**OOPS**

**Concepts**

**Object:**

1. **Any entity that has state and behavior is known as an object.**
2. **For example, a chair, pen, table, keyboard, bike, etc.**
3. **It can be physical or logical.**
4. **An Object can be defined as an instance of a class.**
5. **An object contains an address and takes up some space in memory.**
6. **Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.**

**Class**

1. ***Collection of objects* is called class.**
2. **A class can also be defined as a blueprint from which you can create an individual object.**
3. **It is a logical entity.**
4. **Class doesn't consume any space.**

Module 1

1. Data hiding
2. Abstraction
3. Encapsulation
4. Tightly encapsulated class

Module 2

1. Is A relationship
2. HAS a relationship
3. Method signature
4. Overloading
5. Overriding
6. Static control flow
7. Instance control flow
8. Constructors
9. Coupling
10. Cohesion
11. Type casting
12. Association
13. composition

**1.Data hiding**

* It is simply hiding of our data
* Outside person cant access our internal data directly
* Our internal data should not go our directly after performed authentication or validations right person only can see the data
* E.g.: Gmail inbox
* By declaring data variable as a private we can achieve data hiding
* Advantage: security
* Recommended modifier for variable is private

**2.Abstraction**

* *Hiding internal details and showing functionality* is known as abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)

* A class which is declared as abstract is known as an **abstract class**.
* It can have abstract and non-abstract methods.
* It needs to be extended and its method implemented
* It cannot be instantiated. (cannot create obj)
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

EG: **abstract** **class** A{}

**Abstract method:**

A method which is declared as abstract and does not have implementation is known as an abstract method.

Simple EG:

**abstract** **void** printStatus();//no method body and abstract

**abstract** **class** Bike{

**abstract** **void** run();

}

**class** Honda4 **extends** Bike{

**void** run(){System.out.println("running safely");}

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

Bike obj = **new** Honda4();

obj.run();

}

}

**An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.**

1. //Example of an abstract class that has abstract and non-abstract methods
2. **abstract** **class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract** **void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a C hild class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public** **static** **void** main(String args[]){
14. Bike obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }
19. Interface (100%)

Advantages:

* security - outside person doesn't aware our internal implementation
* Enhancement is easy - we can change the language
* Easiness to use our system - no need to know about internal implementation
* Maintainability become easy - can change implementation without any gui affect

Implementation :

* We can implement interface
* We can achieve multiple inheritance by implementing multiple interface
* Interface can have public, static, abstract ,default, methods
* Interface can have public static final variables

**3.Encapsulation**

* Process of binding data members and corresponding behavior into a single unit is encapsulation.
* The whole idea behind encapsulation is to hide the implementation details from users.
* If a data member is private it means it can only be accessed within the same class. No outside class can not access private data members (variables) of other classes.
* If any component follows data hiding and abstraction and its is encapsulated component
* Eg: java class
* How to implement encapsulation in java:

1) Make the instance variables private so that they cannot be accessed directly from outside class. You can only set and get values of these variables through the methods of the class.

2) Have getter and setter methods in the class to set and get the values of the fields.

* Main advantage is we can achieve security, disadvantage is it increases length of the code and slows down execution

### Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

**4.tightly encapsulated class**

* Each and every variable of the class is declared as a private then the class is a tightly encapsulated class.
* Advantage is security
* Eg: java bean class

**Module 2**

**1.IS-A Relationship `**

* It is also known as inheritance
* One class acquires the property of another class / parent class is known as inheritance.
* Implement by using extends keywords
* Advantage: reusability of parent class property
* Parent class members by default available to child class
* Child methods are not available to parents by default. P p = new P(), p.m2()
* Child reference can call parent method, it is available by default
* Parent reference used to hold child obj, using this obj we can call parent method but not child method.
* Child reference not used to hold parent object
* Object is the root of all java classes
* Every class in java is a child class of object class by either directly or indirectly. So, object class methods by default available to all java class
* If our class does not extend any other class, then our class is direct child class of object
* If our class extends any other class, then our class is indirect child class of object
* There may be a chance of ambiguity problem, hence Java won't provide support for multiple inheritance.
* If both p1, p2 contain the m1 method, in a child object if we called m1 method which one will call.
* So, java provide support for multiple inheritance with respect to interfaces
* Cyclic inheritance is class a extends a which is not allowed and not required in java, it throws error like cyclic inheritance involved

**2. Association**

* Association is a relation between two separate classes which establishes through their Objects.
* In one class use the functionality provided by another class.
* Association can be one-to-one, one-to-many, many-to-one, many-to-many
* **Composition** and **Aggregation** are the two forms of association.

**Ref: https://www.geeksforgeeks.org/association-composition-aggregation-java/**

**3.Has-A Relationship**

* This is most used relationship, which is special form of association
* It is unidirectional association , which is car has a engine but vise versa not true.
* Eg: Engine e = new Engine (); here Engine is a Reference. Means car has a engine reference.
* It is also known as composition or Aggregation
* There is no specific keyword to implement has a relationship
* If it is there instance method create an object and make the functionality available
* If it is static method object also not required
* But most of the time we are depending on new keyword.
* The biggest advantage is reusability only

**4. Composition**

* University has a cse department
* Here university is a container object and department is contained object
* Without existing unive rsity (container) object there is no chance of existing department (contained) object.
* **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.
* Which means container and contained objects are strongly associated. This is called composition
* In composition, both entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist** without the other entity.

Eg:

**5.Aggregation**

* It represents Has-A’s relationship.it is special form of association
* It is a **unidirectional association** i.e. a one-way relationship.
* For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
* IT is weekly associated.
* In Aggregation,**both entries can survive individually** which means ending one entity will not affect the other entity.

|  |  |
| --- | --- |
| Composition | Aggregation |
| Objects are strongly associated | Objects are weekly aggregated |
| Destroy of one class impact the other one.  Eg: University has Department – if university deleted, dept also get impacted | Destroy the one class will not impact other,  Eg: Library has students, if library closed stdednts still there |

**6.Method Signature:**

* **Method signature** contains only method name followed by argument types.
* here returns type is not part of method signature
* eg : public static m1(int j, float f)
* compiler will used method signature to resolve method calls
* compiler will have method table for each class which consist of all method for each class
* within a class 2 methods with same signature not allowed. Eg :void m1(int j) {},int m1(int j){return 10}, CE : m1 (int ) already defined

**7. Polymorphism:**

* **Polymorphism in Java is a concept by which we can perform a single task in multiple ways**
* **There are two types of polymorphism in Java**
  + - 1. **compile-time polymorphism / static polymorphism /early binding**

**Eg : method overloading , method hiding**

* + - 1. **runtime polymorphism / dynamic polymorphism / late binding**

**8.Overloading:**

* two methods are said to be overloaded, if and only if both methods having same name but different argument types
* but in java we can create multiple method with same name and different arguments, these methods are called overloaded method
* eg: abs (int i), abs (long l), abs (float f)
* Having an overloading concept in java reduces complexity of programming.
* In overloading, method resolution always takes by compiler based on reference type, hence overloading also consider as compile time polymorphism or static polymorphism or early binding
* Method resolution means, which method has to execute
* In overloading method resolution compiler will always check exact match, if the match is not their compiler will promote the next level and will check if there any matched method until maximum level
* For ex : int method only there but we are passing char as a argument then the compiler promotes the char to int and executes the int method.
* Hence overloading has automatic promotion while resolving overloaded methods., if exact matched methods are not available then we won’t get compile time error immediately, first it will promote argument to the next level and checks whether matched method available or not, if it is available then It will be consider, if it is not available then compiler promotes argument once again to the next level. This process will be continued until all possible promotion. Still a matched method not available then we will get a CE error. This process is called automatic promotion in overloading
* Compiler always prefer the exact match if it is not found will promote automatically
* Assume we have two overloaded methods void m1(String s){}, void m1(Object o){}. If we are calling t.m1(“string”); string method will be called. t.m1(Object), object method will be called.
* t.m1(null), now it is matched for both object and string method. the object method is parent and the string method is child. If we have both child and parent, those are matched then the child gets preference.
* If we have another child method stringBuffer , now both string and stringBuffer is same level so compiler will throw error like reference to m1 is ambiguity
* Consider we have two method m2(int I, float f) {}, m2(float f, inti) {}
* We are calling t.m2(10,10) complier will throw reference m2() is ambiguous
* We are calling the method t.m2(10.5,10.5) we can’t de promote float to int , so both methods are not matched, so compiler will throw error “Cannot find symbol. Method (float, float) location: class test

Note:

Abs is absolute value not specific to sign.eg: abs (10.5) is 10.5, abs (-10.5) also 10.5

**9.Overriding:**

* Whatever method parent has by default is available to the child through inheritance. If the child class is not satisfied with parent class implementation then, the child is allowed to redefine that method based on its requirement. This process is called overriding
* The parent class method which is overridden is called overridden method and the child class method which is overriding is called overriding method.

EG: class P {

Public void property (){

}

Public void marry (){

}

}

Class C extends P {

Public void marry() {

}

}

Class Test{

p.sv.m(string arg){

//1)

P p= new P();

p.marry() // parent method called

//2)

C c=new C();

c.marry();//child method called

//3)

P p1= new C();//parent reference can hold child object

p1.marry ();

//compiler will check in parent class marry method is available, compiles fine

//at run time JVM will check whether it is parent object or child object

//if it is child class (parent reference child obj) jvm will check whether this method //overriding in child class or not. If it is not overriding, then there is no issue parent //method will execute. If it is overriding, then based on the run time object JVM will execute the child class method.

}

}

In Overriding, method resolution always based on run time object.so it also known as runtime polymorphism or dynamic polymorphism or late binding

**Rules for Overriding:**

* Method signature (method name and arguments types) must be same
* Return type must be the same. This rule applicable upto 1.4 v only. From 1.5v onwards covariant return type is also allowed.

Co-variant return type:

Child class method return type need not to be the same as parent class method, return type (String) can be child of parent method return type(Object).

If parent is string and child is Object then it throws error

This co-variant is applicable for non-primitive only, not primitive data types

Eg: parent double and child int is not applicable

* Parent class private method not available to the child and hence overriding concept not applicable for private methods
* based on our requirement we can define exactly same private method in child class. It is valid but not overriding.
* We cannot override final method
* Parent class Abstract methods we should override in child class to provide implementation
* We can override non abstract method as abstract method, this child abstract method should be implemented by corresponding child, at this stage we are forcing to not override the parent method
* If parent class is non final method, then we can override as final method then corresponding child wont override
* Synchronized and non-synchronized, native and nonnative, strictfp and non strictfp won’t play any restriction in overriding
* While over ridding we can not reduce the scope of access modifier (return type) but we can increase

**10.Method Hiding**

* Parent class method is static and child class method also static it is not over riding and it is method hiding.

Eg:

Class A

public static void m1(){

s.o.p(“====test===in A===”);

}

}

Class B extends A

Public static void m1(){

s.o.p(“====test===in B===”);

}

}

Class Test{

p.s.v.m(String s[]){

P p = new P();

**p.m1(); //**

**//parent method will called**

**C c= new C();**

**c.m1();**

**//child method called**

**P p1= new C();**

**//in overriding child method will be called, bcz method resolution always takes care by jvm based on run time object.**

**//in method hiding bcz method resolution always takes care by compiler // based on reference type**

}

}

|  |  |
| --- | --- |
| Method hiding | Overriding |
| Both parent & child methods should be static | Both methods should be not static |
| Compiler is responsible for method resolution based on reference type | JVM is always responsible for method resolution based on run time object |
| Also known as compile time polymorphism, early binding or static polymorphism | It is also known as run time polymorphism or dynamic polymorphism or late binding |

* In Over riding – old data not there, new data only there. In method Hiding – we are hiding data. Both copies of data available.