UNIT – I

SYLLABUS

The History and Evolution of Java
An Overview of Java
Data Types, Variables and Arrays
Operators
Control Statements
Introducing Classes
A Closer Look at Methods and Classes

1.1 The History and Evolution of Java and

Introduction to java:

- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991.
- This language was initially called "Oak," but was renamed "Java" in 1995.
- **The Java Buzzwords**

Simple	Object-oriented	Interpreted
Secure	Robust	High performance
Portable	Multithreaded	Distributed
	Architecture-neutral	Dynamic

Simple
Java was designed to be easy for the professional programmer to learn and use effectively.

Security
Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

Portability
Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.

Object-Oriented
Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance no objects.

Robust
The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems.

Multithreaded
Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

Architecture-Neutral
Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

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).*

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. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

UNIT – I

1.2 An Overview of Java

Object-Oriented Programming:

- There are following concepts of OOPS:
    1. Data Abstractions
    2. Data Encapsulation
    3. Inheritance
    4. Polymorphism
    5. Dynamic binding
    6. Message passing

Data Abstraction

- The process of grouping the related data and hiding the unnecessary data is called data abstraction.

Data Encapsulation

- The process of grouping or wrapping or binding the data and code (functions) into a single unit.
- The data encapsulation is achieved by using class.
- The advantage of encapsulation is that data cannot access directly. It is only accessible through the member functions of the class.

Inheritance

- Inheritance is a mechanism to obtaining the one class properties into another class and it creates new class from existing class.
- The class which is giving properties is called **base class** (or) **parent class** (or) **super class**.
- The class which is taking properties is called **derived class** (or) **child class** (or) **sub class**.
- Inheritance is basically used for reducing the overall code size of the program (**code reusability**).

Polymorphism

- Polymorphism is basic and important concept of OOPS. Polymorphism, a Greek term, means the ability to take more than one form.
- Polymorphism can be achieved by **overloading** and **overriding** concepts

Dynamic binding

- Binding means to the linking of a function call to the code to be executed in response to the call.
- Dynamic binding means that the code associated with a given function call is not known until the time of the call at run time.
- It is associated with polymorphism and inheritance.

Message passing

- An object-oriented program consists of a set of objects that communicate with each other.
- It involves the following basic steps:
    1. Creating classes that define objects and their behavior,
    2. Creating objects from class definitions, and
    3. Establishing communication among objects.

- ➢ A Message for an object is a request for execution of a function (procedure), and therefore will invoke a function in the receiving object that generates the desired results.
- ➢ *Message passing* involves specifying the name of object, the name of the function (message) and the information to be sent.

Benefits of OOPS

The principal advantages of oops are

1. The principle of data hiding helps the programmer to build the secure programs.
2. Through inheritance, we can eliminate redundant code and extend the use of existing classes.
3. We can build programs from standard working modules hat communicate with one another.
4. It is possible to have multiple instances of an object to co-exist without any interfaces.
5. It is easy to partition the work in a project based on objects.
6. Object oriented system can be easily upgrade from small to large systems.
7. Dynamic binding helps to the code associated with a given function call is not known until the time of the call at run time.
8. Message passing techniques for communication between objects.

UNIT – I

Applications of OOPS:

Main application areas of OOP are

- User interface design such as windows, menu ,…
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation system etc

Differences between POP and OOP

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called**objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides 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.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET

1.3 Data Types, Variables and Arrays

The Primitive Types

The primitive types are also commonly referred to as **simple types.**

1. **Integers:**
   - Java defines four integer types: **byte, short, int, and long**.
   - All of these are signed, positive and negative values. **Java does not support unsigned, positive-only integers**.
   - The **width** of an integer type should not be thought of as the amount of storage it consumes, but rather as the **behavior** it defines for variables and expressions of that type.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

2. **Floating-Point Types:**
   - There are two kinds of floating-point types, **float** and **double**, which represent **single**- and **double-precision** numbers, respectively.
   - Floating-point numbers, also known as ***real numbers***, are used when evaluating expressions that require fractional precision.
   - For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

3. **Characters**
   - The data type used to store characters is **char.**
   - Java uses **Unicode** to represent characters.
   - Unicode defines a fully international character set that can represent all of the characters found in all human languages.
   - In Java char is a **16-bit type**. The range of a char is **0 to 65,536**. There are **no negative chars**.
   - The standard set of characters known as **ASCII still ranges from 0 to 127** as always, and the extended 8-bit character set, **ISO-Latin-1, ranges from 0 to 255**.

4. **Booleans**
   - Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, **true** or **false**.

Variables:

- A variable is name given to the particular memory location and it having a value.
- A variable is defined by the **combination of an identifier, a type, and an optional initializer**.
- Syntax

  type identifier [= value][, identifier [= value] …];
- Here, **type** is one of Java's atomic types.
  identifier is the name of the variable.
  initialize the variable by specifying an equal sign and a value and the expression must result in a value of the same (or compatible).

1.6 Introducing Classes

Class Fundamentals:

- Class is collection of data members and member function.
- Class is a user defined and reference type data type
- A class is declared by use of the class keyword.
- Syntax:

```
class   classname {
        data or data meneber
        code or member function
 }
```

- Example:

```
class   Box {
       double width;
       double height;
       double depth;
}
```

Method:

- A method is set of statements that designed to perform a specific task.

Syntax:

```
returnType      methodName(Parameter List)
{
          Body;
}
```

Object Fundaments:

- The process of allocating the memory to class at run time is called object (instance).

Syntax: ClassName object-name = new ClassName ( );

Example: Box mybox = new Box();

Here, **C*lassName*** is the name of the class that is being instantiated.
Onject-name is a reference variable of the class type being created.
new allocates memory for an object during run time.
ClassName followed by parentheses specifies the ***constructor*** for the class.

Example: demonstrate the class, object and method

```
class Box {                                          //class declaration
        double width;
        double height;
        double depth;
        void Volume()   {                            //method declaration
                vol =width height * depth;
                System.out.println("Volume is " + vol);
        }
}
class BoxDemo {
         public static void main(String args[]) {
                    Box     mybox = new        Box();  //object creation
                    mybox.width = 10;
                    mybox.height = 20;
                    mybox.depth = 15;
                   mybox.Volume();
         }
}
```

UNIT – I

Constructor:

- Constructor is a special member function which is used to **initialize the object**.

(Or)

- Constructor is a special member function which is used to **initialize the instance variables**.
- Syntax:

```
access          ClassName( )
{
        Body;
}
```

- **Characteristics:**
  - ➢ Class name and constructor name must be same.
  - ➢ Constructor does not have return type, therefore it cannot return a value.
  - ➢ Constructors are executed or invoked automatically when the objects are created.
  - ➢ Constructor should be declared in the public section.
  - ➢ They cannot be inherited, through a derived class can call the base class constructor.
  - ➢ Constructor cannot be virtual.

Example: Demonstrating the constructor

```
class Box {                                  //class declaration
        double width;
        double height;
        double depth;
        public    Box( ) {                   //constructor
               width = 5;
               height = 10;
               depth = 20;
        }
        void Volume()   {                    //method declaration
                vol =width * height * depth;
                System.out.println("Volume is " + vol);
        }
}
class BoxDemo {
         public static void main(String args[]) {
                  Box     mybox1 = new      Box( );          //object creation
                  mybox1.Volume( );
                  Box     mybox2 = new      Box( );          //object creation
                  mybox2.Volume( );
         }
}
```

Types of constructor:

1. **Default constructor:** A constructor which is created by the compiler is called default constructor.
2. **Parameter-less constructor:** A constructor that cannot take parameters is called parameter less constructor. We already seen in the above example.
3. **Parameterized constructor:** The constructor that can take the parameters is called parameterized constructors.
   - Parameterized constructor is used to initialize the data members **dynamically** (i.e., initialize the different values to data members when the object is created).

this keyword:

- this is a **reference variable** that refers to the current object.
- this() can be used to invoke current class constructor.
- this can be used to resolve the ambiguity between instance variable and local variable.
- this can be used to invoke current class method (implicitly)
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

Garbage Collection

- The Garbage Collection is used to reclaim the memory of unreferenced object autoatically.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- How can an object be unreferenced?
  - By nulling the reference
  - By assigning a reference to another
  - By anonymous object etc.

finalize() method:

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing.

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

gc() method:

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

Syntax: **public static void** gc(){ }

Example: Simple Example of garbage collection in java

```
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( );
      }
}
```

1.7 A Closer Look at Methods and Classes

static keyword:

- The **static keyword** in Java is used for memory management.
- We can apply java static keyword with **variables, methods, blocks and nested class**.
- The static keyword belongs to the class than an instance (**object**) of the class.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.

1) static variable (class variable)

1. A variable that declared with static keyword is known as a **static variable**.
2. When a variable is declared as static, then a **single copy of variable is created** and shared among all objects at class level.
3. The static variable gets memory only once in the class area at the time of class loading.
4. Syntax:

```
static   datatype      variablename;
```

2) static method (class method)

1. A method that declared with static keyword is known as a **static method**.
2. The most common example of a static method is ***main( )*** method.
3. **Restrictions:**
   - They can only directly call other **static** methods.
   - They can only directly access **static** variable.
   - They cannot refer to **this** or **super** in any way.

Syntax:

```
static    returntype      methodname(par-list)  {
          Body;
}
```

3) static block

1. static block used to initialize the static data member.
2. It is executed before the main method at the time of class loading.

Syntax:

```
static   {
         Body;
}
```

Example: Demonstrate static variables, methods, and blocks.

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

Note: the static members of outside class can be accessed through **class name**.

Introducing Nested and Inner Classes

- A class is placed into another class is called nested class.

Syntax:

```
class    OuterClassName
{
        Code;
        Data;
        class    NestedClassName
        {
                Code;
                Data;
        }
}
```

Nested classes are divided into two types:

1. **Non-static nested classes (Inner class):** A non-static class is placed into a class is called inner class.

   Example: // Demonstrate an inner class.

```
class   Outer
{
        int outer_x = 100;
        void test()
        {
              Inner inner = new Inner();
              inner.display();
        }
        class   Inner
        {
              void display()
              {
              System.out.println("display: outer_x = " + outer_x);
              }
        }
}
class InnerClassDemo
{
        public static void main(String args[])
        {
        Outer outer = new Outer();
            outer.test();
        }
}
```

2. **Static nested classes (Nested class):** A static class is placed into a class is called nested class.

   Example: // Demonstrate an inner class.

```
class OuterClass
{
      private static String msg = "GeeksForGeeks";
      public static class NestedStaticClass
         {
        public void printMessage()
    {
                System.out.println("Message from nested static class: " + msg);
        }
        }
}
class Main
{
    public static void main(String args[])      {
        OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();
        printer.printMessage();
   }
}
```

Polymorphism:

- Polymorphism is a Greek term (poly means **many** and morphs means **forms**), polymorphism as ability to take an object into multiple form.
- Polymorphism can be achieved by **method overloading**, and **method overriding**.

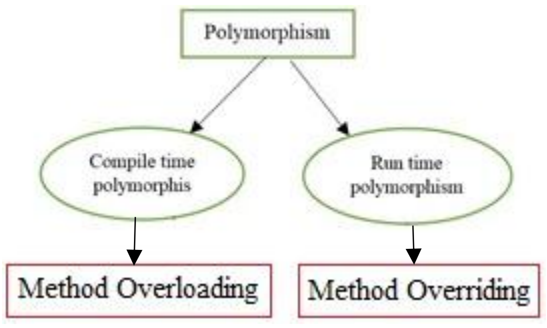

Fig: **achieving polymorphism**

Compile time polymorphism (or) early binding (or) static binding:

- The compiler link the function call to the code (or) function definition at compile time is called compile time polymorphism.
- In compile time polymorphism execution is fast.
- Compile time polymorphism can be achieved by **method overloading.**

Run time polymorphism (or) late binding (or) dynamic binding:

- The compiler link the function call to the code (or) function definition at run time is called run time polymorphism.
- In run time polymorphism execution is slow.
- Run time polymorphism can be achieved by **method overriding**.

Method overloading:

- A class contain **Two or more methods having same method name and but different in number of parameters and/or type of parameters** is called method overloading.
- The function would perform different operations depending upon the argument list in the function call.

Example program on area of different shapes:

```
class Shapes  {
      double  a;
      void Area(double r)
      {
                  a = 3.4 * r * r;
                  System.out.println("Aria of circle :"+a);
      }
      void Area(double b, double h)
      {
                  a = 0.5 * b * h;
                  System.out.println("Aria of Triangle :"+ a);
      }
              void Area(double b1, double b2, double h)
      {
                  a = h * (b1 + b2) / 2;
                  System.out.println("Aria of Trapezoidal :"+a);
      }
}
class MethodOverloading  {
      Public static void main(String args[])
            {
                  shapes s=new Shapes( );
                  s.Area(6.4);              //area of circle
                  s.Area(3.2, 6.7);         //area of Triangle
                  s.Area(4.2, 3.1, 5.0); //area of Trapezoidal
            }
}
```

Output:

```
Area of Circle : 128.614
Area of Triangle : 10.72
Area of Trapezoidal :18.25
```

Constructor overloading:

- A class contain **Two or more constructors having same name and but different in number of parameters and/or type of parameters** is called constructor overloading.
- The constructor would initialize different instance variable depending upon the argument list in the constructor call.

Example program on area of different shapes:

```
class    Student
{
            void Student()
            {
                      System.out.println("Parameter less constructor");
            }

            void Student(int x)
            {
                      System.out.println("Single parameter constructor: "+x);
            }
            void Student(double a, double b)
            {
                      System.out.println("Double parameter constructor: "+a+" and  "+b);
            }
}
class ConstructorOverloading
{
         Public static void main(String args[ ])
         {
            new Student( );
            new Student(6);
            new Student(3.2, 6.7);
          }
}
```

Output:

```
Parameter less constructor
Single parameter constructor: 6
Double parameter Constructor: 3.2 and 6.7
```

The method selection involves the following steps:

1. The compiler first tries to find an exact match, and use that function.
2. If an exact match not found, the compiler uses the integral promotions to the actual arguments, such as
   char to **int**
   float to **double**   to be match.
3. If two steps are fails, the compiler tries to use the implicit conversions to the actual arguments and then use the function whose match is unique.
   If the conversion is possible to have multiple matches, then the compiler will rise an error message.
   Example:
   ```
                   long square(long x)
                   double square(double y)
   ```
   A function such as   **square(10)**   will causes an error.
4. If all of the steps fail, then the compiler will try the explicit conversions in combination with integral promotions and implicit conversions to find a unique match.

Parameter passing mechanism:

There are two ways to pass the arguments to the method

1. Call by value
2. Call by reference

1. Call by value:

- The values of actual parameters are passed to the formal parameters and operation isdone on the formal par ameters.
- Any change in formal parameters made does not affect the actual parameters because formal parameters are just copy of actual parameters.

Example:

```
class   CallByValue
{
        void swap(int x, int y)
        {
                int temp=x;
                x = y;
                y = temp;
        }
}
class   Test
{
        public static void main(String args[])
        {
                int a = 15, b = 20;
                CallByValue ob = new CallByValue();
                System.out.println("a and b before call: " +a + " " + b);
                ob.swap(a, b);
                System.out.println("a and b after call: " +a + " " + b);
        }
}
```

2. Call by reference:

- Here, instead of passing values addresses (reference) passed. Function operates on address rather than values.
- Any change in formal parameters made does affect the actual parameters because the formal parametres are pointer to the actual patameters.

Example:

```
class   CallByValue  {
        int a, b;
        CallByValue(int x, int y)
        {
                a = x;
                b = y;
        }
        void swap(CallByValue obj)
        {
                int temp=obj.a;
                obj.a = obj.b;
                obj.b = temp;
        }
}
class   Test
{
        public static void main(String args[])
        {
                CallByValue ob = new CallByValue(10,20);
                System.out.println("a and b before call: " +ob.a + " " + ob.b);
                ob.swap(ob);
                System.out.println("a and b after call: " +ob.a + " " + ob.b);
        }
}
```

Returning object:

```
class Test {
        int a;
        Test(int i)  {
                a = i;
        }
        Test incrByTen()  {
                Test temp = new Test(a+10);
                return temp;
        }
 }
 class RetOb    {
        public static void main(String args[]) {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen();
                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);
                ob2 = ob2.incrByTen();
                System.out.println("ob2.a after second increase: "+ ob2.a);
        }
}
```

Access specifiers (or) Access modifiers

- The access modifiers specifies the accessibility or scope of a field, method, constructor, or class.
- There are 3 types of access modifiers available in Java: **public, private** and **protected**.

1. **Public:**
   - When a member of a class is modified by **public**, then that member can be accessed by any other method.
   - The data members and member functions declared public can be accessed by other classes too.
   - Hence there are chances that they might change them. So the key members must not be declared public.

2. **Private:**
   - When a member of a class is specified as **private**, then that member can only be accessed by within the class methods only.
   - If someone tries to access the private member, they will get a compile time error.

```
class A  {
     private int data=40;
     private void msg()  {
               System.out.println("Hello java");
     }
}
public class Simple {
     public static void main(String args[]) {
               A obj=new A();
               System.out.println(obj.data);  //Compile Time Error
               obj.msg();                     //Compile Time Error
      }
}
```

3. **Protected:**
   - When a member of a class is specified as **protected**, then that member can only be accessed by within the class methods and its derived classes only.

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be **Public**.

Recursion: A method that calls itself is said to be ***recursion***.

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

Command-Line Arguments

- Pass information into a program when you run it to main() is called command line arguments.
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.

```
// Display all command-line arguments.
class CommandLine
{
        public static void main(String args[])
        {
                for(int i=0; i<args.length; i++)
                        System.out.println("args[" + i + "]: " + args[i]);
        }
}
```

Varargs: Variable-Length Arguments

- A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.
- For example, a method that opens an Internet connection might take a username, password, filename, protocol, and so on, but supply defaults if some of this information is not provided.
- A variable-length argument is specified by three periods (**…**).

```
// Demonstrate variable-length arguments.
class VarArgs
{
        static void vaTest(int ... v)
        {
                System.out.print("Number of args: " + v.length +" Contents: ");
                for(int x : v)
                System.out.print(x + " ");
                System.out.println();
        }
        public static void main(String args[])
        {
        vaTest(10);             // 1 arg
        vaTest(1, 2, 3);        // 3 args
        vaTest();               // no args
        }
}
```

Output:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Overloading Vararg Methods

- You can overload a method that takes a variable-length argument.
- For example, thefollowing program overloads **vaTest( )** three times:

```
// Varargs and overloading.
class VarArgs3
{
     static void vaTest(int ... v)
     {
          System.out.print("vaTest(int ...): " +"Number of args: " + v.length +" Contents: ");
          for(int x : v)
                System.out.print(x + " ");
          System.out.println();
     }
     static void vaTest(boolean ... v)
     {
          System.out.print("vaTest(boolean ...) " +"Number of args: " + v.length +" Contents: ");
          for(boolean x : v)
                System.out.print(x + " ");
          System.out.println();
     }
     static void vaTest(String msg, int ... v)
     {
          System.out.print("vaTest(String, int ...): " +msg + v.length +" Contents: ");
          for(int x : v)
                System.out.print(x + " ");
          System.out.println();
     }
     public static void main(String args[])
     {
          vaTest(1, 2, 3);
          vaTest("Testing: ", 10, 20);
          vaTest(true, false, false);
     }
}
```

Output:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true
false false
```

Varargs and Ambiguity

- Somewhat unexpected errors can result when overloading a method that takes a variable length argument.
- These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method.

```
class VarArgs4
{
     static void vaTest(int ... v)
     {
          System.out.print("vaTest(int ...): " +"Number of args: " + v.length +" Contents: ");
          for(int x : v)
                System.out.print(x + " ");
          System.out.println();
     }
     static void vaTest(boolean ... v)
     {
          System.out.print("vaTest(boolean ...) " +"Number of args: " + v.length +" Contents: ");
          for(boolean x : v)
                System.out.print(x + " ");
          System.out.println();
     }
     public static void main(String args[])
     {
          vaTest(1, 2, 3);                // OK
          vaTest(true, false, false);     // OK
          vaTest();                       // Error: Ambiguous!
     }
}
```

UNIT – I

Self-learning exercises

Chapter I: The History and Evolution of Java

1. Define Java.
2. Discuss various Java Buzzwords.

Chapter II: An Overview of Java

3. Describe OOP principles in detail.
4. Write the benefits of OOP.
5. What are the advantages of OOP?
6. Differences between POL and OOP.

Chapter III: Data Types, Variables and Arrays

1. Explain various data types in java.
2. What is dynamic initialization? Explain with example.
3. Define scope and life time.
4. What is an array? Write about declaration, and initialization for 1-D array with example.

Chapter IV: Operators

5. Define operator precedence.
6. Define operator association.
7. Explain various operators in Java.
8. Write the syntax for ternary operator.

Chapter V: Control Statements

9. Describe about the selection statements in Java.
10. Compare and contrast entry controlled and exit controlled loop in Java
11. Differentiate break and continue.

Chapter VI: Introducing Classes

12. Define class and write its syntax.
13. Define object and write its syntax.
14. What is constructor and write its characteristics with example.
15. What is the use of **this** keyword?
16. Define garbage collection. Explain how to achieve this mechanism in java with example.

Chapter VII: A Closer Look at Methods and Classes

17. Define method overloading. Explain with example.
18. Define constructor overloading. Explain with example.
19. How many ways we can pass the parameters in java.
20. Explain about **static** keyword in java with example.
21. List the different access controls in java.
22. Write about the type casting in java with example.
23. Write about the nested class with example.
24. Write about the inner class with example.
25. What is varargs? Explain it.

Programming exercises on **Class, method, constructor, and object**

1. Define a class which consist of properties and methods for **Employee**.
2. Define a class which consist of properties and methods for **Student**.
3. Define a class which consist of properties and methods for **Vehicle**.
4. Define a class which consist of properties and methods for **Animal**.

Programming exercises on **object passing and returning.**

5. Write a java program to demonstrate the **object passing and returning** mechanism.
6. Define a class which consist of properties and methods for **Time**.
7. Define a class which consist of properties and methods for **Complex Numbers**.
8. Define a class which consist of properties and methods for **Rational numbers**.

UNIT – I

Programming exercises on **static members**

1. Define a class which consist of properties and methods for **Employee** and define static members.
2. Define a class which consist of properties and methods for **Student** and define static members.

Programming exercises on **method overloading**

1. Write java program to implement method overloading for **Area of shapes**.
2. Write java program to implement method overloading for **volume of shapes**.
3. Write java program to implement method overloading for **Arithmetic operations**.
4. Write java program to implement method overloading for **display the different types of data**.