**Object Oriented Programming (OOPS)**

**Data Hiding** :-

Our internal data should not go out directly that is outside person can’t access our internal data directly.

By using private modifier we can implement data hinding.

Ex:

Class Account {

Private double balance;

…………………………………;

…………………………………;

}

After providing proper username and password only, we can acces our Account information.

The main advantage of data hiding is security.

Note: recommended modifier for data members is private.

**Abstraction**:

Hide internal implementation and just highlight the set of services , is called abstraction.

By using abstract and interfaces we can implement abstraction.

Ex:

By using abstract classes and interfaces we can implement abstraction.

Ex:

By using ATM GUI screen bank people are highlighting the set of services, What they are offering without highlighting internal implementation.

**The main advantages of abstraction are**:

We can achieve security as we are not highlighting our internal

Implementation.(i.e., outside person does not aware our internal implementation .)

Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.

It provides more flexibility to the end user to use system very easily.

It improves maintainability of the application.

It improves easiness to use our system.

By using interfaces (GUI screens)we can implement abstraction.

**Encapsulation**:

Binding of data and corresponding method s into a single unit is called Encapsulation.

If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.

Encapsulation= Datahiding +Abstraction

Example :

Class Account

{

Private double balance;

Public double getBalance()

{

//validate user

return balance;

}

Public void setBalance(double balance)

{

//validate user

this.balance=balance;

}

}

Welcome to vbr ban

k

getBalance

setBalance

GUI

Every data member should be declared as private and for every member we have to maintain getter & setter methods.

**The main advantages of encapsulations are:**

We can achieve security.

Enhancement will become very easy.

It improves maintainability and modularity of the application.

It provides flexibility to the user to use system very easily.

The main disadvantage of encapsulation is it increases length of the code and slows down execution.

**Tightly encapsulated class**:

A class is said to be tightly encapsulated if and only if every variable of that class declared as private whether the variable has getter and setter methods are not , and whether these methods declared as public or not these checking are not required to perform.

Ex:

class Account {

private double balance;

public double getBalance(){

return balance;

}

}

Which of the following classes are tightly encapsulated?

Class A

{

private int x=10; (valid)

}

class B extends A

{

Int y=20; (invalid)

}

class C extends A

{

private int z=30;(valid)

}

Which of the following classes are tightly encapsulated ?

Class A {

Int x=10; //not

}

Class B extends A {

private int y=20; //not

}

class C extends B {

private int z=30; //not

}

Note: if the parent class is not tightly encapsulated then no child class is tightly encapsulated.

**IS-A Relationship (inheritance)** :

Also Known as inheritance.

By using “extends” keywords we can implement IS-A relationship.

The main advantages of IS-A relation reusability.

Example :

Class parent {

public void methodOne () {}

}

Class Child extends parent {

Public void methoTwo (){}

}

class Test

{

public static void main(String[]args)

{

Parent p=new parent ();

p.methodOne();

p.methodTwo();

child c=new Child();

c.methodOne();

c.methodTwo();

parent p1 = new Child();

p1.methodOne();

p1.methodTwo();

Child c1 = new Parent();

}

}

**Conclusion**:

Whatever the parent has by default available to the child but whatever the child has by default not available to the parent . hence or the child reference we can call both parents and child class methods but on the parent reference we can call only methods available in the parent class and we can’t call child specific methods.

Parent class reference can be used to hold child class object but by using that references we can call only methods available in parent class and child specific methods we can’t call.

Child class reference cannot be used to hold parent class object.

Example:

The common methods which are required for housing loan, vehicle loan, person loan and education loan we can define into a separate class in parent class loan.so that automatically these methods are availability to every child loan class.

Ex:

Class Loan {

//common methods Which are required for any type of loan.

}

Class HousingLoan extends Loan {

//Housing loan specific methods .

}

Class EducationLoan extends Loan {

//Education Loan specific methods.

}

**For all java classes the most commonly required functionality is define inside object class hence object class acts as a root for all java classes.**

**For all java exception and errors the most common required functionality define inside throwable class hence throwable class acts as a root for exception hierarchy.**

Diagram:

Object

String …………………………………………………..Number Throwable

Exception Error

Byte short integer long float Double

**Multiple Inheritance** :

Having more than one parent class at the same level is called multiple inheritance.

Ex:

Parent1 Parent2

Child

Any class can extends only one class at a time and can’t extends more than one class simultaneously hence java won’t provide support for multiple inheritance.

Ex:

Class A {}

Class B {}

Class C extends A,B

{}

(invalid)

But an interface can extends any no. Of interfaces at a time hence java provides support for multiple inheritance through interfaces.

Example:

Interface A {}

Interface B {}

Interface C extends A,B {}

If our class doesn’t extends any other class then only our class is the direct child class of object.

Ex :

Class A {} Object

Ex 1 :

Class B {} Object

Class A extends B {}

Multilevel inheritance

(valid)

Ex 2:

Object B

A

(Invalid)

Why java won’t provide support for multiple inheritance?

There may be a chance of raising ambiguity problems.

Ex:

Parent methodOne() parent2 methodOne()

c.methodOne(); Ambiguity problem.

Why ambiguity problem won’t be there in interfaces ?

Interfaces having dummy declaration and they won’t have implementation hence no ambiguity problem.

Ex: interfaces having dummy declarations and they won’t have implementation hence no ambiguity problem.

EX:

Interface inter2{

Public void methodOne();

}

Interface inter1

{

Public void methodOne();

}

Interface inter3 extends inter1,inter{}

Class Test implements inter3

{

Public void methodOne(){

System.out.println(“This is methodOne()”);

}

}

**Cyclic inheritance** :

Cyclic inheritance is not allowed in java.

**Ex 1:**

Class A extends B{} (invalid)

Class B extends A{} C.E:cyclic inheritance involving A

EX2:

Class A extends A {} C.E cyclic inheritance involving A

**HAS-A relationship**:

HAS-A relationship is also known as composition (or)aggregation.

There is no specific keyword to implement HAS-A relationship but mostly we can use new operater.

The main advantage of HAS-A relationship is reusability.

Ex:

Class Engine

{

//engine specific functionality

}

Class Car

{

Engine e = new Engine();

//…………………………………..;

//…………………………………..;

//……………………………………;

}

Class Car HAS-A engine reference.

The main dis-advantage of HAS-A relationship increases dependency between the components and creates maintains problems**.**

**Composition vs Aggregation :**

Composition

Without existing container object if there is no chance of existing contained objects then the relationship between container object and contained object is called composition which is a strong association.

EX:

University consists of several departments Whenever university object destroys automatically all the departments objects will be destroyed that is without existing university object there is no chance of existing dependent object hence these are strongly associated and this relationship is called composition.

EX:

Department2

Deparment1

Contained Object

Department4

Department3

Container Object

University

Aggregation :

Without existing container object if there is a chance of existing contained objects such type of relationship is called aggregation. In aggregation object have weak association.

EX:

Within a department there may be a chance of several professors will work whnever we are closing departments still there may be a chance of existing professor object without existing department object the relationship between department and professor is callede aggregation where the object having weak association.

**EX:**

Contained object

Department Container object

**Note:**

In composition containe , contained objects are strongly associated, and but container object hold contained object directly.

But in aggregation container and contained objects are weakly associated and container object just now holds the reference of container objects.

**Method signatur :**

In java ,method signature consists of name of the method followed by argument types.

EX:

Public void methodOne(int I,float f);

methodOne(int,float);

In java return type is not part of the method signature.

Compiler will use method signature while resolving method calls.

Class Test {

Public void m1(double d) {}

Public void m2 (int i) {}

Public static void main (String arg[]) {

Test t = new Test ();

t.m1 (10.5);

t.m2 (10);

t.m3 (10.5); //CE

}

}

**CE:** cannot find symbol

Symbol: method m3 (double)

Location: class Test

Within the same class we can’t take 2 methods with the same signature otherwise we will get compile time error.

**Ex:** public void methodOne() {}

Public int methodOne () {

return 10;

}

Output :

Compile time error

methodOne () is already defined in Test

**Polymorphism :**

Same name with different forms is the concept of polymorphism.

**EX:** we can use same abs()method for int type, long type, float type etc .

EX:

1. Abs(int)
2. Abs(long)
3. Abs(float)

EX2:

we can use the parent reference to hold any child objects.

We can use the same list reference to hold ArrayList object, LinkedList object, vector object ,or Stack object.

EX:

List 1 = new ArrayList();

List 1 = new LinkedList();

List 1 = new Vector();

List 1 = new Stack();

**Diagram: Polymorphism**

Compiletime/static/earlybinding Runtime/dynamic/latebinding

Over loading methodhiding overriding

**Diagram:** pillar of OOPS

Reusability

flexibility security

1. Inheritance talks about reusability.

2. Polymorphism talks about flexibility.

3. Encapsulation talks about security.

**Beautiful definition of polymorphism:**

A boy starts love with the word friendship, but girl ends love with the same word friendship, word is the same but with different attitudes, this concept is nothing but polymorphism.

**Overloading:**

1. Two methods are said to be overload if and only if both having the same name but different argument types.
2. In ‘C’ language we can’t take 2 methods with the same name and different types.

If there is a change in argument type compulsory we should go for new method name.

EX:

abs() for int type

labs() for long type

fabs() for float type

.

.

Etc

1. Lack of overloading in “C” increases complexity of the programming.
2. But in java we can take multiple methods with the same name and different argument types.

EX:

abs(int)

abs(long)

abs(float)

.

.

.

1. Having the same name and different argument types is called method overloading.
2. All these methods are considered as overloaded methods.
3. Having overloading concept in java reduces complexity of the programming.

EX:

class Test {

public void methodOne(){

system.out.println(“no-arg method”);

}

public void methodOne(int){

systm.out.println(“int-arg method”); //overloaded methods

}

Public void methodOne (double d){

System.out.println(“double-arg method”);

}

public static void main(String[] args)

Test t = new Test();

t.methodOne(); //no-arg method

t.methodOne(10); //int-arg method

t.methodOne(10.5;) //double-argmethod

}

Conclusion: in overloading compiler is responsible to perform method resolution(decision)based on the reference type(but not based on run time object).hence overloading as compile time polymorphism(or)

Static polymorphism (or)early binding.

Case 1 : Automatic promotion in overloading.

In overloading if compiler is unable to find the method with exact match we won’t get any compile time error immediately.

1st compiler promotes the argument to the next level and checks whether the matched method is available or not if it is available then that method will be considered if it is not available then copiler promotes the argument once again to the next level. This process will be continued until all possible promotions still if the matched method is not available then we will get compile time error. This processes called automatic promotion in overloading.

The following are various possible automatic promotions in overloading.

**Diagram :**

byte short

int long float double

char

EX:

Class Test

{

Public void methodOne(int i){

System.out.println(“int-arg method”);

}

Public void methodOne(float f) //overloaded methods

{

System.out.println(“float-arg method”);

}

Public static void main(String[] args){

Test t= new Test ();

//t.methodOne (‘a’); //int-arg method

//t.methodOne(101); //float-arg method

t.methodOne(10.5); //C.E: cannot find symbol

}

}

Case 2:

Class Test

{

Public void methodOne(String s)

{

System.out.println(“String version”);

}

Public void methodOne(“Object o”); //Both methods are said to overload methods

{

System.out.println(“Object version”);

}

Public static void main (String[] args)

{

Test t = new Test();

t.methodOne(“arum”); //string version

t.methodOne(new Object()); //object version

t.methodOne(null); //string version

}

}

Note:

While resolving overloaded methods exact match will always get high priority, while resolving overloaded methods child class will get the more priority then parent class.

**Case3:** class Test{

Public void methodOne(String s){

System.out.println(“String version”)

Public void methodOne(StringBuffer s){

System.out.println(“StringBuffer version”);

}

Public static void main (String[] args){

Test t = new Test();

t.methodOne(“arun”); //String version

t.methodOne(new StringBuffer(sai“)); //StringBuffer version

t.methodOne(null); //CE: reference to m1() is ambiquous

}

}

**Output:**

Object

String StringBuffer

**Case 4:**

Class Test {

Public void methodOne(int i, float f){

System.out.println(“int-float method”);

}

Public void methodOne(float f, int i){

System.out.println(“float-int method”);

}

Public static void main(String[] args){

}

Test t = new Test ();

t.methodOne (10,10.5f);//int-float method

t.methodOne(10.5f,10);//float-int method

t.methodOne (10,10); //C.E:

//CE : reference to methodOne is ambiguous,

//both method methodOne(int, float) in Test

//and methodOne(float,int) in Test match

t.methodOne(10.5f,10.5f);//C.E:

cannot find symbol

symbol : methodOne(float, float)

location : class Test

}

}

**Case 5:**

Class Test{

Public void methodOne (int i){

System.out.println(“general method”);

}

Public void ethodOne(int…i){

System.out.println(“var-arg method”);

}

Public static void main(String[] args){

Test t = new Test();

t.methodOne (); //var-arg method

t.methodOne(10,20);// var-arg method

t.methodOne(10); //general method

}

}

In general var –arg method will get less priority that is if no other method matched then only var-arg method will get chance for execution it is almost same as default case inside switch.

**Case 6:**

Class Animal {

Class Monkey extends Animal{}

Class Test{

Public void methodOne(Animal a) {

System.out.println(“Animal version”);

}

Public void methodOne(monkey m)

System.out.println(“Monkey version”);

}

Public static void main(String[] args) {

Test t=new Test();

Animal a=new Test();

t.methodOne (a);//Animal version

Monkey m=new Monkey();

t.methodOne(m);//Monkey version

Animal a1=new Monkey():

t.methodOne(a1);//Animal version

}}

In Overloading method resolution is always based on reference type and runtime object won’t play any role in overloading.

# Overriding:

1.Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child isallow to redefine that Parent class method in Child class in its own way thisprocess is called overriding.

2. The Parent class method which is overridden is called overridden method.

3. The Child class method which is overriding is called overriding method.

Example 1:

class Parent

public void property () {

System.out. println ( "cash+land+gold");

}

public void marry()

System.out . println (" subbalakshmi"); //overridden method

}

}

class Child extends Parent{

public void marry ()

System.out.println (" 3sha/4me/9 tara/anushka");

//overriding method

class Test {

public static void main (String [ ] args) {

Parent p=new Parent ();

p.marry );//subbalak shmi (parent method)

Child c=new Child ();

c.marry )// Trisha/nayanatara/anushk a (child method)

Parent p1=new Child ();

p1.marry ) ;//Trisha/nayanatara/anushka (child method)

In overriding method resolution is always takes care by JVM based on runtime

Object hence overriding is also considered as runtime polymorphism or dynamic

Polymorphism or late binding.

30. The process of overriding method resolution is also known as dynamic method Dispatch.

Note: In overriding runtime object will play the role and reference type is dummy.

**Rules for cverriding :**

1. In overriding method names and arguments must be same. That is method

signature must be same.

2. Until 1.4 version the return types must be same but from 1.5 version onwards co-

variant return types are allowed.

3. According to this Child class method return iype need not be same as Paren

class method return type its Child types also allowed.

**Example:**

class Parent {

public Object methodOne (){

return null;

}

}

class Child extends Parent {

public String methodone ()

return null;

}

}

C:> javac -source 1.4 Parent.java //error

**It is valid in "1.5" but invalid in "1.4".**

**Diagram:**

Parent method: object Number String double

Return type

Child method: String Integer Object Int

Return type Right Right wrong wrong

Co-variant return type concept is applicable only for object types but not for primitives.

17. Private methods are not visible in the Child classes hence overriding concept isnot applicable for private methods. Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.

**Example:**

class Parent

{

private void methodOne()

it is valid but not overriding.

}

class Child extends Parent

{|private void methodOne){}

}

Parent class final methods we can't override in the Child class

**Example:**

class Parent{

public final void me thodOne (){}

}

class Child extends Parent {

public void methodone () {}

**Output:**

Compile time error:

methodOne () in Child cannot override methodone ()

in Parent; overridden method is final

**Parent class non final metho ds we can override as final in child class. We can**

**override native methods in the child classes.**

**We should override Parent class abstract methods in Child classes to provide**

**implementation.**

**Example:**

abstract class Parent (

public abstract void methodone ();

}

class Child extends Parent

public void methodone () ( )

}

**Diagram:**

Abstract

**Overriding is possible**

Nonabstract

We can override a non-abstract method as abstract

This approach is helpful to stop availability of parent method implementation to the next level child classes.

**Example:**

Class parent{

Public void methodOne() {}

}

Abstract class child extends parent {

Public abstract void methodOne();

}

**Synchronized, strictfp, modifiers won’t keep any restrictions on overriding.**

**Diagram:**

Final nonfinal native abstract Synchronized strictfp

Nonfinal final nonnative nonabstract nonSynchronized nonstrictfp

Wrong right right right right right

While overriding we can’t reduce the scope of acceses modifier.

Example:

Class parent {

Public void methodOne() {}

}

class Child extends Parent {

protected void methodOne() {}

}

Output:

Compile time error :

methodOne() in Child cannot override methodOne() in parent ;

attempting to assign weaker access privileges ; was public

Diagram:

Public protected <default> private

Public protected/public <default>/protected/public overriding concept is

Not applicable

**Right right right wrong**

**Private<default<protected<public**

**Checked VS Un-checked Exceptions:**

The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.

The exceptions which are not checked by the compiler are called un-checked exception.

Runtime exceptions and its child classes, Errors and its child classes are unchecked except these the remaining are checked exceptions.

**Diagram:**

Object

Throwable

Exception Error

RuntimeException IOE Exception ,SQL Exception ,Servlet exception Linkage error

Arithmatic Expression EOFException virtualmachinerror Assertion

Nullpointerexception FileNOtFoundException VerifyError

IndexoutofboundExceeption interruptedOException

ArraryindexOutOfBounException outofmemeoryerror StackOverFlowErr

StringIndexOutOfBoundsException

illegalArgumentException

NumberFormatException

ClassCastException

illegalStateException

**Rule:** While Overriding if the child class method throws any checked exception compulsory the parent class method should throw the same checkd exception or its parent otherwise we will get compile tim error.

**But there are no restrictions for un-checked exceptions**.

**E**xaple:

Class Parent {

Public void methodOne() {}

}

Class Child extends Parent{

Punlic void methodOne() throws Exception()

}

**Out put:**

Compile time error :

methodOne() in Child cannot override methodOne() in parent;

overridden method does not throw java.lang.Exception.

**Example:**

1. Parent: public void methodOne()throws Exception

Child: public voidmethodOne() valid

1. Parent: public void methodOne()

Child: public void methodOne()throws Exception invalid

1. Parent: publilc void mehtodOne()tjrows Exception

Child: public void methodOne()throws Exception valid

1. Parent: public void methodOne()throws IOException

Child: public void methodOne()throws Exception invalid

1. Parent: public void methodOne()throws IOException

Child: public void mehtodOne()throws EOFException, FileNotFoundException valid

1. Parent: public void methodOne()throws IOException

Child: public void methodOne()throws EOFException, interrupted Exception invalid

1. Parent: public void methodOne()throws IOException

Child: public void methodOne()throws EOFException,ArithmeticException valid

1. Parent: public void methodOne()

Child: public void methodOne()throws

ArithmeticException,NullPointerException, ClassCastException, RuntimeException valid