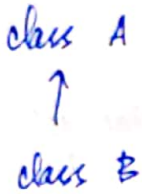
13.1.2020

Types of Inheritance :

To develop IS-A relationship b/w classes

Single :

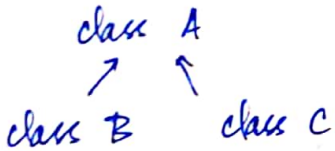


Advantages :

→ Reuse
→ Method Overriding
→ Everything can be inherited from parent to main

Multilevel

Hierarchical



`class subclassname extends parentclassname`

↓

replaced by implements while using interface.

Single Inheritance :

```
class animal
{
    void eat()
    {
        S.O.P("eating...");
    }
}
class dog extends animal
{
    void bark()
    {
        S.O.P("barking...");
    }
}
```

```
class test
{
  public static void main (String a[])
  {
      dog d = new dog();
      d.bark();
  }   d.eat();
}
```

```
class employee
{
  float salary = 9000f;
}
class pgmer extends employee
{
  int bonus = 10000;
}class test {
public static void main (String a[])
{
  pgmer p = new pgmer();
  S.O.P(p.bonus + " " + p.salary");
}}
```

Multilevel Inheritance :

```
class animal
{
   void eat()
   { }
}
class dog extends animal
{
}
class babydog extends dog
{
}
```

// Here creating object for class babydog will be enough to access all others.

super keyword :

→ Can be used with . variable
. method
. constructor

→ If we have same name for parent and child, (or) methods of parent and child, super is used so as to differentiate in such cases.

→ this() and super() cannot be accessed simultaneously.

super() refers immediate parent class

With variable :

```
class animal
{
    String color = "white";
}
class dog extends animal
{
    String color = "black";
    void printcolor()
    {
        S.O.P(color);          // o/p : black
        S.O.P(super.color);    // o/p : white
    }

public static void main(String a[])
{
    dog d = new dog();
    d.printcolor();
}
}
```

To use method with super keyword :

```
class animal
{
    void eat()
    {
        S.O.P("eating");
    }
}
```

```
class dog extends animal
{
    void eat()
    { S.O.P("Eating bread");
    }
    void bark()
    { S.O.P("barking");
    }
    void work()
    {
        super.eat();    // Parent class
        eat();          // Current class
        bark();
    }
}
```

super constructors :

```
class animal
{
    animal()
    { S.O.P("animal...");
    }
}
class dog extends animal
{
    dog()
    { super();          // Refers immediate parent class's constructor
      S.O.P("dog...");
    }
}
```

Even if super() isn't given, it will be automatically called during object creation (ie) implicitly.

super constructors with parameters :

```
class person
{
    int id;
    String name;
    person(int id, String name)
    {
        this.id = id;
        this.name = name;
    }
}
```

```
class emp extends person
{
    float salary;
    emp(int id, String name, float salary)
    {
        super(id, name);
        this.salary = salary;
    }

void display()
{
    S.O.P(id+" "+name+" "+salary);
}
}
class test
{
    public static void main(String a[])
    {
        emp e = new emp(1, "a", 500);
        e.display();    // 1 a 500
    }
}
```

21.1.2020

final keyword :

→ Used with
- variable - stop value change
- method - stop method overriding
- class - stop inheritance

→ Variable declared as final with no value initialized in it - blank final variable (or) uninitialized final variable
- can be initialized in constructor. If it is static, it can be initialized in static block.

→ ⊛ Once variable is initialized with final, it can't be changed. Cannot change value (constant)

final variable :

```
class bike
{
    final int speed = 90 ;
    void run()
    {
        speed = 400 ;    // Error. Value can't be changed.
    }

public static void main(String a[])
{
    bike b1 = new bike() ;
    b1.run() ;
}
}
```

final method :

```
class bike
{
    final void run()
    {
        S.O.P("running");
    }
}
class tvs extends bike
{
    void run()
    {
        S.O.P("running safely");
    }
    public static void main(String a[])
    {
        tvs t1 = new tvs() ;
        t1.run() ;          // Run-time error
    }
}
```

final class :

```
final class bike
{

}
class tvs extends bike
{
```

```
    void run()
    {
        S.O.P("running");
    }
    public static void main(String a[])
    {
        trs tr = new trs();
        tr.run();                    // Error, can't be inherited.
    }
}
```

blank final variable:

Constructor:

```
class bike
{
    final int speed;
    bike()
    {
        speed = 70;
        S.O.P(speed);
    }
    public static void main(String a[])
    {
        bike bk = new bike();   // Op: 70
    }
}
```

static block:

```
class a
{
    static final int data;
    static
    { data = 50;                    // This can be done.
    }
    public static void main(String a[])
    {
        S.O.P(a.data);              // Op: 50
    }
}
```

☹ final can't be given with constructor.

Arrays :

→ java.util

toString(arr) → Converted into String array

Sort(int[] a, int fromIndex, int toIndex)

binarySearch(int[] a, int key) → return -1 if key not found

Returns integer array ←
- int[] CopyOf(int[] original, int newLength) → If extra length, duplicate
- int[] CopyOfRange(int[] original, int from, int to)

fills(int[] a, int fromIndex, int toIndex, int val) → Mandatory

equals(arr1, arr2) → Same : true, else : false } Returns boolean character

Collections :

→ Predefined datastructure
→ Used to store group of related objects
→ ArrayList : inbuilt class
→ Package : java.util
→ Dynamically changes its size to accomodate more elements

ArrayList <T>

↓

Placeholder → Replace it with type of element

Methods available in ArrayList

add → Added from end
clear → To remove all elements
contains → like search
get → To retrieve
indexOf[] → 1st stored element's index
remove
size
trimtoSize
removeAll

addAll → To merge all elements into other (Not static)

Iterator ←
- hasNext() → Checks if next element is available (or) not
- next() → To find next element

(Eg:)

```
import java.util.*;
class testcollection
{
   public static void main (String a[])
   {
      ArrayList <String> a1 = new ArrayList<String> ();
      a1.add ("abc");
      a1.add ("pqr");
      a1.add ("xyz");
      ArrayList <String> a2 = new ArrayList <String> ();
      a2.add ("IT");
      a2.add ("MIT");
      a1.addAll (a2);
      Iterator itr = a1.iterator();
      while (itr.hasNext())
      {
         S.O.P(itr.next());
      }
   }
}
```

instanceof :

→ Compares the instance with type
→ Returns T/F

```
class sample
{
   public static void main (String a[])
   {
      sample s = new sample();
      S.O.P (s.instanceof sample);
   }
}
```

↓

Returns T/F

Checks whether object is belonging to class (or) not

```
class animal
{ }
class dog extends animal
{
    public static void main (String a[])
    {
        dog d = new dog();
        S.O.P (d. instanceof animal);  // Returns false
    }
}
```

2.1.2020

Polymorphism :

many forms

Two types :

⇒ Compile time (Static polymorphism)
- → Function Overloading

⇒ Run time
- → Dynamic method dispatch
- → Call to an overridden method (or) resolved at runtime method

Upcasting and Downcasting :

Casting : Taking an object of one type and assigning it to reference variable of another type.

Upcasting : Reference variable of parent class refers to object of child class.

class reference variable → instantiated as object of child class

Downcasting : Not as simple as upcasting. Hence, not used commonly.

```
class parent
{
    int x = 10;
    void show()
    {
        S.O.P("Parent");
    }
    void onlyparent()
    {
        S.O.P("Only Parent class");
    }
}
class child extends parent
{
    int x = 20;
    void show()
    {
        S.O.P("child");
    }
    void onlychild()
    {
        S.O.P("Only child class");
    }
}
class p
{
    public static void main(String a[])
    {
        parent p = new child();    // Upcasting
        p.show();  ------>  // This can be done/accessible
                            only if that method is
                            overwritten
        p.onlyparent();
        S.O.P(p.x);
    }
}
                            // p.onlychild is not accessible
                            as it is present in
                            child class.
```

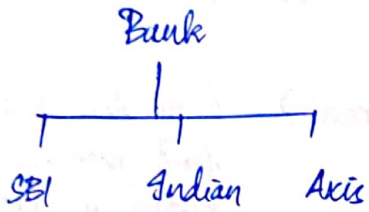
Values can't be initialized / changed at runtime.
it is assigned as soon as memory is allocated (compiling).
Hence p.x = 10 from parent class.

Downcasting : child c = new parent(); // This can't be done. Results in compile-time error.

Instead, we should write as

```
parent p = new child();
child c = (child) p;
```

(Eg.)




```
class bank
{
   float getrateofinterest()
   {
      return 0;
   }
}

class SBI extends bank
{
   float getrateofinterest()
   {
      return 8.4f;
   }
}
class Indian extends bank
{
   float getrateofinterest()
   {
      return 7.3f;
   }
}
class Axis extends bank
{
   float getrateofinterest()
   {
      return 9.7f;
   }
}
```

```
class testpoly
{
    public static void main(String a[])
    {
        Bank b;
        b = new SBI();        // Upcasting
        S.O.P("SBI" + b.getrateofinterest);
        b = new Indian();
        S.O.P("Indian" + b.getrateofinterest);
        b = new Axis();
        S.O.P("Axis" + b.getrateofinterest); // If Axis class doesn't have method as given method in Parent class will be invoked.
    }
}
```

Static binding: Early binding
Compile-time

Dynamic binding: Late binding → Only at run-time we will be deciding which should be used
Run-time

Try Multilevel →

Student
↑
UG
↑
PG