**OOP Features**

* **Data hiding**
* **Abstraction**
* **Encapsulation**
* **Method signature**
* **IS-A relationship**
* **HAS-A relationship**
* **Overloading**
* **Overriding**
* **Polymorphism**
* **Constructors**
* **Interface**

**Data hiding**

Data means variables. Data hiding means hiding data from other classes.

We can achieve this using private modifier for variables.

Ex:

**package** pack1;

**class** B

{

**Privateint**x=1111;

}

**Publicclass** A extends B

{

**Publicstaticvoid**main(String[] args)

{

B b= **new** B();

System.***out***.println(b.x);//CE

}

}

Abstraction:

Hiding implementation details while exposing service names.

We can achieve this using interfaces.

The advantage is that we will get security.

Ex:

**interface** Bank

{

Abstract **int** withdraw(**int** accno);

}

Encapsulation:

it is the process of binding variables and methods together into a single unit.

We can achieve this by using class concept.

Ex:

Class A

{

int x=10;

void m1()

{

}

}

Method signature:

Method signature means method name followed by parameter types.

Method signature doesn’t include AMs, NAMs, return type, throws clause.

Ex1:

m1(int i)

----> m1(int)

{

}

Ex2:

m2(float f)----> m2(float)

{

}

Ex3:

m1(int i, float f)---->m1(int, float)

{

}

Ex4:

m1()---> m1()

{

}

Ex5:

m1(String s, float f)---->m1(String,float)

{

}

Note:

A class cannot contain 2methods with same signature, otherwise it’s a CE.

Ex:

**package** pack1;

**publicclass** A

{

**Publicvoid**m1(**int**i)

{

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

}

**Privateint**m1(**int**j)

{

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

}

**publicstaticvoid**main(String[] args)

{

A a=**new** A();

a.m1(10);//CE

}

}

Overloading:

Two methods of a class are said to be overloaded if they have same name but different parameters.

Ex1:

**publicclass** A

{

**Publicvoid** m1(**int**i)

{

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

}

**Privateint** m1(**float**i)

{

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

**return** 10;

}

}

Ex2:

**publicclass** A

{

**publicvoid** m1(**int**i,**float**f)

{

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

}

**privateint** m1(**float**f,**int**i)

{

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

**return** 10;

}

}

In method overloading, method resolution is taken care by the compiler based on the reference type that is used to invoke the method and the reference type or primitive type that is passed as argument.

Method resolution: which method has to be executed for what method call.(yeh method call ki, yeh method execute kavali ani cheppadanne method resolution antamu).

Ex:

**package** pack1;

**publicclass** Test

{

**publicvoid** m1(**int**i)

{

System.***out***.println("int arg method");

}

**int** m1(**float**f)

{

System.***out***.println("float arg method");

**return** 10;

}

**publicvoid** m1(String s)

{

System.***out***.println("string arg method");

}

**publicstaticvoid**main(String[] args)

{

Test t=**new** Test();

t.m1(10);

t.m1(10.5f);

t.m1(new String(“hyd”));//passing object directly OR

String s=new String(“hyd”);

t.m1(s);

t.m1('a');//int-arg method

}

}

Case1:automatic promotion in method overloading

In method overloading if exact method match is not found then compiler doesn’t rise any CE. Compiler tries to promote the method argument to the next level and checks if any method match is available or not, if available then it will executed otherwise compiler promotes the method argument again to the next level. In this way compiler checks all the possible promotions, finally if no method is matched then compiler rises error.

The possible promotions are:

1 2 4 8 4 8

byte->short->int->long->float->double

char

2

ex1:

**package** pack1;

**publicclass** Test

{

**int** m1(**float**f)

{

System.***out***.println("float arg method");

**return** 10;

}

**publicvoid** m1(String s)

{

System.***out***.println("string arg method");

}

**publicstaticvoid**main(String[] args)

{

Test t=**new** Test();

**byte**b=10;

t.m1(b);

t.m1(new String(“hyd”));

}

}

Ex2:

**package** pack1;

**publicclass** Test

{

**int** m1(Object o)

{

System.***out***.println("object arg method");

**return** 10;

}

**publicvoid** m1(Number n)

{

System.***out***.println("Number arg method");

}

**publicvoid** m1(Integer I)

{

System.***out***.println("Integer arg method");

}

**publicstaticvoid**main(String[] args)

{

Test t=**new** Test();

Integer I=**new**Integer(10);

t.m1(I);//Integer-arg method

/\*if m1(Integer) method is not available then m1(Number) method will be executed.

If m1(Number) method is not available then m1(Object) method will be executed \*/

t.m1(new Integer(10));//Integer-arg method

}

}

Ex3:

**package** pack1;

**class** Animal

{

}

**class** Tiger **extends** Animal

{

Parameter name

Parameter type

}

**publicclass** Test

{

**int** m1(Animal a)//method definition

{

System.***out***.println("Animal version method");

**return** 10;

}

**publicvoid** m1(Tiger t)//method definition

{

System.***out***.println("Tiger version method");

}

**publicstaticvoid**main(String[] args)

{

Test t=**new** Test();

Animal a=**new**Animal();

t.m1(a);//method call - Animal version method

Tiger tr=**new**Tiger();

t.m1(tr);//method call - Tiger version method

Animal a1=**new**Tiger();//confusion starts here.

/\*type of a1 is Animal but a1 is referring Tiger object. \*/

t.m1(a1);//method call - Animal version method

}

}

Method Overriding :

Parent class variables and methods are by default available to child class.

If the child doesn’t like parent class inherited methods then child class can redefine the functionality, this concept is called as overriding.

Ex:

**package** pack1;

**class** Parent1

{

**int**workhard()

{

System.***out***.println("wakeup early,gotocollege");

**return** 10;

}

**float**care()

{

System.***out***.println("utmost care");

**return** 10.5f;

}

}

**class** Child1 **extends** Parent1

{

**int**workhard()

{

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

}

**void**love()

{

System.***out***.println("im in love");

}

}

**publicclass** Test3 {

**publicstaticvoid**main(String[] args) {

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

Parent1 p1=**new** Parent1();

p1.workhard();//college method

//if both reference type and object type

//are same then same class method

//will be executed.

Child1 c1=**new** Child1();

c1.workhard();//bar method

//if both reference type and object type

//are same then same class method

//will be executed.

Parent1 p2=**new** Child1();

//parent class reference variable can refer child class object

p2.workhard();//bar method

//p2.love();//CE

p2.care();//valid

}

}

Level of access modifiers:

public>protected>default>private

Overriding rules:

i>about access modifiers

while overriding, the scope of child class method access modifier should be same or higher than parent class access modifier.

public>protected>default>private.

Ex:

Parent method child method

Public public

Protected protected,public(but it cannot be default or private)

Default default,protected,public

Private we cannot override private methods

ii>about non-access modifiers

-only parent class instance methods can be overridden.

We cannot override parent class static methods.

-we cannot override parent class final methods.

-abstract methods of parent class must be overridden in the child class.

* strictfp non-strictfp
* Native non-native
* Synchronized non-synchronized

iii>

a)if parent class methods return type is a primitive type then the child class overriding method’s return type should be same.

b)if parent class methods return type is a class type then child class overriding method’s return type can be same class or sub-class, but it cannot be super class.

Ex1:valid

**class** Parent1 {

Number workhard() {

System.***out***.println("wakeup early,gotocollege");

**return** 10;

}

}

**class** Child1 **extends** Parent1 {

Numberworkhard() {

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

}}

Ex2:valid

**class** Parent1 {

Number workhard() {

System.***out***.println("wakeup early,gotocollege");

**return** 10;

}

}

**class** Child1 **extends** Parent1 {

Integerworkhard(){

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

} }

Integer class is child class of Number class.

Number class is child class of Object class.

Ex3:invalid

**class** Parent1 {

Number workhard() {

System.***out***.println("wakeup early,gotocollege");

**return** 10;

}

}

**class** Child1 **extends** Parent1 {

Object workhard() {

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

} }

iv> about exceptions

if parent class method throws a checked exception, then the child class overriding method can

a>throw same checked exception OR

b> child checked exception OR

c>nothing(child class overriding method need not throw any checked exception).

d>child class overriding method cannot throw parent checked exception.

Ex1:

**class** Parent1 {

Number workhard() **throws**IOException

{

System.***out***.println("wakeup early,gotocollege");

**return** 10;

}}

**class** Child1 **extends** Parent1{

Number workhard() **throws**IOException

{

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

}}

Ex2:

**class** Parent1 {

Number workhard() **throws**IOException

{

System.***out***.println("wakeup early,gotocollege");

**return** 10;

} }

**class** Child1 **extends** Parent1 {

Number workhard() **throws**FileNotFoundException

{

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

} }

FileNotFoundException is child checked exception to the IOException.

Ex3:nothing

**class** Parent1 {

Number workhard() **throws**IOException {

System.***out***.println("wakeup early,gotocollege");

**return** 10;

} }

**class** Child1 **extends** Parent1 {

Number workhard() {

System.***out***.println("wakeup anytime,goto bar");

**return** 20;

} }

Polymorphism:

Polymorphism is the ability of an object or a method to take multiple forms.

a>about an object

i>parent class reference variable can refer/hold child class objects.

ii>interface reference variable can refer implementation class object.

Ex:

Parent p1=new Child1();

;

;

;

p1=new Child2();

;

;

;

P1=new Child3();

b) about method

overloading is static polymorphism and overriding is dynamic polymorphism.

Static polymorphism: It is the process in which a call to an overloaded method is resolved at compile time not at runtime.

In this process an overloaded method is called using the reference variable of a class and the same class method is executed.

Test t=new Test();

t.m1(10);

t.m1(tr);

Dynamic polymorphism: It is the process in which a call to an overridden method(it means it is parent method) is resolved at runtime not at compile time.

In this process an overridden method is called using parent reference variable but execution of the method is determined by the runtime object.

Parent p1=new Child();

p1.m1();//m1() method is overridden method of parent class.

In the above m1() method of parent class is called.But m1() method of child class is executed.

Constructors:

Ex1:not recommended, because every object will have same default values.

**package** pack1;

**publicclass** Student {

**int**rollno;

String sname;

**publicstaticvoid**main(String[] args) {

Student s1=**new**Student();

Student s2=**new**Student();

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

}

}

Output:

O

Null

O

Null

Ex2:not recommended to initialize instance variables at the time of declaration because every object will have same values.

**package** pack1;

**publicclass** Student {

**int**rollno=101;

String sname="uday";

**Publicstaticvoid**main(String[] args) {

Student s1=**new**Student();

Student s2=**new**Student();

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

}

}

Output

101

Uday

101

Uday

Ex3:not recommended to initialize instance variables inside instance block because every object will have same values.

For every object creation same instance block gets executed.

**package** pack1;

**publicclass** Student {

**int**rollno;

String sname;

{//instance block

rollno=101;

sname="uday";

}

**Publicstaticvoid**main(String[] args) {

Student s1=**new**Student();

Student s2=**new**Student();

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

}

}

Output:

101

Uday

101

Uday

Ex4:this works fine,but not recommended because it increases the length of the code.

**package** pack1;

**publicclass** Student {

**int**rollno;

String sname;

**Publicstaticvoid**main(String[] args) {

Student s1=**new**Student();

s1.rollno=101;

s1.sname="uday";

Student s2=**new**Student();

s2.rollno=102;

s2.sname="kumar";

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

}

}

Output:

101

Uday

102

Kumar

The above problems can be resolved using constructor.

A constructor is a piece of code that gets executed whenever we create an object.

Constructors are used to initialize object’s state.

Ex5:

**package** pack1;

**publicclass** Student {

**int**rollno;

String sname;

Student(**int**r,Stringn)

{

rollno=r;

sname=n;

}

**Publicstaticvoid**main(String[] args) {

Student s1=**new** Student(101,"uday");

Student s2=**new** Student(102,"kumar");

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

}

}

Output:

101

Uday

102

Kumar

Ex6:usage of this keyword.

**package** pack1;

**publicclass** Student {

**int**rollno;

String sname;

Student(**int**rollno,Stringsname)

{

**this**.rollno=rollno;

**this**.sname=sname;

}

**Publicstaticvoid**main(String[] args) {

Student s1=**new** Student(101,"uday");//line A

Student s2=**new** Student(102,"kumar");//line B

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

}

}

“This” refers current object.

When the control is at line A, this refers s1 object.

When the control is at line B, this refers s2 object.

Ex7:

For every object creation corresponding constructor must be there.

**package** pack1;

**publicclass** Student {

**int**rollno;

String sname;

Student(**int**rollno,Stringsname)

{

**this**.rollno=rollno;

**this**.sname=sname;

}

Student(**int**rollno)

{

**this**.rollno=rollno;

System.***out***.println(s1.rollno);

}

**Publicstaticvoid**main(String[] args) {

Student s1=**new** Student(101,"uday");

Student s2=**new** Student(102,"kumar");

Student s3=**new** Student(103);

System.***out***.println(s1.rollno);

System.***out***.println(s1.sname);

System.***out***.println(s2.rollno);

System.***out***.println(s2.sname);

System.***out***.println(s3.rollno);

}

}

Rules for writing constructors:

1.constructor name should be same as class name.

2.consturctors will not have return types.

3.a constructor can take all the 4 AMs (public,protected,default,private).

4.none of the NAMs are applicable for a constructor.

5.default constructor:

a>When we don’t write any constructor then compiler gives us a no-arg constructor called as default constructor.

If we write any constructor then compiler will not give default constructor.

b>the AM of default constructor is same as class AM, this rule is applicable only for public, default classes.

Ex:

Programmers code compiler generated code

class Test

class Test

{

Test()

{

super();

}

}

{

}

super() is a call to super class no-arg constructor.

Class Test

{

Test()

{

super();

}

}

class Test

{

Test()

{

}

}

Note:

The first line of every constructor should be super() or this().

When we don’t write anything then compiler places super().

super() is a call to super class no-arg constructor.

this() is a call to current class no-arg constructor.

Diff. between

super(), this()

1.these are constructor calls.

2.used to call super class, current class no-arg constructors resp.

3.we can use these 2 constructor calls only inside constructors.

super.,this.

1.these are keywords

2.used to refer super class, current class instance members.

3.we cannot use these keywords inside static block, static method.

Static control flow:

Ex:

**package** pack1;

**publicclass** Test {

*a*=10;

1.**staticint**

2.**static**

{//static block

System.***out***.println("FSB"); .8

*m1*(); .9

}

3.**staticvoid** m1()

{//static method

System.***out***.println(*a*); 10

System.***out***.println(*b*); 11

}

4.**publicstaticvoid**main(String[] args)

{

*m1*();

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

}

5.**staticint***b*=20;

6.**static**

{

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

*m1*();

}

}

1.identification of static members from top to bottom(Step 1 to 6).

a=0,b=0

Static members means: static variables, static methods, static blocks.

In this phase JVM gives default values to the static variables.

2.execution of static variable assignments and static blocks from top to bottom.

In this phase JVM gives original values to static variables.

3.execution of main method.

Output:

FSB

10

0

Interfaces:

Interface in a concrete class.And interface contains 100% abstract methods.

Or it is a piece of code that contains only abstract methods.

interface Interf1

{

void m1();

void m2();

int x=10;

}

Interface methods are always public and abstract whether we declare or not.

Interface variables are always public, static and final whether we declare or not.

The above interface is same as below one:

interface Interf1

{

Public abstract void m1();

Public abstract void m2();

Public static final int x=10;

}

Ex:

**package** pack1;

**interface** X

{

**void** m1();

**int*x***=10;

}

**publicclass**Interf**implements** X {

**publicvoid** m1()

{

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

}

**publicstaticvoid**main(String[] args) {

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

Interfi=**new**Interf();

i.m1();

System.***out***.println(i.***x***);

}

}