This blog is aimed to provide a complete guide to OOPs concept using java from beginner as well as from interview perspective. This blog will not give you knowledge on how Java was released but it will provide all the necessary information and basic concepts, not only using theoretical definitions trivial vehicle and car examples but issues from real time projects. So let’s start…

**Introduction**

Why we need OOPS?

Programming is nothing but solving complex real time problems, which will take a lot of manual effort. **So, key part of programming is to solve problem.**

In normal Procedural language we are able to do coding but there are certain restrictions:

1. You have to go through pre-defined procedures like you have to define variables only at the top and keep track of that. Why this can be a problem? Consider 2 million lines of code!! You can do it but it will become messy right?
2. Even you have to keep track of memory which you allocated for variables or say some structures etc. You will say we can handle it, but try to imagine whole big applications. Writing code is one thing but maintaining it, is totally a different story. A single memory leak or stack overflow or any memory issues can give you a nightmare.
3. When you think of building any application then slowly you will realize that it can be viewed as some independent component. For instance say “Playing a video” or doing online “Payment” or just maintain some kind of valid lookup information with id and values, which can be used at several places.
4. Sometimes you need modules to behave in a particular ways without looking into its implementations and reuse its code to other places.

Oops language solves all the above problems by providing modularity, robust and secure application. You will get know about all these things, as we deep dive into Oops concept by using some codes. So what all OOPS offers:

1. **Inheritance / Code Reusability**: If you separate common functionality of modules into separate module, then it can be reused by other modules, avoiding duplicacy. For example Authentication module, which can be used by different authentication modules like LDAP Authentication, Database based user authentication etc.
2. **Encapsulation**: We always want privacy and share only required information with different persons. You can package information / state and behaviour in one packet and then whoever needs it, can take permission and use it accordingly, without worrying about how it is done under the hood. All data inside packet are secured and cannot be directly accessed. Just like you cannot know the exact age of person without asking him. **Eg: You can store Name, Id, Address, Salary and Manager of an Employee in one packet and allow only Address to be editable field for Employee, rest information can only be updated by Administrator. If you will let Employee edit their salary then anyone can update their salary.**
3. **Abstraction**: This feature allows us to hide complex functionality of our application. Why we need it?? Suppose you do a Payment with PAYTM / TEZ, you are just interested in completing the payment by clicking on **“PAY”** button, and not the whole complex process of Payment process under the hood, right? **Now you are confused that this means Encapsulation and Abstraction are same because both hide complex functionality and you are absolutely correct!! Both are same but there intents are different**. Abstraction is used at design level and Encapsulation is the concrete implementations of that design.
4. **Polymorphism**: This feature gives us ability to behave differently as per the requirement. Eg: we can have one Payment Module which can be used to pay using Credit Card as well as Debit Card.

**Note: We are not saying here that problem cannot be solved using Procedural language like C but just trying to state that solutions will be easy if we use Oops concept. In some cases we cannot move away from Procedural languages, when we do low level programming (for instance chip level coding or software driver development).**

**So now let’s get our hand dirty by understanding above concepts using sample program. Don’t worry if you do not get few keywords because we will again revisit them.**

**First thing first, we will pen down our requirement:**

Abstract: Create a Payment Module supporting below modes of payment:

1. Credit Card
2. Debit Card
3. UPI

Now we will design a Payment System in such a way that a single entry point will process all payment modes. So it will look like below:

Client Payment Request

Payment

Credit Card

Payment

UPI

Payment

Debit Card

Payment

Payment Module

Payment

Let’s first create Payment Design, which can be used for Payment by Clients.

**package** oops.abstraction;  
  
**import** oops.inheritance.PaymentInfo;  
  
*/\*\*  
 \* Abstract Design of payment Module. Now This design will be implemented by different Payment Modules  
 \*/***public interface** Payment  
{  
 **boolean** doPayment(PaymentInfo paymentInfo);  
}

Now we need a packet to store common information required for payment like card number, expiry month, upi id etc.

Remember we store sensitive information in class, so now we are going to create a class with all these attributes, which can later be used by different Payment modes.

*/\*\*  
 \* Parent class with common Payment property packaged as Payment Info,  
 \* so now any subclass can extend and use it accordingly  
 \*/***public class** PaymentInfo {  
 **protected** Long **cardNumber**;  
 **protected int expiryMonth**;  
 **protected int expiryYear**;  
 **protected** String **upiID**;  
 **protected** String **nameOnCard**;  
 **protected** String **pin**;  
 **protected int otp**;  
  
 **public** PaymentInfo() {  
 }  
  
 **public** PaymentInfo(String upiID, String pin, **int** otp) {  
 **this**.**upiID** = upiID;  
 **this**.**pin** = pin;  
 **this**.**otp** = otp;  
 }  
  
 **public** PaymentInfo(Long cardNumber, **int** expiryMonth, **int** expiryYear, String nameOnCard, String pin, **int** otp) {  
 **this**.**cardNumber** = cardNumber;  
 **this**.**expiryMonth** = expiryMonth;  
 **this**.**expiryYear** = expiryYear;  
 **this**.**nameOnCard** = nameOnCard;  
 **this**.**pin** = pin;  
 **this**.**otp** = otp;  
 }

}

//Getter And Setters

Now we have design in place and common property, so let’s inherit common property from Payment Info and implement our Payment design for different Payment Modes.

*/\* Inheritance and Encapsulation Example  
\* Child class importing properties from Parent PaymentInfo and giving implementation of Payment Module Design  
\* \*/***public class** CreditCardPayment **extends** PaymentInfo **implements** Payment{  
  
 **public** CreditCardPayment() {  
 }  
  
 **public** CreditCardPayment(Long cardNumber, **int** expiryMonth, **int** expiryYear, String nameOnCard, String pin, **int** otp) {  
 **super**(cardNumber, expiryMonth, expiryYear, nameOnCard, pin, otp);  
 }  
 @Override  
 **public boolean** doPayment(PaymentInfo paymentInfo) {  
 System.***out***.println(**"doing Payment using credit card"**);  
 LocalDate date = LocalDate.*now*();  
  
 **int** year = date.getYear();  
 **int** month = date.getMonthValue();  
  
 **if**(paymentInfo.getExpiryYear()>=year && paymentInfo.getExpiryMonth() >=month)  
 {  
 System.***out***.println(paymentInfo);  
 System.***out***.println(**"Payment done successfully"**);  
 **return true**;  
 }  
 System.***out***.println(**"Payment unsuccessful :("**);  
 **return false**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "CreditCardPayment{"** +  
 **"cardNumber="** + **cardNumber** +  
 **", expiryMonth="** + **expiryMonth** +  
 **", expiryYear="** + **expiryYear** +  
 **", nameOnCard='"** + **nameOnCard** + **'\''** +  
 **", pin='"** + **pin** + **'\''** +  
 **", otp="** + **otp** +  
 **'}'**;  
 }  
}

*/\* Inheritance and Encapsulation Example  
\* Child class importing properties from Parent PaymentInfo and giving implementation of Payment Module Design  
\* \*/***public class** DebitCardPayment **extends** PaymentInfo **implements** Payment {  
  
 **public** DebitCardPayment() {  
 }  
  
 **public** DebitCardPayment(Long cardNumber, **int** expiryMonth, **int** expiryYear, String nameOnCard, String pin, **int** otp) {  
 **super**(cardNumber, expiryMonth, expiryYear, nameOnCard, pin, otp);  
 }  
  
 @Override  
 **public boolean** doPayment(PaymentInfo paymentInfo) {  
 System.***out***.println(**"doing Payment using debit card"**);  
 LocalDate date = LocalDate.*now*();  
 **int** year = date.getYear();  
 **int** month = date.getMonthValue();  
  
 **if**(paymentInfo.getExpiryYear()>=year && paymentInfo.getExpiryMonth() >=month)  
 {  
 System.***out***.println(paymentInfo);  
 System.***out***.println(**"Payment done successfully :)"**);  
 **return true**;  
 }  
 System.***out***.println(**"Payment unsuccessful :("**);  
 **return false**;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "DebitCardPayment{"** +  
 **"cardNumber="** + **cardNumber** +  
 **", expiryMonth="** + **expiryMonth** +  
 **", expiryYear="** + **expiryYear** +  
 **", nameOnCard='"** + **nameOnCard** + **'\''** +  
 **", pin='"** + **pin** + **'\''** +  
 **", otp="** + **otp** +  
 **'}'**;  
 }  
}

*/\* Inheritance and Encapsulation Example  
\* Child class importing properties from Parent PaymentInfo and giving implementation of Payment Module Design  
\* \*/***public class** UPIPayment **extends** PaymentInfo **implements** Payment {  
  
 **public** UPIPayment() {  
 }  
  
 **public** UPIPayment(String upiID, String pin, **int** otp) {  
 **super**(upiID, pin, otp);  
 }  
  
 @Override  
 **public boolean** doPayment(PaymentInfo paymentInfo) {  
 System.***out***.println(**"doing Payment using Upi"**);  
 **if**(paymentInfo.getUpiID().contains(**"@"**))  
 {  
 System.***out***.println(paymentInfo);  
 System.***out***.println(**"Payment Successful"**);  
 **return true**;  
 }  
 System.***out***.println(**"Payment unsuccessful :("**);  
 **return false**;  
 }  
  
  
 @Override  
 **public** String toString() {  
 **return "UPIPayment{"** +  
 **"upiID='"** + **upiID** + **'\''** +  
 **", pin='"** + **pin** + **'\''** +  
 **", otp="** + **otp** +  
 **'}'**;  
 }  
}

Above classes gives us implementation of each mode. So we are good from the payment processing. In real world scenario doPayment(PaymentInfo paymentinfo) method will call different bank APIs for respective payment mode. From above code you would have got the idea about Module design and implementation.

Now we have to create our Payment Request, which should contain PaymentInfo and Payment Type, using which we can get payment request information.

*/\*This is also example of Encapsulation, where we are packaging Payment information in Request  
\* and using it to pass to different services  
\*/***public class** PaymentRequest  
{  
 **private** PaymentInfo **paymentInfo**;  
 **private** String **paymentType**;  
  
 **public** PaymentRequest(PaymentInfo paymentInfo, String paymentType) {  
 **this**.**paymentInfo** = paymentInfo;  
 **this**.**paymentType** = paymentType;  
 }

//Getter Setter  
  
}

We also have to ensure about Payment type supported by our application and as of now we are only supporting three modes of payment, so we will create enum holding these values:

*/\*\*  
 \* Enum to define Payment Types supported by Application  
 \*/***public enum** PaymentType  
{  
 ***CC***(**"credit card"**),  
 ***DC***(**"debit card"**),  
 ***UPI***(**"upi"**);  
  
 **private** String **type**;  
  
 PaymentType(String type)  
 {  
 **this**.**type**=type;  
 }  
  
 **public** String getType() {  
 **return type**;  
 }  
}

Now we have to create a class which will give us Payment Object based on Payment type. In Programming world we call it as Factory Design Pattern. We will get into its detail in design pattern tutorial.

*/\*Object Factory Class to build and return Payment Object depending on Payment Type  
\* \*/***public class** PaymentFactory {  
  
 **public static** Payment getInstance(PaymentRequest paymentRequest)  
 {  
 PaymentInfo paymentInfo=paymentRequest.getPaymentInfo();  
 String paymentType=paymentRequest.getPaymentType();  
 **if**(PaymentType.***CC***.name().equalsIgnoreCase(paymentType))  
 {  
 **return new** CreditCardPayment(paymentInfo.getCardNumber(),paymentInfo.getExpiryMonth(),paymentInfo.getExpiryYear(),  
 paymentInfo.getNameOnCard(),paymentInfo.getPin(),paymentInfo.getOtp());  
 }  
 **else if**(PaymentType.***DC***.name().equalsIgnoreCase(paymentType))  
 {  
 **return new** DebitCardPayment(paymentInfo.getCardNumber(),paymentInfo.getExpiryMonth(),paymentInfo.getExpiryYear(),  
 paymentInfo.getNameOnCard(),paymentInfo.getPin(),paymentInfo.getOtp());  
 }  
 **else if**(PaymentType.***UPI***.name().equalsIgnoreCase(paymentType))  
 {  
 **return new** UPIPayment(paymentInfo.getUpiID(),paymentInfo.getPin(),paymentInfo.getOtp());  
 }  
  
 **throw new** UnsupportedOperationException(**"Payment type: "**+paymentType+**" not supported"**);  
 }  
}

Till now we have created Modules depicting Abstraction, Encapsulation and Inheritance. Now it’s time to show Polymorphism.

So now think that wouldn’t it will be elegant to create h single entry point which will handle and process any payment modes. For instance, if we use Debit Card mode then it will behave as Debit Card and if we use Credit Card then it behave as Credit Card. That’s what we call as Polymorphism, behaving differently based on requirement. Let’s create our Polymorphic module:

*/\* This represents Polymorphism feature as it process all payment types(Credit Card, Debit Card or UPI).  
\* Whatever the Payment Request it will process it.  
\* It is single entry point for all payment requests.  
\* \*/***public class** ProcessPayment {  
  
 **public boolean** paymentProcess(PaymentRequest paymentRequest)  
 {  
 Payment payment = PaymentFactory.*getInstance*(paymentRequest);  
 **return** payment.doPayment(paymentRequest.getPaymentInfo());  
 }  
}

It’s just two lines of code to do all this complex job, isn’t it great!!

So what it does? It calls Factory Class get instance method to get required payment Object and calls doPayment on it. Now whichever child class Object is returned, its corresponding doPayment method will be called.

So we are almost done. In real world scenario, request will be coming from some UI to your Rest API but since we don’t have that now, so we will create Client Driver class and write our request structure in simple main method with below operations:

1. Create valid credit card payment request
2. Create Invalid debit card payment request with wrong expiry year. It will make Payment unsuccessful
3. Create valid UPI payment request
4. Create invalid payment type request with UNKNOWN payment type

*/\*  
\* Client to send Payment request. In our case it is main method but in real world this request will be coming from  
\* Web UI or Android App where user gives input while doing payment\*/***public class** PaymentDriver  
{  
 **public static void** main(String [] args)  
 {  
 ProcessPayment processPayment = **new** ProcessPayment();  
  
 *//Initiating transaction with valid payment request 1* processPayment.paymentProcess(*creditCardPaymentRequest*());  
  
 *//Initiating transaction with invalid payment request 2* processPayment.paymentProcess(*debitCardPaymentRequest*());  
  
 *//Initiating transaction with valid payment request 3* processPayment.paymentProcess(*upiPaymentRequest*());  
  
 *//Initiating transaction with invalid payment type request 4* **try** {  
 processPayment.paymentProcess(*unSupportedPaymentRequest*());  
 }  
 **catch** (Exception e)  
 {  
 System.***out***.println(**"Error: while processing payment: "**+e.getMessage());  
 }  
 }  
  
 *// Creating valid credit card payment request* **public static** PaymentRequest creditCardPaymentRequest()  
 {  
 CreditCardPayment creditCardPayment = **new** CreditCardPayment(123456789112345L,1,  
 2020,**"Test User1"**,**"123"**,2435);  
  
 **return new** PaymentRequest(creditCardPayment, PaymentType.***CC***.name());  
 }  
  
 *// Creating Invalid debit card payment request with wrong expiry year. It will make Payment unsuccessful* **public static** PaymentRequest debitCardPaymentRequest()  
 {  
 DebitCardPayment debitCardPayment= **new** DebitCardPayment(123456789112345L,1,  
 2019,**"Test User1"**,**"123"**,2435);  
  
 **return new** PaymentRequest(debitCardPayment, PaymentType.***DC***.name());  
 }  
  
 *// Creating valid UPI payment request* **public static** PaymentRequest upiPaymentRequest()  
 {  
 UPIPayment upiPayment = **new** UPIPayment(**"TestUser1@upi"**,**"123"**,2435);  
  
 **return new** PaymentRequest(upiPayment, PaymentType.***UPI***.name());  
 }  
  
 *// Creating invalid payment type request* **public static** PaymentRequest unSupportedPaymentRequest()  
 {  
 PaymentInfo paymentInfo= **new** PaymentInfo();  
  
 **return new** PaymentRequest(paymentInfo, **"UNKNOWN"**);  
 }  
}

Now you must be thinking what Class, Object and enum is?

**Class**: It is a template to create Object and it does not exist physically i.e. it does not take any memory in your computer.

**Object**: It exists physically in heap memory area and holds information as described in Class and provided by User.

**Enum**: It is used to represent group of named constants. Java Enums are more powerful Compared to other languages as we can add constructor, variables and methods to it. In our enum you can see that we have added type description for all Payment Type.

We use Enums when we know all possible values of Constant. Like in our case we know that we are going to support only 3 types of Pyament Mode as of now.

We will dive into more details as we discuss these topics in our subsequent tutorials.

So In this blog we learnt about Oops concept through real life Project scenario. We have also learnt few basic design principles, like creating factory for multiple Object creation. Remember, whenever you are creating any module, and then think about properties it will hold and can it be shared among different modules? Always try to decouple your code dependency, so that in future you can add more features with minimal code change. We mentioned it earlier and again reiterating the same thing, writing code is one thing but maintaining it is a totally different story.

In our application, say tomorrow we need to add AmazonPay or Paytm as our Payment modes then we just need to 3 file changes:

1. Create a new class by inheriting PaymentInfo and implementing doPayment method from Payment interface.
2. Add new payment type in our enum
3. Enhance getinstance method to return new Payment Mode Object and we are done.

You can access source code from Github by clicking [here](https://github.com/amberdas/aktutorials/tree/master/oops).

This is how Oops concept makes our life easy. We will see some more concepts in next tutorials.

* Amber Kumar Das