**Object Oriented Programming**

- In object oriented programming approach there is a collection of objects. Each object consists of corresponding data structures and the required operations (procedures). This is basically **the bottom up problem** solving approach.
- The object oriented programming approach is developed in order to remove some of the flaws of procedural programming.
- OOP has a complete focus on data and not on the procedures. In OOP, the data can be protected from accidental modification.
- In OOP the most important entity called object is created.
- Each object consists of data attributes and the methods. The methods or functions operate on data attributes.

There are various characteristics of object oriented programming and those are -

(1) Abstraction (2) Object and Classes (3) Encapsulation

(4) Inheritance and (5) Polymorphism.

**Abstraction**

**Definition:** Abstraction means representing only essential features by hiding all the implementation details. In object oriented programming languages like C++, or Java class is an entity used for data abstraction purpose.

**Example**

```
class Student
{
        int roll;
        char name [10];
public;
        void input ( );
        void display ( );
}
```

In main function we can access the functionalities using object. For instance

```
        Student obj;
        obj.input ( );
        obj.display ( );
```

Thus only abstract representation can be presented, using class.

**Object**

- Object is an instance of a class.
- Objects are basic run-time entities in object oriented programming.
- In C++ the class variables are called objects. Using objects we can access the member variable and member function of a class.
- Object represent a person, place or any item that a program handles.

**For example** - If the class is country then the objects can be India, China, Japan, U.S.A and so on.

- A single class can create any number of objects.

**Declaring objects -**

The syntax for declaring object is -

Class_Name Object_Name;

**Example**

Fruit f1;

For the class **Fruit** the object **f1** can be created.

**Classes**

- A class can be defined as an entity in which data and functions are put together.
- The concept of class is similar to the concept of structure in C.

**Syntax of class is as given below**

```
class name_of_class
{
        private :
        variables declarations;
        function declarations;
        public :
        variable declarations;
        function declarations;
} ; do not forget semicolon
```

**Example**

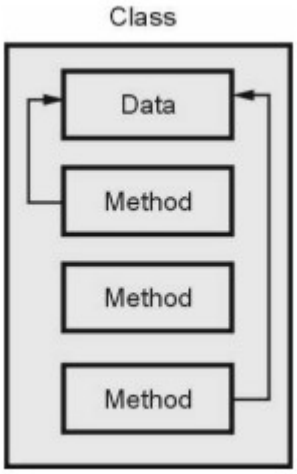
```
class rectangle
        {
        private :
        int len, br;
        public :
        void get_data ( ) ;
        void area( );
        void print_data ( );
        };
```

**Explanation**

- The class declared in above example is **rectangle.**
- The class name must be preceded by the keyword **class.**
- Inside the body of the class there are two keywords used **private** and **public**. These are called **access specifiers.**

| S.No | Class | Object |
|---|---|---|
| 1. | For a single class there can be any number of objects. For example - If we define the class as River then Ganga, Yamuna, Narmada can be the objects of the class River. | There are many objects that can be created from one class. These objects make use of the methods and attributes defined by the belonging class. |
| 2. | The scope of the class is persistent throughout the program. | The objects can be created and destroyed as per the requirements. |
| 3. | The class can not be initialized with some property values. | We can assign some property values to the objects. |
| 4. | A class has unique name. | Various objects having different names can be created for the same class. |

**Encapsulation**




- Encapsulation is for the detailed implementation of a component which can be hidden from rest of the system.
- In C++ the data is encapsulated.

**Definition :** Encapsulation means binding of data and method together in a single entity called class.

- The data inside that class is accessible by the function in the same class. It is normally not accessible from the outside of the component.

**Difference between Encapsulation and Abstraction**

| S.No | Data encapsulation | Data abstraction |
|---|---|---|
| **1.** | It is a process of binding data members of a class to the member functions of that class. | It is the process of eliminating unimportant details of a class. In this process only important properties are highlighted. |
| **2.** | Data encapsulation depends upon object data type. | Data abstraction is independant upon object data type. |
| **3.** | It is used in software implementation phase. | It is used in software design phase. |
| **4.** | Data encapsulation can be achieved by inheritance. | Data abstraction is represented by using abstract classes. |

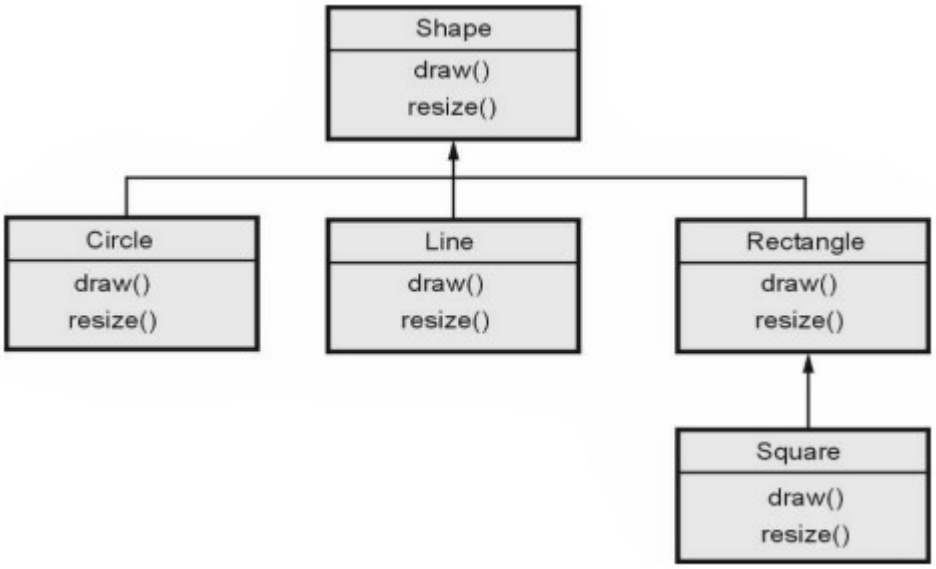
**Inheritance**

**Definition :** Inheritance is a property by which the new classes are created using the old classes. In other words the new classes can be developed using some of the properties of old classes.

- Inheritance support hierarchical structure.

- The old classes are referred as **base classes** and the new classes are referred as **derived classes**. That means the derived classes inherit the properties (data and functions) of base class.

**Example :** Here the **Shape** is a base class from which the **Circle, Line** and **Rectangle** are the derived classes. These classes inherit the functionality **draw()** and **resize()**. Similarly the **Rectangle** is a base class for the derived class **Square**. Along with the derived properties the derived class can have its own properties. For example the class Circle may have the function like **backgrcolor()** for defining the back ground color.




**Polymorphism**

- Polymorphism means many structures.

**Definition :** Polymorphism is the ability to take more than one form and refers to an operation exhibiting different behavior in different instances (situations).

- The behavior depends on the type of data used in the operation. It plays an important role in allowing objects with different internal structures to share the same external interface.
- Without polymorphism, one has to create separate module names for each method.

**For example** the method **clean** is used to clean a **dish** object, one that cleans a **car** object, and one that cleans a **vegetable** object.

- With polymorphism, you create a single "clean" method and apply it for different objects.

| S.No | Inheritance | Polymorphism |
|---|---|---|
| **1.** | Inheritance is a property in which some of the properties and methods of base class can be derived by the derived class. | Polymorphism is ability for an object to used different forms. The name of the function remains the same but it can perform different tasks. |
| **2.** | Various types of inheritance can be single inheritance, multiple inheritance, multilevel inheritance and hybrid inheritance. | Various types of polymorphism are compile time polymorphism and run time polymorphism. In compile time polymorphism there are two types of overloading possible. - Functional overloading and operator overloading. In run time polymorphism there is a use of virtual function. |

**Benefits and Drawbacks of OOP**

**Benefits**

Following are some advantages of object oriented programming -

1. Using inheritance the redundant code can be eliminated and the existing classes can be used.

2. The standard working modules can be created using object oriented programming. These modules can then communicate to each other to accomplish certain task.

3. Due to data hiding property, important data can be kept away from unauthorized access.

4. It is possible to create multiple objects for a given class.

5. For upgrading the system from small scale to large scale is possible due to object oriented feature.

6. Due to data centered nature of object oriented programming most of the details of the application model can be captured.

7. Message passing technique in object oriented programming allows the objects to communicate to the external systems.

8. Partitioning the code for simplicity, understanding and debugging is possible due to object oriented and modular approach.

**Drawbacks**

Following are some drawbacks of OOP -

1. The object oriented programming is complex to implement, because every entity in it is an object. We can access the methods and attributes of particular class using the object of that class.

2. If some of the members are declared as private then those members are not accessible by the object of another class. In such a case you have to make use of inheritance property.

3. In Object oriented programming, every thing must be arranged in the forms of classes and modules. For the lower level applications it is not desirable feature.

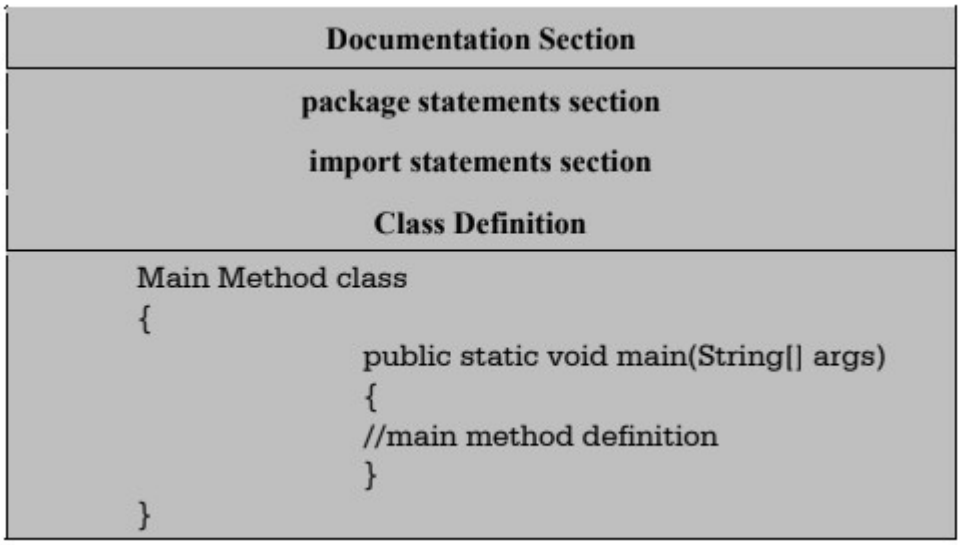
**Applications of OOP**

Various applications in which object oriented programming is preferred are –

- Business logic applications
- Knowledge based and expert systems
- Web based applications
- Real time systems
- Simulation and modelling
- Object oriented databases
- Applications in distributed environment
- CAD/CAM systems
- Office automation system
- Games programming

**Structure of Java Program**

The program structure for the Java program is as given in the following figure -




**Documentation section**: The documentation section provides the information about the source program. This section contains the information which is not compiled by the Java. Everything written in this section is written as comment.

**Package section:** It consists of the name of the package by using the keyword **package.** When we use the classes from this package in out program then it is necessary to write the package statement in the beginning of the program.

**Import statement section :** All the required java API can be imported by the import statement. There are some core packages present in the java. These packages include the classes and method required for java programming. These packages can be imported in the program in order to use the classes and methods of the program.

**Class definition section:** The class definition section contains the definition of the class. This class normally contains the data and the methods manipulating the data.

**Main method class:** This is called the main method class because it contains the main() function. This class can access the methods defined in other classes.

**Introduction to classes**

**Fundamentals of the class**

A class is a group of objects that has common properties. It is a **template or blueprint** from which objects are created. The objects are the instances of class. Because an object is an instance of a class, you will often see the two words *object and instance* used interchangeably.

A class is used to define new type of the data. Once defined, this new type can be used to create objects of its type. The class is the logical entity and the object is the logical and physical entity.

The general form of the class

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a **class** definition is shown here:

```
class classname
{
type instance-variable1;
type instance-variable2;
……..
……..
type instance-variableN;
type method1(parameterlist)
{
//body of the method1
}
type method2(parameterlist)
{
//body of the method2
}
……..
……..
type methodN(parameterlist)
{
```

```
//body of the method
}
}
```

- The data or variables, defined within the class are called, **instance variable.**
- The methods also contain the code.
- The methods and instance variable collectively called as **members.**
- Variable declared within the methods are called **local variables.**

**A Simple Class**

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width, height, and depth.** Currently, Box does not contain any methods.

```
class Box
{ //instance variables
double width;
double height;
double depth;
}
```

As stated, a class defines new data type. The new data type in this example is, Box. This defines the template, but does not actually create object.

**Creating the 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' key word is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

**Step 1:**

**Box b;**

**Effect:** b null

Declares the class variable. Here the class variable contains the value **null.** An attempt to access the object at this point will lead to Compile-Time error.

**Step 2:**

**Box b=new Box();**

Here new is the keyword used to create the object. The object name is b. The **new** operator allocates the memory for the object, that means for all instance variables inside the object, memory is allocated.

**Effect:** b


**Box Object**

**Step 3:**

There are many ways to initialize the object. The object contains the instance variable.The variable can be assigned values with reference of the object.

```
b.width=12.34;
b.height=3.4;
b.depth=4.5;
```

Here is a complete program that uses the **Box** class:

```
class Box
{
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo
```

```
{
public static void main(String args[])
{
//declaring the object (Step 1) and instantiating (Step 2) object
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables (Step 3)
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

**Introduction to Methods**

Classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility.

**This is the general form of a method:**

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

Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void.** The name of the method is specified by name. This can be any legal identifier other than those already used by other items within the current scope. The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the **arguments** passed to the

method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

**return *value;***

Here, value is the value returned.

**Adding a method to the Box class**

```
class Box
{
Box.java
double width, height, double depth;
// display volume of a box
void volume()
{
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

Here the method name is "volume()". This methods contains some code fragment for computing the volume and displaying. This method can be accessed using the object as in the following code:

```
class BoxDemo3
{
BoxDemo3.java
public static void main(String args[])
{
Box mybox1 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
```

```
mybox1.depth = 15;
/* assign different values to mybox2's
// display volume of first box
mybox1.volume();
// display volume of second box
}
}
```

**Constructors**

It can be tedious to initialize all of the variables in a class each time an instance is created. Even if we use some method to initialize the variable, it would be better this initialization is done at the time of the object creation.

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 is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

**Example Program:**

```
class Box
{
double width;
double height;
double depth;
// This is the constructor for Box.
Box()
{
```

```
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume()
{
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
double vol;
// get volume of first box vol =
mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
}}
```

**Parameterized Constructors**

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor.

```
/* Here, Box uses a parameterized constructor to initialize the dimensions of
a box. */
class Box
{
double width;
double height;
```

```
double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
{
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box vol =
mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box vol =
mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

**Constructor-overloading**

In Java it is possible to define two or more class constructors that share the same name, as long as their parameter declarations are different. This is called constructor overloading.

When an overloaded constructor is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded constructor to actually call. Thus, overloaded constructors must differ in the type and/or number of their parameters.

**Example: All the constructors' names will be same, but their parameter list is different.**

**Garbage Collection**

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection. Java has its own set of algorithms to do this as follow

There are Two Techniques: 1) Reference Counter 2) Mark and Sweep. In the Reference Counter technique, when an object is created along with it a reference counter is
maintained in the memory. When the object is referenced, the reference counter is incremented
by one. If the control flow is moved from that object to some other object, then the counter value
is decremented by one. When the counter reaches to zero (0), then it's memory is reclaimed.

In the Mark and Sweep technique, all the objects that are in use are marked and are called

live objects and are moved to one end of the memory. This process we call it as compaction. The memory occupied by remaining objects is reclaimed. After these objects are deleted from the memory, the live objects are placed in side by side location in the memory. This is called copying.

It works like this: when noreferences to an object exist, that object is assumed to be no longer needed, and the memoryoccupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.Garbage collection only occurs during the execution of your program. The main job of this is to release memory for the purpose of reallocation.

Furthermore, different Java run-time implementations will take varying approaches to garbage collection.

**The finalize( ) Method**

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize( )method, you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects

The finalize( ) method has this general form:

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

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.

**Overloading methods**

In Java it is possible to define **two or more methods** within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. ***Method overloading*** is one of the ways that Java supports ***polymorphism***.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call

**Here is a simple example that illustrates method overloading:**

```
// Demonstrate method overloading.

class OverloadDemo

{

void test()

{

System.out.println("No parameters");

}

// Overload test for one integer parameter.

void test(int a)

{

System.out.println("a: " + a);

}

}

class Overload

{
```

```
public static void main(String args[])
{
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
}
}
```

The test() method is overloaded two times, first version takes no arguments, second version takes one argument. When an overloaded method is invoked, Java looks for a match between arguments of the methods.Method overloading supports polymorphism because it is one way that Java implementsthe ―one interface, multiple methods‖ paradigm.

**The this Keyword**

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the ***this*** keyword. ***this*** can be used inside any method to refer to the current object. That is, ***this*** is always a reference to the object on which the method was invoked. You can use ***this*** any where a reference to an object of the current class' type is permitted.To better understand what this refers to, consider the following version of ***Box( )***:

```
// A redundant use of this.
Box(double w, double h, double d)
{
this.width = w;
this.height = h;
this.depth = d;
}
```

**Note: This is mainly used to hide the local variables from the instance variable.**

**Example:**

```
class Box {
//instance variable
double width, height, depth;
Box(double width, double height, double depth)
{
//local variables are assigned, but not the instance variable
width=width;
height=height;
depth=depth;
}}
```

**To avoid the confusion, this keyword is used to refer to the instance variables, as follows:**

```
class Box {
//instance variable
double width, height, depth;
Box(double width, double height, double depth)
{
//the instance variable are assigned through the this keyword.
this.width=width;
this.height=height;
this.depth=depth;
}}
```

**Data Types in Java**

Data types are mainly used in Java since, it is a strongly typed language. In Java, compiler evaluates all the type of variables, values and expressions in order to maintain type compatibility. Meanwhile, errors are reduced and reliability is increased. Primitive data type is one of the simplest data type that consist of a single value.

Java supports eight types of primitive data types which are grouped into four types as shown below,

(i) Integer data type

(ii) Floating-point data type

(iii) Character data type

(iv) Boolean data type.

**Integer Data Type**

The integer type can represent the signed integer values that may be positive or negative. Java does not allow unsigned

integer values since, they are not required. This data type itself consists of four different types. They are as follows,

(a) byte

(b) short

(c) int

(d) long

**a)byte**

The byte data type is used to maintain integer values that have a size of 1 byte i.e., 8 bits. The range of byte data type is from –128 to 127.

**Syntax:** byte variable_name;

**Example:** byte x = 3;

**b)short**

The short data type represents the signed integer values and the size of this data type is 2 bytes i.e., 16 bits. The range of this data type is from –32, 768 to 32, 767.

**Syntax:** short variable_name;

**Example**

short a, b;

a = 2376;

**c) int**

The int datatype is most preferable among all the integer data types. Since, it can control index of arrays and loops. The size of this data type is 4 bytes i.e., 32 bits and the range is from –2,147,483,648 to 2,147,483,647.

**Syntax:** int variable_name;

**Example**

int a, b;

a = 2376789;

**d) long**

The long data type is a signed 8 byte (64 bit) which has a range from –9,223,372,036,854,775,808 to 9,223,372, 036,854,775,807. Generally, this data type is used to manage the integer values that have a range greater than int.

**Syntax:** long variable_name;

**Example**

long x, y;

x = 12345678910L;

**Program**

```
import java.io.*;
import java.util.*;
class IntegerExample
```

```
{
public static void main(String[ ] args)
{
byte x;
short y;
int z;
long p;
x = 2;
y = 23456;
z = 117438;
p = 732227363054L;
System.out.println("The value of byte data type is:" + x);
System.out.println("The value of short data type is:" + y);
System.out.println("The value of int data type is:" + z);
System.out.println("The value of long data type is:" + p);
}}
```

**ii) Floating-point Data Type**

Floating-point type can be defined as a data type which indicates the fractional values. The two types of floating-point data types are as follows,

(a) float

(b) double

**a) float**

The float data type is used to represent single-precision numbers and the size of this data type is 4 bytes i.e., 32 bits. The maximum value of float literal is approximately 3.4 × 1038.

**Syntax:** float variable_name;

**Example:** float x = 3.14F;

**b) double**

Double data type is used to represent double-precision numbers and the size of this data type is 8 bytes i.e., 64 bits. The largest value of double

literal is approximately 1.8 × 3308. The double data type is frequently used in java since, the math functions in Java class library uses double values.

**Syntax:** double variable_name;

**Example:** double x;

**Program**

```
import java.io.*;
import java.util.*;
class FloatExample
{
public static and main(String [ ] args)
{
float x;
double y;
x = 3.14F;
y = Math.sqrt(x);
System.out.println("The float value of x is:" + x);
System.out.println("The double value of y is:" + y);
}}
```

**iii) Character Data Type**

The character (i.e., char) data type represents characters and the size of this data type is 16-bits (2 bytes). Java language uses 2 bytes since it supports unicode characters. Unicode characters can be defined as a set of characters that indicates all the characters of human languages. The character data type stores unsigned 16-bit characters with a range of 0 to 65,536. The character value must be enclosed in single quotes.

**Syntax:** char variable_name;

**Example**

```
char x;
x = 'n';
```

**Program**

```
import java.io.*;
import java.util.*;
class CharacterExample
{
public static void main(String[ ] args)
{
char y;
y = 'N';
System.out.println("The value of y is:" + y);
y – –; //The char value is decremented
System.out.println("The new value of y is:" + y);
y = 82; //The char data type can be assigned with integer
System.out.println("The another value of y is:" + y);
}}
```

**iv) Boolean Data Type**

The boolean data type indicates whether the value of an expression is either true or false. Therefore, a variable or expression which is declared as boolean type can use these two keywords.

**Syntax:** boolean variable_name;

**Example**

```
boolean a;
a = true;
```

or

```
a = false;
```

Program

```
import java.io.*;
import java.util.*;
class BooleanExample
{
public static void main(String[ ] args)
```

```
{
boolean a;
int x, y, z;
x = 10;
y = 20;
z = x > y;
a = false;
if(a)
{
System.out.println(“a is :” + a);
System.out.println(“x is not greater than y”);
}
z = y > x;
a = true;
if(a)
{
System.out.println(“a is :”+a);
System.out.println(“y is greater than x”);
}}}
```

The above program can print the boolean values true/ false on to the standard output device using println( ). Next, the control statement ‘if’ can be managed by using the Boolean variable ‘a’ directly. However, there is no need to use the statement, which is shown below,

```
if(a = = true)
{
//statements
}
```

**Variable**

A variable is the name given to a unit/memory location that stores data value of the variable. The name given to the variable is known as Identifier. The variable value may change several times during the execution of program. Each variable is associated with its scope that depicts the life time and visibility of the variables.

**Declaration of a Variable**

A variable can be declared in order to avoid the confusion to the compiler. The declaration of a variable is preceded by its data type so that it can accept data values of its associated type.

**Syntax**

***datatype variable_name;***

Here, data type is the type of variable such as int, double, char and variable_name indicates the name of the variable.

**Example**

```
int x;
double y;
```

When a variable is declared, an instance is created for its data type which identifies the capability of variable. This is because, when a variable is declared as int then it cannot store the values of boolean data type. Thus, strong type checking is supported by Java.

**Initialization of a Variable**

Initialization of a variable can be defined as a process of assigning a value to the variable. This can be done directly during the declaration of a variable or after the declaration of variable. The syntax for initializing a variable is as follows,

**Syntax**

***datatype variable_name = value;***

***datatype variable_name;***

***variable_name = value;***

**Example**

```
int x = 10;
float y = 1.5F;
char ch;
ch = 'N';
```

Multiple variables that are declared with similar type can be initialized simultaneously. This can be done by using comma operator.

**Example**

```
int x, y, z = 8, p = 9;
```

**Dynamic Initialization of a Variable**

Dynamic initialization of a variable can be defined as a process of initializing the variable at run-time i.e., during execution of the program. This can be performed by assigning a valid expression to required variable.

**Program**

```
import java.io.*;
import java.util.*;
class VariableExample
{
public static void main(String[ ] args)
{
int x = 10, y = 20;
int z;
z = x + y;
System.out.println("Addition of x, y = " + z);
z = x – y;
System.out.println("Difference between x, y ="+ z);
} }
```

In the above program, two variables x, y are declared and initialized with 10 and 20 respectively. Another variable z is declared and can be initialized dynamically. After the computation of addition operation, z is initialized with 30 and after subtraction, z is initialized with –10. This depicts that, the value of a variable may vary during execution of the program.

**Type Conversion**

Type conversion refers to the process of converting the data of one type into another type. It is one of the major concept of Java. Proper care need to be taken while type conversion by the programmers(since they are likely to commit errors). For example, consider two integer variables namely x = 4 and y = 7. When a division operation is performed on them the result would be 0.57 which is a floating point number. Since x and y are declared as integers, the Java compiler considers their result(x/y) also as integer by discarding the decimal value eventually. Hence, proper type conversion is necessary to ensure accurate results. There are certain basic rules that are to be followed while type conversion.

1. Unsigned char and unsigned short can be converted into unsigned integer.
2. Character and short can be converted into integer.
3. Float can be converted into double.

**Example**

```
import java.io.*;
class conversion {
public static void main(String args[ ]) {
float sum,x;
int k;
sum = 0.0f;
for(k =1; k <=5; k++)
{
x = 1/(float)k;
sum = sum + x;
System.out.println("Value of x:" + x);
}
System.out.println("sum:" + sum);
}}
```

**Output**

Value of x:1.0

Value of x:0.5

Value of x:0.33333334

Value of x:0.25

Value of x:0.2

Sum=2.2833335

**Type Casting**

'Type casting' is an explicit conversion of a value of one type into another type. And simply, the data type is stated using parenthesis before the value. Type casting in Java must follow the given rules,

1. Type casting cannot be performed on Boolean variables. (i.e., boolean variables can be cast into other data type).
2. Type casting of integer data type into any other data type is possible. But, if the casting into smaller type is performed, it results in loss of data.
3. Type casting of floating point types into other float types or integer type is possible, but with loss of data.
4. Type casting of char type into integer types is possible. But, this also results in loss of data, since char holds 16-bits the casting of it into byte results in loss of data or mix up characters.

The general form of type casting is as follows,

**Syntax**

(target-type) expression;

Here, target-type is the specification to convert the expression into required type.

**Example**

```
float a, b;
int c = (int) (a + b);
```

The result of float type variables can be converted into integer type explicitly. The parentheses for the expression a + b is necessary. Otherwise, the

variable can be converted into int but not the result of a + b. The casting of double to int is necessary since, they are not compatible. Furthermore, Java performs assignment of value of one type to a variable of another type without performing type casting. Here, the conversion is performed automatically which is referred to as automatic type conversion. It can take place when destination type hold sufficient precedence to store source value.

While performing the conversion between two incompatible type, the use of word cast is necessary. It is an explicit type conversion.

**Syntax**

(target_type) value target type.

It is a desired type to convert the given value. While casting the types, it is necessary to prevent the narrowing conversion which causes data lost. If the long value is converted to short then there may be a chance to loss of data. Similarly, when a floating-point value 3.25 is casted to an int value, then the value will be converted to 3 and remaining data 0.25 will be lost. It is better to avoid narrowing conversions to prevent loss of data.

**Example Program**

```
import java.io.*;
import java.util.*;
class CastingExample
{
public static void main(String[ ] args)
{
char c;
int x;
double a, b;
byte p;
a = 8.0;
b = 9.0;
x = (int) (a + b);
System.out.println(“The integer value of (a +
```

```
b) is" + x);

x = 80;
p = (byte) x;
//Byte can store 80. So no data loss.
System.out.println("The byte value is" + p);
p = 97;
//It is the ASCII code of a
c = (char) b;
System.out.println("The char value is:" + c);
//The ASCII code of 97 is 'a'. It can be
//assigned to c.
}
}
```

**Differences between Type Casting and Type Conversion**

| S.No | Type Casting | Type Conversion |
|---|---|---|
| 1. | Type casting is an explicit conversion of a value of one type into another type. | Type conversion is the process of converting the data of one type into another type. |
| 2. | It can be used for two incompatible data types. | It can be used for two compatible data types. |
| 3. | It uses parenthesis ( ) operator for casting. | It does not use any operator for conversion. |
| 4. | The size of source data type is more than the destina-tion data type. | The size of source data type is less than the destination data type. |
| 5. | It is a narrow conversion type. | It is a wide conversion type. |
| 6. | **Example**<br>int p;<br>byte q;<br>q = (byte)p; | **Example**<br>int x = 3;<br>float y;<br>y = x; |

**Introduction to Arrays**

An Array is a collection of elements that share the same type and name. The elements from the array can be accessed by the index. To create an array, we must first create the array variable of the desired type. The general form of the One Dimensional array is as follows

***type var_name[];***

Here type declares the base type of the array. This base type determine what type of elements that the array will hold.

**Example:**

**int month_days[];**

Here type is int, the variable name is month_days. All the elements in the month are integers.Since, the base type is int.

In fact, the value of ***month_days*** is set to ***null***, which represents an array with no value. To link ***month_days*** with an actual, physical array of integers, you must allocateone using ***new*** and assign it to ***month_days.*** ***new*** is a special operator that allocates memory.

**There are two types of arrays –**

- One dimensional arrays
- Two dimensional arrays

**One Dimensional Array**

Array is a collection of similar type of elements. Thus grouping of similar kind of elements is possible using arrays.

- Typically arrays are written along with the size of them.

**The syntax of declaring array is –**

***data_type array_name[ ];***

and to allocate the memory –

***array_name=new data_type[size];***

where array_name represent name of the array, ***new*** is a keyword used to allocate the memory for arrays, data_type specifies the data type of array elements and size represent the size of an array.

**For example:**

a=new int[10];

After this declaration the array a will be created as follows

**Array a[10]**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- That means after the above mentioned declaration of array, all the elements of array will get initialized to zero. Note that, always, while declaring the array in Java, we make use of the keyword ***new***, and thus we actually make use of dynamic memory allocation.
- Therefore arrays are ***allocated dynamically*** in Java. Let us discuss on simple program based on arrays.

**Java Program [SampleArray.Java]**

```
/* This is a Java program which makes use of arrays */

class SampleArray

{

public static void main(String args[])

{

int a[];

a=new int[10];

System.out.println("\tStoring the numbers in array");

a[0]=1;

a[1]=2;

a[2]=3;

a[3]=4;
```

```
a[4]=5;
a[5]=6;
a[6]=7;
a[7]=8;
a[8]=9;
a[9]=10;
System.out.println("The element at a[5] is: "+a[5]);
}}
```

**Output**

```
Storing the numbers in array
The element at a[5] is : 6
```

**Program explanation**

In above program we have created an array of 10 elements and stored some numbers in that array using the array index. The array that we have created in above program virtually should look like this –

| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

The **System.out.println** statement is for printing the value present at a[5]. We can modify the above program a little bit and i.e instead of writing

```
int a[];
a=new int[10];
```

these two separate statements we can combine them together and write it as -

```
int a[]=new int[10];
```

That means the declaration and initialization is done simultaneously.

Another way of initialization of array is

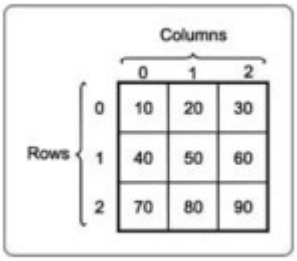
```
int a[ ]= {1,2,3,4,5,6,7,8,9,10};
```

That means, as many number of elements that are present in the curly brackets, that will be the size of an array. In other words there are total 10 elements in the array and hence size of array will be 10.

**Two Dimensional Arrays**

The two dimensional arrays are the arrays in which elements are stored in rows as well as in columns.

**For example**




The two dimensional array can be declared and initialized as follows

**Syntax**

***data_type array_name=new data_type[size];***

**For example**

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

Let us straight away write a Java program that is using two dimensional arrays –

**Java Program**

```
/* This is a Java program which makes use of 2D arrays */

class Sample2DArray

{

public static void main(String args[])

{

int a[][]=new int[3][3];

int k=0;
```

```
System.out.println("\tStoring the numbers in array");
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{
a[i][j]=k+10;
k=k+10;
}}
System.out.println("You have stored...");
for(int i=0;i<3;i++)
{
for(int j=0;j<3;j++)
{    System.out.print(" "+a[i][j]);     }
System.out.println();
}}}
```

**Output**

Storing the numbers in array

You have stored...

10 20 30

40 50 60

70 80 90

The typical application of two dimensional arrays is performing matrix operations. Let have some simple Java program in which various matrix operations are performed.

**Operators**

An operator performs an operation on one or more operands. Java provides a rich operator environment. An operator that performs an operation on one operand is called unary operator. An operator that performs an operation on two operands is called binary operator.

Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations. In Java under Binary operator we have Arithmetic, relational, Shift, bitwise and assignment operators. And under Unary operators we have ++, - - , !( Boolean not), ~(bitwise not) operators.

| Type | Operator | Meaning | Example |
|---|---|---|---|
| Arithmetic | + | Addition or unary plus | c=a+b |
| | – | Subtraction or unary minus | d= –a |
| | * | Multiplication | c=a*b |
| | / | Division | c=a/b |
| | % | Mod | c=a%b |
| Relational | < | Less than | a<4 |
| | > | Greater than | b>10 |
| | <= | Less than equal to | b<=10 |
| | >= | Greater than equal to | a>=5 |
| | == | Equal to | x==100 |
| | != | Not equal to | m!=8 |
| Logical | && | And operator | 0&&1 |
| | \|\| | Or operator | 0 \|\|1 |
| Assignment | = | is assigned to | a=5 |
| Increment | ++ | Increment by one | ++i or i++ |
| Decrement | – – | Decrement by one | – – k or k– – |

**Arithmetic Operators**

The arithmetic operators are used to perform basic arithmetic operations. The operands used for these operators must be of numeric type. The Boolean type operands cannot be used with arithmetic operators.

**Java Program**

```
//Program for demonstrating arithmetic operators
class ArithOperDemo
{
public static void main(String[] args)
{
System.out.println("\n Performing arithmetic operations ");
int a=10,b=20,c;
System.out.println("a= "+a);
System.out.println("b= "+b);
c=a+b;
System.out.println("\n Addition of two numbers is "+c);
c=a-b;
System.out.println("\n Subtraction of two numbers is "+c);
c=a*b;
System.out.println("\n Multiplication of two numbers is "+c);
c=a/b;
System.out.println("\n Division of two numbers is "+c);
}}
```

**Output**

```
Performing arithmetic operations
a= 10
b= 20
Addition of two numbers is 30
Subtraction of two numbers is -10
Multiplication of two numbers is 200
Division of two numbers is 0
```

**Relational Operators**

The relational operators typically used to denote some condition. These operators establish the relation among the operators. The <,><=,>= are the relational operators. The result of these operators is a Boolean value.

**Java Program**

```
//Program for demonstrating relational operators
import java.io.*;
import java.lang.*;
import java.util.*;
public class RelOper
{
public static void main(String args[])
{
int a,b,c;
a=10;
b=20;
if(a>b)
{
System.out.println("a is largest");
}
else
{
System.out.println("b is largest");
}}}
```

**Logical Operators**

The logical operators are used to combine two operators. These two operands are Boolean operators.

**Java Program**

```
//Program for demonstrating logical operators
import java.io.*;
import java.lang.*;
```

```
public class LogicalOper
{
public static void main(String args[])throws IOException
{
boolean oper1,oper2;
oper1=true;
oper2=false;
boolean ans1,ans2;
ans1=oper1&oper2;
ans2=oper1|oper2;
System.out.println("The oper1 is: "+oper1);
System.out.println("The oper2 is: "+oper2);
System.out.println("The oper1&oper2 is: "+ans1);
System.out.println("The oper1|oper2 is: "+ans2);
}
}
```

**Output**

```
The oper1 is: true
The oper2 is: false
The oper1&oper2 is: false
The oper1|oper2 is: true
```

**Special Operators**

- There are two special operators used in Java-instance of and dot operators.
- For determining whether the object belongs to particular class or not-an instance of operator is used.

**For example-**

- Ganga instance of River if the object Ganga is an object of class River then it returns true otherwise false.
- The dot operator is used to access the instance variable and methods of class objects.

**For example -**

Customer.name //for accessing the name of the customer

Customer.ID //for accessing the ID of the customer

**Conditional Operator**

The conditional operator is "?"

The syntax of conditional operator is

***Condition?expression1:expression2***

Where expression1 denotes the true condition and expression2 denotes false condition.

**For example :**

**a>b?true:false**

This means that if a is greater than b then the expression will return the value true otherwise it will return false.

**Increment and Decrement**

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.

For example, this statement:

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--;

**Note:** If we write increment/decrement operator after the operand such expression is called post increment decrement expression, if written before operand such expression is called pre increment/decrement expression

**The following program demonstrates the increment and decrement operator**

```
// Demonstrate ++ and --
class IncDec
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c;
int d;
c = ++b; //pre increment
d = a--; //post decrement
c++; //post increment
d--; //post decrement
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

**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.

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

These operators are again classified into two categories: Logical operators, and Shift operators.

**The Bitwise Logical Operators**

The bitwise logical 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 |

**The Bitwise NOT**

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010

becomes

11010101

after the NOT operator is applied.

**The Bitwise AND**

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

00101010 42

& 00001111 15

00001010 10

**The Bitwise OR**

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

00101010 42

| 00001111 15

00101111 47

**The Bitwise XOR**

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

Otherwise, the result is zero. The following example shows the effect of the ^.

00101010 (42)

^ 00001111 (15)

00100101 (37)

**Using the Bitwise Logical Operators**

The following program demonstrates the bitwise logical operators:

```
// Demonstrate the bitwise logical operators.
class BitLogic
{
public static void main(String args[])
{
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b); int
g = ~a & 0x0f;
System.out.println(" a|b = " +c);
System.out.println(" a&b = " +d);
System.out.println(" a^b = " +e);
System.out.println("~a&b|a&~b = " +f);
System.out.println(" ~a = " + g);
}}
```

**Bitwise Shift Operators: (left shift and right shift)**

**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 left-shift 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.

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

```
class ShiftBits
{
ShiftBits.java
public static void main(String args[])
{
byte b=6;
int c,d;
//left shift
c=b<<2;
//right shift
d=b>>3;
System.out.println("The left shift result is :"+c);
System.out.println("The right shift result is :"+d);
}
}
```

**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. The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

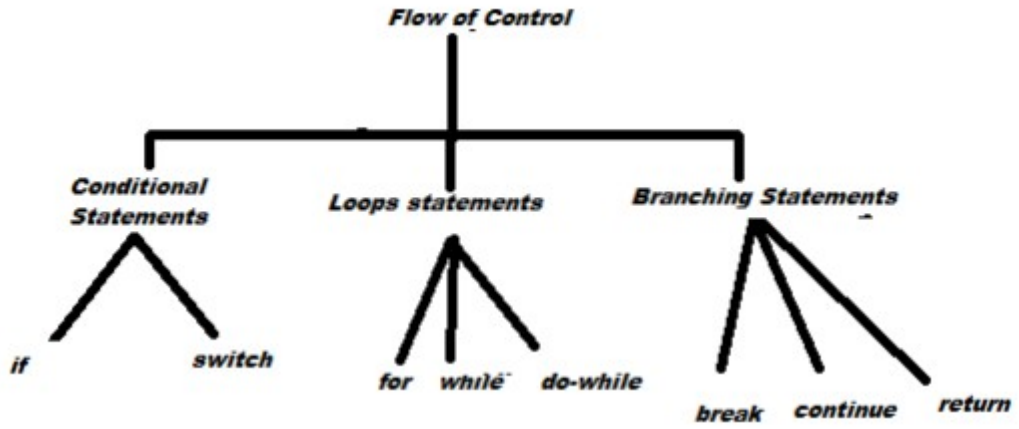
**Control Statements**

The control statements are used to control the flow of execution and branch based on the status of a program. The control statements in Java are categorized into 3 categories:

i. Conditional Statement (Selection Statements/Decision Making Statements)

ii. Loops (Iteration Statements)

iii.




Branching Statements (Jump Statements)

**Conditional Statement (Selection Statements/Decision Making Statements):**

These include if and switch. These statements allow the program to choose different paths of execution based on the outcome of the conditional expression.

***1. if statement***

The if statement is of two types

i) ***Simple if statement :*** The if statement in which only one statement is followed by that is statement.

**Syntax**

if(apply some condition)

statement

**For example**

if(a>b)

System.out.println("a is Biiiiig!");

ii) ***Compound if statement :*** If there are more than one statement that can be executed when if condition is true. Then it is called compound if statement. All these executable statements are placed in curly brackets.

**Syntax**

if(apply some condition)

{

statement 1

statement 2

.

.

.

statement n }

***2. if...else statement***

**The syntax for if...else statement will be -**

if(condition)

statement

else

statement

***For example***

```
if(a>b)
System.out.println("a is big")
else
System.out.println("b :big brother")
```

The if...else statement can be of compound type even. For example

```
if(raining==true)
{
System.out.println("I won't go out");
System.out.println("I will watch T.V. Serial");
System.out.println("Also will enjoy coffee");
} else {
System.out.println("I will go out");
System.out.println("And will meet my friend");
System.out.println("we will go for a movie");
System.out.println("Any how I will enjoy my life");
}
```

**if...else if statement**
**The syntax of if...else if statement is**
**if(is condition true?)**
**statement**
**else if(another condition)**
**statement**
**else if(another condition)**
**statement**
**else**
**statement**

**For example**

```
if(age==1)
```

```
System.out.println(“You are an infant”);
else if(age==10)
System.out.println(“You are a kid”);
else if(age==20)
System.out.println(“You are grown up now”);
else
System.out.println(“You are an adult”);
```

Let us now see Java programs using this type of control statement-

**The Switch statement**

The switch statement is Java’s multi-way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. 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
 }
```

```
import java.io.*;
class SwitchTest
{
SwitchTest.java
public static void main(String args[]) throws IOException
{char ch;
ch=(char)System.in.read();
switch(ch)
{
//test for small letters
case 'a': System.out.println("vowel");
break;
case 'e': System.out.println("vowel");
break;
case 'i': System.out.println("vowel");
break;
case 'o': System.out.println("vowel");
break;
case 'u': System.out.println("vowel");
break;
//test for capital letters
case 'A': System.out.println("vowel");
break;
default: System.out.println("Consonant");
}
}
}
```

**Loops (Iteration Statements)**

**while statement**

This is another form of while statement which is used to have iteration of the statement for the any number of times.

**The syntax is**

```
while(condition)
{
statement 1;
statement 2;
statement 3;
…
statement n;
}
```

***For example***

```
int count=1;
while(count<=5)
{
System.out.println("I am on line number "+count);
count++;
}
```

Let us see a simple Java program which makes the use of while construct.

**Java Program [whiledemo.java]**

```
/*
This is java program which illustrates
while statement
*/
class whiledemo
{
public static void main(String args[])
{
int count=1,i=0;
while(count<=5)
```

```
{
i=i+1;
System.out.println("The value of i= "+i);
count++;
}}}
```

**Output**

```
The value of i= 1
The value of i= 2
The value of i= 3
The value of i= 4
The value of i= 5
```

**do... while statement**

- This is similar to while statement but the only difference between the two is that in case of do...while statement the statements inside the do...while must be executed at least once.
- This means that the statement inside the do...while body gets executed first and then the while condition is checked for next execution of the statement, whereas in the while statement first of all the condition given in the while is checked first and then the statements inside the while body get executed when the condition is true.

**Syntax**

```
do
{
statement 1;
statement 2;
…
statement n;
}while(condition);
```

**For example**

```
int count=1;
do
{
System.out.println(“I am on the first line of do-while”);
System.out.println(“I am on the second line of do-while”);
System.out.println(“I am on the third line of do-while”);
System.out.println(“I am on the forth line of do-while”);
System.out.println(“I am on the fifth line of do-while”);
count++;
}while(count<=5);
```

**Java Program**

```
/* This is java program which illustrates do...while statement */
class dowhiledemo
{
public static void main(String args[])
{
int count=1,i=0;
do
{
i=i+1;
System.out.println("The value of i= "+i);
count++;
}while(count<=5);
}}
```

**Output**

```
The value of i= 1
The value of i= 2
The value of i= 3
The value of i= 4
The value of i= 5
```

**switch statement**

You can compare the switch case statement with a Menu-Card in the hotel. You have to select the menu then only the order will be served to you.

Here is a sample program which makes use of switch case statement -

**Java Program**

```
/* This is a sample java program for explaining Use of switch case */
class switchcasedemo
{
public static void main(String args[])
throws java.io.IOException
{
char choice;
System.out.println("\tProgram for switch case demo");
System.out.println("Main Menu");
System.out.println("1. A");
System.out.println("2. B");
System.out.println("3. C");
System.out.println("4. None of the above");
System.out.println("Enter your choice");
choice=(char)System.in.read();
switch(choice)
{
case '1':System.out.println("You have selected A");
break;
case '2':System.out.println("You have selected B");
break;
case '3':System.out.println("You have selected C");
break;
default:System.out.println("None of the above choices made");
}}}
```

**Output**

Program for switch case demo

Main Menu

1. A

2. B

3. C

4. None of the above

Enter your choice

2

You have selected B

**Note** that in above program we have written main() in somewhat different manner as - public static void main(String args[])

throws java.io.IOException

This is IOException which must be thrown for the statement System.in.read(). Just be patient, we will discuss the concept of Exception shortly! The System.in.read() is required for reading the input from the console[note that it is parallel to scanf statement in C]. Thus using System.in.read() user can enter his choice. Also note that whenever System.in.read() is used it is necessary to write the main()with IOException in order to handle the input/output error.

**for loop**

for is a keyword used to apply loops in the program. Like other control statements for loop can be categorized in simple for loop and compound for loop.

**Simple for loop :**

for (statement 1;statement 2;statement 3)
execute this statement;
Compound for loop :
for(statement 1;statement 2; statement 3)
{
execute this statement;
execute this statement;
execute this statement;

that's all;
}

Here
Statement 1 is always for initialization of conditional variables,
Statement 2 is always for terminating condition of the for loop,
Statement 3 is for representing the stepping for the next condition.

**For example :**

```
for(int i=1;i<=5;i++)
{
System.out.println("Java is an interesting language");
System.out.println("Java is a wonderful language");
System.out.println("And simplicity is its beauty");
}
```

Let us see a simple Java program which makes use of for loop.

**Java Program**

```
/* This program shows the use of for loop */
class forloop
{
public static void main(String args[])
{
for(int i=0;i<=5;i++)
System.out.println("The value of i: "+i);
}}
```

**Output**

```
The value of i: 0
The value of i: 1
The value of i: 2
The value of i: 3
The value of i: 4
The value of i: 5
```

**Access Specifiers**

Access modifiers control access to data fields, methods, and classes. There are three modifiers used in Java –

- **Public**
- **Private**
- **protected**
- **default modifier**

**public** allows classes, methods and data fields accessible from any class

**private** allows classes, methods and data fields accessible only from within the own class.

If public or private is not used then by default the classes, methods, data fields are assessable by any class in the same package. This is called package-private or package-access. A package is essentially grouping of classes.

**Protected** mode is another access specifier which is used in inheritance. The protected mode allows accessing the members to all the classes and subclasses in the same package as well as to the subclasses in other package. But the non subclasses in other package can not access the protected members.

The effect of access specifiers for class, subclass or package is enlisted below

| Specifier | class | subclass | package |
|---|---|---|---|
| private | Yes | - | - |
| protected | Yes | Yes | Yes |
| public | Yes | Yes | Yes |

**CONSTRUCTORS**

- Constructors are special member functions whose task is to initialize the objects of its class.
- It is a special member function, it has the same as the class name.
- Java constructors are invoked when their objects are created. It is named such because, it constructs the value, that is provides data for the object and are used to initialize objects.
- Every class has a constructor when we don't explicitly declare a constructor for any java class the compiler creates a default constructor for that class which does not have any return type.
- The constructor in Java cannot be abstract, static, final or synchronized and these modifiers are not allowed for the constructor.

There are two types of constructors:

1. Default constructor (no-arg constructor)

2. Parameterized constructor

**Default constructor (no-arg constructor)**

A constructor having no parameter is known as default constructor and no-arg constructor.

**Example:**

/* Here, Box uses a constructor to initialize the dimensions of a box. */

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box() {

System.out.println("Constructing Box");

width = 10;

height = 10;

```
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println(&quot;Volume is &quot; + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println(&quot;Volume is &quot; + vol);
}
}
```

**Output:**

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

new Box( ) is calling the Box( ) constructor. When the constructor for a class is not explicitly defined , then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to their default values, which are zero, null, and false, for numeric types, reference types, and boolean, respectively.

**Parameterized Constructors**

A constructor which has a specific number of parameters is called parameterized constructor.

Parameterized constructor is used to provide different values to the distinct objects.

**Example:**

```
/* Here, Box uses a parameterized constructor

to initialize the dimensions of a box. */

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// compute and return volume

double volume() {

return width * height * depth;

}

}
```

```
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println(&amp;quot;Volume is &amp;quot; + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println(&amp;quot;Volume is &amp;quot; + vol);
}
}
```

**Output:**

Volume is 3000.0

Volume is 162.0

Box mybox1 = new Box(10, 20, 15);

The values 10, 20, and 15 are passed to the Box( ) constructor when new creates the object.

Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15

respectively.

Overloading Constructors

**Example:**

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
```

```
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```
class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println(&quot;Volume of mybox1 is &quot; + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println(&quot;Volume of mybox2 is &quot; + vol);
// get volume of cube
vol = mycube.volume();
System.out.println(&quot;Volume of mycube is &quot; + vol);
}
}
```

**Output:**

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

**The this Keyword**

this keyword is used to to refer to the object that invoked it. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked. this() can be used to invoke current class constructor.

**Syntax:**

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

**Example:**

```
class Student
{
int id;
String name;
student(int id, String name)
{
this.id = id;
this.name = name;
}
void display()
{
System.out.println(id+&amp;quot; &amp;quot;+name);
}
public static void main(String args[])
{
```

```
Student stud1 = new Student(01,&amp;quot;Tarun&amp;quot;);

Student stud2 = new Student(02,&amp;quot;Barun&amp;quot;);

stud1.display();

stud2.display();

}

}
```

**Output:**

```
01 Tarun

02 Barun
```

**Overloading Methods**

When two or more methods within the same class that have the same name, but their parameter declarations are different. The methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism.

There are two ways to overload the method in java

1. By changing number of arguments

2. By changing the data type

**Example:**

```
// Demonstrate method overloading.

class OverloadDemo {

void test() {

System.out.println(&amp;quot;No parameters&amp;quot;);

}

// Overload test for one integer parameter.

void test(int a) {

System.out.println(&amp;quot;a: &amp;quot; + a);
```

```
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println(&quot;a and b: &quot; + a + &quot; &quot; + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println(&quot;double a: &quot; + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println(&quot;Result of ob.test(123.25): &quot; + result);
}}
```

**Output:**

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

**Method Overriding**

When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

**Example:**

```
// Method overriding.

class A {

int i, j;

A(int a, int b) {

i = a;

j = b;

}

// display i and j

void show() {

System.out.println(&quot;i and j: &quot; + i + &quot; &quot; + j);

}

}

class B extends A {

int k;

B(int a, int b, int c) {

super(a, b);

k = c;

}

// display k – this overrides show() in A

void show() {
```

```
System.out.println(&amp;quot;k: &amp;quot; + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

**Output:**

k: 3

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A. If you wish to access the superclass version of an overridden method, you can do so by using super. For example, in this version of B, the superclass version of show( ) is invoked within the subclass' version. This allows all instance variables to be displayed

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c; }
void show() {
super.show(); // this calls A&amp;#39;s show()
System.out.println(&amp;quot;k: &amp;quot; + k);
} }
```

If you substitute this version of A into the previous program, you will see the following

**Output:**

i and j: 1 2

k: 3

Here, super.show( ) calls the superclass version of show( ).

**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.

Objects are dynamically allocated by using the new operator, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation automatically this is called garbage collection. 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. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

**Parameter passing**

Parameter passing is a technique that passes data from one function to another function. There are two parameter passing techniques,

1. Call by value.

2. Call by reference.

**Call by Value**

In this method, values of the actual arguments in the 'calling function' are copied into the corresponding parameters in the 'called function'. Hence, the modifications done to the arguments in the 'called' function will not be reflected back to the actual arguments.

**Example**

```
import java.io.*;
import java.util.*;
class A
{
void fun(int x, int y)
{
x = x + y;
y =–y;
}
}
class Demo
{
public static void main(String[ ] args)
{
A a = new A( );
int p = 2, q = 5;
a.fun(p, q);
System.out.println(“Values of p and q” + p + “ ” + q);
}}
```

**Call by Reference**

In this method, the addresses of the actual arguments are copied to the corresponding parameters in the ‘called function’. Hence, any modifications done to the formal parameters in the ‘called function’ cause the actual parameters to change.

**Example**

```
import java.io.*;
import java.util.*;
class A
{
int x, y;
fun(int p, int q)
{
x = p;
y = q;   }
void fun1(A a)
{  a.x = a.x + a.y;
a.y = –a.y;
}}
class Demo
{  public static void main(String[ ] args)
{  A a = new A(2, 5);
a.fun1(a);
System.out.println(“Values of p and q” +a.x + “ ” +a.y);
}}
```

**Recursion**

Recursion is the process or technique by which a function calls itself. A recursive function contains a statement within its body, which calls the same function. Thus, it is also called as circular definition. A recursion can be classified into direct recursion and indirect recursion. In direct recursion, the function calls itself and in the indirect recursion, a function (f1) calls another function (f2), and the called function (f2) calls the calling function (f1). When a recursive call is made, the parameters and return address gets saved on the stack. The stack gets wrapped when the control is returned back.

**Advantages**

1. Recursion solves the problem in the most general way as possible.
2. Recursive function is small, simple and more reliable than other coded versions of the program.
3. Recursion is used to solve complicated problems that have repetitive structure.
4. Certain problems can be easily be understood using recursion.

**Disadvantages**

1. Usage of recursion incurs overhead, since this technique is implemented using function calls.
2. Whenever a recursive call is made, some of the systems memory is consumed.

**Example**

```
class Fact
{
int factorial(int x)
{
int res;
if(x = = 1)
return 1;
res = factorial(x – 1)*x;
return res;
}
}
class Rec
{
public static void main(String arg[ ])
{
fact f = new fact( );
System.out.println(“Factorial of 6 = ” + f.factorial(6));
}
}
```

## String Class and String Handling Methods

### String

A sequence of characters together is called as a string. Strings in Java are class objects. They are implemented through the use of classes such as String and StringBuffer. They are reliable and predictable when compared to that of other languages.

A string can be declared int two ways as shown below,

(i) String stringName = “SEAGI ”;

(ii) String stringName = new String(“Computers”);

**Example**

```
import java.lang.*;
class Demo4
{
public static void main(String args[ ])
{
String S1;
S1 = new String(“SEAGI”);
String S2= “Computers”;
System.out.println(S1);
System.out.println(S2);
}
}
```

### String Methods

The most commonly used string methods provided by Java are discussed below,

1. **String( ):** This method is used to create a string object.

2. **length(str):** This method is used to return the length of the given string.

3. **charAt (int location_index):** This method is used to return the character present at the specified location.

4. **subString(int startindex, int endindex):** This method is used to return the substring starting at startindex and ending at endindex in a string.

5. **endswith(String end String):** This method is used to determine if the string ends with a string argument.

6. **IndexOf(String substring):**This method is used to return the index of the first occurrence of the substring in the string.

7. **toLowerCase( ):** This method is used to convert the string to lower case.

8. **toUpperCase( ):** This method is used to convert the string to uppercase.

9. **CompareTo(String string-to-match):** This method is used to compare the two given strings.

10. **equals( ):** This method is used to compare two strings for equality.

11. **RegionMatches(int Startindex, String S, int S Startindex, int nchar):** This method is used for matching a part of string with a string region that is invoked.

12.**Indexof(Character C):** This method is used to return the index of a character of the string object which is invoked.

13. **LastIndexof(character C):** This method is used to return the index of last occurrence of the character of the string object that is invoked.

14. **LastIndexOf(String S):** This method is used to return the index of the last occurrence of the string argument of the string object that is invoked.

15. **subString(int Startindex):** This method is used to extract or retrieve the string from starting to ending of the string object.

16. **startswith(String StartString):** This method is used to determine if the starting of the string is started with a string argument.

17. **String ValueOf(int is):** This method is used for converting the primitive data type integer to string.

18. **String ValueOf(float f):** This method is used for converting the primitive data type float to string.

19. **String ValueOf(long l):** This method is used for converting the primitive data type long to string.

20.**String ValueOf(double d):** This method is used for converting the primitive data type double to string.

**Example**

```
import java.io.*;
import java.util.*;
class Test
{
public static void main (String[] args)
{
String str= “Welcome to SEAGI”;
System.out.println(“String length =” + str.length());
System.out.println(“Character at 3rd position =”+ str.charAt(3));
System.out.println(“Substring” + str.substring(3));
System.out.println(“Substring =” + str.substring(2,5));
String str1= “Hello”;
String str2 = “World”;
System.out.println(“Concatenated string =” +str1.concat(str2));
String w1= “Good Morning”;
System.out.println(“Changing to lower Case” +w1.toLowerCase());
String w2 = “Good Afternoon”;
System.out.println(“Changing to UPPER Case” +w2.toUpperCase());
}
}
```

**********