Object-Oriented Principles:

Abstraction in real world:

* Process of hiding the implementation details and shows only the functionality.
* The user will have information only about what the object does and not how it does.

Abstraction in Java:

* A class which is declared as abstract is known as an **Abstract Class**.
* There are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class that is not complete on its own.
* Since abstract class allows concrete methods as well, it does not provide 100% abstraction. It provides partial abstraction. Interfaces on the other hand provides 100% abstraction.
* Abstraction in java is achieved through,
  + Abstract Class
  + Interfaces.

Abstract Class :

Rules:

* Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods (those that were abstract in parent but implemented in child class).
* The class itself cannot be made final but it can have final methods.
* A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.
* If class has at least one abstract method, the class **MUST** be declared abstract.
* It can have abstract and non-abstract methods.
* Abstract method is a method that is declared without an implementation.
* If a child does not implement all the abstract methods of abstract parent class, then the child class need to be declared abstract as well.

Example:

**package** OOPs;  
  
**abstract class** Bank  
***{* abstract int** getRateOfInterest***()***;  
***}*class** CommBank **extends** Bank  
***{* int** getRateOfInterest***()  
 {* return** 7;  
 ***}  
}*class** ANZ **extends** Bank  
***{* int** getRateOfInterest***()  
 {* return** 8;  
 ***}  
}*class** Westpac **extends** Bank  
***{* int** getRateOfInterest***()  
 {* return** 10;  
 ***}  
}*public class** abstraction  
***{* public static void** main***(***String args***[])  
 {*** Bank b;  
 b=**new** CommBank***()***;  
 System.***out***.println***(*"Rate of Interest for CommBank is: "**+b.getRateOfInterest***()***+**" %"*)***;  
 b=**new** ANZ***()***;  
 System.***out***.println***(*"Rate of Interest for ANZ is: "**+b.getRateOfInterest***()***+**" %"*)***;  
 b = **new** Westpac***()***;  
 System.***out***.println***(*"Rate of Interest for Westpac is: "**+b.getRateOfInterest***()***+**" %"*)***;  
 ***}  
  
}***

Interfaces:

* Interfaces are used to achieve 100% abstraction.
* Interface looks like a class, but it is not a class. An interface can have methods and variables just like the class, but the **methods declared in interface are by default abstract** (only method signature, no body). Also, the **variables declared in an interface are public, static & final by default**.
* Interface cannot be declared as private, protected or transient.
* We can’t instantiate an interface in java.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.
* Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. More than one interface can be implemented in the class.
* **class implements an interface and interface extends interface.**
* A class cannot implement two interfaces that have methods with same name but different return type.
* Interface variables must be initialized at the time of declaration otherwise compiler will throw an error. Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final.

Interface Error

{

int x = 20; //else compilation error

}

Syntax:

Interface InterfaceName

{}

Example:

**package** OOPs;  
  
**interface** IndianBank  
***{*int** rateOfInterest***()***;  
***}*class** SBI **implements** IndianBank  
***{* public int** rateOfInterest***() {* return** 5;  
 ***}  
}*class** IOB **implements** IndianBank  
***{* public int** rateOfInterest***() {* return** 7;  
 ***}  
}*class** InterfaceBank  
***{* public static void** main***(***String***[]*** args***)  
 {*** IndianBank sbi = **new** SBI***()***;  
 System.***out***.println***(*"Rate of Interest for SBI: "** +sbi.rateOfInterest***())***;  
 IndianBank iob = **new** IOB***()***;  
 System.***out***.println***(*"Rate of Interest for IOB: "**+iob.rateOfInterest***())***;  
 ***}  
}***

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, multiple inheritance is not allowed, however you can use interface to make use of it as you can implement more than one interface.

Encapsulation/Data Hiding:

* Encapsulation allows us to protect the data stored in a class from system-wide access.
* Encapsulation can be implemented by keeping the fields (class variables) private and providing public getter and setter methods to each of them.
* If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.
* The fields can be made read-only (If we don’t define setter methods in the class) or write-only (If we don’t define the getter methods in the class). For e.g. If we have a variable that we don’t want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.

Rules:

* + Fields are set to private.
  + Each field has a getter and setter method.
  + Getter methods return the field.
  + Setter methods let us change the value of the field.

Example:

Account.java:

**package** OOPs.Encapsulation;  
  
**public class** Account  
***{*** *//private variables* **private int acc\_no**;  
 **private** String **name** ,**email**;  
 **private float amount**;  
*// Public Getter and Setter.* **public int** getAcc\_no***() {* return acc\_no**;  
 ***}* public void** setAcc\_no***(*int** acc\_no***) {* this**.**acc\_no** = acc\_no;  
 ***}* public** String getName***() {* return name**;  
 ***}* public void** setName***(***String name***) {* this**.**name** = **name**;  
 ***}* public** String getEmail***() {* return email**;  
 ***}* public void** setEmail***(***String email***) {* this**.**email** = email;  
 ***}* public float** getAmount***() {* return amount**;  
 ***}* public void** setAmount***(*float** amount***) {* this**.**amount** = amount;  
 ***}  
}***

AccountApplication.java:

**package** OOPs.Encapsulation;  
  
**public class** AccountApplication  
***{* public static void** main***(***String***[]*** args***) {*** *//creating instances of a class* Account acc = **new** Account***()***;  
 *//setting values to the setter method* acc.setAcc\_no***(***123456789***)***;  
 acc.setName***(*"SreeVidhya"*)***;  
 acc.setEmail***(*"sree.vidhya@myob.com"*)***;  
 acc.setAmount***(***65000***)***;  
 *//getting the values using getter methods.* System.***out***.println***(*"Name: "** + acc.getName***()***+ **"\n"** + **"Email: "** + acc.getEmail***()***+ **"\n"** + **"Account no.: "** + acc.getAcc\_no***()***+ **"\n"** + **"Amount: "** + acc.getAmount***()***+ **"\n"*)***;  
 ***}  
}***

Polymorphism:

* Polymorphism allows us to perform a single action in different ways. Define one interface and have multiple implementations.
* There are two types of polymorphism.
  + Run time/Dynamic Polymorphism.
  + Compile time/Static polymorphism.
* Polymorphism that is resolved during compiler time is known as static polymorphism.
* Method overloading is an example of compile time polymorphism.
* same method add () which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

Compile time polymorphism/Method Overloading :

Example:

**package** OOPs;  
 **class** SimpleCalculator  
 ***{* int** add***(*int** a, **int** b***)  
 {* return** a+b;  
 ***}* int** add***(*int** a, **int** b, **int** c***)  
 {* return** a+b+c;  
 ***}  
 }* public class** MthdOverLoading  
 ***{* public static void** main***(***String args***[])  
 {*** SimpleCalculator obj = **new** SimpleCalculator***()***;  
 System.***out***.println***(***obj.add***(***10, 20***))***;  
 System.***out***.println***(***obj.add***(***10, 20, 30***))***;  
 ***}  
 }***

## Runtime Polymorphism (or Dynamic polymorphism)/Method Overriding:

* Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that’s why it is called runtime polymorphism.
* A method is overridden, not the variables, so runtime polymorphism can't be achieved by variables.

Example:

**package** OOPs;  
  
**public class** RuntimePolymorphismDog  
***{* void** appearance***()  
 {*** System.***out***.println***(*"Looks like "*)***;  
 ***}  
}*class** Samoyed **extends** RuntimePolymorphismDog  
***{* void** appearance***()  
 {*** System.***out***.println***(*"White Fluff!! "*)***;  
 ***}  
}*class** Labrador **extends** RuntimePolymorphismDog  
***{* void** appearance***()  
 {*** System.***out***.println***(*"Chocolate Fluff!! "*)***;  
 ***}  
}*class** Looks***{* public static void** main***(***String***[]*** args***) {*** RuntimePolymorphismDog Summer = **new** Samoyed***()***;  
 Summer.appearance***()***;  
 Summer = **new** Labrador***()***;  
 Summer.appearance***()***;  
 ***}  
}***