loop.
• The Java *continue statement* is used to continue the loop.
• It continues the current flow of the program and **skips the remaining code at the specified condition.**

**EX1:**

```
class ContinueExample {
        public static void main(String[] args) {
                for(int i=1;i<=10;i++) {
                        if(i==5) {
                                continue;
                        }
                        System.out.println(i);
                }
        }
}
```

**EX2:**

```
class Test {
        public static void main(String args[]) {
                int [] numbers = {10, 20, 30, 40, 50};
                        for(int x : numbers )
                        {
                                if( x == 30 )
                                {
                                        continue;
                                }
                                System.out.print( x );
                                System.out.print("\n");
                        }
                }
}
```

• As with the **break statement, continue may specify a label to describe which enclosing** loop to continue.

**EX3:**

```
class ContinueLabel {
        public static void main(String args[]) {
                outer: for (int i=0; i<6; i++) {
                        for(int j=0; j<6; j++) {
                                if(j > i) {
                                        System.out.println(); continue outer;
                                }
                                System.out.print(" " + (i * j));
                        }
                }
                System.out.println();
        }
}
```

- The continue statement in this example terminates the loop counting j and continues with the next iteration of the loop counting i.

**Return:**

- The last control statement is **return.**
- **The return statement is used to explicitly return from** a method.
- **That is, it causes program control to transfer back to the caller of the method.**
- At any time in a method the **return statement can be used to cause execution to branch** back to the caller of the method.
- Thus, the **return statement immediately terminates the** method in which it is executed.

**EX:**

```
public class SampleReturn1 {
        public int CompareNum() {
                int x = 3; int y = 8;
                System.out.println("x = " + x + "\ny = " + y);
                if(x>y)
                        return x;
                else
                        return y;
        }
        public static void main(String ar[]) {
                SampleReturn1 ob = new SampleReturn1();
                int result = ob.CompareNum();
                System.out.println("The greater number am ong x and y is: " + result);
        }
}
```

## Java Identifiers:

• All Java components require names. **Names used for classes, variables and methods are called identifiers.**

• In Java, there are several points to remember about identifiers. They are as follows:

• All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).

• After the first character identifiers can have any combination of characters.

• A key word cannot be used as an identifier.

• Most importantly identifiers are case sensitive.

• Examples of legal identifiers: age, $salary, _value, __1_value

• Examples of illegal identifiers: 123abc, -salary

**Ex**:
```
public class Test {
        public static void main(String[] args) {
                int a = 20;
        }
}
```
• In the above java code, we have 5 identifiers namely :
- **Test** : class name.
- **main** : method name.
- **String** : predefined class name.
- **args** : variable name.
- **a** : variable name.

## Literals

• In java, **literals** refer to fixed values that are represented in their human-readable form.
• **Literals** are also commonly called **constants.**
• Java literals can be any of the primitive data types.
• The way each literal is represented depends on its type.
• **Character constants** are enclosed in **single quotes.**

**Ex:**
• char g='a'
• char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };
• In java, **single** and **double quotes** have special meaning is there. So you cannot use them directly.
• You have to refer backslash(\) character constants.
**Ex:**
```
        char ch='\";
```
• Java supports other type of literal: the **string**
• The **string literal** is a set of characters enclosed by **double quotes.**
**Ex:**
```
        String s="mca rnsit";
        String a= "he said \"Hello\" to me."
```

## Input Characters from the Keyboard:

• To read a **character** from the keyboard we will use **System.in.read().**
• **System. in** is the complement to **System. out**
• It is the input object attached to the keyboard.
• The **read()** method waits until the user presses a key and then returns the result.
• The character is returned as an **integer,** so it must be

**cast** into a **char** to assign it to a **char** variable.
**Ex**
```
import java.io.*; class ex13
{
        public static void main(String[] args) {
                char c;
                System.out.println("enter a key");
                c=(char)System.in.read();
                System.out.println("your key is="+c);
```

```
        }
}
```

## Reference variables and assignment

• We can assign value of reference variable to another reference variable.
• This means that you are changing the object that the reference variable refers to, not making a copy of the object.

**Ex:**

• Rectangle r1 = new Rectangle();
• Rectangle r2 = r1;
• r1 is reference variable which contain the address of Rectangle Object.
• r2 is another reference variable
• r2 is initialized with r1 means – "**r1 and r2**" both are referring same object , thus it does not create duplicate object , nor does it allocate extra memory.

```
class Rectangle {
        double length;
        double breadth;
}

class RectangleDemo {
        public static void main(String args[]) {
                Rectangle r1 = new Rectangle();
                Rectangle r2 = r1;
                Rectangle r3 = r2;
                r1.length = 20;
                r2.length = 25;
                System.out.println("Value of R1's Length : " + r1.length);
                System.out.println("Value of R2's Length : " + r2.length);
                System.out.println("Value of R2's Length : " + r3.length);
        }
}
```

## Methods

• A method contains the statements that define its actions.
• In well-written java code, each method performs only one task.
• You can give a method whatever name you please as long as it is a valid identifier.
• Remember that **main()** is reserved for the method that begins execution of your program.
• A method will have parentheses after its name.
• **For example,** if a method's name is **getval,**it will written **getval()** when its name is used in a sentence
• The general form of a method is,

**return-type name(parameter-list)** {
**//body of method** }

• Here, **ret-type** specifies the type of data returned by the method.
• This can be any valid type, including class types that you can create.
• If the method does not return a value, its return type must be **void.**

**Adding a Method to the class**

```
class Vehicle {
        int passengers;
        int fuelCap;
        int mpg;
        void range() {
                System.out.println("Range is " + fuelCap * mpg); }
        }
class AddMeth {
        public static void main(String[] args) { Vehicle minivan = new Vehicle();
                Vehicle sportscar = new Vehicle();
                int range1, range2;
                minivan.passengers = 7;
                minivan.fuelCap = 16;
                minivan.mpg = 21;
                sportscar.passengers = 2;
                sportscar.fuelCap = 14;
                sportscar.mpg = 12;
                System.out.print("Minivan can carry " + minivan.passengers + ". ");
                minivan.range();
                System.out.print("Sportscar can carry " + sportscar.passengers + ". ");
                sportscar.range();
        }
}
```

**Returning a value**

- Returns values are used for a variety of purposes in programming.
- In some cases, such as **sqrt(),** the return value contains the outcome of some calculation.
- In other cases, the return value simply indicate success or failure.
- In still others, it may contain a status code.
- Methods return a value to the calling routine using this form of **return:**

**return value:**

```
class Vehicle {
        int passengers;
        int fuelCap; int mpg;
        int range() {
                return mpg * fuelCap;
        }
}
class RetMeth {
        public static void main(String[] args) {
        Vehicle minivan = new Vehicle();
        Vehicle sportscar = new Vehicle();
        int range1, range2;
        minivan.passengers = 7;
        minivan.fuelCap = 16;
        minivan.mpg = 21;
        sportscar.passengers = 2;
        sportscar.fuelCap = 14;
        sportscar.mpg = 12;
```

```
                range1 = minivan.range();
                range2 = sportscar.range();
                System.out.println("Minivan can carry " + minivan.passengers + " with
                range of " + range1 + " miles");
                System.out.println("Sportscar can carry " + sportscar.passengers + "
                with range of " + range2 + " miles");
        }
}
```

## Constructors

• A **constructor** *initializes an object immediately upon creation.*
• *It has the* ***same name as the*** **class** in which it resides and is syntactically similar to a method.
• Once defined, the constructor is automatically called immediately after the object is created, before the **new operator completes.**
• Constructors look a little strange because they have no return type, not even **void.**
• This is because the implicit return type of a class constructor is the class type itself.
• It constructs the values i.e. provides data for the object that is why it is known as constructor.
• Typically ,you will use a **constructor** to give initial values to the instance variables defined by the class.

### Types of java constructors

- There are two types of constructors:
- Default constructor (no-arg constructor)
- Parameterized constructor

### Java Default Constructor

- A constructor that have no parameter is known as

### Default constructor

```
class Bike1 {
        Bike1() {
                System.out.println("Bike is created");
        }
        public static void main(String args[]) {
                Bike1 b=new Bike1();
        }
}
```

Default constructor provides the default values to the object like **0, null** etc. depending on the type.

**Ex:**

```
class Student3 {
int id; String name;
        void display() {
                System.out.println(id+" "+name); }
                public static void main(String args[]) {
                Student3 s1=new Student3();
                Student3 s2=new Student3();
                s1.display();
                s2.display();
        }
}
```

• **Java parameterized constructor**
• A constructor that have parameters is known as parameterized constructor.
• Parameterized constructor is used to provide different values to the distinct objects.

**Ex1:**

```
class Student4 {
        int id; String name;
        Student4(int i,String n) {
                id = i; name = n;
        }
        void display() {
                System.out.println(id+" "+name);
        }
        public static void main(String args[]) {
                Student4 s1 = new Student4(111,"Kiran");
                Student4 s2 = new Student4(222,"Arya");
                s1.display();
                s2.display();
        }
}
```

**EX2:**

```
class Box {
        double width; double height; double depth;
        Box(double w, double h, double d) {
                width = w; height = h;
                depth = d;
        }
        double volume() {
                return width * height * depth; }
        }
class BoxDemo7 {
        public static void main(String args[]) {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box(3, 6, 9);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

## this keyword in java

• There can be a lot of usage of **java this keyword**.
• In java, this is a **reference variable** that refers to the current object.
• **this** keyword can be used to refer **current class instance variable.**
• **this** keyword can be used to invoke **current class method (implicitly)**
• **this** can be passed as an argument in the method call.
• **this** can be passed as argument in the constructor call.
• **this** keyword can also be used to return the current class instance.

**Ex:1**

```
class Student10{
        int id; String name;
        Student10(int id,String name){
                id = id;
                name = name;
        }
        void display(){
                System.out.println(id+" "+name);
        }
        public static void main(String args[]){
                Student10 s1 = new Student10(111,“kiran");
                Student10 s2 = new Student10(321,"Ajay");
                 s1.display();
                s2.display();
        }
}
```

• In the above example, parameters(formal arguments) and instance variables are same that is why we are using **this keyword** to distinguish between local variable and instance variable.

**Ex 1:**

```
class Student11
{
        int id; String name;
        Student11(int id,String name) {
                this.id = id; this.name = name;
        }
        void display() {
                System.out.println(id+" "+name);
        }
        public static void main(String args[]) {
                Student11 s1 = new Student11(111,"Kiran");
                Student11 s2 = new Student11(222,"Ajay");
                s1.display();
                s2.display();
        }
}
```

**Ex2:**

```
class A {
        void m() {
                System.out.println("hello m");
        }
        void n() {
                System.out.println("hello n"); this.m();
        }
}
class TestThis4 {
        public static void main(String args[]){
                A a=new A();
                a.n();
        }
}
```

## Arrays:

• **array** is a collection of similar type of elements that have contiguous memory location.
• **Java array** is an object that contains elements of similar data type.
• It is a data structure where we store similar elements. We can store only fixed set of elements in a **java array**.
• Array in java is index based, first element of the array is stored at 0 index.

### Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

### Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime.

### There are two types of array.

Single or One Dimensional Array Multidimensional Array

**A One Dimensional Array** is a list of related variables. Such lists are common in programming .
• To declare a one-dimensional array, you will use this general form:

**type[] array-name=new type[size];**

• Here, **type** declares the **element type** of the array.(The element type is also sometimes referred to as the **base type**).
• The element type determines the **data type** of each element contained in the array.
• The number of elements that the array will hold is determined by **size**.
• Since **arrays** are implemented as **objects,**
• The creation of array is **two-step process**.
• **First**, you declare an **array reference variable**.
• **Second**, you allocate memory for the array, assigning the reference to that memory to the array variable.
• Thus, arrays are dynamically allocated using the **new operator.**

**Ex:**

**int[] sample=new int[10];**

• This declaration works like an object declaration.
• The **sample** variable hold a reference to the memory allocated by **new.**
• This memory is large enough to hold 10 elements of type int.
• It is possible to break the preceding declaration in two.

**int[] sample;**
**sample=new int[10];**

**Ex:**

```
class ArrayDemo {
        public static void main(String[] args) {
                int[] sample = new int[10];
                int i;
                for(i = 0; i < 10; i = i+1) {
                        sample[i] = i;
                        System.out.println("This is sample[" + i + "]: " + sample[i]);
                }
        }
}
```

**Ex2:**

```
class MinMax {
        public static void main(String[] args) {
                int[] nums = new int[5];
                int min, max;
                nums[0] = 99;
                nums[1] = 200;
                nums[2] = 100;
                nums[3] = 18;
                nums[4] = -978;
                min = nums[0];
                max = nums[0];
                for(int i=1; i < 5; i++) {
                        if(nums[i] < min) min = nums[i];
                        if(nums[i] > max) max = nums[i];
                }
                System.out.println("min and max: " + min + " " + max);
        }
}
```

• Arrays can be initialized when they are created.
• The general form for initializing a one-dimensional array is,

**type[] array-name={val1,val2,val3.....valN}**

• Here, the initial value are specified by **val1 through valN.**
• They are assigned in sequence, left to right, in index order.
• Java automatically allocates an array large enough to hold the initializers that you specify.
• There is **no need** to explicitly use the **new operator.**

Ex:

```
class MinMax2 {
        public static void main(String[] args) {
                int[] nums = { 99,5623, 463, 287, 49 };
                int min, max;
                min = nums[0];
                max = nums[0];
                for(int i=1; i < 5; i++) {
                        if(nums[i] < min)
                                min = nums[i];
                        if(nums[i] > max)
                                max = nums[i];
                }
                System.out.println("Min and max: " + min + " " + max);
        }
}
```

Multidimensional arrays

• In java, **multidimensional array or two dimensional array** is a array of arrays.
• A two-dimensional array can be thought of as creating a table of data, with the data organized by **row and column**.
• An individual item of data is accessed by specifying its **row and column** position.
• To declare a two-dimensional array, you must specify the **size of both dimensions**.

**int [][] table=new int[10][20];**

Here table is declared to be a two-dimensional array of int with the size 10 and 20.

**Ex:**

```
class TwoD {
public static void main(String[] args) { int t, i;
int[][] table = new int[3][4]; for(t=0; t < 3; t++) {
for(i=0; i < 4; i++) {
table[t][i] = (t*4)+i+1; System.out.print(table[t][i] + " ");
} System.out.println();
```

## Initializing multidimensional Arrays

• A multidimensional array can be initialized by enclosing each dimension's initializer list within its own set of braces.
• The general form is

```
type[][] array-name= {      {val,val,val.....val},
                            {val,val,val.....val},
                            .... .....
                            {val,val,val.....val}
                          };
```

• Here, **val indicates** an initialization value.
• Each inner block designates a row.
• Within each row, the first value will be stored in the first position of the subarray, the second value in the second array, and so on…
• Notice that commas separate the initializer blocks and that a semicolon follows the closing }.

```
class Testarray3 {
        public static void main(String args[]) {
                int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
                for(int i=0;i<3;i++)
                {
                        for(int j=0;j<3;j++) {
                                System.out.print(arr[i][j]+" ");
                        }
                        System.out.println();
                }
        }
}
---------------------------
class Test5 {
        public static void main(String args[]) {
                int[][] a={{1,3,4},{3,4,5}};
                int[][] b={{1,3,4},{3,4,5}}
                int c[][]=new int[2][3];
                for(int i=0;i<2;i++) {
                        for(int j=0;j<3;j++) {
                                c[i][j]=a[i][j]+b[i][j]; System.out.print(c[i][j]+" ");
                        }
                        System.out.println();//new line
                }
        }
}
```

### Using the length MEMBER

• Arrays are implemented as objects, each array has associated with it a **length** instance variable that contains the number of elements that the array can hold.
• In other words, length contains **the size of the array.**

**Ex:**

```
class LengthDemo {
        public static void main(String[] args) {
                int[] list = new int[10];
                int[] nums = { 1, 2, 3 };
                int[][] table = {
                                        {1, 2, 3},
                                        {4, 5},
                                        {6, 7, 8, 9}
                                };
                System.out.println("length of list is " + list.length);
                System.out.println("length of nums is " + nums.length);
                System.out.println("length of table is " + table.length);
                System.out.println("length of table[0] is " + table[0].length);
                System.out.println("length of table[1] is " + table[1].length);
                System.out.println("length of table[2] is " + table[2].length);
                System.out.println();
        }
}
```

### Irregular arrays or Jagged arrays

• When you allocate memory for a multidimensional array, you need to specify only the memory for the first (leftmost) dimension.
• You can allocate the remaining dimensions separately.
• Each row can have different number of columns.

**Ex:**

```
        int table[][] = new int[3][];
        table[0] = new int[1];
        table[1] = new int[2];
        table[2] = new int[3];
```

**Ex:**

```
class CheckEquality {
        public static void main (String [] arg) {
                int [][] jagged = { { 10, 8 }, { 40, 88, 20, 18 }, { 90, 28, 50 } };
                int big = jagged[0][0];
                for ( int row = 0 ; row < jagged.length ; row++)
                        for ( int col = 0 ; col < jagged[row].length ; col++)
                                if ( big< jagged[row][col] ) big = jagged[row][col];
                                        System.out.println("Biggest = " + big) ;
        }
}
```

### Alternative array declaration systax

• There is a second form that can be used to declare an array.

**type var-name[];**

• Here the square brackets follows the name of the array variable, not the type specifier.

**Ex:** int rnsit[]=new int[5];
int[] rnsit=new int[5];

• This alternative declaration lets you declare both array and nonarray variables of the same type in a single declaration.

**Ex: int alpha,beta[],gamma;**

### Assigning array references

• As with other objects, when you assign one array reference variable to another, you are simply changing what objects that variables refers to.

**Ex:**

```
class AssignARef {
        public static void main(String[] args) {
                int i;
                int[] nums1 = new int[10];
                int[] nums2 = new int[10];
                for(i=0; i < 10; i++)
                        nums1[i] = i; for(i=0; i < 10; i++)
                nums2[i] = -i;
                System.out.print("Here is nums1: ");
                for(i=0; i < 10; i++)
                        System.out.print(nums1[i] + " ");
                System.out.println();
                System.out.print("Here is nums2: ");
                for(i=0; i < 10; i++)
                        System.out.print(nums2[i] + " ");
                System.out.println();
                nums2 = nums1;
                System.out.print("Here is nums2 after assignment: ");
                for(i=0; i < 10; i++)
                        System.out.print(nums2[i] + " ");
                System.out.println();
        }
}
```

## Overloading Methods

• If a class have multiple methods by **same name** but **different parameters,** it is known as **Method Overloading.**

• There are two ways to overload the method in java By changing number of arguments By changing the data type

**EX:1**

```
class OverloadDemo {
        void test()
        {
                System.out.println("No parameters");
        }
        void test(int a)
        {
                System.out.println("a: " + a);
        }
        void test(int a, int b)
        {
                System.out.println("a and b: " + a + " " + b);
```

```
        }
        void test(double a)
        {
                System.out.println("double a: " + a*a);
        }
}

class Overload {
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                double result;
                ob.test();
                ob.test(10);
                ob.test(10, 20);
                ob.test(123.25);
        }
}
```

• In some cases, Java's automatic type conversions can play a role in overload resolution.

**EX:2**

```
class OverloadDemo {
        void test()
        {
                System.out.println("No parameters");
        }
        void test(int a, int b)
        {
                System.out.println("a and b: " + a + " " + b);
        }
        void test(double a)
        {
                System.out.println("Inside test(double) a: " + a);
        }
}
class Overload {
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                int i = 88;
                ob.test();
                ob.test(10, 20);
                ob.test(i);
                ob.test(123.2);
        }
}
```

**EX:3**

```
class mca10 {
        int arun(int a)
        {
                return a;
        }
        double arun(double a, double b)
        {
```

```
                return a*b;
        }
}

class mca85 {
        public static void main(String args[]) {
                mca10 s=new mca10();
                System.out.println(s.arun(10));
                System.out.println(s.arun(20.6, 12.5));
        }
}
```

## Overloading Constructors

• In addition to overloading normal methods, you can also overload constructor methods.

**EX:**

```
class Box {
double width; double height; double depth;
Box(double w, double h, double d)
{
        width = w; height = h;
        depth = d; }
        Box()
        {
                width = -1; height = -1; depth = -1;
        }
        Box(double len)
        {
                width = height = depth = len; }
        double volume()
        {
                return width * height * depth;
        }
}
class OverloadCons {
        public static void main(String args[]) {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```

## Recursion

• Java supports *recursion. Recursion is the process of defining something in terms of itself.*
• *As* it relates to Java programming, recursion is the attribute that allows a method to call itself.
• A method that calls itself is said to be *recursive.*

**Ex 1:**

```
class Factorial {
        int fact(int n) {
        I       nt result; if(n==1)
                        return 1;
                result = fact(n-1) * n;
                return result;
        }
}
class Recursion {
        public static void main(String args[]) {
                Factorial f = new Factorial();
                System.out.println("Factorial of 3 is " + f.fact(3));
                System.out.println("Factorial of 4 is " + f.fact(4));
                System.out.println("Factorial of 5 is " + f.fact(5));
        }
}
```

**Ex 2:**

```
class StarDrawer {
        void drawStars(int n) {
                if(n == 1)
                        System.out.print("*");
                else {
                        System.out.print("*");
                        drawStars(n-1);
                }
        }
}
class StarDrawingDemo {
        public static void main(String[] args) {
                StarDrawer d = new StarDrawer();
                d.drawStars(1);
                System.out.println();
                d.drawStars(2);
                System.out.println();
                d.drawStars(3);
                System.out.println();
                d.drawStars(10);
                System.out.println();
        }
}
```

## Understanding static

• Normally, a class member must be accessed only in conjunction with an **object of its class.**

• However, it is possible to create a member that can be used by itself, without reference to a specific instance.
• To create such a member, precede its declaration with the keyword **static.**
• We can apply java static keyword with **variables, methods, blocks.**
• The static can be: variable (also known as class variable) method (also known as class method) block
• Methods declared as **static have several restrictions:**
• They can only call other **static methods.**
• They must only access **static data.**
• They cannot refer to **this or super in any way.**

**Ex 1:**

## Static variables

```
class Student8 {
        int rollno; String name; static String college ="RNSIT";
        Student8(int r,String n) {
                rollno = r; name = n;
        }
        void display () {
                System.out.println(rollno+" "+name+" "+college);
        }
        public static void main(String args[]) {
                Student8 s1 = new Student8(100,"Kiran");
                Student8 s2 = new Student8(222,"Arya"); //Student8.college="BBDIT";
                s1.display(); s2.display();
        }
}
```

## Static Methods

• If you apply static keyword with any method,it is known as **static method.**
• A static method belongs to the **class** rather than the **object of a class**.
• A static method can be invoked without the need for creating an instance of a class.
• A static method can access static data member and can change the value of it.

**Ex:2 //TestStatic.java**

```
class Student9 {
        int rollno; String name;
        static String college = "BMSIT";
        static void change()
        {
                college = "RNSIT";
        }
        Student9(int r, String n) {
                rollno = r; name = n;
        }
        void display () {
                System.out.println(rollno+" "+name+" "+college);
        }
        public static void main(String args[])
        {
                Student9.change();
                Student9 s1 = new Student9 (111,"Kiran");
                Student9 s2 = new Student9 (222,"Ravi");
```

```
                Student9 s3 = new Student9 (333,“Ajay");
                s1.display();
                s2.display();
                s3.display();
        }
}
```

**Java static block**

• Is used to initialize the static data member.
• It is executed before main method at the time of classloading.

**Ex 1:**

```
class A2
{
        static{
                System.out.println("static block is invoked");
        }
        public static void main(String args[]) {
                System.out.println("Hello main");
        }
}
```

**Ex 2:**

```
class UseStatic {
        static int a = 3; static int b;
        static void meth(int x)
        {
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
        }
        static
        {
                System.out.println("Static block initialized.");
                b = a * 4;
        }
        public static void main(String args[]) {
                meth(42);
        }
}
```

## Garbage Collection

• In java, garbage means unreferenced objects.
• Garbage Collection is process of reclaiming the runtime unused memory automatically.
• **In other words, it is a way to destroy the unused objects**.
• To do so, we were using **free()** function in C language and **delete()** in C++.
• But, in java it is performed automatically.
• So, java provides better memory management.
• when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

**How can an object be unreferenced?**

• There are many ways:

• By nulling the reference

```
Employee e=new Employee();
        e=null;
```

• By assigning a reference to another

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2; //now the first object referred by e1 is available e for garbage collection
```

• By anonymous object. **new Employee();**

## finalize() method

• The finalize() method is invoked each time before the object is garbage collected.
• This method is called **finalize()**,and it can be used in very specialized case to ensure that an object **terminates cleanly.**
• To add a finalizer to class, you must define the **finalize()** method.
• The general form:

```
protected void finalize()
{
        //code
}
```

• It is important to understand that **finalize( ) is only called just prior to garbage collection.**