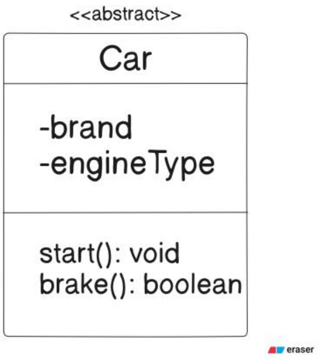
- LLD has 3 key features
  - Scalability
  - Maintainability (easy to debug)
  - Re-usability (loosely coupled)
- HLD:
  - Tech stack
  - DB (relational/non-relational/both)
  - Server scale
  - Cost optimization
- In simple terms:
  - HLD: System Architecture
  - LLD: Code Architecture
  - DSA: Algorithms
- Abstraction
  - It is to hide the methods which are not required to the client
  - The client can see the methods (non-vulnerable) but it doesn't have to know it to use the class.
  - Ex: To drive a Car, the driver doesn't need to know how the Engine works.
- Encapsulation
  - Write all the characteristics (variables) and behaviours (methods) in a capsule.
  - Here **data security** comes into picture, where properly the things are hidden that the client shouldn't see.
  - **getters** and **setters** are highly preferable.
- Polymorphism
  - **Dynamic** (or Run-time) polymorphism: Method Overriding
  - **Static** (or Compile-time) polymorphism: Method Overloading

➢ ----- UML Diagram -----

- It has 2 parts:
  - Structural
    - Static structure
    - Components & their links
    - Ex: **Class diagram**
  - Behavioral
    - Dynamic structure
    - Interaction between objects
    - Ex: **Sequence Diagram**

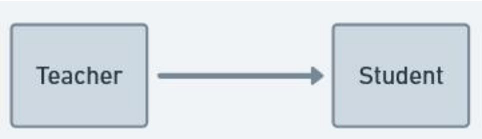
➢ Class diagram

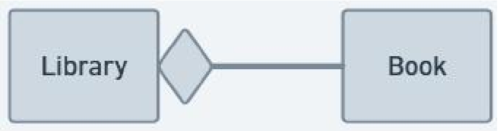
- **Diagram rules**
  - Class is represented in vertical rectangle having 3 parts
    - Top: class name
    - Middle: characteristics (variables)
    - Bottom: behaviours (methods)
  - To represent access modifiers:
    - Public : **+**
    - Private: **-**
    - Protected: **#**
  - If the class is abstract, then write **<<abstract>>** on the top of the rectangle.

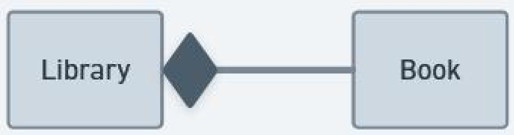


➢ Associations

- Types:
  - It is of 2 types:
    - Class Association
    - Object Association
  - **Class Association**
    - Inheritance
  - **Object Association**
    - Simple Association
    - Aggregation
    - Composition
- **Class Association**
  - **"is-a"** relationship
  - Let say, *Human* is inheriting *Animal* class: we can say *"Human* ***is-a*** *Animal"*

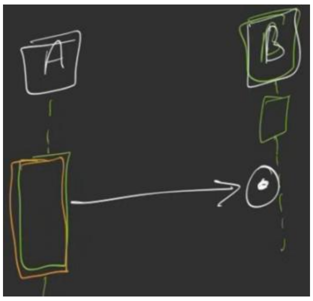
- **Object Association**
  - **Simple Association**
    - Definition: Basic relationship where 2 objects are connected; but **exist independently**.
    - Ownership: No ownership; neither controls life-cycle of others;
    - Example: *Teacher* associate with *Student*; both exist independently
    - Key point: Objects can exist without each other; the relationship just a link or usage.
    - Symbol: ➞ or ⟶ (one to another)
    - 

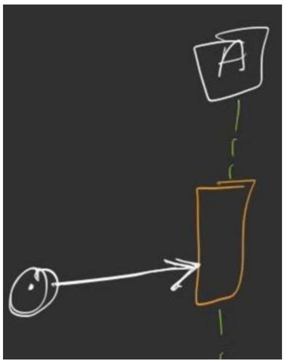
  - **Aggregation**
    - Definition: weak “**has-a**” relationship; one object (whole) contains/uses another object (part) but the part object can exist independently.
    - Ownership: **whole *owns* part**; but part’s lifecycle is independent.
    - Example: Library (whole) and Book (part); Book can exist independently
    - Key point: shared ownership; contained object (part) can exist independently of the container.
    - Symbol: ◇— (diamond head; not filled)
      - diamond head will be towards *the whole*
    - 

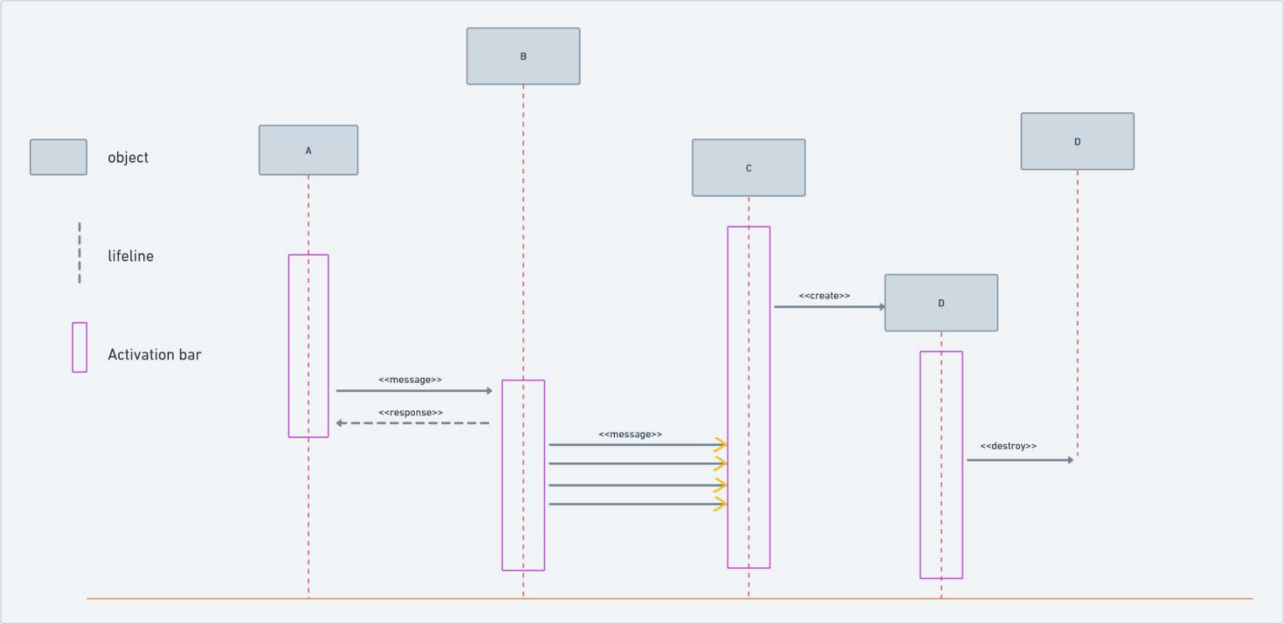
  - **Composition**
    - Definition: strong “**part-of**” relationship; one object (whole) exclusively ***owns*** other object (part)
    - Ownership: *the part*’s lifecycle is tied to *the whole*. If *the whole* is destroyed, *the part* will also be destroyed.
    - Example: *House* (whole) and *Room* (part); without House, Room will not exist.
    - Key Point: exclusive ownership; strong dependency;
    - Symbol: ◆— (diamond head; filled)
      - diamond head will be towards *the whole*
    - 

  - practically, only Composition exists;
  - in simple terms:
    - **Simple Association**: both **actually exists** independently. Both are **just linked**.
    - **Aggregation**: *the part* **can exist** independently; *the whole* **contains** *the part*
    - **Composition**: *the part* **cannot exist** independently; *the whole* is **made up of** *the part*.

Sequence diagram

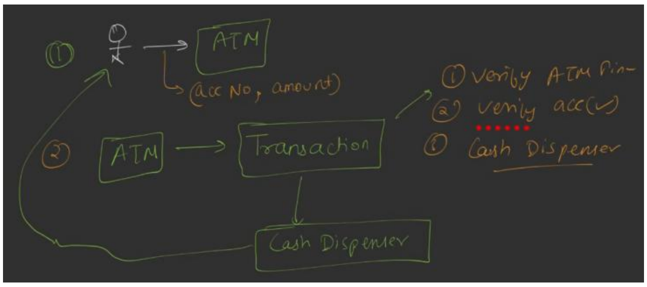
- Representation:
  - No need to represent a class with 3-level structure unlike Class diagram; just write class name inside a block.
  - Use arrow (one for unidirectional, 2 for bidirectional) to represent the relationship;
- Lifeline
  - Vertical line will be there for each object that represent how long the object is needed and then destroyed.
- Activation bar
  - Lifeline defines how long an object will exist; Activation bar represents how long an object will remain active.
- Messages
  - According to wait/not-wait
    - Async: send messages repeatedly without waiting for response.
      - Send message: solid **opened** arrow with <<message>>
    - Sync: send one message and wait for its response.
      - Send message: solid **closed** arrow with <<message>>
      - Response: dashed arrow with <<response>>
  - According to lifeline 2 types of messages:
    - **Create** : to create a object (start lifeline): solid **closed** arrow with <<create>>
    - **Destroy** : to destroy a object (end lifeline): solid **closed** arrow with <<destroy>>
  - According to reach-ability, 2 types of messages: dot inside a circle with arrow
    - **Lost message**: the message couldn't reach the destination
      - The destination object was inactive while sending message
      - 

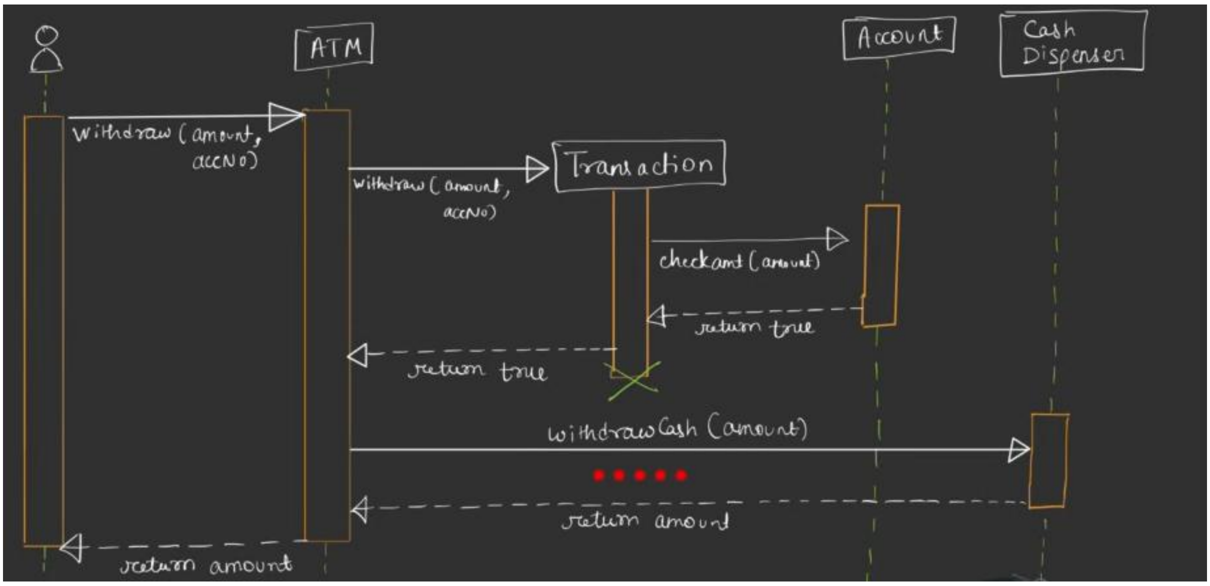
    - **Found message**: the sender of a message is lost.
      - The source object became inactive after sending message.
      - 



Made with Whimsical

➢

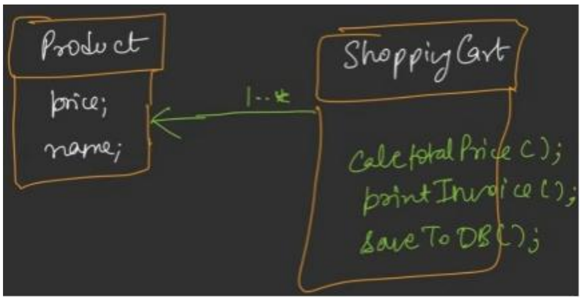
- Consider the below requirement
  - Use case
    - 

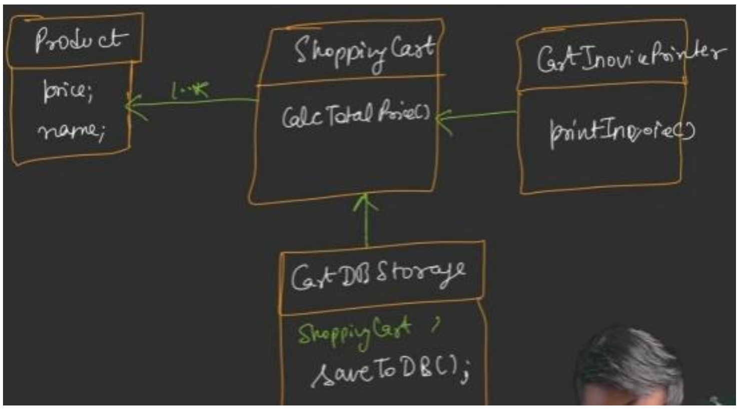
    - 1. User will go to ATM with *account number, amount*
    - 2. ATM will call transaction; it'll do 3 checks: *verifyAtmPin, verifyAccount, cashDispenser*
    - 3. User will get the money.
  - Objects
    - Atm
    - User
    - Transaction
    - Account
    - Dispenser
  - Sequence Diagram
    - 

      (ignore those red dots)
    - Here, withdraw and return amount are synchronous <<message>> and <<response>> (from user to ATM)
    - ATM & Transaction
      - ATM to Transaction: **sync** <<create>> <<message>>
      - Transaction to ATM: **sync** <<response>>
    - Transaction & Account
      - Transaction to Account: **sync** <<message>>
      - Account to Transaction: **sync <<destroy>>**
    - ATM & Cash Dispenser
      - **Sync** <<message>> **&** <<response>>

----- SOLID -----

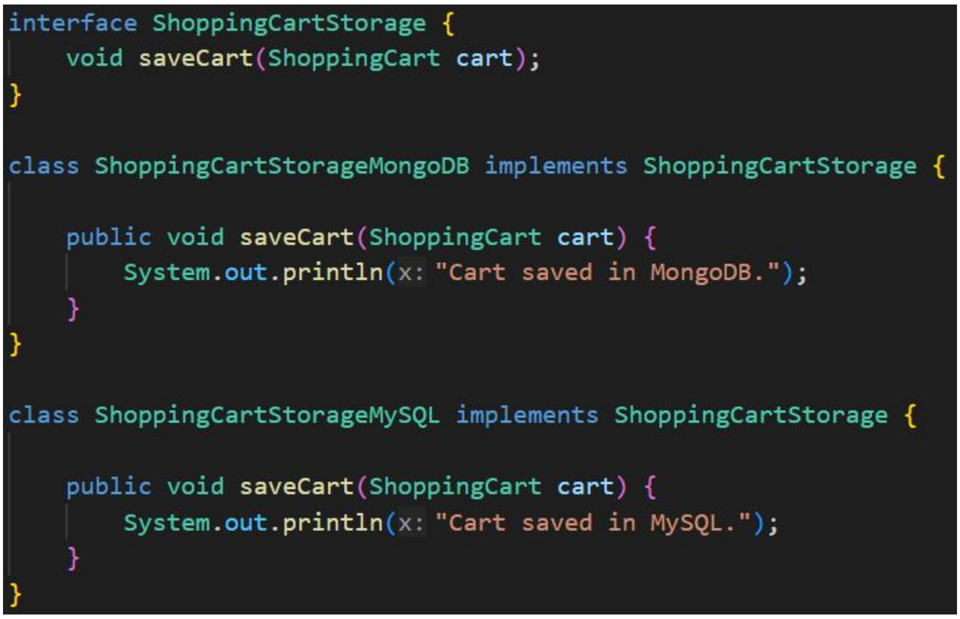
- SOLID
  - **Single Responsibility Principle (SRP)**
  - **Open-Close Principle (OCP)**
  - **Liskov Substitution Principle (LSP)**
  - **Interface Segregation Principle (ISP)**
  - **Dependency Inversion Principle (DIP)**

- **SRP (Single Responsibility Principle)**
  - One class should do only one thing or handle only one responsibility
  - *It doesn't mean, one class should have one method only; it can have multiple methods but all those should do a single type of work.*
  - 

    - Here, *ShoppingCart* has too many responsibility like calculate total price, print invoice, save to DB.
    - If we want to change the DB saving mechanism, we'll have to change this again which is not correct.
  - 

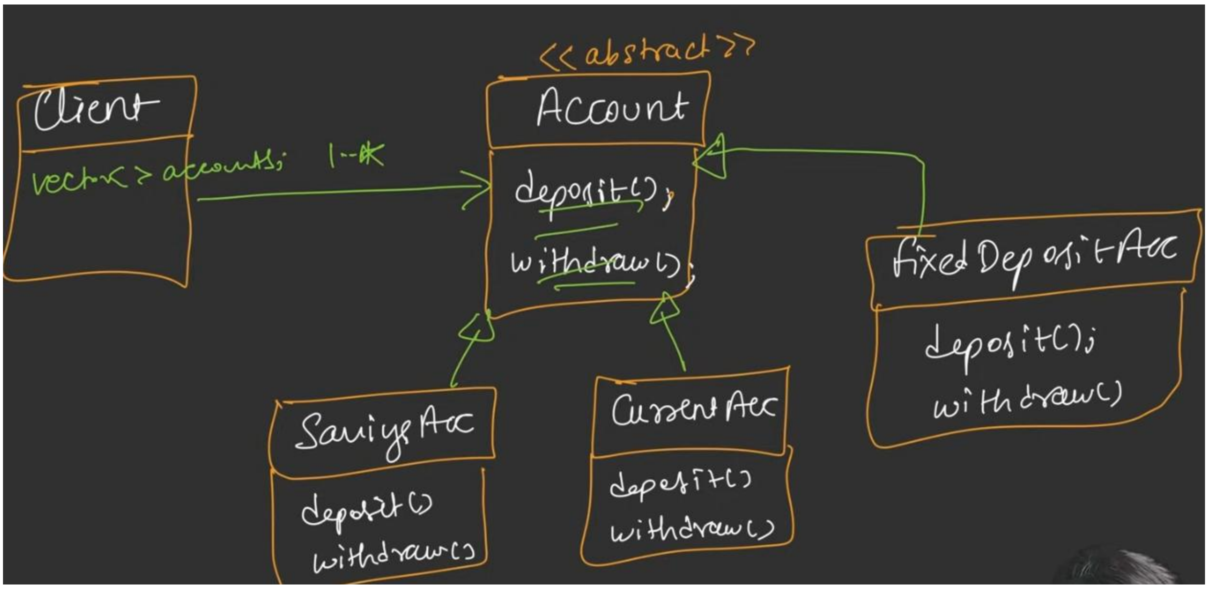
    - Here every class contains their own set of responsibility.
    - it is having "has-a" relationship.

- **OCP (Open-Close Principle)**
  - A class should be open for extension, but close for modification.
  - Use interface to define the methods, and use concrete classes implementing those interfaces to define those methods.
  - Use variable of type Interface and store objects of type Concrete class.
  - For example: DBStorage is a interface; DBStorageSQL, DBStorageMongoDB can be the concrete classes.

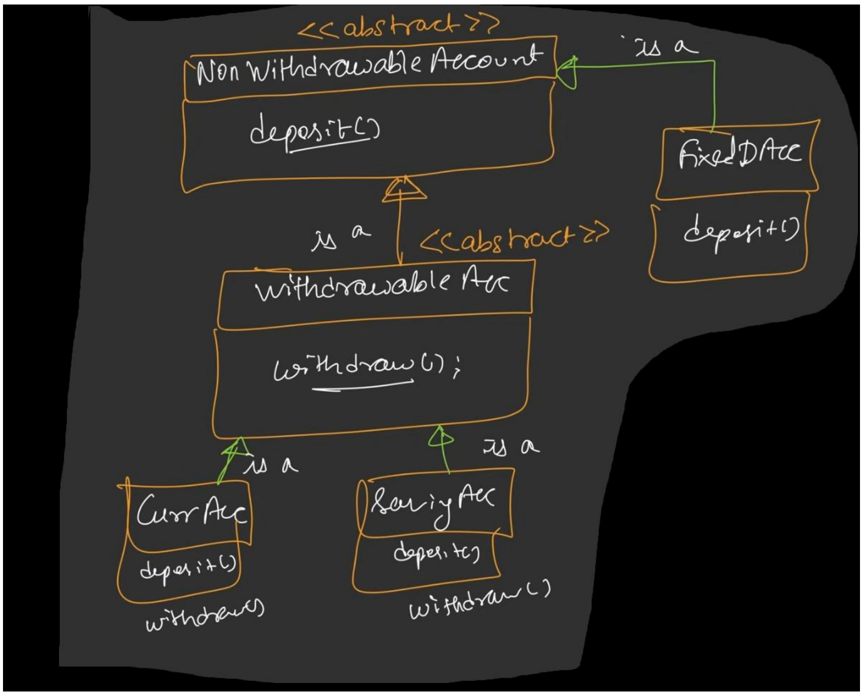


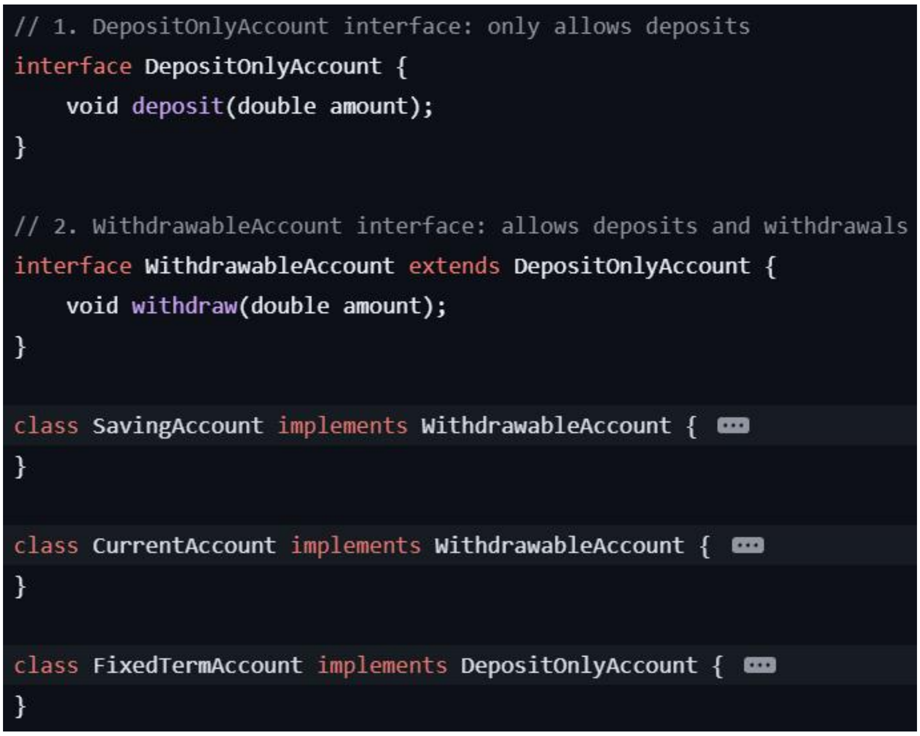
```java
interface ShoppingCartStorage {
    void saveCart(ShoppingCart cart);
}

class ShoppingCartStorageMongoDB implements ShoppingCartStorage {

    public void saveCart(ShoppingCart cart) {
        System.out.println(x: "Cart saved in MongoDB.");
    }
}

class ShoppingCartStorageMySQL implements ShoppingCartStorage {

    public void saveCart(ShoppingCart cart) {
        System.out.println(x: "Cart saved in MySQL.");
    }
}
```

- **LSP (Liskov Substitution Principle)**
  - *Sub classes* should be substitutable for their *Base classes.*
  - Sub class: Child class;   Base class: Parent class
  - Child class must contain all the methods present in Parent class/interface; Child class can have more methods but not less.
  - For example:



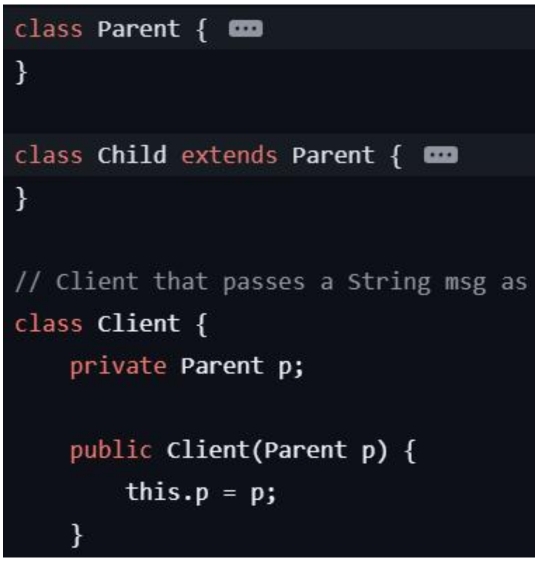
    - Here, FixedDepositeAccount cannot have *withdraw( )* method, so normally we ***throw an exception*** in this method;
    - but client doesn't know that withdraw() will give exception as it should be supported.
  - The below is the solution to this; there should be levels of the type of accounts.





```java
// 1. DepositOnlyAccount interface: only allows deposits
interface DepositOnlyAccount {
    void deposit(double amount);
}

// 2. WithdrawableAccount interface: allows deposits and withdrawals
interface WithdrawableAccount extends DepositOnlyAccount {
    void withdraw(double amount);
}

class SavingAccount implements WithdrawableAccount { ⋯
}

class CurrentAccount implements WithdrawableAccount { ⋯
}

class FixedTermAccount implements DepositOnlyAccount { ⋯
}
```

- **Rules**
- **Signature Rule** (**broader**: parent of any level, **narrow**: child of any level)
  - **Method Argument Rule**
    - A subclass *must not* **narrow** method parameters; it may accept the **same** or **broader** types.
      - Anyways it is not supported in any OOP language; you must write the same argument type as that of super class in case of method overriding.
    - Better to use a parent interface/class type as the method argument, so that whenever the object is passed, it can be either parent or child.

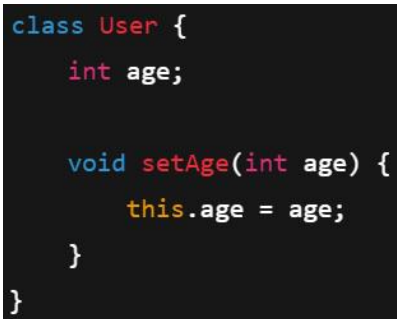


```
class Parent { ⋯
}

class Child extends Parent { ⋯
}

// Client that passes a String msg as
class Client {
    private Parent p;

    public Client(Parent p) {
        this.p = p;
    }
```

- **Return Type Rule**
  - An overridden method *may return* the **same** or a **narrower** type.
    - It is also not supported in most OOP language. The return type should be same as its super class method in case of overriding.
- **Exception Rule**
  - A subclass *may throw* the **same** or a **narrower** checked exception; *unchecked exceptions are unrestricted.*
- One line rule: **Parameters → wider**, **Return → narrower**, **Exceptions → narrower**

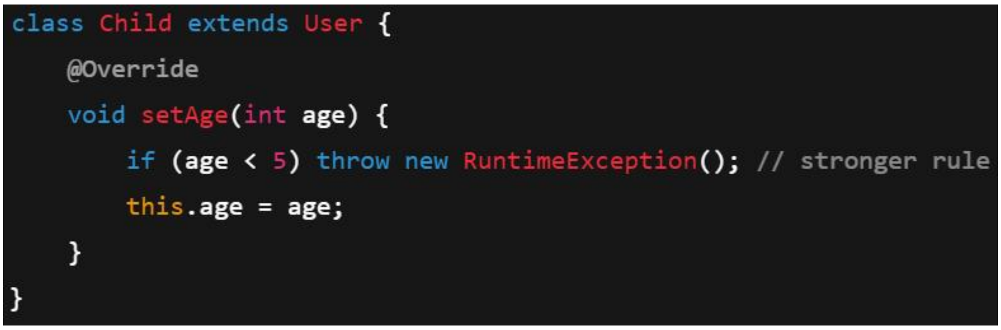
Property Rule

- **Class Invariant**
  - A subclass must preserve ALL invariants of its superclass.
  - Invariant is the custom rules for a class that it must follow.
  - Parent class:

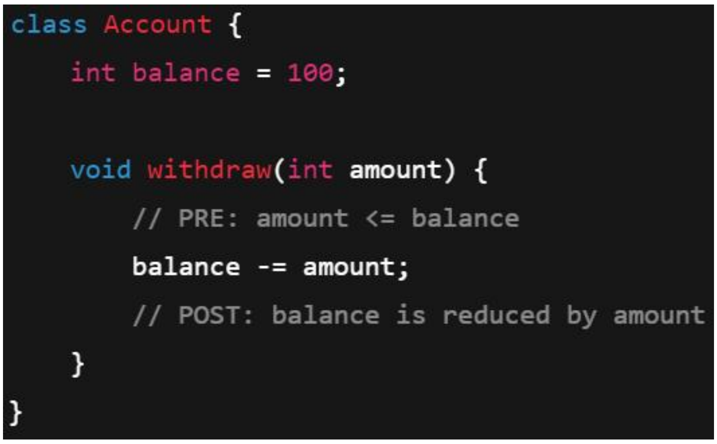


```
class User {
    int age;

    void setAge(int age) {
        this.age = age;
    }
}
```

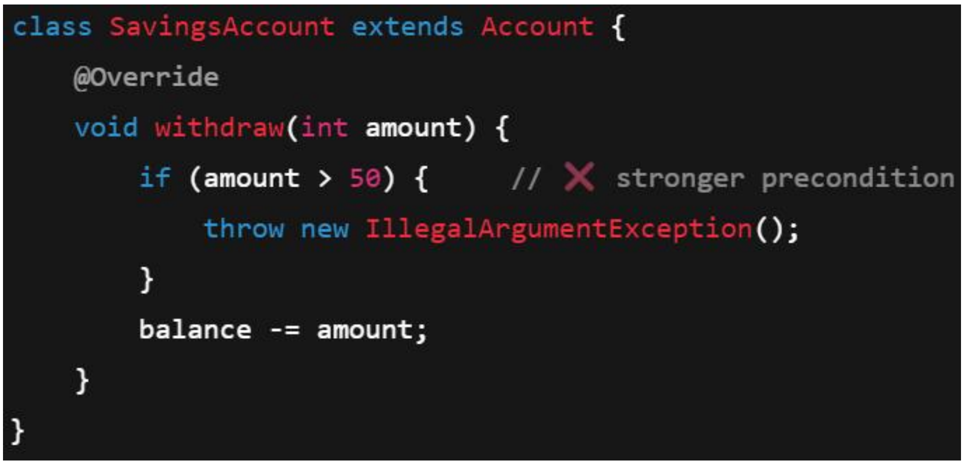
  - It requires the invariant **age >= 0**
- Child class:



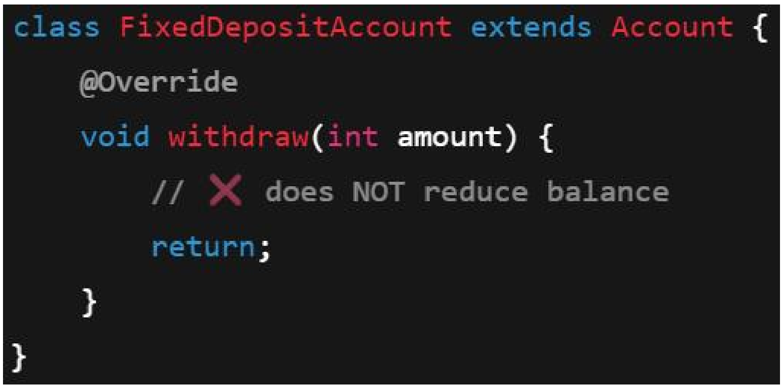
```
class Child extends User {
    @Override
    void setAge(int age) {
        if (age < 5) throw new RuntimeException(); // stronger rule
        this.age = age;
    }
}
```

        - It demands **age >= 5** (broke parent's invariant rule)
    - **History Constraint**
        - Subclass must allow what parent allows, and must not allow what parent forbids.
- **Method Rules**
    - **Precondition**
        - Precondition means "What must be true before a method is called"
        - A subclass must **NOT strengthen** preconditions.
        - *Example:* Lets say one password validator method of parent class requires the minimum length should be *8*; its child class's method can make its minimum length less than 8, but not greater.
    - **Postcondition**
        - Postcondition means "What is guaranteed after the method finishes"
        - A subclass must **NOT weaken** postconditions.
    - 

```
class Account {
    int balance = 100;

    void withdraw(int amount) {
        // PRE: amount <= balance
        balance -= amount;
        // POST: balance is reduced by amount
    }
}
```

    -  (pre)

```
class SavingsAccount extends Account {
    @Override
    void withdraw(int amount) {
        if (amount > 50) {      // ❌ stronger precondition
            throw new IllegalArgumentException();
        }
        balance -= amount;
    }
}
```

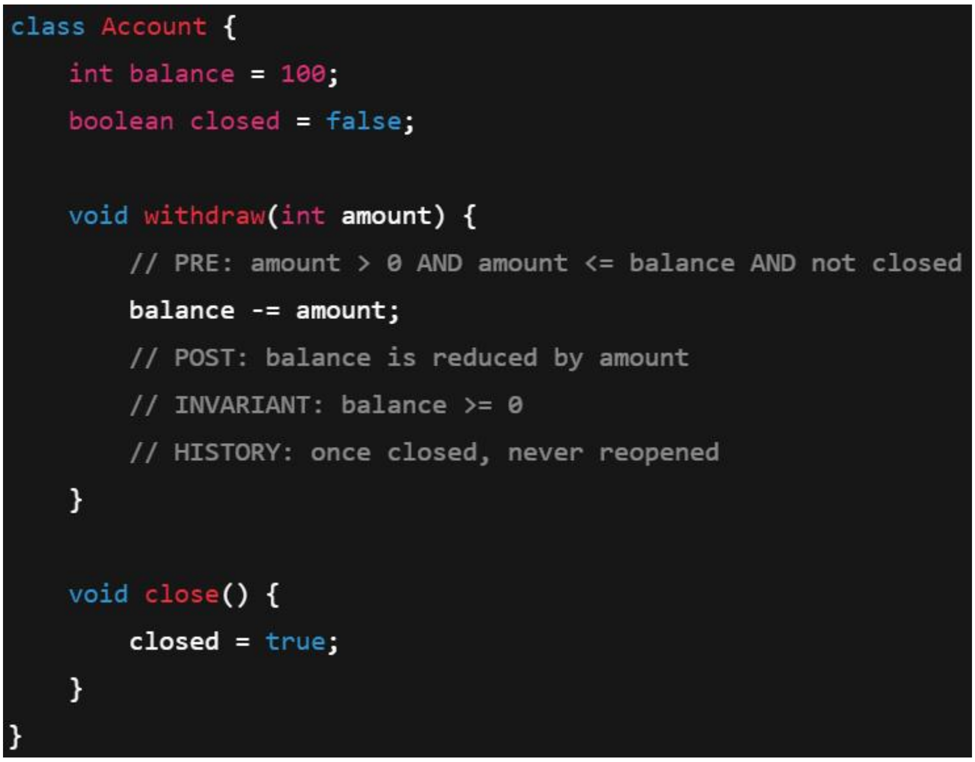


```java
class FixedDepositAccount extends Account {
    @Override
    void withdraw(int amount) {
        // ❌ does NOT reduce balance
        return;
    }
}
```

(post)

NOTE

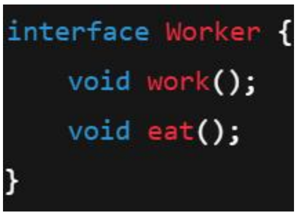
invariant and **history** are object level; **precondition** and **postcondition** are method level;



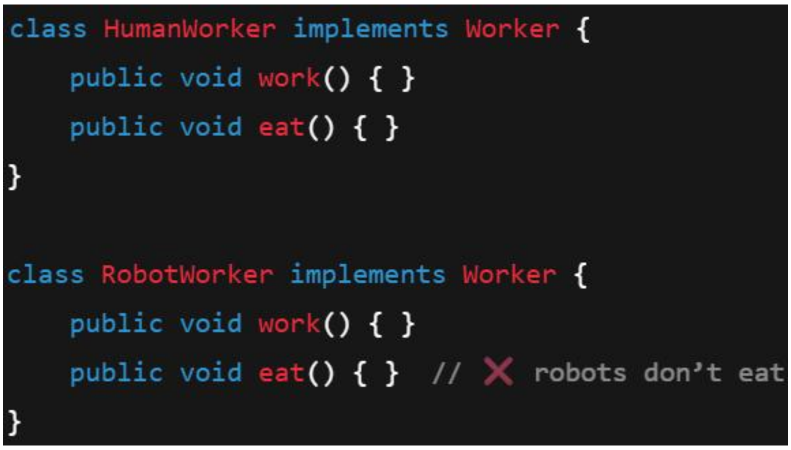
```java
class Account {
    int balance = 100;
    boolean closed = false;

    void withdraw(int amount) {
        // PRE: amount > 0 AND amount <= balance AND not closed
        balance -= amount;
        // POST: balance is reduced by amount
        // INVARIANT: balance >= 0
        // HISTORY: once closed, never reopened
    }

    void close() {
        closed = true;
    }
}
```

- **Precondition**: can I call the method now? (amount <= balance)
- **Postconditoin**: what does the method must guarentee after it runs? (balance reduce)
- **Invariant**: is this object valid all the times? (balance >= 0)
  - In an account, balance cannot be less than 0 (assuming no minus balance account)
- **History**: Can this state ever go backward? (OPEN → CLOSED; no reopen)

- **ISP (Interface Segregation Principle)**
  - Many small, focused interfaces are better than one big “fat” interface..
  - Clients should not be forced to depend on interfaces they do not use.
  - Bad design:

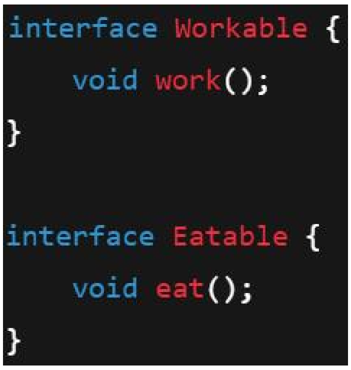


```
interface Worker {
    void work();
    void eat();
}
```

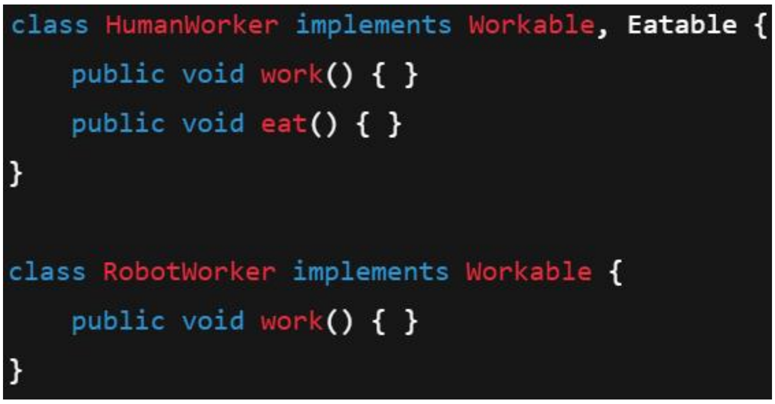


```
class HumanWorker implements Worker {
    public void work() { }
    public void eat() { }
}

class RobotWorker implements Worker {
    public void work() { }
    public void eat() { }  // ❌ robots don’t eat
}
```

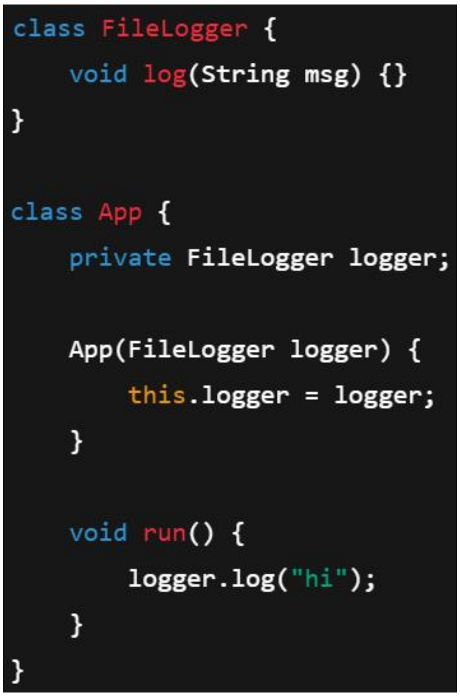
  - Good design:



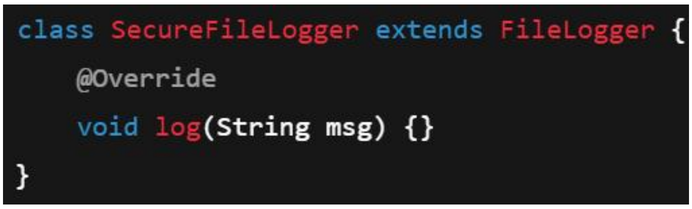
```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}
```



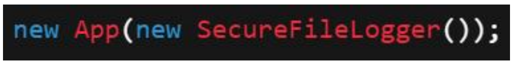
```
class HumanWorker implements Workable, Eatable {
    public void work() { }
    public void eat() { }
}

class RobotWorker implements Workable {
    public void work() { }
}
```

- **DIP (Dependency Inversion Principle)**
  - High-level code should depend on abstractions (interfaces), not on concrete implementations.
  - write concrete classes implementing the interfaces so that the code can be written **loosely-coupled**.
  - High level modules should not talk to the lower level modules directly; the communication should happen through an interface.
    - Ex: application wants to use DB, but it should not talk to SqlDB or MongoDB directly, it should talk to one interface to which both SqlDB and MongoDB implement.
- **NOTE**
  - OCP and DIP looks same, but it is not;
  - DIP is a way to achieve OCP but these 2 are not same.
  - **To check OCP:** "To add a new feature, do I have to edit this existing class?"
  - **To check DIP:** "Does this high-level class depend on a concrete class or on an abstraction?"
  - Abstraction means ***interface*** or ***abstract class***.
  - Consider the below example:
    - 

```java
class FileLogger {
    void log(String msg) {}
}

class App {
    private FileLogger logger;

    App(FileLogger logger) {
        this.logger = logger;
    }

    void run() {
        logger.log("hi");
    }
}
```

    - 

```java
class SecureFileLogger extends FileLogger {
    @Override
    void log(String msg) {}
}
```



```java
new App(new SecureFileLogger());
```

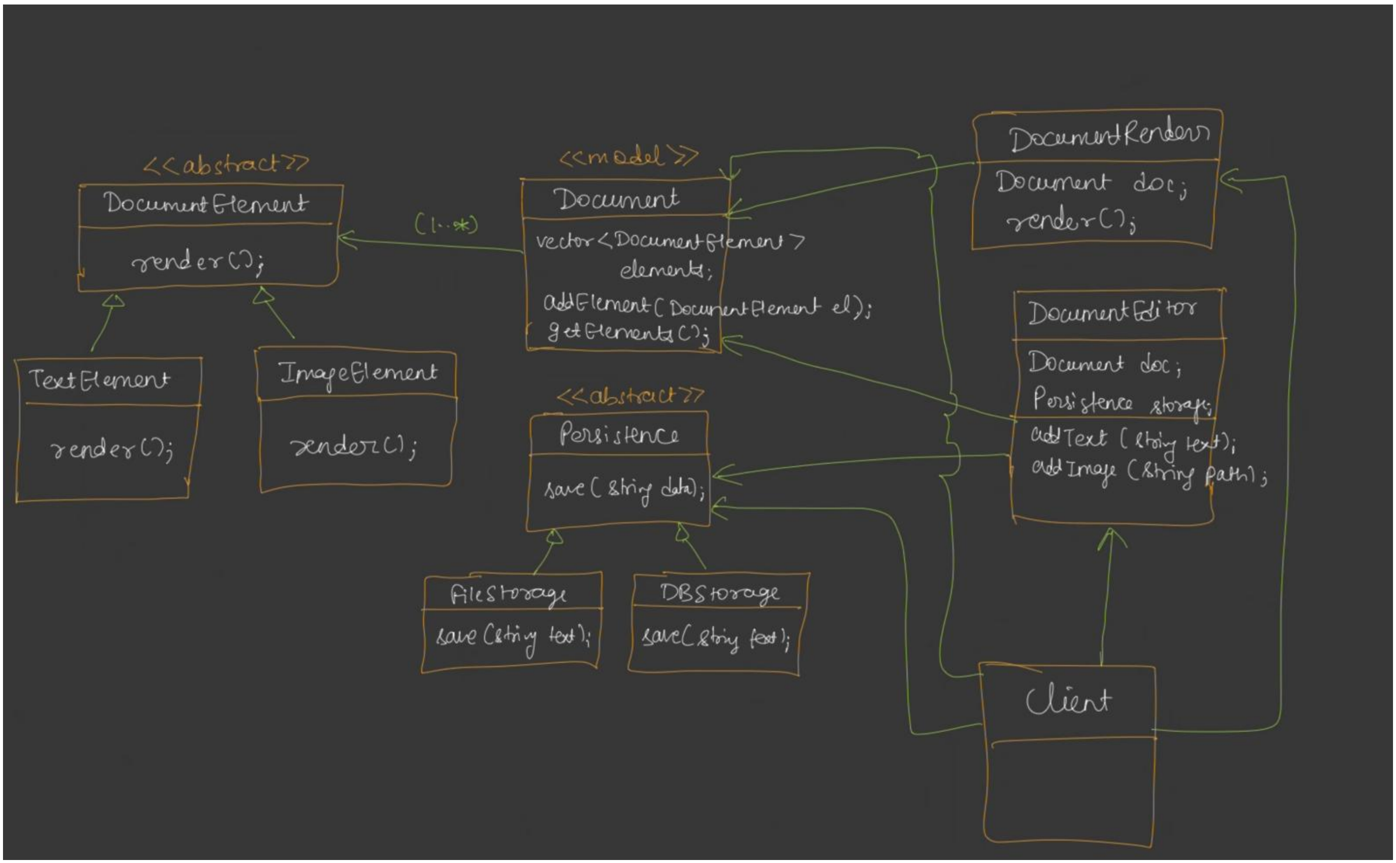
    - To scale it, Did we edit the existing code? **No,** we just created one class *SecuredFileLogger* inheriting the previous class. So **OCP satisfied**.

- Does App depends on a concrete class? **Yes**, *FileLogger* is a concrete class; not an interface or abstract class; So, **DIP violated**

- How to write real-world code maintaining the SOLID principles?
  - Question 1: **If this file changes, will it change for ONE reason only?**
  - Question 2: **If I add a new feature, will I edit this class or add a new one?**
  - Question 3: **Can I replace a parent with a child and nothing breaks?**
  - Question 4: **Am I forced to implement methods I don't use?**
  - Question 5: **Does my core logic depend on interfaces or concrete classes?**

- F

Google Docs LLD with SOLID

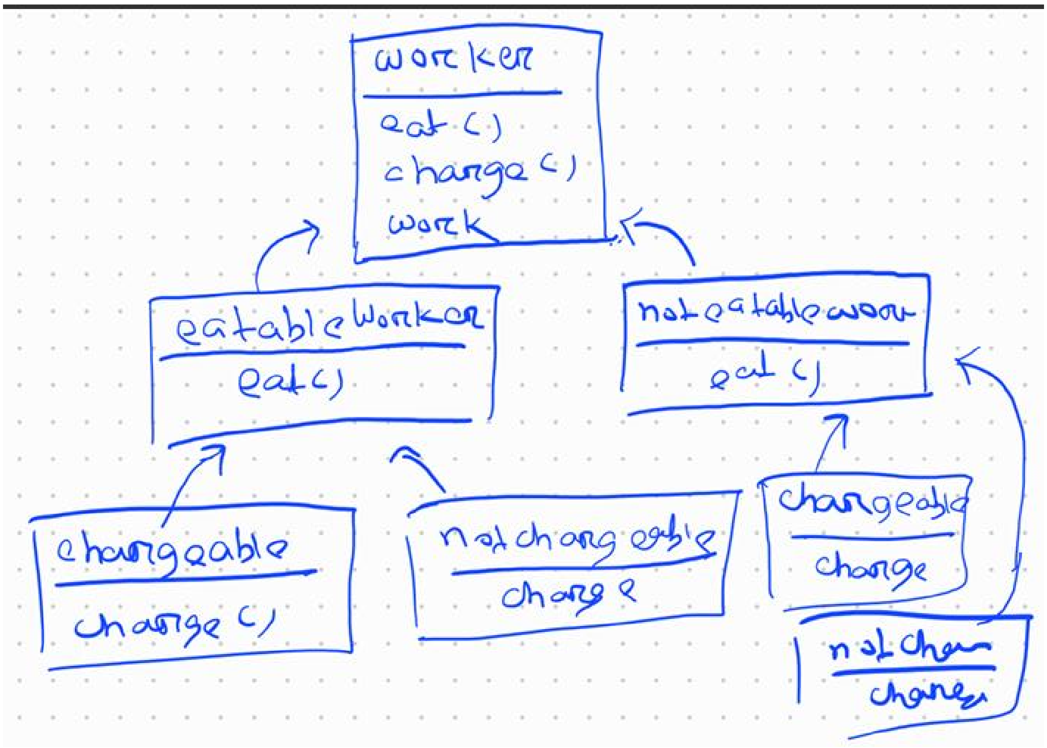


➢

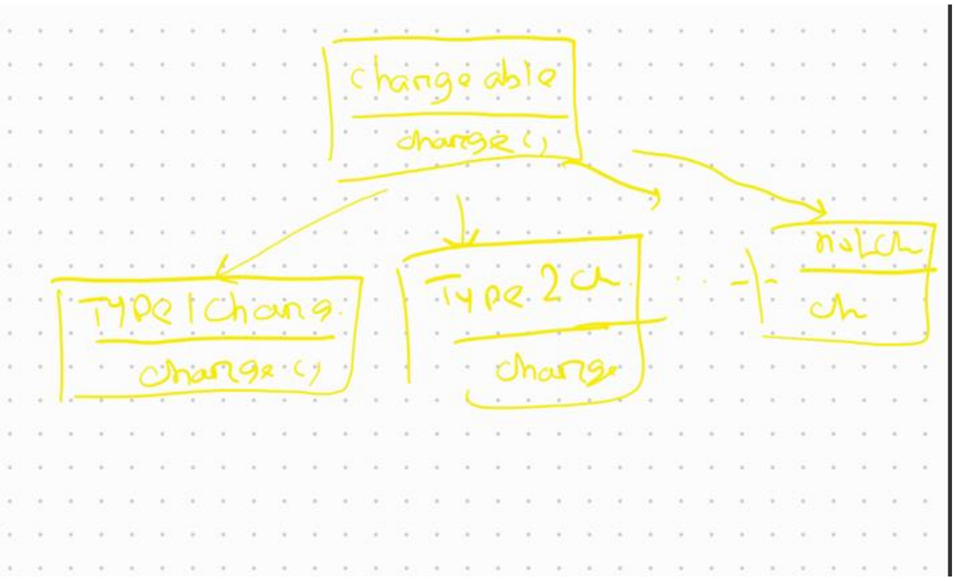
➢ F

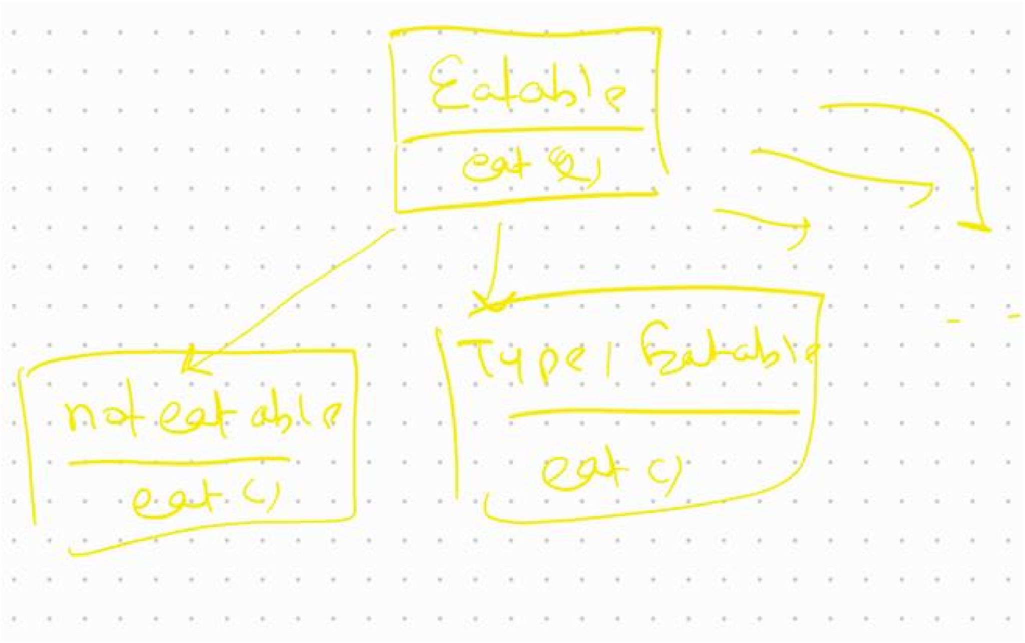
--- Stratagy Design Pattern ---

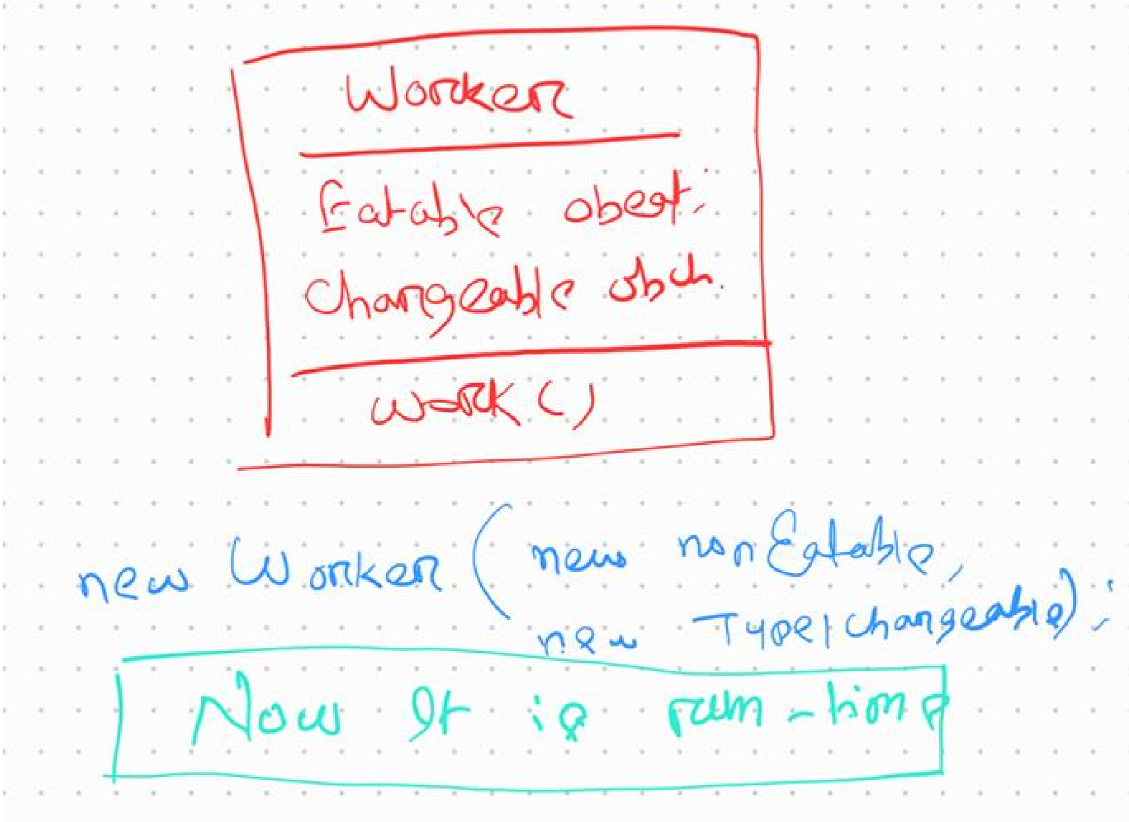
- It is used to avoid Inheritance Hell.
- Separate “what changes” from “what stays the same”, and use composition instead of inheritance.
- *Inheritance* fixes behavior at **compile time**. *Strategy* lets behavior vary at **runtime**.
- Consider the below design:



- Here, for every type of changes in **eat()** and **charge()** method, one more class has to be created inheriting the parent one.
- For each combination, we need to maintain; here **chargeable()** and **notchargeable()** are written 2 times to accommodate 2 types of **eatable()** and **noneatable()**
- If more type of things come, then this inheritance will grow exponentially.







- Here we separated the **changing** and **non-changing** part.
- **work()** is fixed, so it is left as it is; **eatable()** and **charge()** will vary, so created interfaces for those and kept one one variable of those.
- Now, while creating worker object, the client can give which type of object he want, now no more redundent code, its clean and no inheritance hell.

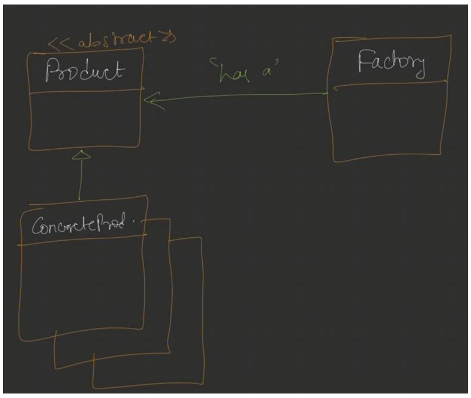
F

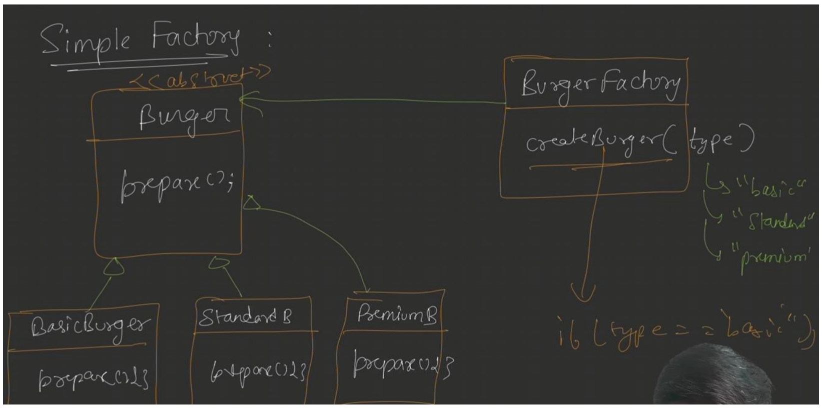
Factory Design Pattern

- Object creation logic and Business logic should be kept separated; Factory design pattern creates objects and supply those.
- 3 types:
  - Simple Factory
  - Factory Method
  - Abstract Factory Method

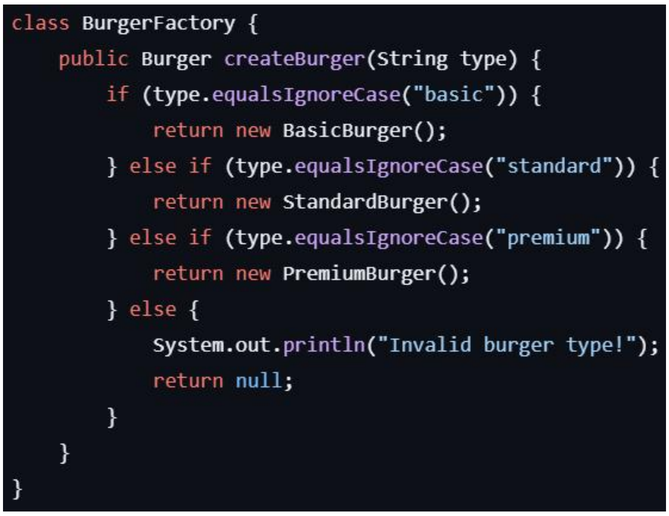
Simple Factory

- Standard UML of Simple Factory



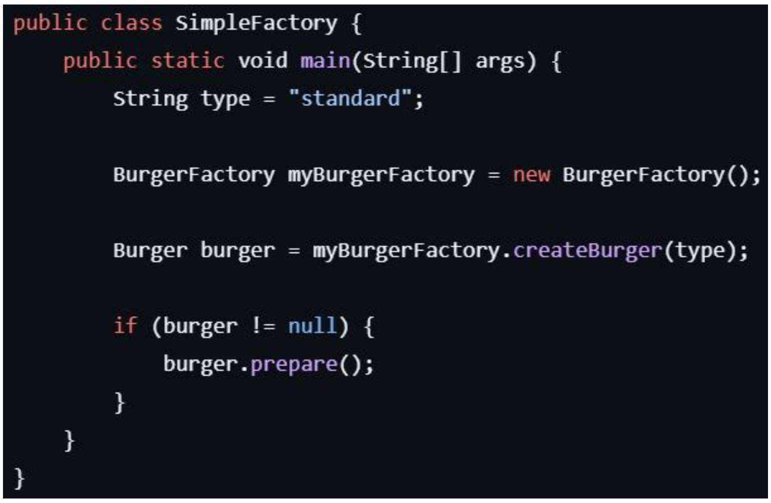


- Here, we have one interface **Burger** and the classes are implementing this for different types of burgers.
- ***BurgerFactory*** is a factory class then create the objects according to the type .



```
class BurgerFactory {
    public Burger createBurger(String type) {
        if (type.equalsIgnoreCase("basic")) {
            return new BasicBurger();
        } else if (type.equalsIgnoreCase("standard")) {
            return new StandardBurger();
        } else if (type.equalsIgnoreCase("premium")) {
            return new PremiumBurger();
        } else {
            System.out.println("Invalid burger type!");
            return null;
        }
    }
}
```

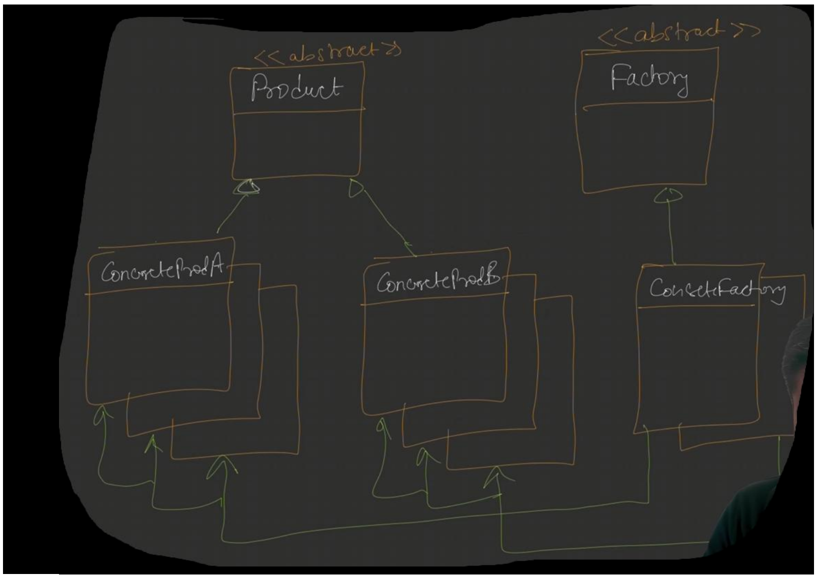
- In this case, if one new type of burger comes then we need to edit the BurgerFactory class which violates OCP of SOLID principle.
- Also, here only one factory is there.
- The below is the use of Simple Factory



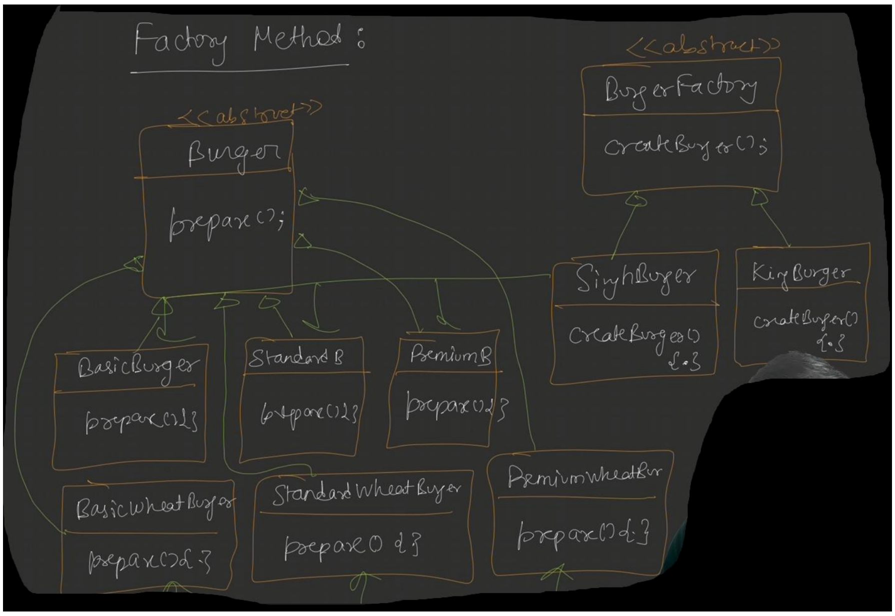
```
public class SimpleFactory {
    public static void main(String[] args) {
        String type = "standard";

        BurgerFactory myBurgerFactory = new BurgerFactory();

        Burger burger = myBurgerFactory.createBurger(type);

        if (burger != null) {
            burger.prepare();
        }
    }
}
```

Factory Method

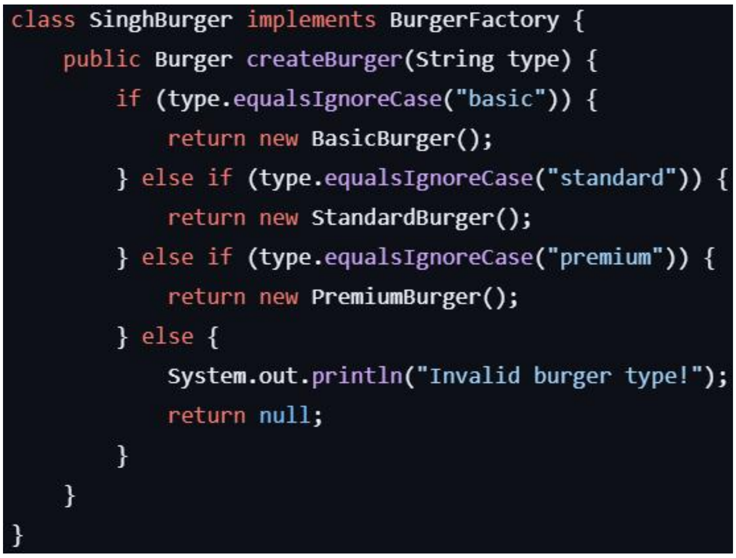
- Simple UML



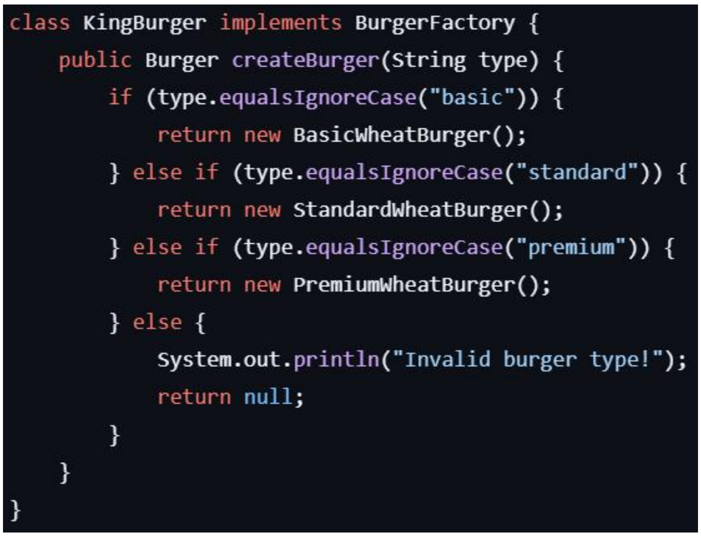
- Defines an interface for creating Objects but allows subclasses to decide which class to instantiate.
- In the Simple Factory, we had only one factory which was ***BurgerFactory***, imagine this as a franchise.
- If multiple franchise are there who makes different types of burgers then it'll break.
- Lets say there are some **6** types of burgers, and 2 franchise there, each make 3 types of burgers out of those 6.
- So, to implement this, we need to make BurgerFactory as abstract and create new factory classes implementing this.



- Here we have to BurgerFactory, *SinghBurger* and *KingBurger;*
  - *SinghBurger* makes Basic, Standard, Premium burgers;
  - SinghBurger makes BasicWheat, StandardWheat, PremiumWheat burgers.

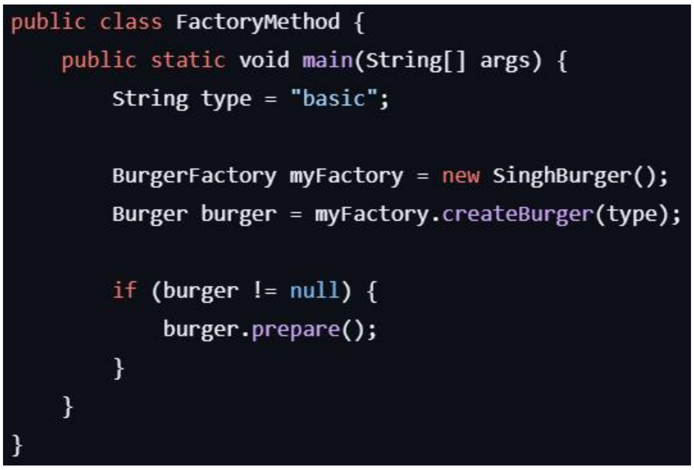


```java
class SinghBurger implements BurgerFactory {
    public Burger createBurger(String type) {
        if (type.equalsIgnoreCase("basic")) {
            return new BasicBurger();
        } else if (type.equalsIgnoreCase("standard")) {
            return new StandardBurger();
        } else if (type.equalsIgnoreCase("premium")) {
            return new PremiumBurger();
        } else {
            System.out.println("Invalid burger type!");
            return null;
        }
    }
}
```



```java
class KingBurger implements BurgerFactory {
    public Burger createBurger(String type) {
        if (type.equalsIgnoreCase("basic")) {
            return new BasicWheatBurger();
        } else if (type.equalsIgnoreCase("standard")) {
            return new StandardWheatBurger();
        } else if (type.equalsIgnoreCase("premium")) {
            return new PremiumWheatBurger();
        } else {
            System.out.println("Invalid burger type!");
            return null;
        }
    }
}
```

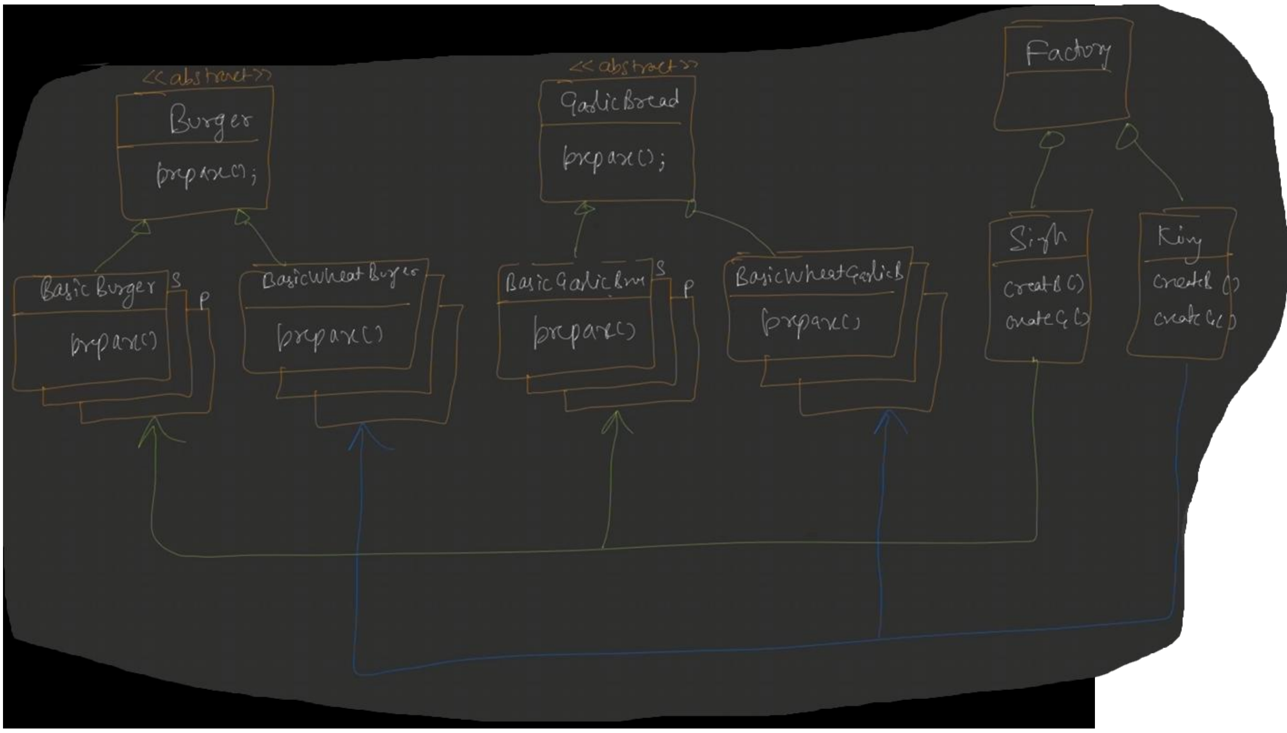
- The below is the use of **Factory Method** classes.



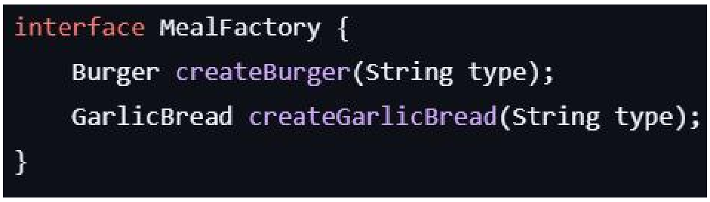
```
public class FactoryMethod {
    public static void main(String[] args) {
        String type = "basic";

        BurgerFactory myFactory = new SinghBurger();
        Burger burger = myFactory.createBurger(type);

        if (burger != null) {
            burger.prepare();
        }
    }
}
```

Abstract Factory Method

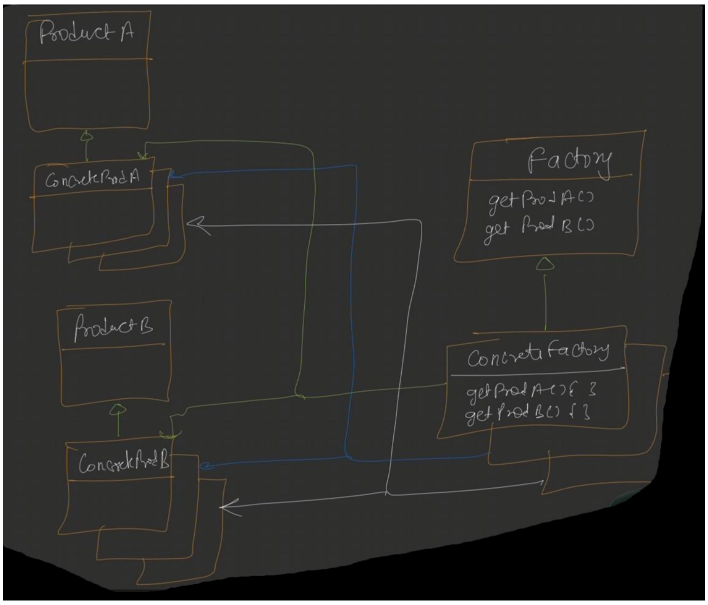
- Till now, the factory was only creating one type of objects which was "Burger". But in real world, it needs to create different types of objects.
- Provides an interface for creating families of related objects without specifying their concrete classes.

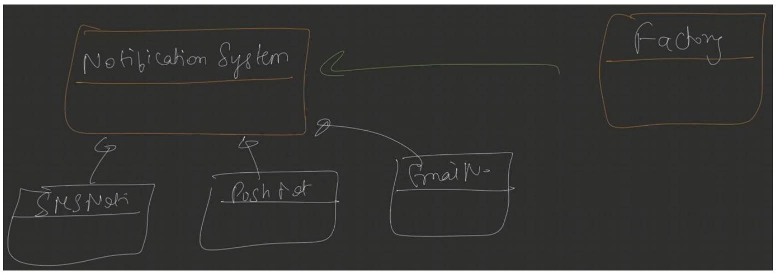


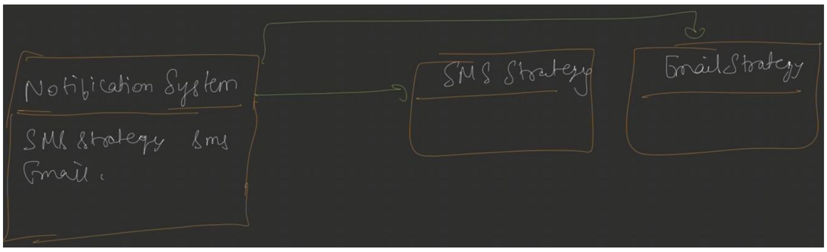
- Nothing difference, before the factory classes were having only one method to create Burger type of objects, not it'll be having one more method to create GarlicBread objects as well.
- Here the factory method will contain 2 methods:



```
interface MealFactory {
    Burger createBurger(String type);
    GarlicBread createGarlicBread(String type);
}
```

- 

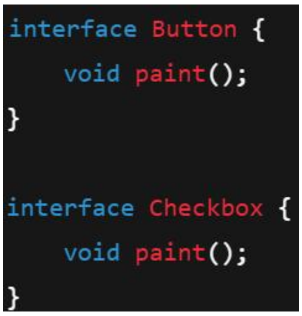
- Example: **Notification System**
    - 

    - Here you might be thinking, why not using ***Strategy Design Pattern*** here, we can assume notification system as a strategy.
        - 

    - In this case think like this:
        - **If I want to vary algorithm during run time : Strategy Design Pattern**
            - In this case we can assume the objects are already created somewhere.
        - **If I want to separate object creation logic : Factory Design Pattern**

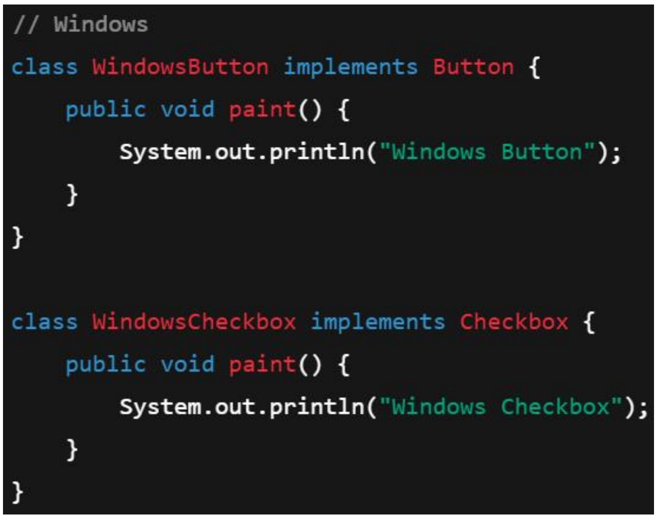
➢ Example

➢ Products

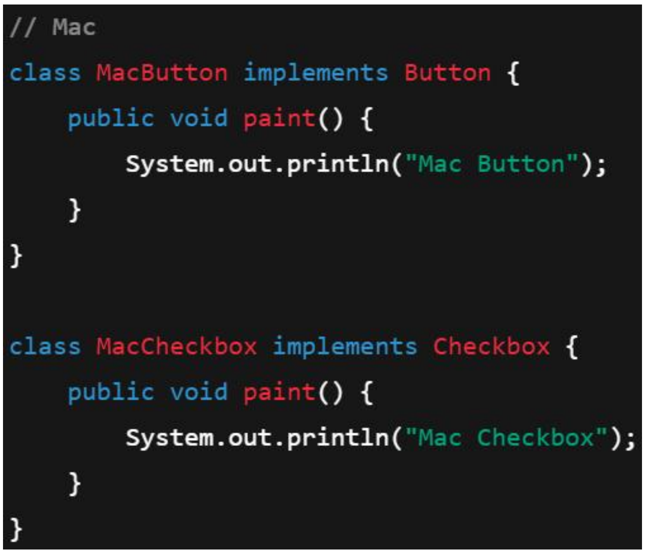
- **Abstract Products**
    - 

```
interface Button {
    void paint();
}

interface Checkbox {
    void paint();
}
```

- Concrete Products

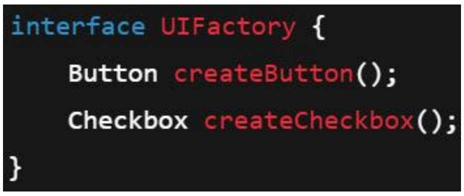


```java
// Windows
class WindowsButton implements Button {
    public void paint() {
        System.out.println("Windows Button");
    }
}

class WindowsCheckbox implements Checkbox {
    public void paint() {
        System.out.println("Windows Checkbox");
    }
}
```



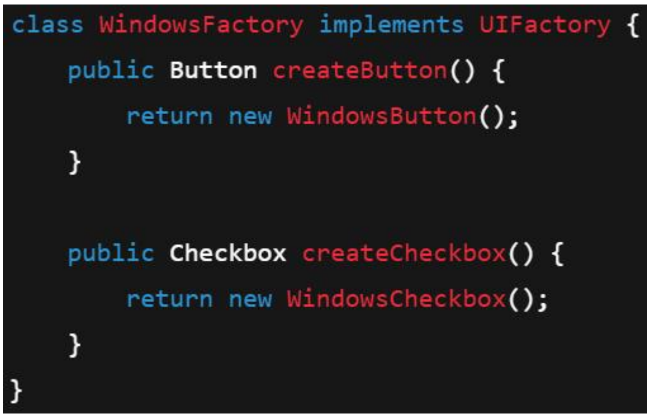
```java
// Mac
class MacButton implements Button {
    public void paint() {
        System.out.println("Mac Button");
    }
}

class MacCheckbox implements Checkbox {
    public void paint() {
        System.out.println("Mac Checkbox");
    }
}
```

- **Factories**
  - Abstract Factory



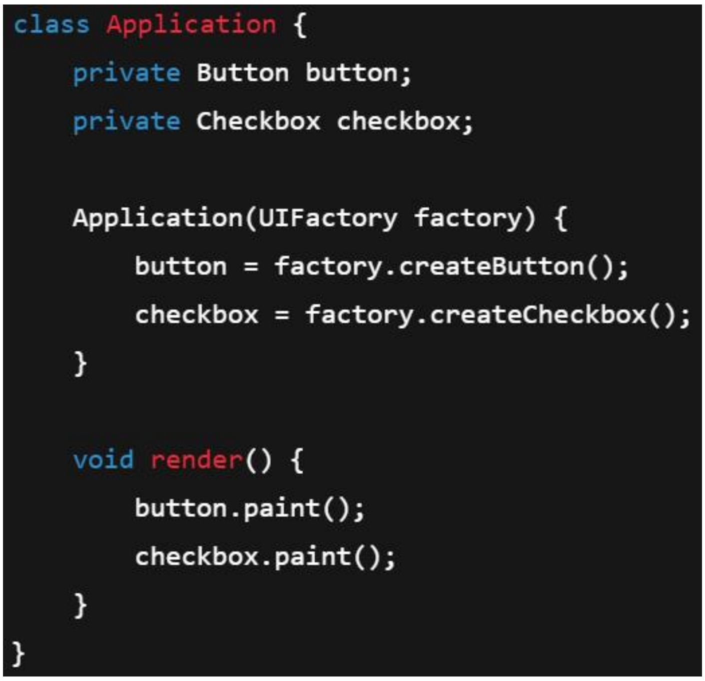
```java
interface UIFactory {
    Button createButton();
    Checkbox createCheckbox();
}
```

  - Concrete Factories



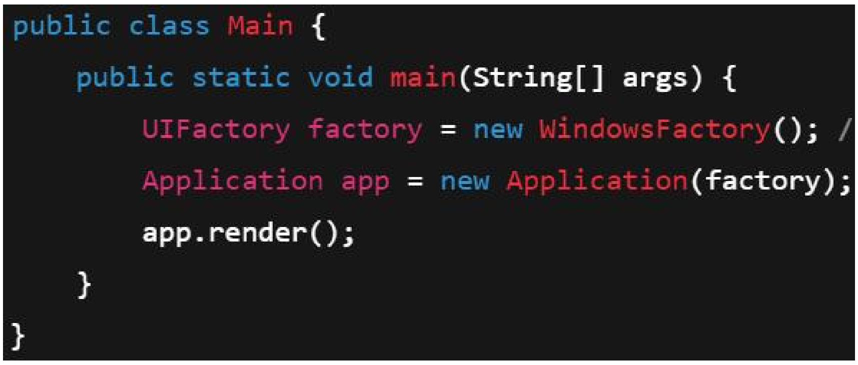
```java
class WindowsFactory implements UIFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}
```

- **Client**



```
class Application {
    private Button button;
    private Checkbox checkbox;

    Application(UIFactory factory) {
        button = factory.createButton();
        checkbox = factory.createCheckbox();
    }

    void render() {
        button.paint();
        checkbox.paint();
    }
}
```

- **Usage**

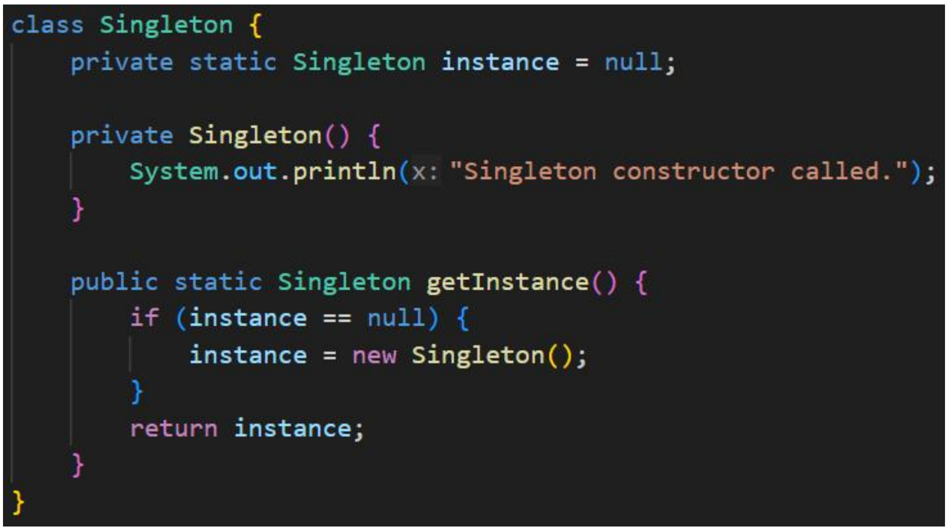


```
public class Main {
    public static void main(String[] args) {
        UIFactory factory = new WindowsFactory(); /
        Application app = new Application(factory);
        app.render();
    }
}
```

- F
- F
- F
- F
- F
-

Singleton Design Pattern

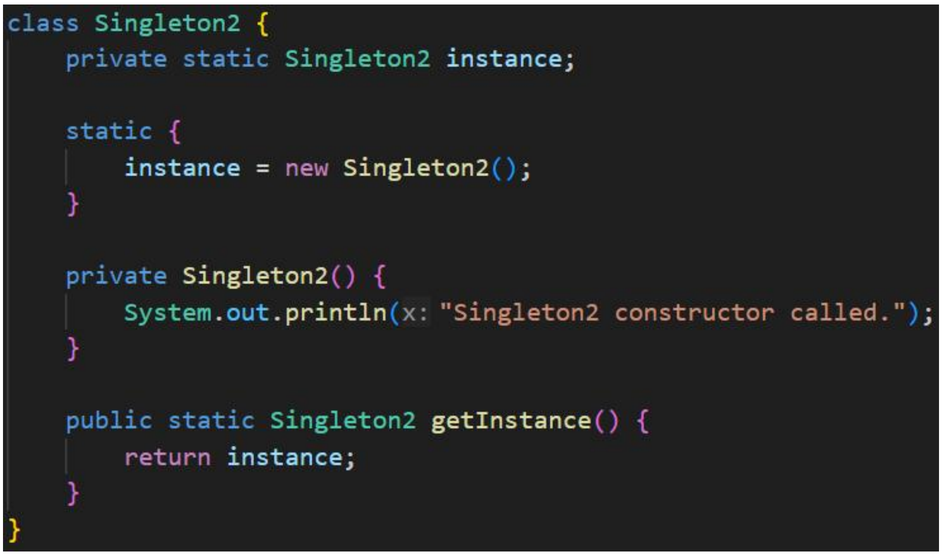
- Only one object is created from a class and that object is used throughout the application.
- Steps to create a singleton class:
  - First make the constructor private, now object cannot be created with **new** keyword.
  - Now create a static variable to keep the instance, and a method to return that instance.



```java
class Singleton {
    private static Singleton instance = null;

    private Singleton() {
        System.out.println(x: "Singleton constructor called.");
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

  - **Problem: This one is not thread-safe, i.e. if 2 threads call this getInstance() method at same time, then 2 instances will be created.**

- --------------- OR ---------------

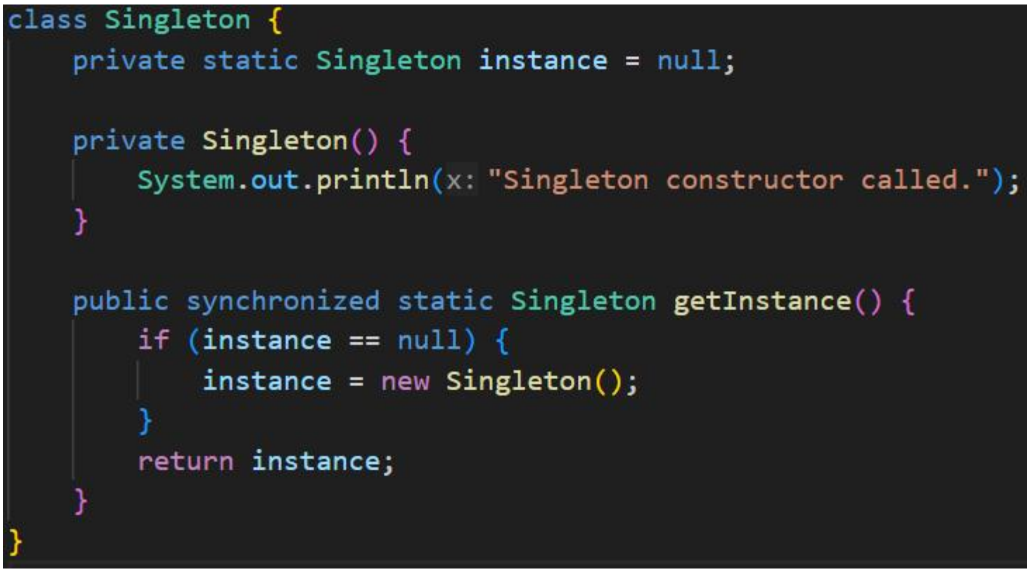


```java
class Singleton2 {
    private static Singleton2 instance;

    static {
        instance = new Singleton2();
    }

    private Singleton2() {
        System.out.println(x: "Singleton2 constructor called.");
    }

    public static Singleton2 getInstance() {
        return instance;
    }
}
```

  - You can use **static block** in Java.
  - **Problem: No lazy loading; it is thread safe, but even if the object is not required, the object will be created in this case.**

Thread-safe Singleton Implementation

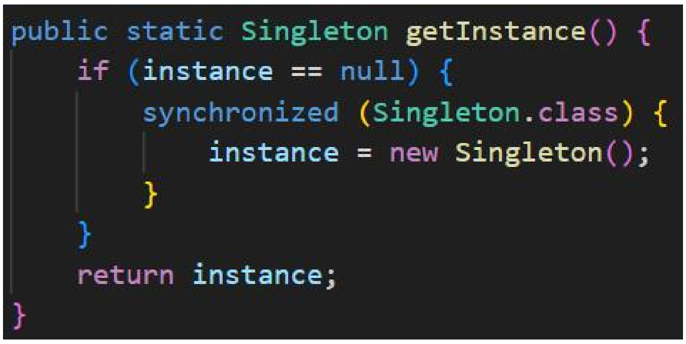
Lock the **getInstance()** method

- 

```java
class Singleton {
    private static Singleton instance = null;

    private Singleton() {
        System.out.println(x: "Singleton constructor called.");
    }

    public synchronized static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

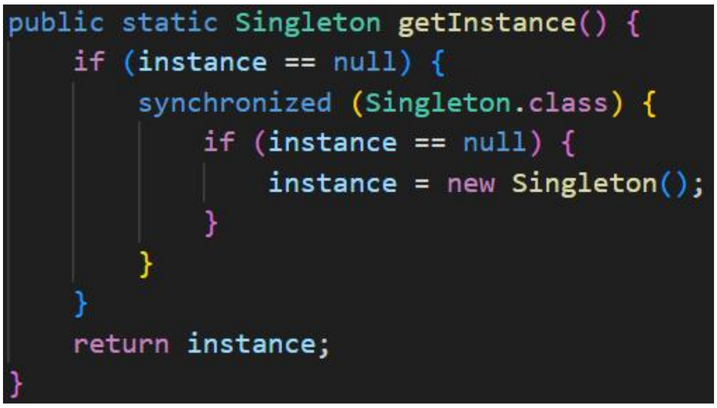
  (**synchronized** is used to lock)
- **Problem: even after the instance is created, multiple threads can not call this method at same time.**
- Only first time, when the instance is not created, only one call should happen; after that multiple threads can call this method at same time; it'll not cause any problem.
- So, we should not apply **lock** on the method itself.

Lock only the *instance creation part inside if block* (lock should be applied on **Class** level)

- 

```java
public static Singleton getInstance() {
    if (instance == null) {
        synchronized (Singleton.class) {
            instance = new Singleton();
        }
    }
    return instance;
}
```

  - (**synchronized** not there at method level)
- It looks fine, but there is a big problem here.
- **Problem:**
  - Lets say **T1** and **T2** (2 threads) call the **getInstance()** method at same time.
  - As the method is not locked, both can enter inside the *if* block.
  - Lets say **T1** applied the lock, so now **T2** has to wait till **T1** is release the lock.
  - So now, after **T1** is done, one instance will be created and stored.
  - Now, **T2** was waiting inside that *if* block. So, after **T1** releases the lock **T2** will again create that instance.
  - So, now 2 instances got created in place of 1
- We need to implement another check inside **synchronized** block.

- 

```java
public static Singleton getInstance() {
    if (instance == null) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

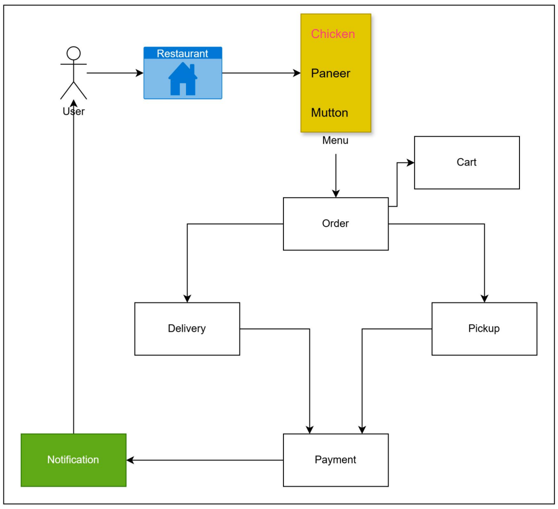
✓

  - Now it’ll work, after **T1** releases, in **T2** it’ll again check if instance is null, as **T1** already created the instance so it’ll not create again.

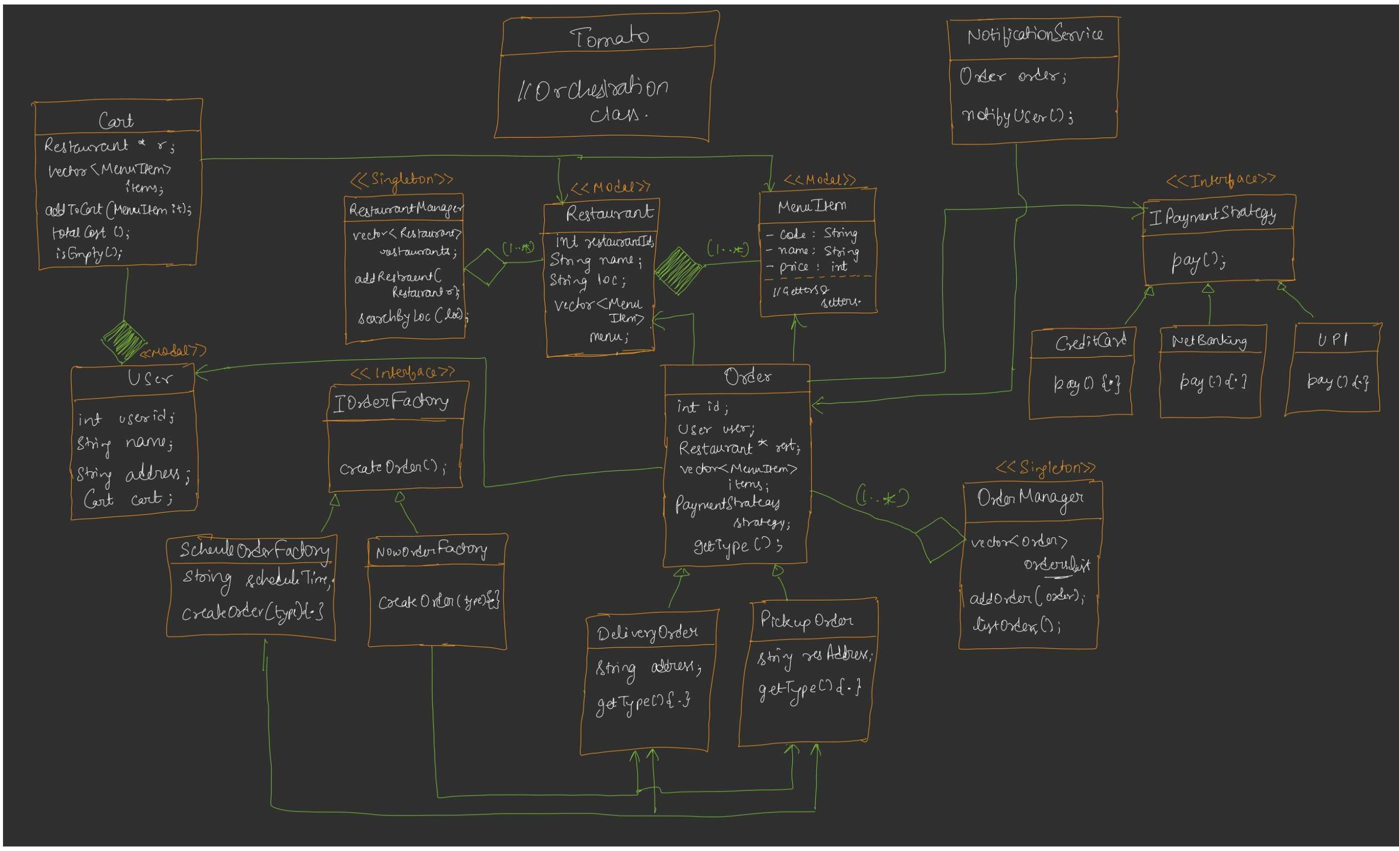
- **Real World use-cases**
  - Logging systems
  - Database connections
  - Configuration manager

Design a Food Delivery App

- Requirements
  - Functional requirements:
    - User can search for restaurants based on location
    - User can add items to cart
    - User can checkout by making payment
    - User should be notified once order is placed successfully
  - Non-Functional requirements:
    - Each part of design should be scalable and modifiable
- Happy flow:



- UML diagram   ("Tomato" is the name of the App):

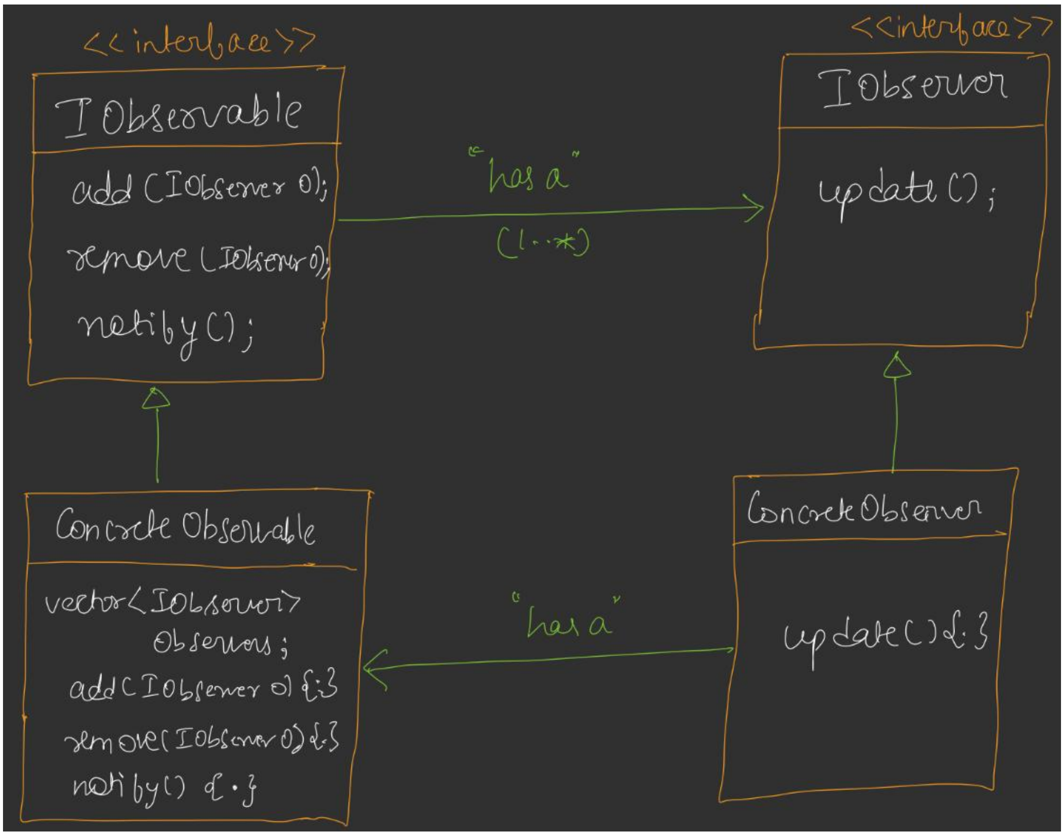


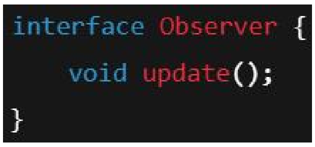
- Here we have assumed, Menu item doesn’t exist without Restaurant, so it’s a ***composition*** relation.
- **<<model>>** means kind of Entity, which have getters and setters.
- No one should talk to **Restaurant** directly, he has to talk to **RestaurantManager** to access Restaurants. So, it’ll make a **loose coupling**.
- One **RestaurantManager** should be there in whole application that manages the list of restaurants and rest. So, it is better to make is **<<Singleton>>**.
- Now, **User** which will be having ***composition*** relationship with **Cart**, cart will not exist without user.
- **Cart** to **Restaurant**, one-to-one relationship (empty cart can be present, so aggregation).
- **Cart** to **MenuItem**, one-to-many relationship (empty cart can be present, so aggregation).
-

- F
- F
- F
- F
- F
- F
- F

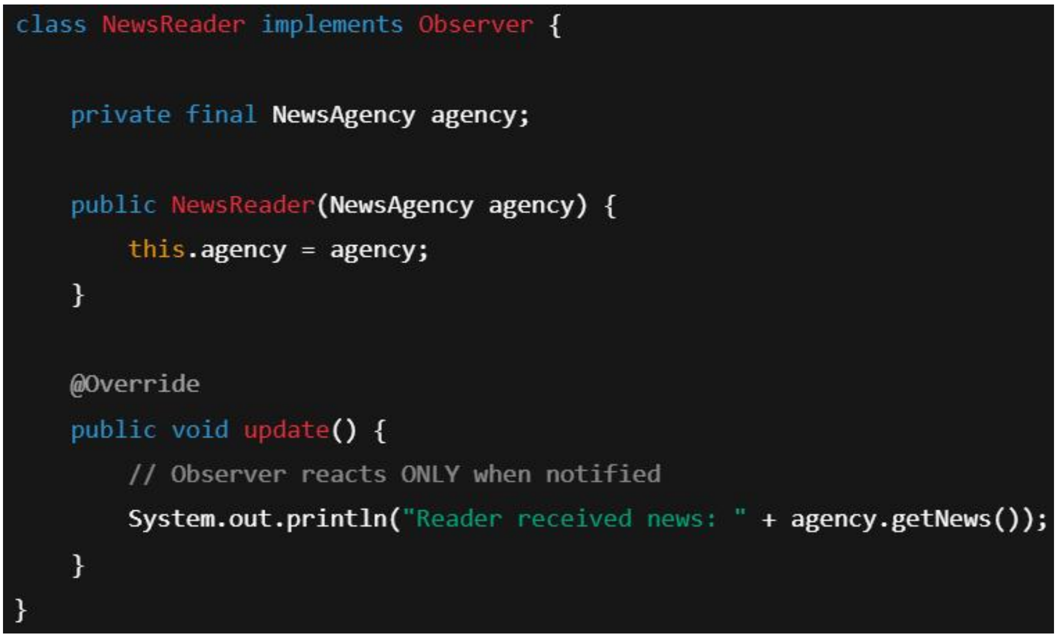
Observer Design Pattern

- Description
  - In this design pattern, 2 types of things are there:
    - **Subject**
    - **Observer**
  - Subject to Observer: **one-to-many** relationship.
    - A single **subject** can be observed by multiple **observers**.
  - Here, Observer observe the Subject if any state changes happens in the Subject.
  - The following ways are possible:
    - The observer will ask the subject if any state change happened repeatedly, and the subject will send response; then observer will get to know about the state change.
      - But this is very time consuming, and too much calls are there.
    - Other way is, **subject** keeps track of all of its **observers** and notifies those if any state change happens. This is the actual case in Observer Design Pattern.
- Defines a one-to-many relationship between objects so that when one object changes state, all of its dependents are notified and get updated automatically.
- UML Diagram

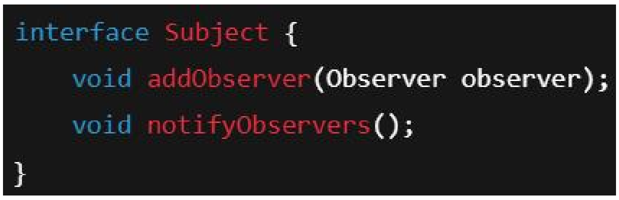


- Flow:
    - Subject (Observ**able**) contains a ***notify()*** method (name can be different) which calls the **update()** method of each *observer*.
    - **update()** method inside the Observer calls the **getValue()** method (can be different name) from the *Subject* to get the latest state.
    - In below example,
        - notify() ----------> notifyObservers()
            - Its calling **update()** method of Observer
        - update() -----------> update()
            - Its calling **getNews()** method of Subject
        - getValue() ----------> getNews()
            - Its returning the latest state value
- Example
    - Observer
        - 

```
interface Observer {
    void update();
}
```

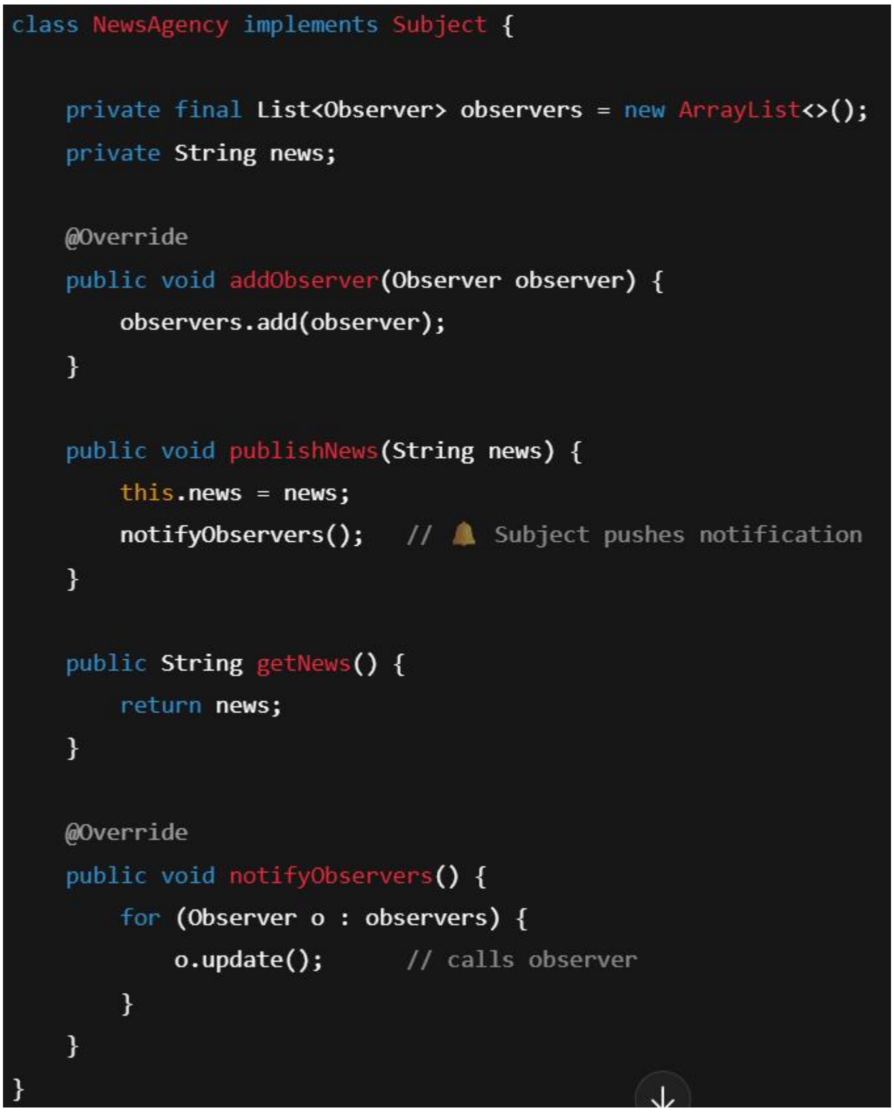
        - 

```
class NewsReader implements Observer {

    private final NewsAgency agency;

    public NewsReader(NewsAgency agency) {
        this.agency = agency;
    }

    @Override
    public void update() {
        // Observer reacts ONLY when notified
        System.out.println("Reader received news: " + agency.getNews());
    }
}
```

    - Subject
        - 

```
interface Subject {
    void addObserver(Observer observer);
    void notifyObservers();
}
```

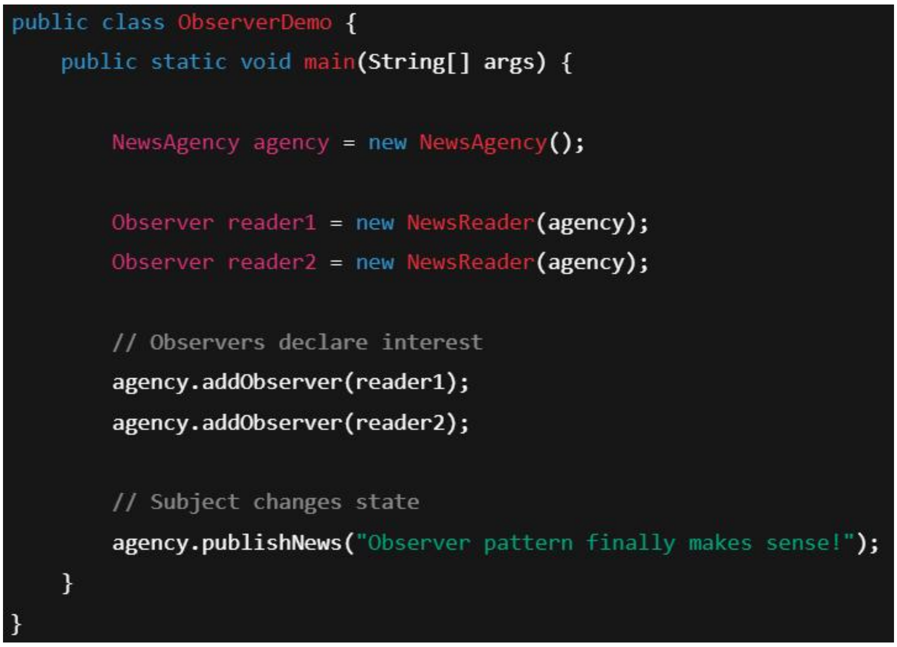
        - **notifyObservers()** is the method which notifies for each state change.



```java
class NewsAgency implements Subject {

    private final List<Observer> observers = new ArrayList<>();
    private String news;

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void publishNews(String news) {
        this.news = news;
        notifyObservers();   // 🔔 Subject pushes notification
    }

    public String getNews() {
        return news;
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update();      // calls observer
        }
    }
}
```

ء

م **Client**



```java
public class ObserverDemo {
    public static void main(String[] args) {

        NewsAgency agency = new NewsAgency();

        Observer reader1 = new NewsReader(agency);
        Observer reader2 = new NewsReader(agency);

        // Observers declare interest
        agency.addObserver(reader1);
        agency.addObserver(reader2);

        // Subject changes state
        agency.publishNews("Observer pattern finally makes sense!");
    }
}
```

ء

- ➢ F

- F
- F
- F
- F
- F
-