

## Introduction to Oops:

"Welcome to the Object oriented world – a virtual world". These sound like words from the movie Matrix – right. Yes, the moment when you made a decision to learn about the [Object oriented programming](#), you are going to be in two worlds – The real world – what you are in and a virtual world where your objects act like what you have scripted.

The learning and programming in Object oriented programming model is going to most excited if you think in objects.

The real aim of the programming language is to simulate the real world into the computers – Right!. Yes, think of the game Road rash in which your car speeds at 140 kmph and when hits a tree or a lamp post, its speed comes down to zero as in the real world.

Object oriented Programming is the most successful among the programming models in helping to simulate the real world into the computers using programs.

Some of the languages which object based programming languages are given below:

- Smalltalk
- C++
- Java
- .NET

## History:

Before Object oriented programming was born, there were structural programming, procedural programming and modular programming.

### Structural Programming:

The statements in this programming model are executed one after the other. To access some functionality or expression, GOTO statements will be used which makes the programs too complex. Fortran is based on Structural programming.

### Procedural Programming:

To overcome the complexity of the structural programming, procedural programming has evolved where the reusable functionality is separated as procedures and can be called anywhere using call back statements. These are also called functions.

### Modular Programming:

Maintaining a huge number of procedures can be cumbersome. Modular programming helps to group the procedures to keep them as modules.

### Issues with these programming models:

The major issue with these programming models is the lack of [data security](#). The data can be accessed by any procedure and there was no proper way of assigning the attributes.

### Evolution of Object oriented programming:

All the issues faced by the other models of programming are solved in Object oriented programming.

Its was in the year 1980, the Object based programming language – Smalltalk was released followed by C++.

## Thinking in Objects

To program in Object based languages, it is extremely important to understand what are objects.

A programmer should think and keep in mind that "**Everything in this world is an object**". We are living with many number of objects around us and use a lot of them to make our daily life keep going.

For example, we use car to reach to office, use pen to write, mobile to make a call.

So, everything around us are objects including ourselves.

The next question is that what really an object has? Every object has some properties and behavior.

Let us consider a pen we use. It has a price, name, color, weight etc., The responsibility or behavior of pen is to write.

We use pen to write.

So, Pen as an object has price, name, color and weight as properties which can be stored using [data types](#) like int, string, double respectively.

The behavior write can be represented as a method write() in the pen. This behavior is there to be invoked by other objects like student, teacher etc.,

Let us consider a small scenario of a college where a Student learns from a teacher who teaches the students. Students use pen to write the exam and teachers use pen to correct the papers and both use car to reach to college.

In the object oriented world, any object can invoke any object in the system.

Read the detailed article on [identifying the objects in a requirement](#).

## Objects are instances of types

Every object in the system should belong to a type. So what is a type actually?

Let me explain this with a simple scenario. You have document written by your grand father in his hand writing. Now you have took three photocopies of that document. We call the photocopies as instance of the original document.

In the same fashion, a type is a class in programming which defines the properties and behavior of the objects and any number of instances (objects) can be created using the new operator.

Refer to a detailed article on Classes and instances in OOPs.

Four pillars of Object oriented programming

To represent the real world into the programming, Object oriented programming relies on four important concepts.

1. [Inheritance](#)
2. [Encapsulation](#)
3. [Abstraction](#)
4. [Polymorphism](#)

## Identifying Objects in a requirement

Identifying Objects is the most important task in modeling and creating types in [Object oriented programming](#). In this post let us take a simple requirement which needs to be implemented using the Object oriented programming language.

### Requirement Description:

*An Employee management system has to be created for a factory. The requirements are as given below:*

- *All the Employee should report at the security to log the time when they come to duty and also log the out time when they leave factory.*
- *The security persons should check the register by end of day and check if any employee has not logged his time and should mark absent.*
- *The accountant should calculate the number of days an employee has worked and should credit the salary to the employee's bank account.*
- *The employee will have an employee id, name, department, bank account number etc.,*

*The software should ask for an employee id, his in time and out time of a day and should give the details of number days an employee has worked.*

Let us identify the Objects in this requirement.

To identify the objects, we need to analyze which or who is acting or does some work. That is first we need to identify the actors from the requirement.

- Employee is the first actor who should report at security.
- Security is second actor who logs the time to the system.
- The accountant is the third actor who calculates the salary to be paid.

You can observe that all the actors here are actually nouns from the sentences of the requirements .

Most of the time, the nouns form the actors from a requirement description. But do remember that not all nouns turn to be actors. The actors which have some action in the system are actors.

Now let us model the object. We know that every object has attributes and methods as members which represent the data and behavior of the actor respectively.



First let us consider the actor Employee.

- In the requirement, it is mentioned that the employee has employee id, name, account number etc., There are the data which belongs to employee which can be stored using java data types.
- The second thing is to identify the Employees behavior. The Employee has to report the in time at security. Employee has to report the out time at security.
- So, the behavior of the employee is report in time and report out time. We can use the same phrases to name the methods in the Employee as reportInTime() and reportOutTime().

The second actor is Security.

- The security has no data given in the requirement. He just has the behavior to log the time details to the systems.
- So, we can go ahead to add a behavior – logReportTimings().

The third actor is the accountant.

- His behavior is to calculate the number of days worked and pay salary to employees. so, we can go ahead to create the methods calculateTotalDaysWorked() and paySalary() as the methods in accountant.

Given below are the **java classes** which represent these types as object in the system.

```

class Employee{
    int employeeId;
    String employeeName;
    String empBankAccNumber;
    String department;

    void reportInTime(){
        // Functionality of reporting the in time.
    }

    void reportOutTime(){
        //Functionality to report out time.
    }
}

Security:

class Security{
    void logReportTimings(){
        //Functionality to log the timings of the employess.
    }
}

class Accountant{

```

```

void calculateTotalDaysWorked() {
}
int paySalary() {
}
}

```

Please note that the name of the class should represent an actor and not the work done by the actor in the system.

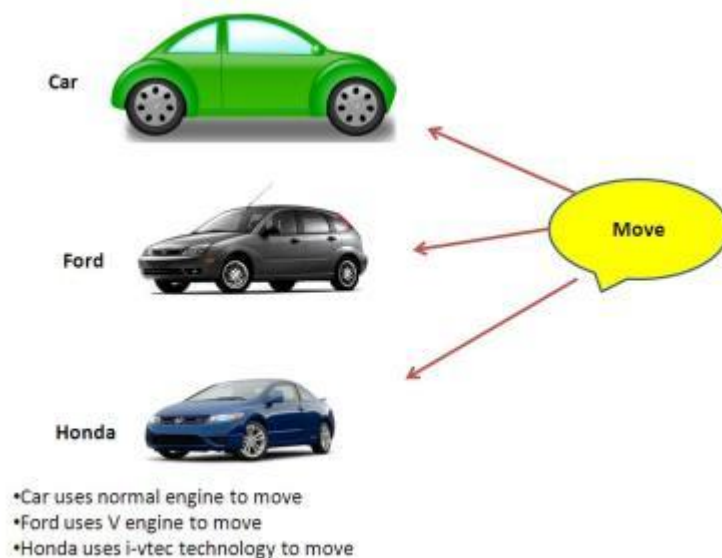
## Polymorphism in Object oriented Programming

The definition of **Polymorphism** as per the dictionary is 'The occurrence of something in different forms, in particular'.

As per biology, the **polymorphism** is nothing but exhibiting different forms by a species. **Poly means many**. The same principle is applicable in [Object oriented programming](#) where the Objects at runtime decide what behavior will be invoked.

**Polymorphism** is implemented in **Java** using **method overloading** and **method overriding** concepts.

### Polymorphism with example:



Let us Consider Car example for discussing the **polymorphism**. Take any brand like Ford, Honda, Toyota, BMW, Benz etc., Everything is of type Car.

But each have their own advanced features and more advanced technology involved in their move behavior.

Now let us create a basic type Car

**Car.java**

```

public class Car {
    int price;
    String name;
    String color;

    public void move() {
        System.out.println("Basic Car move");
    }
}

```

Let us implement the Ford Car example.

Ford extends the type Car to inherit all its members(properties and methods).

#### Ford.java

```
public class Ford extends Car{
    public void move(){
        System.out.println("Moving with V engine");
    }
}
```

The above Ford class extends the Car class and also implements the move() method. Even though the move method is already available to Ford through the [Inheritance](#), Ford still has implemented the method in its own way. This is called method overriding.

#### Honda.java

```
public class Honda extends Car{
    public void move(){
        System.out.println("Move with i-VTEC engine");
    }
}
```

Just like Ford, Honda also extends the Car type and implemented the move method in its own way.

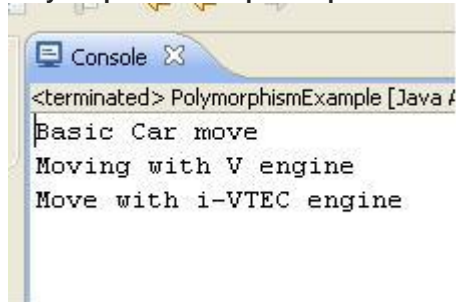
Method overriding is an important feature to enable the **Polymorphism**. Using Method overriding, the Sub types can change the way the methods work that are available through the inheritance.

#### PolymorphismExample.java

```
public class PolymorphismExample {
    public static void main(String[] args) {
        Car car = new Car();
        Car f = new Ford();
        Car h = new Honda();

        car.move();
        f.move();
        h.move();
    }
}
```

#### Polymorphism Example Output:



In the **PolymorphismExample** class main method, i have created three objects- Car, Ford and Honda. All the three objects are referred by the Car type.

Please note an important point here that **A super class type can refer to a Sub class type of object but the vice versa is not possible**. The reason is that all the members of the super class are available to the subclass using inheritance and during the compile time, the compiler tries to evaluate if the reference type we are using has the method he is trying to access.

So, for the references car,f and h in the **PolymorphismExample**, the move method exists from Car type. So, the compiler passes the compilation process without any issues.

But when it comes to the run time execution, the virtual machine invokes the methods on the objects which are sub types. So, the method move() is invoked from their respective implementations.

So, all the objects are of type Car, but during the **run time**, the execution depends on the **Object** on which the invocation happens. This is called **polymorphism**.

### Polymorphism using abstract and interface:

**Polymorphism** can be implemented using the **abstract** and **interface** keywords, which enforce to implement the methods in the sub types. More on this on **Method overriding** and **overloading**.

As per the **Object oriented analysis and Design**, **polymorphism** is a very important feature to change the behavior of the application in the run time. By changing the object on which the method is invoked, we have the control to change the behavior of the application.

There are lot of [design patterns](#) that depend on this concept to make complex things simple.

## Summary:

**Polymorphism** is a very important concept in **object oriented programming** which enables to change the behavior of the applications in the run time based on the object on which the invocation happens. This enables to easily change the system without making much changes to the application. **Polymorphism** is implemented using the concept of **Method overloading** and **method overriding**. This can only happen when the classes are under the parent and child relationship using **inheritance**.

## Inheritance in Object oriented programming

### Inheritance in OOPs

Inheritance is a very important feature in [object oriented programming](#).

Inheritance is extremely useful in enabling the **re-usability** of the code and is enabled by using the **extends** keyword.

### Inheritance and its advantages

Just like a child inherits some behaviors from their parents, Objects in the programs also get some methods and properties from their parent classes.

Inheritance helps to access the basic functionality already declared in the parent types and gives an opportunity to declare new functionality in the sub types.

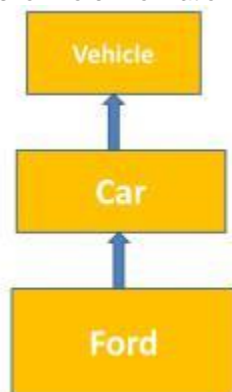
This gives the child an opportunity to declare some new methods and properties in it and still be able to have access to its super classes methods and properties through its instances.

Please note that **there is no [multiple inheritance](#) in Java**. A class can extend only one class. One has to consider using interfaces to add more functionality.

Let me explain **Inheritance with an example**.

Consider we have a type called vehicle. Its has a basic behavior move. Next we are creating a type called Car. It has more behaviors like move forward and backward. Next i am creating a type called Ford. This has more advanced behavior to move faster.

Let us create the classes for these. Refer to my article on [Identifying object in requirement](#) and classes and types for more information on creating types.



To create a new type we need to create classes. So let me create the class Vehicle.

```
class Vehicle{  
  
    void move () {
```

```

        System.out.println("Vehicle moves");
    }
}

class Car extends Vehicle{
    void moveForward(){
        System.out.println("Car moving forward");
    }
    void moveBackward(){
        System.out.println("Car moving backward");
    }
}

class Ford extends Car{
    int moveFaster(){
        int speed = 140;
        System.out.println("Ford-moving at speed");
        return speed;
    }
}

```

Now as per the inheritance rule, when i create an instance of Ford type, it should be able to access the methods of Car and Vehicle using the Fords Instance.

In the same fashion, when i create an instance of Car type, i will be able to access the cars and vehicles methods but not its child Ford.

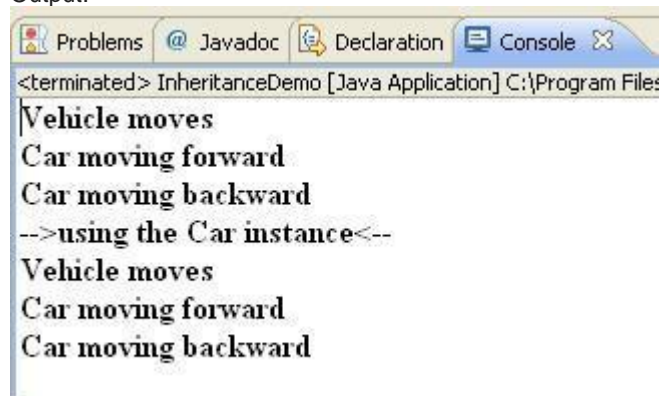
So, Please note that in inheritance, a child instance has access to all the members of its super classes but a parent will not have any idea what types are actually extending it.

```

class InheritanceDemo{
    public static void main(String[] args){
        Ford f = new Ford();
        f.move();
        f.moveForward();
        f.moveBackward();
        //Car instance
        System.out.println("using the Car instance");
        Car c = new Car();
        c.move();
        c.moveForward();
        c.moveBackward();
    }
}

```

Output:



```
<terminated> InheritanceDemo [Java Application] C:\Program Files
Vehicle moves
Car moving forward
Car moving backward
-->using the Car instance<--
Vehicle moves
Car moving forward
Car moving backward
```

### Using Super keyword:

**Super** keyword is used to access the super class methods and properties in the sub class methods. This keyword helps to access the super class members without creating any instances. **Super keyword** can only be used in a class with extends another class.

**Super keyword can also be used to invoke the super class constructors. Super(Parameters) should be used in the first line of the constructor.**

**Note:** The accessibility of the properties and methods of a type are subject to the [access modifiers](#) applied to the respective members. The methods and properties of super class are visible to the sub classes only when the access levels are set to public, default, protected. More about these in [Java access modifiers](#).

## Abstraction in Object Oriented Programming

### Abstraction in OOPs

**Abstraction** in [Object Oriented Programming](#) helps to hide the irrelevant details of an object. **Abstraction** is separating the functions and properties that logically can be separated to a separate entity which the main type depends on.

This kind of **Abstraction** helps in separating the members that change frequently. Abstraction is one of the key principles of the [OOAD](#) (Object oriented analysis and Design). Applying **Abstraction** during the design and domain modeling, helps a lot in designing a system which is flexible and maintainable.

**Abstraction is achieved by Composition.**

### Abstraction Example:

A Car has Engine, wheels and many other parts. When we write all the properties of the Car, Engine, and wheel in a single class, it would look this way:

```
public class Car {
    int price;
    String name;
    String color;
    int engineCapacity;
    int engineHorsePower;
```



```

String wheelName;
int wheelPrice;

void move() {
    //move forward
}
void rotate() {
    //Wheels method
}

void internalCombustion() {
    //Engine Method
}
}

```

In the above example, the attributes of wheel and engine are added to the Car type. As per the programming, this will not create any kind of issues. But when it comes to maintenance of the application, this becomes more complex.

## Abstraction has three advantages:

1. By using **abstraction**, we can separate the things that can be grouped to another type.
2. Frequently changing properties and methods can be grouped to a separate type so that the main type need not under go changes. This adds strength to the **OOAD** principle - "**Code should be open for Extension but closed for Modification**".
3. Simplifies the representation of the **domain models**.

**Applying the abstraction with composition**, the above example can be modified as given below:

```

public class Car {
    Engine engine = new Engine();
    Wheel wheel = new Wheel();

    int price;
    String name;
    String color;

    void move() {
        //move forward
    }
}

```

```

public class Engine {
    int engineCapacity;
    int engineHorsePower;

    void internalCombustion() {
        //Engine Method
    }
}

```

```

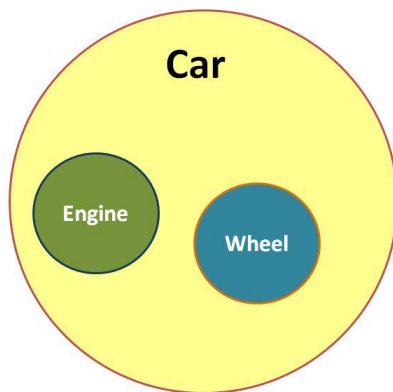
public class Wheel {
    String wheelName;
    int wheelPrice;

    void rotate() {
        //Wheels method
    }
}

```

You can see that the attributes and methods related to the Engine and Wheel are moved to the respective classes.

Engine and Wheel are referred from the Car type. When ever an instance of Car is created, both Engine and Wheel will be available for the Car and when there are changes to these Types(Engine and Wheel), changes will only be confined to these classes and will not affect the Car class.



## Summary:

**Abstraction** is one of the fundamental principles of **Object Oriented Programming languages**.

It helps to reduce the complexity and also improves the maintainability of the system. When combined with the concepts of the [Encapsulation](#) and **Polymorphism**, **Abstraction** gives more power to the Object oriented programming languages.

## Encapsulation in OOPs

### Encapsulation in Object Oriented Programming

Encapsulation is one of the four fundamentals of the [Object oriented programming](#).

#### What is Encapsulation?

Encapsulation is a language mechanism to restrict the access of the Objects components to other Objects or Classes.

Encapsulation helps in enforcing the security of the data related to a particular object. In the programming models like Structural, procedural and Modular programming, there was not much provision to safeguard the data from the other procedures or functions. This can lead to the misuse of data or even the design by responsibility principle of [OOAD](#) can be some times breached.

Every Object in Object oriented programming has full control of what members(Fields and methods) to be accessible to its peers in a package, outside the package and should only be visible to its own members alone. Encapsulation in Object Oriented Programming is implemented by mentioning the [Access Modifiers](#) like public, protected, private and default access.

The access to the data and methods can be restricted to a package, classes level etc.,

## Encapsulation with Example

Let me explain the concept of Encapsulation with an example. Say, you are a student in a class and some of the things you and your friends know, say nick names, should not be revealed to the professors, your parents does not have knowledge of you not attending the classes or poor performance in a subject etc.,

In our own life, we would like to shield/share some information to a group of people. In the same fashion, object as instances of a class has full control of its members.

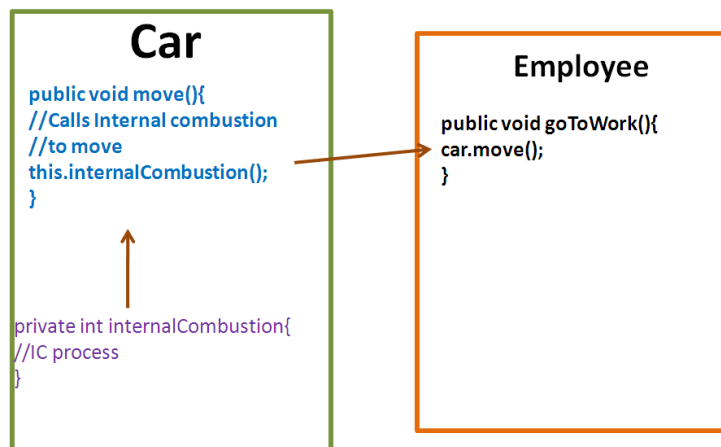
When a class is created and its members are defined, the access modifiers are applied to its fields and methods.

When an object is created using that particular class, the access will be restricted based on the defined access of the members.

Encapsulation is nothing but hiding the features that are not relevant for other objects but are essential for their own features.

For Example, I have create a class Car which has two methods:

## Encapsulation



1. `move()`.
2. `internalCombustion()`

The move method depends on internalCombustion() method. But an Employee needs only move method and need not be aware of internalCombustion behavior. So, the internalCombustion behavior can be declared private so that it can be only visible to the members inside the class.

## Summary:

Encapsulation is a primary fundamental principle in Object oriented programming. It is used to enforce the security to restrict the visibility of the members to other components.