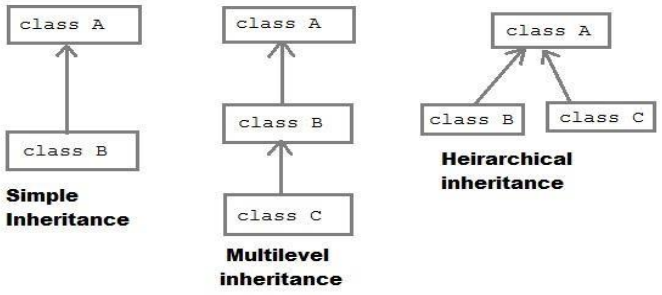
Inheritance:

- Acquiring the properties from one class to another class is called inheritance (or) creating new class from already existing class is called inheritance.
- Existing class is called super class and created new class is called sub class.
- **Reusability of code** is main advantage of inheritance.
  In Java inheritance is achieved by using **extends** keyword.



- Java does not support **multiple inheritance** and **hybrid inheritance.**

```
class Parent            // Super Class
{
        String parentName;
        String familyName;
}
class Child extends Parent    // Sub Class
{
        String childName;
        int childAge;
        void printMyName()
        {
                System.out.println ("My name is"+childName+" "+familyName);
        }
}
```

- In this example Parent is the super class and child is the sub class.
- Super class default constructor is available to sub class by default.
- First super class default constructor is executed then sub class default constructor is executed.
- Super class parameterized constructor is not automatically available to subclass.

Super keyword:

- super can be used to refer super class **variables** as: super.variablename.
- super can be used to refer super class **methods** as: super.methodname ().
- super can be used to refer super class **constructor** as: super (values).

Example:

```
class A
{
        int x;
        A (int x)
        {
                this.x = x;
        }
        void show( )
        {
        System.out.println("super class method: x = "+x);
        }
}
class B extends A
{
        int y;
        B (int a,int b)
        {
                super(a); // (or) x=a;
                y=b;
        }
        void show( )
        {
                super.show ();
                System.out.println ("y = "+y);
                System.out.println (" super x = " + super.x);
        }
}
class SuperUse
{
        public static void main(String args[])
        {
                B ob = new B (10, 24);
                ob.show ( );
        }
}
```

- Super key word is used in **sub class** only.
- The statement calling super class constructor should be the first one in sub class constructor.

Method overriding:

- ✓ If subclass (child class) has the same method as declared in the parent class, it is known as method overriding.

Advantage of Method Overriding:

- ✓ Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- ✓ Method Overriding is used for Runtime Polymorphism.

Rules for Method Overriding:

- ✓ Method must have same name as in the parent class
- ✓ Method must have same parameter as in the parent class.
- ✓ Must be IS-A relationship (inheritance).

```
class Animal
{
        void move()
        {
        System.out.println ("Animals can move");
        }
}
class Dog extends Animal
{
   void move()
   {
   System.out.println ("Dogs can walk and run");
   }
}
public class OverRide
{
        public static void main(String args[])
        {
        // Animal reference and object
           Animal a = new Animal ();

        // Animal reference but Dog object
        Animal b = new Dog ();

a.move (); // runs the method in Animal class
b.move (); //Runs the method in Dog class
        }
}
```

Variable shadowing:

- ✓ Same variable declare and sub class which present in super class then sub class variable called shadowing variable.

```
class A
{
        int i=10;
}
Class B extends A
{
        int i=20; // shadowing variable.
}
```

---

final keyword:

- ✓ We can use final keyword with variable, methods or class.
- ✓ final keyword before a class prevents inheritance.
  - o e.g.: final class A
  - o class B extends A //invalid
- ✓ final keyword before a method prevents overriding.

final variable : We can not change the value once it assigned.

example :

o final int i = 100;
o final double PI = 3.14;

final method : It is used to restrict overriding.

example :

```
class A {
        final void display(){}
}
class B extends A {
/* Overriding is not possible because overridden
method is final
void display(){}
*/
}
```

final class : We can not inherit from final class.

e.g.

```
final class A {
}
class B extends A /* It gives an error cannot
inherit from final class */
{
}
```

Abstract:

- The abstract methods and abstract class should be declared using the keyword abstract.
- We **cannot create objects** to abstract class because it is having incomplete code.
- Whenever an abstract class is created, subclass should be created to it and the abstract methods should be implemented in the subclasses, then we can create objects to the subclasses.
- An abstract class is may contains abstract or non-abstract methods.
- An abstract class contains instance variables & concrete methods in addition to abstract methods.
- It is not possible to create objects to abstract class. But we can **create a reference** of abstract class type.
- All the abstract methods of the abstract class should be implemented in its sub classes.
- If any method is not implemented, then that sub class should be declared as 'abstract'.
- Abstract class reference can be used to refer to the objects of its sub classes.
- Abstract class references cannot refer to the individual methods of sub classes.
- A class cannot be both 'abstract' & 'final'.
    - e.g.: final abstract class A // invalid.

Example of Runtime Polymorphism

```
abstract class RBI
{
        abstract float getRate();
}

class AXIS extends RBI
{
        float getRate(){
                return 9.2;
        }
}
class SBI extends RBI
{
        float getRate(){
                return 8.5;
        }
}
class TestBank{
        public static void main(String []args){
                RBI r = new SBI();
                System.out.println(r.getRate());
                r = new AXIS();
                System.out.println(r.getRate());
        }}
```

- Here in above example if RBI points to SBI, method of SBI will be execute and if it points to AXIS, method of AXIS will be execute.
- It is also known as Run-time polymorphism or Dynamic Method Dispatch.

Interface:

- Interface is used to achieve total abstraction.
- An interface is a specification of method prototypes.
- All the methods in an interface are abstract methods.
- All the methods of interface are public, abstract by default.
- An interface may contain variables which are by default public static final.
- All the methods of the interface should be implemented in its implementation classes.
- If any one of the methods is not implemented, then that implementation class should be declared as abstract.
- We cannot create an object to an interface.
- We can create a reference variable to an interface.
- An interface cannot implement another interface.
- An interface can extend another interface.
- A class can implement multiple interfaces.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

Example

```
// A simple interface
interface Player
{
        final int id = 10;
        int move();
}

// Java program to demonstrate working of
//interface.

interface MyInterface
{
        // public, static and final
        final int a = 10;

        // public and abstract
        void display();
}
```

```
// A class that implements interface.
class testClass implements MyInterface
{
        // Implementing the capabilities of
        // interface.
        public void display()
        {
                System.out.println("Atmiya");
        }

        // Driver Code
        public static void main (String[] args)
        {
                testClass t = new testClass();
                t.display();
                System.out.println(a);
        }
}
```

Package:

- ✓ A package is a container of related classes and interfaces.
- ✓ A package represents a directory that contains related group of classes and interfaces.

General form for creating a package:
package packagename;
e.g.: package pack;

- ✓ The first statement in the program must be package statement while creating a package.
- ✓ While creating a package except instance variables, declare all the members and the class itself as public then only the public members are available outside the package to other programs.

```
package pack;
public class Addition
{
        private double d1,d2;
        public Addition(double a, double b)
        {
                d1 = a;
                d2 = b;
        }
        public void sum()
        {
                System.out.println ("Sum of two
given numbers is : " + (d1+d2) );
        }
}
class Test{
        public static void main(String []args){
          Addition a = new Addition(12.5,12.5);
          a.sum();
        }
}
```

To Execute Package

- Create one folder myproj and then create 2 subfolders src and classes.
- Store above Addition.java in src folder.

```
D:\MyProj\src>javac -d ..\classes Addition.java

D:\MyProj\src>java -cp ..\classes pack.Test
Sum of two given        numbers is : 25.0
```

- -d is used to generate package directory of class files on mentioned path.
- -cp will search for class files in mentioned directory.

Access Specifier:

- ✓ Specifies is the scope of the data members, class and methods.
- ✓ **private members** of the class are available within the **class only.** The scope of private members of the class is "CLASS SCOPE".
- ✓ **public members** of the class are available **anywhere.** The scope of public members of the class is "GLOBAL SCOPE".
- ✓ **default members** of the class are available **within the class, outside the class and in its sub class of same package.** It is not available outside the package. So the scope of default members of the class is "PACKAGE SCOPE".
- ✓ **protected members** of the class are available **within the class, outside the class and in its sub class of same package and also available to subclasses in different package.**

Class member Access	private	default	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

Example:

Write a program to create class A with different access specifiers.

```
//create a package same
package same;
public class A
{
        private int a=1;
        public int b = 2;
        protected int c = 3;
        int d = 4;
}
```

Write a program for creating class B in the same package.

```
//class B of same package
package same;
public class B
{
        public static void main(String args[])
        {
                A obj = new A();
                System.out.println(obj.a);
                System.out.println(obj.b);
                System.out.println(obj.c);
                System.out.println(obj.d);
        }
}
```

Compiling the above program

```
D:\MyProj\src>javac -d ..\classes A.java

D:\MyProj\src>javac -d ..\classes -cp ..\classes B.java
B.java:8: error: a has private access in A
                System.out.println(obj.a);
                                   ^
1 error
```

```
package another;
import same.A;
public class C extends A
{
        public static void main(String args[]){
                C obj = new C();
                System.out.println(obj.a);
                System.out.println(obj.b);
                System.out.println(obj.c);
                System.out.println(obj.d);
        }
}
```

Compiling the above program:

```
D:\MyProj\src>javac -d ..\classes -cp ..\classes C.java
C.java:7: error: a has private access in A
                System.out.println(obj.a);
                                   ^
C.java:10: error: d is not public in A; cannot be accessed from outside
                System.out.println(obj.d);
                                   ^
2 errors
```

The –d option tells the Java compiler to create a separate directory and place the .class file in that directory (package). The (.) dot after –d indicates that the package should be created in the current directory.

String:

- ✓ A String represents group of characters. Strings are represented as String objects in java.
- ✓ String is the **final** and **immutable** class.
- ✓ String class implements from comparable, charSequance and Serializable interface.

Creating Strings:

- ✓ We can declare a String variable and directly store a String literal using assignment operator.

  String str = "Hello";
- ✓ We can create String object using new operator with some data.

  String s1 = new String ("Java");
- ✓ We can create a String by using character array also.

  char arr[] = { 'p','r','o','g','r','a','m'};
  String s2 = new String (arr);
- ✓ We can create a String by passing array name and specifying which characters we need:

  String s3 = new String (str, 2, 3);
- ✓ Here starting from 2nd character a total of 3 characters are copied into String s3.
- ✓ **String Class Methods:**

Method Description:

- ✓ String concat (String str) : Concatenates calling String with str. Note: + also used to do the same
- ✓ int length () : Returns length of a String
- ✓ char charAt (int index) : Returns the character at specified location (from 0)
- ✓ int compareTo(String str) : Returns a negative value if calling String is less than str, a positive value if calling String is greater than str or 0 if Strings are equal.
- ✓ boolean equals (String str) : Returns true if calling String equals str.
- ✓ boolean equalsIgnoreCase (String str) : Same as above but ignores the case
- ✓ boolean startsWith (String prefix) : Returns true if calling String starts with prefix
- ✓ boolean endsWith (String suffix) :Returns true if calling String ends with suffix
- ✓ int indexOf (String str) : Returns first occurrence of str in String.
- ✓ int lastIndexOf(String str) Returns last occurrence of str in the String.
- ✓ Note: Both the above methods return negative value, if str not found in calling String. Counting starts from 0.
- ✓ String replace (char oldchar, char newchar) : returns a new String that is obtained by replacing all characters oldchar in String with newchar.
- ✓ String substring (int beginIndex) : Returns a new String consisting of all characters from beginIndex until the end of the String
- ✓ String substring (int beginIndex, int endIndex) : returns a new String consisting of all characters from beginIndex until the endIndex.
- ✓ String toLowerCase () : converts all characters into lowercase
- ✓ String toUpperCase () : converts all characters into uppercase
- ✓ String trim () : eliminates all leading and trailing spaces.

Wrapper Classes:

- ✓ Wrapper Classes are used to convert primitive data types into objects.
- ✓ A wrapper class is a class whose object wraps the primitive data type.
- ✓ Wrapper classes are available in java.lang package.

Primitive data type	Wrapper class
Double	Double
Float	Float
Long	Long
int	Integer
Short	Short
Byte	Byte
Boolean	Boolean
Char	Character

- ✓ All wrapper class **except Character** class has **two constructors.**
- ✓ One constructer would take the corresponding **primitive type** as the parameter another constructor which would take a **String** as a parameter.
- ✓ In case of Character class we have **only one constructor** which takes **only char** as the parameter.
- ✓ Similar to Constructor, we have a static method called ***valueOf()*** in all wrapper classes, with similar arguments as the constructor.
- ✓ All the valueOf() methods returns instance of the corresponding wrapper class.

Boxing and Unboxing conversion:

- ✓ The Boxing conversion is the conversion of a **primitive value to its wrapper type.**
- ✓ The unboxing conversion is the conversion of a **wrapper instance into its corresponding primitive value.**

Example:

```
class conversion
{
        public static void main(String args[])
        {
                int i = 10;
                Integer i1 = i; //boxing conversion
                i = i1; // unboxing conversion
        }
}
```

Pass By Value and Pass By Reference

Pass By Value:

- Change in Formal variable value does not change actual value known as Pass by value.
- In call by value primitive Data types are passed as an argument.
- The method parameter values are copied to another variable and then the copied object is passed, that's why it's called pass by value.

Pass By Reference:

- Change in formal variable, makes change in actual argument is known as Pass by reference
- In call by reference, Reference Data types are passed as an argument.
- An alias or reference to the actual parameter is passed to the method, that's why it's called pass by reference.

PassByValue_Ref - Notepad

File Edit Format View Help

```
class Test
{
        int i,j;
        Test(){}
        Test(int i,int j)
        {
                this.i=i;
                this.j=j;
        }
        void calc(int i,int j)
        {
                i*=2;
                j/=2;
                System.out.println("Inside Calc : "+i+" "+j);
        }
        void calc(Test t)
        {
                t.i*=2;
                t.j/=2;
                System.out.println("Inside Calc : "+t.i+" "+t.j);
        }
}
class TestDemo
{
        public static void main(String []args)
        {
                Test t = new Test();
                int i=20,j=30;
                System.out.println("Pass By Value: ");
                System.out.println("Before Calling : "+i+" "+j);
                t.calc(i,j);
                System.out.println("After Calling : "+i+" "+j);


                System.out.println("\nPass By Reference: ");
                Test t1=new Test(10,20);
                System.out.println("Before Calling : "+t1.i+" "+t1.j);
                t1.calc(t1);
                System.out.println("After Calling : "+t1.i+" "+t1.j);
        }
}
```

Command Prompt

```
D:\>javac PassByValue_Ref.java

D:\>java TestDemo
Pass By Value:
Before Calling : 20 30
Inside Calc : 40 15
After Calling : 20 30

Pass By Reference:
Before Calling : 10 20
Inside Calc : 20 10
After Calling : 20 10

D:\>
```

	String	StringBuffer	StringBuilder
Storage Area	Constant String Pool	Heap	Heap
Modifiable	No (immutable)	Yes (Mutable)	Yes (Mutable)
Thread Safe	Yes	Yes	Not safe for multiple thread
Performance	Fast	Very Slow	Fast
Synchronization	No guarantee	Methods are synchronized	No guarantee of synchronization.
Examples	char c[] = {'J', 'a', 'v', 'a'}; String s1 = new String(c); System.out.println(s1);	StringBuffer sb = new StringBuffer("I Java!"); sb.insert(2, "like "); System.out.println(sb);	StringBuilder str = new StringBuilder("ATMIYA "); str.append("MCA"); System.out.println(str);

	Interface	Abstract
Multiple Inheritance	A class may implement several interface	A class may extend only one abstract class
Default implementation	An interface cannot provide any code at all, much less default code.	An abstract class can provide complete code, default code, and/or just stubs that have to be overridden.
Constants	Static final constants only, can use them without qualification in classes that implement the interface.	Both instance and static constants are possible. Both static and instance intialiser code are also possible to compute the constants.
Methods	An interface contains only methods declaration without abstract keyword.	An abstract class can contain abstract and non – abstract methods both.
Constructors	Interface cannot contain constructor.	Abstract class may have constructors.
Default Access Specifier for Members	public	Default
Speed	Slow, requires extra indirection to find the corresponding method in the actual class.	Fast

Syntax	[access_specifier] interface name
	{
	return_type name(param);
	data_type var_name = value;
	}
		[access_specifier] abstract class class_name
		{
		abstract return_type name(param);
		access_specifier return_type method_name (param)
		{
		// non-abstract method
		}

Example	interface Printable
	{
	int i=100; // default final static
	void print(); // public abstract
	}
		abstract class Vehicle
		{
		abstract int noofwheel();
		int noofseats(){return 0;}
		}

	Class Test implements Printable, Comparable
	{
	public void print(){}
	public int compareTo(Object o){
	return this.i - ((Test)o).i;
	}}
		class Car extends Vehicle
		{
		int noofwheel(){
		return 4;
		}
		}

Point	Comparable	Comparator
Package	java.lang.Comparable	java.util.Comparator
Method	public int compareTo(Object o1)	public int compare(Object o1, Object o2)
Use	compareTo() method is used to compare the contents of passed object with calling object. **(Compares values and returns an int)**	compare() method is used to compare two objects using another object. **(Compares values of two objects)**
Sorting logic	Sorting logic must be in same class whose objects are being sorted. Hence this is called natural ordering of objects.	Sorting logic may be in separate class. Hence we can write different sorting based on different attributes of objects to be sorted.
Implementation	Class whose objects to be sorted must implement this interface.	Class whose objects to be sorted do not need to implement this interface. Some other class can implement this interface.
Sorting method	public int compareTo(Object o1) This method compares this object with o1 object and returns a integer.Its value has following meaning • **positive** – this object is greater than o1 • **zero** – this object equals to o1 • **negative** – this object is less than o1	int compare(Object o1,Object o2) This method compares o1 and o2 objects. and returns a integer. Its value has following meaning. • **positive** – o1 is greater than o2 • **zero** – o1 equals to o2 • **negative** – o1 is less than o1
Calling method	Collections.sort(List) Here objects will be sorted on the basis of CompareTo method	Collections.sort(List, Comparator) Here objects will be sorted on the basis of Compare method in Comparator

Example	class Test{
	int i;
	public int compareTo(Object o){
	return this.i - ((Test)o).i;
	}
	}
		class Test{
		int i;
		public int compare(Object o1,Object o2){
		return ((Test)o1).i - ((Test)o2).i;
		}
		}

==	equals()	compareTo()	compare()
It is an operator.	It is a method of Object class.	It is abstract method of java.lang.Comparable interface.	It is abstract method of java.util.Comparator interface
Used for primitives as well as Objects.	Not Applicable for primitives	Not Applicable for primitives	Not Applicable for primitives
variable == variable	public boolean equals(Object o)	public int compareTo(Object o)	public int compare(Object o1, Object o2)
It is used to compares the object's locations in memory. (Compares references not values)	equals() method is actually meant to **compare the contents** of 2 objects, and not their location in memory. (**Compares values for equality)**	compareTo() method is compare the contents of passed object with calling object. **(Compares values and returns an int)**	compare() method is used to compare two objects using another object. **(Compares values of two objects)**
returns boolean	returns boolean either true or false	return int • 0 if both are equals • Positive if calling Object is > passed Object. • Nagative if calling Object is < passed Object.	return int • 0 if both are equals • Positive if calling Object > passed Object. • Nagative if calling Object < passed Object.
	equals: Compares this string to the specified object.	compareTo : compares two strings lexicographically.	compare : compares two strings lexicographically.

int a=100;
int b = 200;
 	
boolean c = a==b;
	class Test{
	int i;
	public boolean equals(Object o){
	if(this.i != ((Test)o).i)
	return false;
	return true;
	}
	}
		class Test{
		int i;
		public int compareTo(Object o){
		return this.i - ((Test)o).i;
		}
		}
			class Test{
			int i;
			public int compare(Object o1,Object o2){
			return ((Test)o1).i - ((Test)o2).i;
			}
			}

a==b , a!=b	a.equals(b)	a.compareTo(b)	compare(a,b)