OOPS (Object Oriented Programming System): Object, Class, Inheritance, Polymorphism, Abstraction, Encapsulation

Object: entity that has state and behaviour example, chair, car etc

An object is an instance of class

Class is a group of objects which has common properties

A class in java contain Fields, methods, constructors, blocks, nested class and interface

Example: Object and class - main within class

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** a;

String b;

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

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

HelloWorld Hw =**new** HelloWorld();

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

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

}

}

Example: Object and class - main outside class

HelloWorld.java:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** a;

String b;

}

HelloWorld2.java:

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

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

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

}

}

3 ways to initialize object – reference variable, method, constructor

Reference variable Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** a;

String b;

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

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

HelloWorld Hw =**new** HelloWorld();

Hw.a =2;

Hw.b = "hello";

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

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

}

}

Access modifiers

Public - accessible by all classes

Private – accessible only in classes which it is defined

Protected – accessible within same package or subclasses in a different package

Default – accessible from the same package only

Method: way to perform some task, In java method is a block of code to perform certain task.

Declaration : public/private(access specifier) void/int(return type) Sum(method name) empty or int a, int b ( pararmeter list)

Example: public void car() or public int sum(int a, int b)

Two types – predefined and user defined

Predefined: built in methods like length(), equals(), sqrt()…etc

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

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

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

System.***out***.println("maximum number is "+Math.*max*(9,20));

}

}

User-defined method: method written by programmer

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** findEvenOdd(**int** a) {

**if** (a%2==0) {

System.***out***.println("even number"+a);

}**else** {

System.***out***.println("odd number"+a);

}

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.findEvenOdd(2);

hw.findEvenOdd(3);

}

}

Non-void method it mean returns a value

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **int** method(**int** a, **int** b) {

**int** c = a+b;

**return** c;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

System.***out***.println(hw.method(3, 4));

}

}

Initializing a variable using method

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** **void** assign(**int** a, **int** b) {

assgn1 = a;

assgn2 = b;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.assign(2, 4);

hw.show();

}

}

Another example where values passed to super class using the object of base class and displaying them

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** e;

**int** f;

**public** **int** method(**int** a,**int** b) {

**int** e = a + b;

**this**.e=e;

**return** e;

}

**public** **int** method(**int** a,**int** b, **int** c) {

**int** f = a + b + c;

**this**.f=f;

**return** f;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**int** c;

**int** d;

**public** **int** method1(**int** a,**int** b) {

**int** c = a + b;

**this**.c=c;

**return** c;

}

**public** **int** method2(**int** a,**int** b, **int** c) {

**int** d = a + b + c;

**this**.d=d;

**return** d;

}

**public** **void** show1() {

System.***out***.println("method1 answer is "+e);

System.***out***.println("method2 answer is "+f);

}

**public** **void** show2() {

System.***out***.println("method1 answer is "+c);

System.***out***.println("method2 answer is "+d);

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.method(2, 3);

hw2.method(2, 6, 4);

hw2.method1(3, 4);

hw2.method2(3, 4, 6);

hw2.show1();

hw2.show2();

}

}

This Keyword: this is a reference variable that refers to current object

Below is an example where if method/constructor parameters and instance variable name is same then passing parameter values to instance variables without using this keyword will not have any effect because it’s just initialising again to its own variable

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** **void** assign(**int** assgn1, **int** assgn2) {

assgn1 = assgn1; //no effect since passing to same variable

assgn2 = assgn2; //no effect since passing to same variable

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.assign(2, 4);

hw.show();

}

}

Below is an example where if method/constructor parameters and instance variable name is same then passing parameter values to instance variables using this keyword then the parameter variable is initialised to instance variable

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** **void** assign(**int** assgn1, **int** assgn2) {

**this**.assgn1 = assgn1; //initialises to instance variable

**this**.assgn2 = assgn2; //initialises to instance variable

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.assign(2, 4);

hw.show();

}

}

Like Above same using constructor or this keyword

Without this keyword example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** HelloWorld(**int** assgn1, **int** assgn2) {

assgn1 = assgn1;

assgn2 = assgn2;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld(2,4);

hw.show();

}

}

With using this keyword example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** HelloWorld(**int** assgn1, **int** assgn2) {

**this**.assgn1 = assgn1;

**this**.assgn2 = assgn2;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld(2,4);

hw.show();

}

}

Invoking method or constructor using this keyword

For Constructor example: this keyword invokes current class constructor. Also called constructor chaining generally used for reuse

Below this(3,4) invokes current class parameterised constructor

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** HelloWorld() {

**this**(3,4);

}

**public** HelloWorld(**int** assgn1, **int** assgn2) {

**this**.assgn1 = assgn1;

**this**.assgn2 = assgn2;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.show();

}

}

For Method example: this keyword invokes current class method

Below this.method2(10,20) invokes current class method with parameters

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** **void** method1() {

**this**.method2();

}

**public** **void** method2() {

**this**.assgn1 =10;

**this**.assgn2 =20;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.method1();

hw.show();

}

}

**Constructors**

It is a block of code similar to method. Called when instance of class is created.

Every time when an object is created using new() keyword it calls at least one constructor

It calls default constructor if no constructor is available and that default constructor is created by Java compiler.

Two types : no-arg constructor, parameterized constructor

Rules: constructor name same as class name, no return type, cannot be abstract, static, final

Default constructor created by compiler:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

//here default constructor will be created by compiler

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.show();

}

}

No-arg constructor created by programmer: used when no need to provide values

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** HelloWorld(){

**this**.assgn1 =10;

**this**.assgn2 =20;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.show();

}

}

Parameterized constructor: have some parameters, used to provide values

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**public** HelloWorld(**int** assgn1, **int** assgn2) {

**this**.assgn1 = assgn1;

**this**.assgn2 = assgn2;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld(10,20);

hw.show();

}

}

Constructor Overloading: just like method constructor can be overloaded as well, sometimes based on requirement different number of values may be passed to constructor hence constructor overloading is used.

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** assgn1;

**int** assgn2;

**int** assgn3;

**public** HelloWorld(**int** assgn1, **int** assgn2) {

**this**.assgn1 = assgn1;

**this**.assgn2 = assgn2;

}

**public** HelloWorld(**int** assgn12, **int** assgn22, **int** assgn23) {

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

**this**.assgn1 = assgn12;

**this**.assgn2 = assgn22;

**this**.assgn3 = assgn23;

}

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

System.***out***.println("values are " +assgn1+ " and "+assgn2+ " and "+assgn3);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld(10,20);

HelloWorld hw2 = **new** HelloWorld(10,20,30);

hw.show();

hw2.show();

}

}

Static Keyword: mainly used for memory management. Can be used for Variables (class variable), method (class method), Block, Nested class

Static Variable: declare variable as static it is called Static variable. Static variable gets memory only once…advantage is saves memory

Can assign value to variable without creating an object

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**static** **int** *id* =10;

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld.*id* =10;

System.***out***.println(HelloWorld.*id*);

}

}

Without using static keyword then we must create object to initialise non-static variable

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** id =10;

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw =**new** HelloWorld();

hw.id =10;

System.***out***.println(hw.id);

}

}

Let’s see without using static variable: Although it can’t be seen, but num gets memory created for every new object creation. Hence if you see below although two object creation happen hw and hw2 the num value will be printed as 1 only since memory is created each time and starts with zero.

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** id;

String name;

**int** num = 0;

**public** HelloWorld(**int** id,String name) {

**this**.id = id;

**this**.name = name;

num++;

}

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

System.***out***.println("values are " +id+ " and "+name+ " and "+num);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld(1,"Ram");

HelloWorld hw2 = **new** HelloWorld(2,"Raju");

hw.show();

hw2.show();

}

}

Let’s see with using static variable: Although it can’t be seen but num gets memory created only once. Hence if you see below although two object creation happen hw and hw2 the sum value for first time will be 1 since the object creation happened and memory got allocated and same memory used for second object creation and value increment to 2 and second time it prints 2

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** id;

String name;

**static** **int** *num* = 0;

**public** HelloWorld(**int** id,String name) {

**this**.id = id;

**this**.name = name;

*num*++;

}

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

System.***out***.println("values are " +id+ " and "+name+ " and "+*num*);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld(1,"Ram");

hw.show();

HelloWorld hw2 = **new** HelloWorld(2,"Raju");

hw2.show();

}

}

Static method: static keyword for a method is called static method

Static method belongs to class rather than object of class

Can be invoked without creating instance of class

Static method can access static data member and change value of it

Restrictions- static method cannot use non static data members or call non-static method directly (that’s why we use non-static method methods to call them we create object from main program)…we have seen an example above.

This and Super cannot be used in static context.

Why Java main method static – because object is not required to call static method. If it was not static method then every time we should create object to access static methods or variables from different classes and this is memory waste.

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **static** **void** method1() {

System.***out***.println("this is static method");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld.*method1*();

}

}

Another example

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**static** **int** *num* = 0;

**public** **static** **void** change() {

*num* =2;

}

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

System.***out***.println("values are "+*num*);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld.*change*();

HelloWorld.*show*();

}

}

Below is an example for static reference to a non-static field

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** num = 0;

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

System.***out***.println("values are "+num);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld.num =20;

HelloWorld.show();

}

}

Static block: initializes the static data member, static block is executed before the main method

Example:

**package** com.lokesh;

**public** **class** HelloWorld2 {

**static** {

System.***out***.println("this will be printed first");

}

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

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

System.***out***.println("this will be printed second");

}

}

**Inheritance**

It acquires all the properties and behaviour from parent class or objecct. Use is create new classes using existing classes. You can have new method and fields in child class

Inheritance --- IS –A relationship

Advantage is reusability

Terms used in inheritance

Class: group of objects which have common properties

Sub class/child class: inherits the parent class

Super class/parent class: super class is class where a sub class inherits its features

EXTENDS keyword indicates new class derived from existing class….here inherited class is parent class and inherting class is sub class

Below is an example for inheritance as you can see relationship is Programmer IS-A employee

**package** com.lokesh;

**public** **class** Employee {

**int** salary = 4000;

}

**package** com.lokesh;

**public** **class** Programmer **extends** Employee {

**int** bonus =1000;

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

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

Programmer p =**new** Programmer();

System.***out***.println(p.salary);

System.***out***.println(p.bonus);

}

}

Types of inheritance: Single, Multilevel, hierarchical, multiple(not supported…done using interface), hybrid

Single example:

**package** com.lokesh;

**public** **class** Dog {

**public** **void** name1() {

System.***out***.println("I am dog");

}

}

**package** com.lokesh;

**public** **class** Animal **extends** Dog {

**public** **void** name2() {

System.***out***.println("I am Animal");

}

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

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

Animal an = **new** Animal();

an.name1();

an.name2();

}

}

Hierarchial example:

**package** com.lokesh;

**public** **class** Dog {

**public** **void** name1() {

System.***out***.println("I am dog");

}

}

**package** com.lokesh;

**public** **class** Cat **extends** Dog {

**public** **void** name2() {

System.***out***.println("I am cat");

}

}

**package** com.lokesh;

**public** **class** Animal **extends** Cat {

**public** **void** name3() {

System.***out***.println("I am Animal");

}

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

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

Animal an = **new** Animal();

an.name1();

an.name2();

an.name3();

}

}

Aggregation: class have entity reference is known as Aggregation. It represents HAS-A relationship.

Below is an example where Employee has a address relationship. The advantage is this address class and its field can be used in any class by initiating data like below and use it this helps us not to create these fields again and again.

Example:

**package** com.lokesh;

**public** **class** Address {

String city;

String state;

String country;

**public** Address(String city, String state, String country) {

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

}

**package** com.lokesh;

**public** **class** Employee {

**int** id;

String name;

Address addr;

**public** Employee(**int** id, String name, Address addr) {

**this**.id =id;

**this**.name = name;

**this**.addr = addr;

}

**public** **void** display() {

System.***out***.println(id+" "+name);

System.***out***.println(addr.city+" "+addr.state+" "+addr.country);

}

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

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

Address ad1 = **new** Address("Hyderabad", "telangana", "India");

Address ad2 = **new** Address("Pune", "Maharashtra", "India");

Employee emp1 = **new** Employee(1, "Ramu", ad1);

Employee emp2 = **new** Employee(2, "Raju", ad2);

emp1.display();

emp2.display();

}

}

**Polymorphism**:

Method overloading and Method overriding

Method Overloading: methods have same name but different parameters . advantage is readability of program.

Two ways to overload method:

Changing number of parameters

Changing the data type

Changing number of parameters example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** c;

**int** d;

**public** **int** method(**int** a, **int** b) {

**this**.c = a+b;

**return** c;

}

**public** **int** method(**int** a, **int** b,**int** c) {

**this**.d = a+b+c;

**return** d;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

System.***out***.println(hw.method(3, 4));

System.***out***.println(hw.method(20, 30, 40));

}

}

Another way resultant variable isn’t created example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **int** method(**int** a, **int** b) {

**return** a+b;

}

**public** **int** method(**int** a, **int** b,**int** c) {

**return** a+b+c;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

System.***out***.println(hw.method(3, 4));

System.***out***.println(hw.method(20, 30, 40));

}

}

By changing the data type of arguments(parameters)

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **int** method(**int** a, **int** b) {

**return** a+b;

}

**public** **double** method(**double** a, **double** b) {

**return** a+b;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

System.***out***.println(hw.method(3, 4));

}

}

Method overloading is not possible by changing return type because of ambiguity (compiler will not understand to which method it should use)

We can overload main method as well. But JVM calls main() method which receives string array only

Type promotion – nothing but let’s say there is method1 with parameters int a and long b and if we pass parameter as a =20 and b = 40 the 40 is promoted to long………..byte can be promoted to short, int, long, float, double

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** method(**int** a, **long** b) {

System.***out***.println(a+b);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.method(2, 3); //3 is promoted to long

}

}

Below is an example of type promotion ambiguity since we passing value it doesn’t understand to which it should assign

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** method(**int** a, **long** b) {

System.***out***.println(a+b);

}

**public** **void** method(**long** a, **int** b) {

System.***out***.println(a+b);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

hw.method(2, 3); //3 is promoted to long

}

}

Method Overriding: if subclass has same method as parent class it is called method overriding

Used for runtime polymorphism, used to provide specific implementation of a method (each method has its functionality) which is already provided by parent/super class

Rules: Same method name in sub class and super class, same parameter in sub class and super class, must be IS-A relationship

Example: Consider a scenario where each bank has its own interest rate and bank is a class that provides functionality getInterest() . Now every each subclass changes as per its own requirement

**package** com.lokesh;

**public** **class** Bank {

**public** **int** getInterest() {

**return** 0;

}

}

**package** com.lokesh;

**public** **class** SBI **extends** Bank{

**public** **int** getInterest() {

**return** 7;

}

}

**package** com.lokesh;

**public** **class** ICICI **extends** Bank {

**public** **int** getInterest() {

**return** 8;

}

}

**package** com.lokesh;

**public** **class** MainClass {

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

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

SBI s =**new** SBI();

ICICI ic= **new** ICICI();

System.***out***.println(s.getInterest());

System.***out***.println(ic.getInterest());

}

}

Another Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** method() {

System.***out***.println("overridden method");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** **void** method() {

System.***out***.println("overriding method");

}

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

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

HelloWorld hw = **new** HelloWorld();

hw.method();

}

}

Another Example:

**package** com.lokesh;

**public** **class** Animal {

**public** **void** reveal() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal {

**public** **void** reveal() {

System.***out***.println("i am dog");

}

}

**package** com.lokesh;

**public** **class** MainClass **extends** Dog {

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

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

MainClass mc = **new** MainClass();

mc.reveal();

}

}

We cannot override static method because static method is bound with class whereas instance method is bound with object. Static belongs to class area while instance belongs to heap area).

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **static** **void** method1() {

System.***out***.println("static super class method");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** **void** method1() { //cannot override static method

System.***out***.println("trying to override static method of super class");

}

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

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

HelloWorld2 hw = **new** HelloWorld2();

hw.method1();

}

}

Co-variant return type: return type may vary same direction as sub class, before Java 5 not possible to override any method by changing return type

In below example return type of get() method of class HelloWorld is HelloWorld and for class HelloWorld2 get() method it is HelloWorld2. Both the methods have different return type but it is method overriding. This is called covariant return type

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** HelloWorld get() {

**return** **this**;

}

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

System.***out***.println("first class");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** HelloWorld2 get() {

**return** **this**;

}

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

System.***out***.println("second class");

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.get().show();

HelloWorld hw1 = **new** HelloWorld();

hw1.get().show();

}

}

Just for FYI

I tried changing return value “this” to “null” and I got error “cannot invoke com.lokesh.HellWorld.show()” because return value “com.lokesh.HellWorld.get()” is null

Super Keyword: Used to refer immediate parent class object

Uses: refer immediate parent class instance variable, invoke immediate parent class method, invoke immediate parent class constructor

Refer immediate parent class instance variable example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** String color ="white";

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

String color ="black";

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

System.***out***.println("color is"+color);

}

String color2 =**super**.color;

**public** **void** show2() {

System.***out***.println("color is "+color2);

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.show();

hw2.show2();

}

}

Invoke immediate parent class method:

Basic example:

without super keyword

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** method1() {

System.***out***.println("I am super class");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** **void** method1() {

System.***out***.println("i am sub class");

}

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

method1();

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.show();

}

}

With Super Keyowrd:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** method1() {

System.***out***.println("I am super class");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** **void** method1() {

System.***out***.println("i am sub class");

}

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

**super**.method1();

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.show();

}

}

In above we just used super keyword to call only super class below is example where both super and sub class method is executed

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **void** method1() {

System.***out***.println("I am super class");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** **void** method1() {

System.***out***.println("i am sub class");

}

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

**super**.method1();

method1();

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.show();

}

}

Another Example:

Below is an example without using super keyword where the methods ins super class didn’t get called only method1() in sub class got triggered

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** e;

**int** f;

**public** **int** method1(**int** a,**int** b) {

**int** e = a + b;

**this**.e=e;

**return** e;

}

**public** **int** method1(**int** a,**int** b, **int** c) {

**int** f = a + b + c;

**this**.f=f;

**return** f;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**int** c;

**int** d;

**public** **int** method1(**int** a,**int** b) {

**int** c = a + b;

**this**.c=c;

**return** c;

}

**public** **int** method1(**int** a,**int** b, **int** c) {

**int** d = a + b + c;

**this**.d=d;

**return** c;

}

**public** **void** show1() {

System.***out***.println("method1 answer is "+e);

System.***out***.println("method2 answer is "+f);

}

**public** **void** show2() {

System.***out***.println("method1 answer is "+c);

System.***out***.println("method2 answer is "+d);

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.method1(2, 3);

hw2.method1(2, 6, 4);

hw2.show1();

hw2.show2();

}

}

Below is an example where methods in super class are triggered using super keyword

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** e;

**int** f;

**public** **int** method1(**int** a,**int** b) {

**int** e = a + b;

**this**.e=e;

**return** e;

}

**public** **int** method1(**int** a,**int** b, **int** c) {

**int** f = a + b + c;

**this**.f=f;

**return** f;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**int** c;

**int** d;

**public** **int** method1(**int** a,**int** b) {

**super**.method1(a, b);

**int** c = a + b;

**this**.c=c;

**return** c;

}

**public** **int** method1(**int** a,**int** b, **int** c) {

**super**.method1(a, b, c);

**int** d = a + b + c;

**this**.d=d;

**return** c;

}

**public** **void** show1() {

System.***out***.println("method1 answer is "+e);

System.***out***.println("method2 answer is "+f);

}

**public** **void** show2() {

System.***out***.println("method1 answer is "+c);

System.***out***.println("method2 answer is "+d);

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.method1(2, 3);

hw2.method1(2, 6, 4);

hw2.show1();

hw2.show2();

}

}

Invoke immediate super class contructor:

In Construcors class, base constructor is called implicitly, that means super keyword is provide by compiler itself I you don’t provide it

In below example I haven’t provided super keyword, compiler provided it

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** HelloWorld() {

System.***out***.println("I am super class");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** HelloWorld2() {

System.***out***.println("i am sub class");

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

}

}

In below example I have provided explicitly

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** HelloWorld() {

System.***out***.println("I am super class");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** HelloWorld2() {

**super**();

System.***out***.println("i am sub class");

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

}

}

Another Example using super keyword:

**package** com.lokesh;

**public** **class** Person {

**int** id;

String name;

**public** Person(**int** id, String name) {

**this**.id=id;

**this**.name =name;

}

}

**package** com.lokesh;

**public** **class** Employee **extends** Person {

**float** salary;

Employee(**int** id, String name, **float** salary) {

**super**(id,name);

**this**.salary = salary;

}

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

System.***out***.println(id+" "+name+" "+salary);

}

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

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

Employee emp1 = **new** Employee(1,"Ramu",200f);

emp1.show();

}

}

Instance initializer block: used to initialize the instance data member .It runs each time when the object of class is created.

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** a;

**public** HelloWorld() {

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

}

{

a =10;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 {

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

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

HelloWorld hw = **new** HelloWorld();

}

}

Another Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** HelloWorld() {

System.***out***.println("super class constructor");

}

{

System.***out***.println("Super class instance initializer");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** HelloWorld2() {

System.***out***.println("sub class constructor");

}

{

System.***out***.println("sub class instance initializer");

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

}

}

Final keyword: used to restrict user (final means you cannot change the value), it can be variable, method or class

Final variable: if you make any variable as final, you cannot change the value of final variable (it will be constant)

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**final** **int** a =100;

**public** **void** method1() {

**this**.a =20; //final field cannot be assigned

}

}

Another Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**final** **int** a =100;

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.a =20; //final field cannot be assigned

}

}

Final method: if you make method as final, you cannot override it

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **final** **void** method1() {

System.***out***.println("final method");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** **void** method1(){ // cannot override final method

System.***out***.println("trying to overidr final method");

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

}

}

Final class: if you make any class as final, you cannot extend it.

**package** com.lokesh;

**import** java.util.Scanner;

**public** **final** **class** HelloWorld {

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld { // helloWorld2 cannot subclass the final class HelloWorld1

**public** **void** method1(){

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

}

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

}

}

Final method can be inherited but cannot override it. Below is an example

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**public** **final** **void** method1() {

System.***out***.println("final method of super class which cannot be overriden");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

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

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

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.method1();

}

}

Variable Uninitialized final variable:

A final variable that is not initialized at the time of declarartion is known as blank final variable.

If you want to create a variable that is initialized at the time of object creation and once initialized may not be changed. For example PAN CARD.

How to initialize then – using constructor

Example:

**package** com.lokesh;

**import** java.util.Scanner;

**public** **class** HelloWorld {

**int** name;

**final** **int** PANCARD;

**public** HelloWorld(**int** name, **int** PANCARD) {

**this**.name = name;

**this**.PANCARD = PANCARD;

}

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

System.***out***.println(name+" "+PANCARD);

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld {

**public** HelloWorld2(**int** name, **int** PANCARD) {

**super**(name, PANCARD);

}

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

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

HelloWorld2 hw = **new** HelloWorld2(1,123456);

hw.show();

}

}

We cannot define constructor as final because constructor is never inherited.

**Static and Dynamic binding:**

Connecting a method call to method body is called binding

Two types: Static (early binding) and dynamic (late binding) binding

Let’s understanding type of instance

Variable have a type: it may be a primitive or non-primitive

int a =10;

Above a variable is a type of int

References have a type: below d1 is a type of dog.

**package** com.lokesh;

**public** **class** Dog{

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

Dog d1;

}

}

Object has a type: Object is an instance of particular java class, but it is also instance of super class

In below d1 is instance of Dog class, but it also instance of super class i.e. Animal class

**package** com.lokesh;

**public** **class** Animal {

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

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

Dog d1 =**new** Dog();

}

}

Static Binding: Object is determined at compile time

If there is any final, static or private method in a class there is static binding

**package** com.lokesh;

**public** **class** Dog{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Dog d1 =**new** Dog();

d1.method1();

}

}

Dynamic Binding: Object is determined at runtime

In below example object type cannot be determined by compiler, because the instance of Dog is also an instance of Animal. So compiler doesn’t know its type, only its base type.

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method1() {

System.***out***.println("I am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Animal d1 =**new** Dog();

d1.method1();

}

}

Instance of operator (comparison operator): used to check whether the object is instance of specified type (class or subclass or interface)

Example:

**package** com.lokesh;

**public** **class** Dog{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Dog d1 =**new** Dog();

System.***out***.println(d1 **instanceof** Dog);

}

}

Another Example:

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method1() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Dog d1 =**new** Dog();

System.***out***.println(d1 **instanceof** Animal);

}

}

Another Example:

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method1() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Dog d1 =**new** Dog();

**if**(d1 **instanceof** Dog) {

d1.method1();

}

}

}

Downcasting with use of instanceof operator: when subclass type refers to the object of parent class is called downcasting. If we perform it directly, compiler gives compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.

Below is the example if we perform directly it gives compilation error.

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method1() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Dog d1 =**new** Animal(); //cannot convert from Animal to Dog

}

}

Below is an example if we do typecast it throws ClassCastException

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method1() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method1() {

System.***out***.println("I am dog");

}

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

Dog d1 =(Dog)**new** Animal(); //ClassCastException is thrown

d1.method1();

}

}

Possibility of downcasting with instanceof:

Below is an example where we prints the method from Animal class by downcasting

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method(Dog d1) {

**if**(d1 **instanceof** Animal) {

Animal d2= (Animal)d1;

d2.method1();

}

}

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

Dog d1 =**new** Dog();

d1.method(d1);

}

}

Another Example: can do without instance of as well

**package** com.lokesh;

**public** **class** Animal {

**public** **void** method1() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** Dog **extends** Animal{

**public** **void** method1() {

System.***out***.println("i am dog");

}

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

Dog d1 =**new** Dog();

Animal d2 =(Animal)d1;

d2.method1();

}

}

**Abstraction:**

Process of hiding implementation details and showing only functionality to the user.

Abstraction lets you focus what the object does instead of how it does

Two ways to achieve abstraction – Abstract class (0 to 100%), Interface (100%)

Abstract class: A class which is declared as abstract is called abstract class. It can have abstract and non-abstract methods. It needs to be extended and it method implemented. It cannot be instantiated.

Rules for Abstract class:

An Abstract class must be declared with an abstract keyword

Can have abstract and non-abstract methods

Cannot be instantiated

Can have constructors and static methods also

Can have final methods which will force the subclass not to change the body of the method

Abstract class –

abstract class HelloWorld{

}

Abstract method – abstract method doesn’t have any method body

abstract void method1();

Below is an example of abstract class that has an abstract method

Below is an example where walk implementation is hidden in abstract class and only functionality is provided

**package** com.lokesh;

**import** java.util.Scanner;

**public** **abstract** **class** HelloWorld {

**public** **abstract** **void** walk(); //no body

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld{

**public** **void** walk() {

System.***out***.println("I am walking...implementation provided by this class");

}

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

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

HelloWorld hw2 = **new** HelloWorld2();

hw2.walk();

}

}

Another example:

**package** com.lokesh;

**public** **abstract** **class** Bank {

**public** **abstract** **int** getInterestRate();

}

**package** com.lokesh;

**public** **class** Axis **extends** Bank {

**public** **int** getInterestRate() {

**return** 7;

}

}

**package** com.lokesh;

**public** **class** ICICI **extends** Bank{

**public** **int** getInterestRate() {

**return** 8;

}

}

**package** com.lokesh;

**public** **class** MainBankClass {

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

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

Bank ax = **new** Axis();

System.***out***.println(ax.getInterestRate());

Bank ic = **new** ICICI();

System.***out***.println(ic.getInterestRate());

}

}

Below is an example of Abstract class having constructor, abstract method and non-abstract method

**package** com.lokesh;

**import** java.util.Scanner;

**public** **abstract** **class** HelloWorld {

**public** HelloWorld() {

System.***out***.println("this is Constructor");

}

**public** **abstract** **void** walk(); //abstract method....no body

**public** **void** run() { //non-abstract method

System.***out***.println("I am running");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2 **extends** HelloWorld{

**public** **void** walk() {

System.***out***.println("I am walking...implementation provided by this class");

}

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

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

HelloWorld hw2 = **new** HelloWorld2();

hw2.walk();

hw2.run();

}

}

Interface: interface is blueprint of class. Interface is mechanism to achieve abstraction.

There can be only abstract methods in interface, not method body. It used to achieve abstraction and multiple inheritance.

In other words, interface can only have abstract methods and variables. It cannot have method body.

By default interface fields are public, static and final and methods are public and abstract.

Interface represent IS-A relationship

Just like Abstract class, interface also cannot be instantiated

From Java 8 we can have default and static methods in interface

From Java 9 we can have private methods in interface

Three reason to use Interface - achieve abstraction, we can support functionality of multiple inheritance, used to achieve loose coupling

Declaration of interface: interface is declared using interface keyword. It provides total abstraction that means all the methods are declared with no body, and all the fields are public static and final by default. A class that implements interface must implement all the methods declared in interface

Syntax:

**package** com.lokesh;

**public** **interface** interface\_name {

**int** ***a***; //fields that are public, static and final (means constant)

**void** draw(); // methods that are abstract by default

}

Relationship between class and interface: Class extends class, interface extends interface but a class implements interface

Example:

**package** com.lokesh;

**public** **interface** Printable {

**public void** print();

}

**package** com.lokesh;

**public** **class** Dog **implements** Printable{

**public** **void** print() {

System.***out***.println("I am dog");

}

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

Printable d1 =**new** Dog();

d1.print();

}

}

Below is an another example interface is implemented by two different classes

**package** com.lokesh;

**public** **interface** Printable {

**public void** print();

}

**package** com.lokesh;

**public** **class** Dog **implements** Printable{

**public** **void** print() {

System.***out***.println("I am dog");

}

}

**package** com.lokesh;

**public** **class** Animal **implements** Printable {

**public** **void** print() {

System.***out***.println("i am animal");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass {

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

Printable a1= **new** Animal();

a1.print();

}

}

Another example:

**package** com.lokesh;

**public** **interface** Bank {

**public** **int** getInterestRate();

}

**package** com.lokesh;

**public** **class** Axis **implements** Bank {

**public** **int** getInterestRate() {

**return** 7;

}

}

**package** com.lokesh;

**public** **class** ICICI **implements** Bank{

**public** **int** getInterestRate() {

**return** 8;

}

}

**package** com.lokesh;

**public** **class** MainBankClass {

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

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

Bank ax = **new** Axis();

System.***out***.println(ax.getInterestRate());

Bank ic = **new** ICICI();

System.***out***.println(ic.getInterestRate());

}

}

Multiple inheritance by interface: class implements multiple interfaces or an interface extends multiple interfaces it is known as multiple inheritance.

**package** com.lokesh;

**public** **interface** Printable {

**void** print();

}

**package** com.lokesh;

**public** **interface** Showable {

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

}

**package** com.lokesh;

**public** **class** Apple **implements** Printable,Showable {

**public** **void** print() {

System.***out***.println("Print an apple picture");

}

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

System.***out***.println("Show an real apple");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass{

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

Apple ap= **new** Apple();

ap.print();

ap.show();

}

}

Multiple inheritance is no supported by class because of ambiguity, it is supported in interface because there is no ambiguity. It is because its implementation is provided by implementation class.

Below is an example where Printable and Showable has same method Print() but its implementation is provided by the class MainInterfaceClass class, so there is no ambiguity

**package** com.lokesh;

**public** **interface** Printable {

**void** print();

}

**package** com.lokesh;

**public** **interface** Showable {

**public** **void** print();

}

**package** com.lokesh;

**public** **class** Apple **implements** Printable,Showable {

**public** **void** print() {

System.***out***.println("Print implementation is done here");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass{

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

Apple ap= **new** Apple();

ap.print();

}

}

Inheritance interface example: Below is an example of inheritance interface where an one interface extends another interface and the child interface is implemented by the class

**package** com.lokesh;

**public** **interface** Printable {

**void** print();

}

**package** com.lokesh;

**public** **interface** Showable **extends** Printable {

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

}

**package** com.lokesh;

**public** **class** Apple **implements** Showable {

**public** **void** print() {

System.***out***.println("print implementation is done here");

}

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

System.***out***.println("show implementation is done here");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass{

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

Apple ap= **new** Apple();

ap.print();

ap.show();

}

}

Default method in interface (from Java8): Since Java 8 we can have default method in interface, but we need to make it default method

Example:

**package** com.lokesh;

**public** **interface** Printable {

**void** print();

**default** **void** show() {

System.***out***.println("i am default method");

}

}

**package** com.lokesh;

**public** **class** Apple **implements** Printable {

**public** **void** print() {

System.***out***.println("I'm printing something");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass{

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

Apple ap= **new** Apple();

ap.print();

ap.show();

}

}

Another Example:

**package** com.lokesh;

**public** **interface** Printable {

**void** print();

**default** **int** add(**int** a,**int** b, **int** c) {

c = a + b;

**return** c;

}

}

**package** com.lokesh;

**public** **class** Apple **implements** Printable {

**public** **void** print() {

System.***out***.println("I'm printing something");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass{

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

Apple ap= **new** Apple();

ap.print();

System.***out***.println(ap.add(2, 3, 0));

}

}

Static method in interface (from Java 8) below is an example:

**package** com.lokesh;

**public** **interface** Printable {

**void** print();

**static** **int** add(**int** a,**int** b, **int** c) {

c = a + b;

**return** c;

}

}

**package** com.lokesh;

**public** **class** Apple **implements** Printable {

**public** **void** print() {

System.***out***.println("I'm printing something");

}

}

**package** com.lokesh;

**public** **class** MainInterfaceClass{

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

Apple ap= **new** Apple();

ap.print();

System.***out***.println(Printable.*add*(2, 3, 0));

}

}

Marker or tagged interface: which has no member for example Serializable, cloneable, etc

Used to provide some essential information to JVM so that JVM may perform some useful operation.

Example:

**package** com.lokesh;

**public** **interface** Serializable {

}

**Encapsulation:**

Package: group of similar type of classes, interfaces and packages

Two types – built-in package and user defined package

Built-in packages – for example java.lang, java.awt, java.io, java.util….etc

Advantages – used to categorize the classes and interfaces so that they are easily maintained, provides access protection, removes naming collision

User-defined package:

Below is an simple example

**package** com.lokesh;

**public** **class** HelloWorld2{

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

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

System.***out***.println("welcome to package");

}

}

How to access package from another package?

Three ways

Import package.\*;

Import package.classname;

Fully qualified name

Using packagename.\*: If you use package.\* then all the classes and interfaces of this package will but accessible but not sub packages

Import keyword is used to make packages and interfaces of another package accessible to the current package.

**package** com.lokesh;

**public** **class** HelloWorld2{

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

System.***out***.println("This is first class package");

}

}

**package** com.lokesh1;

**import** com.lokesh.\*;

**public** **class** HelloWorld3 {

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

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

System.***out***.println("This is second package class");

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.show();

}

}

Using packagename.classname: import package.classname then only that class of this package is accessible

**package** com.lokesh;

**public** **class** HelloWorld2{

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

System.***out***.println("This is first class package");

}

}

**package** com.lokesh1;

**import** com.lokesh.HelloWorld2;

**public** **class** HelloWorld3 {

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

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

System.***out***.println("This is second package class");

HelloWorld2 hw2 = **new** HelloWorld2();

hw2.show();

}

}

Using qualified name: if you use fully qualified name then only that class is accessible. Now there is no need to import package. But you need to use fully qualified name every time when you are accessing the class or interface

**package** com.lokesh;

**public** **class** HelloWorld2{

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

System.***out***.println("This is first class package");

}

}

**package** com.lokesh1;

**public** **class** HelloWorld3 {

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

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

System.***out***.println("This is second package class");

com.lokesh.HelloWorld2 hw2 = **new** com.lokesh.HelloWorld2();

hw2.show();

}

}

Subpackage: package inside a package

**package** com.lokesh.subpack;

**public** **class** Subpackclass {

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

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

System.***out***.println("subpack is subpackge of package com.lokesh and the class Subpackclass is in subpackage");

}

}

Access modifiers:

Public - accessible by all classes

Private – accessible only in classes which it is defined

Protected – accessible within same package or subclasses in a different package

Default – accessible from the same package only

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Access Modifier | Within class | Within package | Outside package  by subclass only | Outside package |
| Private | Y | N | N | N |
| Default | Y | Y | N | N |
| Protected | Y | Y | Y | N |
| Public | Y | Y | Y | Y |

Private: accessible only in classes which it is defined

**package** com.lokesh;

**public** **class** HelloWorld {

**private** **int** a =10;

**private** **void** show() {

System.***out***.println("this is private method");

}

}

**package** com.lokesh;

**public** **class** HelloWorld2{

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

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

HelloWorld hw =**new** HelloWorld();

System.***out***.println(hw.a); //compile time error a is not visible

hw.show(); // compile time error method show() not visible

}

}

Default : accessible from the same package only

**package** com.lokesh;

**public** **class** HelloWorld {

**int** a =10;

**void** show() {

System.***out***.println("this is default method");

}

}

**package** com.lokesh1;

**import** com.lokesh.\*;

**public** **class** HelloWorld3 {

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

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

System.***out***.println("This is second package class");

HelloWorld hw = **new** com.lokesh.HelloWorld();

System.***out***.println(hw.a); //compile time error a is not visible

hw.show(); // compile time error method show() not visible

}

}

Protected: accessible within same package or subclasses in a different package

**package** com.lokesh;

**public** **class** HelloWorld {

**protected** **int** a =10;

**protected** **void** show() {

System.***out***.println("this is private method");

}

}

**package** com.lokesh1;

**import** com.lokesh.\*;

**public** **class** HelloWorld3 **extends** HelloWorld{

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

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

System.***out***.println("This is second package class");

HelloWorld3 hw3 = **new** HelloWorld3();

System.***out***.println(hw3.a);

hw3.show();

}

}

Public - accessible by all classes

**package** com.lokesh;

**public** **class** HelloWorld {

**public** **int** a =10;

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

System.***out***.println("this is private method");

}

}

**package** com.lokesh1;

**import** com.lokesh.\*;

**public** **class** HelloWorld3 {

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

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

System.***out***.println("This is second package class");

HelloWorld hw = **new** HelloWorld();

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

hw.show();

}

}

**Encapsulation:** process of wrapping code and data together into a single unit.

We can create a fully encapsulated class by making all the data member of class private and now we can use setter and getter methods to set and get the data in it.

Advantages

By providing setter and getter methods, you can make the class read-only or write only i.e. you can skip getter or setter methods

Provides control over data. For example if you want value of id greater than 100 only, you can write logic inside setter method that not to store less than 100.

It is way to achieve data hiding, other class cannot access data through private data members

Example:

**package** com.lokesh;

**public** **class** HelloWorld {

**private** **int** id;

**private** String name;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2{

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

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

HelloWorld hw =**new** HelloWorld();

hw.setId(1);

hw.setName("Ramu");

System.***out***.println(hw.getId()+" "+hw.getName());

hw.setId(2);

hw.setName("Raju");

System.***out***.println(hw.getId()+" "+hw.getName());

}

}

Read-only class using encapsulation: In below example you can see the HelloWorld2 class can only read the data

**package** com.lokesh;

**public** **class** HelloWorld {

**private** **int** id =1;

**private** String name ="Ramu";

**public** **int** getId() {

**return** id;

}

**public** String getName() {

**return** name;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2{

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

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

HelloWorld hw =**new** HelloWorld();

System.***out***.println(hw.getId()+" "+hw.getName());

}

}

Write-only class using encapsulation: In below example you can see the HelloWorld2 class can only write the data

**package** com.lokesh;

**public** **class** HelloWorld {

**private** **int** id;

**private** String name;

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** **void** setName(String name) {

**this**.name = name;

}

}

**package** com.lokesh;

**public** **class** HelloWorld2{

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

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

HelloWorld hw =**new** HelloWorld();

hw.setId(1);

hw.setName("Ramu");

hw.setId(2);

hw.setName("Raju");

}

}