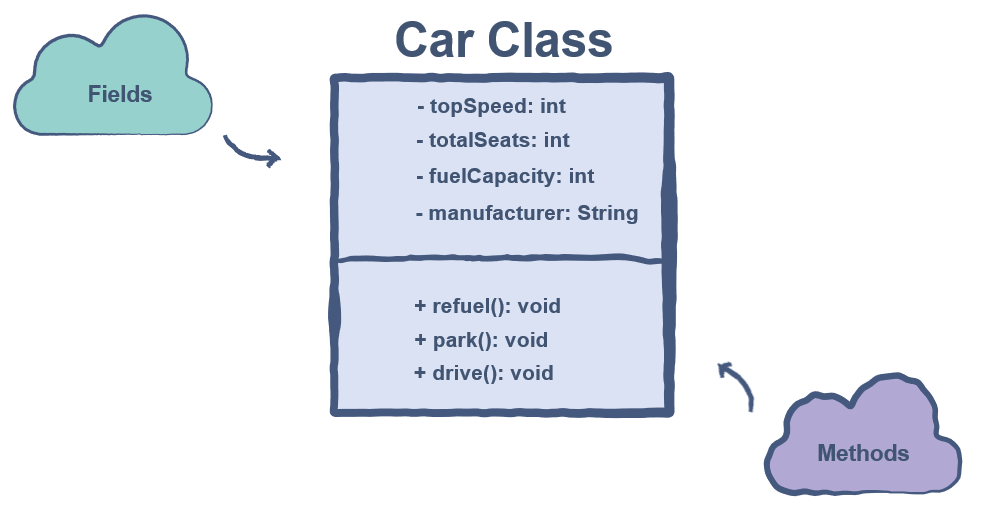
**Learn Object-Oriented Programming in Java**

An **object** is an instance of a module, and a class is its definition.

Let’s start with an example of a Car class. Below, we can see the **state** of a car object represented as variables:



We can see two types of attributes in the Car class above. In general, these two categories are present in all classes.

### Fields [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/xV6gx5M19XP#fields)

These are also known as the **member variables** of a class. This is because they contain the information relevant to the object of the class. A car object would have a top speed, a certain number of seats, and so many other pieces of data that we could store in variables.

### Methods [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/xV6gx5M19XP#methods)

This category of attributes enables the class object to perform operations using the fields. In the case of the car class, the refuel() function would fill up the fuelCapacity property of the object.

## Benefits of Using Classes [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/xV6gx5M19XP#benefits-of-using-classes)

The concept of classes allows us to create complex objects and applications in Java. This is why classes are the basic building blocks behind all of the OOP’s principles.

Classes are also very useful in compartmentalizing the code of an application. Different components could become separate classes which would interact through interfaces.

## Private

# It is a popular practice to keep the data members private since we do not want anyone manipulating our data directly. We can make members private using the keyword private.

## Public [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/xlyoyVW7kQJ#public)

This tag indicates that the members can be directly accessed by anything which is in the same scope as the class object.

# Member functions are usually public as they provide the interface through which the application can communicate with our private members.

Public members of a class can be accessed by a class object using the . operator. For example, if we have an object c of type Cop, we could access getGun() like this:

package justice;

class Cop {

  private int gun; // Private variable

  public int getGun(){

    return gun;  // The private variable is directly accessible over here!

  }

  protected void fire(){

    System.out.println("shoot!")

  }

}

Cop c = new Cop(); // Object created

c.getGun(); // Can access the gun

c.gun = 0; // This would cause an error since gun is private

## Protected

# In Java, the default access modifier is protected. The primary use of the protected tag can be found when using inheritance, which is the process of creating classes out of other classes.

# The protected data members can be accessed inside a Java package. However, outside the package, they can only be referred to through an inherited class.

package crime;

import justice.\*;

class Thief{

  public static void main(String args[]){

   Cop obj = new Cop();

   Cop.fire(); //Compile Time Error

  }

}

## Static and Non-static Fields [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qV2X5pyAvo7#static-and-non-static-fields)

Java supports static and non-static fields.

### Static Field [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qV2X5pyAvo7#static-field)

A static field resides in a class. All the objects we create will share this field and its value.

Static fields reside in the class. We don’t need an instance of the class to access static fields. We can access the static fields of a class by just writing the class name before the field:

class Car {

  // static fields

  static int speed;

  static int capacity;

}

// Static fields are accessible in the main

System.out.println(Car.speed);

System.out.println(Car.capacity);

### Non-Static Field [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qV2X5pyAvo7#non-static-field)

Non-static fields are located in the instances of the class. Each instance of the class can have its own values for these fields.

As non-static fields doesn’t reside in the class, So we need an instance of the class to access non-static fields.

## The Purpose of Methods [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/JE8Gg1qqlDD#the-purpose-of-methods)

# These methods can either alter the content of the data fields or use their values to perform a certain computation. All the useful methods that receive calls from outside of the class should be public, although, some methods which do not need to be accessed from the outside could be kept private.

## Advantages of Method Overloading [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/JE8Gg1qqlDD#advantages-of-method-overloading)

# 

# The Purpose of Getter and Setters on Private Variables [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/JE8Gg1qqlDD#the-purpose-of-methods)

1. Never create any setter for a secured property of your object. Only a getter can be provided.
2. Create setters only for those properties, which can change during the course of program.
3. Use setters if you want to apply certain restrictions on some properties e.g. apply invalid value checks, pre-population, logical analysis, populating another depending property, defensive copying etc
4. Getters/setters helps in maintaining the software entropy of a system. Read about software [**entropy**](http://en.wikipedia.org/wiki/Software_entropy).
5. Do not create getters/setters where it is not required as its leads to [**Boilerplate**](https://en.wikipedia.org/wiki/Boilerplate_code) code.
6. Getters/setters helps in changing the underlying implementation for future Extensions of Programs e.g. Upgrading Logging libraries etc

## this Reference Variable [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qV10v1215Y0#this-reference-variable)

The this reference variable exists for every class. It refers to the class object itself. We use the this when we have an argument which has the same name as a data member. this.memberName specifies that we are accessing the memberName variable of the particular class.

In Java, we can call a constructor from a constructor. When you call a constructor from another constructor, you use the this keyword to refer to the constructor.

class Date {

  private int day;

  private int month;

  private int year;

  private String event;

  // Default constructor

  public Date() {

    // We must define the default values for day, month, and year

    this.day = 0;

    this.month = 0;

    this.year = 0;

  }

  // Parameterized constructor

  public Date(int day, int month, int year){

    // The arguments are used as values

    this.day = day;

    this.month = month;

    this.year = year;

  }

  // Parameterized constructor

  public Date(int day, int month, int year, String event){

    this(day, month, year); // calling the constructor

    this.event = event;

  }

  // A simple print function

  public void printDate(){

    System.out.println("Date: " + day + "/" + month + "/" + year + "  --> " + event);

  }

}

class Demo {

  public static void main(String args[]) {

    // Call the Date constructor to create its object;

    Date date = new Date(1, 1, 2019, "New Year"); // Object created with specified values! // Object created with default values!

    date.printDate();

  }

}

# What is Data Hiding?

In layman’s terms, data hiding refers to the concept of **hiding the inner workings of a class** and simply providing an **interface** through which the outside world can interact with the class without knowing what’s going on inside.

The purpose is to implement classes in such a way that the instances (objects) of these classes should not be able to cause any unauthorized access or change in the original contents of a class. One class does not need to know anything about the underlying algorithms of another class. However, the two can still communicate.

## A Real Life Example [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/YQVkDpLkvNM#a-real-life-example)

Take the doctor-patient model. In case of an illness, the patient consults the doctor, after which he or she is prescribed the appropriate medicine. The patient only knows the process of going to the doctor. The logic and reasoning behind the doctor’s prescription of a certain medicine are unknown to the patient. This is a classic example of the patient class interacting with the doctor class without knowing the inner workings of the doctor class.

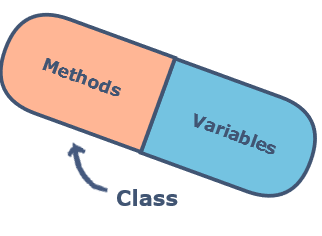
## Components of Data Hiding [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/YQVkDpLkvNM#components-of-data-hiding)

Data hiding can be divided into two primary components:

1. Encapsulation
2. Abstraction

Encapsulation is a fundamental programming technique in OOP used to achieve data hiding.

**Encapsulation** in OOP refers to binding the **data** and the **methods to manipulate that data** together in a single **unit** (class). Encapsulation refers to the concept of binding **data and the methods operating on that data** in a single unit also called a class.

As a rule of thumb, a good convention is to declare all the *data members or instance variables* of a class private. This will restrict direct access from the code outside that class.

Well, the answer to this is simple. One has to implement public methods to let the outside world communicate with this class. These methods can be getters, setters and any other custom methods implemented by the programmer.

### Advantages of Encapsulation [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/YQlg26YgLRW#advantages-of-encapsulation)

* Classes are easier to change and maintain.
* We can specify which data member we want to keep hidden or accessible.
* We decide which variables have read/write privileges (increases flexibility).

# What is Inheritance?

## The Terminologies [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qAXDNK8QB6y#the-terminologies)

* **SuperClass (Mother Class or Base Class):** This class allows the *re-use* of its non-private members in another class.
* **SubClass (Child Class or Derived Class):** This class is the one that *inherits* from the superclass.

**Inheritance** provides a way to create a new class from an existing class. The new class is a specialized version of the existing class such that it **inherits all the *non-private* fields (*variables*) and *methods*** of the existing class. The existing class is used as a starting point or as a *base* to create the new class.

the next question that comes to your mind is What is the use case of inheritance? Well, the answer is that wherever we come across an ***IS A*** relationship between objects, we can use inheritance.

* A square ***IS A*** shape
* Java ***IS A*** programming language
* Car ***IS A*** vehicle

Some classes cannot be inherited. Such classes are defined with the keyword, final. An example of such a class is the built-in [Integer class](https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html) - this class cannot have derived classes.

// Base Class Vehicle

class Vehicle {

  // Private Fields

  private String make;

  private String color;

  private int year;

  private String model;

  // Parameterized Constructor

  public Vehicle(String make, String color, int year, String model) {

    this.make = make;

    this.color = color;

    this.year = year;

    this.model = model;

  }

  // public method to print details

  public void printDetails() {

    System.out.println("Manufacturer: " + make);

    System.out.println("Color: " + color);

    System.out.println("Year: " + year);

    System.out.println("Model: " + model);

  }

}

// Derived Class Car

class Car extends Vehicle {

  // Private field

  private String bodyStyle;

  // Parameterized Constructor

  public Car(String make, String color, int year, String model, String bodyStyle) {

    super(make, color, year, model);  //calling parent class constructor

    this.bodyStyle = bodyStyle;

  }

  public void carDetails() {  //details of car

    printDetails();         //calling method from parent class

    System.out.println("Body Style: " + bodyStyle);

  }

}

class Main {

  public static void main(String[] args) {

    Car elantraSedan = new Car("Hyundai", "Red", 2019, "Elantra", "Sedan"); //creation of car Object

    elantraSedan.carDetails(); //calling method to print details

  }

}

## What is the super Keyword? [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qZNQMQV6MDR#what-is-the-super-keyword)

The super keyword in Java is used to refer to the *SuperClass* members from inside the **immediate *Subclass*.**

## Accessing Parent Class Fields [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qZNQMQV6MDR#accessing-parent-class-fields)

Consider the fields named as fuelCap defined inside a Vehicle class to keep track of the *fuel capacity* of a vehicle. Another class named as Car *extends* from this Vehicle class. We declare a field inside the Car class with the same name i.e. fuelCap but different value. Now if we want to refer to the fuelCap field of the *SuperClass* inside the *Subclass,* we will then have to use the super keyword.

class Vehicle { //Base class vehicle

  int fuelCap = 90; //fuelCap field inside SuperClass

}

class Car extends Vehicle { // sub class Car extending from Vehicle

  int fuelCap = 50; //fuelCap field inside SubClass

  public void display() {

    //accessing the field of parent class using super\*/

    System.out.println("Fuel Capacity from the Vehicle class: " + super.fuelCap);

    //without using super the field of current class shadows the field of parant class\*/

    System.out.println("Fuel Capacity from the Car class: " + fuelCap);

  }

}

class Main {

  public static void main(String[] args) {

    Car corolla = new Car();

    corolla.display();

  }

}

### Calling a Parent Class Method [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qZNQMQV6MDR#calling-a-parent-class-method)

Just like the fields, super is also used with the methods. Whenever a *SuperClass* and the immediate *SubClass* have any methods with the **same name** we use super to access the methods from the *SuperClass* inside the *SubClass*. Let’s go through an example:

class Vehicle {          //Base class vehicle

  public void display() {   //display method inside SuperClass

    System.out.println("I am from the Vehicle Class");

  }

}

class Car extends Vehicle { // sub class Car extending from Vehicle

  public void display() { //display method inside SubClass

    System.out.println("I am from the Car Class");

  }

  public void printOut(){

    System.out.println("The display() call with super:");

    super.display();  //calling the display() of Vehicle(SuperClass)

    System.out.println("The display() call without super:");

    display();        //calling the display() of the Car(SubClass)

  }

}

class Main {

  public static void main(String[] args) {

    Car corolla = new Car();

    corolla.printOut();

  }

}

### Using with Constructors [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/qZNQMQV6MDR#using-with-constructors)

**Important Note:** When you create an Object of a *SubClass* type at the same time, an Object of *SuperClass* type is created by calling implicitly the constructor of *SuperClass.*

super();  //calls the (no argument) constructor if a no argument constructor is defined in the SuperClass  
   
super(parameters); //calls the parameterized constructor of the SuperClass with matching parameters from the SubClass constructor

# Advantages of Inheritance

[Re-usability](https://www.educative.io/courses/learn-object-oriented-programming-in-java/x1zjRJG90Gl#re-usability) [Avoiding Duplication of Code](https://www.educative.io/courses/learn-object-oriented-programming-in-java/x1zjRJG90Gl#avoiding-duplication-of-code) [Extensibility](https://www.educative.io/courses/learn-object-oriented-programming-in-java/x1zjRJG90Gl#extensibility) [Data Hiding](https://www.educative.io/courses/learn-object-oriented-programming-in-java/x1zjRJG90Gl#data-hiding)

Using inheritance, one can extend the base class logic as per the business logic of the derived class. This is an easy way to upgrade or enhance specific parts of a product without changing the core attributes. An existing class can act as a base class to derive a new class having upgraded features.

# [Implementing two interfaces in a class with same method. Which interface method is overridden?](https://stackoverflow.com/questions/2801878/implementing-two-interfaces-in-a-class-with-same-method-which-interface-method)

If a type implements two interfaces, and each interface define a method that has identical signature, then in effect there is only one method, and they are not distinguishable. If, say, the two methods have conflicting return types, then it will be a compilation error. This is the general rule of inheritance, method overriding, hiding, and declarations, and applies also to possible conflicts not only between 2 inherited interface methods, but also an interface and a super class method, or even just conflicts due to type erasure of generics.

# What is Polymorphism?

what if we want our derived class to inherit a method from the base class and have a different implementation for it? That is when polymorphism, a fundamental concept in the OOP paradigm, comes into play.

Example [#](https://www.educative.io/courses/learn-object-oriented-programming-in-java/B8ng0xp6MoW#example)

Here we consider the example of a **Shape** class, which is the base class while many shapes like *Rectangle and Circle* extending from the base class are derived classes. These classes contain the **getArea()** method which calculates the area for the respective shape.

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Overloading happens at **compile time.** | Overriding happens at **runtime** |
| Gives better performance because the binding is being done at compile time. | Gives less performance because the binding is being done at run time. |
| **Private** and **final** methods can be overloaded. | **Private** and **final** methods can not be overridden. |
| Return type of method does not matter in case of method overloading. | Return type of method must be the same in the case of overriding. |
| Arguments must be different in the case of overloading. | Arguments must be the same in the case of overriding. |
| It is being done in the same class. | Base and derived classes are required here. |
| Mostly used to increase the readability of the code. | Mostly used to provide the implementation of the method that is already provided by its base class. |

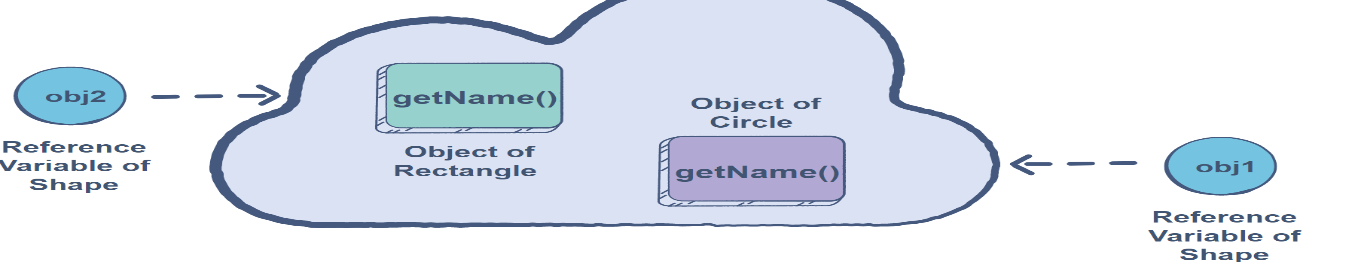
**Dynamic polymorphism** is the mechanism by which methods can be defined with the same name, return type, and parameters in the base class and derived classes. The call to an overridden method is decided at the runtime.

Shape obj1 = new Circle(3);

Shape obj2 = new Rectangle(2, 3);

obj1.getName();

obj2.getName();



obj1 is a reference to the Circle class

obj2 is a reference to the Rectangle class

| **Static Polymorphism** | **Dynamic Polymorphism** |
| --- | --- |
| Polymorphism that is resolved during compile time is known as static polymorphism. | Polymorphism that is resolved during run time is known as dynamic polymorphism. |
| Method overloading is used in static polymorphism. | Method overriding is used in dynamic polymorphism. |

# What is Abstraction?

**Abstraction** in Object-Oriented Programming refers to showing only the essential features of an object to the user and hiding the inner details to reduce complexity. It can be put this way that the user only has to know *“what an object does?”* rather than *“how it does?”*.

* A user can only use and interact with the limited features of an application and is unaware of the implementation details or the way the application was developed. Usually, the users are only concerned with the functionality of an application.

The Math.min() method is used to find the minimum of the two numbers

The Math.pow() method is used to find 2 to the power 2.

An abstract data type (or class) is a type that only defines ‘what operations are to be performed?’ rather than ‘how to be performed?’

An example of abstract data type can be a built-in [stack](https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html) class in Java for which the user knows that it has the push(), pop(), size() e.t.c. methods but doesn’t know how these are implemented.

In Java, we use the following components to achieve abstraction:

* *Abstract Classes*
* *Interfaces*

## The abstract Keyword [**#**](https://www.educative.io/courses/learn-object-oriented-programming-in-java/mEQxA4pvxAR#the-abstract-keyword)

In Java, it is impossible to achieve abstraction without using the abstract keyword. The abstract keyword can be used with the methods and classes only.

Those methods or classes then only specify “what operations should be done” and whoever will use this method or class in their code will have to deal with the implementation details of this method or class.

* An *abstract method* **cannot** be declared *private* as it has to be implemented in some other class.
* An **abstract method** can be declared inside an abstract class or an interface only.
* public abstract void methodName(parameter(s));
* An **abstract class** is a class which is declared using the keyword abstract.
* An abstract class ***cannot*** be instantiated i.e. one cannot create an object of an *abstract class*.

**Interfaces**

* An interface can have:
  + abstract method(s)
  + default method(s)
  + static method(s)
  + public static final variable(s)
* All the methods declared or implemented in an interface are by default public and all the variables are by default public static final.
* Just like an abstract class, an interface cannot be instantiated.
* A class cannot extend from more than one class, but it can implement **any number** of interfaces.”?
* An interface can extend from another interface.
* An interface cannot be declared private or protected.

## Advantages of Interfaces [**#**](https://www.educative.io/courses/learn-object-oriented-programming-in-java/JEX2zpqGjNP#advantages-of-interfaces)

* Interfaces can be used to achieve *loose coupling* in an application. This means that a change in one class doesn’t affect the implementation of the other class.
* By the use of interfaces, one can break up complex designs and clear the dependencies between objects.
* In Java, multiple inheritance can be implemented using interfaces.

| **Interfaces** | **Abstract Classes** |
| --- | --- |
| Can have abstract method(s) only. | Can have concrete (non-abstract) & abstract method(s) |
| Support multiple inheritance | Don’t support multiple inheritance |
| All members are public | Can have private, protected and public members |
| All data members are static and final | Can have non-static and non-final members too |
| Can’t have constructors | Constructors can be defined |

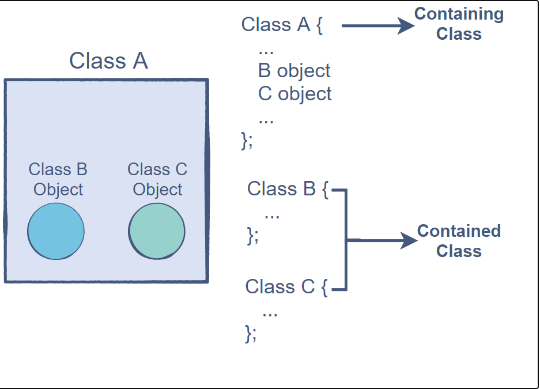
## Interaction Between Class Objects [**#**](https://www.educative.io/courses/learn-object-oriented-programming-in-java/7nVBGyRPmEA#interaction-between-class-objects)

The next step for us is to use different class objects to create the design of an application. This means that objects of independent classes would have to find a way to interact with each other.

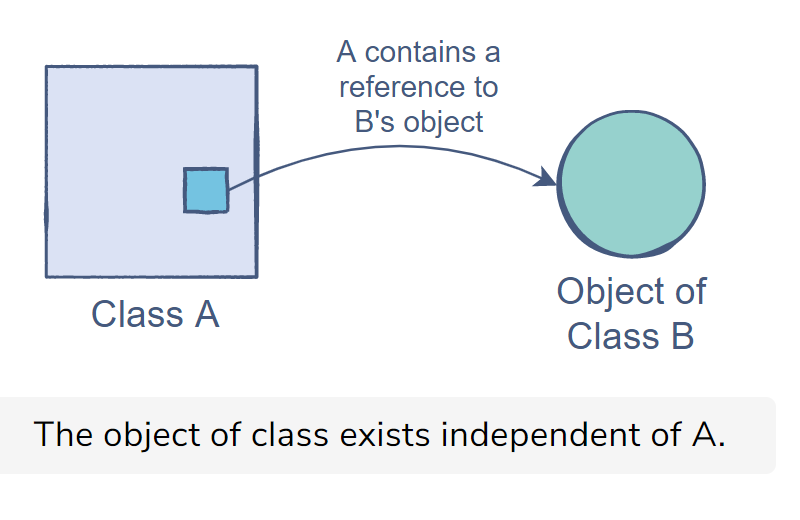
## Relationships Between Classes [**#**](https://www.educative.io/courses/learn-object-oriented-programming-in-java/7nVBGyRPmEA#relationships-between-classes)

## We have studied the **IS A** relation in the [Inheritance](https://www.educative.io/collection/page/10370001/5692479535841280/6489377230290944). We’ll study the other two below:

### Part-of



### Has-a



**Aggregation** follows the **has-A** model. In **aggregation**, the lifetime of the owned object does not depend on the lifetime of the owner.

In aggregation, the parent only contains a **reference** to the child, which removes the child’s dependency.

// Player class

class Player {

  String name;

  int id;

  String team;

  Player(String name, int id, String team) {

    this.name = name;

    this.id = id;

    this.team = team;

  }

}

/\* Team class contains a list of Player

Objects.\*/

class Team {

  String name;

  private List<Player> players;

  Team(String name, List<Player> players) {

    this.name = name;

    this.players = players;

  }

  public List<Player> getPlayers() { // This function returns the "players"

    return players;

  }

}

/\* School class contains a list of Team

Objects.\*/

class School {

  String schoolName;

  private List<Team> teams;

  School(String schoolName, List<Team> teams) {

    this.schoolName = schoolName;

    this.teams = teams;

  }

  /\* Count total players of all teams

    in a given school \*/

  public int getTotalPlayersInSchool() {

    int noOfPlayers = 0;

    List<Player> players;

    for(Team team : teams) {

      players = team.getPlayers();

      for(Player p : players) {

        noOfPlayers++;

      }

    }

    return noOfPlayers;

  }

}

// Main class

class Main {

  public static void main (String[] args) {

    /\* Declaring all the players \*/

    Player p1 = new Player("Harris", 1, "Red");

    Player p2 = new Player("Carol", 2, "Red");

    Player p3 = new Player("Johnny", 1, "Blue");

    Player p4 = new Player("Sarah", 2, "Blue");

    /\* Making a List of

        "Red" team Players. \*/

    List <Player> red\_players = new ArrayList<Player>();

    red\_players.add(p1);

    red\_players.add(p2);

    /\* Making a List of

        "Blue" team Players. \*/

    List <Player> blue\_players = new ArrayList<Player>();

    blue\_players.add(p3);

    blue\_players.add(p4);

    /\* Declaring Team objects \*/

    Team red = new Team("Red", red\_players);

    Team blue = new Team("Blue", blue\_players);

    // Creating a list of teams and adding "red" and "blue" teams to it.

    List <Team> teams = new ArrayList<Team>();

    teams.add(red);

    teams.add(blue);

    // Creating an instance of School.

    School mySchool = new School("ABC", teams);

    System.out.println("Total players in my school: ");

    // Getting total prayers in the school.

    System.out.println(mySchool.getTotalPlayersInSchool());

  }

}

Composition is the practice of creating other class objects in your class. In such a scenario, the class which creates the object of the other class is known as the *owner* and is responsible for the lifetime of that object. In **composition**, the lifetime of the owned object depends on the lifetime of the owner.

class Engine {

  private int capacity;

  public Engine(){

    capacity = 0;

  }

  public Engine(int cap) {

    capacity = cap;

  }

  public void engineDetails() {

    System.out.println("Engine details: " + capacity);

  }

}

class Tires {

  private int noOfTires;

  public Tires() {

    noOfTires = 0;

  }

  public Tires(int nt) {

    noOfTires = nt;

  }

  public void tireDetails() {

    System.out.println("Number of tyres: " +  noOfTires);

  }

}

class Doors {

  private int noOfDoors;

  public Doors() {

    noOfDoors = 0;

  }

  public Doors(int nod) {

    noOfDoors = nod;

  }

  public void doorDetails() {

    System.out.println("Number of Doors: " + noOfDoors);

  }

}

class Car {

  // We are creating obj with in the Car Class so these objects are “Part Of”

// Car class and these obj get deleted once the Car’s obj is deleted.

  private Engine eObj;

  private Tires tObj;

  private Doors dObj;

  private String color;

  public Car(String col, int cap, int nt, int nod) {

    this.eObj = new Engine(cap);;

    this.tObj = new Tires(nt);;

    this.dObj = new Doors(nod);

    color = col;

  }

  public void carDetail() {

    eObj.engineDetails();

    tObj.tireDetails();

    dObj.doorDetails();

    System.out.println("Car color: " + color);

  }

}

class Main {

  public static void main(String[] args) {

    Car cObj = new Car("Black", 1600, 4, 4);

    cObj.carDetail();

  }

}

We have created a Car class which contains the objects of Engine, Tires and Doors classes. The Car class is responsible for the lifetime of the owned objects, i.e., when the Car dies, so does the *tires*, *engine* and *doors*.

Another Example :

// Car class

class Car {

  // Declaring data members and methods

  private int id;

  private String model;

  private String color;

  public void carFeatures() { // Function to print out car features

    System.out.println("Car Model: " + model);

    System.out.println("Car Color: " + color);

  }

  public void setModel(String model) {

    this.model = model;

  }

  public void setColor(String color) {

    this.color = color;

  }

}

// Toyota Class, which is a child class of Car class.

class Toyota extends Car {

  // Inherits all properties of Car class

  public void setStart() {

    // Declaring an engine object and calling the start() function of the engine.

    ToyotaEngine engine = new ToyotaEngine();

    engine.start();

  }

}

// Engine class

class ToyotaEngine {

  // This function simply prints out on screen that Engine has been started!

  public void start() {

    System.out.println("Engine has been started.");

  }

  // This function simply prints out on screen that Engine has been stopped!

  public void stop() {

    System.out.println("Engine has been stopped.");

  }

}

class Main {

  public static void main(String[] args) {

    // Declaring and initializing Toyota object

    Toyota t = new Toyota();

    t.setModel("Fortuner");

    t.setColor("Silver");

    t.carFeatures();

    t.setStart();

  }

}