Data Hiding: use access modifiers like private to hide from outside world. Used for security purposes.

Abstraction: Hide implementation and just provide set of services we are offering. Achieved by use of interfaces and abstract keyword. Used for security.

Encapsulation: Tightly encapsulated if member variables are private. Improves modularity of the program. Encapsulation means encapsulating member variables and methods in a class.

Encapsulation = Data Hiding + Abstraction.

If parent class is not tightly encapsulated, implies child classes are also not tightly encapsulated.

Is-A: inheritance, using extends keyword. Reusability of code.

Whatever child class has, is not available to the parent class. Hence using a parent class reference, cannot access child specific members. Whereas, whatever parent class has is available to child classes too, hence can be accessed using parent class reference.

Q) class A extends B{

}

Class B extends A{}

o/p: CE cyclic inheritance not supported.

Q) class A extends A{}

o/p: CE cyclic inheritance not supported

Has-A: Composition & Aggregation. Creates maintenance problem as a class has other class reference, if one changes, difficult to maintain.

Composition: If container object is destroyed, contained objects are destroyed as well.eg: University has several departments. If university closes down, depts. Will also be closed.

Aggregation: If container is destroyed, contained objects may still exist. i.e. Container object just has the references of contained objects.

Method Overloading: In overloading, resolution is done by compiler based on reference type. Called compiletime polymorphism/ static polymorphism/ early binding. Automatic promotion also takes place in overloading. If matched method with specified argument not found, it is promoted and checked. Return type is not compared.

Byte-> short -> int -> long -> float -> double

Char

Q) class Test{

Public void m1(){}

Public int m1(){return 0;}

} o/p: CE

For overloading, name has to be the same and argument must be different.

Q) package Day1;

class Test2{

public void m1(StringBuffer sf){

System.out.println("in string buffer method");

}

public void m1(String sf){

System.out.println("in string method");

}

public void m1(int sf){

System.out.println("in int method");

}

public void m1(Integer sf){

System.out.println("in Integer method");

}

public void m1(String[] sf){

System.out.println("in string array method");

}

/\*public void m1(String... sf){

System.out.println("in string- var method");

}\*/

//String[] and var-arg string is same.

public static void main(String args[]){

Test2 obj = new Test2();

obj.m1("hello");

obj.m1(new StringBuffer("hi"));

obj.m1(null);

}}

CE: m1(null) is ambiguous.

Child argument will have a higher priority than a parent argument

class Test2{

public void m1(Object sf){

System.out.println("in object method");

}

public void m1(String sf){

System.out.println("in string method");

}

public static void main(String args[]){

Test2 obj = new Test2();

obj.m1("hello");

obj.m1(new StringBuffer("hi"));

obj.m1(null);

}

}

o/p: in string method

in object method

in string method

Method Overriding: Method name and argument must be same. Return types are matched but Covariant return types are allowed. Covariant return type is valid only for objects, not primitive datatype.

Q) class Test2{

public void m1(String sf){

System.out.println("in string method");

}

}

class T extends Test2{

public int m1(String sf){

System.out.println("in string method");

}} o/p: CE

Q) class Test2{

public Object m1(String sf){

System.out.println("in string method");

return null;

}}

class T extends Test2{

public String m1(String sf){

System.out.println("in string method");

return null;

}}

o/p compiles fine.

Q) class Test2{

public int m1(String sf){

System.out.println("in string method");

return 0;

}}

class T extends Test2{

public long m1(String sf){

System.out.println("in string method");

return 0l;

}} o/p: CE

Q) class Test2{

public int m1(String sf){

System.out.println("in string method");

return 0;

}}

class T extends Test2{

public Integer m1(String sf){

System.out.println("in string method");

return 0;

}

} o/p : CE

While overriding we cannot limit the access scope of the modifier. Private<default<protected<public. Child class cannot throw any new exception unless it’s a unchecked exception.

Q) class Test2{

public int m1(String sf){

System.out.println("in string method");

return 0;

}

}

class T extends Test2{

public int m1(String sf) throws IOException{

System.out.println("in string method");

return 0;

}

}

o/p: CE

Q) class Test2{

public int m1(String sf)throws IOException{

System.out.println("in string method");

return 0;

}

}

class T extends Test2{

public int m1(String sf) throws NullPointerException{

System.out.println("in string method");

return 0;

}

}

o/p: Compiles fine.

Method Hiding: Both parent and child class methods are static. Method called by compiler based on reference type, considered as compile time polymorphism. In overriding, resolution is based on runtime object , called runtime polymorphism.

class Test21{

int m1(String sf)throws IOException{

System.out.println("in m1 string method of parent class");

return 0;

}

public static int m2(String sf) {

System.out.println("in m2 string method of parent class");

return 0;

}}

class Test2 extends Test21{

@Override

public int m1(String sf) {

System.out.println("in m1 string method of child class");

return 0;

}

public static int m2(String sf) {

System.out.println("in m2 string method of child class");

return 0;

}

public static void main(String args[]) throws IOException{

Test21 obj = new Test2();

obj.m2("hello");

obj.m1("hello");

}}

o/p: in m2 string method of parent class

in m1 string method of child class

Q) class Test21{

int m1(String... sf)throws IOException{

System.out.println("in m1 string method of parent class");

return 0;

}

public static int m2(String sf) {

System.out.println("in m2 string method of parent class");

return 0;

}

}

class Test2 extends Test21{

@Override

public int m1(String[] sf) {

System.out.println("in m1 string method of child class");

return 0;

}

public static int m2(String sf) {

System.out.println("in m2 string method of child class");

return 0;

}

public static void main(String args[]) throws IOException{

Test21 obj = new Test2();

obj.m2("hello");

obj.m1(new String[]{"hello", "hi"});

}

}

o/p: in m2 string method of parent class

in m1 string method of child class

warning: a var-arg should only override/ be overridden by var-arg

In overloading, we only check name and arguments, exceptions, return types, access modifiers are not checked. In overriding, we check everything.

Polymorphism: Compile time: overloading & method hiding + Runtime: overriding.

Q) class Test21{

static {

System.out.println(x);

}

static int x=10;}

o/p: CE. Cannot reference a field before it is defined.

class Test2{

1 static int x=10; 7

2 static {

m1(); 8

System.out.println("FSB"); 10

}

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

m1(); 13

System.out.println("main method"); 15

}

4 public static void m1(){

System.out.println(y); 9 ,14

}

5 static{

System.out.println("SSB"); 11

}

6 static int y=20;} 12

o/p: 0

FSB

SSB

20

main method

First it reads all static members, then executes them in top to bottom approach, then executes the main method.

**Read Indirectly, Write only state:** If a variable is in this state, then we can’t perform read operation, else CE. Illegal forward reference.

Q) **class** Test2{

**static** **int** *x*=10;

**static** {

*m1*();

System.*out*.println("base SB");

}

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

*m1*();

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

}

**public** **static** **void** m1(){

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

}

**static** **int** *y*=20;

}

**class** Derived **extends** Test2{

**static** **int** *i*= 100;

**static**{

*m2*();

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

}

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

*m2*();

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

}

**public** **static** **void** m2(){

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

}

**static**{

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

}

**static** **int** *j*=200;

}

o/p: java Derived:

0

base SB

0

DFSB

DSSB

200

derived main method

java Test2:

0

base SB

20

base main method

When we are loading a child class, automatically parent will get loaded, hence executing first parent static blocks.

**class** Test2{

**static** **int** *x*=10;

**static** {

*m1*();

System.*out*.println("base SB");

}

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

Derived.*main*(args);

*m1*();

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

}

**public** **static** **void** m1(){

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

}

**static** **int** *y*=20;

}

**class** Derived{

**static** **int** *i*= 100;

**static**{

*m2*();

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

}

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

*m2*();

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

}

**public** **static** **void** m2(){

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

}

**static**{

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

}

**static** **int** *j*=200;

}

o/p: 0

base SB

0

DFSB

DSSB

200

derived main method

20

base main method

Q) **class** Test2{

**int** x=10;

{

m1();

System.*out*.println("base SB");

}

Test2(){

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

}

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

Test2 obj= **new** Test2();

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

}

**public** **void** m1(){

System.*out*.println(y);

}

{

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

}

**int** y=20;

}

o/p: 0

base SB

SIIB

Constructor

base main method

Q) **package** Day1;

**class** Test2{

**int** x=10;

{

m1();

System.*out*.println("parent SB");

}

Test2(){

System.*out*.println("Parent Constructor");

}

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

Test2 obj= **new** Test2();

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

}

**public** **void** m1(){

System.*out*.println(y);

}

**int** y=20;

}

**class** Child **extends** Test2{

**int** i=100;

{

m2();

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

}

Child(){

System.*out*.println("child constructor");

}

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

Child obj= **new** Child();

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

}

**public** **void** m2(){

System.*out*.println(j);

}

**int** j=200;

}

Java Child:

0

parent SB

Parent Constructor

0

CIIB

child constructor

child main