### OOP - Encapsulation

**Encapsulation** is process of wrapping of data and method into single unit.  
  
A Java class is example of encapsulation.  
Java Bean is **fully encapsulated class** because all the data members are private here.  
  
Encapsulation provides two features.  
1. Data Hiding  
2. Abstraction  
  
**Data Hiding**  
**Outside person does not access** the data directly by applying member variable as private.  
  
**Abstraction**  
- **Hides the internal implementation**.  
- Outside person **does not aware about** the internal implementation.  
- Shows the only functionality to the user.

Program to interfaces, not implementations is the **principle for Abstraction**.  
**Abstraction** provides the general structure of a class and leave the details for the implementer.  
**Encapsulation** is **to create and define the permissions and restrictions of an object** and its member variables and methods.

**Abstraction** means hiding the implementation complexities by using interfaces and abstract class.  
**Encapsulation** means hiding data by using setters and getters.  
  
**Abstraction** is implemented by using interfaces and abstract  classes.  
**Encapsulation** is implemented by using Access Modifiers (private, default, protected, public).

[**Encapsulation**](http://contribute.geeksforgeeks.org/encapsulation-in-java/) is **data hiding** (**information hiding**) while **Abstraction** is **detail hiding (implementation hiding**).

While **encapsulation** groups together data and methods that act upon the data, **data abstraction** deals with **exposing the interface to the user** and **hiding the details of implementation**.

In Java, abstraction is achieved through interfaces and abstract classes.  
  
**Encapsulation** = Data Hiding + Abstraction

### OOP - Inheritance

Inheritance is **process of obtaining the data members and methods** from one class to another class.  
For example, Class B extends Class A. Class B inherits data members and method of the Class A.  
  
Inheritance expresses the "is a" relationship between two objects.

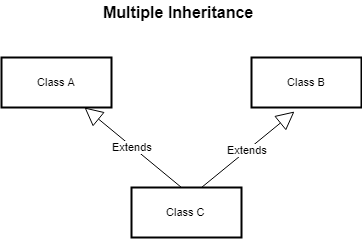
Inheritance – IS-A relationship between super class and its subclasses. The process where one object acquires the members of another, plus can have its own.

**Example**

Car (Subclass) is a type of Vehicle (Super class). So, Car can inherit(reuse) the members of Vehicle class; plus it can have its own behavior and properties.   
  
Whenever method is available in the parent class, it will be automatically available in the child class. We are not going to rewrite in the child class. it is called as re-usability.  
  
Parent class - Base class      - Super class  
Child class    - Derived class - Sub class  
  
**Different types of Inheritance**  
**1. Single inheritance**  
Class B extends A {}  
**2. Multiple inheritance**

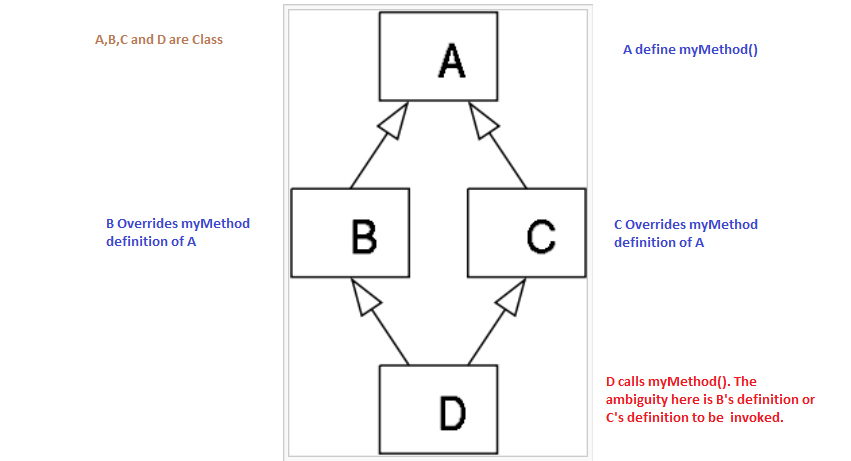
Class A extends B,C {}

A child class inheriting states and behavior from multiple parent classes is known as Multiple Inheritance.

**[](https://1.bp.blogspot.com/-IMp3IBjixqg/W-lY8OGdpkI/AAAAAAAAEpQ/jDqK9SIlJU0AqxHiGCp2DVNXOhMDmQXBQCLcBGAs/s1600/multiple%2Binheritance.png)**

**Diamond problem** refers to an **ambiguity** that occurs due to **allowing multiple inheritance**.

In case of Multiple Inheritance, suppose Class A has two sub classes B and C, and class D has two super class B and C. If a method present in A is overridden by both B and C but not by D, then from which class D will inherit that method from class B or C? This problem is known as Diamond Problem.



**Java does not support multiple inheritance that introduces complexities like the diamond problem.**

In Java, multiple inheritance is not allowed for classes and permitted only for interfaces to eliminates this serious issue.  
         It is not supported by Java

**3. Multilevel inheritance**  
        Class C extends B {}  
        Class B extends A {}  
**4. Hierarchical inheritance**  
        Class A extends C {}  
        Class B extends C {}  
**5. Hybrid inheritance**  
        It is not supported by Java.  
        It is combination of multi-level, hierarchical and multiple inheritance.  
**6. Cyclic inheritance**  
        It is not supported by Java. It is not really required.  
        Class A extend B {} class B extends A {}  
If the class does not extends any class, it internally extends java.lang.Object.  
  
Parent class maintains the common methods.  
Child class maintains the specific methods.  
  
**Parent reference** **can hold child reference**.  
**Child reference cannot hold parent reference.**

|  |  |
| --- | --- |
| Class P {  public void m1 () {  System.out.println(“Parent”);  }  } | Class C extends P {  public void m2 () {  System.out.println(“Child”);  } } |
| P p = new P ();  p.m1();  p.m2(); // Compile time Error | P p = new C ():  p.m1();  p.m2(); // compile time error |
| C c = new C ();  c.m1();  c.m2(); // no Issues | C c = new P ();  // Compile time error |

**OOP - Polymorphism**

Polymorphism means one name many forms.

It is further divided into two types.

1. Static Polymorphism 2. Dynamic Polymorphism

Static Polymorphism is achieved using method overloading.

Dynamic polymorphism is achieved using method overriding.

Static Polymorphism – Compile time Polymorphism – Method Overloading – Early Binding

Dynamic Polymorphism – Run time Polymorphism – Method Overridden – Late Binding

**Method Overloading**

- Same method name can be used more than one time in the same class with different parameter.

- Method resolutions are taken care by compiler-based reference type.

- Two method with same signature is not allowed in Java class.

**Method Overriding**

**-**Same method name and parameter can be overridden by any one of sub classes.

- Method resolution is taken care by JVM based on the run time object.

-  A call to overridden method is resolved at run time rather that at compile time.

- JVM checks run time object whether it is overriding method.  
 **Method Overriding Rule**

**Return type** – Co variant return types are allowed to override.

   Object to String - Allowed

   String to Object – Not Allowed

**Overriding method**

- Private method can be overridden in Child class. Private method in Parent class is not visible to Child class.

- Private method can be overridden by default, protected or public method.

- Final method cannot be overridden. It will give compile time error.

- Final method cannot be overridden by any method including final method.

- Non final method can be overridden by final method.

- Static method can be overridden by static methods.

- Static method cannot be overridden by non-static methods.

- non-static method can be overridden by non-static methods.

- non-static method cannot be overridden by static methods.

**Least Scope**

  Private < default < protected < public

- Private method can be overridden by default, protected and public.

- Public method can be overridden by Public method and not by others

**Exception**

* Overridden rule is applicable to checked exception. No rule for Unchecked Exception.
* If child class throws the checked exception, Parent class must throw checked exception or its parent. If does not fulfils, it will **throw the compile time exception**.
* we can remove throws clause from overriding method in sub class irrespective of whether super class’s overridden method throws any checked or un-checked exception.

**Overriding with static method**

Both the methods are static. Based on reference type, it will resolve. It is called as**Method Hiding**

**class** Parent {

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

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

}

}

**public** **class** Child **extends** Parent{

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

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

}

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

*m2*(); //Resolved in same class

Child c = **new** Child();

c.*m2*(); //child

Parent p = **new** Child();

p.*m2*(); //parent

// Based on reference type, it will resolve the method. it is called method hiding

System.***out***.println("Main method");

Child.m2(); //child

Parent.m2(); //parent

}

}

**Let us think more with the following points on static overriding.**

1. As static methods cannot be inherited, no question of overriding.
2. Static methods are treated as class methods (like static variables are known as class variables) or class level methods. So, they cannot be inherited and overridden.
3. Static methods are part of class but not part of object. As a part static methods can be overloaded but cannot be overridden.
4. Overridden methods support dynamic binding and which method is to be dispatched for execution is decided at runtime. As static methods are binded at compile time (known as static binding), they cannot support dynamic binding.

Let us perform a single action in different ways.  
Polymorphism means different forms.

|  |  |  |
| --- | --- | --- |
| **Property** | **Overloading** | **Overriding** |
| Method name | Must be same | Must be same |
| Argument type | Must be different | Must be same |
| Private/final/static method | Can be overloaded | Cannot be overridden |
| Return type | No Restriction | Co-Variants return types are allowed. |
| Throws Clause | No Restriction | Restriction in Checked Exception.   If the child class method throws any checked Exception, Parent class method throws same checked exception or its parent.  No Restriction in unchecked Exception |
| Method Resolution | Compiler based on reference type | JVM based return type |
| Other names | Compile time Polymorphism, Static Binding, Early Binding | Run time polymorphism, Late Binding |

**OOP - Association**

Association represents a **relationship between two or more objects** **where all objects have their own life cycle** and there is no owner.

The name of association specifies the nature of relationship between the objects.

Association can be one-to-one, one-to-many, many-to-one, many-to-many

**Aggregation**

- Aggregation is **a special form of association** where

o   It represents **Has a relationship**. E.g. A has B

o   It is unidirectional association

-      Aggregation **represents a part of whole relationship** where a part can exist without a whole. It has a **weaker relationship**.  
       
 - In Aggregation, **Both the entries can survive individually** which means **ending one entity will not affect other** **entity**.  
      
Let’s take an example of **the relationship between Department and Teacher**. A Teacher may belong to multiple departments. Hence **Teacher is a part of multiple departments**. But if we delete a Department, Teacher Object will not destroy.

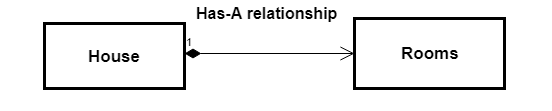
**[](https://4.bp.blogspot.com/-uoFDQg8WAmY/W-lZgqWhufI/AAAAAAAAEpk/Tf5xnvkOiQMeATrKYx0lH-HxvITzsTISACLcBGAs/s1600/aggregation%2B%25281%2529.png)**

**Composition**  
Composition is a **restriction form of aggregation** in which two entities are highly dependent on each other.

-   Composition is an **association represents a part of whole relationship** where a part cannot exist without a whole. **If a whole is deleted, then all parts are deleted. It has stronger relationship.**  
  
Composition **represents the part of relationship** e.g. **B is part of A.**   
Example- Human and Heart. Heart does not exist without human. Human does not exist without heart.

**In composition, both the entities are dependent on each other.**

House is a whole, and Rooms are parts. If a house is deleted, then all corresponding rooms for that house should be deleted.



|  |  |
| --- | --- |
| Is a Relationship | Inheritance |
| Has a Relationship | Aggregation |
| Part of Relationship | Composition |

**Cohesion**

Cohesion is **a measure of how the methods of a class or module are meaningfully and strongly related** and how focused they are in providing well defined purpose to the system.  
  
Cohesion is used to **indicate the degree to which a class has a single well focused responsibility**.  
  
**Two types of cohesive**  
1. Low Cohesive  
2. High Cohesive  
  
**Low Cohesive**   
- A class is identified as a low cohesive class **when it contains many unrelated method or function within it.**  
- We need to avoid big classes **with unrelated functions hamper their maintaining.**  
 **High Cohesive**  
- The code **must be very specific in its operations**.  
- **The responsibilities /methods are highly related to the class/module**

**Coupling**

**Coupling** refers to **the degree to which one class knows about another class.**  
**If one class uses another class, that is called as coupling.**  
  
**There are two types of coupling**  
1. Tightly Coupled   
2. Loosely Coupled  
  
**Tightly Coupled**

**A class uses another class directly.**

**Classes are dependent on one another. It reduces the flexibility and reusability of code.**If one class is tightly coupled with other class  
 **Loosely Coupled**  
- **Low dependencies between the classes.**  
- There should not be too much dependency between the classes even if there is a dependency, it should via the interfaces and should be minimal.

- **Avoid tightly coupling for collaboration between two classes.**   
Example. Dependency Injection or Inversion of control

|  |  |
| --- | --- |
| **Tightly Coupled** | **Loosely Coupled** |
| Public class **Car** {     Public void move () {      System.out.println(“car is moving”);  } }  Class **Traveller** {  Car c = new Car (); // Tightly Coupled.  Public void startJourney () {          c.move();  }} | Interface vehicle {      Public void move ();  }  Class Car implements vehicle {   Public void move () {     System.out.println(“car is moving”);  } }  Class Bike implements vehicle {   Public void move () {     System.out.println(“Bike is moving”);  } }  Class Traveller {  Private vehicle vehicle;  Public vehicle getVehicle () {       Return vehicle;  }  Public void setVehicle (Vehicle vehicle) {    this.vehicle = vehicle;  }  Public void startJourney (){        vehicle.move ();  }  } |