

While folks are joining

Get you laptops ready and login to your **replit** accounts.

We will be coding away in the session!

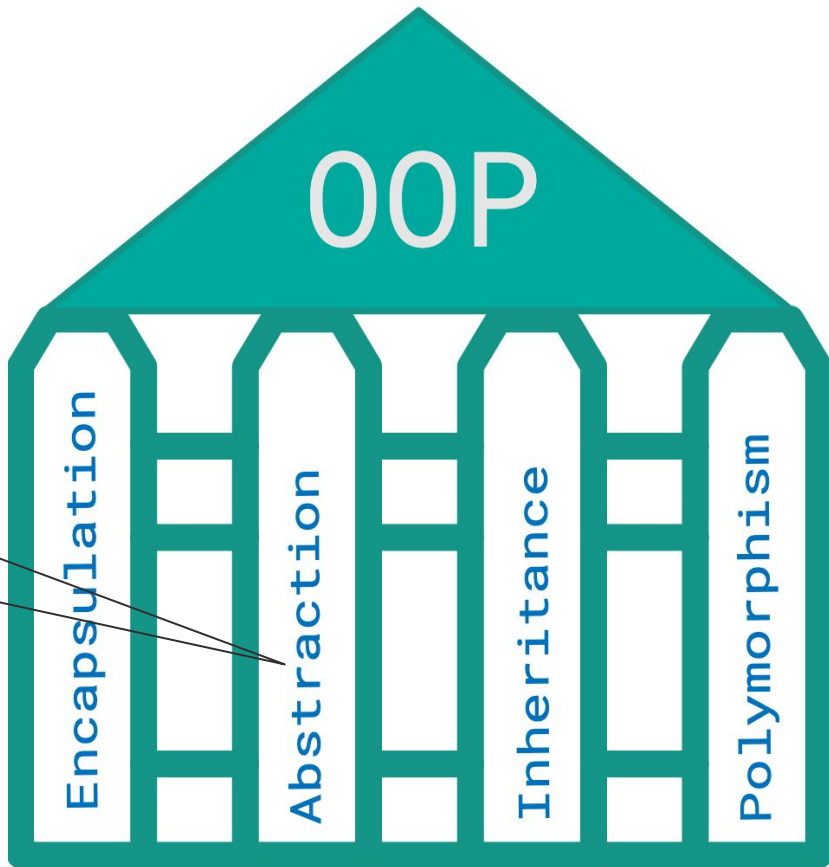


Crio Sprint : DEV-OOP-1

Session 3



Four Pillars of OOP



Will be Discussed
Today!



What is Abstraction?

- Hide Implementation Complexity
- **Expose only relevant functionality** to the user.
- Focus on ***what the*** object does instead of ***how it*** does it.
- For eg, A computer can connect to a network using (Ethernet, Wi-Fi, dial-up modem, etc.)
 - Web Browser doesn't care which one is being used.
 - **"Connection to the network"** is the **abstraction**.
 - **Ethernet** and **Wi-Fi** are **implementations** which enables connection.

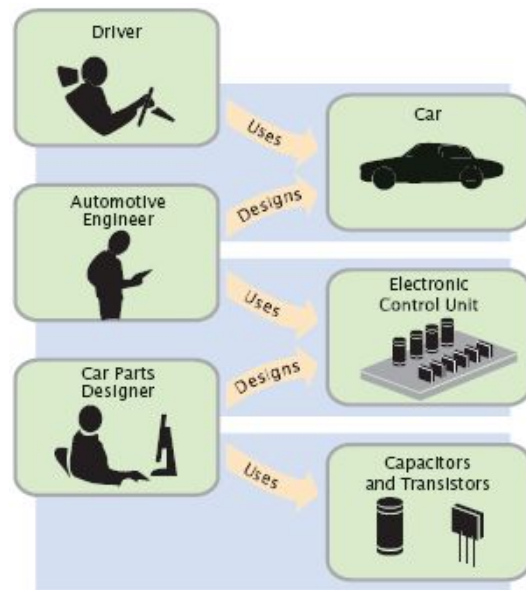


Figure 1

Levels of Abstraction in Automotive Design

In simple words, **Abstraction** is the practice of **breaking a large problem** into **smaller components**, so **each smaller problem** can be **worked on in (relative) isolation**.



Abstraction - Scenario #1 ATM

- What operations can be done at an ATM?
 - Cash withdrawal
 - Mini statement
 - Change password, etc.
- List down the steps for cash withdrawal
 - Enter the Card
 - Input the Pin
 - Input Amount for cash withdrawal
 - Validate Amount (whether account has balance)
 - Deduct Amount from Account
 - Dispense Cash



Activity #1 - ATM Machine

```
class ATMMachine {
    public void enterCard(){
        System.out.println ("Card Verification");
    }
    public void enterPin(){
        System.out.println ("Pin Verification");
    }
    public void cashWithdrawal (){
        System.out.println ("To withdraw cash from ATM");
    }
    public void validateWithdrawAmount(){
        System.out.println ("Validate the Amount to be withdrawn");
    }
    public void updateAmount(){
        System.out.println ("Update the Amount after withdrawal");
    }
    public void cashDispense(){
        System.out.println ("Dispense the cash from ATM");
    }
    public void miniStatement () {
        System.out.println ("Get the mini statement");
    }
}
```

Compile and Run the below program.

```
public class Main{
    // Mimic user behavior
    public static void main (String[] args){
        ATMMachine am = new ATMMachine();
        am.enterCard();
        am.enterPin();
        am.cashWithdrawal();
        am.validateWithdrawAmount();
        am.updateAmount();
        am.cashDispense();
    }
}
```

- Look at the output. Does it make sense?
- Remove the below method and run again.
 - validateWithdrawAmount()
 - updateAmount()
- Does it make sense?
- Where should these removed methods be invoked?



Activity #1.1 - ATM Machine

```
class ATMMachine{
    public void enterCard (){
        System.out.println ("Card Verification");
    }
    public void enterPin (){
        System.out.println ("Pin Verification");
    }
    public void cashWithdrawal(){
        System.out.println ("To withdraw cash from ATM");
        validateWithdrawAmount();
        updateAmount();
        cashDispense();
    }
    private void validateWithdrawAmount(){
        System.out.println ("Validate the Amount to be withdrawn");
    }
    private void updateAmount(){
        System.out.println ("Update the Amount after withdrawal");
    }
    private void cashDispense (){
        System.out.println ("Dispense the cash from ATM");
    }
    public void miniStatement (){
        System.out.println ("Get the mini statement");
    }
}
```

Run the program again.

```
public class Main{
    public static void main (String[]args){
        ATMMachine am = new ATMMachine();
        am.enterCard();
        am.enterPin();
        am.cashWithdrawal();
    }
}
```

- Look at the output. Does it make sense?
- What did we accomplish?



Other Ways to Achieve Abstraction

- **abstract class** (0% - 100% Abstraction)
- **interface** (100% Abstraction)



Activity #2 - Payroll

- Your manager has asked you to develop a payroll program for a company.
- The company has two groups of employees
 - **Full-time employees**
 - Gets paid a fixed salary
 - **Hourly Employees**
 - Gets paid by hourly wages
- The program prints out a payroll that includes employee names and their monthly salaries.
- To model the payroll program in an object-oriented way, we come up with the following classes:
 - Employee
 - FulltimeEmployee
 - HourlyEmployee
 - and Payroll
- You provided this solution
 - [Payroll-Java](#)
- Your manager didn't like what you did with the *Employee* class. Can you identify the issue?
- How can we resolve the issue?
 - Using *abstract* methods and *abstract* classes.

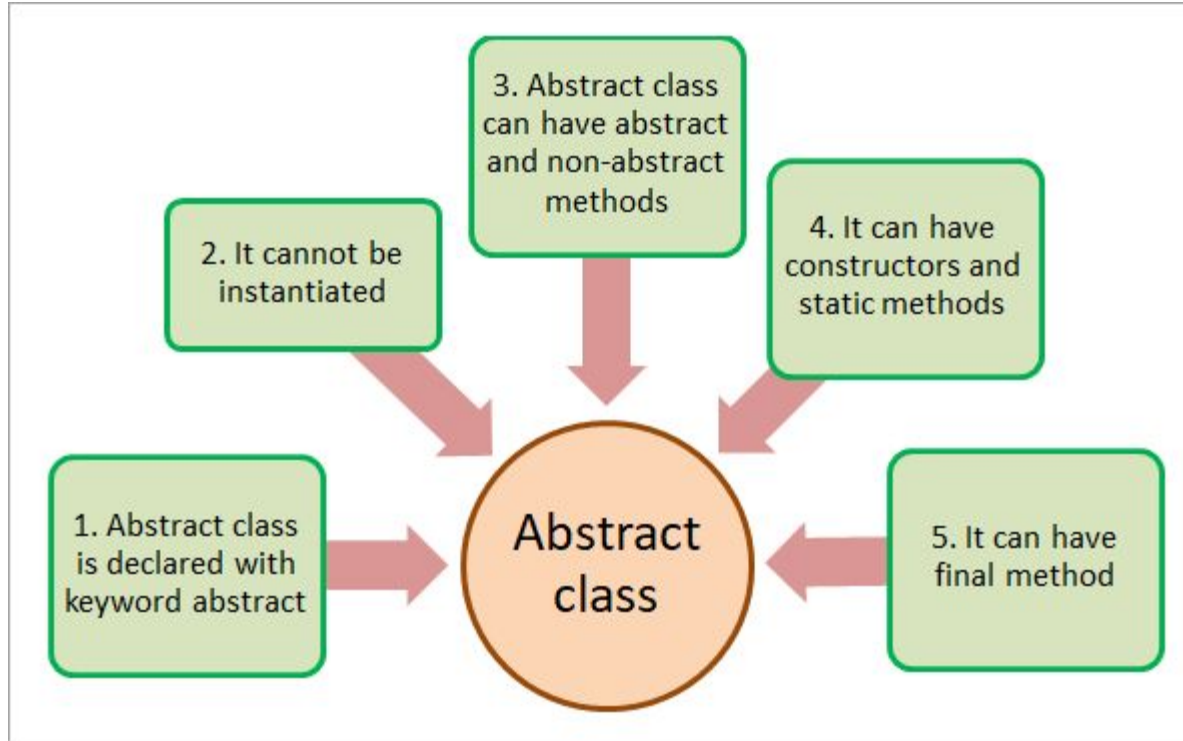


Activity #2 - Payroll

- Improved Solution
 - [Payroll Improved](#)
- Real-World Analogy
 - **Abstract class** is similar to a **Car Platform** where minimum skeleton is provided.
 - The base platform provides bare minimum functionality common between all the cars.
 - Different types of car models can be manufactured on top of car platform.
 - **Important to Note:-** This skeleton is only useful if it is fully developed into a car.



Abstract class in Java (For Reference)



Activity #2 - Payroll: Python Implementation

[main.py - Payroll - Replit](#)

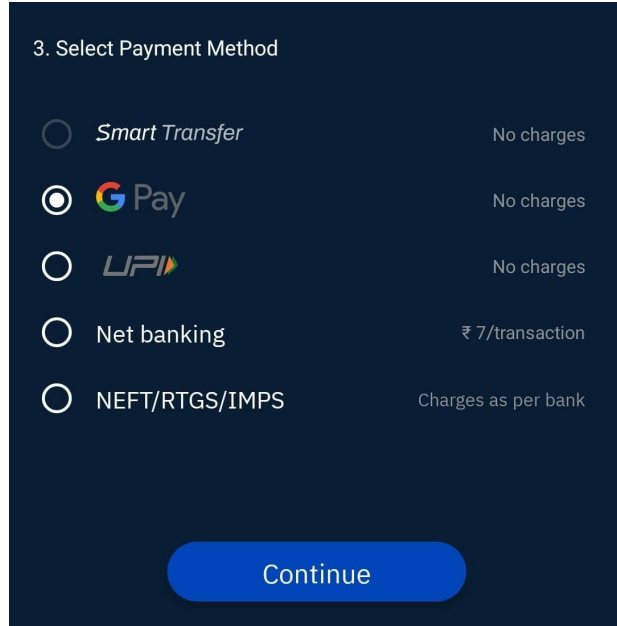


5 minute break



Activity #3 - Payment Option

- Have you ever paid your credit card bill?
- What are the different payment options available?



3. Select Payment Method

☐ Smart Transfer No charges

☒ G Pay No charges

☐ UPI No charges

☐ Net banking ₹ 7/transaction

☐ NEFT/RTGS/IMPS Charges as per bank

Continue

- Compile and run the below program.
 - [Payment Option](#)
- Can you support 100 more payment methods?
- What is the drawback of the current approach?
- How can we improve it?
 - Using Interfaces



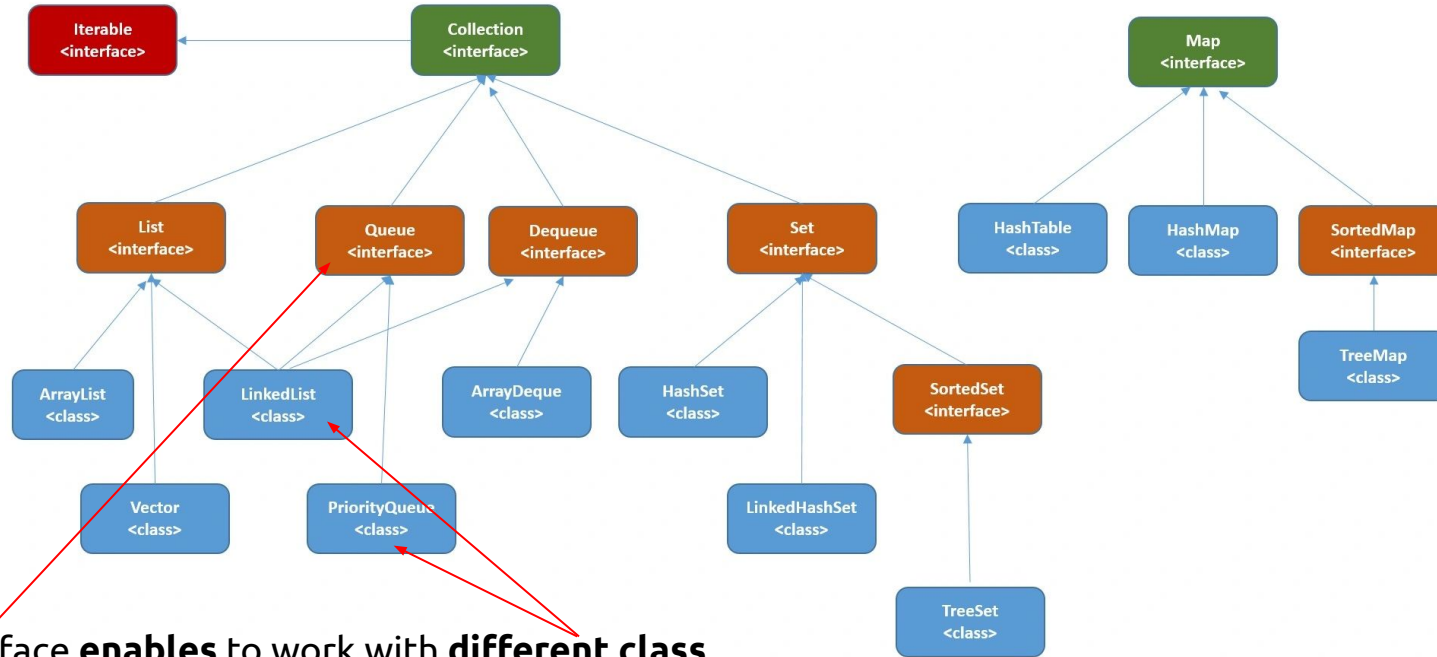
Activity 3.1 - Payment Option

- Improved Solution - [Payment Option using Interfaces](#)
 - Can we accommodate as many payment methods as possible easily?
 - What did we achieve here?
 - Total abstraction of Payment Options from CreditCard class.
 - Loose coupling using interface.
- Real-World Analogy
 - **Interface** is similar to **USB-C** on smartphones.
 - **USB-C** has a standard specification which charger manufacturers must comply.
 - Smartphone can be charged regardless of the brand who manufactured as long as it honours the specification.



Application of Interfaces in Java Collection

Collection Framework Hierarchy



Interface **enables** to work with **different class types** even if they are **not related** to each other.



Curious Cats



- Can we achieve multiple inheritance in Java?
 - Not possible with classes.
 - But,



Multiple Inheritance in Java

When to use abstract class or interfaces?

- [When to Use Abstract Class and Interface - DZone Java](#)

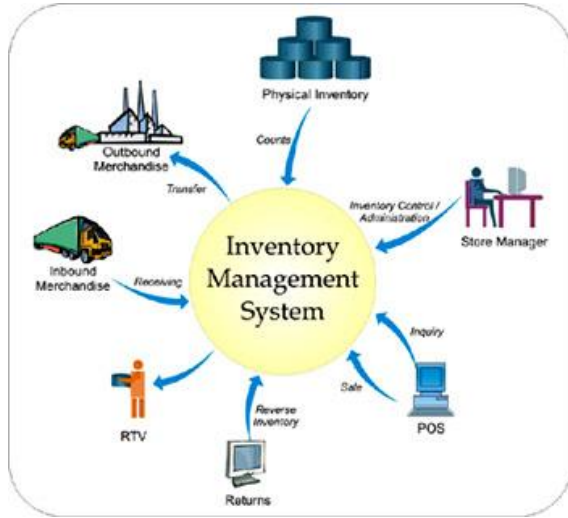


Abstraction Byte Overview

Messaging Application



Products/Teams in an eCommerce Company



Inventory management refers to the process of ordering, storing, and using a company's inventory. These include products as well as warehousing and processing such items.



Vendor management system allows us to take appropriate measures for controlling cost, reducing potential risks related to vendors, ensuring excellent service deliverability and deriving value from vendors.



Logistics management includes customer service, scheduling, packaging, and delivery. It functions across these dimensions - strategic, operational and tactical



Configuration needed for Products for flexibility

Vendor credit limit
Vendor minimum rating
Maximum vendors per product

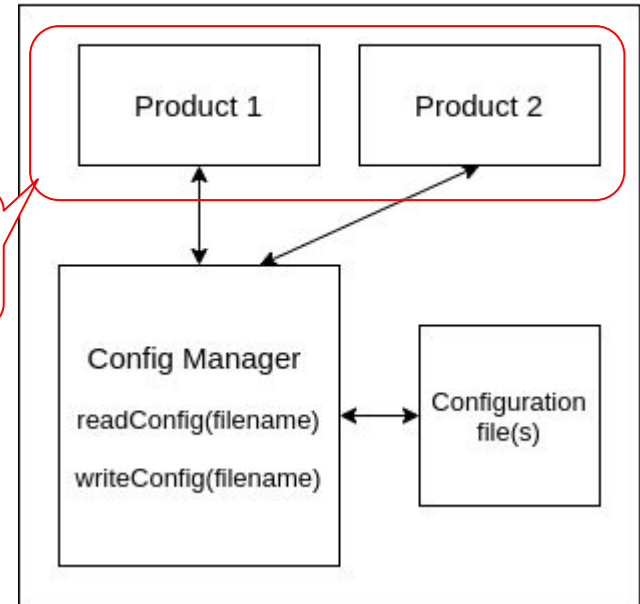
Current allowable discount
Delivery vendor list
Warehouse list

Storage capacity
Maximum pending orders
Default re-order timer

```
{  
  "credit_limit": 100000,  
  "min_rating": 2,  
  "max_vendors_per_product": 10  
}
```

Vendor Management
Logistics Management
Inventory Management

Product(s) can read or write their configuration from/to a file using the Config Manager



New Requirements

Config Manager initially only supports JSON file format. There is a new requirement to support XML File Format.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <credit_limit>100000</credit_limit>
  <min_rating>100</min_rating>
  <max_vendors_per_product>3306</max_vendors_per_product>
</configuration>
```

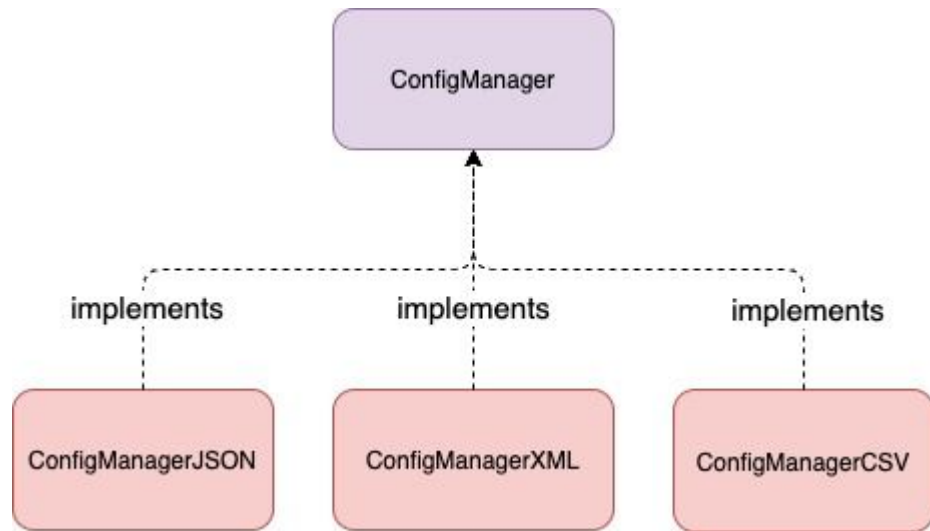


Possible Solution

- You have currently two methods in ConfigManager.java
 - readConfig
 - writeConfig
- Add two new methods to support XML file format
 - readConfigXML
 - writeConfigXML
- Rename the previous methods supporting JSON file format
 - readConfigJSON
 - writeConfigJSON
- What's the issue with the above approach?
 - Adding support for another file type will involve creating even more methods.
 - It no longer serves a single purpose.
 - The Clients using the current methods in multiple places will be affected.
 - Not extensible code



Abstraction Based Solution



Why is this solution better?

- Define common interface which is less likely to change
- Client is unaffected by the internal implementations
- Support multiple implementations for same behaviour
- Can switch between implementations easily



Take home exercises for the session

- You will explore **Abstraction** with this real world scenario in the following:
- For Java Learners
 - [Abstraction Byte : Java - Crio.do](#)
- For Python Learners
 - [Abstraction Byte : Python - Crio.do](#)



Feedback

Thank you for joining in today.

We'd love to hear your thoughts and feedback.



Further Reading

- [Abstraction in Java | Abstract Class, Example - Sciencetech Easy](#)
- [Lab Exercise: Abstraction and Encapsulation](#)
- [What Are Abstractions in Software Engineering with Examples \(thevaluable.dev\)](#)



Thank you

