304185

Unit 3: Methods & Inheritance in JAVA

TE E&TE

Syllabus

Pune Institute of Computer Technology, Pune

- Abstract Methods and classes, Strings, One dimensional and two dimensional arrays, wrapper classes, enumerated types, Command line arguments, Inheritance: Inheritance in Java, Creating Multilevel hierarchy, Constructors in derived class, Method overriding, Dynamic method dispatch.

One Dimensional Array

A *one-dimensional array* is, essentially, a list of like-typed variables.

Java Array is also called as subscripted variable.

The general form of a one-dimensional array declaration is

```
datatype var-name[ ];
```

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold.

For example, the following declares an array named **month_days** with the type “array of int”:

```
int month_days[];
```

Array Initialization during Declaration

- Arrays can be initialized when they are declared.
- An array initializer is a list of comma-separated expressions surrounded by curly braces.
- The commas separate the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use new.

Example

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

    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                         30, 31 };
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

Multidimensional Arrays

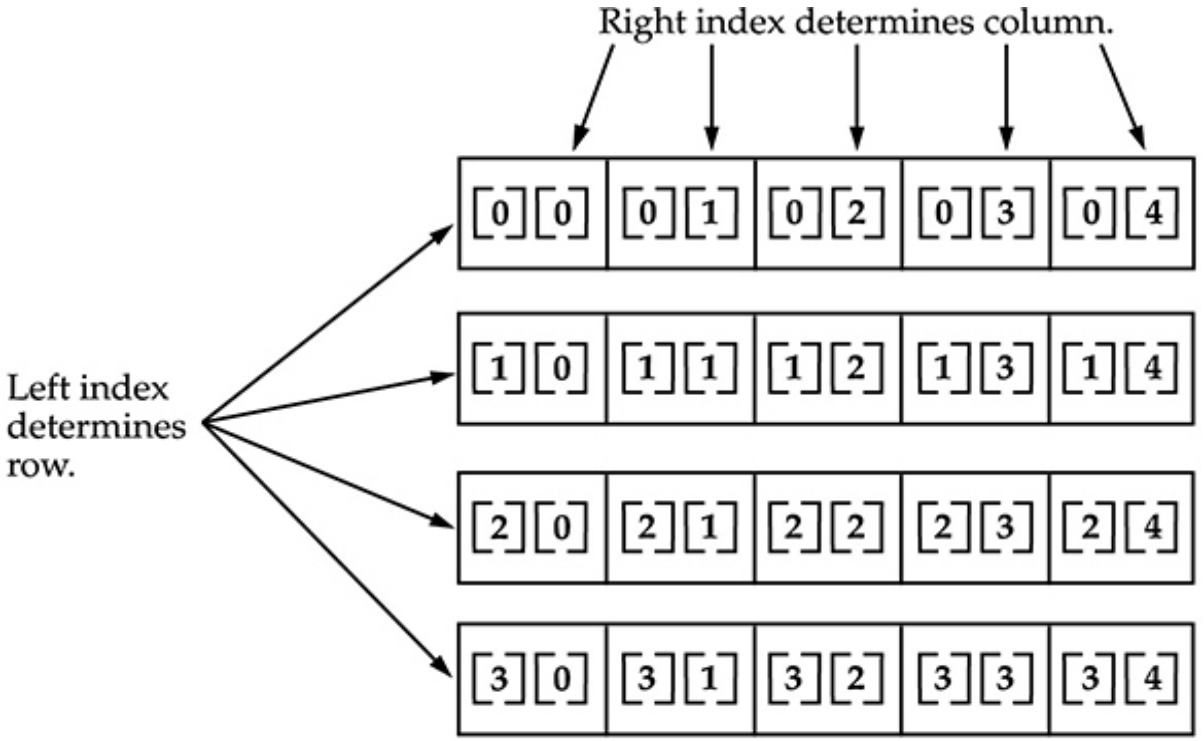
In Java, *multidimensional arrays* are implemented as arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

For example, the following declares a two-dimensional array variable called **twoD**:

```
int twoD[][] = new int[4][5];
```

Multidimensional Arrays

- Internally, this matrix is implemented as an array of arrays of int. Conceptually, this array will look like the one shown in Figure:



Given: int twoD [ ] [ ] = new int [4] [5] ;

Example

```
// Demonstrate a two-dimensional array.
class TwoDArray {
  public static void main(String args[]) {
    int twoD[][]= new int[4][5];
    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<5; j++) {
        twoD[i][j] = k;
        k++;
      }

    for(i=0; i<4; i++) {
      for(j=0; j<5; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

Multidimensional Array Manual Memory Allocation

When the memory for a multidimensional array is allocated, it is necessary to only specify the memory for the first (leftmost) dimension.

The memory allocation of remaining dimensions can be done separately.

For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension separately.

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

Example

Pune Institute of Computer Technology, Pune

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
  public static void main(String args[]) {
    int twoD[][] = new int[4][];
    twoD[0] = new int[1];
    twoD[1] = new int[2];
    twoD[2] = new int[3];
    twoD[3] = new int[4];

    int i, j, k = 0;

    for(i=0; i<4; i++)
      for(j=0; j<i+1; j++) {
      twoD[i][j] = k;
      k++;
    }

    for(i=0; i<4; i++) {
      for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
0
1 2
3 4 5
6 7 8 9
```

Initialization of two dimensional array


```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Initialization of three dimensional array


```
// Demonstrate a three-dimensional array.
class ThreeDMatrix {
  public static void main(String args[]) {
    int threeD[][][] = new int[3][4][5];
    int i, j, k;

    for(i=0; i<3; i++)
      for(j=0; j<4; j++)
        for(k=0; k<5; k++)
          threeD[i][j][k] = i * j * k;

    for(i=0; i<3; i++) {
      for(j=0; j<4; j++) {
        for(k=0; k<5; k++)
          System.out.print(threeD[i][j][k] + " ");
        System.out.println();
      }
      System.out.println();
    }
  }
}
```

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternative Array Declarations


```
type[ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable.

For example, the following two declarations are equivalent:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

Declaring several arrays at the same time,

```
int[] nums, nums2, nums3; // create three arrays
```

It is the same as writing,

```
int nums[], nums2[], nums3[]; // create three array
```

Array as an object

- Arrays are implemented as objects.
- The size of an array that is, the number of elements that an array can hold is found in its **length instance** variable.
- All arrays have this variable, and it will always hold the size of the array.

Example

```
// This program demonstrates the length array member.
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }
}
```

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

String

- String is the most used class in Java's class library.
- Every string you create is an object of type String.
- Even string constants are also String objects.

For example, in the statement

```
System.out.println("This is a String, too");
```

The string "This is a String, too" is a String object.

String Objects

- Every string is an instance of Java's built in String class, thus, Strings are objects.
- Like any object, a string can be created using new as in the following example:

```
String str1 = new String("PICT E&TE");
```

However, as an extra support, Java allows String object to be created without the use of new, as in:

```
String str2="TE 8";
```

This is inconsistent with the way Java treats other classes.

String Construction


```
String myString = "this is a test";
```

Display String

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

Java defines one operator for String objects: +

It is used to concatenate two strings.

```
String myString = "I" + " like " + "Java.";
```

results in **myString** containing

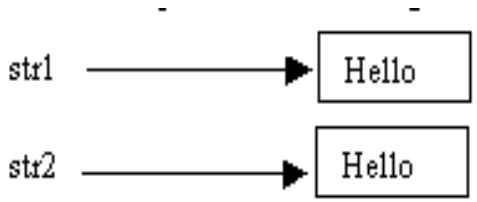
```
"I like Java."
```

String

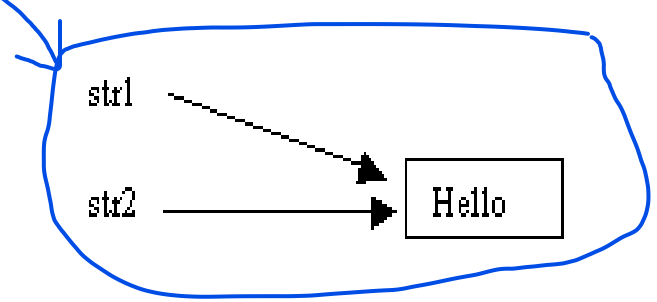
- Objects of type String are **immutable;** once a String object is created, its contents cannot be altered.
- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of String, called StringBuffer and StringBuilder, which allow strings to be altered, so all of the normal string manipulations are still available in Java.
- Once created they cannot be altered and hence any alterations will lead to creation of new string object

E.g.

```
String str1 = “Hello”;
String str2 = “Hello”;
```



But in fact, this is what happens:



String Class

- The String class contains several methods such as:
  - To test two strings for equality by using equals( ).
  - To obtain the length of a string by calling the length( ) method.
  - To obtain the character at a specified index within a string by calling charAt( ).

- The general forms of these three methods are shown here:

```
boolean equals(secondStr)

int length( )

char charAt(index)
```

```
class StringDemo2 {
public static void main(String args[])
{
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1;
System.out.println("Length of strOb1: " + strOb1.length());
System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))
        System.out.println("strOb1 == strOb2");
else
        System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
        System.out.println("strOb1 == strOb3");
else
        System.out.println("strOb1 != strOb3");
}}
```

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Array of Strings


```
// Demonstrate String arrays.
class StringDemo3
{
public static void main(String args[])
 {
String str[] = { "one", "two", "three" };
for(int i=0; i<str.length; i++){
System.out.println("str[" + i + "]: " + str[i]);
 }
}
```

```
output from this program:
str[0]: one
str[1]: two
str[2]: three
```

Strings

- String objects are represented as a sequence of characters indexed from 0.

  Example: **String greeting = "Hello, World";**

- A common operation on Strings is extracting a substring from a given string.
- Java provides two methods for this operation:

substring(start);	Returns the substring from start to the end of the string
substring(start, end);	Returns a substring from start to end but not including the character at end.

Examples:

- **String sub = greeting.substring(0, 2);** "He"
- **String w = greeting.substring(7, 12);** "World"
- **String tail = greeting.substring(7);** "World!"

String

- String s1 = “Example”
- String s2 = new String(“Example”)
- String s3 = “Example”
- The difference between the three statements is that s1 and s3 are pointing to the same memory location i.e., the string pool. s2 is pointing to a memory location on the heap.
- Using a new operator creates a memory location on the heap.
- Concatenating s1 and s3 leads to creation of a new string in the pool.

String Methods

Pune Institute of Computer Technology, Pune

How to call Method	Task performed by method
s1.equalsIgnoreCase(s2)	Returns true if s1=s2,ignoring the case of characters
s1.equals(s2)	Returns 'true' if s1 is equal to s2
s1.length()	Gives length of string s1
s1.compareTo(s2)	Returns negative if s1<s2, positive if s1>s2, and zero if s1 is equal to s2
s2=s1.toLowerCase;	Converts all the elements of string s1 to lowercase
s2=s1.toUpperCase;	Converts all the elements of string s1 to uppercase
s2=s1.replace('A','B');	Replace all appearances of element 'A' by 'B'
s1.concat(s2)	Concatenates s1 and s2

String Constructors

Pune Institute of Computer Technology, Pune

1) String (char [] chars):- Allocates a new string from the given character array

e.g. char chars[]={'P','I','C','T'};

String s =new String(chars);

2) String(char[] chars, int start_Index, int numChars):- Allocates the string from given character array but choose count characters from start index.

e.g. char chars[]={'P','I','C','T'};

String s=new String(chars, 1,3);

3) String(byte ascii[]):- Construct the new string by decoding the byte array

e.g. byte ascii[]={65,66,67,68,69,70};

String s=new String(ascii);

4) String(byte ascii[],int start_index,int length):- Construct the new string from the byte array depending on the start index and length

e.g. . byte ascii[]={65,66,67,68,69,70};

String s=new String(ascii,2,3);

StringBuffer

- StringBuffer supports a modifiable string. (String represents fixed-length, immutable character sequences).
- In contrast, **StringBuffer represents growable and writable character sequences.**
- StringBuffer may have characters and substrings inserted in the middle or appended to the end.
- StringBuffer will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
- StringBuffer is a synchronized and allows to mutate the string.
- StringBuffer has many utility methods to manipulate the string.
- This is more useful when using in a multithreaded environment.
- Always has a locking overhead.

StringBuffer Constructors

1) StringBuffer():- It reserves room for 16 characters without reallocation

   e.g. StringBuffer s =new StringBuffer();

2) StringBuffer(int size):- Allocates Accepts an integer argument that explicitly sets the size of the buffer

   e.g. StringBuffer s =new StringBuffer(20);

3) StringBuffer(String str):- :- It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

   e.g. StringBuffer s=new StringBuffer(“PICT”);

Example

```
package com.examples;

public class MyBuffers {

   public static void main(String[] args) {
       StringBuffer buffer =  new StringBuffer("Hi");
       buffer.append("Bye");
       System.out.println(buffer);
   }
}
```

Methods	**Action Performed**
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method
capacity()	the total allocated capacity can be found by the capacity( ) method
charAt()	specifies the index of the character being obtained
delete()	Deletes a sequence of characters from the invoking object
deleteCharAt()	Deletes the character at the index specified by *loc*
ensureCapacity()	Ensures capacity is at least equals to the given minimum.
insert()	Inserts text at the specified index position
length()	Returns length of the string
reverse()	Reverse the characters within a StringBuffer object
replace()	Replace one set of characters with another set inside a StringBuffer object

StringBuilder

- StringBuilder in Java represents a mutable sequence of characters.
- String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.
- The function of StringBuilder is very much similar to the StringBuffer class, as both provide an alternative to String Class by making a mutable sequence of characters.
- StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.
- StringBuilder more faster than StringBuffer.

Question 1

What is the result of compiling and executing the code?

```
public class Test{
        public static void main(String[] args){
                int[] a = new int[0];
                System.out.print(a.length);
        }
}
```

ANS : 0

Question 2

What is the output of following code?

```
public class Test{
        public static void main(String[] args){
                int[] x = new int[3];
                System.out.println("x[0] is " + x[0]);
        }
}
```

ANS : 0

Question 3

What is the output of following code?

```
public class Test{
        public static void main(String args[]){
                double[] myList = {1, 5, 5, 5, 5, 1};
                double max = myList[0];
                int indexOfMax = 0;
                for(int i = 1; i < myList.length; i++){
                        if(myList[i] > max){
                                max = myList[i];
                                indexOfMax = i;
                        }
                }
                System.out.println(indexOfMax);
        }
}
```

Question 4

Predict the output of following code.

```
public class Test{
        public static void main(String[] args){
                int[] x = {120, 200, 016 };
                for(int i = 0; i < x.length; i++)
                        System.out.print(x[i] + " ");
        }
}
```

Question 5

What is the output?

```
public class Test{
        public static void main(String[] args){
                int[] x = {1, 2, 3, 4};
                int[] y = x;

                x = new int[2];

                for(int i = 0; i < x.length; i++)
                        System.out.print(y[i] + " ");
        }
}
```

ANS : 1 2

Question 6

What is the output?

```
public class Test{
        public static void main(String[] args){
                int[] a = new int[4];
                a[1] = 1;
                a = new int[2];
                System.out.println("a[1] is " + a[1]);
        }
}
```

ANS : 0

Question 7

What is the result of compilation of following code?

```
public class HelloWorld{
      public static void main(String[] args){
            double[] x = new double[]{1, 2, 3};
            System.out.println("Value is " + x[1]);
      }
}
```

ANS : 2.0

Java Package

- A java package is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.

java.util

- java.util is an important package that contains a large assortment of classes and interfaces that support a broad range of functionality.
- For example, java.util has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data.
- The java.util package also contains one of Java's most powerful subsystems: the Collections Framework.
- The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.

Collections Framework

- The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
- Collections were not part of the original Java release, but were added by J2SE 1.2.
- Prior to the Collections Framework, Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects.
- Although these classes were quite useful, they lacked a central, unifying theme.
- The Collections Framework was designed to meet several goals.
  - First, the framework had to be high-performance.
  - Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Third, extending and/or adapting a collection had to be easy.

JAVA Collections

- The Java Collections Framework is a collection of interfaces and classes which helps in storing and processing the data efficiently.
- This framework has several useful classes which have tons of useful functions which makes a programmer task super easy.
- It standardizes the way in which groups of objects are handled by the programs.

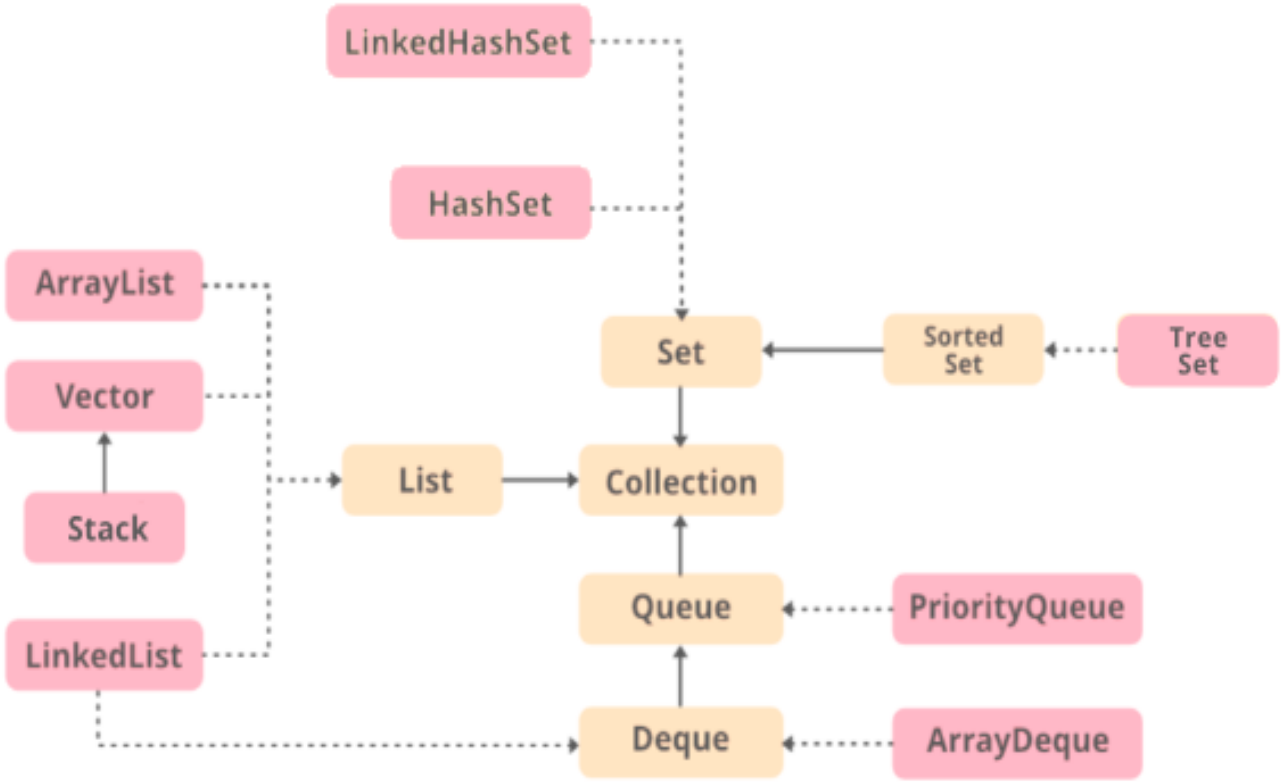
Advantages of Collections

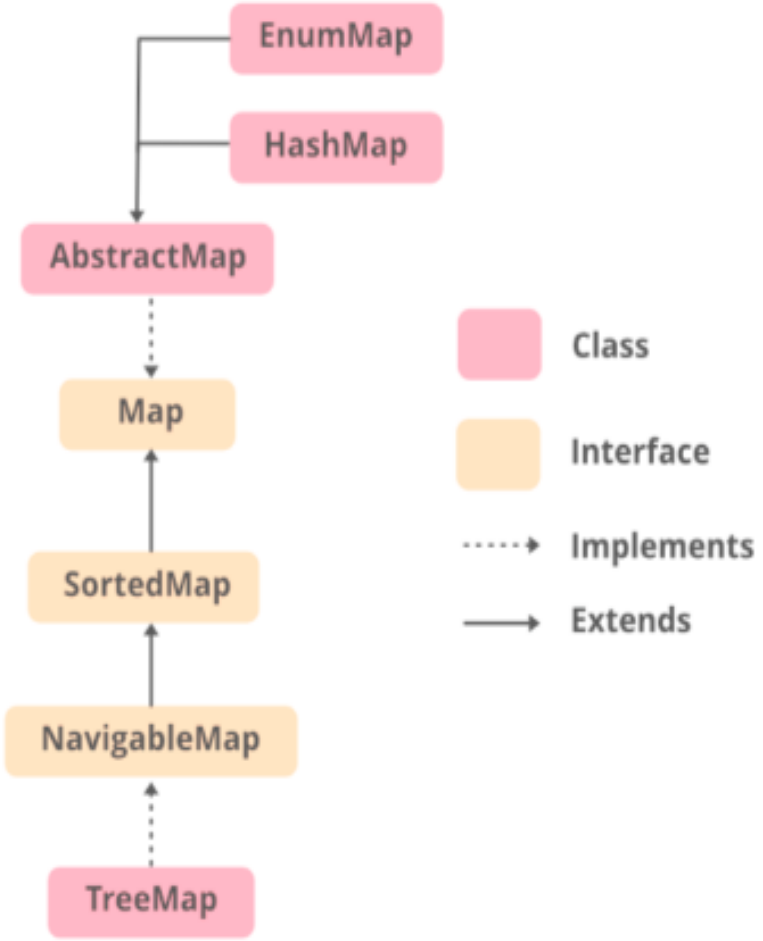
- Reduces programming efforts
- Increases program speed and quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn and to use new APIs
- Reduces effort to design new APIs

Collections Framework

- A collections framework is a unified architecture for representing and manipulating collections.
- All collections frameworks contain the following:
  - ***Interfaces:*** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.
  - ***Implementations:*** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  - ***Algorithms:*** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

Hierarchy of Collections Framework





https://www.geeksforgeeks.org/collections-in-java-2/

ArrayList

- The ArrayList class extends AbstractList and implements the List interface.
- ArrayList is a generic class that has this declaration:

  class ArrayList<E>

- Here, E specifies the type of objects that the list will hold.
- ArrayList supports dynamic arrays that can grow as needed.
- In Java, standard arrays are of a fixed length.
- But, sometimes, you may not know until run time precisely how large an array you need.
- To handle this situation, the Collections Framework defines ArrayList.

ArrayList

- ArrayList inherits AbstractList class and implements the List interface.
- ArrayList is initialized by the size. However, the size is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as a vector in C++.
- ArrayList is not Synchronized. Its equivalent synchronized class in Java is Vector.

ArrayList Constructors

- ArrayList has the constructors shown here:
  - ArrayList( )
  - ArrayList(Collection c)
  - ArrayList(int capacity)
- The first constructor builds an empty array list.
- The second constructor builds an array list that is initialized with the elements of the collection c.
- The third constructor builds an array list that has the specified initial capacity.

```
// Demonstrate ArrayList.
import java.util.*;

class ArrayListDemo {
        public static void main(String args[]) {
        // Create an array list.
        ArrayList<String> al = new ArrayList<String>();
        System.out.println("Initial size of al: " + al.size());
        // Add elements to the array list.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");
        System.out.println("Size of al after additions: " + al.size());
        // Display the array list.
        System.out.println("Contents of al: " + al);
        // Remove elements from the array list.
        al.remove("F");
        al.remove(2);
        System.out.println("Size of al after deletions: " + al.size());
        System.out.println("Contents of al: " + al);
}
}
```

SYNTAX

Delete element at index 2

Initial size of al: 0

Size of al after additions: 7

Contents of al: [C, A2, A, E, B, D, F]

Size of al after deletions: 5

Contents of al: [C, A2, E, B, D]

Obtaining Array from ArrayList

- When working with ArrayList, sometimes we may want to obtain an actual array that contains the contents of the list.
- This can be done by calling toArray( ), which is defined by Collection.
- Several reasons exist why we might want to convert a collection into an array, such as:
  - To obtain faster processing times for certain operations
  - To pass an array to a method that is not overloaded to accept a collection
  - To integrate collection-based code with legacy code that does not understand collections

```java
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
        // Create an array list.
        ArrayList<Integer> al = new ArrayList<Integer>();
        // Add elements to the array list.
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);
        System.out.println("Contents of al: " + al);
        // Get the array.
        Integer b[] = new Integer[al.size()];
        b = al.toArray(b);
        int sum = 0;
        // Sum the array.
        for (int i : b)
        sum += i;
        System.out.println("Sum is: " + sum);
}
}
```

Output:
Contents of al: [1, 2, 3, 4]
Sum is: 10

Inheritance

- Inheritance is an important pillar of Object Oriented Programming.
- It is the mechanism by which one class is allowed to inherit the features (fields and methods) of another class.
- Using inheritance, a general class can be created that defines traits common to a set of related items.
- This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

Example

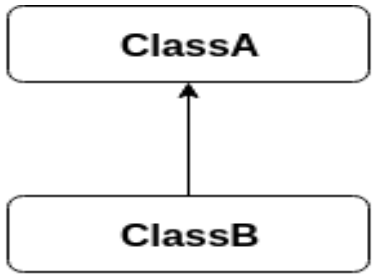
Pune Institute of Computer Technology, Pune

```
// Create a superclass
class Super {
.....
.....
}
// Create a subclass by extending class super
class Sub extends Super {
.....
.....
}
```

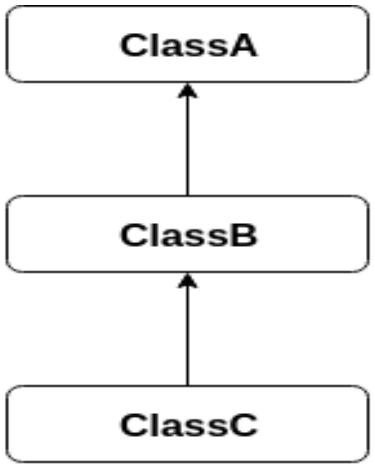
Types of Inheritance

1. *Single Inheritance*
2. *Hierarchical Inheritance*
3. *Multilevel Inheritance*
4. *Hybrid Inheritance*
5. *Multiple Inheritance*

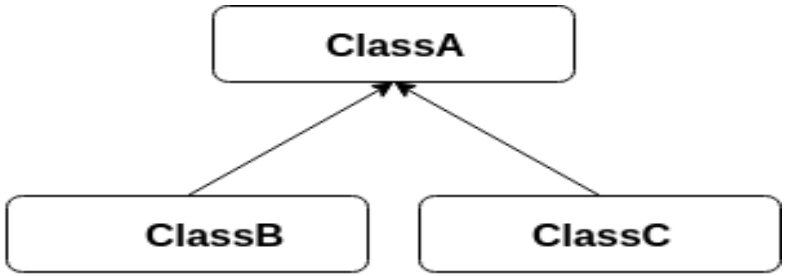
Types of Inheritance



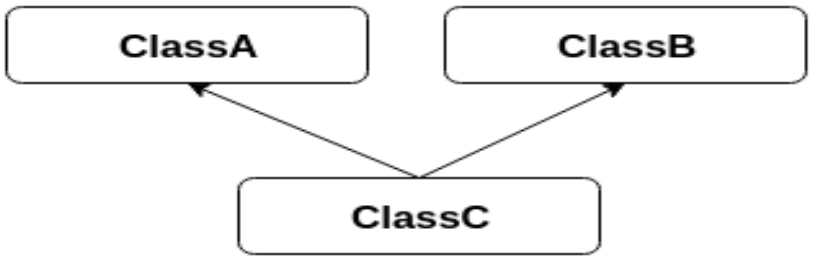
Single Inheritance



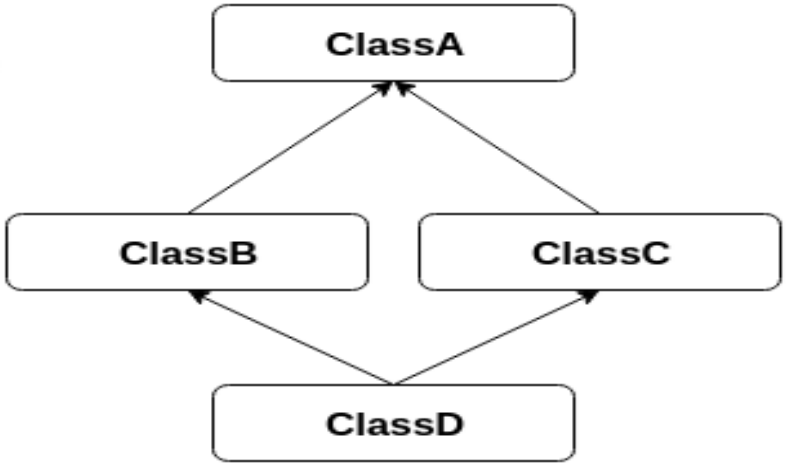
Multilevel Inheritance



Hierarchical Inheritance



Multiple Inheritance



Hybrid Inheritance

https://www.javatpoint.com/typescript-inheritance

Types of Inheritance

- ***Based on class, there can be three types of inheritance in java: single, multilevel and hierarchical.***
- ***In java programming, multiple and hybrid inheritance is supported through interface only.***

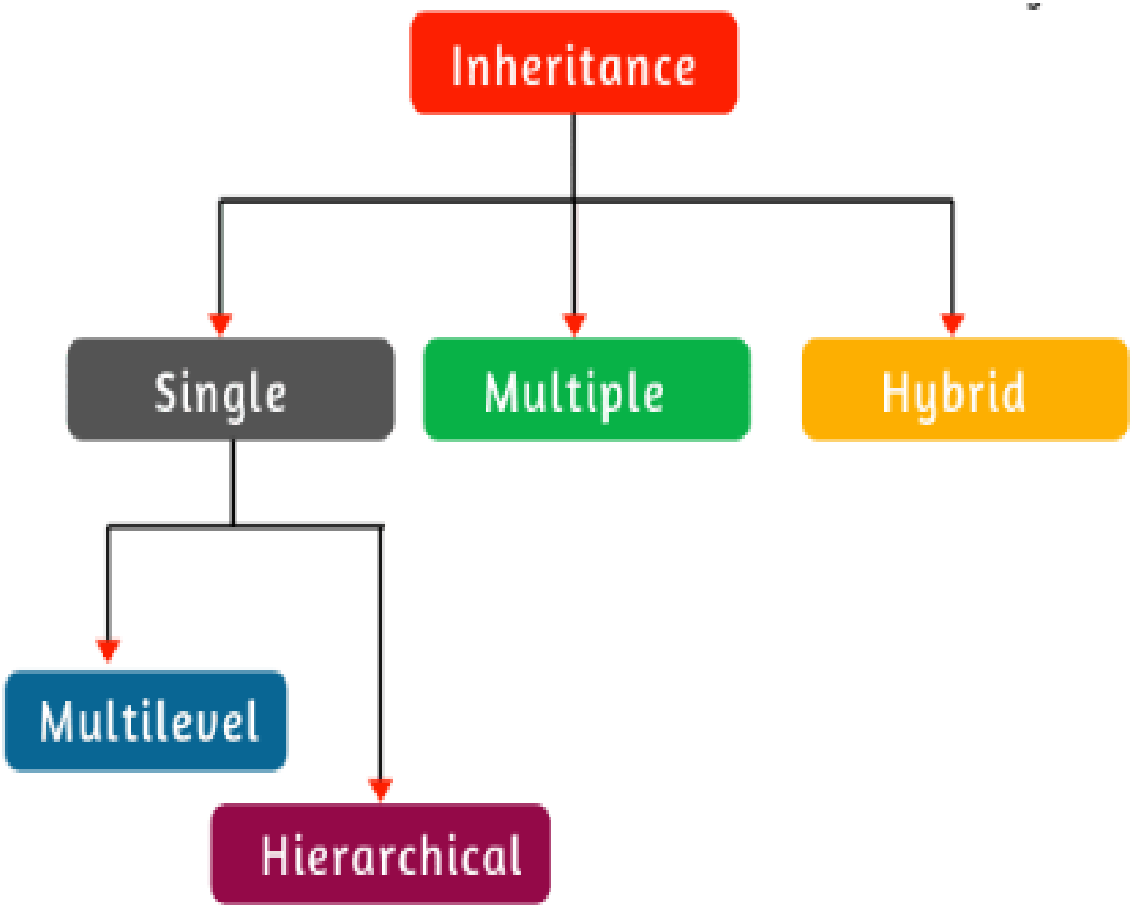


Fig: Classification of Inheritance in Java

Single Inheritance


```
class A
{
    public void methodA()
    {
      System.out.println("Base class method");
    }
}
class B extends A
{
    public void methodB()
    {
      System.out.println("Child class method");
    }
}
class InheritanceDemo
{
    public static void main(String args[])
    {
      B obj = new B();
      obj.methodA(); //calling super class method
      obj.methodB(); //calling local method
   }
}
```

Hierarchical Inheritance


```
class A
{
    public void methodA()
    {
        System.out.println("method of Class A");
    }
}
class B extends A
{
    public void methodB()
    {
        System.out.println("method of Class B");
    }
}
class C extends A
{
   public void methodC()
   {
      System.out.println("method of Class C");
   }
}
```

Continued…


```
class D extends A
{
  public void methodD()
  {
     System.out.println("method of Class D");
  }
}
class JavaExample
{
  public static void main(String args[])
  {
     B obj1 = new B();
     C obj2 = new C();
     D obj3 = new D();
     //All classes can access the method of class A
     obj1.methodA();
     obj2.methodA();
     obj3.methodA();
  }
}
```

```
method of Class A
method of Class A
method of Class A
```

Multilevel Inheritance

Pune Institute of Computer Technology, Pune

```
class X
{
   public void methodX()
   {
     System.out.println("class X method");
   }
}
class Y extends X
{
        public void methodY()
        {
                System.out.println("class Y method");
        }
}
class Z extends Y
{
   public void methodZ()
   {
     System.out.println("class Z method");
   }
}
```

Continued…


```
class MultiDemo
{
        public static void main(String args[])
        {
        Z obj = new Z();
        obj.methodX(); //calling grand parent class method
        obj.methodY(); //calling parent class method
        obj.methodZ(); //calling local method
        }
}
```

Output:

```
        class X method
        class Y method
        class Z method
```

Terminologies

- **Super Class:** The class whose features are inherited is known as superclass (or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass (or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e., when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Inheritance in JAVA

Default superclass: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.

Superclass can only be one: A superclass can have any number of subclasses. But a subclass can have only one superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by java.

Inheriting Constructors: 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.

Private member inheritance: A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (like getters and setters) for accessing its private fields, these can also be used by the subclass.

Subclasses

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Example

```
class A
{
        int i, j;
        void showij() {
                System.out.println("i and j: " + i + " "+ j);
        }
}

class B extends A
 {
        int k;
        void showk() {
                System.out.println("k: " + k);
        }

        void sum() {
                System.out.println("i+j+k: " + (i+j+k));
        }
}
```

Example Ctd…


```
class SimpleInheritance {
        public static void main(String args []) {
        A superOb = new A();
        B subOb = new B();
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();
        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
}   }
```

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

Member Access and Inheritance

Pune Institute of Computer Technology, Pune

```
/* In a class hierarchy, private members remain
   private to their class.

   This program contains an error and will not
   compile.
*/

// Create a superclass.
class A {
  int i; // public by default
  private int j; // private to A

  void setij(int x, int y) {
    i = x;
    j = y;
  }
}

// A's j is not accessible here.
class B extends A {
  int total;

  void sum() {
    total = i + j; // ERROR, j is not accessible here
  }
}
```

```
class Access {
  public static void main(String args[]) {
    B subOb = new B();

    subOb.setij(10, 12);

    subOb.sum();
    System.out.println("Total is " + subOb.total);
  }
}
```

A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

Example


```
// This program uses inheritance to extend Box.
class Box {
  double width;
  double height;
  double depth;

  // construct clone of an object
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use  -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
// Here, Box is extended to include weight.
class BoxWeight extends Box {

double weight; // weight of box

// constructor for BoxWeight
BoxWeight(double w, double h, double d, double m) {
  width = w;
  height = h;
  depth = d;
  weight = m;
}
}

class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;

    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    System.out.println("Weight of mybox1 is " + mybox1.weight);
    System.out.println();

    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    System.out.println("Weight of mybox2 is " + mybox2.weight);
  }
}
```

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

Reference of a Subclass Object to Superclass Variable

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("Volume of weightbox is " + vol);
    System.out.println("Weight of weightbox is " +
                       weightbox.weight);
    System.out.println();

    // assign BoxWeight reference to Box reference
    plainbox = weightbox;

    vol = plainbox.volume(); // OK, volume() defined in Box
    System.out.println("Volume of plainbox is " + vol);

    /* The following statement is invalid because plainbox
       does not define a weight member. */
//  System.out.println("Weight of plainbox is " + plainbox.weight);
  }
}
```

Reference of a Subclass Object to Superclass Variable

- Here, **weightbox** is a reference to BoxWeight objects, and **plainbox** is a reference to Box objects. Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.
- It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.
- This is why plainbox can’t access weight even when it refers to a BoxWeight object.

‘super’ Keyword

- In the preceding examples, classes derived from Box were not implemented as efficiently or as robustly as they could have been. For example, the constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box.
- Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members.
- However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private).
- In this case, there would be no way for a subclass to directly access or initialize these variables on its own.

‘super’ Keyword

- Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

- super has two general forms.
  - The first calls the superclass’ constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Use of super to Call Superclass Constructors

A subclass can call a constructor defined by its superclass by use of the following form of super:

```
super(arg-list);
```

- Here, arg-list specifies any arguments needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass' constructor.
- To see how super( ) is used, consider this improved version of the BoxWeight class:

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box{
        double weight; // weight of box
        // initialize width, height, and depth using super()
        BoxWeight(double w, double h, double d, double m) {
                super(w, h, d); // call superclass constructor
                weight = m;
        }
}
```

Use of ‘super’ with members

- The second form of super always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:

```
super.member
```

- Here, member can be either a method or an instance variable.
- This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Use of ‘super’ with members

Pune Institute of Computer Technology, Pune

```
// Using super to overcome name hiding.
class A {
        int i;
}
// Create a subclass by extending class A.
class B extends A {
        int i; // this i hides the i in A
        B(int a, int b) {
             super.i = a; // i in A
              i = b; // i in B
        }
void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
        }
}
class UseSuper {
        public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
}}
```

```
i in superclass: 1
i in subclass: 2
```

Use of ‘super’ with methods


```
class Person{
        void message(){
                System.out.println("This is person class");
        }
}
class Student extends Person{
        void message(){
                System.out.println("This is student class");
        }
        void display(){
                message();
                super.message();
        }
}
class Test{
        public static void main(String args[]){
                Student s = new Student();
                s.display();
        }
}
```

```
This is student class
This is person class
```

Important Points about ‘super’

- Call to super() must be first statement in Derived Class constructor.
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object does have such a constructor, so if Object is the only superclass, there is no problem.
- If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called **constructor chaining.**

Execution of Constructors

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed?

The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used.

Execution of Constructors

Pune Institute of Computer Technology, Pune

```
//Create a super class.
class A
{
        A()
        {
            System.out.println("Inside A's constructor.");
        }
}
//Create a subclass by extending class A.
class B extends A
{
B()
{
System.out.println("Inside B's constructor.");
}
}
//Create another subclass by extending B.
class C extends B
{
C()
{
System.out.println("Inside C's constructor.");
}
}
```

```
class CallingCons
{
public static void main(String args[])
{
C c = new C();
}
}
```

```
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

Method Overriding

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

Example


```
class A {
   int i, j;
   A(int a, int b) {
   i = a; j = b;
}
void show() {
   System.out.println("i and j: " + i + " " + j);
   }
}
class B extends A {
   int k;
   B(int a, int b, int c) {
         super(a, b);
         k = c;
   }
   void show() {
         System.out.println("k: " + k);
   }
}
```

```
class Override
public static void main(String args[]){
   B subOb = new B(1, 2, 3);
   subOb.show();
   }
}
```

k: 3

When show() is invoked on an object of type B, the version of show() defined in B is used.

The version of show() in A is hidden through overriding.

Method Overriding

If you wish to access the superclass version of an overridden method, you can do so by using super.

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

i and j: 1 2
k: 3
```

Method Overriding

- Method overriding occurs only when the names and the type signatures of two methods are identical. If they are not, then the two methods are simply overloaded.

```
// Methods with differing type signatures are overloaded - not
// overridden.
class A {
  int i, j;

  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

// Create a subclass by extending class A.
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // overload show()
  void show(String msg) {
    System.out.println(msg + k);
  }
}
```

```
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

    subOb.show("This is k: "); // this calls show() in B
    subOb.show(); // this calls show() in A
  }
}
```

The output produced by this program is shown here:

```
This is k: 3
i and j: 1 2
```

Method Overloading vs Overriding

Pune Institute of Computer Technology, Pune

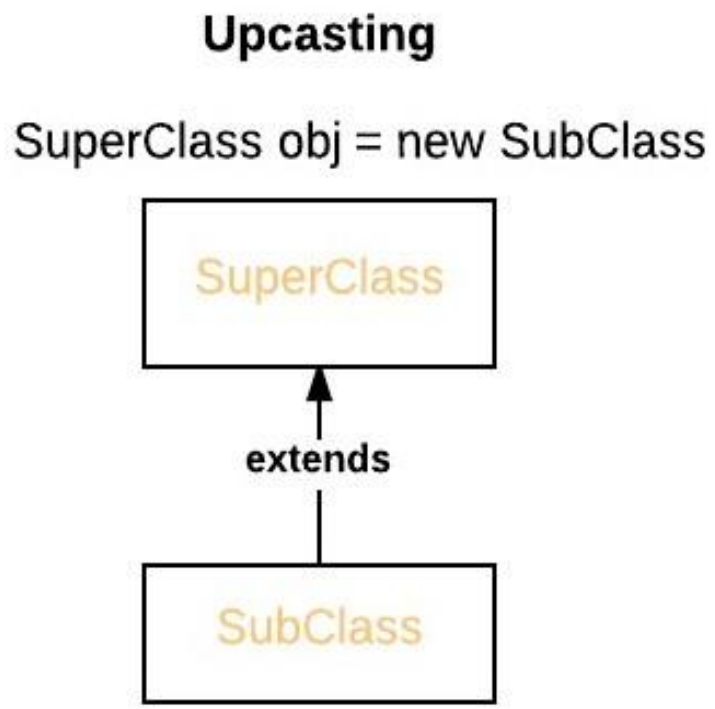
Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
It helps to increase the readability of the program.	It is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or can not be the same.	In method overriding, the return type must be the same or co-variant.
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
It gives better performance.	Poor performance
Private and final methods can be overloaded.	Private and final methods can’t be overridden.
Argument list should be different while doing method overloading.	Argument list should be same in method overriding.

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.
- When an overridden method is called through a superclass reference, Java determines which version (superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- This determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.

Upcasting

If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.



Example


```
// Dynamic Method Dispatch
class A {
        void callme() {
                System.out.println("Inside A's callme method");
        }
}
class B extends A {
        void callme() {
                System.out.println("Inside B's callme method");
        }
}
class C extends A {
        void callme() {
                System.out.println("Inside C's callme method");
        }
}
```

```
class Dispatch {
        public static void main(String args[]) {
                A a = new A(); // object of type A
                B b = new B(); // object of type B
                C c = new C(); // object of type C

                A r; // obtain a reference of type A
                r = a; // r refers to an A object
                r.callme(); // calls A's version of callme
                r = b; // r refers to a B object
                r.callme(); // calls B's version of callme
                r = c; // r refers to a C object
                r.callme(); // calls C's version of callme
        }
}
```

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

```
// Using run-time polymorphism.
class Figure {
        double dim1;
        double dim2;
        Figure(double a, double b){
                dim1 = a;
                dim2 = b;
        }
        double area(){
                System.out.println("Area for Figure is undefined.");
                return 0;
        }
}
class Rectangle extends Figure{
        Rectangle(double a, double b) {
                super(a, b);
        }
        double area(){
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
        }
```

```
class Triangle extends Figure {
        Triangle(double a, double b) {
                super(a, b);
        }
        double area(){
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class FindAreas {
        public static void main(String args[]){
                Figure f = new Figure(10, 10);
                Rectangle r = new Rectangle(9, 5);
                Triangle t = new Triangle(10, 8);
                Figure figref;
                figref = r;
                System.out.println("Area is " + figref.area());
                figref = t;
                System.out.println("Area is " + figref.area());
                figref = f;
                System.out.println("Area is " + figref.area());
        }
}
```

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from Figure, then its area can be obtained by calling area( ).

Question 1

```
class A{
	private void printName(){
		System.out.println("Value-A");
	}
}
class B extends A{
	public void printName(){
		System.out.println("Name-B");
	}
}
public class Test{
	public static void main (String[] args){
		B b = new B();
		b.printName();
	}
}
```

Name-B

Question 2

Pune Institute of Computer Technology, Pune

```
class A{
	int b=10;
	private A(){
		this.b=7;
	}
	int f(){
		return b;
	}
}
class B extends A{
	int b;
}
public class Test{
	public static void main(String[] args){
		A a = new B();
		System.out.println(a.f());
	}
}
```

Compilation Fails

Question 3

Pune Institute of Computer Technology, Pune

```
class Small{
	public Small(){
		System.out.print("a ");
	}
}
class Small2  extends Small{
	public Small2(){
		System.out.print("b ");
	}
}
class Small3 extends Small2{
	public Small3(){
		System.out.print("c ");
	}
}
public class Test{
	public static void main(String args[]){
		new Small3();
	}
}
```

a b c

Question 4


```
class One{
        final int a = 15;
}

class Two extends One{
        final int a = 20;
}

public class Test extends Two{
        final int a = 30;

        public static void main(String args[]){
                Test t = new One();
                System.out.print(t.a);
        }
}
```

Compiler Error

Question 5

```
class A{
	int i = 10;
	public void printValue(){
		System.out.print("Value-A");
	}
}

class B extends A{
	int i = 12;
	public void printValue(){
		System.out.print("Value-B");
	}
}

public class Test{
	public static void main(String args[]){
		A a = new B();
		a.printValue();
		System.out.print(a.i);
	}
}
```

Value-B 10

Question 6

```
class Parent{
	public void method(){
		System.out.println("Hi i am parent");
	}
}
public class Child extends Parent{
	protected void method(){
		System.out.println("Hi i am Child");
	}
	public static void main(String args[]){
		Child child = new Child();
		child.method();
	}
}
```

Compile time error

final Keyword

Pune Institute of Computer Technology, Pune

Final keyword in Java is used to restrict the access of an item from its super class to subclass

1. Using *final* to prevent overriding:-

   methods declared as final cannot be overridden

   variable cannot be access in subclass

2. Using *final* to prevent inheritance:-

   class declared as final cannot be inherited

Hey, I am final !
You can not change my value
You can not override me
You can not inherit me

Wrapper Classes

- A Wrapper class is a class whose object wraps or contains primitive data types.
- When an object to a wrapper class is created, it contains a field and, primitive data types can be stored in this field.

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

Necessity

1. They convert primitive data types into objects. Objects are needed to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Autoboxing and Unboxing

- Autoboxing: Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.
- For example – conversion of int to Integer, long to Long, double to Double etc.
- Unboxing: It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.
- For example – conversion of Integer to int, Long to long, Double to double, etc.

enum in JAVA

- Representation of group of named constants.
- Generally used when all possible values are known.
- A Java enumeration is a class type.
- An enum is not instantiated using new, but it has the same capabilities as other classes.
- This makes Java enumeration a very powerful tool.
- We can have constructor, add instance variables and methods, and even implement interfaces in enums.
- Unlike classes, enumerations neither inherit other classes nor can get extended (i.e become superclass).
- Enum declaration can be done outside a Class or inside a Class but not inside a Method.

Example


```
enum Color {
        RED,
        GREEN,
        BLUE;
}

public class Test {
        // Driver method
        public static void main(String[] args)
        {
                Color c1 = Color.RED;
                System.out.println(c1);
        }
}
```

Features

- Every enum is internally implemented by using Class.
- Every enum constant represents an object of type enum.
- enum type can be passed as an argument to switch statements.

```
/* internally above enum Color is converted to
class Color
{
     public static final Color RED = new Color();
     public static final Color BLUE = new Color();
     public static final Color GREEN = new Color();
}*/
```

Features

- Every enum constant is always implicitly public static final. Since it is static, we can access it by using the enum Name. Since it is final, we can't create child enums.
- We can declare the main() method inside the enum. Hence, we can invoke enum directly from the Command Prompt.

```
enum Color {
        RED,
        GREEN,
        BLUE;

        // Driver method
        public static void main(String[] args)
        {
                Color c1 = Color.RED;
                System.out.println(c1);
        }
}
```

Features

- All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.
- enum and constructor:
  - enum can contain a constructor and it is executed separately for each enum constant at the time of enum class loading.
  - We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.
- enum and methods:
  - enum can contain both concrete methods and abstract methods. If an enum class has an abstract method, then each instance of the enum class must implement it

Question 1

```
enum Levels
{
    private TOP,

    public MEDIUM,

    protected BOTTOM;
}
```

a) Runtime Error

b) Enum Not Defined Exception

c) It runs successfully

d) Compilation Error

Question 2


```
enum Season
{
    WINTER, SPRING, SUMMER, FALL
};
System.out.println(Season.WINTER.ordinal());
```

a) 0

b) 1

c) 2

d) 3

```
class A
{

}

enum Enums extends A
{
    ABC, BCD, CDE, DEF;
}
```

a) Runtime Error

b) Compilation Error

c) It runs successfully

d) Enum Not Defined Exception

```
enum Enums
{
    A, B, C;

    private Enums()
    {
        System.out.println(10);
    }
}

public class MainClass
{
    public static void main(String[] args)
    {
        Enum en = Enums.B;
    }
}
```

a)

10

10

10

b) Compilation Error

c)

10

10

d) Runtime Exception

enum Methods

values(), ordinal() and valueOf() methods:

- These methods are present inside **java.lang.Enum**.
- **values() method** can be used to return all values present inside the enum.
- Order is important in enums.By using the **ordinal() method**, each enum constant index can be found, just like an array index.
- **valueOf() method** returns the enum constant of the specified string value if exists.

Abstraction in JAVA

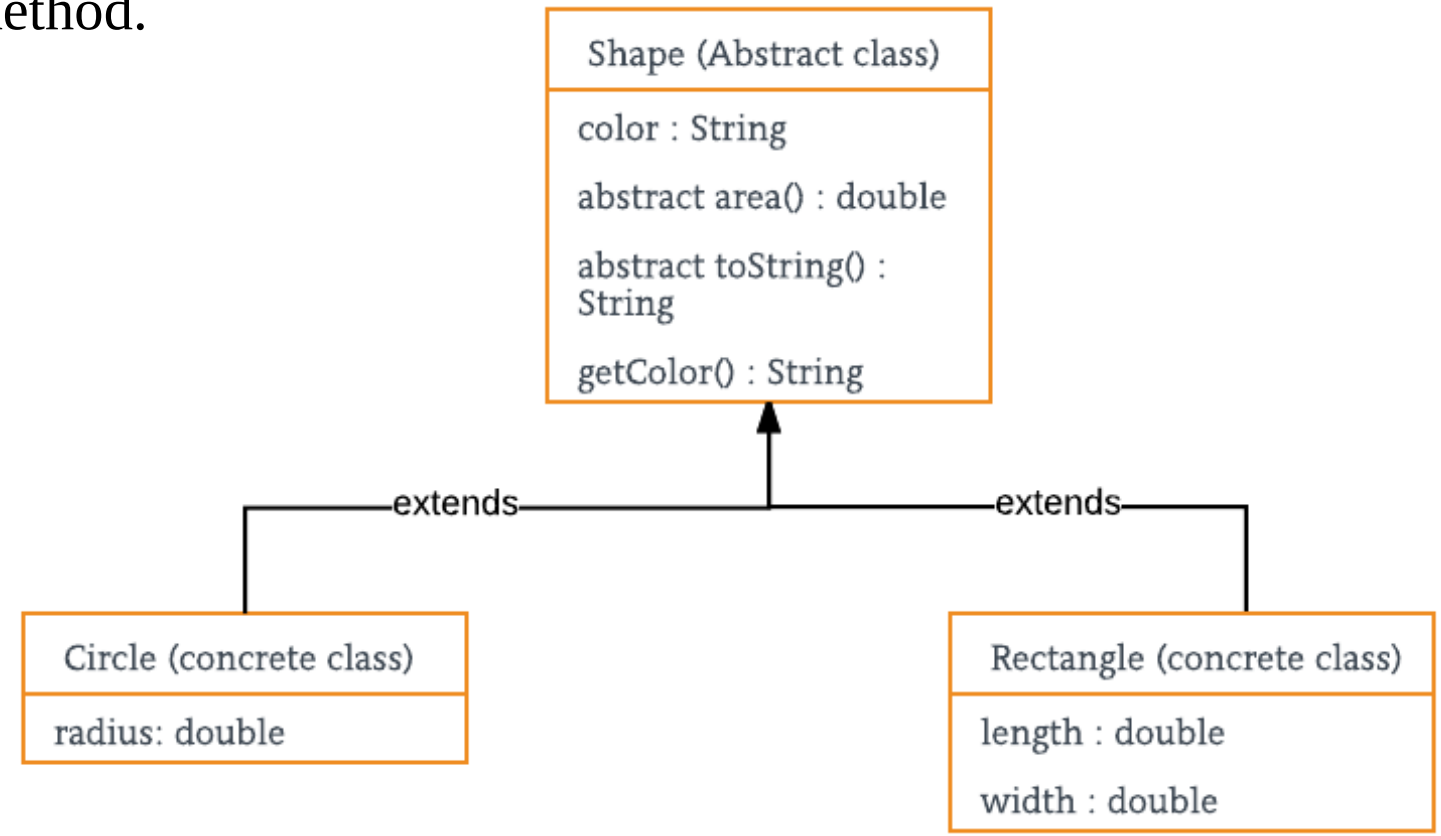
- Data Abstraction is the property by which only the essential details are displayed to the user. The trivial or the non-essential units are not displayed to the user.
- Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.
- The properties and behaviors of an object differentiate it from other objects of similar type and help in classifying/grouping the objects.

Abstract classes and Abstract methods

- An abstract class is a class that is declared with an abstract keyword.
- An abstract method is a method that is declared without implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods.
- A method-defined abstract must always be redefined in the subclass, thus making overriding compulsory or making the subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

Example

A superclass declaration with structure of a given abstraction without providing a complete implementation of every method.



Advantages of Abstraction

- It reduces the complexity of viewing things.
- Avoids code duplication and increases reusability.
- Helps to increase the security of an application or program as only essential details are provided to the user.
- It improves the maintainability of the application.
- It improves the modularity of the application.

Abstraction vs Encapsulation

- While encapsulation groups together data and methods that act upon the data, data abstraction deal with exposing the interface to the user and hiding the details of implementation.
- Encapsulated classes are java classes that follow data hiding and abstraction while abstraction can be implemented by using abstract classes and interfaces.
- Encapsulation is a procedure that takes place at the implementation level, while abstraction is a design-level process.

Abstract Classes

- An instance of an abstract class can not be created.
- Constructors are allowed.
- We can have an abstract class without any abstract method.
- There can be a final method in abstract class but any abstract method in class (abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”
- We can define static methods in an abstract class.
- We can use the abstract keyword for declaring top-level classes (Outer class) as well as inner classes as abstract.
- If a class contains at least one abstract method then compulsory should declare a class as abstract.
- If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method.

Using Abstract Classes

- You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.
- These methods are sometimes referred to as subclass responsibility because they have no implementation specified in the superclass.
- Thus, a subclass must override them—it cannot simply use the version defined in the superclass.
- To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

- As you can see, no method body is present.
- It is not possible to instantiate an abstract class.

```
 abstract class Bike
{
  abstract void run();
}
  class Honda extends Bike
{
  void run()
    {
System.out.println("running safely..");
     }
}
```

```
public class Test {
    public static void main(String args[])
    {
       Bike obj = new Bike();
      Bike obj = new Honda();
       obj.run();
     }
    }
```

Output:- running safely

```
abstract class Bike
{
  Bike()
  {
   S.o.p(“Bike Starts”);
    }
   abstract void run();
   void changeGear()
  {
      S.o.p(“Gear Changed”);
   }
}
 class Honda extends Bike
{
 void run()
  {
      System.out.println("running safely..");
  }
}
public class Test {
   public static void main(String args[])
   {
    Bike obj = new Honda();
     obj.run();
     obj.changeGear();
    }
   }
```

Output:-Bike Starts
Running Safely
Gear Changed

Abstract classes and Abstract methods

Abstract classes	Abstract methods
Abstract classes can’t be instantiated.	Abstract method bodies must be empty.
Other classes extend abstract classes.	Sub-classes must implement the abstract class’s abstract methods.
Can have both abstract and concrete methods.	Has no definition in the class.
Similar to interfaces, but can • Implement methods • Fields can have various access modifiers • Subclasses can only extend one abstract class	Has to be implemented in a derived class.

References

- E Balagurusamy, “Programming with JAVA”, Tata McGraw Hill, 6th Edition.
- Herbert Schildt, “Java: The complete reference”, Tata McGraw Hill, 7th Edition.
- https://www.tutorialspoint.com
- https://www.geeksforgeeks.org