UNIT - I **Introduction**

Introduction: *Introduction to Object Oriented Programming, The History and Evolution of Java,Introduction to Classes, Objects, Methods, Constructors, this keyword, Garbage Collection, Data Types,Variables, Type Conversion and Casting, Arrays, Operators, Control Statements, Method Overloading,Constructor Overloading, Parameter Passing, Recursion, String Class and String handling methods*.

1.1 Introduction to Object Oriented Programming

The 1960s gave birth to structured programming. This is the method of programming championed by languages such as C. With the advent of languages such as c, structured programming became very popular and was the main technique of the 1980's. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired result in terms of bug-free, easy-to- maintain, and reusable programs. To solve these problems, a new way to program was invented, called *object-oriented programming* (OOP).

1.1.1 Object Oriented Programming (*OOP*) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to dowith any particular language. However, not all languages are suitable to implement the OOP concepts easily.

1.1.2 PO Programming Vs OO Programming

All **computer programs consist of two elements**: **code** and **data**. Furthermore, a program can be conceptually organized around its code or around its data. That is, someprograms are written around ― *"what is happening"* and others are written around ―*"who is being affected"*. These are the two paradigms that govern how a program is constructed. The first way is called the *process- oriented* model or *procedural* language. This approach characterizes a program as a series of linear steps (i.e. code). **The process-oriented model can be thought of as code acting on data**. Procedural languages such as C employ this model to considerable success.

To manage increasing complexity, the second approach, called ***object-oriented* programming**, was designed. Object-oriented programming organizes a program around itsdata (i.e. objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

1.1.3 Difference between POP and OOP

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	Program is divided into small parts called ***functions*.**	Program is divided into parts called ***objects.***
Importance	Importance is given to functions as well as sequence of actions to be done	Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	It follows Top-down approach.	It follows Bottom-up approach.
Data Moving	Data can move freely from function to function in the system.	Objects can communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	It provides an easy way to add new data and function.
Data Access	Most function uses Global data for sharing that can be accessed freely from function to function in the system.	Data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	It does not have any proper way for hiding data so it is less secure.	It supports data hiding so provides more security.
Overloading	In POP, Overloading is not possible	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Inheritance	No such concept of inheritance in POP	No such concept of inheritance in POP
Access Specifiers	It does not have any access specifiers.	It has access specifiers named Public, Private, Protected.
Examples	C, BASIC, FORTRAN, Pascal, COBOL.	C++, JAVA, C#, Smalltalk, Action Script.

1.2 History of Java

The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describe the history of java.

1. **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991 at *Sun Microsystems*. The small team of sun engineers called **Green Team**.
2. Originally designed for small, embedded systems in electronic appliances like set-top boxes.
3. Firstly, it was called **"*Greentalk*"** by James Gosling and file extension was ***".gt"***.
4. After that, it was called ***Oak*** and was developed as a part of the Green project.
5. **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc.
6. In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

1.2.1 Why "Java" name

7. Why had they chosen *Java* name for java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.
   According to James Gosling "Java was one of the top choices along with Silk". Since java was so unique, most of the team members preferred *Java*.
8. *Java* is an *island* of *Indonesia* where first coffee was produced (called java coffee).
9. Notice that Java is just a name not an acronym.
10. Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
11. In 1995, Time magazine called Java one of the Ten Best Products of 1995.
12. JDK 1.0 released in(January 23, 1996).

1.2. 2 Java Version History

There are many java versions that have been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. Initial Java Versions 1.0 and 1.1 were released in the year 1996 for Linux, Solaris, Mac and Windows.
3. J2SE 1.2 (Commonly called as Java 2) was released in the year 1998.
4. J2SE 1.3 codename Kestrel was released in the year 2000.
5. J2SE 1.4 codename Merlin was released in the year 2002.
6. J2SE 5.0 codename ‘Tiger’ was released in the year 2004.
7. Java SE 6 codename ‘Mustang’ was released in the year 2006.
8. Java SE 7 codename ‘Dolphin’ was released in the year 2011.
9. Java SE 8 codename ‘Spider’ was released in the year 2014.
10. Java SE 9 was released in the year 2017
11. Java SE 10 and 11 were released in the year 2018
12. Java SE 12 and 13 were released in the year 2019
13. Java SE 14 and 15 were released in the year 2020
14. Java SE 16 and 17 were released in the year 2021
15. Java SE 18 is the current stable version released in September 2022

Five Goals which were taken into consideration while developing Java

1. Keep it simple, familiar and object oriented.
2. Keep it Robust and Secure.
3. Keep it architecture-neural and portable.
4. Executable with High Performance.
5. Interpreted, threaded and dynamic.

1.3 <u>Principles of OOP</u>

Class :A class is a collection of similar objects that share the same properties, methods, relationships and semantics. A class can be defined as a template/blueprint that describes the behavior/state of the object.

Encapsulation : The process binding (or wrapping) code and data together into a single unit is known as Encapsulation. For example: capsule, it is wrapped with different medicines.

Data abstraction : It a process of providing essential features without providing the background or implementation details.

For example: It is not important for the user how TV is working internally, and different components are interconnected. The essential features required to the user are how to start, how to control volume, and change channels.

In java, we use *abstract class* and *interface* to achieve abstraction.

Inheritance : It a process by which an object of one class acquires the properties and methods of another class. It supports the concept of *hierarchical classification*.

For example, the bird, ‘robin’ is a part of class ‘flying bird’ which is again a part of the class ‘bird’.

Polymorphism :*Polymorphism* is another important OOP concept. Polymorphism, a Greek term, means the ability to take more than one form. An operation may exhibit different behavior is different instances. The behavior depends upon the types of data used in the operation.

In Java, we use *method overloading* and *method overriding* to achieve polymorphism.

Object : Object is the basic run time entity in an object-oriented system. It may represent a person, a place, a bank account, a table of data, vectors, time and lists. Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

1.4 <u>Java Features / Buzz words</u>

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

1. Simple
2. Secure
3. Portable
4. Object-oriented
5. Robust
6. Multithreaded
7. Architecture-neutral
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

Simple

Java inherits all the best features from the programming languages like C, C++ and thus makes it really easy for any developer to learn with little programming experience. Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.No need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Secure

When Java programs are executed they don't instruct commands to the machine directly. Instead Java Virtual machine reads the program (***Byte code***) and convert it into the machine instructions. This way any program tries to get illegal access to the system will not be allowed by the JVM. Allowing Java programs to be executed by the JVM makes Java program fully secured under the control of the JVM.

Portable

Java programs are portable because of its ability to run the program on any platform and no dependency on the underlying hardware / operating system.

Object Oriented

Everything in Java is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

Robust

The multi-platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. Following features of Java make it Robust.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows youto write programs that do many things simultaneously.

Architecture-neutral

The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was — *write once; run anywhere, anytime, forever*. To a great extent, this goal was accomplished.

Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

Java Virtual Machine (JVM)

The key that allows Java to solve both the security and the portability problems is the byte code. The output of *Java Compiler* is not directly executable file. Rather, it contains highly optimized set of instructions. This set of instructions is called, "***byte code***". This byte code is designed to be executed by *Java Virtual Machine (JVM)*. The JVM also called as the *interpreter* for byte code.

JVM also helps to solve many problems associated with web-based programs. Translating a Java program into byte code makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given Figure 2 JVM system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java byte code. Thus, the execution of byte code by the JVM is the easiest way to create truly portable programs.

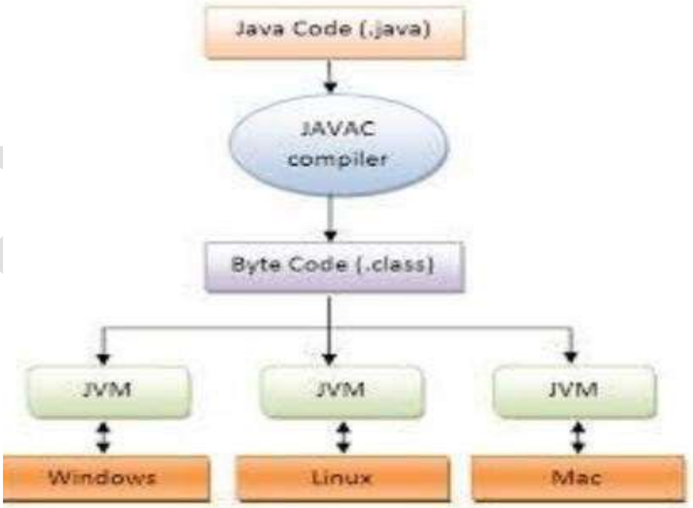


Figure 2 JVM

Internal Architecture of JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

_ JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).

- It is a specification where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.

- Its implementation is known as JRE (Java Runtime Environment).

- Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

1.5 Introduction to Classes

1.5.1 class

→A class is a combination of members (attributes) and methods(operations).

→ It is a **template** or blueprint from which objects are created.

→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.

→ A class is often defined as the blueprint or template for an object. We can create multiple objects from a class. It is a logical entity that does not occupy any space/memory.

→A class defines the shared characteristics like –

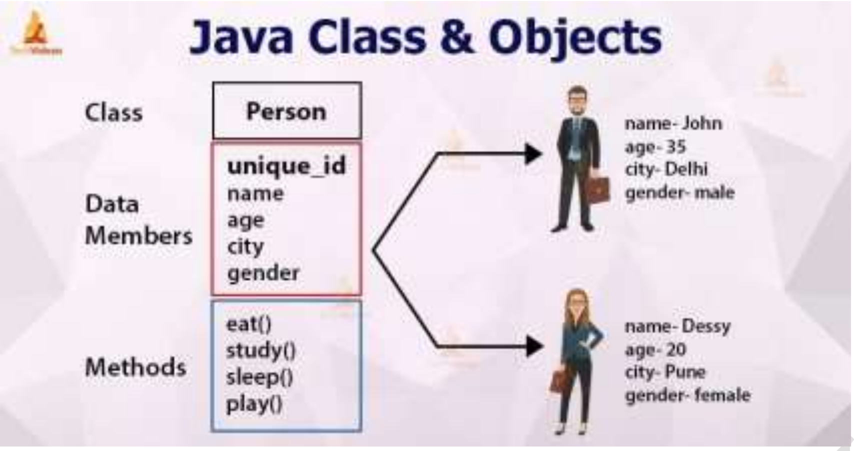
→The set of attributes/properties

→The set of behavior or methods or actions

The general form of the class

A class is declared by use of the **class** keyword. A simplified general form of a class is

```
class  classname
{       datatype variable1;
        datatype  variable2;
              ………..
        datatype variablen;
        type method1(parameterlist)
        {
            statements;
        }
        type method2(parameterlist)
        {
            statements;
        }
              ……..
        type methodN(parameterlist)
        {
            statements;
        }
}
```



Example:

```
class Human
    {
        int id;                     //class member – instance variable
        String name;                //class member – instance variable
        int age;                    //class member – instance variable
        void eat()          // method
            {
            }
        void study()     // method
            {
            }
    }
```

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

Note

→ By convention the First letter of every word of the class name starts with Capital letter, but not compulsory. For example student is written as "Bhanu".

→ The first letter of the word of the method name begins with small and remaining first letter of every word starts with Capital letter, but not compulsory For example appleIsFruit().

1.5.2 Object

An entity that has state and behavior is known as an object.
Example, chair, bike, marker, pen, table, car, etc.

An object has three characteristics:

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

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

Object Definitions:

- An object is *a real-world entity*.

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

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.

classname objectname;

Example Human prakash;   //prakash is an object of Human class.

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

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.

classname objectname = new classname();

Using the **new** keyword is the most popular way to create an object or instance of the class. When we create an instance of the class by using the new keyword, it allocates memory (heap) for the newly created **object** and also returns the **reference** of that object to that memory. The new keyword is also used to create an array. The syntax for creating an object is:

Here is a complete program that uses the **Box** class: (**BoxDemo.java)**

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
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);
}
}
```

When you compile this program, you will find that two **.class** files have been created.

→ one for **Box.class** and one for **BoxDemo.class**

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
    Box b1=new Box();
    Box b2=b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

1.6 Methods

Classes usually consist of two things: ***instance variables and methods***. This is the general form of a method:

```
returntype  methodname(parameter-list)
        {
                    body of method;
        }
```

Here,

→ ***returntype*** specifies the type of data returned by the method. This can be any valid data type, including class types that you create.

→If the method **does not return a value**, its return type must be **void**.

→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 Student class **(Student.java)**

```
    class Student
    {
            int sid;
            String sname;
            String branch;
            void display()          //no return value
            {
                    System.out.println("Student ID"+sid);
                    System.out.println("Student Name: "+sname);
                    System.out.println("Branch:"+branch);
            }
    }
```

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

/*StudentDemo.java */

```
class StudentDemo
         {
         public static void main(String args[])
          {
                  Student std = new Student();
                  s.sid=1234;
                  s.sname=”venkat”;
                  s.branch=”cse”;
                  std.display();                          // display student details
         }
    }
```

1.6.1 Returning a Value

A method can also return the value of specified type. The method after computing the task returns the value to the **caller** of the method.

```
class Student
{
         int sid;
         String sname;
         String branch;
         int sub1, sub2, sub3;
         double percentage()
              {
                  return (sub1+ sub2+sub3)/3;
              }
         void display()
             {
                  System.out.println("Student ID"+sid);
                  System.out.println("Student Name: "+sname);
                  System.out.println("Branch:"+branch);
           }
}
class StudentDemo
    {
         public static void main(String args[])
            {
                  Student std = new Student();
                  std.sid=1234;
                  std.sname=”Prakash”;
                  std.branch=”cse”;
                  std.sub1=89;
                  std.sub2=90;
                  std.sub3=91;
                  std.display();
                  float per = std.percentage() l; //calling the method(method returns double)
```

```
                System.out.println("Percentage:"+per);
        }
}
```

1.6.2 Adding a method that takes the parameters

We can also pass arguments to the method through the object. The parameters separated with comma operator. The values of the actual parameters are copied to the formal parameters in the method. The computation is carried with formal arguments, the result is returned to the caller of the method, if the type is mentioned.

```
        double  percentage(int s1, int s2, int s3)  //formal arguments
        {
                sub1 = s1;
                sub2 = s2;
                sub3 = s3;
                return (sub1+sub2+sub3)/3;
        }
```

1.7 Constructors

A constructor is a special member function used for automatic initialization of an object. Whenever an object is created, the constructor is called automatically. Constructors can be overloaded.

Characteristics

→Constructors have the same name as that of the class they belongs to.
→They automatically execute whenever an object is created.
→Constructors will not have any return type even void.
→Constructors will not return any values.
→The main function of constructor is to initialize objects and allocation of memory to the objects.
→Constructors can be overloaded.
→When the user is not defining a constructor then is called as **default constructor**, which creates by JVM.

```
        class Student
        {
                int sid;
                String sname;
                String branch;
                int sub1, sub2, sub3;
                Student()                       // This is the constructor for Student
                {
                        sid=1234;
                        sname=”Anand”;
                        branch=”IT”;
                        sub1=89;
                        sub2=90;
                        sub3=91;
                }
                double percentage()
                  {
                        return (sub1+ sub2+sub3)/3;
                }

            void display()
              {
                    System.out.println("Student ID"+sid);
                    System.out.println("Student Name: "+sname);
                    System.out.println("Branch:"+branch);
            }
      }
class StudentDemo
      {
            public static void main(String args[])
            {
               Student object Student std = new Student();
               double per = std.percentage();      //calling the method
                std.display();
                System.out.println("Percentage:"+per);
            }
      }
```

1.7.1 <u>Parameterized Constructors</u>

It may be necessary to initialize the various data members of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called ***parameterized constructors***.

```
class Student
  {
      int sid;
     String sname;
     String branch;
      int sub1,sub2,sub3;
      Student(int id, Student(int id, String name, String b, int s1 ,int s2, int s3 )
       {
            sid=id;
            sname=name;
            branch=br;
            sub1=s1;
            sub2=s2;
            sub3=s3;
       }
      public float percentage()
       {
            return (sub1+ sub2+sub3)/3;
       }
      public void display()
      {
            System.out.println("Student ID"+sid);
            System.out.println("Student Name: "+sname);
```

```
                System.out.println("Branch:"+branch);
          }
}
class StudentDemo
{
    public static void main(String args[])
          {
            // declare, allocate, and initialize Student object
            Student std = new Student(1234, "Anand","IT",89,90,91);
            float per = std.percentage();        //getting percentage
            std.display();
            System.out.println("Percentage:"+per);
          }
}
```

1.8 Method Overloading

→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(different signature).

→In this case, the methods are said to be *overloaded*, and the process is referred to as ***method overloading***.

→Method name is same but the parameters and return type is different.

→ Method overloading is one of the ways that Java supports **polymorphism**.

→When an overloaded method is invoked, Java looks for a match between arguments of the methods to determine which **version** of the overloaded method to actually call.

→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:

```
import java.util.Scanner;
class Swaping
{
        void swap(int a,int b)
        {
                int temp;
                temp=a;
                a=b;
                b=temp;
                System.out.println("Swapped integers are:"+a+" "+b);
        }
        void swap(float a,float b)
        {
                float temp;
                temp=a;
                a=b;
                b=temp;
                System.out.println("Swapped floats are:"+a+" "+b);
        }
        void swap(double a,double b)
        {
                double temp;
                temp=a;
                a=b;
                b=temp;
                System.out.println("Swapped doubles are:"+a+" "+b);
        }
        void swap(char a,char b)
        {
                char temp;
                temp=a;
                a=b;
                b=temp;
                System.out.println("Swapped integers are:"+a+" "+b);
        }
}
class SwapImplementation
{
        public static void main(String args[])
        {
                Swaping s=new Swaping();
                Scanner sc=new Scanner(System.in);
                System.out.println("Enter two integers for swapping");
                int a=sc.nextInt();
                int b=sc.nextInt();
                s.swap(a,b);
                System.out.println("Enter two floats for swapping");
                float c=sc.nextFloat();
                float d=sc.nextFloat();
                s.swap(c,d);
                System.out.println("Enter two doubles for swapping");
                double e=sc.nextDouble();
                double f=sc.nextDouble();
                s.swap(e,f);
                System.out.println("Enter two chars for swapping");
                char g=sc.next().charAt(0);
                char h=sc.next().charAt(0);
                s.swap(g,h);
        }
}
```

1.9 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.

```
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 = 0; // use 0 to indicate
            height=0; // an uninitialized
            depth =0; // 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 OverloadedConstructors {
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("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

1.10 this keyword

→Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines *this* keyword.

→**this** can be used inside any method to refer to the ***current object***. i.e. **this** is always a reference to the object on which the method was invoked.

Usage of java this keyword

→*this* can be used to refer current class instance variable.
→*this* can be used to invoke current class constructor.
→*this* can be used to invoke current class method.

this: To refer current class instance variable

→The *this* keyword can be used to refer current class instance variable. When the instance variables and the method or constructor arguments are same in name, then to refer the class in stance variables we use this keyword.

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
// 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);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
}
}
```

1.10.1 *this* can be used to call the same class constructor.

→this keyword can be used to call the same class constructor. Be sure that “this” should be the first statement in the constructor definition.

```
class Apple
{
        Apple()
        {
                System.out.println("I am zero argument constructor");
        }
        Apple(int x)
        {
                this();
                System.out.println("I am one argument constructor");
        }
        Apple(double a,int y)
        {
                this(10);
                System.out.println("I am two argument constructor");
        }
}
class Test123
{
        public static void main(String[] args)
        {
                Apple a1=new Apple(2.4,5);
        }
}
```

this can be used to invoke current class method

→this keyword can also used to call the same class’s method.

→inside the method “this” can used at anywhere and more number of times.

```
class Apple
{
        void m1()
        {
                System.out.println("I am zero argument method");
        }

        void m1(int x)
        {
                this.m1();
                System.out.println("I am one argument method");
                this.m1();
        }
}
```

```
class Test123
{
        public static void main(String[] args)
        {
                Apple a1=new Apple();
                a1.m1(12);
        }
}
```

1.11 Garbage Collection

Since objects are dynamically allocated by using the **new** operator, such objects are destroyed and their memory released for later reallocation. Java handles de-allocation 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.

1.Reference Counter

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.

2.Mark and Sweep.

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.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- **By nulling the reference**
- **By assigning a reference to another**
- **By anonymous object etc.**

By nulling a 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 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 can be used to perform cleanup processing.

This method is defined in Object class as:

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

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing.

The gc() is found in System and Runtime classes.

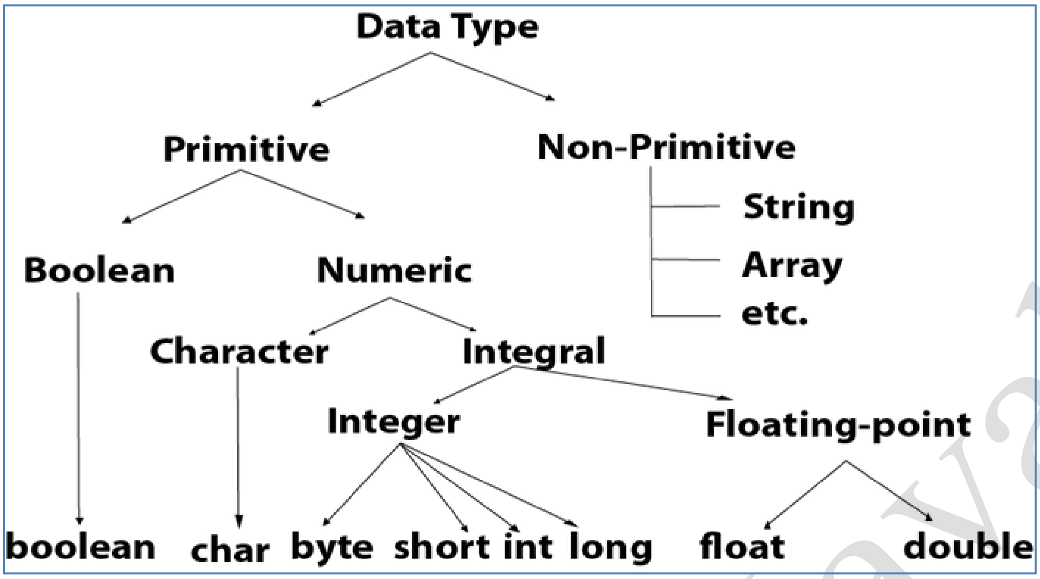
```
          public static void gc(){}
```

Simple Example of garbage collection in java

```
          public class TestGarbage1
          {
                    public void finalize()
                    {
                              System.out.println("object is garbage collected");
                    }
                    public static void main(String args[])
                    {
                               TestGarbage1 s1=new TestGarbage1();
                               TestGarbage1 s2=new TestGarbage1();
                               s1=null;
                               s2=null;
                               System.gc();
                    }
          }
```

1.12 Data Types

Java is strongly typed language. The safety and robustness of the Java language is in fact provided by its strict typing. There are two reasons for this: First, every variable and expression must be defined using any one of the type. Second, the parameters to the method also should have some type and also verified for type compatibility.



Data Type	size(in bytes)	Range	default values
byte	1	-128 to 127	0
short	2	-32768 to 32767	0
int	4	-2147483648 to 2147483647	0
long	8	–9,223,372,036,854,775,808 to 9 ,223,372,036,854,775,807	0
float	4	-3.4e38 to 3.4e	0.0
double	8	-1.7e308 to 1.7e308	0.0
char	2	0 to 6553	single space
boolean	no-size	no-range	false

```
class Dv
  {
        byte b;
        short s;
        int  i;
        long l;
        float f;
        double d;
        char c;
        boolean bl;
        void display()
             {
                System.out.println("default value of byte is "+b);
```

```
                    System.out.println("default value of short is "+s);
                    System.out.println("default value of int is "+i);
                    System.out.println("default value of long is "+l);
                    System.out.println("default value of float is "+f);
                    System.out.println("default value of double is "+d);
                    System.out.println("default value of char is "+c);
                    System.out.println("default value of boolean is "+bl);
               }
      }
class Exedv
   {
         public static void main(String[] args)
           {
                 Dv d1=new Dv();
                 d1.display();
           }
}
```

Output

```
        default value of byte is 0
        default value of short is 0
        default value of int is   0
        default value of long is 0
        default value of float is 0.0
        default value of double is 0.0
        default value of char is
        default value of boolean is   false
```

1.12.1 Variables

→A name given to an identifier, where it can varies its value is called variable.

→All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or array type or class type or enum type or interface type.

→We can create a variable ,where ever we need.

Declaration syntax

datatype   variablename;
datatype   var1,var2,var3, ...varn;

There are three types of variables in java

1. Local variables.

2. Instance variables.

3. Static variables.

1.12.1 Local variables:-

_ The variables which are declare inside a **method or constructor or blocks** those variables are called local variables.

```
        class Test
            {
                 public static void main(String[] args) //execution starts from main method
                        {
                                int a=10; //local variables
                                int  b=20;
                                System.out.println(a+"       "+b);
                        }
                }
```

→ It is possible to access local variables only inside the method or constructor or blocks only, it is not possible to access outside of method or constructor or blocks.

```
void add()
        {
                        int a=10; //local variable
                System.out.println(a); //possible
        }
void mul()
        {
                System.out.println(a); //not-possible
        }
```

→For the local variables memory allocated when method starts and memory released whenmethod completed.
→The local variables are stored in stack memory.

1.12.1.2 Instance variables (non-static variables)

→The variables which are declare inside a class but outside of methods those variables are calledinstance variables.
→The scope (permission) of instance variable is inside the class having global visibility.
→ For the instance variables memory allocated during object creation & memory released whenobject is destroyed.
→ Instance variables are stored in heap memory.

1.12.1.3 Static variables (class variables)

→ The variables which are declared inside the class but outside of the methods with staticmodifier those variables are called static variables.
→ Scope of the static variables with in the class global visibility.
→ Static variables memory allocated during .class file loading and memory released at .class fileunloading time.
→ Static variables are stored in non-heap memory.
Consider the following program which demonstrates the usage of static variable and static method.

```
class Student
        {
   int rollno;
   String name;
   static String section = "ALEXA";
    static void change()
  {       section = "NLP";       }
    Student(int r, String n)
        {
   rollno = r;
   name = n;
   }

   void display(){System.out.println(rollno+" "+name+" "+section);}
}

public class StaticMethodImplentation
        {
   public static void main(String args[])
        {
   Student.change();

   Student s1 = new Student(111,"venkat");
   Student s2 = new Student(222,"sowmya");
   Student s3 = new Student(333,"suma");
```

```
 s1.display();
 s2.display();
 s3.display();
 }
}
```

Static Block:
In simpler language whenever we use a static keyword and associate it to a block then that block is referred to as a static block. Unlike C++, Java supports a special block, called a static block (also called static clause) that can be used for static initialization of a class. This code inside the static block is executed only once: the first time the class is loaded into memory.

Calling of static block in java?

Now comes the point of how to call this static block. So in order to call any static block, there is no specified way as static block executes automatically when the class is loaded in memory. Refer to the below illustration for understanding how static block is called.

For example consider the following program which demonstrates the usage of static block:

```
class Test {

   // Case 1: Static variable
   static int i;
   // Case 2: non-static variables
   int j;

   // Case 3: Static block
   // Start of static block
   static
   {
      i = 10;
      System.out.println("static block called ");
   }
   // End of static block
}

class StaticBlock {

   public static void main(String args[])
   {

      // Although we don't have an object of Test, static
      // block is called because i is being accessed in
      // following statement.
      System.out.println(Test.i);
   }
}
```

Areas of java language:-

There are two types areas in java.

1) Instance Area. 2) Static Area.

```
Instance Area:-                                          Static Area:-
void m1()        //instance method              static void m1()        //static method
{                                                          {
        logics here    //instance area                              Logics here    //static area
}                                                          }
```

1.13 Type Conversion and Casting

Type casting is a way to convert a variable from one data type to another data type. It can beof two types: They are

1. Implicit Conversion
2. Explicit Conversion.

1.13.1 Implicit Conversion

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as *implicit type* conversion or *automatic type promotion*. In this, all the lower data types are converted to its next higher data type.

In the case of Java, An automatic type conversion will take place if the following twoconditions are met:

→The two types are compatible.
→The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place.
For example, the *int* type is always large enough to hold all valid *byte* values,

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions. They are as

- follows:First, all byte, short, and char values are promoted to int, as just
- described.
- Then, if one operand is a long, the whole expression is promoted to
- long.If one operand is a float, the entire expression is promoted to float.

If any of the operands is double, the result is double.

Example (TypePromo.java)

```
class TypePromo
```

```
  {
        public static void main(String args[])
           {
                  int num=10;
                  float sum,f
                  =10; char
                  ch='A';   //converted to ASCII (A=65)
                  sum=num+ch
                  +f;
                  System.out.println("The value of sum = "+sum);
            }
   }
```

O/P:- 85 i.e., 10+65+10

1.13.2 Explicit Conversion (Type casting)

It is intentionally performed by the programmer for his requirement in a Java program. The explicit type conversion is also known as ***type casting***. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit into the target type.

→To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

(target - type) value;

Here the target type specifies the destination type to which the value has to be converted.Example

```
int  a=1234;
byte
b=(byte) a;
```

The above code converts the *int* to *byte*. If the integer's value is larger than the range of a byte,it will be reduced modulo (the remainder of an integer division by the) byte's range. A different type of conversion will occur when a floating-point value is assigned to an integertype: *truncation*.

1.4 Arrays

→An array is a group of similar data type elements.
→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.The array indices start at zero andends at size-1.
→You can access a specific element in the array by specifying its index within square brackets.

Advantages

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.

Disadvantages

Size Limit: We can store only fixed size of elements in the array. It doesn't grow its size atruntime. To solve this problem, collection framework is used in java.

Types of Array

There are three types of array.

1. Single Dimensional Array
2. Two Dimensional Array
3. Multi-dimensional Array

1.14.1 Single Dimensional Array

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 arrayname[ ] = new type[size];

→Here *type* declares the base type of the array. This base type determine what type of elementsthat array can consist.

→*size* specifies the number of elements stored in the array.

→**new** to allocate memory for an array.

→ The elements in the array allocated by **new will automatically be initialized to zero**.

Example: int months[] = new int[12];

Here type is *int*, the array name is *months*. All the elements in the months are *integers*. Since, thebase type is *int*. The elements in the array allocated by **new will automatically be initialized to null**.

Index	0	1	2	3	4	5	6	7	8	9	10	11
Element	null	null	null	null	null	null	null	null	null	null	null	null

For example, this statement assigns the value *"February"* to the second element of **months**.months[1] =28;

Index	0	1	2	3	4	5	6	7	8	9	10	11
Element	null	28	null	null	null	null	null	null	null	null	null	null

Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

datatype arrayname[]={elemements are separated by comma};

Example

int a[]={33,3,4,5};

//Java Program to illustrate the use of declaration, instantiation

```
//and initialization of Java array in a single
        lineclass Testarray1
                {
                        public static void main(String args[])
                        {
                                int a[]={33,3,4,5};
                                for(int i=0;i<a.length;i++)
                                        System.out.println(a[i
                                        ]);
                        }
                 }
```

length: length can be used for int[], double[], String[] to know the length of the arrays.Syntax

arrayname.length

Example Program: Write a Java Program search an element in the given array

```
import java.util.Scanner;
public class Search_Element
{
   public static void main(String[] args)
   {
      int n, x, flag = 0, i = 0;
      Scanner s = new Scanner(System.in);
      System.out.print("Enter no. of elements you want in array:");
      n = s.nextInt();
      int a[] = new int[n];
      System.out.println("Enter all the elements:");
      for(i = 0; i < n; i++)
      {
         a[i] = s.nextInt();
      }
      System.out.print("Enter the element you want to find:");
      x = s.nextInt();
      for(i = 0; i < n; i++)
      {
         if(a[i] == x)
         {
            flag = 1;
```

```
                break;
            }
            else
            {
                flag = 0;
            }
        }
        if(flag == 1)
        {
            System.out.println("Element found at position:"+(i + 1));
        }
        else
        {
            System.out.println("Element not found");
        }
    }
}
```

1.14.2 Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. To declare a multidimensionalarray variable, specify each additional index using another set of square brackets.

For example, the following declares a two dimensional array variable called twod

```
int twod[][] = new int[4][4];
```

This allocates a 4 by 4 array and assigns it to twod. Internally this matrix is implemented as an *array* of *arrays* of int.

Consider the following program for matrix multiplication using two dimensional arrays.

```
class MatrixMul {

    // Function to print Matrix
    static void printMatrix(int M[][], int rowSize, int colSize)
    {
        for (int i = 0; i < rowSize; i++) {
            for (int j = 0; j < colSize; j++)
                System.out.print(M[i][j] + " ");

            System.out.println();
        }
    }

    // Function to multiply two matrices A[][] and B[][]
    static void multiplyMatrix(int row1, int col1, int A[][],    int row2, int col2, int B[][])
    {
        int i, j, k;

        // Print the matrices A and B
        System.out.println("\nMatrix A:");
        printMatrix(A, row1, col1);
        System.out.println("\nMatrix B:");
        printMatrix(B, row2, col2);

        // Check if multiplication is Possible
        if (row2 != col1) {

            System.out.println("\nMultiplication Not Possible");
            return;
        }

        // The product matrix will be of size row1 x col2
        int C[][] = new int[row1][col2];

        // Multiply the two matrices
        for (i = 0; i < row1; i++) {
            for (j = 0; j < col2; j++) {
                for (k = 0; k < row2; k++)
                    C[i][j] += A[i][k] * B[k][j];
            }
        }

        // Print the result
        System.out.println("\nResultant Matrix:");
        printMatrix(C, row1, col2);
    }

    // Driver code
    public static void main(String[] args)
    {

        int row1 = 4, col1 = 3, row2 = 3, col2 = 4;

        int A[][] =           { { 1, 1, 1 },
                                { 2, 2, 2 },
                                { 3, 3, 3 },
                                { 4, 4, 4 } };

        int B[][] =           { { 1, 1, 1, 1 },
                                { 2, 2, 2, 2 },
                                { 3, 3, 3, 3 } };

        multiplyMatrix(row1, col1, A,   row2, col2, B);
    }
}
```

Java supports allocation of different size second dimensions which is shown in the following program.

```
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

1.15.Operators

Operators are used to perform operations on variables and values.

Arithmetic Operators

+ - * / %

Comparison Operators(Relational Operators)

< > <= >= == !=

Logical Operators

&& || !

→The value of the relational expression and logical expression is **boolean type**, that is **either true or false.**

```
 class Testre
    {
        public static void main(String[] args)
          {
               int x = 5, y = 3;
               System.out.println(x == y);     // prints false because 5 is not equal to 3
         } }
```

Assignment Operators

= += -= *= /= %= <<= >>= ^= &= \=

Bitwise operators

<< >> **&** **|** ^ (exclusice-or)

Unary Operators(increment and decrement operators)

unary operators require only one operand.

Unary operators are used to perform various operations like incrementing/decrementing a value by one.

++ --

Ternary Operators

? :

```
 class Ternary
     {
               public static void main(String args[])
                 {
                    int a=2, b=5;
                    int min=(a<b)?a:b;
                    System.out.println(min);
                  }
         }
```

1.16 Control Statements

Control statements are used to control the flow of execution to advance and branch based on changes to the state of a program. Java control statements can be put into the three categories:

1. **Selection**
2. **Iteration**
3. **Jump**

1.16.1 Selection statements

They allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. They are also called as ***Conditional or Decision Making Statements***. These include *if* and *switch*.

Simple if

Syntax:

```
if(condition)
{
        statements;
}
```

if-else

Syntax:

```
if(condition)
        {
                statements;
        }
else
        {
                statements;
        }
```

Nested if

Syntax:

```
if(condition1)
        statement1;
else
         if(condition2)
                statement2;
        else
                 if(condition3)
                        statement3;
                else
                        statement4;
```

/

```
/It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
import java.util.*;
 class Ifelseexample
   {
        public static void main(String[] args)
          {
                Scanner s=new Scanner(System.in);
                int marks=s.nextInt();
                if(marks<50)
                      System.out.println("fail");
              else
                   if(marks>=50 && marks<60)
                              System.out.println("D grade");
                   else
                        if(marks>=60 && marks<70)
                              System.out.println("C grade");
                        else
                                 if(marks>=70 && marks<80)
                                      System.out.println("B grade");
                                else
                                     if(marks>=80 && marks<90){
                                           System.out.println("A grade");
                                   else
                                       if(marks>=90 && marks<100)
                                               System.out.println("A+ grade");
                                        else
                                                     System.out.println("Invalid!");
        }
}
```

switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String .

- 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*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statementis not found, it executes the next case.
- The case value can have a *default label* which is optional.

Syntax:

```
        switch(expression)
          {
                case   labelname1   :       statements;
                                             break; //optional
                case  labelname2    :       statements;
                                             break;  //optional
                ..........................................................................
                case  labelnamen    :       statements;
                                             break;  //optional
                default             :       statements;
                                             //executes when all cases are not matched
          }
```

Write a Java Program to perform arithmeticoperation of given two numbers using switch?

```
import java.util.Scanner;
class SwitchTest
  {
        public static void main(String args[])
        {
                Scanner s=new Scanner(System.in);
        I       System.out.print("Enter any two intgers");
                int a,b,ch;
                a=s.nextInt();
                b=s.nextInt();
                System.out.print("1.add\n2.sub\n3.mul\ndiv\n5/mod\n);
                System.out.print("Enter your choice");
                int ch=s.nextInt()
                switch(ch)
                    {
                        case   1 :      System.out.println("the addition is"+(a+b));
                                        break;
                        case   2 :      System.out.println("the substration  is"+(a-b));
                                        break;
                        case   3 :      System.out.println("the multiplication is"+(a*b));
                                        break;
                        case   4 :      System.out.println("the division is"+(a/b));
                                        break;
                        case   5 :      System.out.println("the addition is"+(a%b));
                                        break;
                        defaulk   :     System.out.println(" wrong option");
                }
        }
}
```

1.16.2 Loops (Iteration Statements)

→A loop repeatedly executes the same set of instructions until a termination condition is met.

→Java's iteration statements are *for*, *while*, and *do-while*.

These statements create what we commonly call loops or Repeation statements or Iterative Statemnets.

while

→The *while* loop is a *pre-test* or *entry-controlled* loop.

→It uses *conditional expression* to control the loop.

→The while loop evaluates (checking) the test expression before every iteration of the loop, so it can execute ***zero* times if the condition is initially false**.

→The initialization of a loop control variable is generally done before the loop separately.

Syntax

```
assigning;
while(condition)
        {
                statements;
                increment/decrement;
           }
```

do - while

→The *do-while* loop is a *posy-test* or *exit-controlled* loop.

→*do-while* loop is similar to *while* loop, however there is one basic difference between them. *do-while* runs at least once even if the test condition is false at first time.

Syntax of do-while loop is:

```
assigning;
do
 {
        statements;
        increment/decrement;
} while(condition);
```

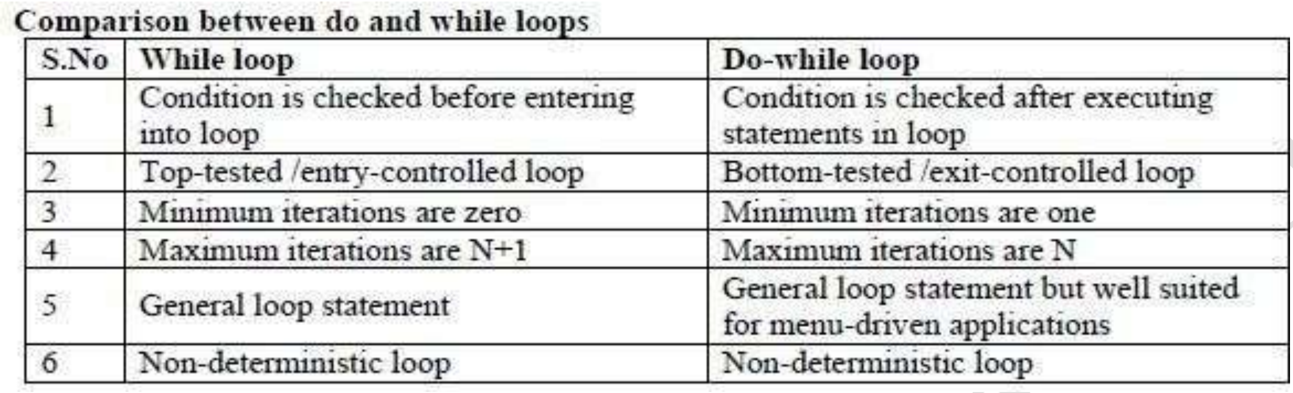
Program using while

```
Class Demowhile
{
   public static void main(String args[])
      {
         int i=1;
         while(i<=20)
            {
               System.out.println(i);
                i=i+1
            }
```

```
      }
}
```

Program using do-while

```
Class Demodowhile
{
   public static void main(String args[])
      {
         int i=1;
           do
            {
                System.out.println(i);                    }
                 i=i+1                              }
            }  while(i<=20);
```



Comparison between do and while loops

S.No	While loop	Do-while loop
1	Condition is checked before entering into loop	Condition is checked after executing statements in loop
2	Top-tested /entry-controlled loop	Bottom-tested /exit-controlled loop
3	Minimum iterations are zero	Minimum iterations are one
4	Maximum iterations are N+1	Maximum iterations are N
5	General loop statement	General loop statement but well suited for menu-driven applications
6	Non-deterministic loop	Non-deterministic loop

for statement

It is the most general looping construct in Java. The *for loop* is commonly used when the number of iterations are exactly known. The syntax of a *for* loop is:

```
for ( initialization; condition; iteration)
   {
        // body of loop

    }
```

Program using do-while

```
Class Demodowhile
{
    public static void main(String args[])
      {
          int i;
           for(i=1;i<=20;i++)
             {
                 System.out.println(i);
             }
       }
}
```

For-Each version of the for loop:

Beginning with JDK 5, a second form of for was defined that implements a ‘for-each’ style loop. The general form of the for-each version of the ‘for’ loop is shown below:

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

Here, type specifies the type and itr-var 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’.

```
// Use a for-each style for loop.
class ForEach {
```

```
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
}
System.out.println("Summation: " + sum);
}
    }
```

1.17 Parameter Passing (call-by-value, call-by-reference)

Using Objects as Parameters

→ we has been using simple types(primitive datatypeas int,float,double,char) as parameters to methods.

→The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the method. Therefore, changes made to the method have no effect on the argument.

→We can pass objects as argument to the methods.When object passing, one object carries all the member's information.

→ *call-by-reference*,In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. The changes made to the parameter will affect the argument used to call the subroutine.

Returning Objects

A method can return any type of data, including class types that you create.

Consider the following program which demonstrates the usage of call-by-reference and returning objects.

```
class Complex
{
        double real,imag;
        Complex(){}
        Complex(double r,double i)
        {
                real=r;imag=i;
        }
        Complex add(Complex x,Complex y)
        {
                Complex temp=new Complex();
                temp.real=x.real+y.real;
                temp.imag=x.imag+y.imag;
                return temp;
        }
        void display()
        {
                System.out.println(real+"+"+imag+"i");
        }
}
class ComplexImplementation
{
        public static void main(String args[])
        {
        Complex x=new Complex(10,20);
        Complex y=new Complex(10,20);
        Complex z=new Complex();
        z=z.add(x,y);

        z.display();
        }
}
```

1.18 Recursion

→Java supports *recursion*.

→calling a method by itself is calles recursion, that means, in a method definition we declaring the same method.

```
                void  prakash()                    //method defination
                        {
                                System.out.println("Teaches java");
                                prakash();      //method declaration
                         }
class Factorial
        {

                // this is a recursive method
                int fact(int n)
                        {
                                int f;
                                if(n==1)
                                        f=1;
                                else
                                        f = fact(n-1) * n;
                                return(f);
                        }
                public static void main(String args[])
                      {
                                Factorial ob = new Factorial();
                                int fa=ob.fact(5)
                                System.out.println("Factorial of 5 is " + fa);
                      }
```

1.19 String class and String handling methods:

String manipulation is arguably one of the most common activities in computer programming. This is especially true in Web systems, where Java is heavily used. In this chapter, we will look more deeply at what is certainly the most commonly used class in the language, String, along with some of its associated classes and utilities.

Java provides the String class from the java.lang package to create and manipulate strings. String has its own methods for working with strings.

String

A string is a sequence of characters. Every character in a string has a specific position in the string, and the position of the first character starts at index 0. The length of a string is the number of characters in it.

Characteristics of Strings

1. Strings are reference types, not value types, such as int or boolean. As a result, a string variable holds a reference to an object created from the String class, not the value of the string itself.
2. String class are immutable. Method in the class that appears to modify a String actually creates and returns a brand new String object containing the modification. The original String is left untouched.
3. Strings can include escape sequences that consist of a slash followed by another character. The most common escape sequences are \n for new line and \t for tab. If you want to include a slash in a string, you must use the escape sequence \\.

Strings Escape Sequences

Escape Sequence	Explanation
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage Return
\f	Form Feed
\’	Apostrophe
\”	Quotation Mark
\\	Backslash

Two ways on how to create Strings

1. Directly assigning a string literal to a String object;

   String strName = “Dayakar”

2. Using the new keyword and String constructor to create a String object.

   String games= new String (“Chess”);

String Builder class:

```
StringBuilder strFinal = new StringBuilder();
strFinal.append("Dayakar ");
strFinal.append("Chess ");
strFinal.append("Rank 1000 in Chess.com");
System.out.println(strFinal);
```

String Buffer class:

```
StringBuffer bufferedString = new StringBuffer();
bufferedString.append("The ");
bufferedString.append("Quick ");
bufferedString.append("Brown ");
bufferedString.append("Fox ");
bufferedString.append("Jumps ");
bufferedString.append("Over ");
bufferedString.append("The ");
```

```
        bufferedString.append("Lazy ");
        bufferedString.append("Dog ");
        System.out.println(bufferedString);
```

The difference between StringBuilder and StringBuffer is that StringBuffer is Synchronized also String Buffer is older than StringBuilder. StringBuilder is meant as a replacement to StringBuffer where synchronization is not necessary.

The String class Methods

The String class provides methods to perform operations on strings. Table 6.2 shows the list of some commonly used String methods in Java. These methods can only create and return a new string that contains the result of the operation.

Assume the following statement for the example of the table below

```
        String string = “Java Programming”;
```

List of the most commonly used methods in class strings

Method	Description	Example
charAt(index)	Returns the character of a string based on the specified index	string.charAt(2);//returns char ‘v’
compareTo(string)	Compares this string to another string, using alphabetical order. Returns -1 if this string comes before the other string, 0 if the strings are the same, and 1 if this string comes after the other string.	String.compareTo(“JaVa Programming”) /*returns an integer value, the operation’s result is a Positive integer 32 */ // ASCII value 0f v – ASCII value 0f V
compareToIgnoreCase(String)	Similar to compareTo but ignores case.	String.compareToIgnoreCase(“JaVa Programming”) // Returns 0 for equality
concat(string)	Returns a new string concatenated with the value of the parameter	string.concat(“ Techniques”);/*returns a new string “Java programming Techniques”*/
contains(CharSequence)	Returns true if this string contains the parameter value. parameter can be a String, StringBuilder, or StringBuffer.	System.out.println(string.contains("Java")); // Returns true – Note this is case sensitive
boolean endsWith(String)	Returns true if this string ends with the parameter string.	System.out.println(string.endsWith("ming")); // Returns true
boolean equals(String)	Returns true if this string has the same value as the parameter string.	System.out.println(string.equals("JAva Programming")); //Returns false – case sensitive

boolean equalsIgnoreCase(String)	Similar to equals but ignores case.	System.out.println(string.equals("JAva Programming")); //Returns true
int indexOf(char)	Returns the index of the first occurrence of the char parameter in this string. Returns -1 if the character is not in the string.	System.out.println(string.indexOf("v")); // Returns 2 – J = 0, a = 1, v = 2
int indexOf(String)	Returns the index of the first occurrence of the String parameter in this string. Returns -1 if the string isn’t in this string	System.out.println(string.indexOf("gram")); // Returns 8
int indexOf(String, int start)	Similar to indexOf, but starts the search at the specified position in the string.	System.out.println(string.indexOf("gram", 5)); // Returns 8 – there is no much different between the previous one except that it starts at a designated position in the string.
int lastIndexOf(char)	Returns the index of the last occurrence of the char parameter in this string. Returns -1 if the character is not in the string.	System.out.println(string.lastIndexOf("a")); //returns 3
int lastIndexOf(String)	Returns the index of the last occurrence of the String parameter in this string. Returns -1 if the string is not in this string.	System.out.println(string.lastIndexOf("a"));
int lastIndexOf (String, int)	Similar to lastIndexOf, but starts the search at	System.out.println(string.lastIndexOf("Pro", 5)); //returns 5

	the specified position in the string.	
int length()	Returns the length of this string.	System.out.println(string.length()); //returns 16
String replace(char, char)	Returns a new string that is based on the original string, but with every occurrence of the first parameter replaced by the second parameter.	System.out.println(string.replace(‘a’, ‘i’)); // Returns Jivi Progrimming
String replaceAll(String old, String new)	Returns a new string that is based on the original string, but with every occurrence of the first string replaced by the second parameter. Note that the first parameter can be a regular expression.	System.out.println(string.replaceAll("Java", "C-Sharp")); // C-Sharp Programming Note : this method will replace all occurrence
String replaceFirst(String old, String new)	Returns a new string that is based on the original string, but with the first occurrence of the first string replaced by the second parameter. Note that the first parameter can be a regular expression.	System.out.println(string.replaceFirst("Prog" , "Log")); //Returns Java Logramming Note: this method will only replace a single occurrence of the string.

String[] split(String)	Splits the string into an array of strings, using the string parameter as a pattern to determine	for(String n:string.split(" "))
		System.out.println(n);
		// returns Java
		Programming

	where to split the strings. boolean	Note: the String is temporarily place in an array and access by the n string variable.
boolean startsWith(String)	Returns true if this string starts with the parameter string. boolean	System.out.println(string.startsWith("rog",)); //returns false
startsWith(String, int)	Returns true if this string contains the parameter string at the position indicated by the int parameter.	System.out.println(string.startsWith("gram", 8)); //Returns true
String substring(int)	Extracts a substring from this string, beginning at the position indicated by the int parameter and continuing to the end of the string.	System.out.println(string.substring(4)); //return “ Programming” Note to emphasize the space I used the “ “
String substring(int, int)	Extracts a substring from this string, beginning at the position indicated by the first parameter and ending at the position one character before the value of the second parameter.	System.out.println(string.substring(8,12)); //returns gram Note 12 is not the length of the string to be extracted but the position of the element or character in the string

char[] toCharArray()	Converts the string to an array of individual characters.	char[] characters = string.toCharArray();
		//returns a character array
		//{‘J’, ‘a’, ‘v’, ‘a’, ‘ ‘, ‘P’, ‘r’, ‘o’, ‘g’, ‘r’, ‘a’, ‘m’, ‘m’, ‘i’, ‘n’, ‘g’}

String toLowerCase()	Converts the string to lowercase.	System.out.println(string.toLowerCase()); // returns java programming
String toUpperCase()	Converts the string to uppercase.	System.out.println(string.toUpperCase()); // returns JAVA PROGRAMMING
String trim()	Returns a copy of the string with all leading and	System.out.println(string.trim());

	trailing white spaces removed.	// It will just remove the space in the beginning and in the end of the string. No middle spaces are remove
String valueOf(primitiveType)	Returns a string representation of any primitive type.	System.out.println(string.valueOf(6)); // returns a numeric string “6”

Program: **Write a class with overloaded method, two arguments method version will find greatest among them and three arguments method version will find smallest among them**

```
import java.util.Scanner;
class GreatestSmallest
{
    int gsm(int x, int y)
    {
            if(x>y)
               return x;
            else
                    return y;
    }
    int gsm(int x, int y, int z)
    {
            if(x<=y && x<=z)
                            return x;
            else if(y<=x && y<=z)
                            return y;
            else
                    return z;
    }
}
class GSI
{
    public static void main(String args[])
    {
            GreatestSmallest gs=new GreatestSmallest();
            Scanner sc=new Scanner(System.in);
            System.out.println("Enter any two integers to find greatest");
            int a=sc.nextInt();
            int b=sc.nextInt();
            int greatest=gs.gsm(a,b);
            System.out.println("The greatest number="+greatest);
            System.out.println("Enter one more integers");
            int c=sc.nextInt();
            int smallest=gs.gsm(a,b,c);
            System.out.println("The smallest number="+smallest);
    }
}
```

2nd time study completed
01/02/2023 09:52