**OBJECT-ORIENTED PRINCIPLES:**

**ABSTRACTION:**

1. A class which is declared as abstract is known as an **Abstract Class**.
2. 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.
3. Abstraction in java is achieved through,
   1. Abstract Class
   2. Interfaces.
4. 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.

**ABSTRACT CLASS:**

**RULES:**

1. 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).
2. The class itself cannot be made final but it can have final methods.
3. A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.
4. If class has at least one abstract method, the class **MUST** be declared abstract.
5. It can have abstract and non-abstract methods.
6. Abstract method is a method that is declared without an implementation. It must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well.
7. **SYNTAX FOR ABSTRACT METHOD:** 
   1. public abstract datatype methodName(params);
8. 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:**

1. Interfaces are used to achieve 100% abstraction.
2. Interface looks like a class, but it is not a class. An interface can have methods and variables just like the class, but:
   1. *methods declared in interface are by default abstract (only method signature, no body).*
   2. *Variables declared in an interface are public, static & final by default.*
3. Interface cannot be declared as private, protected or transient.
4. We can’t instantiate an interface in java.
5. By interface, we can support the functionality of multiple inheritance.
6. An interface which is declared inside another interface or class is called nested interface. They are also known as inner interface.
7. It can be used to achieve loose coupling.
8. Since methods in interfaces do not have body, they must be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface else those methods need to be declared abstract. More than one interface can be implemented in the class.
9. **class implements any number of interface and interface extends interface.**
10. A class cannot implement two interfaces that have methods with same name but different return type.
11. All the interface methods are by default **abstract and public**.
12. While providing implementation in class of any method of an interface, it needs to be mentioned as public.
13. 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*.

**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***())***;  
 ***}  
}***

# **DIFFERENCE BETWEEN ABSTRACT CLASS AND INTERFACE:**

|  |  |  |
| --- | --- | --- |
|  | **Abstract Class** | **Interface** |
| 1 | An abstract class can extend only one class or one abstract class at a time | An interface can extend any number of interfaces at a time |
| 2 | An abstract class can extend another concrete (regular) class or abstract class | An interface can only extend another interface |
| 3 | An abstract class can have both abstract and concrete methods | An interface can have only abstract methods |
| 4 | In abstract class keyword “abstract” is mandatory to declare a method as an abstract | In an interface keyword “abstract” is optional to declare a method as an abstract |
| 5 | An abstract class can have protected and public abstract methods | An interface can have only public abstract methods |
| 6 | An abstract class can have static, final or static final variable with an access specifier | interface can only have public static final (constant) variable |

**ADVANTAGES OF USING INTERFACES ARE AS FOLLOWS:**

1. Without bothering about the implementation part, we can achieve the security of implementation
2. 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.
* There are two types of polymorphism.
  + *Run time / Dynamic-Method Overriding.*
  + *Compile time/Static-Method Overloading.*
* 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:**

* Polymorphism that is resolved during compiler time is known as static polymorphism.
* **Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

**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.
* If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding**
* A method is overridden, not the variables, so runtime polymorphism can't be achieved by variables.
* When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.
* A static method cannot be overridden.

**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***()***;  
 ***}  
}***

|  |  |  |
| --- | --- | --- |
| No. | Method Overloading | Method Overriding |
| 1) | Method overloading is used to increase the readability of the program. | Method overriding is used to provide the specific implementation of the method that is already provided by its super class. |
| 2) | Method overloading is performed within class. | Method overriding occurs in two classes that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, parameter must be different. | In case of method overriding, parameter must be same. |
| 4) | Method overloading is the example of compile time polymorphism. | Method overriding is the example of run time polymorphism. |
| 5) | Method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But parameter must be changed. | Return type must be same in method overriding. |

**INHERITANCE:**

* The process by which one class acquires the properties (instance variables) and functionalities(methods) of another class is called **inheritance**.
* For Method Overriding (so runtime polymorphism can be achieved).
* The aim of inheritance is to provide the reusability of code so that a class must write only the unique features and rest of the common properties and functionalities can be extended from another class.
* The class that extends the features of another class is known as child class, sub class or derived class.
* The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.
* **Syntax:**

class ChildClass extends ParentClass

**{ }**

**TYPES OF INHERITANCE:**

* *Single Inheritance*
* *Multilevel Inheritance*
* *Hierarchical Inheritance*

**Single Inheritance:**

When a class extends from another one class it a single inheritance.

**package** OOPs.Inheritance;  
  
 **class** Animal***{* void** speak***(){***System.***out***.println***(*"Can vary..."*)***;***}  
 }* class** Dog **extends** Animal***{* void** speak***(){***System.***out***.println***(*"barking..."*)***;***}  
 }*public class** SingleInheritance  
***{* public static void** main***(***String args***[])  
 {*** Dog d=**new** Dog***()***; *// Dog has inherited from animal and this is method overriding* d.speak***()***;  
 Animal a = **new** Animal***()***;  
 a.speak***()***;  
 ***}  
}***

**Multilevel Inheritance:**

One class can inherit from a derived class, thereby making this derived class the base class for the new class.

**package** OOPs.Inheritance;  
  
 **class** Father***{* void** transport***(){***System.***out***.println***(*"Mode of transport: walking"*)***;***}  
 }* class** Child **extends** Father***{* void** transport***(){***System.***out***.println***(*"Mode of transport: Car"*)***;***}  
 }* class** grandChild **extends** Child***{* void** fuel***(){*** System.***out***.println***(*"They can be solar powered as well"*)***;  
 ***}  
 }*public class** MultiLevelInheritance  
***{* public static void** main***(***String args***[]){*** grandChild g=**new** grandChild***()***;  
 g.transport***()***;  
 g.fuel***()***;  
 ***}  
}***

**Hierarchical Inheritance:**

A child and parent class relationship where more than one classes extends the same class.

**package** OOPs.Inheritance;  
  
**class** Mother***{* void** eat***(){***System.***out***.println***(*"Allergic to gluten"*)***;***}  
 }* class** Sibling1 **extends** Mother***{* void** allergy1***(){***System.***out***.println***(*"Allergic to Lactose"*)***;***}  
 }* class** Sibling2 **extends** Mother***{* void** allergy2***(){***System.***out***.println***(*"Allergic to nuts"*)***;***}  
 }*public class** HierarchialInheritance  
***{* public static void** main***(***String args***[]){*** Sibling1 c=**new** Sibling1***()***;  
 c.allergy1***()***;  
 c.eat***()***;  
*// c.allergy2(); error.* ***}  
}***

**Association:**

Association the act of establishing a relationship between two unrelated classes.

* Two separate classes are associated through their objects.
* The two classes are unrelated, each can exist without the other one.
* Can be a one-to-one, one-to-many, many-to-one, or many-to-many relationship.

**package** OOPs.Association;  
  
**class** Pizza  
***{*** String **pizzaName**;  
 String **base**;  
 String **ingredient1**;  
 String **ingredient2**;  
 String **ingredient3**;  
 Pizza***(***String pizzaName,String base,String ingredient1,String ingredient2,String ingredient3***)  
 {* this**.**pizzaName** = pizzaName;  
 **this**.**base** = base;  
 **this**.**ingredient1** = ingredient1;  
 **this**.**ingredient2** = ingredient2;  
 **this**.**ingredient3** = ingredient3;  
 ***}  
}*class** Person **extends** Pizza  
***{*** String **personName**;  
 Person***(***String personName, String pizzaName, String base, String ingredient1, String ingredient2, String ingredient3***) {* super*(***pizzaName, base, ingredient1, ingredient2, ingredient3***)***;  
 **this**.**personName** = personName;  
 ***}  
  
}*class** Customer  
***{* public static void** main***(***String***[]*** args***) {*** Person customer = **new** Person***(*"Andy"**,**"Margharita"**,**"Tomato Sauce"**,**"Cheese"**,**"Herbs"**,**"Garlic Seasoning"*)***;  
 System.***out***.println***(***customer.**pizzaName** + **" with Base "** + customer.**base** + **" and the ingredients are: "** + customer.**ingredient1** + **", "**+ customer.**ingredient2** + **" and "**+ customer.**ingredient3** + **" for "**+customer.**personName*)***;  
 ***}  
}***

**Aggregation:**

Association is a special form of association which is a unidirectional one-way relationship between classes. for e.g. Student and Subject classes. Student has a subject, but subject doesn’t need to have student necessarily so it’s a one directional relationship. In this relationship both the entries can survive if other one ends. In our example if subject class is not present, it does not mean that the Student class cannot exist.

**package** OOPs;  
  
**class** Subject  
***{*** String **subjectName**;  
  
 Subject***(***String sN***)  
 {* this**.**subjectName** = sN;  
 ***}  
}*class** Student  
***{*** Subject **studentSubject**;  
 **int rollNum**;  
 String **name**;  
 String **Address**;  
 **int phNum**;  
  
 **public** Student***(***Subject ss,**int** sNo, String name, String address, **int** phNum***) {* this**.**studentSubject** = ss;  
 **this**.**rollNum** = sNo;  
 **this**.**name** = name;  
 **Address** = address;  
 **this**.**phNum** = phNum;  
 ***}  
}*public class** AggregationStudent  
***{* public static void** main***(***String***[]*** args***) {*** Subject subject1 = **new** Subject***(*"Computer Science"*)***;  
 Student student1 = **new** Student***(***subject1,1,**"Andy"**,**"SouthYarra"**,1234567***)***;  
 System.***out***.println***(*"Name: "**+student1.**name*)***;  
 System.***out***.println***(*"Roll number: "**+student1.**rollNum*)***;  
 System.***out***.println***(*"Address: "**+student1.**Address*)***;  
 System.***out***.println***(*"Ph: "**+student1.**phNum*)***;  
 System.***out***.println***(*"Subject Enrolled: "**+student1.**studentSubject**.**subjectName*)***;  
 ***}  
}***

**Composition:**

Composition is a restricted form of Aggregation in which two classes are highly dependent on each other. For e.g. Human Life and Heart. A human needs heart to live and a heart needs a Human body to survive.

**package** OOPs.Association;  
  
 **class** Job ***{* private** String **role**;  
 **private long salary**;  
 **private int id**;  
  
 **public** String getRole***() {* return role**;  
 ***}* public void** setRole***(***String role***) {* this**.**role** = role;  
 ***}* public long** getSalary***() {* return salary**;  
 ***}* public void** setSalary***(*long** salary***) {* this**.**salary** = salary;  
 ***}* public int** getId***() {* return id**;  
 ***}* public void** setId***(*int** id***) {* this**.**id** = id;  
 ***}  
 }* class** People ***{* private** Job **job**;  
  
 **public** People***(){* this**.**job**=**new** Job***()***;  
 **job**.setSalary***(***1000L***)***;  
 **job**.setId***(***00001***)***;  
 **job**.setRole***(*"Assistant"*)***;  
 ***}* public long** getSalary***() {* return job**.getSalary***()***;  
 ***}* public** String getRole***() {* return job**.getRole***()***;  
 ***}* public int** getId***() {* return job**.getId***()***;  
 ***}  
  
 }* class** Emp ***{* public static void** main***(***String***[]*** args***) {*** Job jobs = **new** Job***()***;  
 People person = **new** People***()***;  
 **long** salary = person.getSalary***()***;  
 System.***out***.println***(***person.getSalary***())***;  
 System.***out***.println***(***person.getId***())***;  
 System.***out***.println***(***person.getRole***())***;  
  
 ***}  
 }***