There are 2 types of Relationship in Java:

1. "is-a" relationship

Student "is-a" Human
Plane "is-a" Vehicle
Deer "is-a" Animal

It is achieved through INHERITANCE.

Inheritance refers to the process of coding a project as a hierarchy of classes.
It is achieved by using "extends" keyword.

2. "has-a" relationship

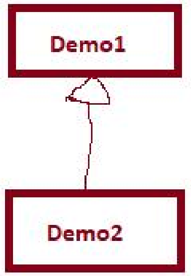
Student "has-a" Book
Plane "has-a" Engine
Deer "has-a" Heart

It is achieved through ASSOCIATION.

Types of Inheritance

Rules of Inheritance

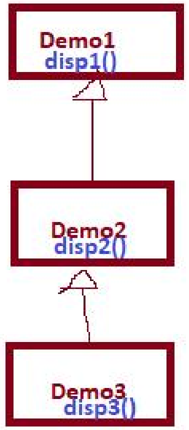
Single Inheritance



In SI, a class inherits from a single parent class.

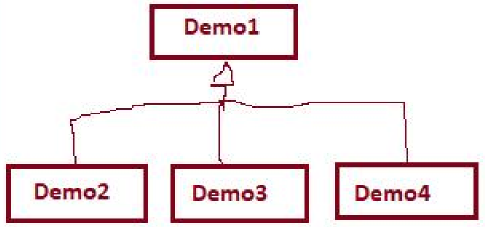
2. Constructors do not participate in Inheritance.

Multilevel Inheritance



```
Demo3 d3 = new Demo3();

d3.disp1();
d3.disp2();
d3.disp3();
```

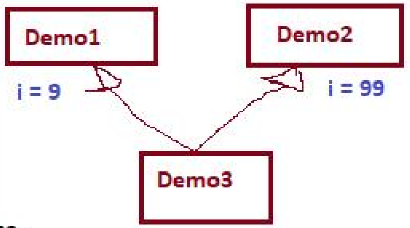
Hierarchical Inheritance



1. private members do not participate in inheritance.

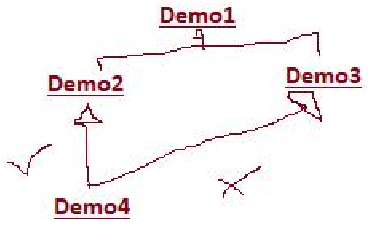
```
class User
{
   private int acc_no = 1234;
   private int pwd = 8679;
}

class Hacker  extends User
{
   void hacking()
   {
         S.o.p(acc_no);   ✗
         S.o.p(pwd);
   }
}
```

Multiple Inheritance



Hybrid Inheritance

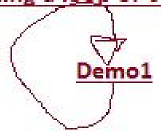
"Conditionally"

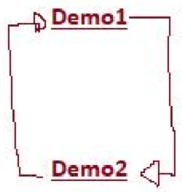


Cyclic Inheritance

-> A class inherits itself

-> Two classes inherits eachother forming a loop or cycle

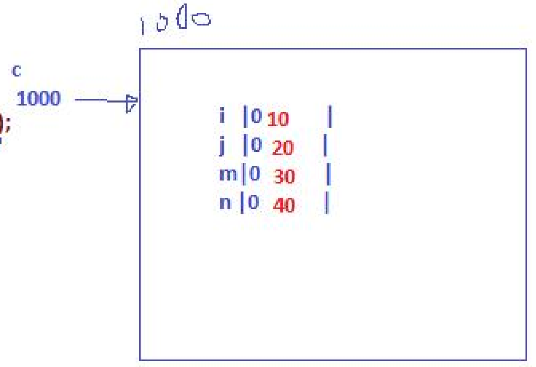




2. Constructors are not inherited but they are executed.
Because of the super().

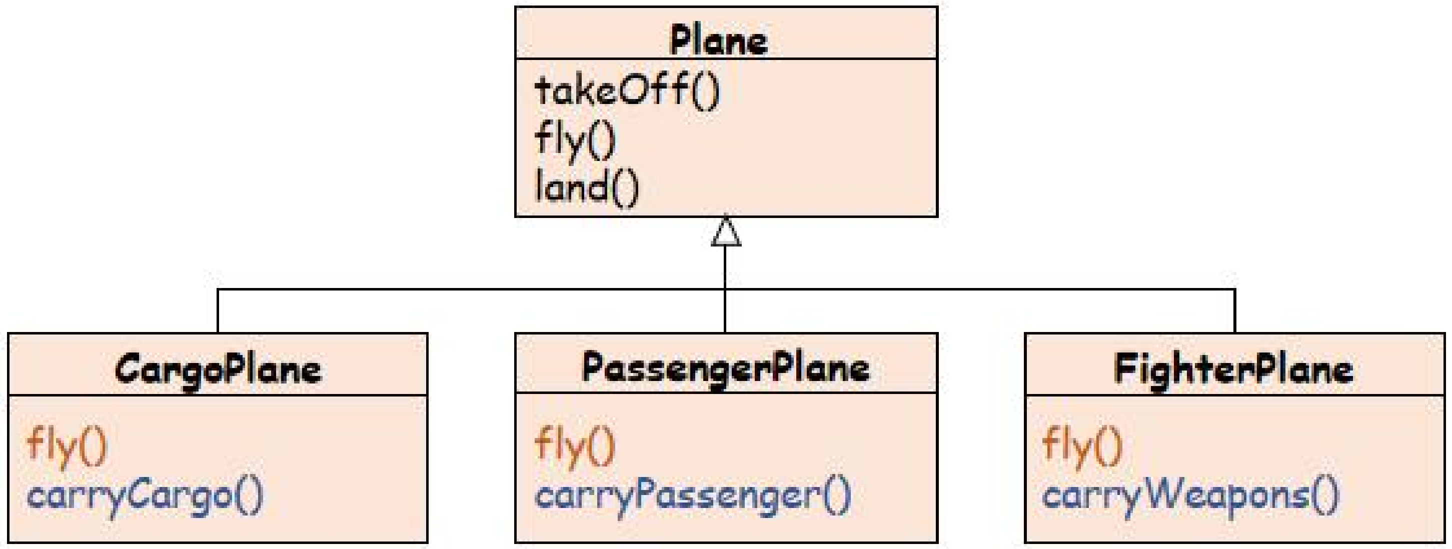
```
class Object
{
    public Object(){
    }
}
class Parent  extends Object
{   int i, j;
    public Parent()
    {     super();
       i = 10; j = 20;
      S.o.p("Parent constructor");
    }
}

class Child extends Parent
{
     int m, n;


    public Child()
    { super();
      m = 30; n = 40;
      S.o.p("Child constructor");
    }
}
```

```
Child c = new Child();
```



Output:
Parent constructor
Child constructor

PLANE HIERARCHY



```
class Plane
{
   public void takeOff()
   {
      S.o.p("Plane is taking off");
   }
   public void fly()
   {
      S.o.p("Plane is flying");
   }
   public void land()
   {
      S.o.p("Plane is landing");
   }
}

class CargoPlane extends Plane
{
   public void fly()
   {
      S.o.p("CargoPlane is flying at lower heights");
   }
   public void carryCargo()
   {
      S.o.p("CargoPlane is caryying cargo");
   }
}
```

```
class PassengerPlane extends Plane
{
   public void fly()
   {
      S.o.p("PassengerPlane is flying at medium heights");
   }
   public void carryPassenger()
   {
      S.o.p("PassengerPlane is caryying passengers");
   }
}

class FighterPlane extends Plane
{
   public void fly()
   {
      S.o.p("FighterPlane is flying at greater heights");
   }
   public void carryWeapons()
   {
      S.o.p("FighterPlane is caryying weapons");
   }
}
```

```
class Launch
{
   p s v main(...)
   {
          CargoPlane cp = new CargoPlane();
          PassengerPlane pp = new PassengerPlane();
          FighterPlane fp = new FighterPlane();

          cp.takeOff();              //Plane is taking off
Inherited cp.fly(); Overridden //CargoPlane is flying at lower heights
          cp.carryCargo();         //CargoPlane is carrying cargo
          cp.land();                  //Plane is landing

                                         Specialized

          pp



          fp

   }
}
```

There are 3 types of Methods in Inheritance:

1. Inherited Methods
2. Overridden Methods
3. Specialized Methods

```
//Dependent class
class Address
{
    int doorNo;
    int streetNo;
    String locality;
    String city;
    String state;
    String country;
}
```

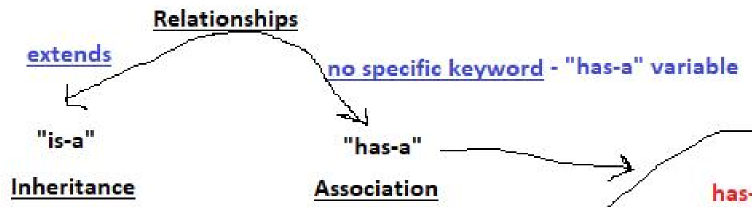
```
//Target class

class Student  ~~extends~~ Address
{
private   String name;
private   int roll_no;
private   float cgpa;
                              HAS-A Variable
private   Address addr;  //1:1 association

            <OR>
          Address[] addr;  //1:M association

          public void setStudent(String name, int roll_no, float cgpa, Address addr)
          {
              this.name = name;
              this.roll_no = roll_no;
              this.cgpa = cgpa;
              this.addr = addr;
          }
          public Student(String name, int roll_no, float cgpa, Address addr)
          {
              this.name = name;
              this.roll_no = roll_no;
              this.cgpa = cgpa;
              this.addr = addr;
          }
}
```

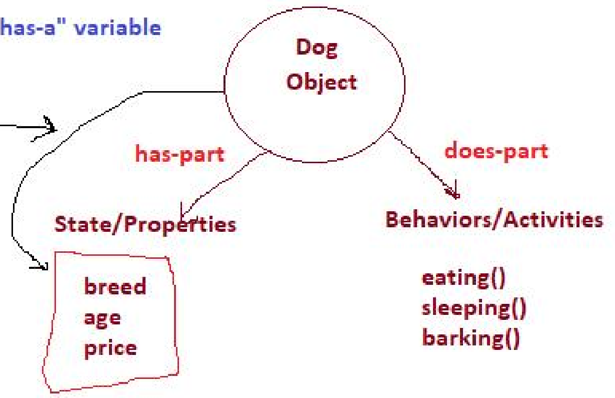
1:1
1:M
M:1
M:M

Dependency Injection:

1. Setters
2. Constructors

S.o.p(s.addr.doorNo);

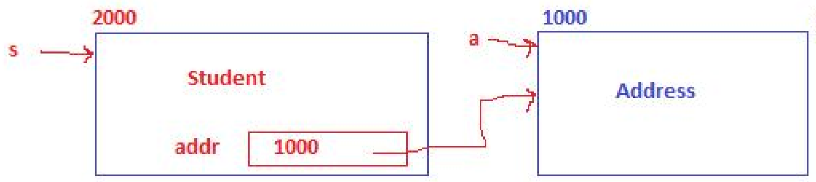




Fields[Variables/Instance/Data Members/Has-A Variables]

```
class Dog
{
    String breed;
    float age;
    int price;
}
```

```
Address a = new Address(32, 12, "Rajajinagar", "Bang","Kar","India");
Student s = new Student("Aayush", 118, 9.6f, a);
```



```
{
        public static void main(String[] args)
        {
                Child1 c1 = new Child1();
                Child2 c2 = new Child2();
                Child3 c3 = new Child3();

                Parent ref;

                ref = c1;
                ref.cry(); <- ~~Child1~~
                ref.eat(); ✗

                ref = c2;
                ref.cry();
                ref.eat(); ✗

                ref = c3;
                ref.cry();
                ref.eat(); ✗
        }
}
```

Java Compiler

Compilation

(Type of Reference)

Method Overloading

Compile-Time Polymorphism/ Static Polymorphism

JVM

Runtime

(Type of Object)

Method Overriding

Runtime Polymorphism / Dynamic Polymorphism

```
float a = 45.5f;

int b;

b = a;     X

b = (int)a;

S.o.p(a);    //45.5

ref.eat(); X

((Child1)(ref)).eat();     "DOWNCASTING"

Parent
```

```
class Launch
{
        public static void main(String[] args)
        {
                CargoPlane cp = new CargoPlane();
                PassengerPlane pp = new PassengerPlane();
                FighterPlane fp = new FighterPlane();
```

300

NON-POLYMORPHIC VERSION

```
cp.takeOff();
cp.fly();
cp.land();

pp.takeOff();
pp.fly();
pp.land();

fp.takeOff();
fp.fly();
fp.land();
```

POLYMORPHIC VERSION Without Adv:

```
Plane ref;

ref = cp;
ref.takeOff();
ref.fly();
ref.land();

ref = pp;
ref.takeOff();
ref.fly();
ref.land();

ref = fp;
ref.takeOff();
ref.fly();
ref.land();
```

Repeated Code

```
        }
}
```

1. CODE REDUCTION ✓
2. CODE FLEXIBILITY ✓

1.Loops ✗
2.Methods ✓

POLYMORPHIC WITH ADVANTAGES

```
class Airport
{
   public void permit( Plane ref)
   {
      ref.takeOff();
      ref.fly();
      ref.land();
   }
}

Airport a = new Airport();

a.permit(cp );
a.permit( pp );
a.permit( fp );
```

ref = cp

ref = pp

ref = fp