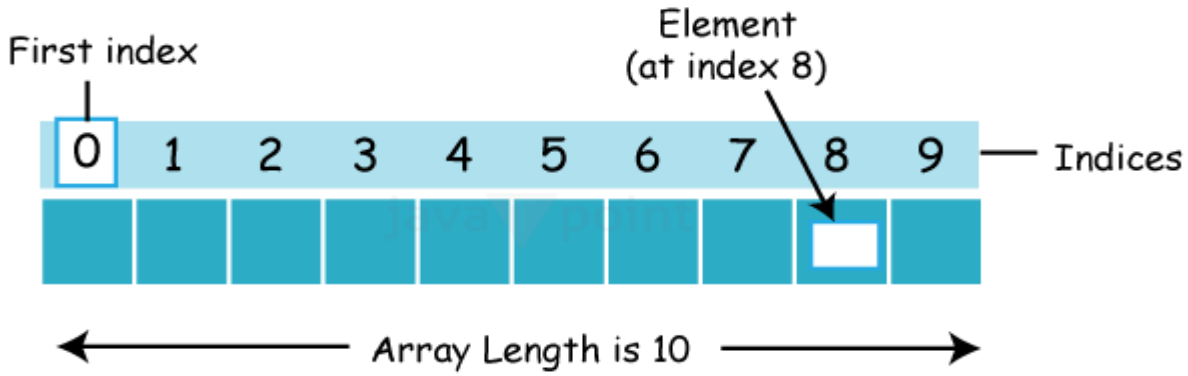
Java Array

An array is typically a grouping of elements of the same kind that are stored in a single, contiguous block of memory.

Java array is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



Advantages:

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

Disadvantages:

Size Limit: Arrays have a fixed size and do not grow dynamically at runtime.

Types of Array in java

- 1. Single Dimensional Array
- 2. Multidimensional Array

1. Single-Dimensional Array in Java

A single-dimensional array in Java is a linear collection of elements of the same data type. It is declared and instantiated using the following syntax:

```
dataType[] arr;
OR
dataType []arr;
OR
dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array
File: Testarray.java

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)   //length is the property of array
System.out.println(a[i]);
}
}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array

In Java, you can declare, instantiate, and initialize an array in a single line, as demonstrated below:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

File: Testarray1

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}
}
Output:
33
3
4
5
```

For-each Loop for Java Array

We can also print the Java array using for-each loop. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

```
for(data_type variable:array){
//body of the loop
}
```

File: Testarray1

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}
}
Output:
33
3
4
5
```

Passing Array to a Method in Java

We can pass the Java array to the method so that we can reuse the same logic on any array. When we pass an array to a method in Java, we are essentially passing a reference to the array. It means that the method will have access to the same array data as the calling code, and any modifications made to the array within the method will affect the original array.

File: Testarray2.java

```
//Java Program to demonstrate the way of passing an array  to
method.
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
  System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}
}
```

```
Output:
3
```

Explanation

This Java program demonstrates the concept of passing an array to a method. The min method takes an integer array arr as a parameter and finds the minimum element in the array using a simple iterative loop. In the main method, an integer array a is declared and initialized with values {33, 3, 4, 5}, and then the min method is called with this array as an argument. The min method iterates through the array to find the minimum element and prints it to the console.

Multidimensional Array in Java

A multidimensional array in Java is an array of arrays where each element can be an array itself. It is useful for storing data in row and column format.

Syntax to Declare Multidimensional Array in Java

```
dataType[][] arrayRefVar;
OR
dataType [][]arrayRefVar;
OR
dataType arrayRefVar[][];
OR
dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in Java

```
int[][] arr=new int[3][3];          //3 row and 3 column
```

Example to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

File: TestMultiArray.java

```
class TestMultiArray {
    public static void main(String args[]) {
      int arr[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // 3x3 matrix
        // Printing the 2D array
```

```
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
1 2 3
4 5 6
7 8 9
```

Explanation

This Java program initializes and prints a 2D array, representing a 3x3 matrix. Initially, a 2D array named arr is declared and initialized with values using array initializer syntax. The array consists of three rows, each containing three columns. The program then iterates through each row and column of the array using nested loops. Within the loops, it prints the value of each element, separated by a space. After printing all the elements of a row, a newline character is printed to move to the next line. This process continues until all elements of the array are printed. As a result, the program outputs the 3x3 matrix with each element displayed in its respective row and column.

Addition of Two Matrices in Java

A fundamental operation in linear algebra and computer science, matrix addition is frequently utilized in a variety of applications, including scientific computing, computer graphics, and image processing. To create a new matrix with the same dimensions as the original, matching elements from each matrix must be added when adding two matrices in Java. This Java program shows how to add matrices effectively by using stacked loops to do a basic example of the process.

Let's see a simple example that adds two matrices.

File: ArrayAddition.java

```
//Java Program to demonstrate the addition of two matrices in Java
class ArrayAddition{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
```

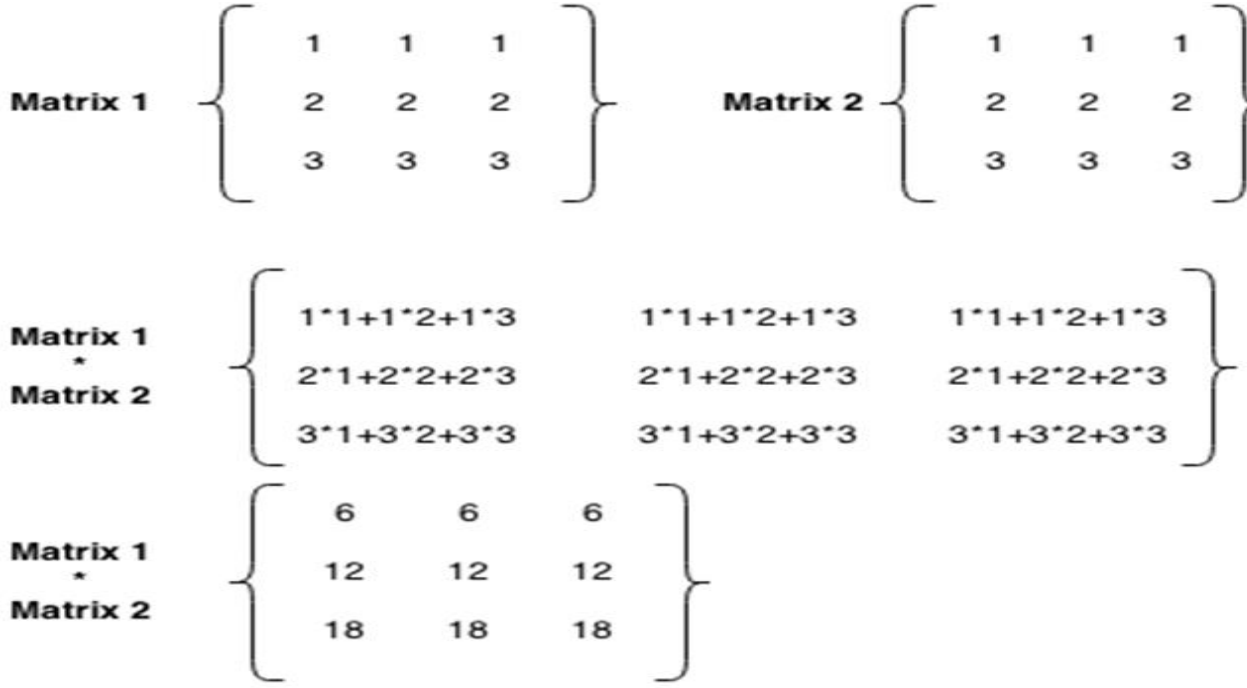
```
System.out.println();//new line
}
 }
}
Output:
2 6 8
6 8 10
```

Explanation

This Java program demonstrates matrix addition by adding two predefined matrices, a and b, element-wise and storing the result in matrix c. The matrices a and b are initialized with values, and matrix c is created to store the sum. It iterates through each element of the matrices using nested loops, adding corresponding elements from a and b and storing the result in the corresponding position in c. Finally, it prints the resulting matrix c. The output displays the element-wise addition of the two matrices, with each element in the resulting matrix c being the sum of the corresponding elements in a and b.

Multiplication of Two Matrices in Java

Matrix multiplication is a crucial operation in mathematics and computer science, often utilized in various applications such as graphics rendering, optimization algorithms, and scientific computing. In Java, multiplying two matrices involves a systematic process where each element in the resulting matrix is computed by taking the dot product of a row from the first matrix and a column from the second matrix.



File: MatrixMultiplicationExample.java

```
//Java Program to multiply two matrices
public class MatrixMultiplicationExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,1,1},{2,2,2},{3,3,3}};
```

```
int b[][]={{1,1,1},{2,2,2},{3,3,3}};

//creating another matrix to store the multiplication of two
matrices
int c[][]=new int[3][3];  //3 rows and 3 columns

//multiplying and printing multiplication of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=0;
for(int k=0;k<3;k++)
{
     c[i][j]+=a[i][k]*b[k][j];
}//end of k loop
System.out.print(c[i][j]+" ");  //printing matrix element
}//end of j loop
System.out.println();//new line
}
}
}
Output:
6 6 6
12 12 12
18 18 18
```

Explanation

This Java program computes the multiplication of two 3x3 matrices, 'a' and 'b', storing the result in matrix 'c'. It initializes matrices 'a' and 'b' with predefined values and creates matrix 'c' to store the result. Using nested loops, it iterates through each element of the resulting matrix 'c', calculating the dot product of corresponding row and column elements from matrices 'a' and 'b'.

Java String

In Java, the String class encapsulates a series of characters. Once instantiated, a String object's content is fixed and cannot be modified, attributing to its immutable nature. This immutability ensures that String objects are safe for concurrent use across threads and are optimally performant in situations where the textual content remains constant. To enhance memory efficiency, Java employs a technique called string interning. This approach optimizes the storage and access of commonly utilized string literals.

Syntax:

Direct assignment using string literals:

```
String str = "Hello, World!";
```

Direct assignment using string literals:

```
String str = "Hello, World!";
```

Using the String constructor:

```
String str = new String("Hello, World!");
```

Filename: StringExample.java

```
public class StringExample {
    public static void main(String[] args) {
        // Creating a String
        String greeting = "Hello";
        // Concatenating strings
        String sentence = greeting + ", World!";
        // Using a method from the String class
        int length = sentence.length();
        System.out.println(sentence); // Output: Hello, World!
        System.out.println("Length: " + length);
               // Output: Length: 13
    }
}
Output:
Hello, World!
Length: 13
```

StringBuffer

StringBuffer represents a mutable sequence of characters that ensures thread safety, making it suitable for scenarios involving multiple threads that modify a character sequence. It includes various string manipulation capabilities, including the ability to insert, delete, and append characters. This design avoids the necessity of generating new objects with each change, leading to enhanced efficiency in situations requiring regular adjustments to the string content.

Syntax

```
StringBuffer sb = new StringBuffer();
```

Filename: StringBufferExample.java

```
public class StringBufferExample {
    public static void main(String[] args) {
        // Creating a StringBuffer
        StringBuffer sb = new StringBuffer("Hello");
        // Appending to the StringBuffer
        sb.append(", World!");
        // Inserting into the StringBuffer
        sb.insert(5, " Java");
        // Deleting from the StringBuffer
        sb.delete(5, 10);
        System.out.println(sb); // Output: Hello, World!
    }
}
Output:
Hello, World!
```

StringBuilder

StringBuilder shares similarities with StringBuffer by being a mutable character sequence. The crucial distinction lies in StringBuilder not being synchronized, rendering it not suitable for thread-safe operations. This absence of synchronization, though, contributes to StringBuilder offering superior performance in environments that are single-threaded or confined to a specific thread. As a result, StringBuilder becomes the favored option for manipulating strings in contexts where the safety of concurrent thread access is not an issue.

Syntax:

```
StringBuilder sb = new StringBuilder();
```

Filename: StringBuilderExample.java

```
public class StringBuilderExample {
    public static void main(String[] args) {
        // Creating a StringBuilder
        StringBuilder sb = new StringBuilder("Hello");
        // Appending to the StringBuilder
        sb.append(", World!");
        // Inserting into the StringBuilder
        sb.insert(5, " Java");
        // Deleting from the StringBuilder
        sb.delete(5, 10);
        System.out.println(sb); // Output: Hello, World!
    }
}
Output:
Hello, World!
```

StringTokenizer

Java's StringTokenizer class, housed within the java.util package, simplifies the process of segmenting a string into multiple tokens, utilizing designated separators. This class is exceptionally beneficial for parsing strings and navigating through their tokens, especially in scenarios involving user inputs, file contents, or network transmissions formatted with straightforward separators such as commas, spaces, or tabs. It offers an efficient means to dissect and examine the tokenized segments of a string, catering to situations that demand basic parsing capabilities.

The StringTokenizer class implements the Enumeration<Object> interface, allowing the tokens of the string to be iterated like other enumeration types in Java. It offers a straightforward approach to tokenization, avoiding the need for more complex regular expressions, making it suitable for simple parsing needs.

Syntax:

```
public StringJoiner(CharSequence delimiter)
Filename: StringTokenizer.java
import java.util.StringJoiner;
public class StringTokenizer {
    public static void main(String[] args) {
```

```
        // Creating a StringJoiner with a comma (,) as the delimiter
        StringJoiner joiner = new StringJoiner(", ");
        // Adding strings to the StringJoiner
        joiner.add("Apple");
        joiner.add("Banana");
        joiner.add("Cherry");
        // Converting the StringJoiner to a String
        String result = joiner.toString();
        // Output the result
        System.out.println(result);  // Output: Apple, Banana, Cherry
    }
}
Output:
Apple, Banana, Cherry
```

StringBuffer class in Java

StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

Here is an example of using StringBuffer to concatenate strings:

```
// Java Program to Demonstrate
// String Buffer

public class StringBufferExample {
    public static void main(String[] args){

          // Creating StringBuffer
   StringBuffer s = new StringBuffer();

          // Adding elements in StringBuffer
        s.append("Hello");
        s.append(" ");
        s.append("world");

          // String with the StringBuffer value
          String str = s.toString();
        System.out.println(str);
    }
}
Output
Hello world
```

Constructors of StringBuffer Class

Constructor	Description	Syntax
StringBuffer()	It reserves room for 16 characters without reallocation	StringBuffer s = new StringBuffer();
StringBuffer(int size)	It accepts an integer argument that explicitly sets the size of the buffer.	StringBuffer s = new StringBuffer(20);
StringBuffer(String str)	It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.	StringBuffer s = new StringBuffer(“GeeksforGeeks”);

Methods of Java StringBuffer Class

Methods	Action Performed
append( )	Used to add text at the end of the existing text.
length( )	The length of a StringBuffer can be found by the length( ) method.
capacity( )	the total allocated capacity can be found by the capacity( ) method.
charAt( )	This method returns the char value in this sequence at the specified index.
delete( )	Deletes a sequence of characters from the invoking object.
deleteCharAt( )	Deletes the character at the index specified by the *loc.*

Methods	Action Performed
ensureCapacity( )	Ensures capacity is at least equal to the given minimum.
insert( )	Inserts text at the specified index position.
length ( )	Returns the length of the string.
reverse( )	Reverse the characters within a StringBuffer object.
replace( )	Replace one set of characters with another set inside a StringBuffer object.

Examples of Java StringBuffer Method

append( ) method

The append( ) method concatenates the given argument with this string.

Example:

```
import java.io.*;
class A {
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("Hello ");
        sb.append("Java"); // now original string is changed
        System.out.println(sb);
    }
}
Output
Hello Java
```

insert( ) method

The insert() method inserts the given string with this string at the given position.

Example:

```
import java.io.*;
class A {
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer("Hello ");
```

```
        sb.insert(1, "Java");
        // Now original string is changed
        System.out.println(sb);
    }
}
Output
HJavaello
```

replace( ) method

The replace() method replaces the given string from the specified beginIndex and endIndex-1.

Example:

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

        StringBuffer sb = new StringBuffer("Hello");
        sb.replace(1, 3, "Java");
        System.out.println(sb);
    }
}
Output
HJavalo
```

delete( ) method

The delete( ) method of the StringBuffer class deletes the string from the specified beginIndex to endIndex-1.

Example:

```
import java.io.*;
class A {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.delete(1, 3);
        System.out.println(sb);
    }
}
Output
Hlo
```

reverse( ) method :

The reverse( ) method of the StringBuilder class reverses the current string.

Example:

```
import java.io.* ;
```

```
class A {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        sb.reverse();
        System.out.println(sb);
    }
}
Output
olleH
```

capacity( ) method

The capacity( ) method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of characters increases from its current capacity, it increases the capacity by (oldcapacity*2)+2.

For instance, if your current capacity is 16, it will be (16 * 2) + 2 = 34.

Example:

```
import java.io.*;
class A {
    public static void main(String args[])
    {
        StringBuffer sb = new StringBuffer();

          // default 16
        System.out.println(sb.capacity());
        sb.append("Hello");

          // now 16
        System.out.println(sb.capacity());
        sb.append("java is my favourite language");

          // (oldcapacity*2)+2
          System.out.println(sb.capacity());
    }
}
Output
16
16
34
```

Differentiate between String and String Buffer:

String	String Buffer
String is a major class	String Buffer is a peer class of String
Length is fixed (immutable)	Length is flexible (mutable)

Contents of object cannot be modified	Contents of object can be modified
Object can be created by assigning String constants enclosed in double quotes.	Objects can be created by calling constructor of String Buffer class using “new”
Ex:- String s=”abc”;	Ex:- StringBuffer s=new StringBuffer (“abc”);

Java Vector

Vector is like the dynamic array which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are fail-fast. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-

Vector is synchronized.

Java Vector contains many legacy methods that are not the part of a collection’s framework.

Java Vector class Declaration

```
public class Vector<E>
extends Object<E>
implements List<E>, Cloneable, Serializable
```

Java Vector Constructors

Vector class supports four types of constructors. These are given below:

Constructor	Description
vector()	It constructs an empty vector with the default size as 10.
vector(int initialCapacity)	It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
vector(int initialCapacity, int capacityIncrement)	It constructs an empty vector with the specified initial capacity and capacity increment.
Vector(Collection<? extends E> c)	It constructs a vector that contains the elements of a collection c.

Java Vector Methods

The following are the list of Vector class methods:

Method	Description
add()	It is used to append the specified element in the given vector.
addAll()	It is used to append all of the elements in the specified collection to the end of this Vector.
addElement()	It is used to append the specified component to the end of this vector. It increases the vector size by one.
capacity()	It is used to get the current capacity of this vector.
clear()	It is used to delete all of the elements from this vector.
clone()	It returns a clone of this vector.
contains()	It returns true if the vector contains the specified element.
containsAll()	It returns true if the vector contains all of the elements in the specified collection.
copyInto()	It is used to copy the components of the vector into the specified array.
elementAt()	It is used to get the component at the specified index.
elements()	It returns an enumeration of the components of a vector.
ensureCapacity()	It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument.
equals()	It is used to compare the specified object with the vector for equality.
firstElement()	It is used to get the first component of the vector.
forEach()	It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
get()	It is used to get an element at the specified position in the vector.
hashCode()	It is used to get the hash code value of a vector.
indexOf()	It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
insertElementAt()	It is used to insert the specified object as a component in the given vector at the specified index.

isEmpty()	It is used to check if this vector has no components.
iterator()	It is used to get an iterator over the elements in the list in proper sequence.
lastElement()	It is used to get the last component of the vector.
lastIndexOf()	It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element.
listIterator()	It is used to get a list iterator over the elements in the list in proper sequence.
remove()	It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged.
removeAll()	It is used to delete all the elements from the vector that are present in the specified collection.
removeAllElements()	It is used to remove all elements from the vector and set the size of the vector to zero.
removeElement()	It is used to remove the first (lowest-indexed) occurrence of the argument from the vector.
removeElementAt()	It is used to delete the component at the specified index.
removeIf()	It is used to remove all of the elements of the collection that satisfy the given predicate.
removeRange()	It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive.
replaceAll()	It is used to replace each element of the list with the result of applying the operator to that element.
retainAll()	It is used to retain only that element in the vector which is contained in the specified collection.
set()	It is used to replace the element at the specified position in the vector with the specified element.
setElementAt()	It is used to set the component at the specified index of the vector to the specified object.
setSize()	It is used to set the size of the given vector.
size()	It is used to get the number of components in the given vector.
sort()	It is used to sort the list according to the order induced by the specified Comparator.

spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the list.
subList()	It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive.
toArray()	It is used to get an array containing all of the elements in this vector in correct order.
toString()	It is used to get a string representation of the vector.
trimToSize()	It is used to trim the capacity of the vector to the vector's current size.

Java Vector Example

```
import java.util.*;
public class VectorExample {
       public static void main(String args[]) {
          //Create a vector
          Vector<String> vec = new Vector<String>();
          //Adding elements using add() method of List
          vec.add("Tiger");
          vec.add("Lion");
          vec.add("Dog");
          vec.add("Elephant");
          //Adding elements using addElement() method of Vector
          vec.addElement("Rat");
          vec.addElement("Cat");
          vec.addElement("Deer");

          System.out.println("Elements are: "+vec);
       }
}
```

Output:

```
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
```

Java Vector Example 2

```
import java.util.*;
public class VectorExample1 {
  public static void main(String args[]) {
   //Create an empty vector with initial capacity 4
   Vector<String> vec = new Vector<String>(4);
   //Adding elements to a vector
   vec.add("Tiger");
   vec.add("Lion");
   vec.add("Dog");
   vec.add("Elephant");
   //Check size and capacity
```

```
   System.out.println("Size is: "+vec.size());
   System.out.println("Default capacity is: "+vec.capacity());
   //Display Vector elements
   System.out.println("Vector element is: "+vec);
   vec.addElement("Rat");
   vec.addElement("Cat");
   vec.addElement("Deer");
   //Again check size and capacity after two insertions
   System.out.println("Size after addition: "+vec.size());
   System.out.println("Capacity after addition is: "+vec.capacity());
   //Display Vector elements again
   System.out.println("Elements are: "+vec);
   //Checking if Tiger is present or not in this vector
   if(vec.contains("Tiger"))
    {
     System.out.println("Tiger    is    present    at    the    index    "
+vec.indexOf("Tiger"));
    }
    else
    {
     System.out.println("Tiger is not present in the list.");
    }
    //Get the first element
System.out.println("The first animal of the vector is =
+vec.firstElement());
    //Get the last element
   System.out.println("The last animal of the vector is =
"+vec.lastElement());
}
}
```

Output:

```
Size is: 4
Default capacity is: 4
Vector element is: [Tiger, Lion, Dog, Elephant]
Size after addition: 7
Capacity after addition is: 8
Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]
Tiger is present at the index 0
The first animal of the vector is = Tiger
The last animal of the vector is = Deer
```

Java Vector Example 3

```
import java.util.*;
public class VectorExample2 {
       public static void main(String args[]) {
        //Create an empty Vector
        Vector<Integer> in = new Vector<>();
        //Add elements in the vector
        in.add(100);
        in.add(200);
        in.add(300);
        in.add(200);
```

```
        in.add(400);
        in.add(500);
        in.add(600);
        in.add(700);
        //Display the vector elements
        System.out.println("Values in vector: " +in);
        //use remove() method to delete the first occurence of an element
        System.out.println("Remove first occourence of element 200:
"+in.remove((Integer)200));
        //Display the vector elements afre remove() method
        System.out.println("Values in vector: " +in);
        //Remove the element at index 4
        System.out.println("Remove element at index 4: " +in.remove(4));
        System.out.println("New Value list in vector: " +in);
        //Remove an element
        in.removeElementAt(5);
        //Checking vector and displays the element
        System.out.println("Vector element after removal: " +in);
        //Get the hashcode for this vector
        System.out.println("Hash code of this vector = "+in.hashCode());
        //Get the element at specified index
        System.out.println("Element at index 1 is = "+in.get(1));
          }
}
```

Output:

```
Values in vector: [100, 200, 300, 200, 400, 500, 600, 700]
Remove first occourence of element 200: true
Values in vector: [100, 300, 200, 400, 500, 600, 700]
Remove element at index 4: 500
New Value list in vector: [100, 300, 200, 400, 600, 700]
Vector element after removal: [100, 300, 200, 400, 600]
Hash code of this vector = 130123751
Element at index 1 is = 300
```

Write a program to create a vector with five elements as (5, 15, 25, 35, 45). Insert new element at 2nd position. Remove 1st and 4th element from vector

```
import java.util.*;
class VectorDemo
{
public static void main(String[] args)
{
Vector v = new Vector();
v.addElement(new Integer(5));
v.addElement(new Integer(15));
v.addElement(new Integer(25));
v.addElement(new Integer(35));
v.addElement(new Integer(45));
System.out.println("Original array elements are");
for(int i=0;i<v.size();i++)
{
     System.out.println(v.elementAt(i));
```

```
}
v.insertElementAt(new Integer(20),1); // insert new element at 2nd position
v.removeElementAt(0);
                                        //remove first element
v.removeElementAt(3);
                                        //remove fourth element
System.out.println("Array elements after insertand remove operation
");
for(int i=0;i<v.size();i++)
{
     System.out.println(v.elementAt(i));
}
}
}
```

Differentiate between array and vector.

Array	Vector
An array is a structure that holds multiple values of the same type.	The Vector is similar to array holds multiple objects and like an array; it contains components that can be accessed using an integer index.
An array is a homogeneous data type where it can hold only objects of one data type.	Vectors are heterogeneous. You can have objects of different data types inside a Vector
After creation, an array is a fixed-length structure.	The size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
Array can store primitive type data element.	Vector are store nonprimitive type data element
Declaration of an array : int arr[] = new int [10];	Declaration of Vector: Vector list = new Vector(3);
Array is the static memory allocation	Vector is the dynamic memory allocation.

Wrapper classes in Java

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.

- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes is given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

File: WrapperExample1.java

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1{
public static void main(String args[]){
                    //Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a); //converting    int    into    Integer explicitly
Integer j=a;              //autoboxing,   now   compiler   will   write Integer.valueOf(a) internally
 System.out.println(a+" "+i+" "+j);
```

```
}
}
Output:
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive
File: WrapperExample2.java

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output:

```
3 3 3
```

Java Wrapper classes Example
File: WrapperExample3.java

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;

//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
```

```
Character charobj=c;
Boolean boolobj=b2;

//Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;

//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}
}
```

Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
```

```
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

Custom Wrapper class in Java

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

File: TestWrapper.java

```
//Creating the custom wrapper class
class Wrapper{
private int i;
Javatpoint(){}
Javatpoint(int i){
this.i=i;
}
public int getValue(){
return i;
}
public void setValue(int i){
this.i=i;
}
@Override
public String toString() {
  return Integer.toString(i);
}
}
//Testing the custom wrapper class
public class TestWrapper{
public static void main(String[] args){
Wrapper j=new Wrapper(10);
System.out.println(j);
}
}
```

Output:

```
10
```

Java Constructors

In Java, a Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new( ) keyword, at least one constructor is called.

Example:

File: Main.java

```
// Java Program to demonstrate Constructor
```

```
import java.io.*;
class Main {

    // Constructor
    Main()
    {
        super();
        System.out.println("Constructor Called");
    }

    // main function
    public static void main(String[] args)
    {
        Main m = new Main();
    }
}
```

Output

```
Constructor Called
```

Note: It is not necessary to write a constructor for a class. It is because the java compiler creates a default constructor (constructor with no arguments) if your class doesn't have any.

Each time an object is created using a new() keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the data members of the same class.

Rules for writing constructors are as follows:

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

Types of Constructors in Java

- 1. Default Constructor
- 2. Parameterized Constructor
- 3. Copy Constructor

1. Default Constructor in Java

A constructor that has no parameters is known as default constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can't change the default constructor. The default constructor can be implicit or explicit.

Implicit Default Constructor: If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn't take any parameters and initializes the object with default values, such as 0 for numbers, null for objects.

Explicit Default Constructor: If we define a constructor that takes no parameters, it's called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically.

Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you.

File: DefaultCons.java

```
// Java Program to demonstrate Default Constructor
import java.io.*;
class DefaultCons
{
DefaultCons ( )
{
     System.out.println("Default constructor");
}
    public static void main(String[] args)
    {
        DefaultCons hello = new DefaultCons ( );
    }
}
```

Output

```
Default constructor
```

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

2. Parameterized Constructor in Java

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

Example:
File: Main.java

```
// Java Program for Parameterized Constructor
import java.io.*;
class Main {
    // data members of the class.
    String name;
    int id;

    Main(String name, int id) {
        this.name = name;
        this.id = id;
    }
}

class GFG
{
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        Main m = new Main("Rasheed", 121);
        System.out.println("Name :" + m.name + " and Id :" + m.id);
```

```
    }
}
```
Output
```
Name :Rasheed and Id :121
```

Now the most important topic that comes into play is the strong incorporation of OOPS with constructors known as constructor overloading. Just like methods, we can overload constructors for creating objects in different ways. The compiler differentiates constructors on the basis of the number of parameters, types of parameters, and order of the parameters.

File: Main.java
```
// Java Program to illustrate constructor overloading
// using same task (addition operation ) for different types of
arguments.

import java.io.*;

class PCons {
    // constructor with one argument
    PCons (String name)
    {
        System.out.println("Constructor with one "
                           + "argument - String : " + name);
    }

    // constructor with two arguments
    PCons (String name, int age)
    {

        System.out.println(
            "Constructor with two arguments : "
            + " String and Integer : " + name + " " + age);
    }

    // Constructor with one argument but with different
    // type than previous..
    PCons (long id)
    {
        System.out.println(
            "Constructor with one argument : "
            + "Long : " + id);
    }
}

class Main {
    public static void main(String[] args)
    {
        // Creating the objects of the class named 'Geek'
        // by passing different arguments

        // Invoke the constructor with one argument of
        // type 'String'.
```

```
        PCons P2 = new PCons ("MMANTC");

        // Invoke the constructor with two arguments
        PCons P3 = new PCons ("Computer", 1726);

        // Invoke the constructor with one argument of
        // type 'Long'.
        PCons P4 = new PCons (325614567);
    }
}
```

Output

```
Constructor with one argument - String : MMANTC
Constructor with two arguments :  String and Integer : Computer 1726
Constructor with one argument : Long : 325614567
```

3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

Note: In Java,there is no such inbuilt copy constructor available like in other programming languages such as C++, instead we can create our own copy constructor by passing the object of the same class to the other instance(object) of the class.

File: Main.java

```
// Java Program for Copy Constructor
import java.io.*;

class CopyCons {
    // data members of the class.
    String name;
    int id;

    // Parameterized Constructor
    CopyCons (String name, int id)
    {
        this.name = name;
        this.id = id;
    }

    // Copy Constructor
    CopyCons (CopyCons obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
class Main {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        System.out.println("First Object");
        CopyCons C1 = new CopyCons("MMANTC", 1726);
```

```
        System.out.println("Name:" +C1.name+ " and Id:" +C1.id);

        System.out.println("Copy Constructor used Second Object");

        // This would invoke the copy constructor.
        CopyCons C2 = new CopyCons(C1);
        System.out.println("Copy Constructor used Second Object");
        System.out.println("Name:" +C2.name +"and I:" +C2.id);
    }
}
```

Output

```
First Object
Name: MMANTC and Id :1726
Copy Constructor used Second Object
Name: MMANTC and Id :1726
```

Constructor overloading in Java

In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

Example

```
public class Student {
int id;
String name;
Student( ){
System.out.println("this a default constructor");
}
Student(int i, String n){
id = i;
name = n;
}
public static void main(String[ ] args) {
Student s = new Student();
System.out.println("\nDefault Constructor values: \n");
System.out.println("Student Id : "+s.id + "\nStudent Name : "+s.name);

System.out.println("\nParameterized Constructor values: \n");
Student student = new Student(10, "David");
System.out.println("Student Id : "+student.id + "\nStudent Name :
"+student.name);
}
}
```

Output:

```
this a default constructor
Default Constructor values:
Student Id : 0
Student Name : null

Parameterized Constructor values:
```

```
Student Id : 10
Student Name : David
```

In the above example, the Student class constructor is overloaded with two different constructors, I.e., default and parameterized.
Here, we need to understand the purpose of constructor overloading. Sometimes, we need to use multiple constructors to initialize the different values of the class.

Use of this ( ) in constructor overloading

However, we can use this keyword inside the constructor, which can be used to invoke the other constructor of the same class.
Consider the following example to understand the use of this keyword in constructor overloading.

File: Student.java

```
public class Student {
//instance variables of the class
int id,passoutYear;
String name,contactNo,collegeName;

Student(String contactNo, String collegeName, int passoutYear){
this.contactNo = contactNo;
this.collegeName = collegeName;
this.passoutYear = passoutYear;
}

Student(int id, String name){
this("9899234455", "IIT Kanpur", 2018);
this.id = id;
this.name = name;
}

public static void main(String[] args) {
//object creation
Student s = new Student(101, "John");
System.out.println("Printing Student Information: \n");
System.out.println("Name:"+s.name+"\nId:"+s.id+"\nContact
No.:"+s.contactNo+"\nCollege Name:"+s.contactNo+"\nPassing Year:
"+s.passoutYear);
}
}
```

Output:

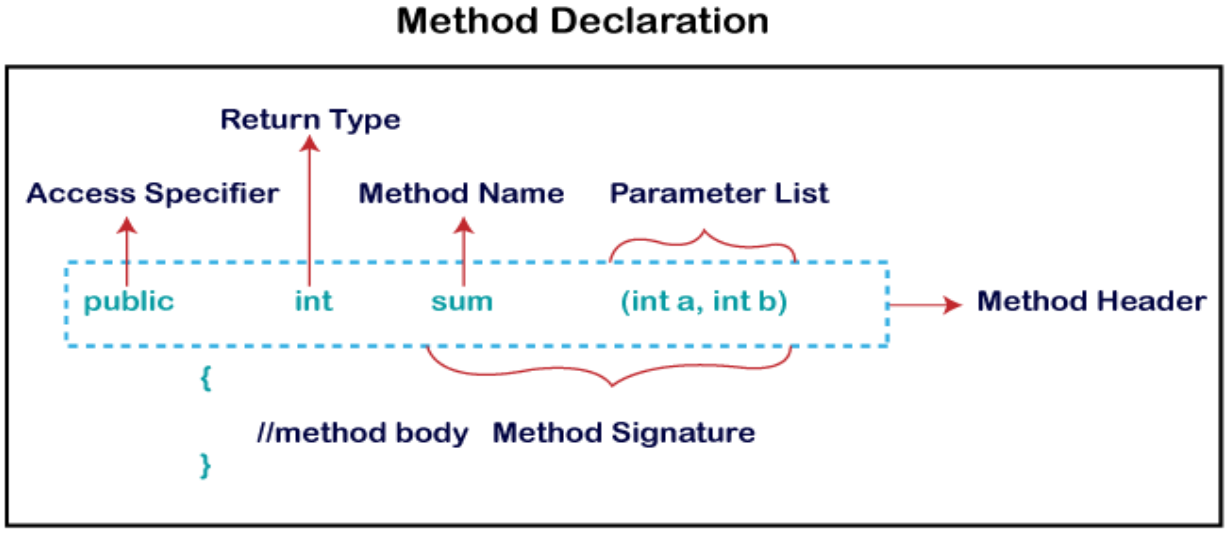
```
Printing Student Information:
Name: John
Id: 101
Contact No.: 9899234455
College Name: 9899234455
Passing Year: 2018
```

Method in Java

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as method header, as we have shown in the following figure.



- **Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.
- **Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:
- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.
- **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.
- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Types of Method

There are two types of methods in Java:

1. Predefined Method

2. User-defined Method

1. Predefined Method

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

File: Demo.java

```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
Output:
The maximum number is: 9
```

User-defined Method

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

How to Create a User-defined Method

Let's create a user defined method that checks the number is even or odd. First, we will define the method.

File: EvenOdd

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from the user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
```

```
}
```
Output 1:
```
Enter the number: 12
12 is even
```

File: findEvenOdd.java
```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
}
```
Output 1:
```
Enter the number: 12
12 is even
```
Output 2:
```
Enter the number: 99
99 is odd
```

File: Addition.java
```
public class Addition
{
public static void main(String[] args)
{
int a = 19;
int b = 5;
//method calling
int c = add(a, b);   //a and b are actual parameters
System.out.println("The sum of a and b is= " + c);
}
//user defined method
```

```
public static int add(int n1, int n2)   //n1 and n2 are formal
parameters
{
int s;
s=n1+n2;
return s; //returning the sum
}
}
Output:
The sum of a and b is= 24
```

Q. Define a class circle having data members pi and radius. Initialize and display values of data members also calculate area of circle and display it.

```
class abc {
float pi,radius;
abc(float p, float r)
{
pi=p;
radius=r;
}
void area()
{
float ar=pi*radius*radius;
System.out.println("Area="+ar);
}
void display()
{
System.out.println("Pi="+pi);
System.out.println("Radius="+radius);
} }
class area
{
public static void main(String args[])
{
abc a=new abc(3.14f,5.0f);
a.display();
a.area();
}
}
```

Static Method

A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method. We can also create a static method by using the keyword static before the method name.
Example of static method

File: Display.java

```
public class Display
{
```

```
public static void main(String[] args)
{
show();
}
static void show()
{
System.out.println("It is an example of static method.");
}
}
```

Output:

```
It is an example of a static method.
```

Method Overloading in Java

Method overloading in Java is the feature that enables defining several methods in a class having the same name but with different parameters lists. These algorithms may vary with regard to the number or type of parameters. When a method is called, Java decides which version of it to execute depending on the arguments given. If we have to perform only one operation, having the same name of the methods increases the readability of the program.

Suppose you have to perform the addition of the given numbers, but there can be any number of arguments if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. Here's an explanation with real-life examples:

Math Operations:

In a math class, you might have multiple methods for adding numbers, each accepting a different number of arguments:

```
public class MathOperations {
    public int add(int a, int b) {
        return a + b;
    }
    public double add(double a, double b, double c) {
        return a + b + c;
    }
}
```

String Manipulation:

In a utility class for string manipulation, you might have overloaded methods for concatenating strings:

```
public class StringUtils {
    public String concatenate(String str1, String str2) {
        return str1 + str2;
    }
    public String concatenate(String str1, String str2, String str3)
{
        return str1 + str2 + str3;
    }
}
```

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

Method overloading in Java allows defining multiple methods with the same name but different parameter lists. One common form of overloading is changing the number of arguments in the method signature. In this example, we have created two methods, the first add() method performs addition of two numbers, and the second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

File: TestOverloading1.java

```
// Class Adder contains overloaded methods to add integers
class Adder {
    // Method to add two integers
    static int add(int a, int b) {
        return a + b;
    }
    // Method to add three integers
    static int add(int a, int b, int c) {
        return a + b + c;
    }
}
public class TestOverloading1 {
    public static void main(String[] args) {
        // Calling the add method with two integers
        System.out.println(Adder.add(11, 11)); // Output: 22
        // Calling the add method with three integers
        System.out.println(Adder.add(11, 11, 11)); // Output: 33
    }
}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

Method overloading in Java also allows changing the data type of arguments in the method signature. Here's an example demonstrating method overloading based on the data type of arguments: In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

File: TestOverloading2.java

```
// Class Adder contains overloaded methods to add numbers
class Adder {
```

```
    // Method to add two integers
    static int add(int a, int b) {
        return a + b;
    }
    // Method to add two doubles
    static double add(double a, double b) {
        return a + b;
    }
}
public class TestOverloading2 {
    public static void main(String[] args) {
        // Calling the add method with two integers
        System.out.println(Adder.add(11, 11)); // Output: 22
        // Calling the add method with two doubles
        System.out.println(Adder.add(12.3, 12.6)); // Output: 24.9
    }
}
```

Output:

```
22
24.9
```

Nesting Of Methods in Java:

Syntax:

```
class Main
{
    method1(){

        // statements
    }
     method2()
    {
       // statements calling method1() from method2()
       method1();
    }
    method3()
    {
      // statements

      // calling of method2() from method3()
      method2();
    }
}
```

Example 1:

Filename: NestingMethodsExample1.java

```
import java.util.Scanner;
public class NestingMethodsExample1 {
     // Method to calculate the perimeter of a rectangle
    int perimeter(int a, int y) {
        int p = 4 * (a + y);
        return p;
```

```
    }
   // Method to calculate the area of a rectangle, using the
perimeter method
    int area(int a, int y) {
        // Calling perimeter method to calculate perimeter
        int p = perimeter(a, y);
        System.out.println("Perimeter: " + p);

        // Calculating area
        int r = a * y;
        return r;
    }
// Method to calculate the volume of a cuboid, using the area method
    int volume(int a, int y, int h) {
        // Calling area method to calculate area
        int r = area(a, y);
        System.out.println("Area: " + r);

        // Calculating volume
        int v = a * y * h;
        return v;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Getting input from user
        System.out.print("Length of the rectangle: ");
        int a = scanner.nextInt();
        System.out.print("Breadth of the rectangle: ");
        int y = scanner.nextInt();
        System.out.print("Height of the cuboid: ");
        int h = scanner.nextInt();

        // Creating object of NestingMethodsExample class
        NestingMethodsExample obj = new NestingMethodsExample();

        // Calling volume method to calculate volume
        int volume = obj.volume(a, y, h);

        // Printing volume
        System.out.println("Volume: " + volume);
    }
}
```

Output:

```
Length of the rectangle: 24
Breadth of the rectangle: 32
Height of the cuboid: 19
Perimeter: 224
Area: 768
Volume: 14592
Example 2:
```

Filename: NestingMethodsExample2.java
import java.util.Scanner;

```
public class NestingMethodsExample2 {
      // Method to calculate the sum of two numbers
    int add(int x, int y) {
        int sum = x + y;
        return sum;
    }
    // Method to calculate the difference between two numbers
    int subtract(int x, int y) {
        int diff = x - y;
        return diff;
    }

    // Method to calculate the product of two numbers, using the add
and subtract methods
    int multiply(int x, int y) {
        // Calling add method to calculate sum
        int sum = add(x, y);

        // Calling subtract method to calculate difference
        int diff = subtract(x, y);

        // Calculating product
        int prod = sum * diff;
        return prod;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Getting input from user
        System.out.print("Enter first number: ");
        int a = scanner.nextInt();
        System.out.print("Enter second number: ");
        int b = scanner.nextInt();

        // Creating object of NestingMethodsExample class
        NestingMethodsExample obj = new NestingMethodsExample();

        // Calling multiply method to calculate product
        int product = obj.multiply(a, b);

        // Printing product
System.out.println("Product of " + a + " and " + b + " is " +
product);
    }
}
```

Output:

```
Enter first number: 5
Enter second number: 3
Product of 5 and 3 is 8
```

Example 3:
Filename: NestingMethodsExample3.java

```
public class NestingMethodsExample3 {

    // Method to swap two numbers
    void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
        System.out.println("After swap: a = " + a + ", b = " + b);
    }

    // Method to multiply two numbers, using the swap method
    int main(int x, int y) {
        System.out.println("Before swap: x = " + x + ", y = " + y);

        // Calling swap method to swap x and y
        swap(x, y);

        // Multiplying x and y
        int product = x * y;
        return product;
    }

    public static void main(String[] args) {
        // Creating object of NestingMethodsExample class
        NestingMethodsExample obj = new NestingMethodsExample();

        // Calling main method to multiply two numbers
        int result = obj.main(5, 10);

        // Printing the product of two numbers
        System.out.println("Product: " + result);
    }
}
```

Output:

```
Before swap: x = 5, y = 10
After swap: a = 10, b = 5
Product: 50
```

Differentiate between method overloading and method overriding.

Sr. No.	Method Overloading	Method Overriding
1	Overloading occurs when two or more methods in one class have the same method name but different parameters	Overriding means having two methods with the same method name and parameters (i.e., method signature)
2	In contrast, reference type determines which overloaded method will be used at compile time.	The real object type in the run-time, not the reference variable's type, determines which overridden method is used at runtime

3	Polymorphism not applies to overloading	Polymorphism applies to overriding
4	overloading is a compile time concept.	Overriding is a run-time concept

Java Command Line Argument:

The java command-line argument is an argument i.e. passed at the time of running the java program
The arguments passed from the console can be received in the java program and it can be used as an input.
So, it provides a convenient way to check the behaviour of the program for the different values. You can pass N (1,2,3 and so on) numbers of arguments from the command prompt.
Command Line Arguments can be used to specify configuration information while launching your application. There is no restriction on the number of java command line arguments.
You can specify any number of arguments.
Information is passed as Strings.
They are captured into the String args of your main method
Simple example of command-line argument in java
In this example, we are receiving only one argument and printing it.
To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}
compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo
Output:
Your first argument is: sonoo
```

Write a program to accept four numbers from user using command line arguments and print the smallest number.

```
public class SmallestNumber{
public static void main(String[] args) {
if (args.length != 4)
{
System.out.println("Please provide exactly 4 numbers as command-line
arguments.");
return;
}
int num1 = Integer.parseInt(args[0]);
int num2 = Integer.parseInt(args[1]);
int num3 = Integer.parseInt(args[2]);
int num4 = Integer.parseInt(args[3]);
int smallest = num1;
if (num2 < smallest)
{
     smallest = num2;
}
```

```
if (num3 < smallest)
{
     smallest = num3;
}
     if (num4 < smallest)
{
     smallest = num4;
}
System.out.println("The smallest number is: " + smallest);
}
}
```

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
class A{
public static void main(String args[]){
for(int i=0;i<args.length;i++)
System.out.println(args[i]);
}
}
compile by > javac A.java
run by > java A mmantc malegaon 1 3 comp
Output: mmantc
malegaon
1
3
comp
```

Garbage collection:

In Java, this process is automated. The garbage collector automatically finds and removes objects that are no longer needed, freeing up memory in the heap. It runs in the background as a daemon thread, helping to manage memory efficiently without requiring the programmer's constant attention.

Working of Garbage Collection

Java garbage collection is an automatic process that manages memory in the heap.
It identifies which objects are still in use (referenced) and which are not in use (unreferenced).
Unreferenced objects can be deleted to free up memory.
The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Types of Activities in Java Garbage Collection

Two types of garbage collection activity usually happen in Java.

Minor or incremental Garbage Collection: This occurs when unreachable objects in the Young Generation heap memory are removed.
Major or Full Garbage Collection: This happens when objects that survived minor garbage collection are removed from the Old Generation heap memory. It occurs less frequently than minor garbage collection.

• Garbage collection is a process in which the memory allocated to objects, which are no longer in use can be freed for further use.
• Garbage collector runs either synchronously when system is out of memory or asynchronously when system is idle.
• In Java it is performed automatically. So, it provides better memory management.
• A garbage collector can be invoked explicitly by writing statement:
System.gc( ); //will call garbage collector.
Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize( ) method before object is garbage collected.

Note: Any suitable example shall be considered.
Example:

```
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();
}
}
```

Access Specifiers in Java

1. public

Accessible from anywhere in the program and used for classes, methods, and variables that need to be accessible globally.

```
Example:
public class PublicExample {
 public void display() {
 System.out.println("This is a public method.");
 }
}
class Main {
 public static void main(String args[]) {
 PublicExample obj = new PublicExample();
 obj.display(); // Accessible from another class
 }
}
```

2. private

Accessible only within the class where it is declared and used to enforce data encapsulation and restrict access to sensitive data.

```
Example:
```

```
class PrivateExample {
 private String message = "This is a private variable.";
 private void display() {
 System.out.println(message);
}
 public void accessPrivate() {
 display(); // Access private method within the same class
 }
}
class Main {
 public static void main(String args[]) {
 PrivateExample obj = new PrivateExample();
 obj.accessPrivate(); // Indirectly access private method
 // obj.display(); // Compilation error
 }
}
```

3. protected

Accessible within the same package or subclasses in different packages and it is used for variables and methods that should be available to subclasses but not to unrelated classes.

Example:

```
public class ProtectedExample {
 protected void display() {
 System.out.println("This is a protected method.");
 }
}
class Subclass extends ProtectedExample {
 public static void main(String args[]) {
 Subclass obj = new Subclass();
 obj.display(); // Accessible in subclass
 }
}
```

4. Default (No keyword required)

Accessible only within the same package and used for methods and variables that don't need to be accessed outside the package.

Example:

```
class DefaultExample {
 void display() {
 System.out.println("This is a default method.");
 }
}
class Main {
 public static void main(String args[]) {
 DefaultExample obj = new DefaultExample();
 obj.display(); // Accessible within the same package
 }
}
```