Om Sakthi
Adhiparasakthi Engineering College
Melmaruvathur - 603319

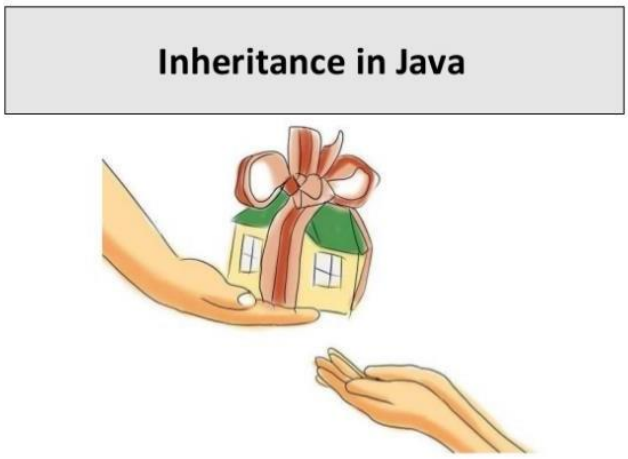
**OBJECT ORIENTED PROGRAMMING**

Subject Code: CS3391

Unit II

**INHERITANCE AND INTERFACES**

Inheritance – Definition:



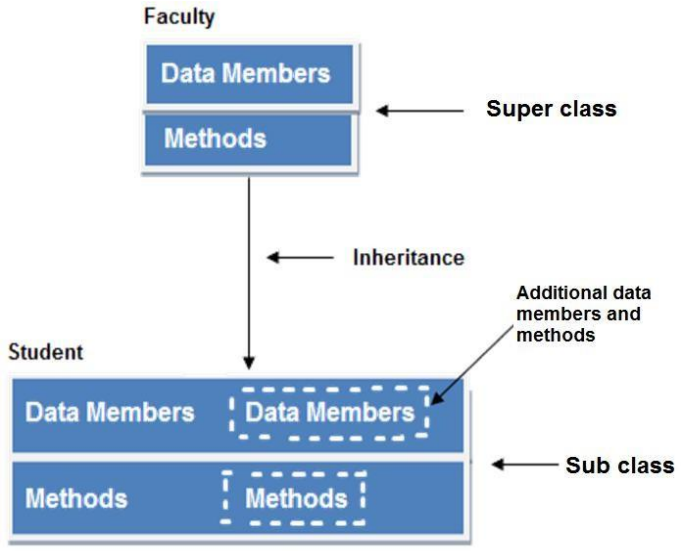

Derive a new class (subclass) from an existing class (base class or superclass) is called as Inheritance. Inheritance concept is mainly used for code reusability.

The idea behind inheritance in java is that new classes are created from the existing classes. When new classes are created, you can reuse methods and fields of parent class, and you can add new methods and fields also. Inheritance represents the IS-A relationship, also known as parent-child relationship.




The parent class or base class is called as super class.

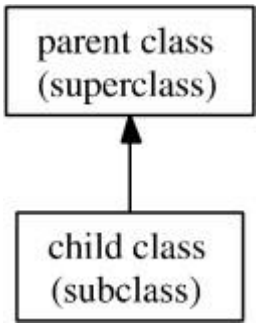
- Super Class: The class whose features are inherited is known as super class (or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as sub class (or a derived class, extended class, or child class).
    - Derived class = Parent class fields + new fields.




- Keywords used:
    - extends - In case of Java class and abstract class
    - implements – In case of Java interface.

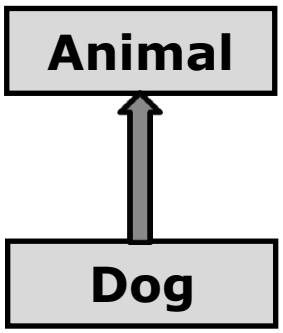
Syntax:

```
class Superclass {
  .....
  .....
}
class Subclass extends Superclass {
  .....
  .....
}
```

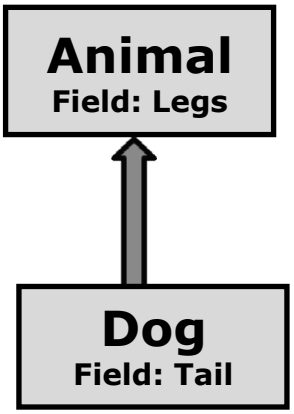



The **extends** keyword indicates that you are making a new class that derives from an existing class.

Example of inheritance:




Super class : Animal
Sub class : Dog




Super class (Animal) has datamember named legs and subclass (Dog) has field named tail.

```
class Animal
{
    int legs;
  . . .
 }

class Dog extends Animal
{
    int tail;
    ...
 }
```

Filename : Dog.java

```
class Animal
{
 int legs = 4;
}

public class Dog extends Animal
{
   int tail = 1;
   public static void main(String args[])
  {
   Dog d = new Dog();
   System.out.println("Dog has "+ d.legs + " legs");
   System.out.println("Dog has " + d.tail + " tail");
  } }
```

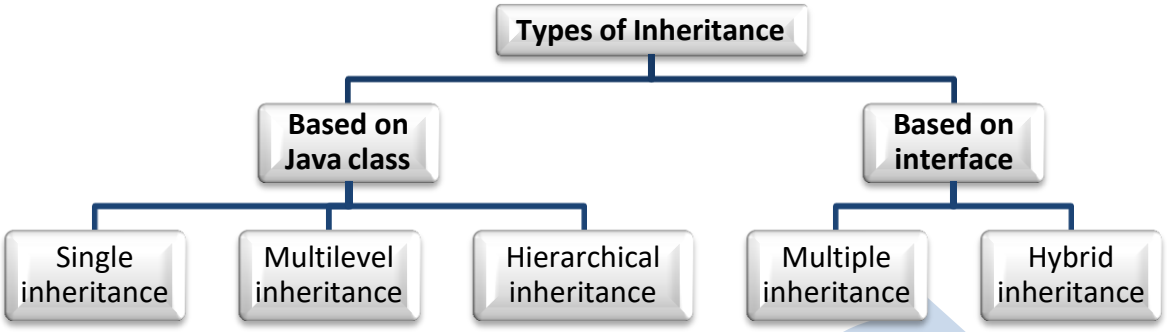
Output:
Dog has 4 legs
Dog has 1 tail

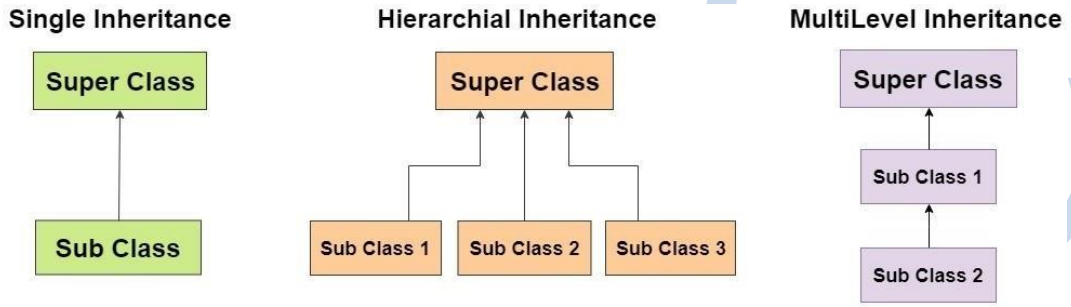
Super class data member is accessed in the sub class

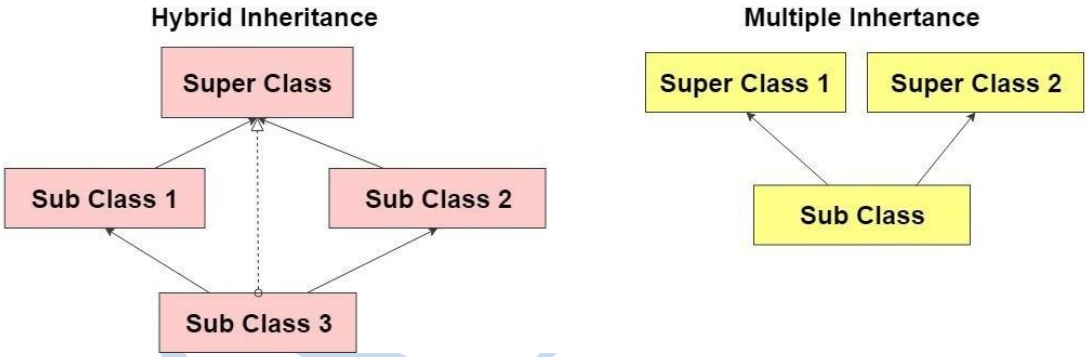
| Access specifier is private: | Access specifier is public: | Access specifier is protected: |
|---|---|---|
| class Animal<br>{<br>**private** int legs = 4;<br>} | class Animal<br>{<br>**public** int legs = 4;<br>} | class Animal<br>{<br>**protected** int legs = 4;<br>} |
| Output:<br>Compile error: legs has private access in Animal | Output:<br>Dog has 4 legs<br>Dog has 1 tail | Output:<br>Dog has 4 legs<br>Dog has 1 tail |

When a class is extended, sub-class inherits all the public, protected and default properties of the super class. Private fields and methods of the super class are **NOT INHERITED** by the sub-class.

Types of Inheritance:

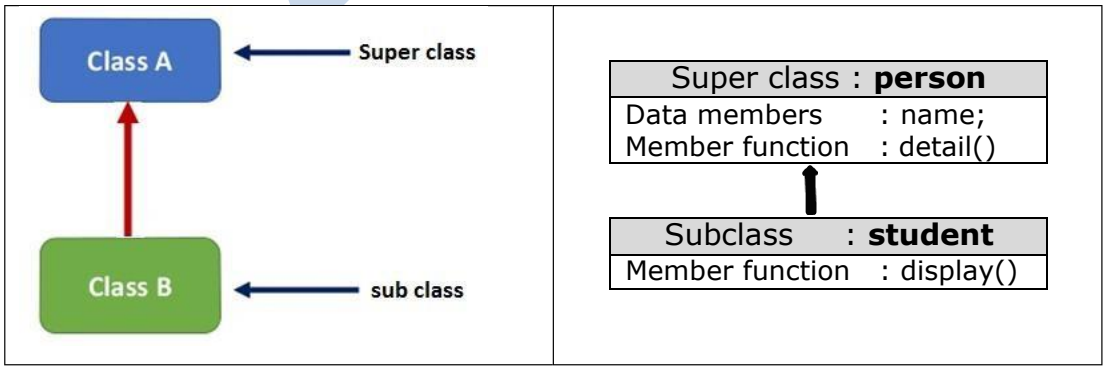



Single Inheritance:

When a subclass is derived from a single super class, is called as Single Inheritance.



Filename : student.java

```
class person
{
   String name = "Gayathri";
   void detail()
   {
      System.out.println("Am studying Java");
   }
}
```

→ **Super class**

```
public class student extends person
{
   void display()
   {
   System.out.println("My name is :" + name);
   }

   public static void main(String args[])
   {
      student p = new student();
      p.display();
      p.detail();
}}
```

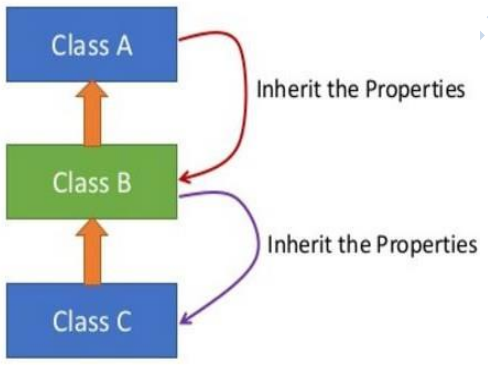
→ **Sub class**

Output:

My name is :Gayathri
Am studying Java

Multilevel Inheritance:

When a class extends a class, which in turn extends another class then it's called as multilevel inheritance. For example class C extends class B and class B extends class A then this type of inheritance is known as multilevel inheritance.




| Super class : **People** |
|---|
| Data members : name; |
| Member function : detail() |

↓

| Subclass : **Employee** |
|---|
| Data members : eno; |

↓

| Subclass : **Leader** |
|---|
| Data members : teamsize; |

People is the superclass
Employee is the subclass of People
Employee is the superclass of Leader
Leader is the subclass of Employee

Filename : Test.java

```
class People
{   String name = "Gayathri";
    void detail()
    {
        System.out.println("Method in Superclass");
    } }

class Employee extends People
{       int eno = 1901; }

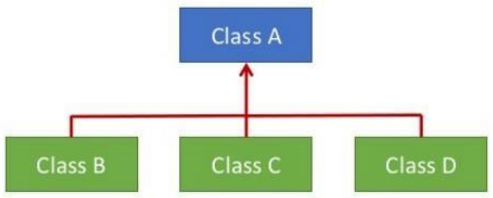
class Leader extends Employee
{    int teamsize = 15; }

public class Test
{
    public static void main(String args[])
    {
        Leader x = new Leader();
        System.out.println("Employee name : "+ x.name);
        System.out.println("Employee no : "+ x.eno);
        System.out.println("Team size : "+ x.teamsize);
        x.detail();
}}
```

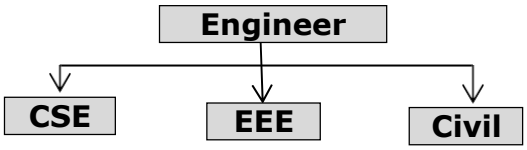
Output:
Employee name : Gayathri
Employee no : 1901
Team size : 15
Method in Superclass

Hierarchical Inheritance

When multiple classes inherit from a single superclass, then this type of inheritance is called as Hierarchical Inheritance. Here, there is one super class and multiple sub classes.




Here Class A acts as the parent for sub classes Class B, Class C and Class D




Engineer is the superclass
CSE, EEE, Civil are the subclasses of superclass, Engineer

Filename : Civil.java

```
class Engineer
{ String college = "IIT"; }

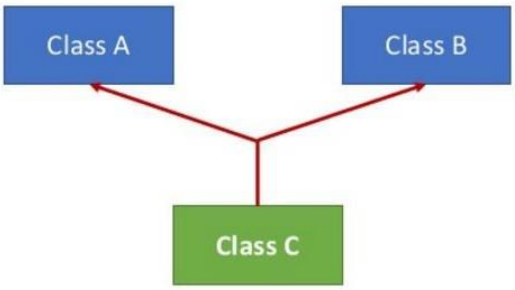
class EEE extends Engineer { }
class CSE extends Engineer  { }

public class Civil extends Engineer {
    public static void main(String args[]) {
        EEE e = new EEE();
        CSE c = new CSE();
        Civil ci = new Civil();
        System.out.println("EEE student is studying in : "+ e.college);
        System.out.println("CSE student is studying in : "+ c.college);
        System.out.println("Civil student is studying in : "+ ci.college);
    } }
```

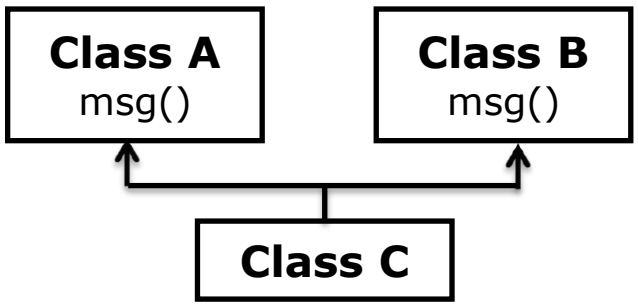
Output:
EEE student is studying in : IIT
CSE student is studying in : IIT
Civil student is studying in : IIT

Multiple inheritance is not supported in java:

In java programming, multiple and hybrid inheritance is not supported through classes but supported through interface only. To reduce the complexity and simplify the language, multiple inheritance is not supported in java.




Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.




Class C object, doesn't know which msg() method to be invoked.

APEC

Filename : C.java

Multiple inheritance NOT supported in Java

```
class A
 {
    void msg()
    {
       System.out.println("Hello");
    } }

class B
{   void msg()
    {
       System.out.println("Welcome");
    } }

public class C extends A,B              //suppose if it were
{
    public static void main(String args[])
    {
       C obj=new C();
       obj.msg();              //which msg() would be invoked?
       } }
```

Purpose of Inheritance

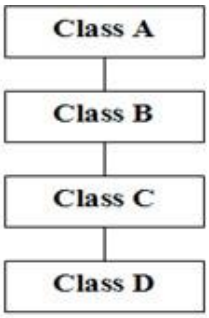
- It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
- It promotes polymorphism by allowing method overriding.

APEC

**Constructors in Java Inheritance:**

In Java, super class default constructor is invoked automatically in sub class. Object is created for the sub class. With the sub class object, super class constructors are invoked.

Constructors in Multilevel Inheritance




**Order of Construction**
**1.Class A**
**2. Class B**
**3. Class C**
**4. Class D**

Filename : Test.java

**Class A is the Super class. Class B is inherited from Class A**

```
class A
{
  A()
  {
    System.out.println("Class A ");
  }
}
```

```
class B extends A
{
  B()
  {
    System.out.println("Class B ");
  }
}
```

**Class C is inherited from Class B. Class D is inherited from Class C.**

```
class C extends B
{
  C()
  {
    System.out.println("Class C ");
  }
}
```

```
class D extends C
{
  D()
  {
    System.out.println("Class D ");
  }
}
```

```
public class Test
{
  public static void main(String[] args)
  {
  D d = new D();
  }
```

Output:
Class A
Class B
Class C
Class D

**Super Keyword in Java**

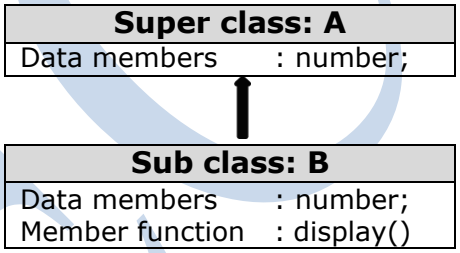
The super keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

- Use of super with variables
- Use of super with methods
- Use of super with constructors

Use of super with variables

When both the super class and sub class has same data members, there is a possibility of ambiguity for the JVM. This is overcome by the keyword "super".

Filename : Test.java




```
class A
{
   int number = 200;
}

class B extends A
{
   int number = 500;

   void display()
   {
      System.out.println(number);
      System.out.println(super.number);
   }
}

public class Test
{
   public static void main(String args[])
   {
      B b =new B();
      b.display();
  } }
```

Super class number = 200 is printed

Sub class number = 500 is printed

Output:
500
200

Use of super with methods

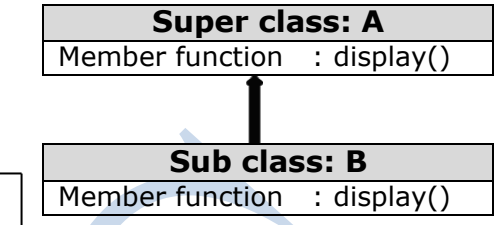
Whenever a parent and child class have methods with the same name, the ambiguity is resolved using the super keyword.

Filename : Test.java

```
class A
{
   void display()
   {
      System.out.println("This is Super class");
   }
}

class B extends A
{
   void display()
   {
      System.out.println("This is sub class");
      super.display();
   }
}

public class Test
{
   public static void main(String args[])
   {
      B b =new B();
      b.display();
 } }
```




Inside the Sub class method display() , the super class method display()is called

Sub class method display() is invoked

Output:

```
This is sub class
This is Super class
```

APEC

Use of super with constructors:

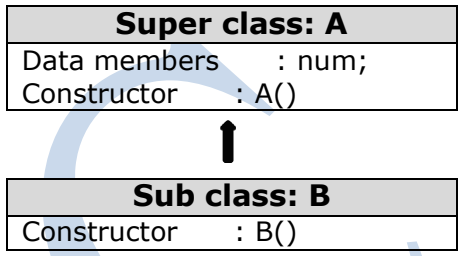
Super keyword can also be used to access the parent class constructor. One more important thing is that, ''super' can call both parametric as well as non-parametric constructors depending upon the situation. Call to super() must be first statement in Derived(Student) Class constructor.

Filename : Test.java

```
class A
{
   int num;
   A()
   {
      System.out.println("A is created");
   } }

class B extends A
{
   B()
   {
      System.out.println("B is created");
   } }

public class Test
{
   public static void main(String args[])
   {
   B b = new B();
}  }
```



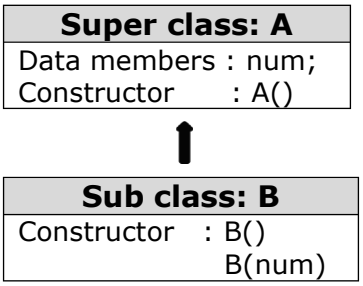

Super class default constructor is invoked first

Output:

```
A is created
B is created
```

APEC

Filename : Test.java




```
class A
{
   int num;
   A()
   {
      System.out.println("A is created");
   } }

class B extends A
{
   B()
   {
      System.out.println("B is created");
   }
  B(int num)
   {
      System.out.println("Number inside sub class");
   }
}

public class Test
{
   public static void main(String args[])
   {
   B b = new B(10);
}  }
```

Super class default constructor is invoked first. Then one argument constructor in the sub class is invoked.

Output:

```
A is created
Number inside sub class
```

APEC

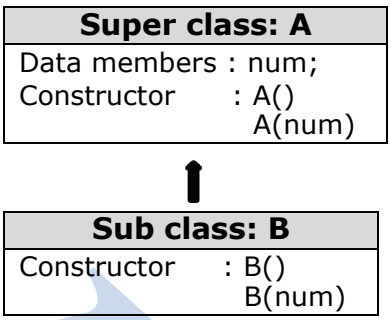
Filename : Test.java

```
class A
{
   int num;
   A()
   {  System.out.println("A is created");
  }
   A(int num)
   {
      System.out.println("Number inside super class");
   } }


class B extends A
{
   B()
   {  System.out.println("B is created");     }


   B(int num)
   {
      System.out.println("Number inside sub class");
   }   }


public class Test
{
   public static void main(String args[])
   {
   B b = new B(10);
  }  }
```




Though one argument constructor in the super class is available, only super class default constructor is invoked.

Output:

```
A is created
Number inside sub class
```

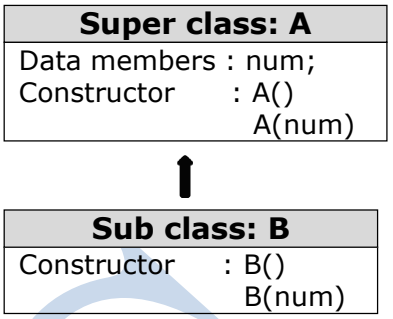
APEC

Filename : Test.java

```
class A
{
   int num;
   A()
   {
      System.out.println("A is created");
   }
   A(int num)
   {
      System.out.println("Number inside super class");
   }
}

class B extends A
{
   B()
   {
      System.out.println("B is created");
   }
   B(int num)
   {
      super(num);
      System.out.println("Number inside sub class");
   } }

public class Test
{
   public static void main(String args[])
   {
   B b = new B(10);
   }  }
```




One argument constructor in the super class is invoked.

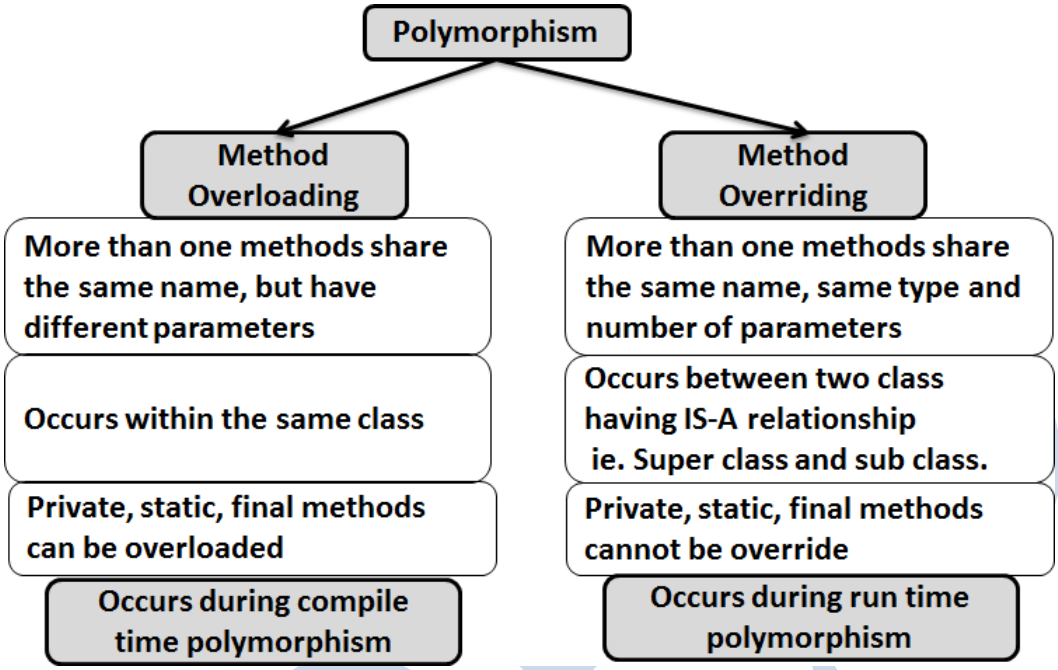
Output:

```
Number inside super class
Number inside sub class
```

APEC

**Polymorphism in Java**

Polymorphism is derived from two Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. The ability to take more than one forms is called as polymorphism.




Method overloading in Java:

When the compiler is able to select the appropriate function at the compile time itself, then it's called as compile time polymorphism. Method overloading or function overloading is compile time polymorphism. It's called as early binding or static binding or static linking.

Same function name with different number of parameters or different order of parameters or different types of parameters, it's called as method overloading.

- Overloading methods differ in number of parameters
  - int add(int);
  - int add(int,int);
- Overloading methods differ in types of parameters
  - int add(float,float);
  - int add(int,int);
- Overloading methods differ in order of parameters
  - int add(float,int);
  - int add(int,float);

Filename : Addition.java

```
public class Addition
{
   void sum(int n1, int n2)
   {
      System.out.println("Addition : "+ (n1+n2));
   }

   void sum(int n1, int n2, int n3)
   {
      System.out.println("Addition : "+ (n1+n2+n3));
   }
   public static void main(String args[])
   {
   Addition a = new Addition();
   a.sum(10,20);  ⟶  Calls the method with 2 parameters.
   a.sum(3,6,9);  ⟶  Calls the method with 3 parameters.
 }  }
```

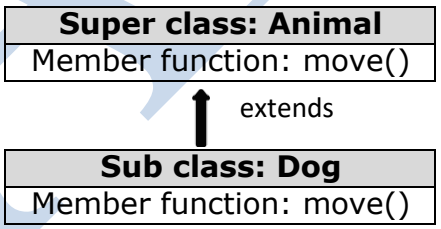
Same function name **sum** with different number of arguments is presented in the same class.

Output:

```
Addition : 30
Addition : 18
```

Method overriding in Java

If both the super class (parent class) and the subclass (child class) have the same method name, it is known as method overriding in java.




Same function name, **move()** in both the super class and sub class.

When a method in a subclass has the same name, as a method in its super-class, then it is said to override the method in the super-class.

```
Filename : Test.java
class Animal
{
   void move()
   { System.out.println("Super class move"); }
}

class Dog extends Animal
{
   void move()
   { System.out.println("sub class move"); }
}

public class Test
{
   public static void main(String[] args)
   {
      Animal ani = new Animal();
      ani.move();

      Dog dd = new Dog();
      dd.move();

      Animal doggy = new Dog();
      doggy.move();
   }
 }
```

Output:
```
Super class move
sub class move
sub class move
```

→ Super class refers to super class object

→ Sub class refers to sub class object

→ Super class refers to sub class object

## Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user. There are two ways to achieve abstraction in java

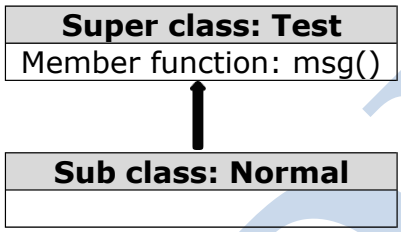
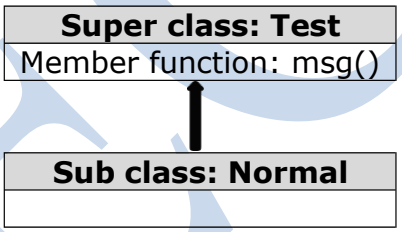
- Abstract class (0 to 100%)
- Interface (100%)

Abstract class in Java

A class that is declared with abstract keyword is known as abstract class in java. It can have abstract and non-abstract methods (method with body). An abstract class is never instantiated.

Syntax : *abstract class classname*
{ *Members*;
*concrete methods*;
*Abstract methods* }

Example : abstract class A { }

| Concrete Class | Abstract Class |
|---|---|
| A concrete class is one which is containing fully defined methods or implemented methods. Fully implemented class is known as concrete or normal class. | A partial implemented class is called an abstract class. |
| <br>In concrete class, member functions are fully defined. | <br>In abstract class, it can have abstract methods or concrete methods. |
| Filename : Normal.java<br><br>class **Test**<br>{<br>void msg()<br>{<br>System.out.println("Hello CSE");<br>}<br>}<br><br>public class **Normal** extends **Test**<br>{<br>public static void main(String args[])<br>{<br>Normal n= new Normal();<br>n.msg();<br>}<br>} | Filename : Normal.java<br><br>**abstract** class Test<br>{<br>**abstract** void msg();<br>}<br><br>public class Normal extends Test<br>{<br>void msg()<br>{<br>System.out.println("Hello CSE");<br>}<br>public static void main(String args[])<br>{<br>Normal n= new Normal();<br>n.msg();<br>} } |

*Rules for abstract class:*

- An abstract class is a class that is declared abstract
- Abstract classes cannot be instantiated
- Abstract classes can be sub-class

- It may or may not include abstract methods
- If subclass doesn't provide implementations then the subclass must also be declared abstract.

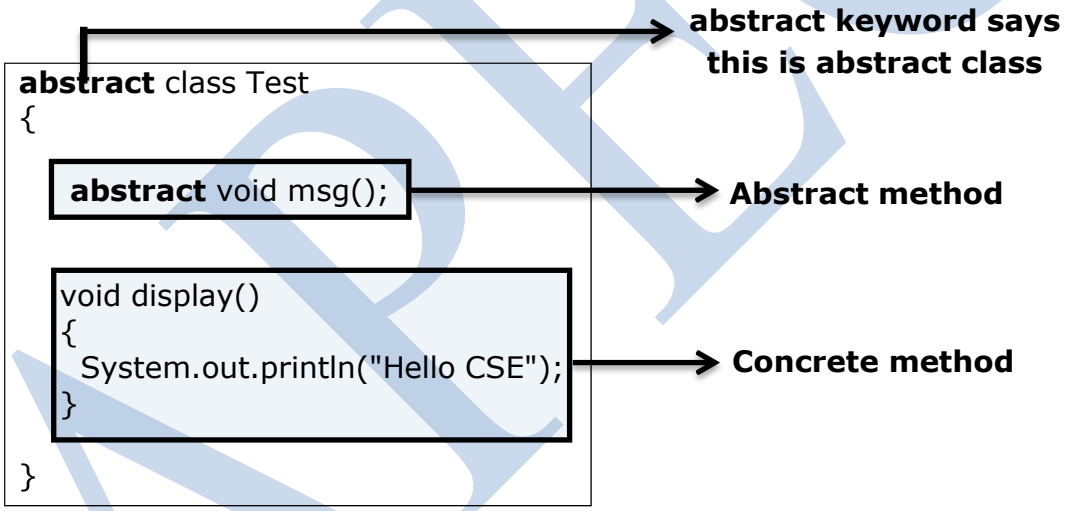
Abstract method in Java

- Method that is declared without any body(implementation) within an abstract class are called abstract method.
- The implementation of the method will be defined by its subclass. Abstract method can never be final and static.

Syntax : $abstract\ ReturnType\ methodName(List\ of\ formal\ parameter);$

Example : abstract void display(); (or) abstract int sum(int a, int b);

| Concrete Method | Abstract Method |
|---|---|
| Concrete class is one which is containing fully defined methods or implemented method | Method that is declared without any body (implementation) within an abstract class is called abstract method. |




Abstract class are never instantiated which means we can't create an object to Abstract class.

```
abstract class Test
{
   void display()
   {  System.out.println("Java Programming");    }
 }

public class Normal extends Test
{
   public static void main(String args[])
   {
     Test n= new Test();
     Test n= new Normal();
 } }
```

**Compile time error**
Object for the abstract class cannot be created

Its **OK.**

Simple example to explain abstract class and abstract method in Java:

Filename : Test.java

```
abstract class Test
{
   abstract void msg();
   void display()
   {  System.out.println("Java Programming");
    }
 }

public class Normal extends Test
{
void msg()
{
   System.out.println("Hello CSE");
   }
   public static void main(String args[])
  {
      Normal n= new Normal();
     n.display();
     n.msg();
    }
 }
```

Super class is the abstract class

Member function is declared, but **NOT DEFINED.** This is **Abstract method**

Member function is defined. This is **Concrete method.**

Abstract method is defined in the sub class

Output:

```
Java Programming
Hello CSE
```

Points to remember for abstract classes:

- Abstract class and interfaces are different concepts.
- Abstract method should be inside the abstract classes.
- Abstract class may or may not have abstract methods. Even a single abstract method is present, and then the class must be declared abstract.
- Abstract class can have Constructors, Member variables and normal methods.
- Abstract class are never instantiated which means we can't create an object to Abstract class.
- Abstract class can be extended.
- An abstract method must be overridden in a subclass.

Advantage of Abstract Class

- Abstract classes can group several related classes together as siblings. Grouping classes together is important in keeping a program organized and understandable.
- Abstract class is a way to organize inheritance

Disadvantage of Abstract Class

- Abstract classes cannot be instantiated

**Final keyword in java**

Final is a non-access modifier applicable only to a variable, a method or a class.

| | |
|---|---|
| **Final variable** | Values cannot be changed |
| **Final methods** | Prevent method overriding |
| **Final classes** | Prevent Inheritance |

Final variable

Final variables are nothing but constants. The value of a final variable cannot be changed once it is initialized.

```
public class race
{
    final int speed = 200 ;
    void run()
    {
      speed = 700;
      }
    public static void main(String args[])
    {
       race r = new race();
       r.run();
    }
}
```

speed is a **final** variable of integer datatype, which is initialized with the value 200

| **Compile time error** |
|---|
| speed = 200 is changed to speed = 700 |

If final variables are not initialized with their values, it can be initialized only in the constructors. It cannot be initialized with the member functions.

*Final variables cannot be initialized inside methods.*

```
public class race
{
    final int speed ;

     void run()
     {
       speed = 700;
        System.out.println("My speed is "+ speed);
     }

     public static void main(String args[])
     {
        race r = new race();
        r.run();
     }
}
```

**Compile time error**

```
cannot assign a value to final variable speed
```

*Final variables can be initialized inside constructors*

```
public class race
{
   final int speed;

   race()
   {
    speed = 1000;
    System.out.println("My speed is "+ speed);
   }

   public static void main(String args[])
   {
   race r = new race();
   } }
```

Inside the constructor, final variables **CAN BE** initialized

Output:
```
My speed is 1000
```

Final Methods

Final method cannot be overridden. Parent class final method can be used, but it can't be overridden in subclass.

| Base class | Derived class | Description |
|---|---|---|
| class race<br>{<br>void **run**()<br>{<br>…<br>}<br>} | public class Test extends race<br>{<br>void **run**()<br>{ …. }<br><br>public static void main(String args[])<br>{<br>race r = new race();<br>r.run();<br>} } | Super class has method, **run()** and the sub class also has the method, **run()**.<br><br>Super class, run() will be invoked |

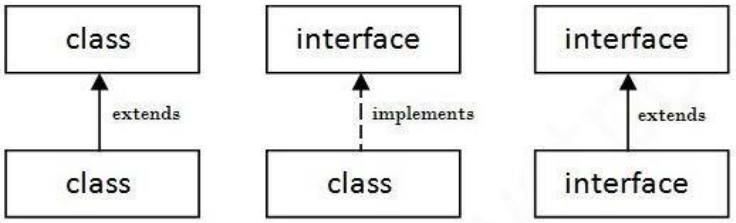
| Base class | Derived class | Description |
|---|---|---|
| class race<br>{<br>**final** void **run**()<br>{<br>…<br>}<br>} | public class Test extends race<br>{<br>void **run**()<br>{ …. }<br><br>public static void main(String args[])<br>{<br>race r = new race();<br>r.run();<br>} } | Super class has method, **run()** and the sub class also has the same method, **run()**.<br><br>Here, final methods **CANNOT** be overridden<br><br>Note:<br>Final methods can be inherited, but cannot be overridden. |

Final class

| **Final class cannot be extended** |
|---|
| **final** class race<br>{ }<br><br>public class Test extends race<br>{ public static void main(String args[])<br>{ }<br>} |

| **Compile time error** |
|---|
| cannot inherit from final race |

**INTERFACES**




- Interface is a pure abstract class. They are syntactically similar to classes, but objects cannot be created. Interface is used to achieve complete abstraction in Java. It has the collection of
  - Abstract methods
  - Static and final constants.
- Java uses interface to implement multiple inheritance.
- Interface is written in a file with a .java extension, with the name of the interface matching the name of the file.

- Interface does not contain any constructors.
- Interface is not extended by a class; it is implemented by a class and it can extend multiple interfaces.

Syntax for declaring Interfaces

```
interface interfacename
{
  // Any number of final, static fields
  // Any number of abstract method declarations
  }
Public class classname implements interfacename
{}
```

Simple example to explain abstract class and abstract method in Java:

Filename : Test.java

```
interface myname
{
   int n = 500;

   void print();
   }

public class Test implements myname
{
   public void print()
   {    System.out.println("Java Interface" + n);    }

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

**myname** is the name of the interface.

Variables are NOT instance variables. They are either final or static. No need of keywords.

Methods in the interface are implicitly abstract. No need to specify abstract keyword.

Methods in an interface are public

Implementing interface using Real world example:

Let's consider the example of bank like SBI, HDFC, ICICI etc. The entire banks have the common functionalities. Make an interface and put all these common functionalities inside. Each bank implements all these functionalities in their class.

Filename : Test.java

```
interface Bank
{
   double ROI = 14.5;
   int period = 4;
}
```

→ Bank Interface

```
class SBI implements Bank
{
   SBI()
   {
     System.out.println("SBI ROI "+ ROI + " with " + period +"yrs");
   }
}
```

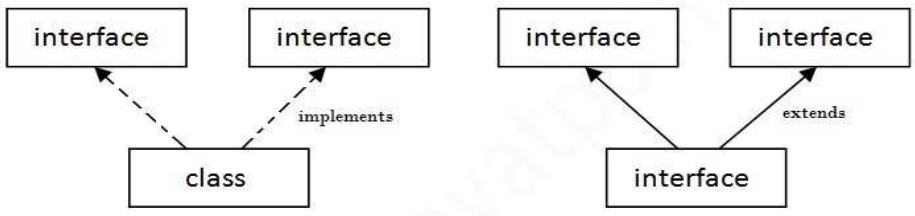
```
class HDFC implements Bank
{
   HDFC()
   {
     System.out.println("HDFC ROI "+ ROI + " with " + period +"yrs");
   }
}
```

```
public class Test
{
   public static void main(String[] args)
   {
      Bank b = new SBI();
      Bank h = new HDFC();
   }
}
```

Output:

```
SBI ROI 14.5 with 4yrs
HDFC ROI 14.5 with 4yrs
```

Interfaces support Multiple Inheritance



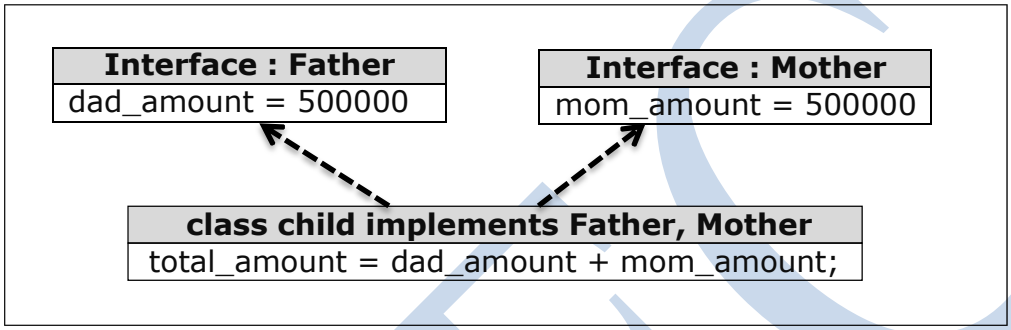

**Multiple Inheritance in Java**

Multiple inheritance is not supported in java because of ambiguity. With interface, multiple inheritance is possible.

Syntax:

| Interface 1 | Interface 2 | Class child | Test class |
|---|---|---|---|
| ***interface*** *one*<br>{<br>…<br>} | ***interface*** *two*<br>{<br>…<br>} | ***class*** *child* ***implements*** *one, two*<br>{<br>…<br>} | *class Test*<br>{<br>*child c = new child();*<br>} |

Simple example:




Filename : Test.java

```
interface Father
{
   int dad_amount = 500000;
}
```

→ First interface - Father

```
interface Mother
{
   int mom_amount = 300000;
}
```

→ Second interface - Mother

```
class child implements Father, Mother
{
  int total_amount;
  child()
  {   total_amount = dad_amount + mom_amount;
    }
}
```

→ class child which implements both father and mother

```
public class Test
{
   public static void main(String[] args)
   {
   child c = new child();
   System.out.println("Child's amount :" + c.total_amount);
  }
}
```

→ object for child class is created

Output:
Child's amount :800000

Name conflict:

Variable names conflicts can be resolved by interface name. Same variable name can be used in different interfaces. They can be accessed with the dot operator.

Syntax: $interfacename.variablename$

Filename : Test.java

```
interface Father
{
   int amount = 500000;
}

interface Mother
{
   int amount = 300000;
}

class child implements Father, Mother
{
  int total_amount;
  child()
  {    total_amount = Father.amount + Mother.amount;
    }
}

public class Test
{
    public static void main(String[] args)
    {
    child c = new child();

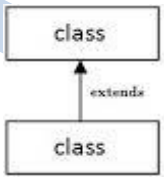
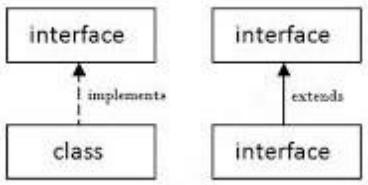
    System.out.println("Child's amount :" + c.total_amount);
   }
}
```

Same variable name in both interfaces, Father and Mother

Name conflict is avoided

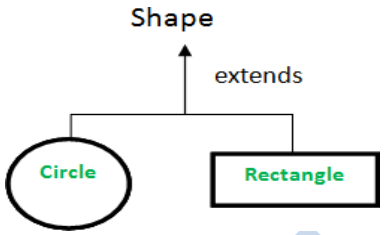
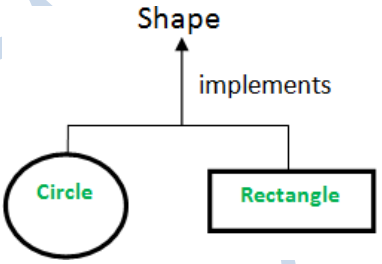
APEC

Key Differences between Class and Interface in Java

| Description | CLASS | INTERFACE |
|---|---|---|
| **Basic** | A class is instantiated to create objects. | An interface cannot create objects. |
| **Keyword** | class | interface |
| **Syntax** | *class classname*<br>{<br>*Members and methods*<br>} | *interface interfacename*<br>{<br>*Final variables*;<br>*Abstract methods*;<br>} |
| **Access specifier** | Data members in the class can be private, public or protected. | Data members in the interface are always public. |
| **Members** | Members of the class need to be always constants. | Members of the interface are always constants. Either static or final. |
| **Methods** | Methods inside the class are defined to perform a specific action. | Methods in the interface are purely abstract. They are not defined. |
| **Inheritance** | Supports only multilevel and hierarchical inheritances but not multiple inheritances. | Supports all types of inheritance – multilevel, hierarchical and multiple |
| **Implement/ Extend** | A class can implement any number of interfaces and can extend only one class. | An interface can extend multiple interfaces but cannot implement any interface. |
| **Constructor** | A class can have constructors to initialize the variables. | An interface can never have a constructor as there is hardly any variable to initialize. |
| **Representation** |  |  |

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

| Description | Abstract class | Interface |
|---|---|---|
| **Keyword** | abstract | interface |
| **Representation** |  |  |
| **Syntax** | *abstract class classname*<br>{<br>*Members;*<br>*concrete methods;*<br>*Abstract methods*<br>} | *interface interfacename*<br>{<br>*Final variables;*<br>*Abstract methods;*<br>} |
| **Access Specifier** | Members may be public, private, protected. | Members of a Java interface are public by default. |
| **Methods** | Both abstract and non-abstract methods. | Interface can have only abstract methods. |
| **Inheritance** | Abstract class doesn't support multiple inheritances. | Interface supports multiple inheritances. |
| **Members** | Abstract class can have final, non-final, static and non-static members. | Interface has only static and final members. |
| **implements** | Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| **extends** | An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
| **Example** | abstract class Shape<br>{<br>abstract void draw();<br>} | interface one<br>{<br>void draw();<br>} |

**Extending Interfaces**

An interface can extend another interface in the same 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. Java class can extend only one parent class. Interfaces are not classes, and an interface can extend more than one interface.

Syntax: ***interface*** *InterfaceName* ***extends*** *interface1, interface2, …*

{ *interface body* }

Example program for extending interface:

Filename : Test.java

```
interface i1
{
   public void f1();
}
```

Parent interface i1.

Method f1() is only declared in the interface and it's **NOT** defined.

```
interface i2 extends i1
{
   public void f2();
}
```

Since interface i1 extends i1, both i1 methods and i2 methods are accessible via i2

```
class cla implements i2
{
   public void f1()
   {
      System.out.println("Interface1");
   }
   public void f2()
   {
      System.out.println("Interface2");
   }
   void f3()
   {
       System.out.println("Class");
   }
}
```

Class cla implements interface i2. It means, class cla implements both i1 and i2

Method f1() definition of interface i1

Method f2() definition of interface i2

Method f3() definition of class cla

```
public class Test
{    public static void main(String[] args)
   {
      i2 obj = new cla();

      obj.f1();
      obj.f2();

      cla c = new cla();
      c.f3();
    }
```

Reference variable of interface i2

Output:
Interface1
Interface2
Class

**Object Cloning in Java**

Already in Java copy constructor, we have seen the ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

***Clone() method in Java***

clone() is a method in the Java programming language for object duplication. Object cloning refers to creation of exact copy of an object. It creates a new instance of the current object and initializes all its fields with exactly the same contents of the corresponding fields. By default, java cloning is 'field by field copy.

The $java.\ lang.\ Cloneable$ interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates $CloneNotSupportedException$.

Syntax of the clone() method:

$$protected\ Object\ clone()\ throws\ CloneNotSupportedException$$

Types of cloning:

Java supports two type of cloning.

- Deep cloning
- Shallow cloning.

By default shallow clone is used in Java. Object class has a method clone() which does shallow cloning.

*Shallow copy* is method of copying an object and is followed by default in cloning. In this method the fields of an old object X are copied to the new object Y. While copying, object Y will point to same location as pointed out by X.

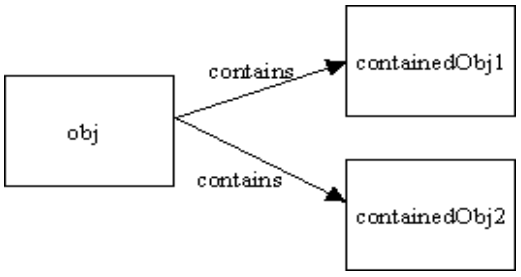



Fig. Original Java object obj

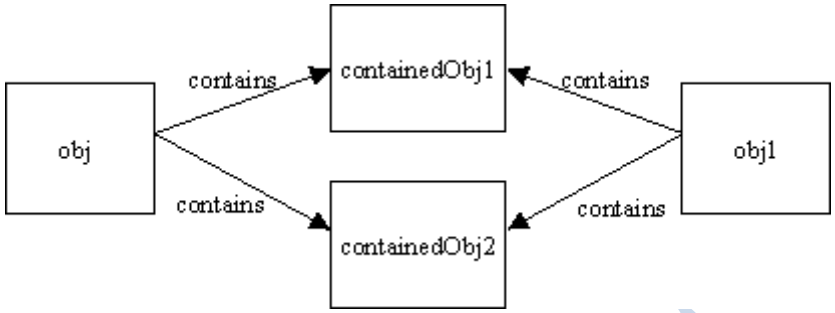



Fig. Shallow copy object obj1

If the field value is a primitive data type it copies the value of the primitive type. Therefore, any changes made in referenced objects in object X or Y will be reflected in other object.

*Deep copy* is to create a copy of object X and place it in a new object Y. New copies of referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields.

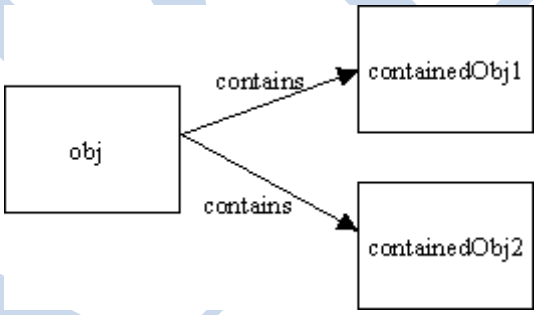



Fig. Original Java object obj

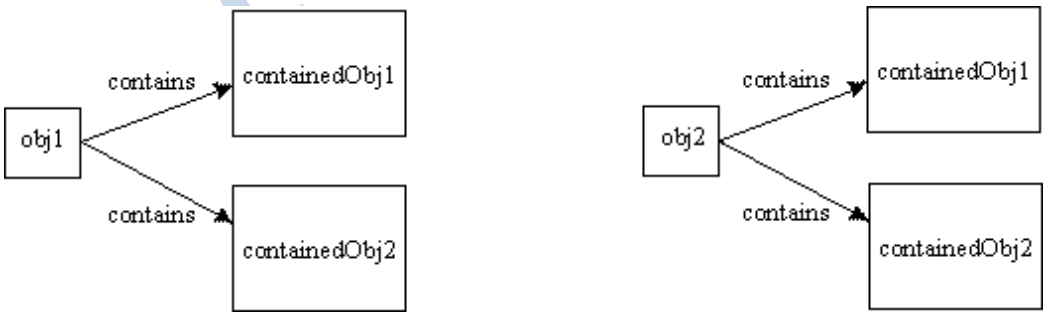



Fig. Deep copy object obj1

Filename : Student.java

```
public class Student implements Cloneable
{
   int rollno;

   Student(int r)
   {
      rollno = r;
   }

public Object clone()throws CloneNotSupportedException
{
   return super.clone();
}

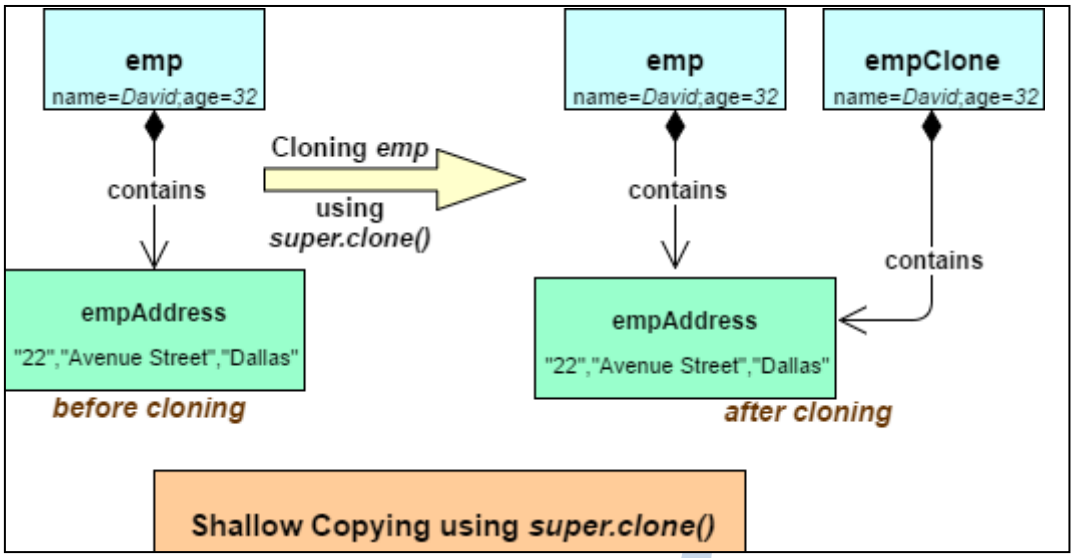
public static void main(String args[])
{
try
{
   Student s1 = new Student(1901);
   Student s2 = (Student)s1.clone();
   System.out.println("s1 object rollno: " + s1.rollno);
   System.out.println("s2 object rollno: " + s1.rollno);

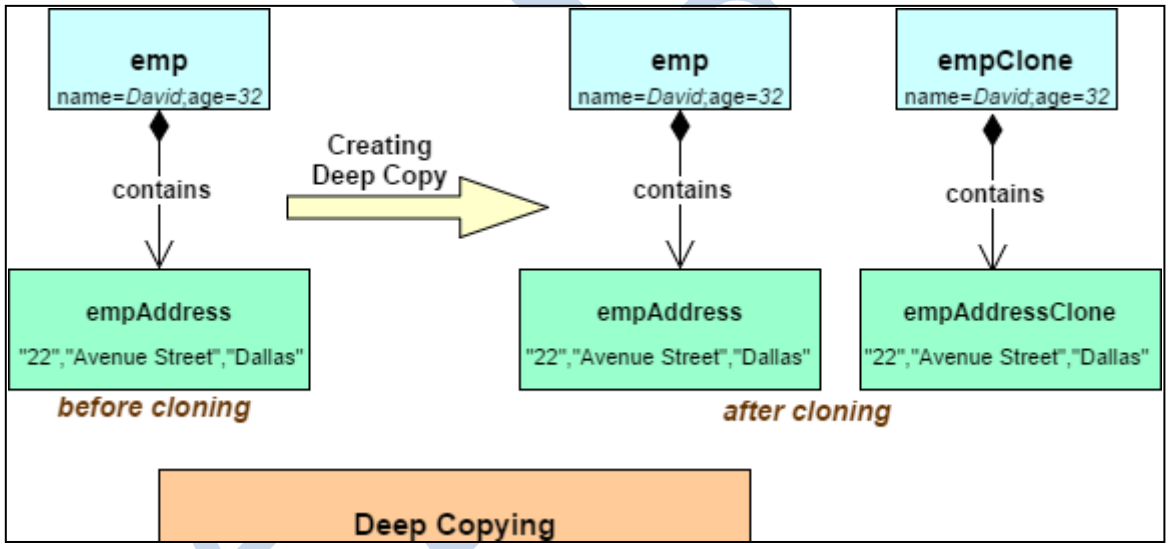
   s1.rollno = 1401;
   System.out.println("s1 object rollno after updation: " + s1.rollno);
   System.out.println("s2 object rollno after updation: " + s1.rollno);
}

catch(CloneNotSupportedException c){}
}
}
```

Output:
s1 object rollno: 1901
s2 object rollno: 1901
s1 object rollno after updation: 1401
s2 object rollno after updation: 1401

APEC


emp
name=David;age=32
contains
empAddress
"22","Avenue Street","Dallas"
before cloning
Cloning emp
using
super.clone()
emp
name=David;age=32
contains
empClone
name=David;age=32
contains
empAddress
"22","Avenue Street","Dallas"
after cloning
Shallow Copying using super.clone()


emp
name=David;age=32
contains
empAddress
"22","Avenue Street","Dallas"
before cloning
Creating
Deep Copy
emp
name=David;age=32
contains
empAddress
"22","Avenue Street","Dallas"
empClone
name=David;age=32
contains
empAddressClone
"22","Avenue Street","Dallas"
after cloning
Deep Copying

Advantages of clone method:

- If we use assignment operator to assign an object reference to another reference variable then it will point to same address location of the old object and no new copy of the object will be created. Due to this any changes in reference variable will be reflected in original object.
- If we use copy constructor, then we have to copy all of the data over explicitly i.e. we have to reassign all the fields of the class in constructor explicitly. But in clone method this work of creating a new copy is done by the method itself. So extra processing is avoided in object cloning.

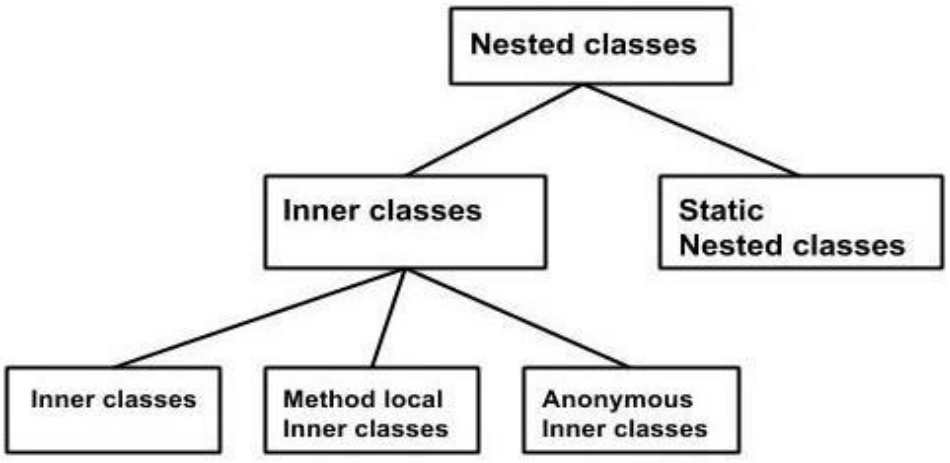
**Java Inner Classes:**

Writing a class within another class 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.

Syntax for Nested class:

```
class Outer
{
    //class Outer members
    class Inner
    {
        //class Inner members
    }
}//closing of class Outer
```

There are two types of nested classes: non-static and static nested classes. The non-static nested classes are also known as inner classes.



| Type | Description |
|---|---|
| **Inner Class** | A class created within another class. |
| **Anonymous Inner Class** | A class created for implementing interface or extending class. Its name is decided by the java compiler. |
| **Local Inner Class** | A class created within method. |
| **Static Nested Class** | A static class created within class. |

Inner class:

A Nested inner class can access any private instance variable in java which can be of any outer class. We can also define the access of this Java inner class, i.e. private, public or protected.

Syntax:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

Filename : Outer.java

```
public class Outer
{
    private int data = 100;
    public class Inner
     {
        void msg(){System.out.println("Data is "+data);}
      }
     public static void main(String args[])
    {
      Outer obj = new Outer();
      Outer.Inner in = obj.new Inner();
      in.msg();
    }
}
```

Inner Class

Inner Class object is created

Output:

```
Data is 100
```

Note:

Outer.Inner in = new Outer().new Inner(); can also be used.

Method Local Inner class

A Method local inner class is a Java inner class defined within a method of an outer class in java. In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

Syntax:

```
class outer
{
   void method()
     { ……..
         class inner
         { }
     }
}
```

Filename : Student.java

```
class Outer
{
   int count;
   public void display()
   {
      for(int i=0;i<5;i++)
      {
      class Inner
      {
         public void show()
         {
         System.out.println("Inside inner "+(count++));
         }
      }
      Inner in=new Inner();
      in.show();
      }
   }
}

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

Method display()

Inside method there is a class , Inner

Output:
Inside inner 0
Inside inner 1
Inside inner 2
Inside inner 3
Inside inner 4

Anonymous Inner Class

A local inner class without name is known as anonymous inner class. An anonymous class is defined and instantiated in a single statement. Anonymous inner class always extend a class or implement an interface. Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class.

Syntax:

```
Anonymous obj = new Anonymous ()
{
  public void my_method()
   {
      ........
   }
};
```

Java anonymous inner class example using class

File: Test.java

```
class Demo
{
   void show()
   {
      System.out.println("i am in show method of super class");
   }
}

public class Test
{
    public static void main(String[] args)
    {
      Demo d = new Demo()
      {
      void show()
         {
            super.show();
            System.out.println("i am in anonymous inner class");
         }
      };
   d.show();
  }
}
```

→ **An anonymous class with Demo as base class**

Output:

```
i am in show method of super class
i am in anonymous inner class
```

APEC

Java anonymous inner class example using interface

File: Demo.java

```
public class Demo
{
  public static void main(String[] args)
  {
    Hello h = new Hello()
    {
       public void show()
       {
          System.out.println("i am in anonymous class");
       }
    };
    h.show();
  }
}

interface Hello
{
    void show();
}
```

**An anonymous class that implements Hello interface**

Output:
i am in anonymous class

Static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or methods

Syntax

```
class Outer
{
  static class Inner
 {
    }
}
```

Java static nested class example

File: Outer.java

```
public class Outer
{
  static int data = 100;
  String str = "APEC";
  void display()
  {
    System.out.println("Given String is APEC");
  }
  static class Inner
  {
   void msg()
   {
      System.out.println("Given data : "+ data);
    }
  }

  public static void main(String args[])
  {
  Outer out = new Outer();
  out.display();

  Outer.Inner obj = new Outer.Inner();
  obj.msg();
  }
```

**Outer class**

**static data is accessed both in outer class and inner class**

**Static Inner class**

**Object for outer class**

**Object for Inner class**

Output:
Given String is APEC
Given data : 100

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are:

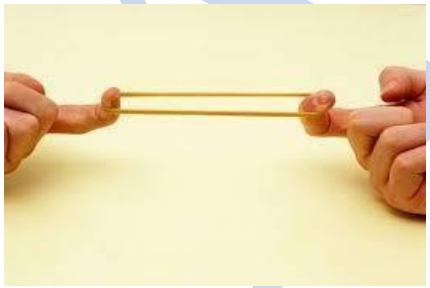
- Nested classes represent a special type of relationship that is it can *access all the members (data members and methods) of outer class* including private.
- Nested classes are used to develop more *readable and maintainable* code because it logically group classes and interfaces in one place only.
- Code Optimization: It requires less code to write.

## Array Lists in Java

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size; however the size can increase if collection grows or shrunk if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList cannot be used for primitive types, like int, char, etc. We need a wrapper class for such cases.



**Array Lists are like Rubber bands fixed**.



**Arrays are like ropes. They are**

Constructors in Java ArrayList:

- ***ArrayList():*** This constructor is used to build an empty array list

- ***ArrayList(Collection c):*** This constructor is used to build an array list initialized with the elements from collection c
- ***ArrayList(int capacity):*** This constructor is used to build an array list with initial capacity being specified.

Methods in Java ArrayList:

| Method | Description |
|---|---|
| clear() | Remove all the elements from the list. |
| add(int index, Object element) | Insert a specific element at a specific position index in a list. |
| trimToSize() | Trim the capacity of the instance of the ArrayList to the list's current size. |
| indexOf(Object O) | Index the first occurrence of a specific element is either returned, or -1 in case the element is not in the list. |
| lastIndexOf(Object O) | Index the last occurrence of a specific element is either returned, or -1 in case the element is not in the list. |
| clone() | Return a shallow copy of an ArrayList. |
| isEmpty() | Check if the list is empty or not. |
| contains(Object o) | Returns true if this list contains the specified element |
| get(int index) | Returns the element of a specified position within the list. |
| toArray() | Return an array containing all of the elements in the list in correct order. |
| toArray(Object[] O) | Returns an array containing all of the elements in this list in the correct order same as the previous method. |
| addAll(Collection C) | Append all the elements from a specific collection to the end of the mentioned list, in such a order that the values are returned by the specified collection's iterator. |
| add(Object o) | Append a specified element to the end of a list. |
| addAll(int index, Collection C) | Insert all of the elements starting at the specified position from a specific collection into the mentioned list. |
| remove(int index) | Remove an element at a specified index from ArrayList |

Syntax:

$ArrayList\ al = new\ ArrayList();$ → Old method

New Generic way of creating Arraylists is:

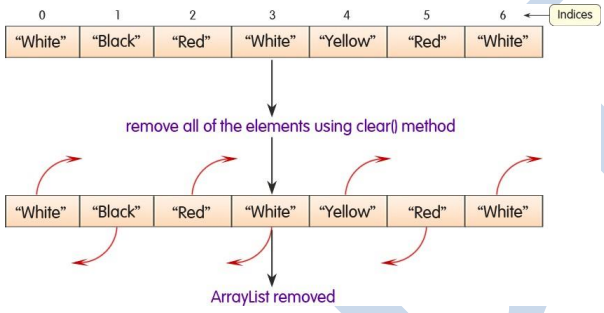
$\boldsymbol{ArrayList < String > al = new\ ArrayList < String > ();}$

Methods in the arraylist are invoked using:

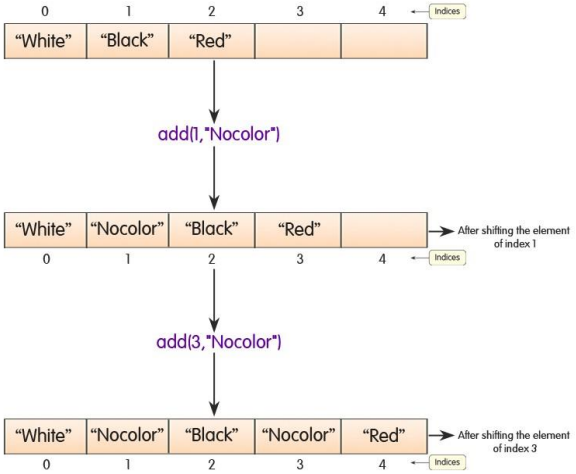
$\boldsymbol{ArrayList.Methodname()}$

**Pictorial representations of some methods in arraylists**:

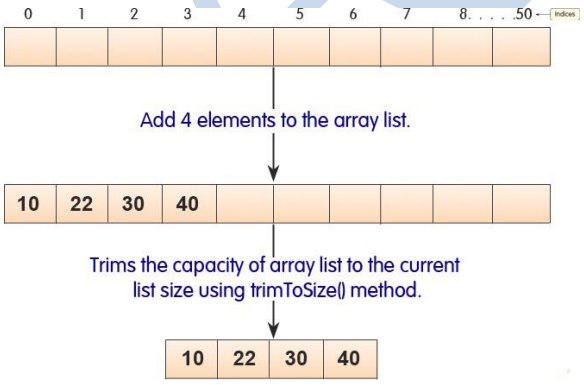
Syntax: ArrayList**.clear()**



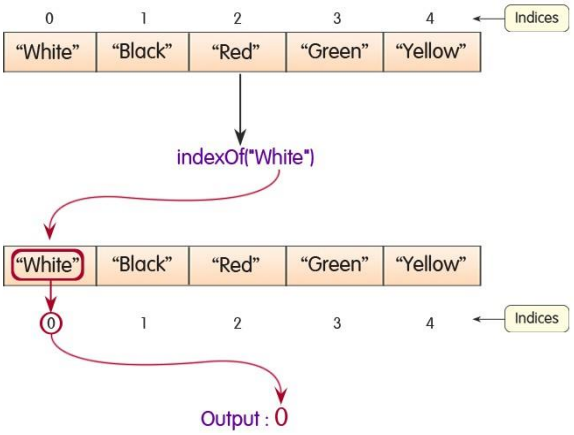

Syntax: ArrayList**.add(int index, E ele)**




Syntax: ArrayList**. trimToSize()**

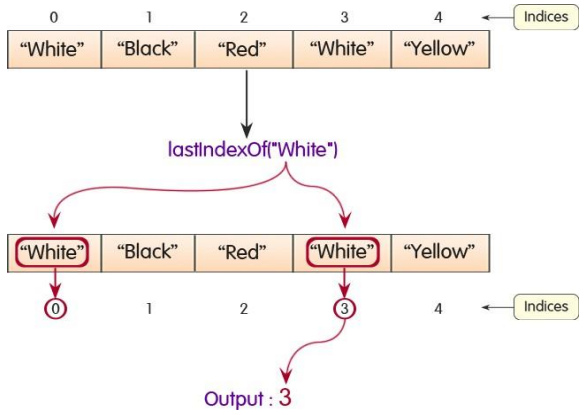



Syntax: ArrayList**. indexOf(Object o)**

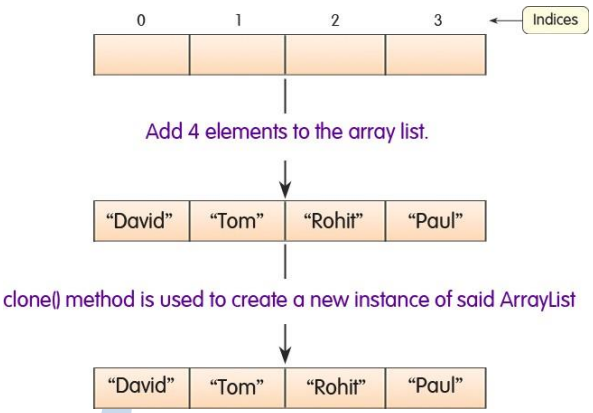


Syntax:

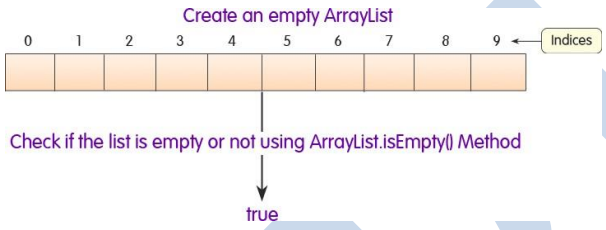
ArrayList.**lastIindexOf(Object o)**



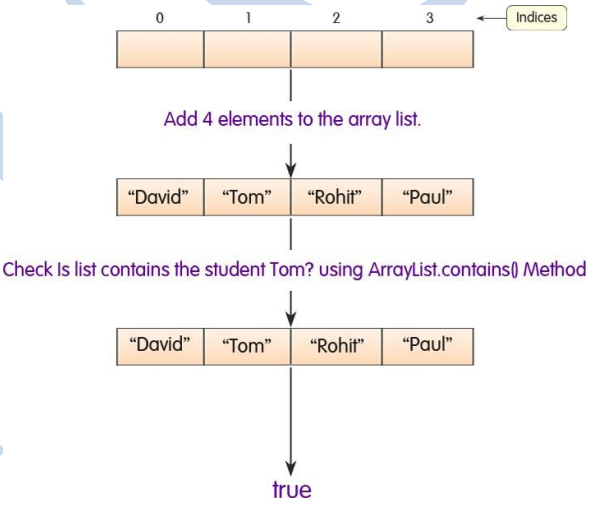

Syntax: ArrayList.**clone()**



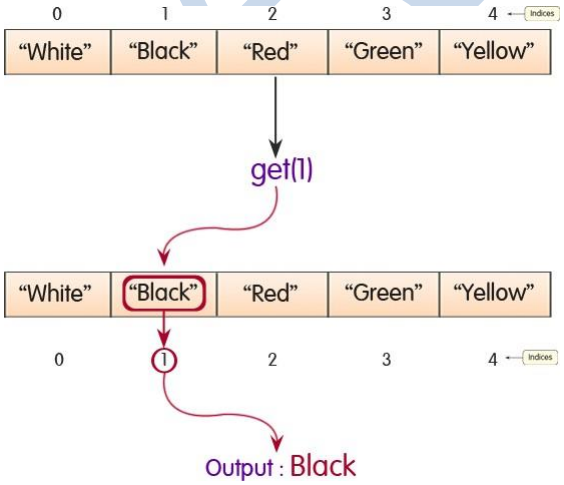

Syntax: ArrayList.**isEmpty()**



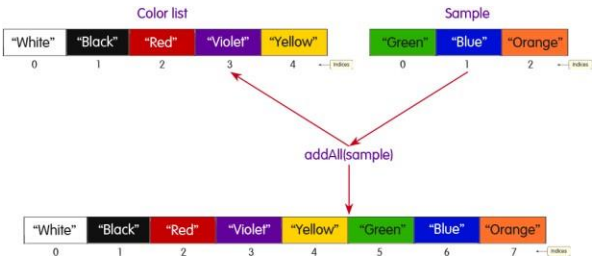

Syntax: ArrayList.**contains(Object o)**



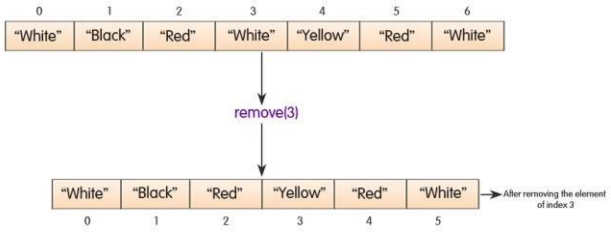

Syntax: ArrayList.**get()**



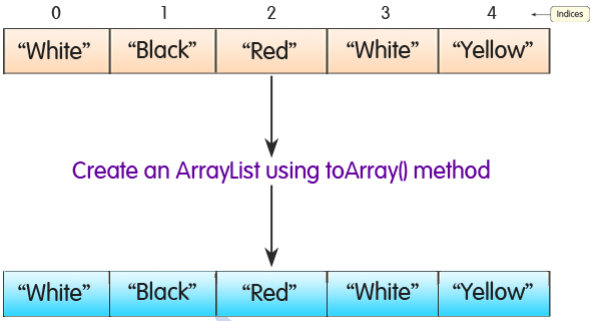

Syntax: ArrayList.**addAll()**



Syntax: ArrayList. **remove(index)**




Syntax: ArrayList.**toArray(T[] a)**




APEC

```
Filename : Test.java
import java.util.*;
public class Test {
   public static void main(String[] args) {

      ArrayList<String> color_list = new ArrayList<String>(10);

      color_list.add("White");
      color_list.add("Black");
      color_list.add("Red");

      System.out.println("Given List:");
      for (int i = 0; i < 3; i++)
         System.out.println(color_list.get(i));

      color_list.add(1,"Green");
      System.out.println("Green color is inserted:");
      for (int i = 0; i < 4; i++)
         System.out.println(color_list.get(i));

      int i = color_list.indexOf("Black");
      System.out.println("Black color is at position :" +(i+1));

      color_list.remove("Black");
      System.out.println("After removing Black from the list");
      for (i = 0; i < 3; i++)
         System.out.println(color_list.get(i));

      ArrayList<String> animal_list = new ArrayList<String>(5);
      animal_list.add("Lion");
      animal_list.add("Tiger");
      animal_list.add("Dog");

      color_list.addAll(animal_list);
      System.out.println("Combining color_list and animal_list");
      for (i = 0; i < 6; i++)
         System.out.println(color_list.get(i));

      String[] str = new String[10];
      str = color_list.toArray(str);

      System.out.println("Array is printed");
      for (String s : str)
         System.out.println(s);

   }
}
```

**Create an empty array list, color_list**

**Use add() method to add values in the list**

**Print out the colors in the ArrayList**

**Append Green color at 2nd position**

**Index of the Black color is determined**

**Black color is removed from the list**

**Create another array list, animal_list**

**Combine color_list and animal_list**

**New array, str is created. Arraylist is converted to array**

**Array is printed**

Output:
**Given List:**
White
Black
Red
**Green color is inserted:**
White
Green
Black
Red
**Black color is at position :3**
**After removing Black from the list**
White
Green
Red
**Combining color_list and animal_list**
White
Green
Red
Lion
Tiger
Dog
**Array is printed**
White
Green
Red
Lion
Tiger
Dog
null
null
null
null

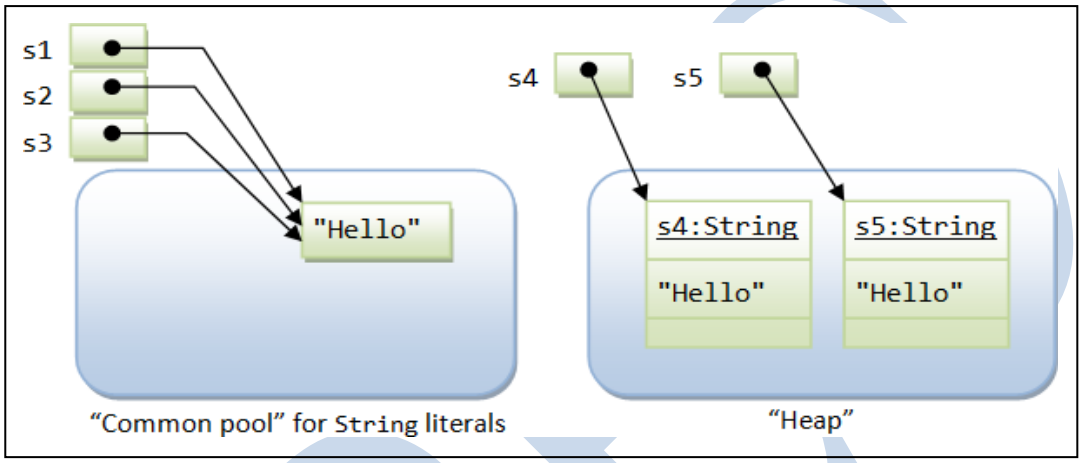
## Java Strings

String is a sequence of characters. In Java programming language, strings are treated as objects. The $\boldsymbol{java.\ lang.\ String}$ class is used to create string object. There are two ways to create a String object:

- Implicit construction using string literal : Java String literal is created by using double quotes.
    - Example: $\boldsymbol{String\ s} =$ "$\boldsymbol{Welcome}$";
- Explicit construction using new keyword : Java String is created by using a keyword "new".
    - Example: $\boldsymbol{String\ s = new\ String}$("$\boldsymbol{Welcome}$");

Difference between string literal and new operator:

```
String s1 = "Hello";                  // String literal
String s2 = "Hello";                  // String literal
String s3 = s1;                       // same reference
String s4 = new String("Hello");      // String object
String s5 = new String("Hello");      // String object
```




Java has provided a special mechanism for keeping the String literals - in a so-called string common pool. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to conserve storage for frequently-used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in heap even if two String objects have the same contents.

Defining and initializing Strings:

Simple example

```
String s1 = "Welcome to Java!";
String s2 = new String("Welcome to Java!");          //same as s1
```

Numbers as strings

```
String s3 ="12345";
String s4 = new String(s3);          //s4 will hold same value as s3
```

Char array as strings

```
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String s5= new String(helloArray);
```

Simple Example for Defining and initializing Strings

Filename : Test.java

```
public class Test
{  public static void main(String args[])
 {
   String s1 = "Welcome to Java!";
   System.out.println("Using String Literal : " + s1);
```

→ **Creating string by java string literal**

```
   String s2 = new String("Welcome to Java!");
   System.out.println("Using new operator : " + s2);
```

→ **Creating java string by new keyword**

```
   String s3 ="12345";
```

→ **Numbers as a string**

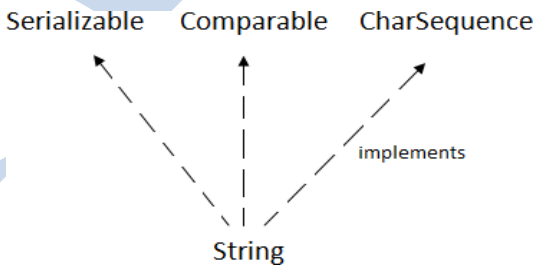
```
   String s4 = new String(s3);
   System.out.println("Printing numbers as a string : " + s3);
   System.out.println(s4);
   char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
   String s5 = new String(helloArray);
```

→ **Converting char array to string**

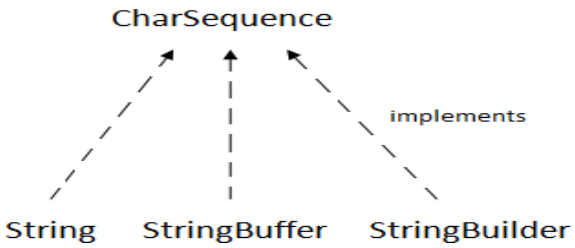
```
   System.out.println("Character Array converted to string : " + s5);
 } }
```

Output:

```
Using String Literal : Welcome to Java!
Using new operator : Welcome to Java!
Printing numbers as a string : 12345
12345
Character Array converted to string : hello.
```




Java String class implements three interfaces, namely – Serializable, Comparable and CharSequence. The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.




In Java, Strings are immutable. By immutable means, Strings are constant; their values cannot be changed after they are created. For mutable string, you can use StringBuffer and StringBuilder classes.

Java String Methods:

| Method | Description | Example |
|---|---|---|
| **length()** | It tells the length of the string by counting the total number of characters present in the String. | String s1="hello";<br>s1.length() = 5 |
| **compareTo()** | Compares the given string with current string.<br>If s1 > s2, it returns a positive number<br>if s1 < s2, it returns a negative number<br>if s1 == s2, it returns 0 | String s1="hello";<br>String s2="hello";<br>String s3="hai";<br>s1.compareTo(s2) = 0<br>s1.compareTo(s3) = 4 |
| **concat()** | Combines a specific string at the end of another string. It is like appending another string. | String s1="Hi";<br>String s2 =”Hello”;<br>s1.concat(" hru!") = Hi hru!<br>s1.concat(s2) = HiHello |
| **IsEmpty()** | Checks whether the String contains anything or not. | String s1="hello";<br>s1.isEmpty() = false<br>String s1="";<br>s1.isEmpty() = true |
| **Trim()** | Removes the leading and trailing spaces. | String s1="Hi  ";<br>String  s2="hru";<br>s1 + s2 = Hi  hru<br>s1.trim() + s2 = Hihru |
| **toLowerCase()** | Converts all the characters of the String to lower case. | String s1="H R You?”;<br>s1.toLowerCase() = h r you? |
| **toUpper()** | Converts all the characters of the String to upper case. | String s1="H R You?”;<br>s1.toUpperCase() = H R YOU? |
| **ValueOf()** | Converts different types of values into string. | int value=20;<br>String s1;<br>s1=String.valueOf(value);<br>s1+18 = 2018 |
| **replace()** | Replace all the old characters with the new characters. | String s1="hello hai hi”;<br>s1 = s1.replace('h','t');<br>s1= tello tai ti<br><br>String s1="Hi, welcome";<br>s1=s1.replace("Hi","CSE");<br>s1 = CSE, welcome |

| Method | Description | Example |
|---|---|---|
| **contains()** | Searches the sequence of characters in the string. If found, return true else return false | String s1 =" hello hai hi!";<br>s1.contains("hi") = true<br>s1.contains("helloo") = false<br>s1.contains("hru") = false |
| **equals()** | Compares the two given strings. If matched, return true else return false.<br>**equalsIgnoreCase ->** Ignore the cases and campare the strings. | String s1="hello";<br>String s2="Hello";<br><br>s1.equals(s2) = false<br>s1.equalsIgnoreCase(s2)= true |
| **endsWith()** | Checks if this string ends with the given suffix. | String s1="hello how are you";<br>s1.endsWith("u") = true<br>s1.endsWith("you") = true<br>s1.endsWith("how") = false |

Java program to sort the string in alphabetical order:

Filename : Test.java

```
import java.util.Arrays;
public class Test
{
   public static void main(String[] args)
   {
      String str = "javaprogramming";
      char a[] = str.toCharArray();
      System.out.println("Input String : " + str);
      Arrays.sort(a);

      System.out.print("Sorted String : " );
      for (int i = 0; i < str.length(); i++)
        System.out.print(a[i]);
   }
}
```

**Import the utility arrays to use the sort function in the program**

**Given string is converted to character array and then it is sorted**

**Sort the character array using sort function**

Output:

```
Input String : javaprogramming
Sorted String : aaaggijmmnoprrv
```

Java program to sort the numbers in alphabetical order:

Filename : Test.java

```
import java.util.Arrays;

public class Test
{    public static void main(String[] args)
   {
      int num[] = {100,67,89,32,65,75};

      System.out.print("Input Array Elements: ");
      for (int i = 0; i < num.length; i++)
         System.out.print(num[i] + " ");

      Arrays.sort(num);

      System.out.println();
      System.out.print("Sorted Array : " );
      for (int i = 0; i < num.length; i++)
        System.out.print(num[i] + " ");
   } }
```

Finding the array size using length

Sorting the Integer array

Output:
Input Array Elements: 100 67 89 32 65 75
Sorted Array : 32 65 67 75 89 100

Java program to sort the strings in alphabetical order:

Filename : Test.java

```
import java.util.Arrays;

public class Test
{
   public static void main(String[] args)
   {
      String str[] = {"thankyou", "hai", "every", "babuji", "god"};
      System.out.print("Input Strings : ");
      for (int i = 0; i < str.length; i++)
         System.out.print(str[i] + " ");

      Arrays.sort(str);

      System.out.println();
      System.out.print("Sorted strings : " );
      for (int i = 0; i < str.length; i++)
        System.out.print(str[i] + " ");
   }
}
```

Declaring the string array and initialized with the values

Sort the string array

Output:
Input Strings: thankyou hai every babuji god
Sorted strings : babuji every god hai thankyou