

Unit - II

Java arrays, Java Strings, Operations on Strings and String Buffer Objects, Class, Objects, Methods and problem solving using classes, objects and relationships. Inheritance, types of Inheritance, packages and interface, exception handling.

UNIT - II

4. Array	41-55
4.1 One-D Array	41
4.2 Two-D Arrays	47
4.3 Multidimensional Arrays	55
5. Java String and String Buffer	56-65
5.1 Introduction String	56
5.2 The StringBuffer Class	60
5.3 <u>Difference Between String, StringBuffer and StringBuilder</u>	62
5.4 <u>Difference Between StringBuffer and StringBuilder</u>	63
6. Class	66-87
6.1 Defining a Class	66
6.2 Object	67
6.3 Access Modifiers/Visibility Label for Class Member	72
6.4 'this' Keyword	74
6.5 Method Overloading	75
6.6 Constructors	77
6.7 Types of Variables	80
6.8 Type of Methods	82
6.9 Nested and Inner Classes	82
6.10 <u>Garbage Collection</u>	85
6.11 <u>Source File Declaration Rules</u>	86
7. Inheritance	88-104
7.1 Inheritance	88

7.2	The Super Keyword	95
7.3	Method Overriding	98
7.4	Final Keyword	100
7.5	Dynamic Method Dispatch	102
7.6	'Instanceof' Operator	102
7.7	Typecasting with Object References	103

8. Package and Interface **105-123**

8.1	Package	105
8.2	Access Protection in Packages	108
8.3	Set CLASSPATH in System Variable	109
8.4	Jar File and its Creation	111
8.5	Abstract Class	112
8.6	Interface	116

9. Exception Handling **124-140**

9.1	Exception	124
9.2	Classifying Exceptions	128
9.3	General Form of Exception	129
9.4	Error	136
9.5	Creating User-Define Exception	136
9.6	Chained Exception	138



Array

4.1 One-D Array

4.1.1 Array Definition

Array is collection of homogenous data type element. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Array is a structured data type with a *fixed* number of components. Every component is of the same type. Components are accessed using their relative positions in the array.

In Java, arrays are objects. Array is the most fundamental data structure. Other data structures are built using arrays. Array provides a convenient way to process large amounts of related data. Java provides a data structure, the *array*, which stores a fixed-size sequential collection of elements of the same type.

The values stored in an array are called *elements*. The individual elements are accessed using an integer *index*.

4.1.2 Array Index

Array index is an integer indicating the position of a value in a data structure. Indexing of array in Java is starting with 0 (element 0, element 1, element 2, and so on).

It might seem more natural to have indexes that start with 1 instead of 0, but Sun decided that Java would use the same indexing scheme that is used in C and C++. This is a convention known as *zero-based indexing*.

4.1.3 Array Declaration

Instead of declaring individual variables, such as n0, n1, ..., and n99, we can declare one array variable. To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. The syntax for declaring an array variable:

```
dataType[] array_Var; // preferred way.  
or  
dataType array_Var[]; // works but not preferred way.
```

Note: The style `dataType[] array_Var` is preferred. The style `dataType array_Var[]` comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example:

```
int arr[];  
or  
int [] arr;
```

Here `arr` is reference variable of `int` array type not array object. In java Array is a Object not primitive type.

```
arr=new int [100];
```

Now new operator creates an array object of int type to store 100 (hundred) integer type values. And reference of this object is assigned to Array variable, that is arr. So, Array reference variable stores the base address and length of Array. As we know in case of C and C++ Array, Array name have no idea about of length of Array. So, C and C++ Array does not check boundary of array. But java Array is much smart and intelligent as java Array checks the boundary of Array and if the User cross the boundary of Array then the Exception (Error occur at runtime is known as exception), called *ArrayIndexOutOfBoundsException* occurs.

There are three ways of array declaration:

1.

```
int arr[];
or
int [] arr;
```

The above statement does two things:

1. It creates an array using new int [arraySize];
2. It assigns the reference of the newly created array to the variable arr.

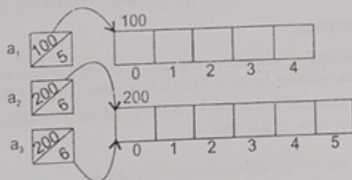
Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

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

If we want to declare more than one array variable of same type than use following syntax:

```
int [] a1, a2, a3;
a1=new int[5];
a2=new int[6];
a3=a2;
```

Here a1, a2, a3 are reference variable of int array type. There are two array objects one having size 5 and another having size 6, not three array objects. Both a2 and a3 are pointing to same array as shown in fig.



So, we can access the array by both variables a2 and a3.

```
int a1[], a2, a3;
```

Here a1 is reference variable array but a2 and a3 are not reference variable of array, they are just int type primitive variable.

2. We can use variable in subscript operator ([]) which is not allow in C and C++.

```
int n=5;
int a[]=new int[n];
```

3. Alternatively you can create arrays as follows:

```
dataType[] array_Var = {value0, value1, ..., value k};
```

The array elements are accessed through the *index*. from 0 to array_Var.length-1.

Example:

```
int a[]={10,20,30,40};
```

Example 1. Program to find maximum in given array.

```
class ArrayMax
{
    public static void main(String arg[])
    {
        int a[]={10,20,30,12,67,45};
        int max=a[0];
        for(int i=1;i<n;i++)
        {
            if(a[i]>max)
                max=a[i];
        }
        System.out.println("Max is="+max);
    }
}
```

Output: Max is =67

Example 2: Write a program in java to print the reverse of given array.

```
import java.util.Scanner;
public class ArrayReverse
{
    public static void main(String arg[])
    {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter Array Size");
        int n=sc.nextInt();
        int a[]= new int[n];

        //input
        for(int i=0;i<n;i++)
        {
            System.out.println("Enter Array element"+(i+1));
            a[i]=sc.nextInt();
        }
        for(int i=0;i<n;i++)
        {
```

```

        System.out.println("Enter Array element"+a[i]);
    }
    System.out.println();
    System.out.println();

    for(int i=n-1;i>=0;i--)
    {
        System.out.println("Reverse"+a[i]);
    }
}

```

Example 3: Write a program in java for linear search.

```

import java.util.Scanner;
class LinSerch
{
    public static void main(String arg[])
    {
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter Array Size");
        int n=sc.nextInt();
        int a[]= new int[n];

        //input
        for(int i=0;i<n;i++)
        {
            System.out.println("Enter Array element"+(i+1));
            a[i]=sc.nextInt();
        }

        System.out.println("Enter item");
        int item=sc.nextInt();
        boolean flag=false;
        //searching logic
        for(int i=0;i<a.length;i++)
        {
            if(a[i]==item)
            {
                flag=true;
                break;
            }
        }
        if(flag)
            System.out.println("Found");
        else
            System.out.println(" Not Found");
    }
}

```

Example 4: Implement Selection sorting algorithm in java.

```

import java.util.Scanner;
public class SelectionSort
{
    public static void main(String arg[])
    {
        Scanner sc =new Scanner(System.in);
        System.out.println("Enter Array size");
        int n=sc.nextInt();
        int m[]=new int[n];
        System.out.println("Enter Array Elements:");
        for(int i=0;i<n;i++)
        {
            System.out.print("Array element is"+(i+1));
            m[i]= sc.nextInt();
        }

        //normal output
        for(int i=0;i<n;i++)

            System.out.print(m[i]+" ");

        System.out.println();

        // Sorting
        for(int i=0;i<n-1;i++)
            for(int j=i+1;j<n;j++)
                if(m[i]>m[j])
                {
                    int t=m[i];
                    m[i]=m[j];
                    m[j]=t;
                }

        // sorted output
        for(int i=0;i<n;i++)

            System.out.print(m[i]+" ");

        System.out.println();
    }
}

```

Example 5: Implement Insertion sorting algorithm in java.

```

import java.util.Scanner;
public class Sort

```

```

{
    public static void main(String arg[])
    {
        Scanner sc =new Scanner(System.in);
        System.out.println("Enter Array size");
        int n=sc.nextInt();
        int m[]=new int[n];
        System.out.println("Enter Array Elements:");
        for(int i=0;i<n;i++)
        {
            System.out.print("Array element is"+(i+1));
            m[i]= sc.nextInt();
        }
        //normal output
        for(int i=0;i<row;i++)

            System.out.print(m [i]+"\t");

        System.out.println();

        // Sorting

        //sorting logic
        for(i=1;i<n;i++)
        {
            itm=a[i];
            //Shifting
            for(j=i-1;j>=0&& a[j]>itm;j--)
                a[j+1]=a[j];
            //insertion
            a[j+1]=itm;
        }
        // sorted output
        for(int i=0;i<n;i++)
        {

            System.out.print(m[i]+"\t");
            System.out.println();
        }
    }
}
//normal output
for(int i=0;i<row;i++)

    System.out.print(m1[i]+"\t");

System.out.println();

// Sorting

```

```

        for(int i=0;i<n-1;i++)
        {
            for(int j=0;j<n-i;j++)
            {
                if(m[j]>m[j+1])
                {
                    t=m[j];
                    m[j]=m[j+1];
                    m[j+1]=t;
                }
            }
        }
        // sorted output
        for(int i=0;i<n;i++)
        {

            System.out.print(m[i]+"\t");

            System.out.println();
        }
    }
}

```

4.1.4 The for-each Loops

JDK 1.5 introduced a new for loop known as for-each loop or enhanced for loop, which enables us to traverse the complete array sequentially without using an index variable.

Example 6: Write a program in java using for-each loop.

```

class Array1
{
    public static void main(String[] args)
    {
        int[] a1 = {5,6,7,8};
        // Print all the array elements
        for (int n: a1)
        {
            System.out.println(n);
        }
    }
}

```

4.2 Two-D Arrays

2-D array is collection of one-D array. 2-D array is also known as matrix. So, two-dimensional array is used to represent a matrix or a table. In Java, you can create n-dimensional arrays for any integer n.

4.2.1 Declaration of 2D Arrays

Syntax:

```
dataType [][] arrayRefVar;
//or dataType arrayRefVar[][];
```

Example:

```
int[][] m1;
m1 = new int[3][4];
//OR
int[][] m1 = new int[3][4];
```

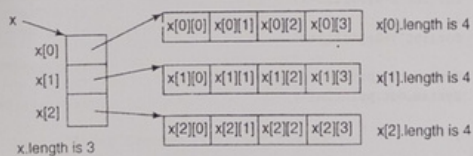
In Java, each subscript must be enclosed in a pair of square brackets. We can also use an array initializer to declare, create and initialize a two-dimensional array. For example,

```
int[ ][ ] array = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12} };
```

or

```
int[ ][ ] array = new int[4][3];
array[0][0] = 1;
array[0][1] = 2;
array[0][2] = 3;
array[1][0] = 4;
array[1][1] = 5;
array[1][2] = 6;
array[2][0] = 7;
array[2][1] = 8;
array[2][2] = 9;
array[3][0] = 10;
array[3][1] = 11;
array[3][2] = 12;
```

```
int[ ][ ] x = new int[3][4];
x.length is 3, x[0].length is 4, x[1].length is 4 and x[2].length is 4.
```



So, we used one-dimensional arrays to model linear collections of elements. And two-dimensional array is a one-dimensional array in which each element is another one-dimensional array.

Example 7: WAP in java to input 2-D array using Console class and display normal output and transposed output.

```
import java.io.Console;
class MatrixTranspose
{
    public static void main(String arg[])
    {
        Console con=System.console();
        System.out.println("Enter array row");
        int r=Integer.parseInt(con.readLine());
        System.out.println("Enter array cols");
        int c=Integer.parseInt(con.readLine());
        int m[][]=new int[r][c];
        System.out.print("Enter Array Elements:");
        for(int i=0;i<r;i++)
            for(int j=0;j<c;j++)
            {
                System.out.print("Array element is"+(i+1)+" "+(j+1)+" : ");
                m[i][j]=Integer.parseInt(con.readLine());
            }

        System.out.println("Normal array");
        for(int i=0;i<r;i++)
        {
            for(int j=0;j<c;j++)
                System.out.print(m[i][j]+" ");

            System.out.println();
        }
        System.out.println("Transpose array");
        for(int i=0;i<c;i++)
        {
            for(int j=0;j<r;j++)
                System.out.print(m[j][i]+" ");

            System.out.println();
        }
    }
}
```


Output:

```

Row number is I matrix of 2
Column number of I matrix
Enter element 1 12
Enter element 1 22
Enter element 2 13
Enter element 2 23
Enter element 2 33
Enter element 2 43
Row number is II matrix of 2
Colm number of II matrix3
Enter element 1 13
Enter element 1 23
Enter element 1 33
Enter element 2 14
Enter element 2 24
Enter element 2 34
Press any key to continue...

```

Example 9: WAP in java to multiply two matrix.

```

import java.util.Scanner;
//import java.io.Console;
class Multi
{
    public static void main(String arg[])
    {
        Scanner sc=new Scanner(System.in);
        // input for first matrix.
        System.out.print("Row number is I matrix of ");
        int r1=sc.nextInt();
        System.out.print("Column number of I matrix");
        int c1=sc.nextInt();
        int m1[][]=new int[r1][c1];

        for(int i=0;i<r1;i++)
        {
            for(int j=0;j<c1;j++)
            {
                System.out.print("Enter Element "+(i+1)+" "+(j+1));
                m1[i][j]=sc.nextInt();
            }
        }
    }
}

```

```

//Input II matrix
System.out.print("Row number is II matrix of ");
int r2=sc.nextInt();
System.out.print("Colm number of II matrix");
int c2=sc.nextInt();
int m2[][]=new int[r2][c2];
for(int i=0;i<r2;i++)
{
    for(int j=0;j<c2;j++)
    {
        System.out.print("Enter Element "+(i+1)+" "+(j+1));
        m2[i][j]=sc.nextInt();
    }
}

if(r2!=c1)
{
    System.out.println("Multioplication not possible");
    System.exit(1);
}

int m3[r1][c2];
for(int i=0; i<r1;i++)
{
    for(int j=0;j<c2;j++)
    {
        for(int k=0;j<c1;k++)
            m3[i][j]=m1[i][k]*m2[k][j];
    }
}

for(int i=0;i<r1;i++)
{
    for(int j=0;j<c2;j++)
    {
        System.out.print(m3[i][j]+" ");
        System.out.println();
    }
}
}

```

4.2.2 Triangular Array

Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths.

```

int [][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];

```

```
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

```
int[][] triangleArray = {
    {1, 2, 3, 4, 5},
    {2, 3, 4, 5},
    {3, 4, 5},
    {4, 5},
    {5}
};
```

Suppose an array matrix is declared as follows:

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

Here are some examples of processing two-dimensional arrays: 0 (Initializing arrays with input values)
The following loop initializes the array with user input values:

```
java.util.Scanner input = new Scanner(System.in);
System.out.println("Enter " + matrix.length + " rows and " + matrix[0].length
+ " columns: ");
for (int row = 0; row < matrix.length; row++)
{
    for (int column = 0; column < matrix[row].length; column++)
    {
        matrix[row][column] = input.nextInt();
    }
}
```

//(Initializing arrays with random values)

Assign random values to the array using the following syntax.

```
for (int r = 0; r < triangleArray.length; r++)
    for (int c = 0; c < triangleArray[r].length; c++)
        triangleArray[r][c] = (int) (Math.random() * 1000);

// (Printing arrays)
for (int r = 0; r < matrix.length; r++)
{
    for (int c = 0; c < matrix[r].length; c++)
    {
        System.out.print(matrix[r][c] + " ");
    }
    System.out.println();
}
```

4.3 Multidimensional Arrays

The following syntax declares a three-dimensional array.

```
double [ ] [ ] [ ] x = new double[2][3][4];
//or
double x [ ] [ ] [ ] = new double[2][3][4];
```

```
x.length is 2 x[0].length is 3, x[1].length is 3 x[0][0].length is 4,
x[0][1].length is 4, x[0][2].length is 4, x[1][0].length is 4,
x[1][1].length is 4, x[1][2].length is 4.
```

Very Short Questions

1. What is array?
2. What is difference between C++ and Java array?
3. What is 2-D array?

Short Questions

1. Write down different ways of array declaration.
2. What is for loop? Explain in detail.

Long Questions

1. What is bubble sort? Implement Bubble sort in java.
2. What is selection sort? Write example to implement selection sort in java.
3. What is insertion sort? Explain in detail.
4. What is binary search? Explain with example of implementation of binary search in java.

CHAPTER » 5

Java String and
String Buffer

5.1 Introduction String

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

In Java, Strings are objects and implemented using two classes, namely String and StringBuffer. Java Strings are more predictable and reliable as compared to C and C++ String. As in case of C, character array ending with null character is known as String. But in case of Java character is totally different from String. There is no null character in Java. Java does not allow operators operation on Strings except +(plus) and +=(plus assignment) operators, which results into String concatenation. String is a built-in class which is available in java.lang package (A package which imports by default) used to create String objects. String is only class which is used as datatype or we can create String class objects without new operators.

5.1.1 String Class

```
String s1=new String("Savita");
String s2="Singh";
System.out.length(s1);
System.out.length(s1.length);
//or
System.out.length("Savita".length);
```

The Java platform provides the String class to create and manipulate strings.

5.1.2 String Class Characteristics

5.1.2.1 String is Object in Java

String represents a sequence of character, but unlike C language which implements String as character array ending with null character, Java represents string as object of class String.

Implementing string as built-in class objects allows Java to provide a full complement of features that makes string handling convenient. String objects can be constructed by a number of ways, make it easy to obtain a string when needed.

5.1.2.2. String is Immutable

Strings are immutable that is we cannot change the string after creation. However a variable declared as a String reference can be changed to point some other String object at any time. We can still perform all type of String operation. Each time we need an altered version of an existing string, a new string object is created that

contains the modification. The original String object remains unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modified string is desired, is a companion class called StringBuffer, whose object contains strings that can be modified after they are created.

5.1.2.3. String Class is Final

Both String and StringBuffer are final. It means we cannot extend String and StringBuffer classes. This allows certain optimization that increases performance of common string operations.

Note: The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder Classes.

5.1.3. Constructors and Methods of String Class

5.1.3.1 Constructors of String Class

1. `public String()`
Initializes a newly created String object so that it represents an empty character sequence.

```
String s1 = new String( );
```

2. `public String(String str)`
Initializes a newly created String object so that it represents the same sequence of characters as the argument, in other words, the newly created string is a copy of the argument string.

```
String s1 = new String("Savita");
String s2 = new String(s1);
```

The string s2 is a copy of string s1. Although contents are same but s1 and s2 points to different string objects.

3. `public String(char ch[])`
Allocates a new String so that it represents the sequence of characters currently contained in character array argument.

```
char ch[] ={'a', 'b', 'c'};
String s1 = new String(ch); //s1 will hold "abc"
```

4. `public String(char value[], int offset, int count)`
Allocates a new String that contains characters from a sub-array of the character array argument.

```
char ch[] ={'a', 'b', 'c', 'd', 'e', 'f'};
String s1 = new String(ch,2,3); //s1 will hold "cde"
```

5. `public String(int value[], int offset, int count)`
Allocates a new String that contains characters from a sub-array of the integer array argument containing unicode of characters.

```
int a[] = {97, 98, 99, 100, 101, 102};
String s1 = new String(a,2,3); //s1 will hold "cde"
```

6. `public String(byte b[], int offset, int count)`
Constructs a new String by decoding the specified sub-array of bytes using the platform's default character set.

```
byte b[] = {97, 98, 99, 100, 101, 102};
String s1 = new String(b,2,3); //s1 will hold "cde"
```

```
7. public String(byte[] )
```

Constructs a new String by decoding the specified array of bytes using the platform's default character set.

```
byte b[] = {97, 98, 99};
String s1 = new String(b); //s1 will hold "abc"
```

Even though Java's char type used 16 bits to represent the unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. In each of the above constructors the byte to character conversion is done by using the default character encoding of the platform.

5.1.3.2 Methods of String Class

```
1. int string.length();
```

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length method, which returns the number of characters contained in the string object.

```
String s1 = "Savita Singh";
int len = s1.length();
System.out.println("String Length is : " + len);
```

```
2. string1.concat(string2)
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat method with string literals, as following:

```
"Savita ".concat("Singh");
Strings are more commonly concatenated with the + operator, as in:
"Hello," + " world"
which results in:
"Hello, world!"
```

```
3. char string1.charAt(int index)
```

Returns the character at the specified index.

```
4. int String1.compareTo(String String2)
```

Compares String1 to another String2.

```
5. int String1.compareToString(String String2)
```

Compares two strings lexicographically.

```
6. int String.compareToIgnoreCase(String String2)
```

Compares two strings lexicographically, ignoring case differences.

```
7. String String1.concat(String str)
```

Concatenates the specified string to the end of this string, which calling the method.

```
8. boolean String1.contentEquals(StringBuffers str)
```

Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.

```
9. static String string1.copyValueOf(char[] data)
```

Returns a String that represents the character sequence in the array specified.

```
10. static String copyValueOf(char[]data, int offset, int count)
```

Returns a String that represents the character sequence in the array specified.

```
11. boolean String1.endsWith(String suffix)
```

Tests if this string ends with the specified suffix.

```
12. byte[] getBytes()
```

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

```
byte[] getBytes(String s);
```

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

```
13. int hashCode()
```

Returns a hash code for this string.

```
14. int indexOf(int ch)
```

Returns the index within this string of the first occurrence of the specified character.

```
15. String replace(char oldChar, char newChar)
```

Returns a new string resulting from replacing all occurrences of old Char in this string with new Char.

```
16. String replaceAll(String regex, String replacement)
```

Replaces each substring of this string that matches the given regular expression with the given replacement.

```
17. String[] split(String regex)
```

Splits this string around matches of the given regular expression.

```
18. String[] split(String regex, int limit)
```

Splits this string around matches of the given regular expression.

```
19. boolean startsWith(String prefix)
```

Tests if this string starts with the specified prefix.

```
20. boolean startsWith(String prefix, int offset)
```

Tests if this string starts with the specified prefix beginning a specified index.

```
21. String substring(beginIndex)
```

Returns a new string that is a substring of this string.

```
22. String substring(intbeginIndex, intendIndex)
```

Returns a new string that is a substring of this string.

```
23. char[] toCharArray()
```

Converts this string to a new character array.

```
24. String toLowerCase()
```

Converts all of the characters in this String to lower case using the rules of the default locale.

25. `String toString()`
Convert any object to String object.
26. `String toUpperCase()`
Converts all of the characters in this String to upper case using the rules of the default
27. `String trim()`
Returns a copy of the string, with removing leading and ending white spaces.

5.2 The StringBuffer Class

We noted earlier that String objects are “immutable” in Java. Once a String object is instantiated, it cannot change in size or content. Any change yields a new String object and the old one is discarded. Strings created with the StringBuffer class, are “mutable”. Once created, characters can change and new characters can be inserted or deleted.

The StringBuffer class is part of the java.lang package.

5.2.1 Common Constructors and Methods are

5.2.1.1 Constructors

1. **StringBuffer():**
Constructs a StringBuffer object with no characters in it and an initialize with capacity or buffer of 16 characters.
2. **StringBuffer(int length):**
It constructs a new StringBuffer object with no characters in it and an initial capacity specified by length.
3. **StringBuffer(String s):**
It constructs a StringBuffer object that is represents the same sequence of characters as that of strings, returns a reference to the new object of StringBuffer.

5.2.1.2 Instance Methods

1. **StringBuffer append(char c):** It appends the character to the stringbuffer and returns a reference to this StringBuffer.
2. **charAt(int i):** returns the char at the specified index in the stringbuffer
3. **deleteCharAt(int i):** it deletes the character at position of StringBuffer, returns a reference to this StringBuffer.
4. **insert(int i, char c):** This methods inserts a character at position, and returns a reference to this StringBuffer.
5. **length():** This method returns and int value that is equals to the length (character count) of the string buffer.
6. **replace(int i, int j, String s):** This method removes characters from position i to j - 1, and then inserts strings, returns a reference to this StringBuffer.
7. **reverse():** This method reverses the characters in the string buffer, returns a reference to this StringBuffer.
8. **setCharAt(int i, char c):** This method sets the character c at position i and returns a reference to this StringBuffer.
9. **String substring(int i):** This method returns a String object equal to a substring of the StringBuffer from position i to the end of the StringBuffer
10. **String toString():** This methods returns a String object equivalent to this StringBuffer.

Note: The append() and insert() methods are overloaded. They can accept any primitive data type as an argument, as well as character arrays or strings.

Example 1: Write a program in java using some StringBuffer methods:

```
import java.util.*;
import java.lang.*;
class StrBuff
{
    public static void main(String args[])
    {
        Scanner sc= new Scanner(System.in);
        StringBuffer str=new StringBuffer("This is StringBuffer Program");
        System.out.println("\n1.Length\n2.Capacity\n3.Setlength\n4.Charat\n5.Setcharat\n6.Append\n7.Deletecharat\n8.Substring\n9.Substring1\n10.Insert\n11.Reverse");
        System.out.println("Enter ur chioce");

        int ch = sc.nextInt();
        switch(ch)
        {
            case 1: System.out.println("The length of the string is:" + str.length());
                    break;
            case 2: System.out.println("The capacity of String:" + str.capacity());
                    break;
            case 3: str.setLength(15);
                    System.out.println("Set length of String:" + str.length());
                    break;
            case 4: System.out.println("The character at 4th position:" + str.charAt(3));
                    break;
            case 5: str.setCharAt(3,'1');
                    System.out.println("The string after setting position is:" +str);
                    break;
            case 6: System.out.println("The string after appending:" + str.append("Sohni");
                    break;
            case 7: System.out.println("The string after deletion:" + str.deleteCharAt(7));
                    break;
            case 8: System.out.println("The substring is:" +str.substring(2));
                    break;
            case 9: System.out.println("The subsstring2 is:" +str.substring(3,8));
                    break;
            case 10: System.out.println(" The string after insertion is:" + str.insert(6,'m');
                    break;
            case 11: System.out.println("The string after reverse is:" + str.reverse());
                    break;
        }
    }
}
```

Output:

```

1.Length
2.Capacity+
3.Setlength
4.Charat
5.Setcharat
6.Append
7.Deletecharat
8.Substring
9.Substring1
10.Insert
11.Reverse
Enter ur choice

```

5.3 Difference Between String, StringBuffer and StringBuilder

5.3.1 String

String objects are *immutable*. It means String once assigned can not be changed. The object created as a String is stored in the *Constant String Pool*. Every immutable object in Java is thread safe that implies String is also thread safe. It means String cannot be used by two threads simultaneously.

```
String s1 = "hello";
// The above object is stored in constant string pool and its value cannot be modified.
```

```
s1="Bye";
//new "Bye" string is created in constant pool and referenced by the s1 variable, but the string "hello" is still exists in string constant pool and its value is not overridden by "Bye" but "hello" string lost its reference variable.
```

5.3.2 StringBuffer

StringBuffer is mutable means one can change the value of the object. The object created through StringBuffer is stored in the *heap* not in *Constant Pool*. StringBuffer has the same methods as that of StringBuilder but each method in StringBuffer is *synchronized* that is *StringBuffer is also thread safe*.

Because of this StringBuffer objects does not allow two threads to simultaneously access the same method. Each method can be accessed by one thread at a time.

But thread safe has a disadvantage as the performance of thread safe object is poor. Thus *StringBuilder* is faster than the *StringBuffer* as *StringBuilder* is not thread safe. So when calling the same methods of each class then the *StringBuilder* objects are much fast then the *StringBuffer* objects.

StringBuffer can be converted to the String by using *toString()* method of String class.

```
StringBuffer s1 = new StringBuffer("Hello"); // The above object stored
//in heap and its value can be changed.
```

```
s1=new StringBuffer("Bye"); // statement is right as it modifies the
//value which is allowed in the StringBuffer
```

5.3.3 StringBuilder

StringBuilder is same as the *StringBuffer*, that is it stores the object in heap and it can also be modified, so objects of *StringBuilder* class are mutable. The main difference between the *StringBuffer* and *StringBuilder* is that *StringBuilder* is also not thread safe. But the *StringBuilder* is fast as it is not *thread safe*.

```
StringBuilder sbl= new StringBuilder("Hello");
```

The above object too is stored in the heap and its value can be modified

```
sbl=new StringBuilder("Bye");
```

Above statement is right as it modifies the value which is allowed in the *StringBuilder*

	String	StringBuffer	StringBuilder
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable)	Yes (mutable)	Yes (mutable)
Thread Safe	Yes	Yes	No
Performance	Fast	Very Slow	Fast

5.4 Difference Between StringBuffer and StringBuilder

StringBuffer	StringBuilder
1. StringBuffer is <i>synchronized</i>	1. String Builder is <i>non-synchronized</i>
2. It is thread safe	2. It is not thread safe
3. Here two threads can't call the same methods of StringBuffer simultaneously.	3. Here two threads can call the methods of StringBuilder simultaneously.
4. It is less efficient.	4. It is more efficient.

Performance Test of StringBuffer and StringBuilder

Example 2: Program using StringBuffer class.

```

class StrBuff
{
    public static void main(String[] args)
    {
        long t = System.currentTimeMillis();
        StringBuffer sb1 = new StringBuffer("Core Java");
        for (int i=0; i<10000; i++)
        {
            sb1.append("J2SE");
        }
        System.out.println("Time taken by StringBuffer class Example : " + (System.
        currentTimeMillis() - t) + "ms");
        t = System.currentTimeMillis();
        StringBuilder sb2 = new StringBuilder("Core Java");
        for (int i=0; i<10000; i++)
        {
            sb2.append("J2SE");
        }
        System.out.println("Time taken by StringBuilder: " + (System.
        currentTimeMillis() - t) + "ms");
    }
}

```

Very Short Questions

1. What is java String?
2. Why StringBuilder class is faster than StringBuffer class?
3. Is StringBuffer class is mutable?

Short Questions

1. What do you mean by String immutable?
2. Why String class is not extends by other classes?
3. What is difference between String and StringBuffere class?
4. Write note on StringBuffer and StringBuilder class.

Long Questions

1. Explain constructors and some important methods of String class.
2. Explain important methods of StringBuffer class.
3. Explain StringBuilder class in detail.

CHAPTER » 6

Class

6.1 Defining a Class

6.1.1 Class

A class declaration only creates a template, it does not create an actual object. A class creates a new data type that can be used to create objects. Class represents ADT (Abstract Data Type). It acts like a template using which we can create multiple objects (instances). Class constructs that defines objects of the same type (set of objects with the same behavior). That is, a class creates a logical framework that defines the relationship between its members. When we declare an object of a class, we are creating an instance of that class. Thus a class is a logical construct. An object has physical reality. That is, an object occupies space in memory.

A class can be defined as a template or blue print that describes the behaviors and states the object of its type. So class encapsulates state and behavior of object, as it allowing the user to encapsulate both data and actions into a single object making the class the ideal structure for representing complex data types. It also defines a collection of state variables, as well as the functionality for working with these variables. Classes are like C structure or Pascal record definitions that allow function definitions within them. Class defines the model for an object.

Definition of data fields: properties of defined objects
 Definition of methods: behaviour of defined objects.

A class is defined using the following template. Class is just a collection of variables (data members), and functions (methods). A simple format for defining a class is given below. The syntax for defining a class:

```
class ClassName
{
    // Define data members; i.e. variables associated with this class
    modifiers datatype varname1;
    modifiers datatype varname1;
    ...
    // Define methods; i.e. functions or methods associated with this class
    modifiers return_type method1(parameter_list);
    modifiers return_type method2(parameter_list);
    ...
}
```

The term "data member" refers to a variable defined in the class.

For example Student class:
 class Student

```
{
```

```
String name;
...
}
```

Here class is keyword, and Student is name of class.

Note: Java class is stored in a ".java" file. The name of the file should match the name of the class having main() method. For example, if you have two classes, one class having main() method, named as Employee, called main class and the other called Money in same file then the file name should be Employee.java.

6.1.2 Member Methods/Function

Methods are functions defined in the class. A method is just a collection of code. The parameter list is used to pass data to the method. Since a method is just like a function, where declaration of return type of the function is must. If the function does not return any type (int, float etc.) value then the method return a special type value called void.

Example:

```
class First
{
    void dispHello()
    {
        System.out.println("Hello");
    }
}
```

6.1.3 Class Variables or Data Members

Class members are declared inside any class and outside any method. Variables defined inside a class are called *member variables*, because they are members of a class. They are also called *instance variables* (if the static keyword is not in declaration) because they are associated with instances of the class. Any method of that class is able to access these member variables.

Example:

```
class Money
{
    .....
    public int dollars;
    public int cents;
}
```

6.2 Object

All objects are instances of a class. Class method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.

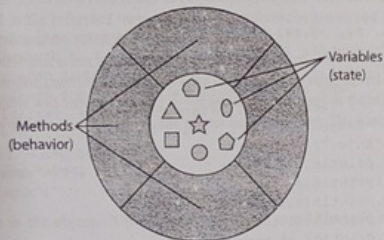
An instance of an object corresponds to the actual object that is created, not the blueprint. In Java, an instance is created when we use the keyword "new". Object is an entity that you can manipulate in your programs. Object has physical occurrence.

A Java class Definition Contains:

Data fields: State of an object. It describes *properties* an object. The values assigned to the fields define the *state* of the object.

Methods: Instructions that accesses or modifies the state of an object. It describes *behaviors* (actions) an object. It can performs the action supply or modify its state.

A Java class is a "*blue print*" for creating objects of that type.

**6.2.1 Creating an Object**

The 'new' keyword is used to create the object. The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In object-oriented programming we describe *types of objects* by defining classes.

We then can create multiple objects from that class and fill in the properties with values specific to each object, and ask the object to perform their behaviors.

Every object belongs to one class and is an *instance of the class*.

In Java, the *new* keyword is used to create new objects. The new keyword allocate memory at runtime (That is dynamic memory allocation).

There are three steps when creating an object from a class:

Declaration: A variable declaration with a variable name with an object type.

```
Student s1;
s1=new Student();
```

```
OR
Student s1=new Student();
```

In the above example s1 is not an object. In C++ s1 will be treated as object but in Java s1 is only a reference variable which can hold reference to an object type Student. It is just like a pointer to an object in C/C++. If this variable is declared inside any block or a method then it is a local variable and will not be initialized to null, but if it is declared in a class then it will be an instance variable and will automatically initialized to null.

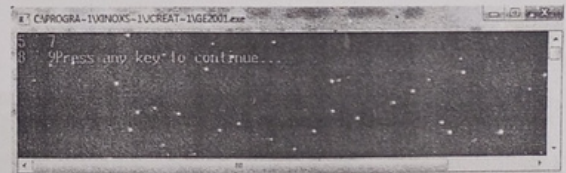
Allocating Memory: In Java memory is allocated dynamically with the help of new operator.

```
s1 = new Student();
```

Here 'new' is a *keyword* and *operator* also. In the above example new operator will allocate memory for an object of class Student, and return its reference, which is then assigned to reference type variable s1. It will also call a default constructor which is made by JVM, to initialize member variables.

Example 1: Program using class member variable.

```
class Abc
{
    int x;
    int y;
}
class AbcTest
{
    public static void main(String arg[])
    {
        Abc a1=new Abc();
        Abc a2=new Abc();
        a1.x=5;
        a1.y=7;
        a2.x=9;
        a2.y=9;
        System.out.println(a1.x+" "+a1.y);
        System.out.print(a2.x+" "+a2.y);
    }
}
```

Output:**Example 2: Program using class member variable and member methods.**

```
import java.util.*;
class A
{
    int x;
    int y;
    void getData()
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter value of x");
    }
}
```

```

x=sc.nextInt();
System.out.print("Enter value of y");
y=sc.nextInt();
}
void setData(int x1, int y1)
{
x=x1;
y=y1;
}
void display()
{
System.out.println(x+" "+y);
}
}
class A2
{
public static void main(String arg[])
{
A a1=new A();
A a2=new A();

a1.getData();
a2.getData();
//a1.setData(5,6);
//a2.setData(9,10);
a1.display();
a2.display();
}
}

```

Output:

```

C:\PROGRAMS\JAVAS\JDK2008\bin>java A2
Enter value of x
5
Enter value of y
6
Enter value of x
9
Enter value of y
10
5 6
9 10
Press any key to continue

```

Example 3: Program having user define class having member variable and member method.

```

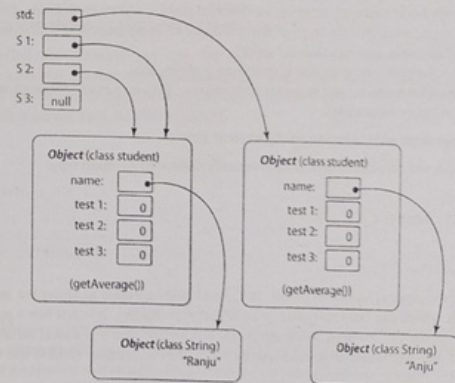
public class Student
{
String name; // Student 's name .

```

```

double test1 , test2, test3; // Grades on three tests .
public double getAverage()
{
(test1 + test2 + test3) / 3;
}
} // end of class Student
class StudentTest
{
public static void main(String ar[])
{
Student s1, s2, s3,s4; // Declare reference variables of Student class
s1 = new Student(); // Create a new object of Student class
s2 = new Student(); // Create a second Student object
s3 = s2; // Copy the reference value in s3 not creating new object
s4 = null; // Store a null reference in the variable s3.
s1.name = "Ranju "; // Set values of some instance variables .
s2.name = "Anju ";
System.out.println( s1.name + " . Your test grades are : ");
System.out.println(s1.test1);
System.out.println(s1.test2);
System.out.println(s1.test3);
System.out.println( s1.name + " . Your test grades are : ");
System.out.println(s1.test1);
System.out.println(s1.test2);
System.out.println(s1.test3);
}
}

```



Here s1 and s2 are two reference variables, which are referencing to different object. The variable s3 and s2 referencing to same object, and s4 reference to a value of null, means doesn't point anywhere. The

72 » Core Java Programming

arrows from s3 and s2 both point to the same object. When one object variable is assigned to another, only a reference is copied. The object referred to is not copied and no new object was created when s3=s2 statement is executed. Both s2 and s3 are referencing to same object.

You can test objects for equality and inequality using the operators == and !=. The condition "if (s2 == s3)", checks whether the values stored in s2 and s3 are the same. But the values are references to objects, not objects. So, you are testing whether s2 and s3 refer to the same object, it means whether they point to the same location in memory.

So the reference variables hold references to objects, not objects themselves. So the conclusion is that the object is not stored in the reference variable but their references are stored in reference variable. The object is somewhere else and the reference variable points to it.

Suppose that a variable that refers to an object is declared to be *final*. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the reference variable exists.

However, this does not prevent the data in the object from changing. The variable is final, not the object.

```
final Student s1 = new Student();
s1.name = "Geeta";
s1.name = "Meena";
```

Here change data in the object. The value stored in s1 is not changed as it still references to the same object.

6.2.2 Fields

The *fields* (instance variables) of an object are the variables that define an object's properties. Fields are class member which are declared inside the class and outside any member method. These variables do not occupy any memory in class but they get memory inside objects. Once an object is created, each field has some value. These values define the *state* of the object and describe the current condition of the object. Class member variable are initialized in objects by their default values if they are not initialized explicitly.

6.2.3 (Instance) Methods

The *behaviors* of an object are defined by the methods we write in the object's class. These (instance) methods report or act upon the data of an object (instance of the class). These methods are accessed by the respective class objects only. But if the methods are declared as static then these can be accessed by class name and object both.

6.3 Access Modifiers/Visibility Label for Class Member

Visibility modifiers are not applicable for local variables, they are applicable for class members only.

6.3.1 Public

A variable or method that is public means that any class can access it. If we specify the modifier public with a member variable or method then that member will be visible to all the classes. This Member can be accessed even outside the package. This is useful for when the variable should be accessible by your entire application. Usually common routines and variables that need to be shared everywhere are declared public.

The main() method is always defined as public because the main() is accessed by JVM which is outside the class in which main() method is defined.

6.3.2 Private

If we specify the modifier private with a member variable or method then that member will not be visible outside the class in which it is declared. This will hide the member of a class from other classes which helps to encapsulate member methods and member variables most effectively. There is only one way to access a private method or variable that is that member can access within the class only that defined them. Private variables and methods are only for the class, which declare them.

6.3.3 Protected

Protected variables and methods allow accessing by the classes inside of the same package, and subclasses of that class available in any package. So a member declared as protected can be accessed from all the classes belonging, the same package. Protected members can also be accessed from any sub class of other packages.

6.3.4 Default Public/Package Public/No Modifier

If no visibility modifier is specified before a member declaration then that member can be accessed from all the classes in the same package. That member cannot be accessed outside the package even by the sub-class.

Example 4: Program using Private class member variable.

```
class A
{
    private int x;
    private int y;
    void setData(int x1, int y1)
    {
        x=x1;
        y=y1;
    }
    void display()
    {
        System.out.println(x+ " " +y);
    }
}
class A2
{
    public static void main(String arg[])
    {
        A a1=new A();
        A a2=new A();
        A1.x=5;//Compile time Error
        a1.setData(5,6);
        a2.setData(9,10);
        a1.display();
        a2.display();
    }
}
```

Output:

```

5
6
9
10
Press any key to continue...

```

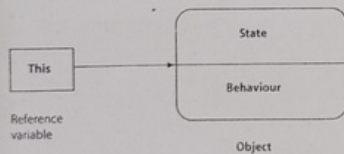
6.4 'this' Keyword

'this' is a reference variable which stores the reference of the object currently used to call the method. The reference of 'this' is replaced by reference of calling object at runtime. Sometime a method will need to refer to the current object.

Scope of class member is in whole class. If local variable have same name as that of class member then local variable have more priority in that block than that of class members, so local variable hides the instance member variable. To distinguish the class member from that of same named local variable, 'this' keyword is used explicitly.

Usage of Java 'this' Keyword

1. 'this' keyword can be used to refer current class instance variable.
2. 'this()' can be used to invoke current class constructor.
3. 'this' keyword can be used to invoke current class method (implicitly)
4. 'this' keyword can be passed as an argument in the method call.
5. 'this' keyword can be passed as argument in the constructor call.
6. 'this' keyword can also be used to return the current class instance.



Example 5: Program using explicit use of 'this' keyword.

```

class A
{
    private int x;
    private int y;
    void setData(int x, int y)
    {
        this.x=x; // this.x: It denotes class member.
    }
}

```

```

// x: denotes local variable.
this.y=y;
}
void display()
{
    System.out.println(x+" "+y);
}
}
class A2
{
    public static void main(String arg[])
    {
        A a1=new A();
        A a2=new A();
        A1.x=5; //Compile time Error
        a1.setData(5,6);
        a2.setData(9,10);
        a1.display();
        a2.display();
    }
}

```

Output:

```

5
6
9
10
Press any key to continue...

```

6.5 Method Overloading

If a class have multiple methods by same name but different in number of argument, differ in type of argument and differ in order of argument then that is known as *Method Overloading*.

While calling an overloaded method it is possible that type of actual parameter passed may not be exactly with the formal parameter of any of the overloaded methods. In that cases parameters are promoted to next higher type till a match is found. If no match is found even after promoting the parameters then there is *compile time error*.

Advantage

1. *Overloaded methods are bonded using static binding in Java*. Which occurs during compile time that is when you compile Java program. During compilation process, compiler bind method call to actual method.

2. *Overloaded methods are fast* because they are bonded during compile time and no check or binding is required during runtime.
3. Most important rule of method overloading in Java is that two overloaded method must have different signature. Method signature means in Java include-
 - 1) Number of argument to a method is part of method signature.
 - 2) Type of argument to a method is also part of method signature.
 - 3) Order of argument also forms part of method signature provided they are of different type.
 - 4) Return type of method is not part of method signature in Java.

Original Method

```
public void show(String message)
{
    System.out.println(message);
}
```

Overloaded method: Number of argument is different

```
public void show(String message, boolean show)
{
    System.out.println(message);
}
```

Overloaded method: Different order of argument

```
public void show (boolean show, String message,)
{
    System.out.println(message);
}
```

Overloaded method: Type of argument is different

```
public void show(Integer n)
{
    System.out.println(n);
}
```

Not Overloaded method: As return type not include in method signature.

```
public boolean show(String message)
{
    System.out.println(message);
    return false;
}
```

Example 6: Program using Method overloading:

```
class A
{
    void sum(int a,int b)
    {
        System.out.println(a+b);
    }
}
```

```
void sum(int a,int b,int c)
{
    System.out.println(a+b+c);
}
void sum(int a,float b)
{
    System.out.println(a+b);
}
void sum(float b,int c)
{
    System.out.println(b+c);
}
/* int sum(int a,int b)//not overloaded method
{
    System.out.println(a-b);
} */

public static void main(String args[]){
    A obj=new A();
    obj.sum(10,10,10);
    obj.sum(20,20);
    obj.sum(5.4f,10);
    obj.sum(20,5.4f);
}
```

Output:

```
C:\Program Files\Winbox Software\UcreatorV3\GE201.exe
30
40
15.4
25.4
Press any key to continue...
```

6.6 Constructors

A class constructor is a method with the same name as the class and no return type. Constructor usually initializes the class member just after the creation of object. Objects are created with the operator, *new*. Constructors are no need to call explicitly as constructor is invoked automatically just after the creation object. Every class has a constructor. If we do not explicitly write a constructor for a class the JVM builds a default constructor for that class which initialize the class member by their default values.

Default values for int data types is zero, for float 0.0f, for double 0.0 and for Boolean false.

Each time a new object is created, at least one constructor will be invoked. It initializes the instance variables of the object. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor with multiple parameters, to differentiate different ways a class may be initialized.

The constructor with no parameters is called the default constructor and it is highly recommended that you always create one.

It is essential that class objects be initialized properly just after their creation. When a class is defined, it is must to set the value for class member. What if the user forgets to initialize the values? This can be such a serious problem but Java provides a mechanism to all class instances is properly initialized. This mechanism is called the class constructor.

Constructors are methods, but they are methods of a special type. They are responsible for creating objects, they exist before any objects have been created. They are more like *static* member methods, but they are not and cannot be declared to be *static*. In fact, according to the JAVA language specification, they are technically not members of the class. Unlike other methods, a constructor can only be called using the *new* operator.

If the programmer doesn't write a constructor definition in a class, then the system will provide a *default* constructor for that class. This default constructor does nothing beyond the basics that is allocate memory and initialize instance variables.

The definition of a constructor looks much like the definition of any other method, *with three differences*.

1. A constructor does not have any return type (not even void). It does not mean that constructor does not return any values, it always return the object of respective class by default.
2. The name of the constructor must be the same as the name of the class.
3. The only modifiers that can be used on a constructor definition are the access modifiers *public*, *private*, and *protected*. A constructor can't be declared *static*.
4. Constructor does not need to call as they always invoked automatically.

However, a constructor does have a method body of the usual form, a block of statements. There is no restrictions on type of statements can be used. And it can have a list of formal parameters. The parameters can provide data to be used in the construction of the object.

Constructor in Java can not be *abstract*, *static*, *final* or *synchronized*.

6.6.1 Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is same as that of method overloading. In case of method overloading, multiple methods with same name but different signature, and in Constructor overloading multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.

6.6.2 Type of Constructor

1. Default Constructor

Default constructor refers to a constructor that is automatically created by compiler in the absence of explicit constructors. This constructor is constructed by compiler if developer does not create any constructor. Once developer creates any constructor explicitly then java compiler does not creates any constructor.

Syntax:

```
class-name ()
{
```

```
-----
-----
-----
}
```

2. Parameterized Constructor

Constructors that can take arguments are then known as parameterized constructors. The number of arguments can be greater or equal to one.

Syntax:

```
Class_name (data type variable_name1, data type variable_name2, ...)
{
-----
-----
-----
}
```

Example 7: Program using constructor overloading.

```
class A
{
    private int x;
    private int y;
    A()//zero argument
    {
        x=y=0;
    }
    A(int n)//one argument
    {
        x=y=n;
    }
    A(int x1, int y1)//zero argument
    {
        x=x1;
        y=y1;
    }
    void display()
    {
        System.out.println(x+" "+y);
    }
}
class A3
{
    public static void main(String arg[])
    {
        A a1=new A();
        a1.display();

        A a2=new A(5);
```

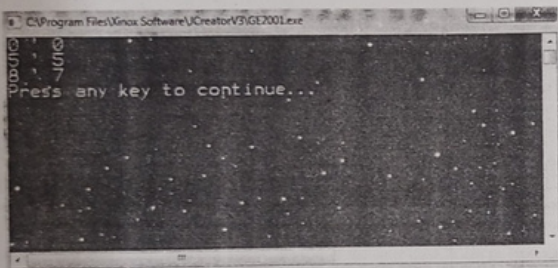
```

a2.display();

A a3=new A(8, 7);
a3.display();
}
}

```

Output:



Constructors are like methods with following differences:

- There is no return type. (The object reference returned always has the type of the class.)
- The name of the constructor is always the name of the class.
- The constructor should initialize all the fields of the object.
- Any Java statement can be in the body of the constructor. For example it might check that a parameter is in the correct range of values.

6.7 Types of Variables

There are three kinds of variables in Java:

1. Local variables
2. Instance variables
3. Class/static variables

6.7.1 Local Variables

1. Local variables are declared in methods, constructors, or blocks.
2. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
3. Access modifiers cannot be used for local variables.
4. Local variables are visible only within the declared method, constructor or block.
5. Local variables are implemented at stack level internally.
6. There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use otherwise there is compile time error.

6.7.2 Instance Variables

1. Instance variables are declared in a class, but outside a method, constructor or any block.
2. When a space is allocated for an object in the heap, a slot for each instance variable value is created.
3. Instance member variable does not get memory inside class, where it is declared.
4. Instance variables occupies memory when an object is created with the use of the keyword 'new' and destroyed (release) when the object is destroyed.
5. Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
6. Instance variables can be declared in class level before or after use.
7. Access modifiers can be given for instance variables.
8. The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However visibility for subclasses can be given for these variables with the use of access modifiers.
9. Instance variables are initialized by their default values. For int the default value is 0, for Booleans it is false, for float it is 0.0f, for double it is 0.0 and for object references it is null. Values can be assigned during the declaration or within the constructor.
10. Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name that is *ObjectReference.VariableName*.

6.7.3 Class/Static Variables

1. Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
2. There would only be one copy of each class variable per class, regardless of how many objects are created from it.
3. Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public, private, final and static. Constant variables never change from their initial value.
4. Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
5. Static variables are created when the program starts and destroyed when the program stops.
6. Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
7. Default values are same as instance variables. For numbers, the default value is 0, for Booleans, it is false, and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
8. Static variables can be accessed by object or by calling with the class name *ClassName.VariableName*.
9. Static member variable does not memory per object, but they get memory only per class that's why static member variable also known as class variables.
10. When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

6.8 Type of Methods

6.8.1 Static Methods

A static method is a method that can be called and executed without creating an object. In general, static methods are used to create instance methods.

Static method can be invoked directly via class name that is we don't have to create an object for a class in order to initiate static method. So static methods can be called by class name or by object.

Disadvantage of static methods is that 'this' and 'super' keyword are not working in static methods.

6.8.2 Instance Methods

These methods act upon the instance variables of a class. Instance methods are being invoked with below syntax:

```
class_object.method_name;
```

6.8.2.1 Instance Methods are Classified into Two Types

a. **Accessor Methods:** These are the methods which read the instance variables that is just go access the instance variables data. Generally these methods are named by prefixing with "get".

b. **Mutator Method:** These are the methods, which not only read the instance variables but also modify the data. Generally these methods are named by prefixing with "set".

Static is another modifier that we can associate with methods or class variables. We indicate that by inserting the keyword static before the method or variable.

6.9 Nested and inner Classes

Nested Classes

Nested class means a class define in another class definition, such a class is known as nested class. In Java, just like member methods, member variables, a class can have another class as its member. Defining a class within another is allowed in Java. The class written within is called the *nested class*, and the class that holds the inner class is called the *outer class*. The scope of nested class is only within the outer class only. But nested class can access the member (even private members) of outer class.

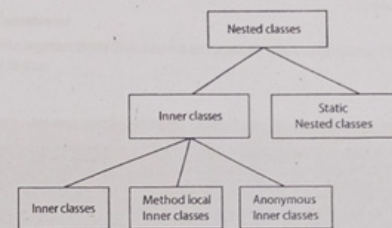
The syntax to write a nested class is given below. Here the class *Outer_Class* is the outer class and the class *Inner_Class* is the nested class.

Syntax

```
class Outer_Class
{
    ...
    ...
    Inner_Class
    {
        ...
        ...
    }
}
```

Nested classes are divided into two types:

- **Non-static nested classes:** These are the non-static members of a class.
- **Static nested classes:** These are the static members of a class.



6.9.1 Inner Classes (Non-static Nested Classes)

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier *private*, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Example 8: Program using non-static inner class.

```
class Outer_Class
{
    int n;

    private class Inner_Class//inner class
    {
        public void print()
        {
            System.out.println("This is an inner class");
        }
    }

    void display_Inner()//Accessing he inner class from the method
    //within
    {
        Inner_Class a1 = new Inner_Class();
        a1.print();
    }
}

public class Inner_1
{
    public static void main(String args[])
    {
        //Instantiating the outer class
    }
}
```

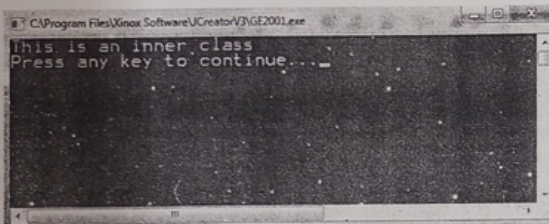


```

Outer_Class ol=new Outer_Class();
//Accessing the display_Inner() method.
ol.display_Inner();
}
}

```

Output:



```

C:\Program Files\Xinox Software\UCreator\3\GE2001.exe
this is an inner class
Press any key to continue...

```

6.9.2 Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class.

Syntax:

```

class Outer_Class
{
.....
static class Inner_class
{
.....
}
}

```

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

Example 9: Program using static inner class.

```

class Inner_2
{
static class Inner_class
{
public void display()
{
System.out.println("This is my static nested class");
}
}
}

```

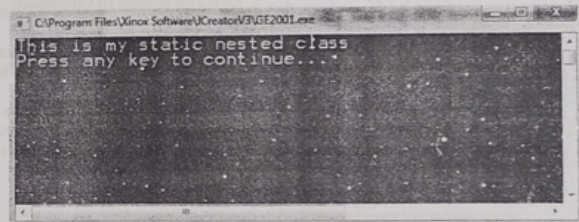
```

}
}

public static void main(String args[])
{
Inner_2.Inner_class al=new Inner_2.Inner_class();
al.display();
}
}

```

Output:



```

C:\Program Files\Xinox Software\UCreator\3\GE2001.exe
this is my static nested class
Press any key to continue...

```

6.9.3 Anonymous Inner Class

An inner class declared without a class name is known as an *anonymous inner class*. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows:

Syntax:

```

Anonymous_Class an_inner= new Anonymous_Class()
{
public void my_method()
{
.....
}
};

```

6.10 Garbage Collection

In JAVA, the destruction of objects takes place automatically. An object exists in the heap, and it can be accessed only through variables that hold references to the object. JAVA uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage".

In many other programming languages, it's the programmer's responsibility to delete the garbage element. Unfortunately, keeping track of memory usage is very difficult, and many serious program bugs are caused by such errors. A programmer might accidentally delete an object even though there are still references to that object.

This is called a dangling pointer error, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted.

6.11 Source File Declaration Rules

1. There can be only one public class per source file.
2. A source file can have multiple non public classes.
3. The public class name should be the name of the source file as well which should be appended by *.java* at the end. For example: the class name is *public class Stud{}* then the source file should be as *Stud.java*.
4. If the class is defined inside a package, then the package statement should be the first statement in the source file.
5. If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
6. Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import or package statements to different classes in the source file.
7. Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. we will be explaining about all these in the access modifiers chapter.



Very Short Questions

1. Write definition of class.
2. What is object?
3. What is member variable?
4. What is member method?
5. What is static member variable?
6. What is static member method?
7. What is instance member variable?
8. What is instance member?
9. What is constructor?

Short Questions

1. What is class? Explain definition of class and different ways of object creation.
2. Explain different access modifiers.
3. Write difference between object and class.
4. Explain instance member and instance methods with example.
5. Explain static member with example.
6. What is difference between class members and instance members?
7. Explain this keyword with example.
8. What is method overloading?
9. What is constructor overloading?
10. Write short note on inner class.

Long Questions

1. What is constructor? Differentiate between constructors and methods. Explain different type of constructors with example.
2. What is 'this' keyword? Explain implicit and explicit use of 'this' keyword with example.
3. What is method overloading? Explain method overloading with example. Write short note on constructor overloading.

CHAPTER » 7

Inheritance

7.1 Inheritance

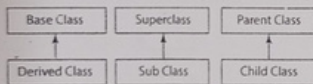
Inheritance is the mechanism of deriving new class from old one, old class is known as superclass and new class is known as subclass. The subclass inherits all of its instances variables and methods defined by the superclass and it also adds its own unique member variable and member function. Thus we can say that subclasses are specialized version of superclass.

Inheritance can be defined as the process where one class acquires the properties of another class. With the use of inheritance the information is made manageable in a hierarchical order.

Inheritance is a mechanism for enhancing existing classes. If it needs to implement a new class and a class representing a more general concept is already available, then the new class can inherit from the existing class.

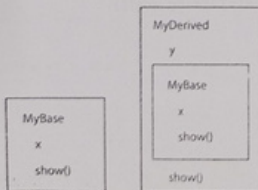
The more general class that forms the basis for inheritance (extended class) is called the superclass. The more specialized class that inherits from the superclass (extending class) is called the subclass.

Inheritance creates a new class definition by building upon an existing definition of the original class. The new class can serve as the basis for another class definition. All Java objects use inheritance. Every Java object can trace back up the inheritance tree to the generic class Object.



Superclass(Base Class)	Subclass(Child Class)
It is a class from which other classes can be derived.	It is a class that inherits some or all members from superclass.

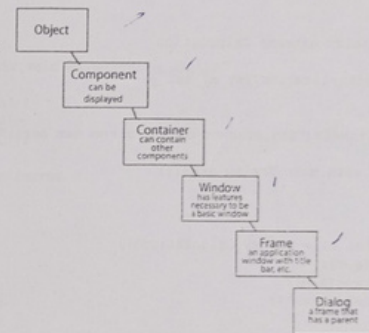
The class which inherits the properties of other is known as subclass *derived class*, *child class* and the class whose properties are inherited is known as superclass *base class*, *parent class*.



7.1.1 Use of Inheritance

1. Reusability of code.
2. Code Sharing.
3. Consistency in using an interface.
4. To model real-world hierarchies.
5. By extending a class, it can add new properties and methods, and you can change the behavior of existing methods.
6. By inheritance declare a method with the same signature and write new code for it.

7.1.2 Hierarchy of Parent to Child



The keyword `extends` is used to base a new class upon an existing class. Below given is the syntax of `extends` keyword.

Syntax:

```

class Super
{
.....
.....
}
class Sub extends Super{
.....
.....
}
    
```

Example 1:

Below given is an example demonstrating Java inheritance. In this example you can observe two classes namely `Calculation` and `My_Calculation`.

Using `extends` keyword the `My_Calculation` inherits the methods `addition` and `Subtraction` of `Calculation` class.

```

class Calculation
{
    int z;
    public void addition(int x, int y)
    {
        z=x+y;
        System.out.println("The sum of the given num bers:"+z);
    }
    public void Substraction(int x,int y)
    {
        z=x-y;
        System.out.println("The difference between the given num bers:"+z);
    }
}
class My_Calculation extends Calculation
{
    public void multiplication(int x, int y)
    {
        z=x* y;
        System.out.println("The product of the given num bers:"+z);
    }
}
public static void main(String args[])
{
    int a=20;
    int b=10;
    My_Calculation a1 = new My_Calculation();
    a1.addition(a,b);
    a1.Substraction(a,b);
    a1.multiplication(a,b);
}

```

Output:

```

The sum of the given num bers:30
The difference between the given num bers:10
The product of the given num bers:200
Press any key to continue.

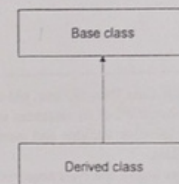
```

Note: A subclass inherits all the members *fields, methods, and nested classes* from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

7.1.3 Types of Inheritance in Java Base Class

7.1.3.1 Single Inheritance



Example 2: Program using single level inheritance

```

class A
{
    public int x;
    public void show()
    {
        System.out.println("x = " + x);
    }
}
class B extends A
{
    public int y;
    public void show()
    {
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
class A1
{
    public static void main(String arg[])
    {
        A a1=new A();
        B b1=new B();
        a1.x=5;
        a1.y=10;//error
        b1.x=15;
        b1.y=25;
        a1.show();
        b1.show();
    }
}

```

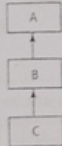
Output:

```
x = 5
x = 15
y = 25
Press any key to continue...
```

Note that private elements become inaccessible to the derived class, this does not mean that they disappear, or that there is no way to affect their values, just that they can't be referenced by name in code within the derived class.

7.1.3.2 Multilevel Inheritance

It is a ladder or hierarchy of single level inheritance. It means if Class A is extended by Class B and then further Class C extends Class B then the whole structure is termed as Multilevel Inheritance. Multiple classes are involved in this inheritance, but one class extends only one. The lowest subclass can make use of all its super classes' members.



Example 3: Program using multilevel inheritance.

```

class A
{
    public int x;
    public void show()
    {
        System.out.println("x = " + x);
    }
}
class B extends A
{
    public int y;
    public void show()
  
```

```

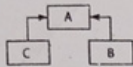
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
class C extends B
{
    public int z;
    public void show()
    {
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("z = " + z);
    }
}
class A1
{
    public static void main(String arg[])
    {
        A a1=new A();
        B b1=new B();
        C c1=new C();
        a1.x=5;
        //a1.y=10;//error
        b1.x=15;
        b1.y=25;
        c1.x=55;
        c1.y=45;
        c1.z=55;
        a1.show();
        b1.show();
        c1.show();
    }
}
  
```

Output:

```
x = 5
x = 15
y = 25
x = 55
y = 45
z = 55
Press any key to continue...
```

7.1.3.3 Hierarchical Inheritance

When one class is extended by many subclasses. It is *one-to-many* relationship.

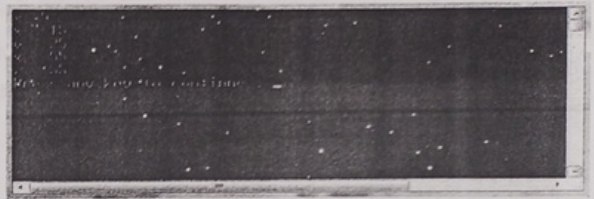


Example 4: Program using multilevel inheritance.

```

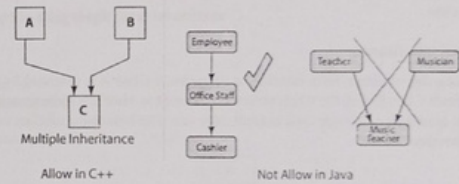
class A
{
    public int x;
    public void show()
    {
        System.out.println("x = " + x);
    }
}
class B extends A
{
    public int y;
    public void show()
    {
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
class C extends A
{
    public int z;
    public void show()
    {
        System.out.println("x = " + x);
        System.out.println("z = " + z);
    }
}
class A1
{
    public static void main(String arg[])
    {
        A a1=new A();
        B b1=new B();
        C c1=new C();
        a1.x=5;
        //a1.y=10;//error
        b1.x=15;
        b1.y=25;
        c1.x=55;
        c1.z=55;
        a1.show();
        b1.show();
        c1.show();
    }
}
  
```

Output:



7.1.3.4 Multiple Inheritance

Multiple inheritances means when one class extends more than one class, which is not allow in java. In C++ this type of inheritance is allow but it result into diamond problem, so java eliminate tne multiple extends.



7.2 The Super Keyword

The 'super' is a reference variable that is used to refer immediate parent class object. The *super* keyword is similar to 'this' keyword following are the scenarios where the super keyword is used. It is used to *differentiate the members* of superclass from the members of subclass, if they have same names.

It is used to invoke the superclass constructor from subclass.

7.2.1 'super' Keyword Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the super class. But if you want to call a parameterized constructor of the super class, you need to use the 'super' keyword as shown below.

super(values);

7.2.2 Uses of Super Keyword are as follows

1. super() is used to invoke immediate parent class constructors
2. super is used to invoke immediate parent class method
3. super is used to refer immediate parent class variable

7.2.3 Restriction for 'Super' Keyword

1. 'super' keyword must be upper first statement in the child constructor or in child method.
2. 'super' keyword is not work in static block or static methods.
3. You cannot do super super to back up two levels.

Example 5: Program using 'super' keyword.

```

ABC(int x1,int y1)
{
    x=x1;
    y=y1;
}
void display()
{
    System.out.println(x+" "+y);
}
}
class B extends ABC
{
    int z;
    B(int x1, int y1, int z1)
    {
        super(x1,y1);
        z=z1;
    }

    void display()
    {
        super.display();
        System.out.println(" "+z+" ");
    }
}
class Inher1
{
    int a;
    public static void main(String arg[])
    {
        B a1=new B(5,6,7);
        B a2=new B(8,9,10);
        a1.display();
        a2.display();
        // System.out.print(" "+a);non static variable can't be used in static method
    }
}

```

Output:

```

5 6
7
8 9
10
Press any key to continue...

```

Since a derived class object contains the elements of a base class object, it is reasonable to use the base class constructor as part of the process of constructing a derived class object.

- But constructors are "not inherited" in derived class. As they would have a different name in the new class and can't be called by name under any circumstances.
- Private members and private methods are also not inherited in derived class.

Example 6: Program to call Program immediate parent Class Constructor using super Keyword.

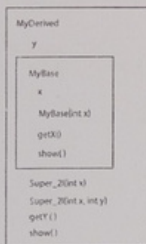
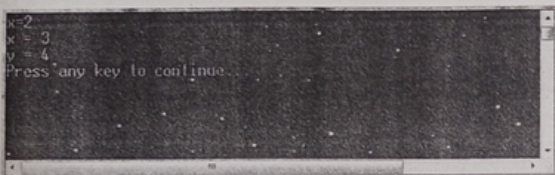
```

class MyBase
{
    private int x;
    public MyBase(int x)
    {
        this.x = x;
    }
    public int getx()
    {
        return x;
    }
    public void show()
    {
        System.out.println("x=" + x);
    }
}
class Super_2 extends MyBase
{
    private int y;
    public Super_2(int x)
    {
        super(x);
    }
    public Super_2(int x, int y)
    {
        super(x);
        this.y = y;
    }
    public int gety()
    {
        return y;
    }
    public void show()
    {
        System.out.println("x = " + getx());
        System.out.println("y = " + y);
    }
}
public class Super2
{
    public static void main(String[] args)
    {

```

```
MyBase b = new MyBase(2);
b.show();
Super_2 d = new Super_2(3, 4);
d.show();
}
```

Output:



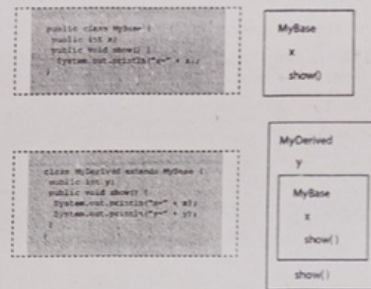
The super() keyword can be used to invoke the Parent class constructor as shown in the above example.

7.3 Method Overriding

When derived class have a method having same signature as that of base class method then, when we call the method by child class object then child class method is invoked not the base class, so the parent (base) class object is override by derived class. But when we call any method which is not define with base class method's signature then by child class object then base class method is invoked.

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not declared as final. The benefit of overriding is ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement. In object-oriented terms, overriding means to override the functionality of an existing method.

As we saw before, you can create a method in the derived class with the same name as a base class method, the new method *overrides* (and hides) the original method. It can still call the base class method from within the derived class if necessary, by adding the super keyword and a dot in front of the method name. It cannot change the return type when overriding a method, since this would make polymorphism impossible.



Example 7: Program using Method Overriding.

```
class A
{
    void m1()
    {
        System.out.println("Base class First method ");
    }
    void m2()
    {
        System.out.println("Base class Second method");
    }
}
class B extends A
{
    void m2()
    {
        System.out.println("Child class overridden method");
    }
}
class A1
{
    public static void main(String arg[])
    {
        A a1=new A();
        B b1=new B();
        a1.m1();
        a1.m2();
        b1.m2();//Child class method is invoked-Method Overriding
        b1.m1();//base class method is invoked
    }
}
```


Output:

```
Base class First method
Base class Second method
Child class overrided method
Base class First method
Press any key to continue...
```

Here m2() method is overridden in the child class. Now when m2() method is invoked from object of child class then compiler will first search for method definition in class from which it is invoked. If method definition is not found then compiler will look for method definition in the parent class.

7.4 Final Keyword

Final keyword can be used in the following ways:

7.4.1 Final Variable

Once a variable is declared as final, its value cannot be changed during the scope of the program.

Example 8: Program using Final Variable.

```
class Final1
{
    public static void main(String arg[])
    {
        final int x=10;
        x=20;// Error as final variable cannot chage there value once //they
        initialized.
    }
}
Error:
D:\Inheritance\Final1.java:7: error: cannot assign a value to final variable x
    x=20;// Error as final variable cannot chage there value once they
    initialized.
    ^
1 error
```

7.4.2 Final Class

A final class cannot be inherited.

Example 9: Program using Final Class.

```
final class A
{
    void m1()
    {
        System.out.println("Base class First method ");
    }
    final void m2()
    {
        System.out.println("Base class Second method");
    }
}
class B extends A
{
    void m2()
    {
        System.out.println("Child class overrided method");
    }
}
class Al
{
    public static void main(String arg[])
    {
        A a1=new A();
        B b1=new B();
        a1.m1();
        a1.m2();
        b1.m2();//Child class method is invoked-Method Overridding
        b1.m1();//base class method is invoked
    }
}
```

```
{
    System.out.println("Base class Second method");
}
}
class B extends A
{
    void m2()
    {
        System.out.println("Child class overrided method");
    }
}
class Al
{
    public static void main(String arg[])
    {
        A a1=new A();
        B b1=new B();
        a1.m1();
        a1.m2();
        b1.m2();//Child class method is invoked-Method Overridding
        b1.m1();//base class method is invoked
    }
}
```

Error:

```
Build Output
D:\javaprg\javaprg\Lecture\Inheritance\Final3.java:12: error: cannot inherit from class A
D:\naatrix\javaprg\Lecture\Inheritance\Final3.java:14: error: m2() in B
    void m2()
    overridden method is final
2 errors
Process completed.
```

If a method was public in the base class, the derived class may not override it with a method that has protected, private or package (default public) access.

This avoids a logical inconsistency, as a base class variable can reference a derived class object, the compiler will allow it to access something that was public in the base class, if the derived class object actually referenced had changed the access level to private, then the element ought to be unavailable, this logic could be applied to any restriction in access level, not just public to private.

So when we override any base class method in child class then the child class method must have same scope or higher scope than that of base class method, only and only overriding takes place otherwise there is no overriding takes place.

7.5 Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism. Method to execution based upon the type of the object being referred to at the time invoke occurs. Thus, this determination is made at run time. In other words, it is the type of the object being referred to that determines which version of an overridden method will be executed.

When there is base class reference and child class object then runtime binding takes place. It is also known as *dynamic binding* or *late binding*.

7.6 'Instanceof' Operator

The instanceof operator is used in comparisons. It gives a boolean answer when used to compare an object reference with a class name

Syntax:

```
referenceVariable instanceof ObjectType
```

- It will yield true if the object is an instance of that class.
- It will also give true if the object is a derived class of the one tested.
- If the test yields true, then you can safely typecast to call a derived class method.

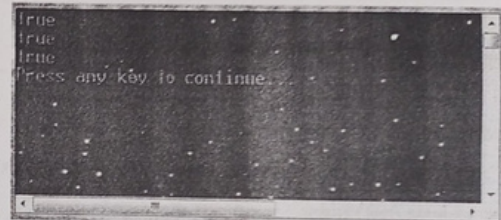
Example 10: Program using instance of operator.

```
class A
{
    //.....
}
class B extends A
{
    //.....
}
class C extends B
{
    //.....
}
class E
{
    //.....
}
class D extends C
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        D d = new D();
        E e = new E();
        System.out.println(b instanceof A);
    }
}
```

```
//System.out.println(e instanceof A); // error: incompatible types:
// E cannot be converted to A

System.out.println(d instanceof B);
System.out.println(d instanceof A);
}
}
```

This would produce the following result:



7.7 Typecasting with Object References

Object references can be typecast only along a chain of inheritance

- if class B is derived from A, then a reference to one can be typecast to the other an *upcast* converts from a derived type to a base type, will done implicitly, because it is guaranteed that everything that could be used in the parent is also in the child class.

But downcasting is not implicit, this require explicit type cast.

Example:

```
Object o;
String s = new String("Hello");
Object o = new String("Hello"); // implicit type cast- in upcasting
String t;
t = (String) o; //explicit type cast- in downcasting.
```

Very Short Questions

1. What is super class?
2. Define child class.
3. Why java does not allow multiple inheritances?
4. What is role of super key word?
5. Define method overriding.
6. Define instanceof operator.

Short Questions

1. Differentiate child classes and base class.
2. What are the uses of inheritance?
3. Explain super keyword with example. Also write down uses and restriction of super keyword.
4. What is method overriding? Explain with example.
5. Explain reference typecasting with example.

Long Questions

1. What is inheritance? Explain multilevel inheritance by using an appropriate example.
2. Explain different type of inheritance in java with example.
3. What is role of final keyword? Explain final keyword with class, variable and method using example.

Package and Interface**8.1 Package**

Package is a collection of related classes, interfaces and sub-packages that provides access protection and namespace management.

- Avoid name conflicts
- Access control

So packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages. Many implementations of Java use a hierarchical file system to manage source and class files. It is easy to organize class files into packages. Packages are used in order to prevent naming conflicts, to control access, to make searching and usage of classes, interfaces, enumerations and annotations easier, etc. A Package can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and name space management.

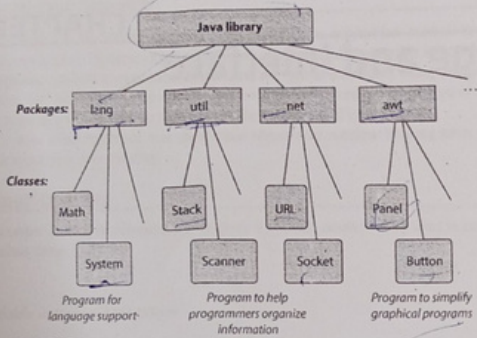
Some of the existing packages in Java are:

- `java.lang` - bundles the fundamental classes.
- `java.io` - classes for input, output functions are bundled in this package
- `java.awt` - Class for GUI designing
- `java.awt.event` - Classes for event Handling.

8.1.1 Advantages of Using a Package

- **Reusability:** Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- **Easy to locate the files.**

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes put collectively. Using packages, it is easier to provide access control and it is also easier to locate the related classes.



8.1.2 User Define Package

While creating a package, 'package' keyword is used. Then choose a name for the package and the package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Syntax

```
package package_name;
class class_name
{
    _____
    _____
    _____
}
```

Example 1:

```
package p1;
public class A
{
    public int x;
}
```

Here Class A is in package p1;

```
package p1;
class B
{
    public static void main(String a[])
    {
        A a1=new A();
    }
}
```

M S U P
S S S B

```
a1.x=5;
System.out.print(a1.x);
}
```

Here class B is also in package p1, so we can use class A directly without import statement.

Now compile the file on command window like

```
javac -d . A.java
javac -d . B.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Then run (execute) class B.class with java command.

8.1.3 Use Predefine Package

8.1.3.1 The 'import' Keyword

If a class wants to use another class in the same package, then import does not need to be used and no need to import another package. Classes in the same package find each other without any special syntax. And if the required class is available in another package then it required to extra syntax in program. The package can be imported using the import keyword.

To import a package use keyword "import" followed by package name and class name. If you want to import all classes use a wildcard * instead of single class name.

Example 2:

```
package p2;
import p1.*;
class C
{
    public static void main(String a[])
    {
        A a1=new A();
        a1.x=10;
        System.out.print(a1.x);
    }
}
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

'import' keyword is used to import built-in and user-defined packages into your java source file. So that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to class that is present in different package Using fully qualified name (But this is not a good practice.)

Example:

```
class MyClass
{
    java.util.Scanner sc = new java.util.Scanner(System.in);
    _____
}
```

8.1.3.2 Import Only Required Class

Example:

```
import java.util.Scanner;
class Myclass
{
    Scanner sc = new Scanner(System.in);
    ...
}
```

8.1.3.3 Import Whole Package

To import package can use an asterisk (*) as a "wild card" and import all the classes in a package at once. The * is a "regular expression operator" that will match any combination of characters. Therefore, this import statement will import all classes and interfaces of that package but not sub-package and its classes.

```
import java.util.*;
class Myclass
{
    Scanner sc = new Scanner(System.in);
    ...
}
```

8.2 Access Protection in Packages

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non subclass	No	No	No	Yes

- We can access a public variable outside the package.
- We cannot access default public variables outside the package. If we do so then compiler gives following error.



- We cannot access a variable with private access modifier outside the package. This is clear case in which you don't have permission to access private variable outside its scope.
- We can access a variable with protected access modifier outside the package.

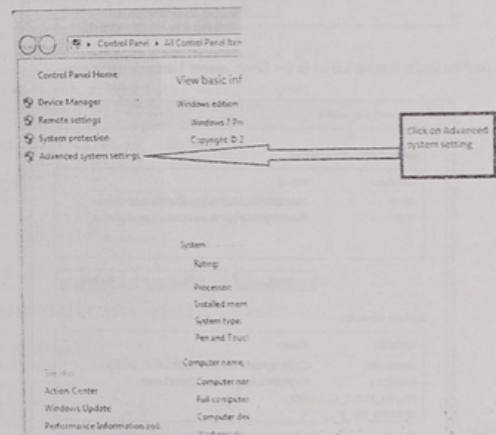
8.3 Set CLASSPATH in System Variable

The full path to the classes directory, is called the class path and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path. Say <package>\classes is the class path, and the package name is com.college, then the compiler and JVM will look for .class files in <package>\classes\com\college.

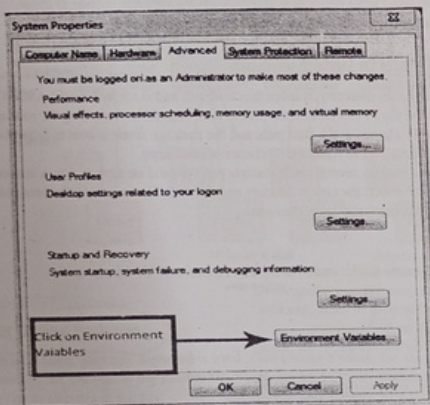
A class path may include several paths. Multiple paths should be separated by a semicolon. By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Steps

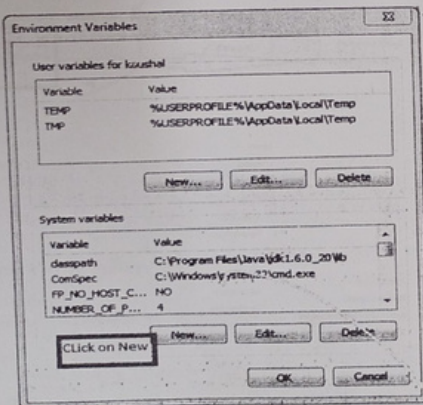
1. First of all create folder (main) in any drive, (assume C: drive) Then make .java file as creating above.
2. Then copy the path of these .java files.
3. Right Click on My Computer
4. Now click on Advanced System Setting.



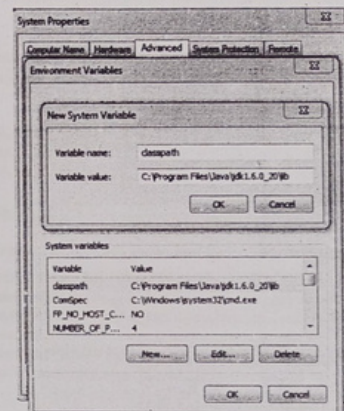
5. Now we will set the classpath and path for java in our windows 7 machine



6. Click on New under system variable on Environment Variables Window.



7. Before Variable Name= classpath and Variable Value =<package>\classes\com\college



8. Now click Ok and again click Ok and Ok on Environment Variables.

Finally we have set classpath and now start java coding, and now you can compile and run your program from anywhere.

8.4 Jar File and its Creation

The JAR file format is a compressed format of java class files to distribute Java applications and libraries. It is built like ZIP file format, and functions in a similar way. many files are compressed and packaged together in a single file, making it easy to distribute the files over a network. *JAR file in Java* is a kind of zip file which holds all contents of a Java application including Class files, resources such as images, sound files and optional Manifest file. *JAR stands for Java Archive* and provides a platform independent deliverable for java programs, libraries and framework. Same jar file can execute on any operating system without any modification like Windows 7, windows 8, Macintosh or Linux. Apart from platform independence and standard delivery method *jar file* also provides *compression of contents* which results in faster download if you are downloading java program from internet.

Open command window apply the following command:

```
C:\> Java jar -cvf HelloJarFile.jar HelloWorld.class
```

Now a jar file is created with HelloJarFile.jar

As per dictionary, abstraction is the quality of dealing with ideas rather than events. Abstraction is a process of hiding the implementation of details from the user, only the functionality will be provided to the user. In other words user will have the information on what the object does instead of how it does. In Java Abstraction are achieved using Abstract classes and Interfaces.

8.5 Abstract Class

A class which contains the *abstract* keyword in its declaration is known as abstract class. Abstract classes may or may not contain *abstract methods*. But, if class containing at least one abstract method, then the class *must* be declared abstract. Otherwise there is compile time error.

If a class is declared abstract it cannot be instantiated. An abstract class cannot be instantiated using the new operator. You can still define its constructors, which are invoked in the constructors of its subclasses. In this case, you cannot create instances of the class using the new operator - this class can be used as a base class for defining a new subclass. This means that the actual values must be objects of *some* subclass of the abstract class but we don't know (or don't care) which. To use an abstract class you have to inherit it from another class, provide implementations to the abstract methods in it.

If you inherit an abstract class you have to provide implementations to all the abstract methods in it.

8.5.1 Abstract Method

An abstract method contains a method signature, but no method body. Instead of curly braces an abstract method will have a semicolon (;) at the end. So, the methods that are declared without any body within an abstract class is known as abstract method. Abstract methods are only for override by sub-class. Abstract methods are never called. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

The method body will be defined by its subclass. Declaring a method as abstract has two consequences:

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note: Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Abstract method can never be final. As final methods are never override and abstract methods are only for overriding.

Abstract method can never be final and static. As static methods says call me by class name and abstract methods says we are never called just only for overriding.

Syntax:

```
abstract return_type function_name (); // No definition
```

Syntax:

```
abstract class class_Name
```

```
{
    void m1()
    {
        .....
    }
    void m2()
    {
        .....
    }
    abstract void m3();
}
```

Abstract class is mainly used when sometimes it is require being a set of subclasses that share a base class, but where it only makes sense to have objects that belong to the subclasses. There will never be an object belonging just to the base class.

"Abstract" base classes are used for this situation. Methods in an abstract class may either be "abstract," in which case no body is given, just a semi-colon.

Normal (non-abstract) methods may be given that use other normal methods and abstract methods to give common implementation.

If a class is declared abstract it cannot be instantiated. And if we do so then there is compile time error, which is discussed in below example.

Example 1:

This is an example of the abstract class to create an abstract class just use the *abstract* keyword before the class keyword, in the class declaration. When the keyword *abstract* appears in a class definition, it means that zero or more of its methods are abstract.

```
import java.util.*;
abstract class A
{
    private int x;
    private int y;
    void getData()
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter value of x");
        x=sc.nextInt();
        System.out.print("Enter value of y");
        y=sc.nextInt();
    }
    void setData(int x1, int y1)
    {
        x=x1;
        y=y1;
    }
    abstract void display();
}
class A2
{
    public static void main(String arg[])
    {
        A a1=new A();
        A a2=new A(5);
        a1.getData();
        a2.getData();
        //a1.setData(5,6);
        //a2.setData(9,10);
        a1.display();
        a2.display();
    }
}
```

```

class A2
{
    public static void main(String arg[])
    {
        A a1=new A();
        A a2=new A();

        a1.getData();
        a2.getData();

        a1.display();
        a2.display();
    }
}

```

As discussed earlier that it is not allow in java to create object of abstract class. If we do so than there is compile time error.

```

Build Output
D:\matrix\javaprg\Lecture\class obj\44.java:25: error: A is abstract: cannot be instantiated
    A a1=new A();
                ^
D:\matrix\javaprg\Lecture\class obj\44.java:26: error: A is abstract: cannot be instantiated
    A a2=new A(5);
                ^
2 errors
Process completed.

```

To use an abstract class inherit it from another class, and provide implementations to the abstract methods in sub class. If there inherit an abstract class you have to provide implementations to all the abstract methods in it as discussed in following example

Example 2:

This is an example of the abstract class to create an abstract class and to use its methods first inherit it, as given below.

```

import java.util.*;
abstract class A

```

```

{
    int x;
    int y;
    void getData()
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter value of x");
        x=sc.nextInt();
        System.out.print("Enter value of y");
        y=sc.nextInt();
    }
    void setData(int x1, int y1)
    {
        x=x1;
        y=y1;
    }
    abstract void display();
}
class Abc extends A
{
    void displayChild()
    {
        System.out.println("This is child method");
    }
    void display()
    {
        System.out.println(x+" "+y);
    }
}
class A2
{
    public static void main(String arg[])
    {
        Abc a1=new Abc();
        Abc a2=new Abc();
        a1.getData();
        a2.getData();
        //a1.setData(5,6);
        //a2.setData(9,10);
        a1.display();
        a2.display();
        a1.displayChild();
        a2.displayChild();
    }
}

```


Output:

```

Enter value of x:
Enter value of y:56
Enter value of x:
Enter value of y:77
This is child method
This is child method
Press any key to continue...

```

8.6 Interface

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. An interface is not a class. Creating an interface is similar to writing a class but they are two different concepts. A class describes the attributes and behaviour of an object. An interface contains behaviors that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class. The keyword 'interface' is used to create interface.

An interface is a reference type in Java, it is similar to class but it is a collection of abstract methods. A class implements an interface thereby inheriting the abstract methods of the interface. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object.

An interface is not a class but interface is a Special Class. Define a set of behaviors. It allow "multiple inheritance" by implementing multiple interfaces

It has no state and no behavior. It merely tells you which methods you should implement. A superclass has state and behavior, and the subclasses inherit them.

Sometimes we want a set of classes to share some behaviour but they do not share a base class. "Interfaces" are used for this situation.

The methods declared in an interface don't have method bodies. By default all the methods in an interface are *public abstract*. Similarly all the variables we define in an interface are essentially constants because they are implicitly *public, static, final*. So, the following definition of interface is equivalent to the above definition.

It has no state and no behavior. It merely tells you which methods you should implement. A superclass has state and behavior, and the subclasses inherit them.

Sometimes we want a set of classes to share some behaviour but they do not share a base class. "Interfaces" are used for this situation.

The methods declared in an interface don't have method bodies. By default all the methods in an interface are *public abstract*. Similarly all the variables we define in an interface are essentially constants because they are implicitly *public, static, final*. So, the following definition of interface is equivalent to the above definition.

8.6.1 Declaring Interfaces

```

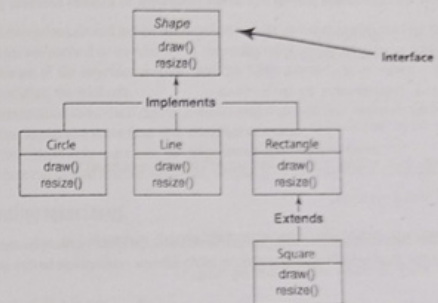
interface MyInterface
{
    /* All the methods are public abstract by default

```

```

Note down that these methods are not having body
*/
public void method1();
public void method2();
}

```



Example 3:

The 'interface' keyword is used to declare an interface. Here is a simple example to declare an interface:

```

public interface Shape
{
    double PI = 3.14; // static and final => upper case
    void draw(); // automatic public
    void resize(); // automatic public
}
class Rectangle implements Shape
{
    public void draw()
    {
        System.out.println ("Rectangle");
    }
    public void resize()
    {
        /* dummy defined method */
    }
}
class Square extends Rectangle
{
    public void draw()
    {
        System.out.println ("Square");
    }
    public void resize()

```

```

    {
        /* dummy defined method */
    }
}

interface MyInterface
{
    public void method1();
    public void method2();
}

class Int1 implements MyInterface
{
    public void method1()
    {
        System.out.println("implementation of method1");
    }
    public void method2()
    {
        System.out.println("implementation of method2");
    }
    public static void main(String arg[])
    {
        MyInterface obj = new Int1();
        obj.method1();
    }
}

```

Output:



```

implementation of method1
Press any key to continue...

```

8.6.2 Interface and Inheritance

If a class does not perform all the behaviors of the interface, the class must declare itself as abstract. A class uses the *implements* keyword to implement an interface. The *implements* keyword appears in the class declaration following the *extends* portion of the declaration.

```

public interface Inf1
{
    public void method1();
}

public interface Inf2 extends Inf1
{
    public void method2();
}

```

```

public class Abc implements Inf2
{
    public void method1()
    {
        //Implementation of method1
    }
    public void method2()
    {
        //Implementation of method2
    }
}

```

Example 4:

```

interface A
{
    void a();
    void b();
    void c();
    void d();
}

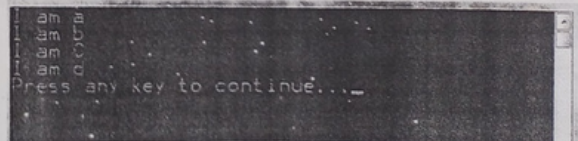
abstract class B implements A{
    public void c(){System.out.println("I am C");}
}

class M extends B{
    public void a(){System.out.println("I am a");}
    public void b(){System.out.println("I am b");}
    public void d(){System.out.println("I am d");}
}

class Test5{
    public static void main(String args[]){
        A a=new M();
        a.a();
        a.b();
        a.c();
        a.d();
    }
}

```

Output:



```

I am a
I am b
I am C
I am d
Press any key to continue...

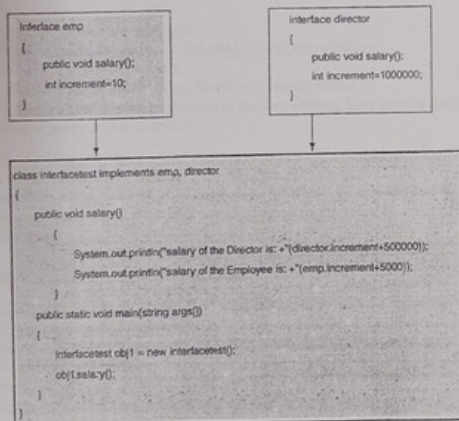
```

A class can extend one class but can implement many interface. Interface can extend interface but not extends a class as syntax given below:

```
interface InterfaceName
{
    // "constant" declarations
    // method declarations
}
// inheritance between interfaces
interface InterfaceName extends InterfaceName
{
    ...
}
// not possible
interface InterfaceName extends ClassName
{ ... }
// not possible
interface InterfaceName extends InterfaceName1, InterfaceName2
{ ... }
```

8.6.3 Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, and an interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.



In the above example, two interfaces are created with names "emp" and "director" and both interfaces contain same method salary(). In the class Main Class we have implemented both the interfaces. Now as per

the definition of interface a method declared must be overridden in the class that implements the interface. Although in the above example there are two salary methods declared in different interfaces, but in the child class there is only single instance of the method. So whenever a call is made to the salary method it is always the overridden method that gets invoked, leaving no chance for ambiguity.

When overriding methods defined in interfaces there are several rules:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.
- A class can implement more than one interface at a time.
- A class can extend only one class, but implement multiple interfaces.
- An interface can extend another interface, similarly to the way that a class can extend another class.

8.6.4 Extending Interfaces

An interface can extend another interface, similarly to the way that a class can extend another class. The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

8.6.5 Tagging/Marked Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

```
package java.util;

public interface EventListener
{ ... }
```

An interface with no methods in it is referred to as a *tagging* interface. The basic design purposes of tagging interfaces is to create a common parent as with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

8.6.6 Benefits of Having Interfaces

Following are the advantages of interfaces:

1. Without bothering about the implementation part, we can achieve the security of implementation.
2. In java, *multiple inheritance* is not allowed, However by using interfaces you can achieve the same. A class can extend only one class but can implement any number of interfaces. It saves you from Deadly Diamond of Death(DDD) problem.

8.6.7 Similarity and Difference Between Classes and Interface

8.6.7.1 An Interface is Similar to a Class

- An interface can contain any number of methods.
- An interface is written in a file with a *.java* extension.

- The byte code of an interface appears in a *.class* file.
- Interfaces appear in packages, and their corresponding byte code file must be in a directory structure that matches the package name.

8.6.7.2 Interface is Different from a Class

Abstract class	Interface
1. Abstract class can have <i>abstract and non-abstract methods</i> .	1. Interface can have <i>only abstract methods</i> .
2. Abstract class <i>doesn't support multiple inheritance</i> .	2. Interface <i>supports multiple inheritance</i> .
3. Abstract class can have <i>final, non-final, static and non-static variables</i> .	3. Interface has <i>only static and final variables</i> .
4. Abstract class can have <i>static methods, main method and constructor</i> .	4. Interface <i>can't have static methods, main method or constructor</i> .
5. Abstract class can <i>provide the implementation of interface</i> .	5. Interface <i>can't provide the implementation of abstract class</i> .
6. The <i>abstract</i> keyword is used to declare abstract class.	6. The <i>interface</i> keyword is used to declare interface.
7. Example: <pre>public abstract class Shape { public abstract void draw(); }</pre>	7. Example: <pre>public interface Drawable { void draw(); }</pre>

Very Short Questions

1. Define package in java with syntax.
2. What is jar file?
3. What is abstract class in java with syntax?
4. What is abstract method with syntax?
5. Define interface with syntax.
6. Is multiple inheritance of interface is allow? Define with syntax.
7. Is inheritance of interface and class is allow simultaneously by a child class? Define it with syntax.

Short Questions

1. What is package in java? Explain advantage of package.
2. Write down note on User define package.
3. Explain different ways of importing pre-define packages.
4. Write short note on access modifiers with package.
5. What is jar file? Write down command to create jar file.

6. What is abstract class in java? Write down features of abstract class.
7. What is abstract method and write down different features of abstract method.
8. What is compile time and runtime binding?
9. Write short note on interface features.
10. Write difference between abstract class and interface.
11. What are similarities between abstract class and interface.

Long Questions

1. What is user define package? Explain pre define package with example.
2. Explain abstract class and abstract method with example. Write down features of abstract class and abstract methods.
3. Explain interface with example and also explain interface with inheritance.

CHAPTER » 9

Exception Handling

9.1 Exception

An Exception is a condition that is caused by a runtime error in the program that breaks the normal flow of program and then program/Application terminates abnormally which is not recommended. When the java interpreter encounters an error such divide by zero, it creates an exception object and throw it. A java exception or error is an object of class Exception or on of their sub-classes whenever exception occurs at runtime. This exception object contains details about the exception which can be accessed.

An exception can occur for many different reasons, some are given below:

1. A user has entered invalid data.
2. A file trying to open cannot be exist.
3. A network connection has lost in the middle of communications.
4. The JVM has run out of memory.
5. The class file want to load may not found or in wrong format.
6. The String which we want to convert in number format, have invalid format of characters.
7. A reference pointing to is in use etc.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

These exceptions are handled in java using some handlers. So an exception is a problem that arises during the execution of a program. And java provides some handler to handle these exceptions, this is known as Exception handling.

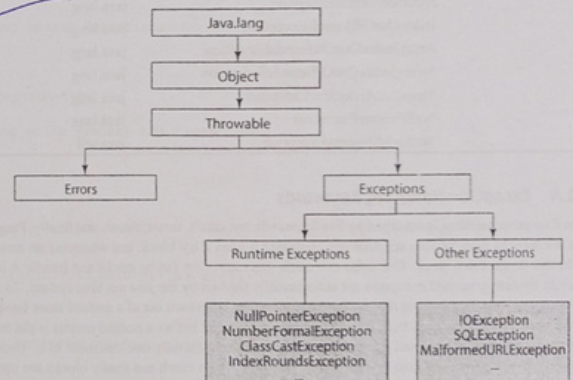
9.1.1 Advantage of Exception Handling

1. The Code having chances to give error is separated error from the remaining code.
2. Code that handles errors and unusual events can be separated from the no code.
3. Errors automatically propagate up the calling chain until they are handled.
4. Errors and special conditions can be classified and grouped according to common properties.
5. Programmer gets a chance to recover from error.
6. With java there is no need to test if an Exception occurs. Adding more errors simply requires more catches blocks without touching remaining program.
7. Increase the readability of program as error containing code is separate from remaining code.

Here are some Exceptions which are occurred at runtime,

Exception Type	Cause of Exception
Arithmetic Exception	Divide by zero.
ArraIndexOutOfBoundsException	Out of limit of array indexes.
ArrayStoreException	Store the wrong type of data in an array.
FileNotFoundException	Access a non-existing file.
IOException	Inability to read from a file.
NullPointerException	Caused by referencing a null object.
NumberFormatException	Occur when conversion between string and number takes place.
OutOfMemoryException	Occur when not enough memory to allocate a new object.
StringIndexOutOfBoundsException	Occur accessing a non-existence character position in String.

9.1.2 Exception Hierarchy



All exception classes are sub-class of the *java.lang.Exception* class. The exception class is as sub-class of the *Throwable* class. Other than the Exception class there is another sub-class called Error which is also sub-class of *Throwable* class.

Errors are not normally handled form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment like JVM is out of Memory. Normally programs cannot recover from these type of errors.

The Exception class has two main subclasses: *IOException* class and *RuntimeException* Class.

9.1.3 Exceptions List

Object	java.lang
Throwable	java.lang
ClassNotFoundException	java.lang
InterruptedException	java.lang
NoSuchMethodException	java.lang
IOException	java.io
EOFException	java.io
FileNotFoundException	java.io
MalformedURLException	java.net
UnknownHostException	java.net
AWTException	java.awt
RuntimeException	java.lang
ArithmeticException	java.lang
ClassCastException	java.lang
IllegalArgumentException	java.lang
NumberFormatException	java.lang
IndexOutOfBoundsException	java.lang
ArrayIndexOutOfBoundsException	java.lang
StringIndexOutOfBoundsException	java.lang
NegativeArraySizeException	java.lang
NullPointerException	java.lang
NoSuchElementException	java.util

9.1.4 Exception Handling Keywords

Java Exception handling is managed by five keywords: try, catch, throw, throws and finally. Program statements that we want to monitor for exception are contained within a try block, and whenever an error occurs then it is thrown to the catch block. Our code can catch this exception (using catch) and handle it in some rational manner. System generated exception are automatically thrown by the java run time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed before a method returns is put in a finally block. The try block must be enclosed between braces even if there- is only one statement in it. There can be one or more catch statements and zero or one finally statement. Both catch and finally blocks are optional but either one catch block or one finally block is must. The code in the try block is executed like any other Java code. If the code inside the try block executes successfully then the control goes to finally block if it is present and then the execution continues from the statement just after the end of finally block.

If some run-time error occurs while executing the code in try block then JVM throws an Exception / Error. This means an object of type Exception or Error or one of its sub-classes is created depending on the type of the run-time error. This is then compared with the Exception/Error types of the catch blocks in top to bottom order. If a matching catch block is found then the exception / error is handled i.e. program will not terminate. The execution continues with the first statement in the catch block. On completion of the catch block, execution continues with the statement just after the end of try & catch statement. Only one catch block is executed irrespective of the

number of catch blocks. If Exception / Error do not match with Exception /Error type of any of the catch blocks, then it is not handled. The execution will immediately return from try block. The code in the finally block will be executed (if finally block is available), even if the Exception / Error is not handled. The Exception / Error will then be handled by outer try block if there is one otherwise it must be handled in the method which called the current method. The JVM then simply terminates the program and displays the exception / error details or the monitor / console if Exception is not handled.

Example 1:

```
class Excp_1
{
    public static void main(String arg[])
    {
        int a=5;
        int b =;
        int c= a/b;
        System.out.println("After Exception");
    }
}
```

Output: Program will terminate due to ArithmeticException.

Example 2:

```
class AgExp extends Exception
{
    void display()
    {
        System.out.print("Hello ");
    }
}
class Excp
{
    public static void main(String arg[])
    {
        A a1=null;
        a1.display();
    }
}
```

Output: In above program NullPointerException occurs, as the reference variable is not initialize with new.

Example 3:

```
class Excp_4
{
    public static void main(String arg[])
    {
        int c[]={1,2,3,4,5};
    }
}
```

```

        System.out.println(c[5]);
    }
}

```

Output: In above case length of array is 4 and array index is from 0 to 4 and we are accessing 6th element, so *ArrayIndexOutOfBoundsException* takes place.

9.1.5 Exception Method

1. public String getMessage()

To obtain about the exception that has occurred. To obtain the error message associated with exception or error. This is initialized in the *Throwable* constructor.

2. public Throwable getCause()

Returns the cause of the exception as represented by a *Throwable* object.

3. public String toString()

Returns the name of the class concatenated with the result of *getMessage()*.

4. public void printStackTrace()

Prints the result of *toString()* along with the stack trace to *System.err*, the error output stream. To print a stack trace showing where the error occurred.

5. public StackTraceElement[] getStackTrace()

Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

6. public Throwable fillInStackTrace()

Fills the stack trace of this *Throwable* object with the current stack trace, adding to any previous information in the stack trace.

9.2 Classifying Exceptions

9.2.1 Checked Exceptions

Checked Exceptions: A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the Programmer should *handle* these exceptions.

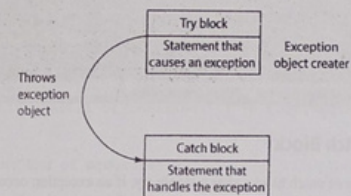
For example: *FileNotFoundException*, *ClassNotFoundException* and all its sub-classes except *RuntimeException* and its subclasses must be handled by an exception handler using *catch* or must be specified using a *throws* clause in method header.

9.2.2 Unchecked Exceptions

Unchecked Exceptions: An Unchecked exception is usually programmer errors, or an exception that occurs at the time of execution, these are also called as *Runtime Exceptions*, these include programming bugs, such as logic errors or improper use of an API. Runtime Exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an *ArrayIndexOutOfBoundsException* occurs.

9.3 General Form of Exception



9.3.1 Syntax of Try Catch

```

...
...
try
{
    Statement having Exception;
    ...
}
catch(Exception-type e)
{
    Statement which Process the Exception;
    ...
}

```

Example 4:

```

class Excp_1
{
    public static void main(String arg[])
    {
        int a=10;
        int b = 0;
        System.out.print("Before Exception");
        try
        {
            int c = a/b;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by zero not possible");
        }
        System.out.println("After Exception");
    }
}

```

Output:



9.3.2 Multiple Catch Block

There can be any number of catch blocks after a single try. If an exception occurs in try block, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches `ExceptionType1`, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Syntax of Multiple catch block:

```
try
{
    Statement having Exception;
    ...
    ...
}
catch (ExceptionType1 e)
{
    //handle exceptions of this type and its subclasses
}
catch (ExceptionType2 e)
{
    // handle these exceptions
}
catch (ExceptionType3 e)
{
    // handle these exceptions
}

try
{
    Statement having Exception;
    ...
    ...
}
```

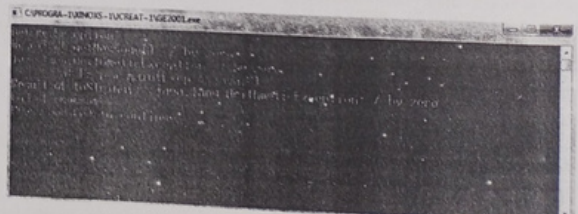
```
catch (ExceptionType1 | ExceptionType2 | ExceptionType2 e )
{
    // handle exceptions of all type
    //which comes first
}
```

Example 5:

```
class Excp_4
{
    public static void main(String arg[])
    {
        try
        {
            System.out.println("Before Exception");
            int a=arg.length;
            int b = 58/a;
            System.out.println(b);
            int c[]={1};
            c[47]=100;

            System.out.println(b);
            System.out.println("After Exception");
        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage());
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        System.out.println("End of program");
    }
}
```

Output:



9.3.3 Nested Try-catch Blocks

When try-catch block uses within try-catch then it is known as nested try-catch block.

Syntax:

```
try
{
...
try
{
Statement having Exception;
...
}
}
catch (ExceptionType1 e)
{
//handle exceptions of this type and its subclasses
}
catch (ExceptionType2 e)
{
// handle these exceptions
}
catch (ExceptionType e)
{
// handle these exceptions
}
```

Example 6:

```
class Excp_4
{
public static void main(String arg[])
{
try
{
int a=arg.length;
int b = 58/a;
System.out.println("number of argument "+a);
try
{
if (a==1)
a=a/a-1;
if(a==2)
{
int c[]={5};
c[4]=100;
}
}
catch(ArrayIndexOutOfBoundsException e)
```

```
{
System.out.println(e);
}
}
catch(ArithmeticException e)
{
System.out.println("Result of getMessage() :"+e.getMessage());
e.printStackTrace();
System.out.println("Result of toString() :"+e.toString());
}
}
System.out.println("End of programe");
}
```

9.3.4 Finally Statements

The finally block follows a try block or a catch block. It may be added immediately after try block or after last catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any cleanup type statements that is must to execute. finally block is not executed only if there is power off or failure of respective hardware due to any reason.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
//Exception containing code
}catch(ExceptionType1 e)
{
//Handling Exception
}catch(ExceptionType2 e2)
{
//Handling Exception
}catch(ExceptionType3 e3)
{
//Handling Exception
}finally
{
//The finally block always executes.
}
```

Example 7:

```
class Excp_1
{
public static void main(String arg[])
{
int a=10;
int b = 0;
```

```

System.out.println("Before Exception");
try
{
    int c= a/b;
}
catch(ArrayIndexOutOfBoundsException e )
{
    System.out.println(" "+e.getMessage());
}
finally
{
    System.out.println("always execute");
}
System.out.println("After Exception");
}

```

In above program the exception is ArithmeticException which is not handled in try block, and even then program is not terminated without execution of finally block.

So, finally block is always executed just before termination of finally block. So a program always check finally block just before termination of program.

Consequences:

- When an exception is occurred in try block, that is automatically thrown to catch block. So catch block is executed only if error is occurs.
- If no catch block is found to respective error, the system make check for finally block, and then terminates the block.
- If and an exception is caught, control continues with the code immediately following the try-catch-finally block in which the exception was caught.

9.3.5 Throw Keyword

As we know, Exception object is thrown by automatically by java-runtime-system. However an exception object is also throw explicitly by programmer using *throw* keyword.

Syntax:

```
throw object;
```

Here, object must be an object of type Throwable or a subclass of Throwable. So we can throw an exception, either a newly instantiated one or an exception that you just caught, by using the *throw* keyword. So, *throw* is used to invoke an exception explicitly.

Example 8:

```

class Use_throw
{
    int ag;
    public void setAge(int ag)
    {
        if(ag>0)

```

```

        this.ag=ag;
    }
    else
    {
        NullPointerException e= new NullPointerException();
        throw e;
        //throw new NullPointerException("Invalid Age");
    }
}

public static void main(String arg[])
{
    try
    {
        Use_throw al=new Use_throw();
        al.setAge(-22);
    }
    catch(NullPointerException e)
    {
        System.out.println(e.getMessage());
    }
}
}

```



9.3.6 Throws Keyword

If a method does not handle a checked exception, the method must declare it using the *throws* keyword. The *throws* keyword appears at the end of a method's signature. If a method is throwing of an exception that it does not handle, it must specify its behavior so that caller method can guard themselves against that unhandled exception. This can done by using a *throws* keyword in the method's declaration/method signature. A *throws* keyword lists the type of exception that a method might throw. This is necessary for all checked exceptions. All exceptions that a method can throw must be declared in the *throws* clause. If they are not, a compile-time error will result.

Syntax:

```

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

```

Try to understand the difference between *throws* and *throw* keywords, *throws* is used to postpone the handline of a checked exception and *throw* is used to invoke an exception explicitly.

136 » Core Java Programming

Example 9:

```

class AgExp
{
    int ag;
    public void setAge(int ag) throws ArithmeticException
    {
        if(ag>0)
            this.ag=ag;
        else
        {
            ArithmeticException e= new ArithmeticException();
            throw e;
        }
    }
    public void disp()
    {
        System.out.print("Age is "+ag);
    }
}
class Age_2
{
    public static void main(String arg[])
    {
        try
        {
            AgExp al=new AgExp();
            al.setAge(-22);
            al.disp();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

9.4 Error

Errors are typically ignored in code because errors are not solved by programs. These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

9.5 Creating User-Define Exception

We can create our own exceptions in Java. To create our own exception it is must that our class must be sub-class of Exception/Error or Throwable. So, we just need to extend the predefined Exception class to create your own Exception. If you want to write a runtime exception, you need to extend the RuntimeException class.

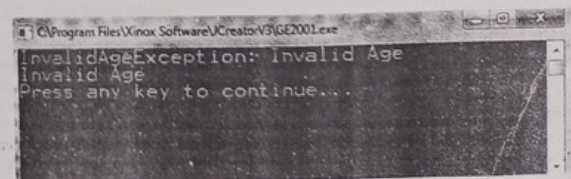
Example 10:

```

class InvalidAgeException extends Exception
{
    InvalidAgeException(String msg)
    {
        super(msg);
    }
}
class User_class
{
    int age;
    public void setAge(int age) throws InvalidAgeException
    {
        this.age=age;
        if(age<0)
        {
            InvalidAgeException e1=new InvalidAgeException("Invalid Age");
            throw e1;
        }
    }
}
public static void main(String arg[])
{
    User_class al=new User_class ();
    try
    {
        al.setAge(-20);
    }
    catch(InvalidAgeException e)
    {
        System.out.println(e);
        System.out.println(e.getMessage());
    }
}
}

```

Output:



The screenshot shows a window titled "C:\Program Files\Innox Software\Creator\VS\GE2001.exe". The console output displays the following text:

```

InvalidAgeException: Invalid Age
Invalid Age
Press any key to continue...

```

9.6 Chained Exception

Chained Exception was added to Java in JDK 1.4. This feature allow you to relate one exception with another exception, that is one exception describes cause of another exception. For example, consider a situation in which a method throws an *ArithmeticException* because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only *ArithmeticException* to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Two new constructors and two methods were added to *Throwable* class to support chained exception.

1. *Throwable* (*Throwable cause*)
2. *Throwable* (*String str, Throwable cause*)

In the first constructor, the parameter *cause* specifies the actual cause of exception. In the second constructor, it allows us to add an exception description in string form with the actual cause of exception. *getCause()* and *initCause()* are the two methods added to *Throwable* class.

- *getCause()* method returns the actual cause associated with current exception.
- *initCause()* set an underlying cause(exception) with invoking exception.

Example 11:

```
class AgeExp extends Exception
{
    AgeExp(String msg)
    {
        super(msg);
    }
}
class FeeExp extends Exception
{
    FeeExp(String msg)
    {
        super(msg);
    }
}
class AdmisionExp extends Exception
{
    AdmisionExp(String msg)
    {
        super(msg);
    }
}
class Admisionp
{
    private int age,fees;
    public void setData(int age, int fees) throws AgeExp, AdmisionExp, FeeExp
    {
        if(age<0)
        {
            AgeExp e1=new AgeExp("Invalid Age");
            AdmisionExp e2=new AdmisionExp("Admission Failed");
            e2.initCause(e1);
```

```
        throw e2;
    }
    else if(fees<1000)
    {
        FeeExp e1=new FeeExp("insufficient fess ");
        AdmisionExp e2=new AdmisionExp("Admission Failed");
        e2.initCause(e1);
        throw e2;
    }
    else
    {
        this.age=age;
        this.fees=fees;
    }
}
public void disp()
{
    System.out.println("Age is "+age);
    System.out.println("Fees is "+fees);
}
}
class ChainExp
{
    public static void main(String arg[])
    {
        try
        {
            Admisionp a1= new Admisionp();
            a1.setData(22,500);
            a1.disp();
        }
        catch(AdmisionExp| AgeExp| FeeExp e)
        {
            System.out.println(e);
            System.out.println(e.getMessage());
            e.printStackTrace();
            System.out.println("Initial cause "+e.getCause());
        }
    }
}
```

Very Short Questions

1. What do you mean by exception?
2. What do you mean by exception handling?
3. Define try and catch block.

4. What is nested try catch?
5. What are finally statements?

Short Questions

1. What do you mean by exception handling? Write down advantage of exception handling.
2. Explain checked and unchecked exceptions.
3. Explain keywords used in exception handling with example.
4. Explain multiple catch blocks with example.
5. What are finally statements? Write Difference between throw and throws keywords.

Long Questions

1. What is exception handling? Write down advantage of exception handling. Explain keywords used in exception handling with example.
2. What is user defined exceptions? Explain with example.
3. What is chained exceptions? Explain with example.

