. A *class* is the blueprint from which individual objects are created.

**Encapsulation**

*Encapsulation* is the mechanism that binds together code and the data it manipulates, and

keeps both safe from outside interference and misuse

**Inheritance**

*Inheritance* is the process by which one object acquires the properties of another object.

**Polymorphism**

*Polymorphism* (from Greek, meaning “many forms”) is a feature that allows one interface to

be used for a general class of actions

a class is

a *template* for an object, and an object is an *instance* of a class

Variables defined within a class are called instance variables because each instance of the

class (that is, each object of the class) contains its own copy of these variables.

obtaining objects of a class is a two-step process.

First, you must declare a variable of the class type. This variable does not define an object.

Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual,

physical copy of the object and assign it to that variable. You can do this using the **new** operator.

The **new** operator dynamically allocates (that is, allocates at run time) memory for an object

and returns a reference to it. This reference is, more or less, the address in memory of the object

allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects

must be dynamically allocated.

*When you assign one object reference variable to another object reference variable,*

*you are not creating a copy of the object, you are only making a copy of the reference.*

Box b1 = new Box();

Box b2 = b1;

// ...

b1 = null;

Here, **b1** has been set to **null**, but **b2** still points to the original object

*array-var* = new *type*[*size*];

A*constructor* initializes an object immediately upon creation. It has the same name as the

class in which it resides and is syntactically similar to a method

**this** is always a reference to the object on which the method was invoked.

It works like this: when no

references to an object exist, that object is assumed to be no longer needed, and the memory

occupied by the object can be reclaimed

By using finalization, you can

define specific actions that will occur when an object is just about to be reclaimed by the

garbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method

Right before an asset is freed,

the Java run time calls the **finalize( )** method on the object.

important to understand that **finalize( )** is only called just prior to garbage collection.

It is not called when an object goes out-of-scope, for example. This means that you cannot

know when—or even if—**finalize( )** will be executed. Therefore, your program should provide

other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )**

for normal program operation.

**Overloading Methods**

In Java it is possible to define two or more methods within the same class that share the

same name, as long as their parameter declarations are different. When this is the case, the

methods are said to be *overloaded,* and the process is referred to as *method overloading..*

// Objects are passed by reference.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// pass an object

void meth(Test o) {

o.a \*= 2; o.b /= 2;

}

}

class CallByRef {

public static void main(String args[]) {

Test ob = new Test(15, 20);

System.out.println("ob.a and ob.b before call: " +

ob.a + " " + ob.b);

ob.meth(ob);

System.out.println("ob.a and ob.b after call: " +

ob.a + " " + ob.b);

}

}

Inner class:

**class** outer

{

**int** num1;

**public** **int** getNum1() {

**return** num1;

}

**public** **void** setNum1(**int** num1) {

**this**.num1 = num1;

}

**class** inner

{

**int** i;

}

}

**public** **class** innerclass

{

**public** **static** **void** main(String args[])

{

outer obj=**new** outer();

outer.inner obj1=obj.**new** inner();//to decalre inner calss object;

obj1.i=10;

}

}

Arrays:

In java arrays are objects

**int** a[]=**new** **int**[4]; //size is important

jagged array:

**int** d[][]={

{1,2,3,4},

{5,6,7},

{8,9,10,11}

};

**for**(**int** i=0;i<d.length;i++)

{

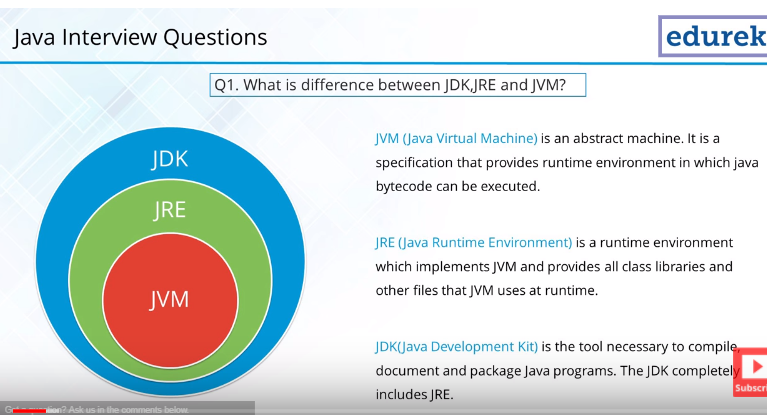
**for**(**int** j=0;j<d[i].length;j++)

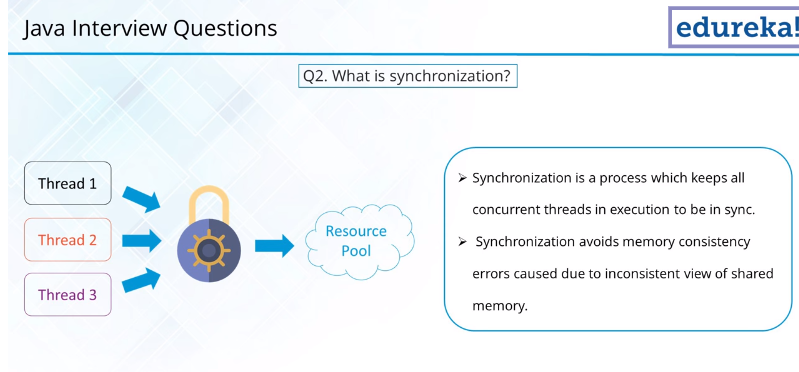
{

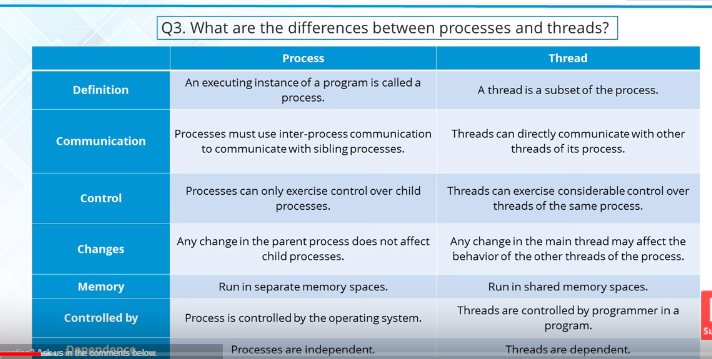
System.***out***.println(d[i][j]);

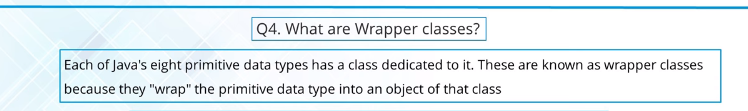
}

}









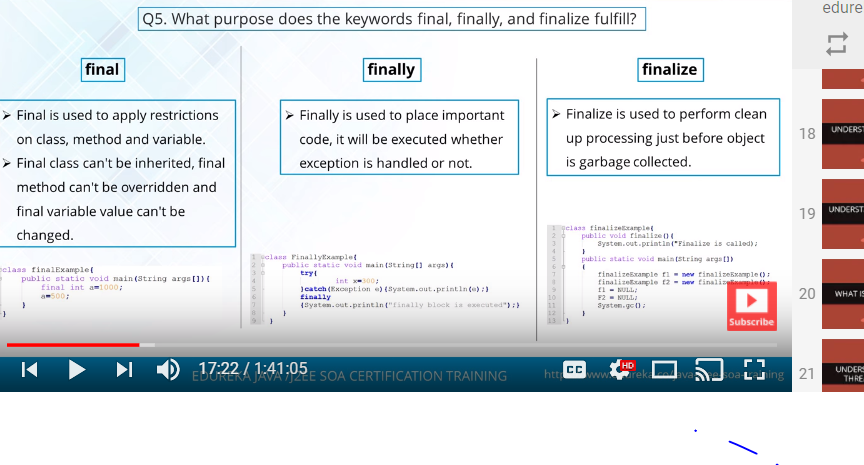
**int** i=10; //Single value container

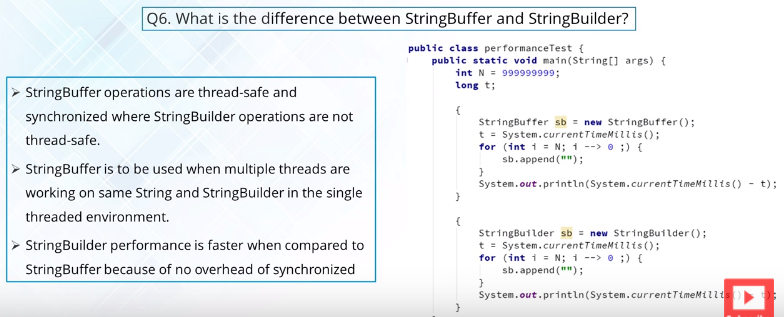
Integer j=**new** Integer(i); //boxed(constructing the object)

**int** k=j.intValue(); //unbox Extracting trh value from object

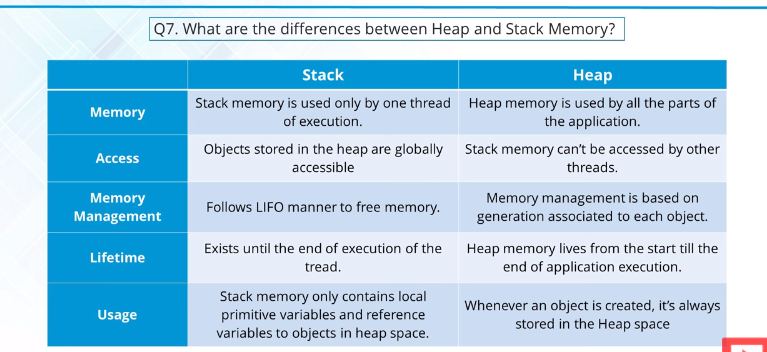
Integer y=k; //autoboxing

**int** z=y; //autounboxing

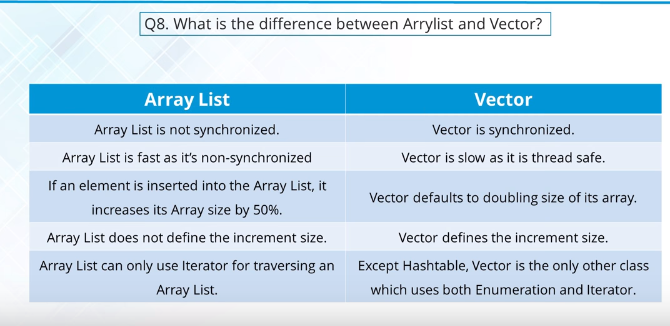


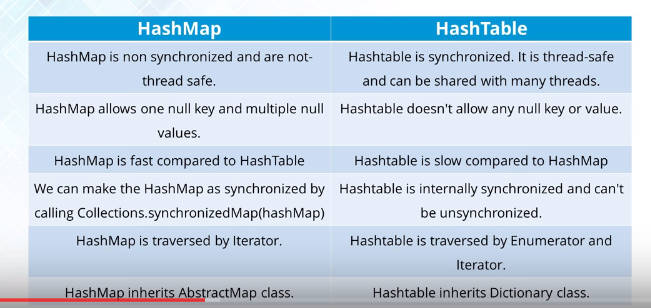


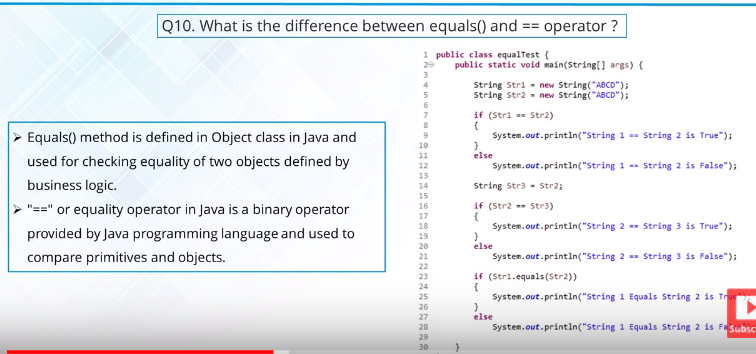
String is a immutable final class and cannot be changed



The object creation at runtime is done in heap at runtime







String str1=**new** String("Hello");

String str2=**new** String("Hello");

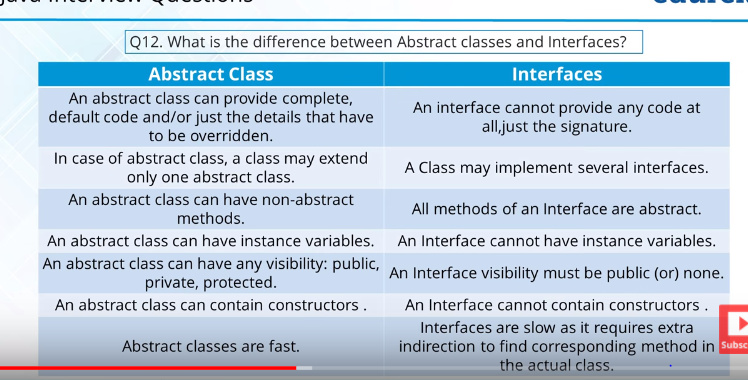
**if**(str1==str2) //prints not equal as only references are compared

System.***out***.println("equal");

**else**

System.***out***.println("not equal");

Equlas is used for comparing data in the objects



Sub to super is upcasting is allowed in java

**public** **class** Shape {

**public** **static** **void** main(String args[])

{

shape s=**new** circle();

s.draw(); //prits circle

}

}

**class** shape

{

**public** **void** draw()

{

System.***out***.println("shape");

}

}

**class** circle **extends** shape

{

**public** **void** draw()

{

System.***out***.println("shape");

}

}

Abstrac class:

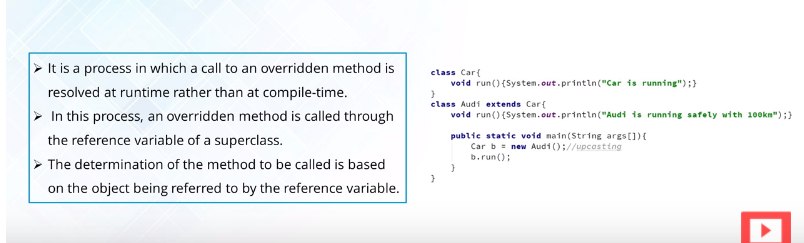
**abstract** **class** shape

{

**abstract** **public** **void** draw();

}

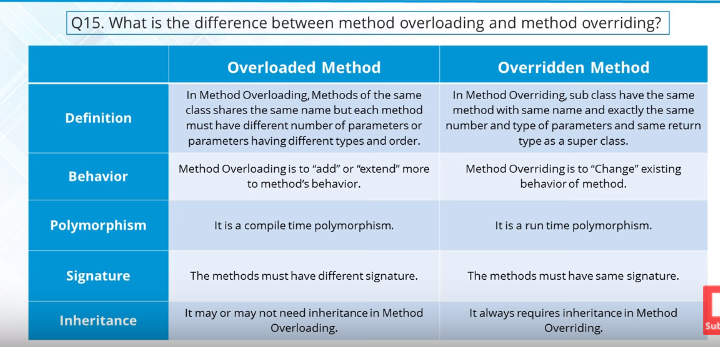
Runtime environment can create object of shape

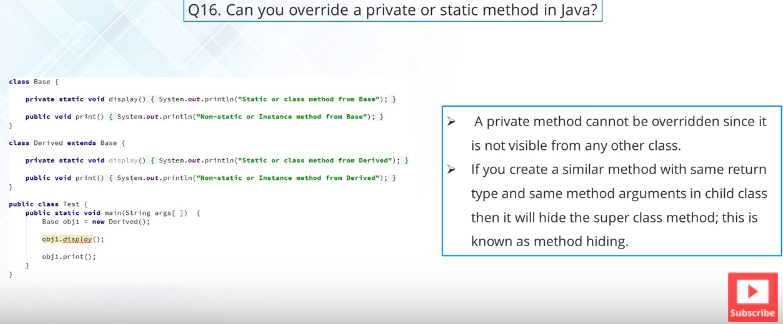


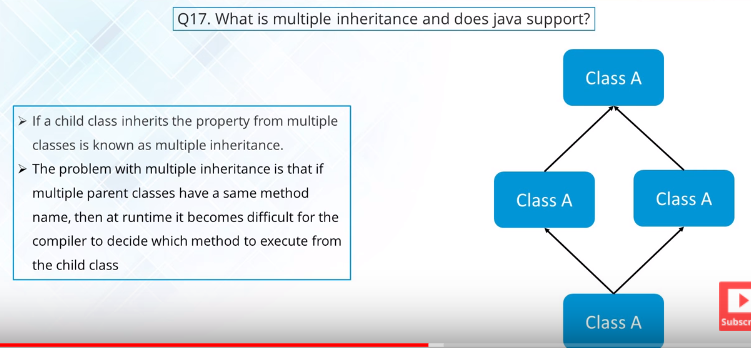
hashCode() is used for bucketing in Hash implementations like HashMap, HashTable, HashSet, etc.

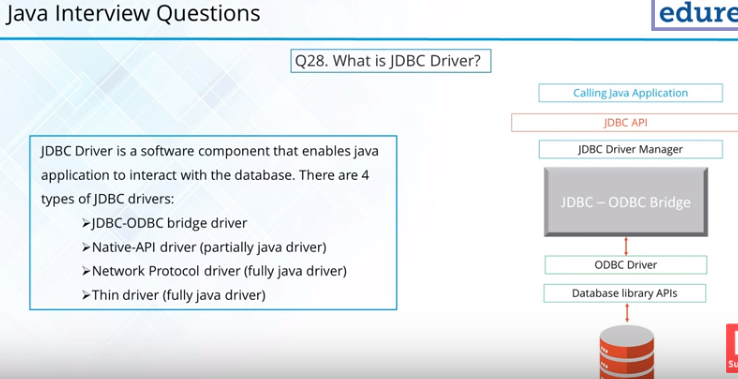
The value received from hashCode() is used as the bucket number for storing elements of the set/map. This bucket number is the address of the element inside the set/map.

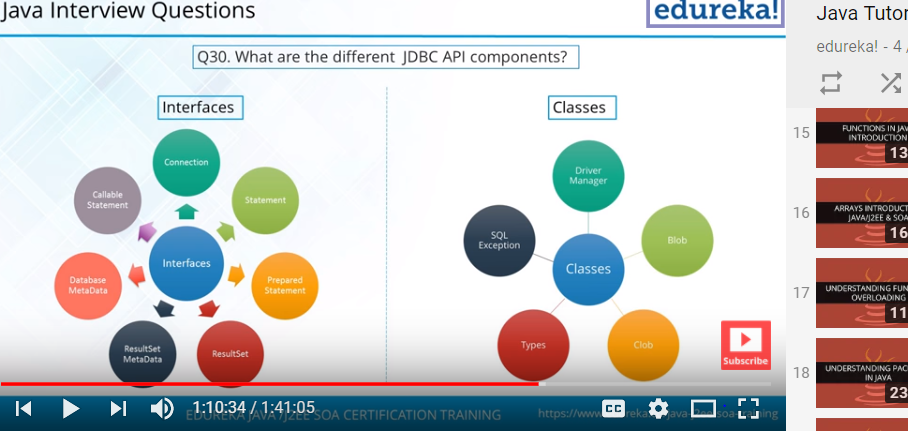
When you do contains() it will take the hash code of the element, then look for the bucket where hash code points to. If more than 1 element is found in the same bucket (multiple objects can have the same hash code), then it uses the equals() method to evaluate if the objects are equal, and then decide if contains() is true or false, or decide if element could be added in the set or not.

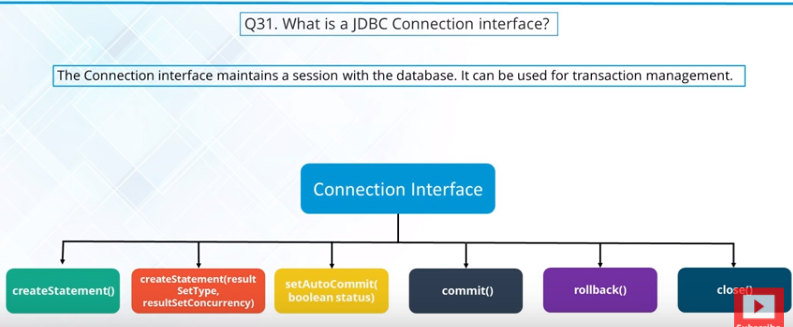




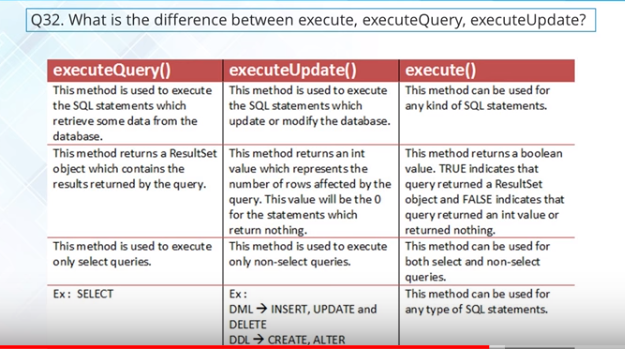


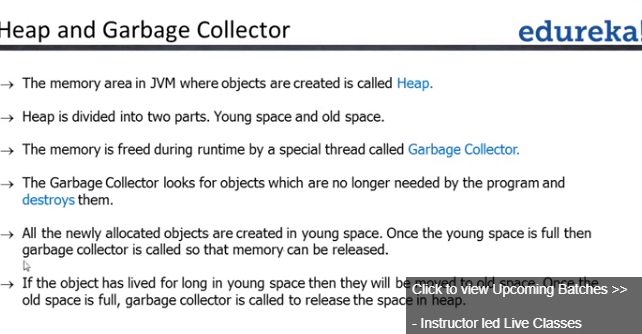


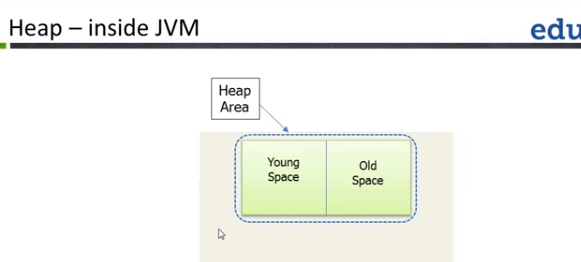




‘p







Young space-newly allocated objects are placed

Static:

* Whenever we load a object in java we use loader memory
* If we need variable to be shared,not object specific then we can make it as static.
* We can use class name to access the static variable
* Static block is used for initializing static variables and is execute only once and when class is loaded
* We can access only static variables in static methods

Inheritance:

**package** practise;

**class** calcu //super,parent,base

{

**public** **int** add(**int** i,**int** j)

{

**return** i+j;

}

}

**class** calculator **extends** calcu //sub.child,Derived

{

**public** **int** sub(**int** i,**int** j)

{

**return** i-j;

}

}

**public** **class** inheritance {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

calculator c1=**new** calculator();

System.***out***.println(c1.sub(2, 3));

System.***out***.println(c1.add(2, 3));

}

}

Extends –is a

Class has object of another class-has a

Create object of subclass,both super and subclass constructor is called

**package** practise;

**class** A

{

**public** A() {

System.***out***.println("in A");

// **TODO** Auto-generated constructor stub

}

**public** A(**int** i) {

System.***out***.println("in B"+i);

// **TODO** Auto-generated constructor stub

}

}

**class** B **extends** A

{

**public** B() {

System.***out***.println("in B");

// **TODO** Auto-generated constructor stub

}

**public** B(**int** i) {

System.***out***.println("in B"+i);

// **TODO** Auto-generated constructor stub

}

}

**public** **class** superdemo {

**public** **static** **void** main(String args[])

{

// A obj=new A(); // output is in A

// B obj1 =new B();

//output is in A

//in B

B obj2=**new** B(5);

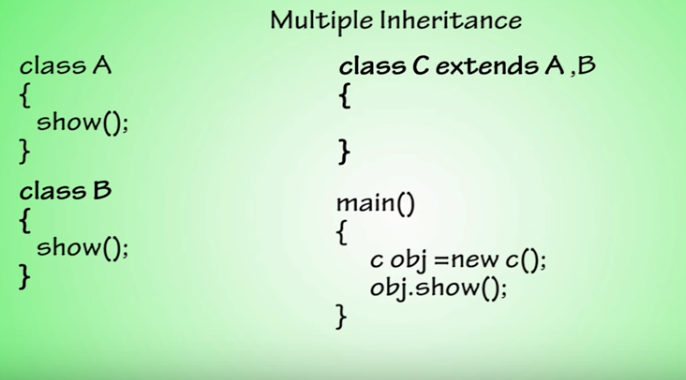
//in A

//in B5

}

}

Multiple inheritance issue



Methodoverriding:

Sub class reference contain sub class obj

**class** Abc

{

**public** **void** show()

{

System.***out***.println("in A");

}

}

**class** Bbcd **extends** Abc

{

**public** **void** show()

{

System.***out***.println("in B");

}

}

**public** **class** methodoveriding {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Bbcd obj=**new** Bbcd();

obj.show();

}

}

o/p:in B

sub class reference contain super class obj cause error

Bbcd obj=new Abc();

To overcome it apply casting:

**public** **class** methodoveriding {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Bbcd obj=(Bbcd)**new** Abc();

obj.show();

}

}

o/p: Exception in thread "main" java.lang.ClassCastException: practise.Abc cannot be cast to practise.Bbcd

at practise.methodoveriding.main(methodoveriding.java:25)

Super class reference contain super class obj

**class** Abc

{

**public** **void** show()

{

System.***out***.println("in A");

}

}

**class** Bbcd **extends** Abc

{

**public** **void** show()

{

System.***out***.println("in B");

}

}

**public** **class** methodoveriding {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Abc obj=**new** Abc();

obj.show();

}

}

o/p:in A

Super class refernce contains sub class obj:

**package** practise;

**class** Abc

{

**public** **void** show()

{

System.***out***.println("in A");

}

}

**class** Bbcd **extends** Abc

{

**public** **void** show()

{

System.***out***.println("in B");

}

}

**public** **class** methodoveriding {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Abc obj=**new** Bbcd();

obj.show();

}

}

o/p:in B

working fine combination of method overriding

* Super:static,Sub: static
* Super:static,Sub:non static (error)
* Super:non static,Sub:static (error)

Super class refernce contains sub class obj for static method overriding,only parent is executed;

**package** practise;

**class** Abc

{

**public** **static** **void** show()

{

System.***out***.println("in A");

}

}

**class** Bbcd **extends** Abc

{

**public** **static** **void** show()

{

System.***out***.println("in B");

}

}

**public** **class** methodoveriding {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Abc obj=**new** Bbcd();

obj.*show*();

}

}

* Static method overriding is done based on reference
* Binding data with methods is encapsulation
* Parse int takes int and returns integer value

Interface with inheritance

Interface extends in parent class

**package** thread;

**interface** i

{

**void** show();

}

**class** A **implements** i

{

**public** **void** show()

{

System.***out***.println("in A");

}

}

**class** B **extends** A

{

**public** **void** show()

{

System.***out***.println("in B");

}

}

**public** **class** inheritance {

**public** **static** **void** main(String args[])

{

i obj=**new** B();

obj.show(); //op:in B

i obj1=**new** A();

obj1.show(); //op:in A

}

}

Interface extends in child class:

**package** thread;

**interface** i

{

**void** show();

}

**class** A

{

**public** **void** show()

{

System.***out***.println("in A");

}

}

**class** B **extends** A **implements** i

{

**public** **void** show()

{

System.***out***.println("in B");

}

}

**public** **class** inheritance {

**public** **static** **void** main(String args[])

{

i obj=**new** B();

obj.show(); //in B

i obj1=**new** A(); //error

obj1.show();

}

}