**OOP’s (Object oriented programming ):**

**Inheritance:**

Inheritance is the process of deriving child class from the parent class ,through the process child is inheriting some properties of the parent ,(variables and methods).

Parent class- base class/super class

Child class- derived class/child class/sub class

There are mainly five types of inheritance :

**1.Single inheritance:**

Single child class is inheriting from parent class that is called single inheritance.

class SingleInheritance

{

Public static void main(String args[])

{

Child c=new child();

}

}

class parent

{

void parent()

{

Syso(“parent class”);

}

}

class child extends parent

{

void child()

{

Syso(“child class”);

}

}

Parent → child

**2.Multilevel inheritance:**

There are levels of inheritance, like a base class where subclass1 is derived from the superclass, and subclass2 is derived from subclass1.

It is like the father inheriting from our grandfather, and the child inheriting from the father.

grandfather → father → child

class MultipleInheritance

{

Public static void main(String args[])

{

Child c=new child();

}

}

class grand

{

void parent()

{

Syso(“parent class”);

}

}

class parent extends grand

{

void child()

{

Syso(“child class”);

}

}

class child extends parent

{

void child()

{

Syso(“child method”);

}

}

**3.Haririchal Inheritance :**

More than one subclass derived from a single parent class is called hierarchical inheritance.

class Hirarichal\_Inheritance

{

Public static void main(String args[])

{

Child c=new child();

}

}

class grand

{

void parent()

{

Syso(“parent class”);

}

}

class parent extends grand

{

void child()

{

Syso(“child class”);

}

}

class child extends grand

{

void child()

{

Syso(“child method”);

}

}

**Constructor:**

* The main use of the constructor is to initialize the instance variables.
* Constructor name should be the same as class name by using constructors we can initialize the instance variables.
* this keyword is used to initialize the instance variables.

Rules:

* Constructor Name as class name.
* Constructor should not contain any return type not even void.
* Constructors can be overloaded.
* Constructers are not be override.

Because, we are creating constructor same as class name each and every class have own constructors, with respective class names.

* Declaring constructor is not required when we not supply any constructor to a class . java compiler automatically creates a default constructor which is empty, by default it call its super class using super(); method .

(Since object class is the parent class for all the classes in java , if there is no explicit parent class specified, the superclass of the class is `Object`. The default constructor will then implicitly call `super()`)

* Compiler won’t generate any default constructor if there is a constructor defined already inside the class.
* Constructors are not inherited.
* Constructors can have access modifiers :  
  public, private, default, protected other than this we are not suppose to use the other access modifiers for the class.
* The default constructor have some access modifier same as the class
* The constructor calling should be first statement by using within the constructor.  
  this() {used to access same class constructor)  
  super() {used to access parent class constructor)  
    
  calling one constructor inside the another constructor is calling constructor chaining.

**this:**

this is a keyword is used to access the current class variables and constructors

and methods**.**

By using this keyword in the program it as actually good practice to write the codes because when someone see the code can easily understand ,corresponding variable or method or constructor belongs to the same class only.

**Polymorphism:**

**Ability of an object take in many forms, the most common use of polymorphism in OOPs is ,when parent class reference is used for child class object.**

**Example:**

I am an example of polymorphism because at home, I am treated as a son, while in an institute, I am treated as a student, and when I go to a shop, I am treated as a customer. Here, the person is the same but is treated differently depending on the context. This is what polymorphism is about.

Similarly, in Java, an object can be used in many forms.

**There are two types of polymorphism :**

* **Compile time Polymorphism**:  
   🡪 Constructor overloading  
   🡪 Method overloading
* **Runtime Polymorphism:**

**🡪**Method overriding

**Note:**

Compile time polymorphism is depends on programmer because ,type of arguments provided by user at the time of object creation ,so here instruction provided by user to compiler .

**Constructer overloading:**

Within the same class, writing 2 or more constructors with different parameters.

Constructor overloading in Java allows a class to have multiple constructors with different parameter lists. The constructor that gets called during object creation is determined by the arguments provided by the programmer. Depending on the number and types of arguments passed to the constructor, the appropriate overloaded constructor is selected by the Java compiler at compile time

**Method overloading:**

Within the same class 2 or more methods having same method name but different number of parameters.

There are some rules:

* Same method but parameters should be different.
* May have different return types
* May have different modifier’s.

Examples:

String methods and println() is an example of method overloading

System.out.println(10); // Calls println(int x)

System.out.println("Hello"); // Calls println(String x)

System.out.println(5.5); // Calls println(double x)

String example = "Hello World";

int index1 = example.indexOf('o'); // Calls indexOf(int ch)

int index2 = example.indexOf("World"); // Calls indexOf(String str)

int index3 = example.indexOf("World", 5); // Calls indexOf(String str, int fromIndex)

**Rum time polymorphism:**

**Method overriding:**

When 2 or more methods having same method name with same method signature (same number of parameters).and there should be an IS-A relationship among the classes.

Till java 1.4 version:

The parent class return type should be same as child class return type, while overriding the method after java 1.4 version of java, provides the special functionality called covariant return types.

Definition:

covariant return types allow a method in a subclass to override a method in a superclass with a more specific return type.

Co-variant:

* After java 1.4 version covariant return types are available.
* This will allow programmer to override the method in parent class with child class method with same return type   
   (or)  
  sub class method return type method may be the sub class of super class method return type.

Example:

public class Method\_Overiding

{

public static void main(String[] args)

{

over obj =new over1();

obj.add();

}

}

class over

{

Object add()

{

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

return ;

}

}

class over1 extends over

{

*@override*

Integer add()

{

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

return 0;

}

}

class over

{

int add()

{

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

return 0;

}

}

class over1 extends over

{

*@override*

int add()

{

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

return 0;

}

}



class over

{

int add()

{

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

return 0;

}

}

class over1 extends over

{

*@override*

float add()

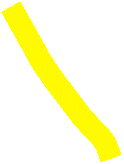
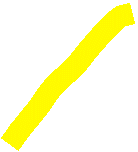
{

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

return 0;

}

}



**Final keyword:**

|  |  |
| --- | --- |
| Type | Description |
| Final variable  Final method  Final class | Cannot reassign  Cannot override  Cannot extended or inherited |

* The main use of final variable is, we cannot further reassign the value for a variable, when we are working the unique data like Adhaar or pan number is really good, because the data is fixed, not changeable .at the time we are using final keyword for a variable.
* Overriding is not possible, if a method is declared as final keyword.
* Class is defined with final keyword we are not further extended

**static keyword:**

* if a method is defined with static keyword, we no need to create an object
* for static method ,we directly call a method inside the main method  
  or classname.methodname()
* The main use of static method is memory management ,object creation is not needed for static method, when we create a static method

For non-static variables we need to create an object and access using obj can access the data

Class hello

{

int a=10;

{

public static void main(String args[])

{

Hello obj =new hello();

Syso(obj.a);

}

}

*@override*after this annotation we must and should write the override method ,if suppose we are trying to write another method it throws error

public class Method\_Overiding

{

public static void main(String[] args)

{

over obj =new over1();

obj.*hello*();

}

}

class over

{

static void hello()

{

System.***out***.println("hello hi bye good night");

}

}

class over1 extends over

{

static void hello()

{

System.***out***.println("hii madam");

}

}

**Method hiding:**

* A static method in a subclass may hide

the another method in the super class ,

this complete process is called method

method hiding.

* it is the process of hiding the method with

another method.

**Upcasting**

Parent class reference given to child class object this is called upcasting.

public class Method\_Overiding

{

public static void main(String[] args)

{

**over obj =new over1();**

obj.add();

}

}

class over

{

int add()

{

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

return 0;

}

static void hello()

{

System.***out***.println("hello hi bye good night");

}

}

class over1 extends over

{

static int *a*=89;

*@override*

int add()

{

*a*=6;

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

return 0;

}

static void hello()

{

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

}

}

* by using upcasting creating object for class child class the compiler first will check the methods in parent class , weather present or not ,if present it overrides the methods with child class methods .

**Note:**

other than parent class methods is not allowed to write/create in the child class see the example below:

public class Method\_Overiding

{

public static void main(String[] args)

{

over obj =new over1();

obj.add();  
 obj.mul(); **//throws error**

}

}

class over

{

int add()

{

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

return 0;

}

static void hello()

{

System.***out***.println("hello hi bye good night");

}

}

class over1 extends over

{

static int *a*=89;

*@override*

int add()

{

*a*=6;

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

return 0;

}

static void hello()

{

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

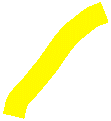
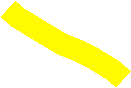
}

void mul()

{

}

}



**Blocks:**

* **static blocks:**
* **instance blocks:**

**Static blocks:**

* **it executes only once’s in the program.**
* **when program is executed first static blocks is executes then remaining code is executed.**
* **if we want to execute a block of code only once at the time we are using the static blocks.**
* **we can initialize the static variables by using static blocks.**

**Instance Blocks:**

* **it will be executed for every object creation**
* **after object creation it automatically executes there is no need to call explicitly.**
* **Throughout the code for every object creation that, many times it will printing the instance blocks.**

public class Hello

{

static int *a*;

static

{

*a*=5;

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

**//static blocks**

}

{

**//instance blocks**

}

public static void main(String[] args)

{

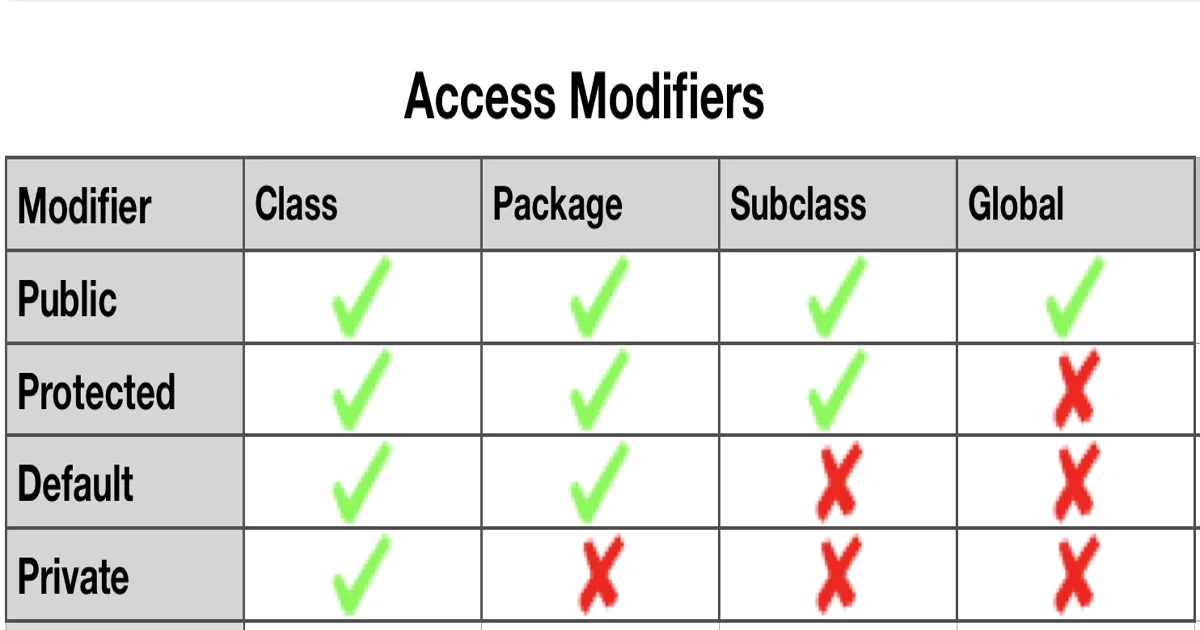
Hello obj=new Hello();

}

}

**Access modifiers vs modifiers:**

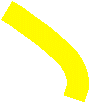
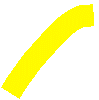
* private, public, protected, default these are called access specifiers in **earlier versions** of java remaining all are access modifiers .
* but not there is **no access specifier’s** **in java** all are access modifiers only.
* for each and every access specifier has their own scope ,we are not supposed to access outside the scope.
* we can increase the scope there is no problem
* we are not allowed to decrease the scope, it will an error.



example:

private hello() public hello() default hello()

public hello() private hello() public hello()



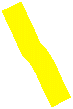
**Abstraction:**

* **Abstraction is the process of hiding internal details and display only necessary functionalities to user .**
* **Abstraction is done by interfaces and abstract class in java.**
* **Advantages:**
  + **data privacy maintains**
  + **reduce complexity and increase the readability of the code**
  + **reusability.**

**Note:  
Abstraction is generally aching using abstract keyword in java.**

If a class contains abstract keyword, it may or may not contains single abstract methods in the abstract class,(but the thing is, suppose the class is defined with abstract keyword ,not able to create an object for abstract class and provide implementation in another abstract class).

Example:



easy example for abstraction



public class abstraction

{

public static void main(String[] args)

{

ram obj=new ram();

obj.sum();

}

}

abstract class ravi

{

}

class ram extends ravi

{

void sum()

{

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

}

}

public class abstraction

{

public static void main(String[] args)

{

ram obj=new ram();

obj.sum();

}

}

abstract class ravi

{

abstract void sum();

}

class ram extends ravi

{

void sum()

{

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

}

}

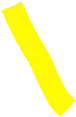
I

Implemented methods unimplemented methods

Public void sum() public abstract void sum();

{

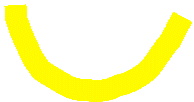
}



* These are unimplemented methods because the implementation is not provided here
* Method ends with ;

These are implemented methods because the implementation is provided here inside the method

|  |  |  |
| --- | --- | --- |
| **Class** | **Abstract class** | **Interface** |
| Non abstract methods | Non abstract +abstract | methods |



class is used to provide the implementation for the



non abstract methods.  
(or)

provide implementation for unimplemented methods

overriding

Nonabstract.java abstractclass.java

package abstraction;

public class Nonabstract extends abstractclass

{

*@Override*

void sum()

{

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

}

*@Override*

void add()

{

System.***out***.println("hello Hi bye good night");

}

}

package abstraction;

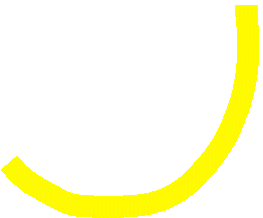
public abstract class abstractclass

{

abstract void sum();

abstract void add();

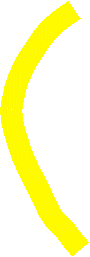
}



By using non abstract class we are providing implementation

for the unimplemented methods which are present in the

abstract class.



Text2.java

package abstraction;

public class Text2 {

public static void main(String[] args)

{

Nonabstract obj1=new Nonabstract();

obj1.add();

obj1.sum();

}

}



By using non abstract class we are creating obeject and

Access the data using object name.methodname()

Interfacemethod.java interfacemethod.java

package abstraction;

public class Text1 implements interfacemethod

{

*@Override*

public void mul()

{

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

}

*@Override*

public void div()

{

System.***out***.println("i am in div methdod");

}

}

package abstraction;

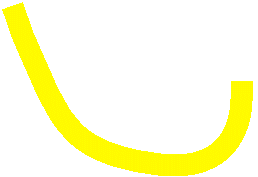
public interface interfacemethod

{

public void div();

public void mul();

}



Text2.java

package abstraction;

public class Text2 {

public static void main(String[] args)

{

Nonabstract obj1=new Nonabstract();

obj1.add();

obj1.sum();

}

}



**Interfaces:**

* **By using interfaces can achieve 100% abstraction in java.**
* **Interfaces have methods .**
* **After java 1.8 version, we have default and static methods are available in java.**
* **By default, the methods in interfaces methods have public and abstract no need to specify explicitly.**
* **Scope decrease is allowed in interfaces.**
* **Dimond problem is solved by using interfaces.**
* **Multiple inheritance is supported achieved by using interfaces.**
* **Abstract/ unimplemented methods in interfaces cannot implemented in interfaces, must and should only implemented in classes only.**
* **Examples:**

interface Animal

{

public void sound(); // public scope

}

interface Dog extends Animal

{

@Override

protected void sound(); // decreasing the scope to protected (allowed)

}

**Multiple inheritance:**

child is inherited with parent and grand parent characteristics.

package abstraction;

public class Multipleinheritancce

{

public static void main(String args[])

{

child c=new child();

c.parent();

c.grand();

}

}

interface grand

{

void grand();

}

interface parent

{

void parent();

}

class child implements grand,parent

{

*@Override*

public void parent()

{

System.***out***.println("grand father");

}

*@Override*

public void grand()

{

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

}

}

**Diamand problem(Hybrid inheritance):**

Combination any two inheritance is called hybrid inheritance.

class GrandFather

{

void grand()

{

System.out.println("I am a grand father");

}

}

interface Father

{

void parent();

}

class FatherImpl extends GrandFather implements Father

{

@Override

public void parent()

{

System.out.println("I am a father");

}

}

class Child extends FatherImpl

{

@Override

public void parent()

{

System.out.println("I am a child");

}

}

public class HybridInheritance {

public static void main(String args[]) {

Child c = new Child();

c.grand();

c.parent();

}

}

**Exception handling:**

Exception are nothing but runtime errors. due to these exception the abnormal flow of execution of the program is disturbed and terminated abnormally.

**Cheked exceptions:**

Checked exceptions are the exceptions, these are checked by compiler at compile time, for smooth termination of program in runtime.

note:

what ever the exception is they must be occurred only at runtime

Handling checked exceptions:

we can handle checked exceptions in two ways.

* try 🡨🡪 catch blocks {use for handling both checked and unchecked exceptions}
* throw 🡨 🡪throws blocks {methods and constructors}

public class exception

{

public static void main(String[] args)

{

try

{

System.***out***.println(10/0);

}

catch(ArithmeticException e)

{

System.***out***.println("handed by catch block");

}

}

}

|  |  |
| --- | --- |
| Checked exceptions | Unchecked exceptions |
| * The exception are checked by compile time * Compiler checks the exceptions * We can avoid this exception we can handle these exceptions by proper writing the code. * IOException ,SQLexception ,ClassNotFoundException * not in user/programmer control | * These exceptions are checked at runtime * Jvm checks the exceptions * We can avoid this ,while writing the code we may aware of these to prevent these exceptions * ArthematicException,NullPointerException ,ArrayIndexoutOfBounceException etc. * under programmer control |

|  |  |
| --- | --- |
| Throw | Throws |
| Throw the exception explicitly  Used for both checked and unchecked exceptions  Specially inside the method implementation  Only one exception will be throw at a time | One of the mechanisms to handle exception  Used for only declaring checked exceptions  Declare exception with method signature  Can handle multiple exceptions |