1. Inheritance
    - Use of inheritance
        - Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.
        - Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.
        - In the terminology of Java, a class that is inherited is called a superclass.
        - The class that does the inheriting is called a subclass.
        - To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.
        - The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
// body of class
}

// A simple example of inheritance.
// Create a superclass.
class A {
        int i, j;
                void showij() {
                                System.out.println("i and j: " + i + " " + j);
                                }
        }
// Create a subclass by extending class A.
class B extends A {
        int k;
                void showk() {
                                System.out.println("k: " + k);
                        }
                void sum() {
                                System.out.println("i+j+k: " + (i+j+k));
                        }
                }

class SimpleInheritance {
                public static void main(String args[]) {
                A superOb = new A();
                B subOb = new B();
                // The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

- Member Access and Inheritance
    - Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

```
// Create a superclass.
class A {
        int i; // public by default
        private int j; // private to A
        void setij(int x, int y)
        {
        i = x;
        j = y;
        }
        }
// A's j is not accessible here.
class B extends A {
        int total;
        void sum() {
        total = i + j; // ERROR, j is not accessible here
        }
         }
```

```
class Access {
public static void main(String args[])
{
        B subOb = new B();
        subOb.setij(10, 12);
        subOb.sum();
        System.out.println("Total is " + subOb.total);
}
}
```

- Method overriding
  - In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.

```
// Method overriding.
class A
{
        int i, j;
        A(int a, int b)
        {
        i = a;
        j = b;
        }
        // display i and j
        void show()
        {
        System.out.println("i and j: " + i + " " + j);
        }
}
class B extends A {
        int k;
        B(int a, int b, int c) {
        super(a, b);
        k = c;
        }
// display k – this overrides show() in A
        void show() {
        System.out.println("k: " + k);
                        }
}
```

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

- A Superclass Variable Can Reference a Subclass Object
  - A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class Base
{
        void Msg1()
        {
                System.out.print("Hello");
        }
}

class Dirived extends Base
{
        public void Msg1()
        {
        System.out.print("Hi");
        }
}

class Dirived2 extends Base
{
        public void Msg1()
        {
                System.out.print("Byeee");
        }

}


class DemoRef
{
        public static void main(String args[])
        {
        Base ref;
        Dirived obj = new Dirived();
        ref = obj;
        ref.Msg1();
        }
    }
```

- Using super
    - Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.
    - Usage of Java super Keyword
        - super can be used to refer immediate parent class instance variable.
            - We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of
Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

- super can be used to invoke immediate parent class method.
    - The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

      ```
      class Animal{
      void eat(){System.out.println("eating...");}
      }
      class Dog extends Animal{
      void eat(){System.out.println("eating bread...");}
      void bark(){System.out.println("barking...");}
      void work(){
      super.eat();
      bark();
      }
      }
      class TestSuper2{
      public static void main(String args[]){
      Dog d=new Dog();
      d.work();
      }}
      ```

      In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

      To call the parent class method, we need to use super keyword.

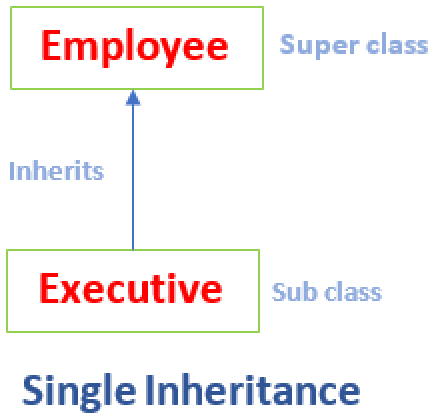
- super() can be used to invoke immediate parent class constructor.

- The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

- Object Class
  - The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
  - If a class does not extend any other class then it is a direct child class of Object and if extends another class then it is indirectly derived.
  - The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.
  - Using Object Class Methods
    - The Object class provides multiple methods which are as follows:
      - tostring() method
      - hashCode() method
      - equals(Object obj) method
      - finalize() method
      - getClass() method
      - clone() method
      - wait(), notify() notifyAll() methods

- types of inheritance
  Java supports the following four types of inheritance:
  - Single Inheritance
    - In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as simple inheritance.

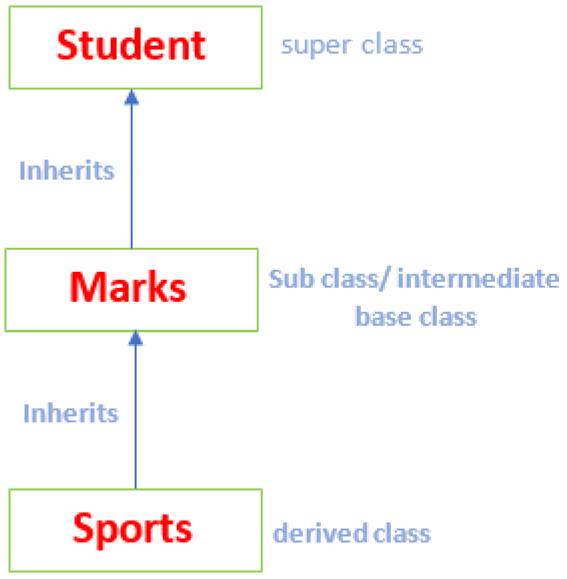


    - In the above figure, Employee is a parent class and Executive is a child class. The Executive class inherits all the properties of the Employee class.

```
class Employee
{
float salary=34534*12;
}
public class Executive extends Employee
{
float bonus=3000*6;
public static void main(String args[])
{
Executive obj=new Executive();
System.out.println("Total salary credited: "+obj.salary);
System.out.println("Bonus of six months: "+obj.bonus);
}
}
```

  - Multi-level Inheritance
    - In multi-level inheritance, a class is derived from a class which is also derived from another class is called multi-level inheritance. In simple words, we can say that a class that has more than one parent class is called multi-level inheritance. Note that the classes

must be at different levels. Hence, there exists a single base class and single derived class but multiple intermediate base classes.



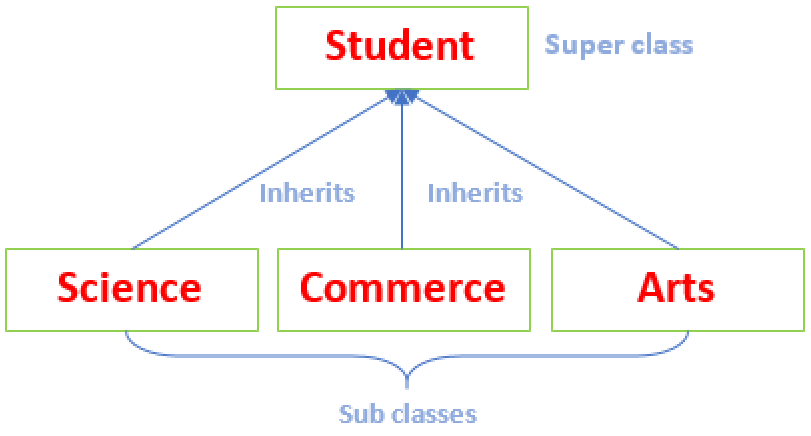
Multi-level Inheritance

- In the above figure, the class Marks inherits the members or methods of the class Students. The class Sports inherits the members of the class Marks. Therefore, the Student class is the parent class of the class Marks and the class Marks is the parent of the class Sports. Hence, the class Sports implicitly inherits the properties of the Student along with the class Marks.

```
//super class
class Student
{
int reg_no;
void getNo(int no)
{
reg_no=no;
}
void putNo()
{
System.out.println("registration number= "+reg_no);
}
}
//intermediate sub class
class Marks extends Student
{
```

```
float marks;
void getMarks(float m)
{
marks=m;
}
void putMarks()
{
System.out.println("marks= "+marks);
}
}
//derived class
class Sports extends Marks
{
float score;
void getScore(float scr)
{
score=scr;
}
void putScore()
{
System.out.println("score= "+score);
}
}
public class MultilevelInheritanceExample
{
public static void main(String args[])
{
Sports ob=new Sports();
ob.getNo(0987);
ob.putNo();
ob.getMarks(78);
ob.putMarks();
ob.getScore(68.7);
ob.putScore();
}
}
```

- Hierarchical Inheritance
  - If a number of classes are derived from a single base class, it is called hierarchical inheritance.

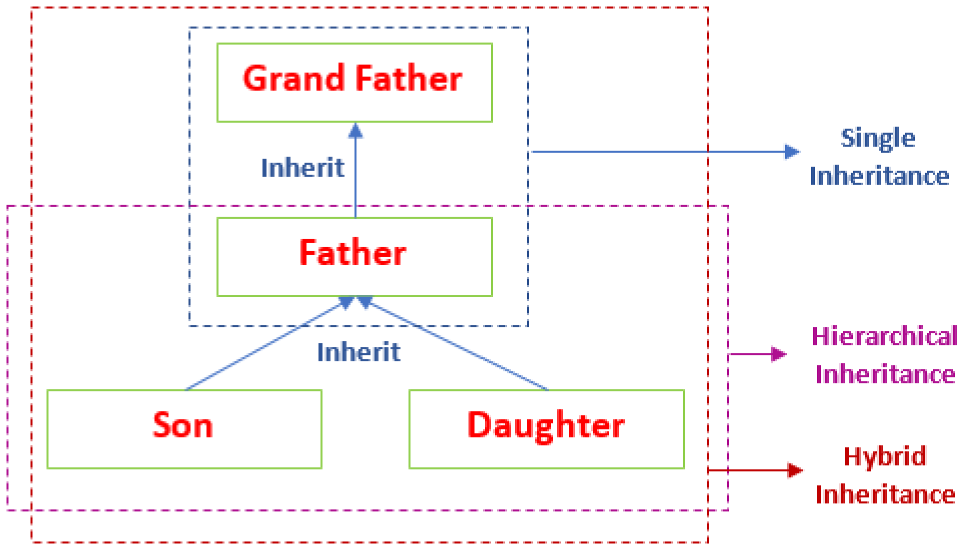


Hierarchical Inheritance

  - In the above figure, the classes Science, Commerce, and Arts inherit a single parent class named Student. Let's implement the hierarchical inheritance mechanism in a Java program.

```
//parent class
class Student
{
public void methodStudent()
{
System.out.println("The method of the class Student invoked.");
}
}
class Science extends Student
{
public void methodScience()
{
System.out.println("The method of the class Science invoked.");
}
}
class Commerce extends Student
{
public void methodCommerce()
{
System.out.println("The method of the class Commerce invoked.");
}
}
class Arts extends Student
{
public void methodArts()
{
System.out.println("The method of the class Arts invoked.");
}
}
public class HierarchicalInheritanceExample
{
public static void main(String args[])
{
Science sci = new Science();
Commerce comm = new Commerce();
Arts art = new Arts();
//all the sub classes can access the method of super class
sci.methodStudent();
comm.methodStudent();
art.methodStudent();
}
}
```

- Hybrid Inheritance
  - Hybrid means consist of more than one. Hybrid inheritance is the combination of two or more types of inheritance.



Hybrid Inheritance

- In the above figure, GrandFather is a super class. The Father class inherits the properties of the GrandFather class. Since Father and GrandFather represents single inheritance. Further, the Father class is inherited by the Son and Daughter class. Thus, the Father becomes the parent class for Son and Daughter. These classes represent the hierarchical inheritance. Combinedly, it denotes the hybrid inheritance.

```
//parent class
class GrandFather
{
public void show()
{
System.out.println("I am grandfather.");
}
}
//inherits GrandFather properties
class Father extends GrandFather
{
public void show()
{
System.out.println("I am father.");
}
}
//inherits Father properties
class Son extends Father
{
public void show()
{
System.out.println("I am son.");
}
}
//inherits Father properties
public class Daughter extends Father
{
public void show()
{
System.out.println("I am a daughter.");
}
public static void main(String args[])
{
Daughter obj = new Daughter();
obj.show();
}
}
```

- **Multiple Inheritance (not supported, just for info.)**
  Java does not support multiple inheritances due to ambiguity. For example, consider the following Java program.

```
class Wishes
{
void message()
{
System.out.println("Best of Luck!!");
}
}
class Birthday
{
void message()
{
System.out.println("Happy Birthday!!");
}
}
public class Demo extends Wishes, Birthday  //considering a scenario
{
public static void main(String args[])
{
Demo obj=new Demo();
//can't decide which classes' message() method will be invoked
obj.message();
}
}
```

The above code gives error because the compiler cannot decide which message() method is to be invoked. Due to this reason, Java does not support multiple inheritances at the class level but can be achieved through an interface.

- Abstract class and Abstract method
    - Abstraction is a process of hiding the implementation details and showing only functionality to the user.
    - Abstraction lets you focus on what the object does instead of how it does it.
    - **Abstract class**
    - A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
        - An abstract class must be declared with an abstract keyword.
        - It can have abstract and non-abstract methods.
        - It cannot be instantiated.
        - It can have constructors and static methods also.
        - It can have final methods which will force the subclass not to change the body of the method.

        ```
        abstract class A{}
        ```
    - Abstract method
        - A method which is declared as abstract and does not have implementation is known as an abstract method.

        ```
        abstract void printStatus();//no method body and abstract
        ```

        Example:

        ```
        abstract class Bike{
          abstract void run();
        }
        class Honda4 extends Bike{
        void run(){System.out.println("running safely");}
        public static void main(String args[]){
         Bike obj = new Honda4();
         obj.run();
        }
        }
        ```

Example:

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+"
%");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+"
%");
}}
```

- Dynamic method dispatch
    - Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
    - Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
    - It is based on principal of superclass reference variable can refer to a subclass object.
    - When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

      Example:

      ```
      class A {
              void callme() {
              System.out.println("Inside A's callme method");
              }
      }
      ```

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

class Dispatch {
        public static void main(String args[]) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C
        A r; // obtain a reference of type A
        r = a; // r refers to an A object
        r.callme(); // calls A's version of callme
        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme
        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
}
}
```

- final keyword (variable, method, class) (Refer Unit 1)

2. Interface
   - Defining an Interface
     - Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
     - Using the keyword interface, you can fully abstract a class' interface from its implementation.
     - Using interface, you can specify what a class must do, but not how it does it.
     - An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

- Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code.
- name is the name of the interface, and can be any valid identifier.
- the methods which are declared have no bodies.
- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class.
- All methods and variables are implicitly public if the interface, itself, is declared as public.

```
interface Callback {
void callback(int param);
}
```

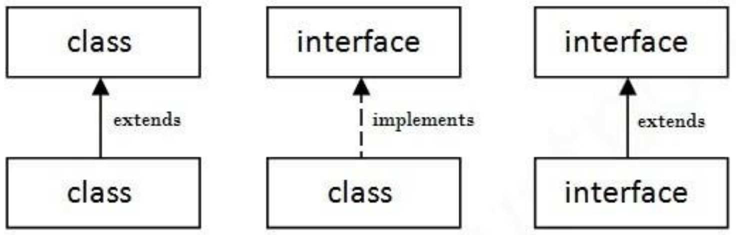
- Implementing Interfaces
    - Once an interface has been defined, one or more classes can implement that interface.
    - To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

```
access class classname [extends superclass]
[implements interface [,interface...]] {
// class-body
}
```

Example:

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

- Applying Interfaces



```
//Interface declaration: by first user
interface Drawable
        {
        void draw();
        }

//Implementation: by second user
class Rectangle implements Drawable
        {
        public void draw()
        {
        System.out.println("drawing rectangle");
        }
        }

class Circle implements Drawable
        {
        public void draw()
        {
```

```
        System.out.println("drawing circle");
        }
}

class TestInterface1
{
        public static void main(String args[])
        {
        Drawable d=new Circle();//In real scenario, object is provided by method
        d.draw();
        }
}
```

- Variables in Interfaces

  You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.

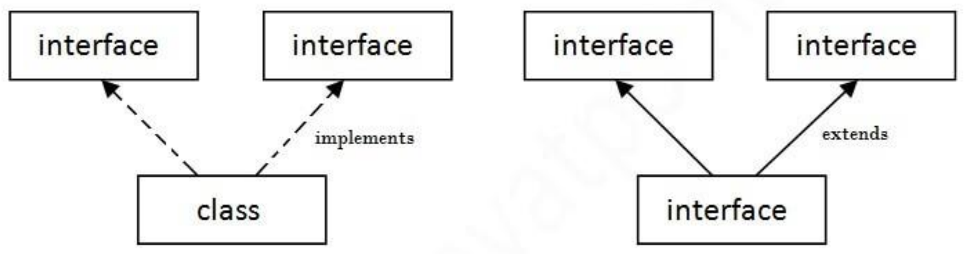
```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}

class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
```

}

Multiple inheritance in java by interface

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?



Multiple Inheritance in Java

```
interface Printable
        {
        void print();
        }
interface Showable
        {
        void show();
        }

class A implements Printable,Showable
        {
        public void print(){System.out.println("Hello");}
        public void show(){System.out.println("Welcome");}

        public static void main(String args[])
        {
        A obj = new A();
        obj.print();
        obj.show();
         }
        }
```

- Extended Interfaces
  A class implements an interface, but one interface extends another interface.

```
interface Printable
        {
        void print();
        }
interface Showable extends Printable
        {
        void show();
        }

class TestInterface4 implements Showable
        {
        public void print(){System.out.println("Hello");}
        public void show(){System.out.println("Welcome");}

        public static void main(String args[])
        {
        TestInterface4 obj = new TestInterface4();
        obj.print();
        obj.show();
         }
        }
```

- interface vs abstract class vs concrete class
  The simplest difference between all three of these classes is:

  an interface has no method implementation
  abstract classes may or may not have method implementation
  concrete classes MUST have method implementation