Aggregation

A class has an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation; Employee object contains information such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country etc. as given below.

EX:

```
class Employee{
int id;
String name;
Address address; //Address is a class (Aggregation)

...
}
```

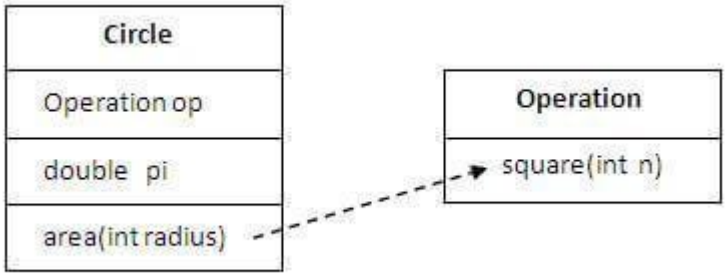
In such case, Employee has an **entity reference** address, so relationship is

Employee HAS-A address.

Why use Aggregation? Ans :For Code Reusability.

Simple Example of Aggregation

In this example, we have created the reference of Operation class in the Circle class.




```
class Operation{
 int square(int n){
  return n*n;
 }
}
class Circle{
```

```
 Operation op;  //aggregation
 double  pi=3.14;
 double area(int radius){
   op=new Operation();
   return pi* op.square(radius);  //   code reusability
 }
public static void main(String args[]){
   Circle c=new Circle();
   System.out.println(c.area(5));
 }
}
```

When use Aggregation?

Code reuse is also best achieved by aggregation when there is no is-a relationship.

Understanding meaningful example of Aggregation

In this example, Employee has an object of **Address**, address object contains its own information such as city, state, country etc. In such case relationship is Employee **HAS-A** address.

```
public class Address {
String city, state,country;
public Address(String city, String state, String country) {
    this.city = city;
    this.state = state;
    this.country = country;
}
}
```

```
public class Emp {
int id;
String name;
Address address;
public Emp(int id, String name, Address address) {
   this.id = id;
   this.name = name;
   this.address=address;
}
void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}
public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);
e.display();
e2.display();
}
}
```

Output:111 varun
gzb UP india
112 arun
gno UP india

Enumerations :

An enum (**Enumerations**) is a special "class" that represents a group of constants (**unchangeable** variables, like final variables).

To create an enum, use the **enum** keyword (**instead of class or interface**), and separate the constants with a comma. **Note that they should be in uppercase letters**:

Example

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
```

You can access enum constants with the dot syntax:

```
Level myVar = Level.MEDIUM;
```

Note: Enum is short for "**enumerations**", which means "specifically listed".

Enum inside a Class

You can also have an enum inside a class:

Example

```
public class Main {
  enum Level {
    LOW,
    MEDIUM,
    HIGH
  }
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
}
```

The output will be:

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY,
WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY}
public static void main(String args[]){
Day day=Day.MONDAY;

switch(day){
case SUNDAY:
 System.out.println("sunday");
 break;
case MONDAY:
 System.out.println("monday");
 break;
default:
System.out.println("other day");
}
}}
```

MEDIUM

Difference between Enums and Classes

- An **enum** can, just like a class, have attributes and methods. The only difference is that **enum** constants are **public, static and final (unchangeable - cannot be overridden)**.
- An enum cannot be used to create objects, and it **cannot extend other classes** (**but it can implement interfaces**).

Why And When To Use Enums?

Use enums **when you have values that you know aren't going to change**, like month days, days, colors etc.

Method Overriding

If subclass (child class) has the **same method** as declared in the parent class, it is known as method **overriding** in Java.

In other words, if a subclass provides the **specific implementation** of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for **runtime polymorphism.**

Rules for Java Method Overriding

- The method must have the **same name** as in the parent class
- The method must have the **same parameter** as in the parent class.
- There must be an **IS-A relationship** (inheritance).

Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike extends Vehicle{
  public static void main(String args[]){
    Bike obj = new Bike();
    obj.run();
  }
}
```

Output is Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is **IS-A relationship** between the classes, so there is method overriding.

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
  void run(){System.out.println("Bike is running safely");}
    public static void main(String args[]){
```

```
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

Bike is running safely

```
class Bank{
int getRateOfInterest(){return 0;}
}
class Bank1 extends Bank{
int getRateOfInterest(){return 8;}
}
  class Bank2 extends Bank{
int getRateOfInterest(){return 7;}
}
class Bank3 extends Bank{
int getRateOfInterest(){return 9;}
}
class Test2{
public static void main(String args[]){
Bank1 s=new Bank1 ();
Bank2 i=new Bank2 ();
Bank3 a=new Bank3 ();
System.out.println("Bank1 Rate of Interest: "+s.getRateOfInterest());
System.out.println("Bank2 Rate of Interest: "+i.getRateOfInterest());
System.out.println("Bank3 Rate of Interest: "+a.getRateOfInterest());
```

Output:

Bank1 Rate of Interest: 8

Bank2 Rate of Interest: 7

Bank3 Rate of Interest: 9

}

}

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Can we override java main method?

No, because the main is a static method.

What are the Difference between method overloading and method overriding?

Abstraction

Abstract Class is a type of class in OOPs that declare one or more abstract methods. These classes can have **abstract methods** as well as **normal methods**. A normal class **cannot have abstract methods**. An abstract class is a class that contains at **least one abstract method**.

Abstract class: **is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class**).

Abstract Method is a method that has just the method definition but does not contain implementation. A method **without a body** is known as an Abstract Method.

Abstract classes and Abstract methods :

- An abstract class is a class that is declared with an abstract keyword.
- An abstract method is a method that is declared without implementation.
- An abstract class may or may not have all abstract methods. Some of them can be concrete methods

- A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make the subclass itself abstract.
- Any class that contains one or more abstract methods must also be declared with an abstract keyword.
- There can be no object of an abstract class.
- An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

```
abstract class Animal {
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
```

From the example above, it is not possible to create an object of the Animal class:

Animal myObj = new Animal(); // will generate an error

To access the abstract class, it must be inherited from another class. Let's convert the Animal class we used in the **Polymorphism**

Example

```
abstract class Animal {
  // Abstract method (does not have a body)
  public abstract void animalSound();
  public void sleep() {
    System.out.println("Zzz");
  }
}
class cat extends Animal {
```

```
  public void animalSound() {
    System.out.println("The cat says: mee mee");
  }
}
class Main {
  public static void main(String[] args) {
    cat mycat = new cat (); // Create a cat object
    mycat.animalSound();
    mycat.sleep();
  }
}
```

Note: Abstraction can also be achieved with Interfaces,

Encapsulation vs Data Abstraction

- Encapsulation is data hiding(**information** hiding) while Abstraction is detailed hiding(**implementation** hiding).
- While encapsulation groups together data and methods that act upon the data, data - abstraction deal with exposing the interface to the user and hiding the details of implementation.

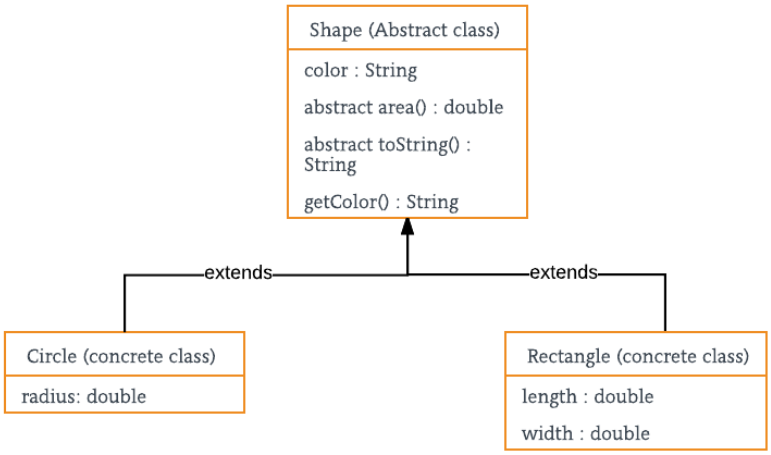
Advantages of Abstraction

- It reduces the complexity.
- Avoids code duplication and increases reusability.
- Helps to increase the security of an application or program as only important details are provided to the user.

When can we use abstract classes ?

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a **generalization** form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.

Consider a classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape" and each shape has a color, size, and so on. From this, specific types of shapes are derived (inherited)-circle, square, triangle, and so on — each of which may have additional characteristics and behaviors. For example, certain shapes can be flipped. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.




```
abstract class Shape {
   String color;
   abstract double area();
   public abstract String toString();
    public Shape(String color)
   {
      System.out.println("Shape constructor called");
```

```
        this.color = color;
    }
     public String getColor() { return color; }
}
class Circle extends Shape {
    double radius;
     public Circle(String color, double radius)
    {
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }
     @Override double area()
    {
        return Math.PI * Math.pow(radius, 2);
    }
     @Override public String toString()
    {
      return "Circle color is " + super.getColor()
            + "and area is : " + area();
    }
}
class Rectangle extends Shape {
     double length;
    double width;
     public Rectangle(String color, double length, double width)
    {
```

```
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }
     @Override double area() { return length * width; }
     @Override public String toString()
    {
        return "Rectangle color is " + super.getColor()
            + "and area is : " + area();
    }
}
public class Test {
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);
          System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

Output

```
Shape constructor called
Circle constructor called
Shape constructor called
Rectangle constructor called
Circle color is Redand area is : 15.205308443374602
Rectangle color is Yellowand area is : 8.0
```