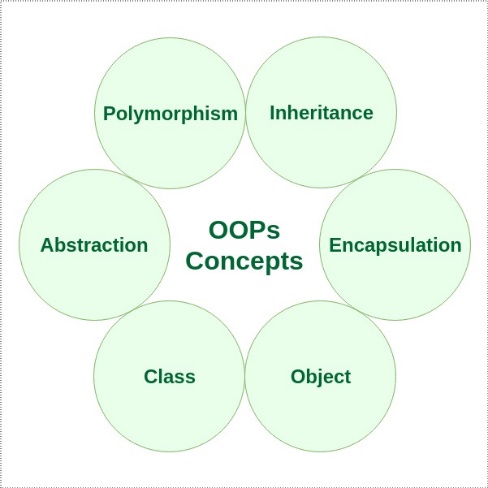
Object Oriented Programming



4 pillars of OOPs

1. Abstraction
2. Encapsulation
3. inheritance
4. polymorphism

Java source file structure

In java we can take any number of classes

Class A

{

}

Class B

{

}

--

--

Note:

You can use any name to java package

Even it content several classes a java program **contain at most one public class** and **save with public class name**

Class A

{

}

Public class B

{

}

--

--

Name: B.java

If program doesn’t contain any public class then we can save it by any name but the number of classes will create is same as total number of other classes in program

Class A{

P s v m(String[] args){

Sopln(“A class is main”);

}

}

Class B{

P s v m(String[] args){

Sopln(“ B class is main”);

}

}

Class C{

P s v m(String[] args){

Sopln(“class C is main”)

}

}

Program name: Durga.java

Here 3 classes will be generate A.class, B.class, C.class

**Import Statement**

public class Test{

p s v m(String[] args){

java.util.ArrayList al=new java.util.ArrayList();

}

}

due to which readability goes down

import java.util.ArrayList; //fully qualify name

public class Teat{

p s v m(String[] args){

ArrayList al=new ArrayList();

}

}

import

explicit implicit

java.util.ArrayList; java.util.\*;

Note: even if you are using 100’s of classes from same package you should go for explicit import because of

rediability

we are not required to use import the classes and interface present inside java.lang package

all classes present inside current directory doesn’t required import statement.

Whenever we are using import statement for package all classes & interface present inside are available but not sub packages.

**Packages**

It is grouped of related classes and interfaces

because of package

1. Naming conflict
2. Modularity
3. Maintainability
4. Security

Syntax

package com.durgasoft.ocja;

used client working domain name.

eg;

package com.durgasoft.ocja;

public class Test{

p s v m(String[] args){

sopln(“package demo”);

}

}

E:

com

durgasoft

ocja

Test.class

**Compiling**

javac –d ./E.Test.java

**run**

java com.durgasoft.ocja.Test

Note:

**In any java source file at most one package statement is allowed**

**Package statement must be first statement**

Class level modifiers

**Modifiers specifies the behaviour of class**

1. public

it means class is available to inside or outside of package both.

1. default

it means accessible inside package only

1. abstract

object creation is not possible for a class with abstract keyword

1. final

if class is final the child class creation is not possible.

Inner class means class inside another class

|  |  |
| --- | --- |
| Top level class modifier | Inner class modifier |
| public | private |
| <default> | protected |
| Abstract (important) | static |
| final |  |
|  |  |

1. public

package pack1;

public class A{

}

package pack2;

import pack.A

import class B{

p s v m(String[] args){

A obj=new A();

}

}

Because A is public class it is accessible in package 2 also

1. <default>

If there is no modifier then it is default

1. abstract

partially not completely

-applicable for method

-applicable for calss

abstract method

public class vehicle{

public abstract int getWheelOfVehicle();

}

-abstract method has only declaration but not implementation

-child class are responsible to provide implementation

-abstract method compulsory end with ;

-abstract method cannot have body

Syntax

Public abstract void m1();

abstract class

implementation of class is not complete such type of partial implemented classes are called abstract class

we cannot create object for abstract class.

abstract class Test{

p s v m(String[] args){

}

}

abstract class vs abstract method

1. if class contain at least one abstract method then class should be declare as a abstract

abstract class Test{

public abstract void main();

{ }

}

1. even though class doesn’t contain abstract method we can declare class as abstract.

http servlet is abstract which doesn’t contain any abstract method.

Note:

As child class is responsible to provide implementation for parent class

Therefore if parent class contain more then one abstract method then child class is also required to provide implementation to that number of abstract methods.

abstract class Vehicle{

public abstract int getNoOfWheels();

}

class Bus extends Vehicle{

public int getNoOfWheels(){

return 6;

}

}

class Auto extends Vehicle{

public int getNoOfWheels()

{

return 3;

}

class Test{

p s v m(String[] args){

Bus b=new Bus();

Sopln(b.getNoOfWheels());

Auto a=new Auto();

Sopln(a.getNoOfWheels());

}

}

If we don’t declare method as abstract then child class may or may not provide implementation to parent class

Member Modifiers(public and default)

Members modifiers are the modifiers which are applicable to methods and variables

If member is declare as public it can be accessible to anywhere within or outside the package only if corresponding method is declare as public

package pack1;

class A{

public void m1();

{

sopln(“A-class Method”);

}

}

package pack1;

import pack1.A;

public class B{

p s v m(String[] args){

A a=new A();

a.m1();

}

}

As class A is not public, therefore while compiling class B, compiler will throw error.

Otherwise not.

default member we can assess within package only

private member(variable or method)

private=class level

<default>=package level

public= global level

data modifiers= variable

recommended modifier for variable=private

recommended modifier for method=public

protected(member)

if variable/method is declare as protected then it can be **accessible anywhere within same package** but **only in child method/class of another package.**

**protected=<default> + child**

eg:

package pack1;

public class A{

protected void m1(){

sopln(“A class protected method”);

}

}

public class B extends A{

p s v m(String[] args){

A a=new A();

a.m1();

B b=new B();

b.m1();

A a1=new B();

a1.m1();

}

}

-since child class is also in the same package, we can use protected method with parent reference, child reference and parent reference with child method also.

-but child class is outside the package then we can only use child reference for the method.

Eg:

package pack1;

public class A{

protected void m1(){

sopln(“A class is protected method”);

}

}

package pack2;

import pack1;

public class B extends A{

p s v m(String[] args){

A a=new A();

a.m1();

B b=new B(); work fine

b.m1(); give CE

A a1=new B();

a1.m1();

}

}

Summary of public, protected and <default>

|  |  |  |  |
| --- | --- | --- | --- |
| **visibility** | **public** | **protected** | **<default>** |
| Within the same class | yes | yes | yes |
| From child class of same package | yes | yes | yes |
| From non-child class of same package | yes | yes | yes |
| From child class of another package | yes | yes(only if reference is also child class) | no |
| From non-child class of another package | yes | no | no |

**private<default<protected<public**

Inheritance

1. it is a relationship
2. code reusability
3. extends keyword

class B{

public void m1(){

sopln(“parent”);

}

}

Class C extends B{

Public void m2(){

Sopln(“child”);

}

}

-one method available for parent class.

-two methods available for child class.

-members of parent class by default available to the child class.

-members can be method/variable etc.

Class Test{

P s v m(String[] args){

B p=new B();

p.m1(); //possible

p.m2(); //not possible

-on parent reference p we can call any method which is available in parent class but not the method which is available in child class.

-on parent method we can’t call child specific members.

C c=new C();

c.m1(); //possible

c.m2(); //possible

-on child specific method we can call both parent and child specific method/variable

Importance of inheritance

-code reusability

-reduce development time

Total java API classes are implemented using inheritance

-parent content common methods

-child content specific methods

Types of Inheritance

1. single Inheritance

class A

class B

class B extends A

1. Multiple Inheritance

-here parent classes are multiple

Class A class B

class C

class C extends A, B

-single class inherit multiple parent class at a time.

-java doesn’t support multiple inheritance.

1. Multilevel Inheritance

class A

class B

class C

class B extends A

class C extends B

1. Hierarchal Inheritance

class A

class B class C class D

class B extends A

class C extends A

class D extends A

1. Hybrid Inheritance

It the combination of inheritances

class A

class B

class C

class D class E

class F

-java doesn’t support Hybrid inheritance

Note: hybrid and multiple inheritance does not in java.

Won’t multiple inheritance doesn’t support in java

Class C extends P1, P2

{

}

-ambiguity problem.

-diamond access problem

-if both parent class contain same method and if child class call that method then JVM get confuse to call which m1 method

-ambiguity problem

Class P1{

Public void m1(){

}

Class P2{

Public void m1(){

}

Class C extends P1, P2{ //not possible

}

}

Note:

In interface java support multiple inheritance.

-m1() -m1()

In interface only declaration but no implementation

Pi1 p1=new Test();

p1.m1();

-multiple declaration but one implementation that in child class.

Interface P1{

}

Interface P2{

}

Interface C extends P1, P2

{

}

JAVA

Public void m1(){

{

Sopln(“p1 method”);

}

}

Class P2{

Public void m2(){

Sopln(“p2 method”);

}

}

Class C extends P1, P2{

}

Which gives error.

PYTHON

class P1:

def m1(self):

print(“p1 method”)

class P2:

def m2(self):

print(“p2 method”)

class C extends(P1,P2):

pass

C =c()

c.m1()

**Note:**

**In case of python while extending parent class the class which calling first the method present in that call will execute, in this case as class P1 is calling first therefore m1 method present inside P1 will call.**

Polymorphism

**Method signature**

Public int m1(int i, float f)

Method signature consist of method name followed by argument type

m1(int, float);

-compiler is going to use method signature

-for every class compiler maintain method table

class Test{

public void m1(int i){

}

public void m2(String s){

}

} Test

|  |
| --- |
| m1(int) |
| m2(String) |

Method table

Test t=new Test();

t.m1(10);

t.m2(abc);

t.m2(10.6); //give error

Note:

Within same class two method with same signature is not allowed.

**Overloading**

If both methods having same name but different argument type then such type of methods are call overloaded methods.

m1(int);

m1(float);

m1(String);

class Test

{

public void m1(){

sopln(“no arg method”);

}

public void m1(int i);{

sopln(“int arg method”);

}

public void m1(double d){

sopln(“double arg method”);

}

}

p s v main(String[] args);{

Test t=new Test();

t.m1(); //no arg method

t.m1(10); // int arg method

t.m1(10.5); // double arg method

Case 1

Automatic promotion of argument types in method overloading

byte short

int long float double

char

-in this case if defined argument type is doesn’t contain in calling method then compiler will promote that argument type and display output.

class Test

{

public void m1(int i){

sopln(“int argument type”);

}

public void m1(float){

sopln(“float argument type”);

}

p s v m(String[] args){

Test t=new Test();

t.m1(10); //int argument type

t.m1(10.5); // float argument type

t.m1(“abc”); //promoted to int argument type

t.m1(10.5d); // error as double can’t be promote

case 2

In overloading method exact match will get highest priority.

class Test

{

Public void m1(object o);{

Sopln(“object version”);

}

Public void m1(String s);{

Sopln(“string version”)

}

P s v m(String[] args){

Test t=new Test();

t.m1(new object); // object version

t.m1(“Durga”); //string version

t.m1(null); //string version

}

Null is valid for any type of object reference

For null

Object -parent

String -child

**If parent and child argument are there then child will get preference**

Case 3

class Test

{

Public void m1(string s){

Sopln(“string version”);

}

Public void m1(stringBuffer sb){

Sopln(“stringbuffer version”);

}

}

P s v m(String[] args){

Test t=new Test();

t.m1(“durga”); //string version

t.m1(new stringBuffer(“durga”); //stringBuffer version

t.m1(null); //CE

**Note: if both methods arguments match and there is no parent child relation then CE will get**.

Object

String stringBuffer

Case 4

Class Test

{

Public void m1(int i){

sopln(“genric version”);

}

Public void m1(int ……… i){

Sopln(“var agr version”);

}

}

P s v m(String[] args){

Test t=new Test();

t.m1(); //var agr version

t.m1(10); //generic version

t.m1(10,120) //var agr version

**Note:**

**Exact match will get highest priority**

Case 4

class Test

{

Public void m1(int i, float f){

Sopln(“int-float version”);

}

Public void m1(float f, int i){

Sopln(“float-int version”);

}

P s v m(String[] args);

Test t=new Test();

t.m1(10, 10.5f); //int-float version

t.m1(10.5f, 10); // float-int version

t.m1(10,10); //CE

t.m1(10.5f,10.5f); //CE

}

Case 5

Class Animal

{  
}

Class Monkey extends Animal{

}

Class Test

{

Public void m1(Animal a){

Sopln(“animal version”)

}

Public void m1(monkey m){

Sopln(“monkey version”)

}

P s v m(String[] args){

Test t=new Test();

Animal a=new Animal();

t.m1(a); //animal version

Monkey m=new Monkey();

t.m1(m); //monkey version

Animal a1=new Monkey();

t.m1(a1); // animal version

}

}

**In method overloading method resolution is always take care by compiler based on reference type, runtime object never play any role.**

**Overloading also known as compile time polymorphism, static polymorphism or early binding.**

Method Overriding

Sometimes child class may not satisfy parent method implementation, then child is allowed to redefined method according to its requirement it is call method overriding.

Method overriding is also known as runtime polymorphism, dynamic polymorphism or late binding.

class P

{

Public void property(){

Sopln(“cash+gold”);

}

Public void marry(){

Sopln(“sita”+”gita”);

}

}

Class C extends P

{

Public void marry(){

Sopln(“sunny leone”)

}

}

Class Test

{

P s v m(String[] args){

P p=new P();

p.marry(); //parent method

C c=new C();

c.marry(); //child method

P p1=new C();

p1.marry(); //child method

}

}

How method resolution works

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

**Note:**

**While overriding we can’t reduce scope of modifiers but can increase**.

Parent public protected default

Child public protected/public default/protected/public

**For private overriding cannot applicable**

Method Hiding

class P

{

Public static void m1()[

Sopln(“parent”);

}

}

Class C extends P

{

Public static void m1(){

Sopln(“child”);

}

}

Class Test

{

P s v m(String[] args){

P p=new P();

p.m1(); //parent

C c=new C();

c.m1(); //child

P p1=new C();

p1.m1(); //parent

**Method resolution is always based on reference type.**

Polymorphism

Poly= many

Morphs=forms

Method overriding, method overloading comes under polymorphism

Polymorphism

Static| compile time| early binding dynamic| runtime| late binding

Overloading method hiding overriding