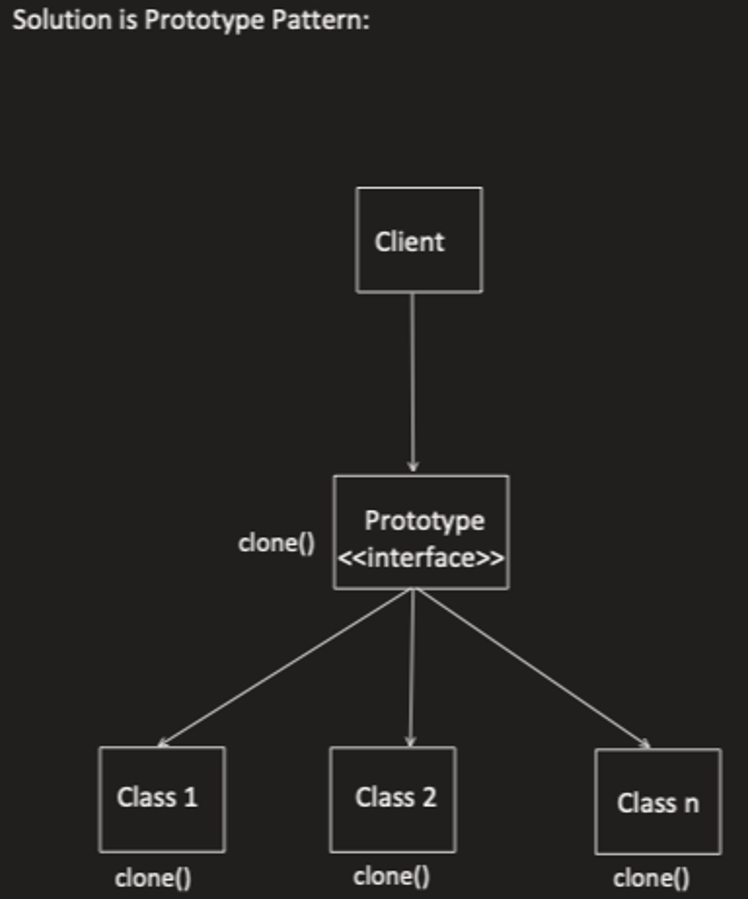
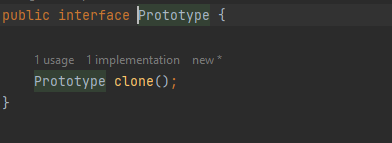
**CREATIONAL DESIGN PATTERN**

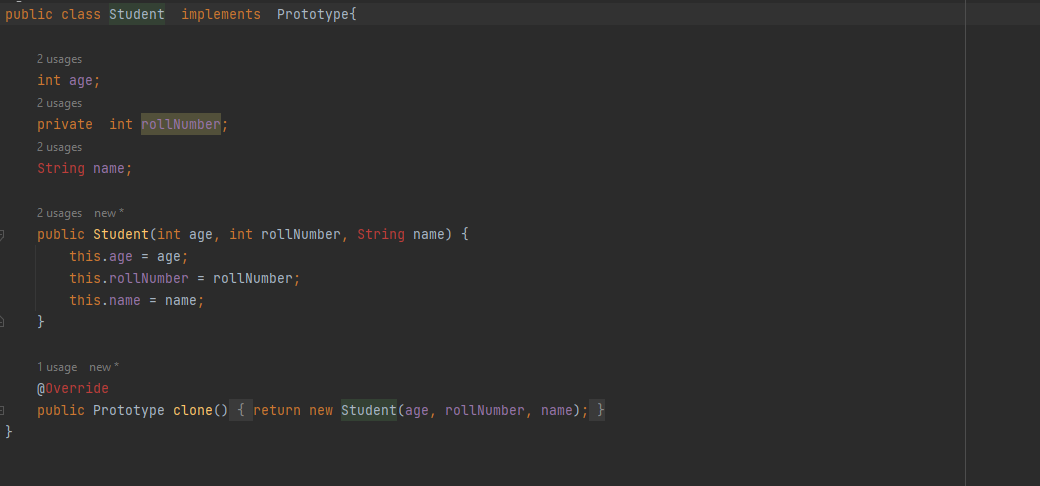
Controls creation of an object.

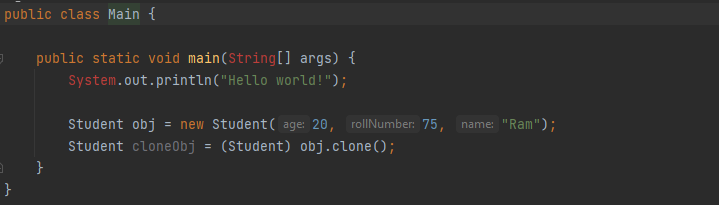
**PROTOTYPE**

Used when we have to copy/clone from existing obj. Whichever class we need to copy will have an impl of the clone method.  
It is the responsibility of the class which would be cloned in future to define the clone method. We need the clone method in order to access the private data members of the class.  
  
This is used when it is expensive to create the original obj and we might think of creating a new class for a new feature



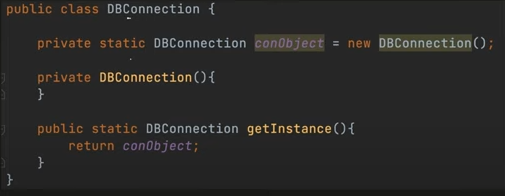
****

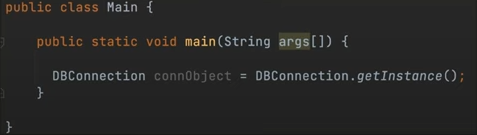
****

****

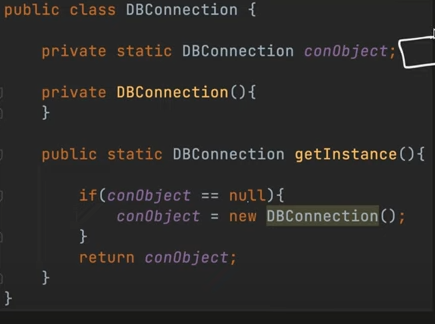
**SINGLETON PATTERN  
  
4 ways  
  
Remember:  
Make the obj private and static and constructor private. Control would be done only using getInstance() which again we need to make as static.**

1. **Eager Initialization**

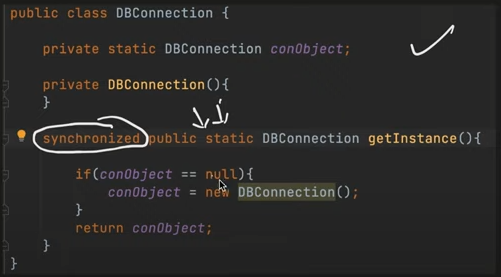
****

****

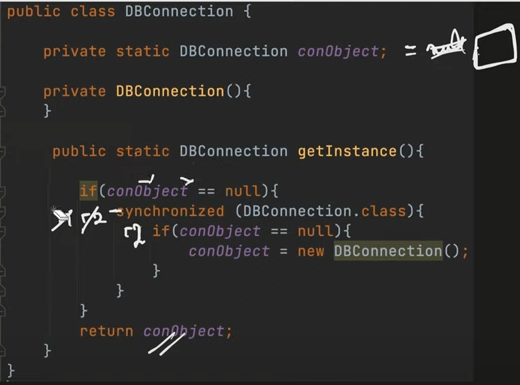
1. **Lazy**

**  
  
The problem with this approach is in case we have multiple threads entering the critical section where we check conObject == null. In this scenario both the threads will try to create the conObject and 2 objects will be created.**

1. **Synchronized**

**  
  
The problem we saw in lazy is solved with this. But since locking each and everytime getInstance() is called is very expensive we do not use synchronized method because of this.**

1. **Double Locking**

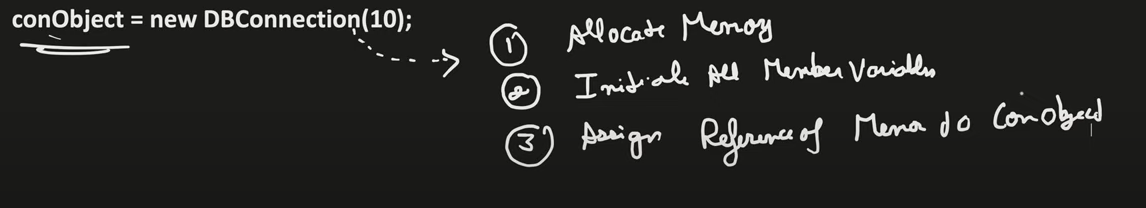
****

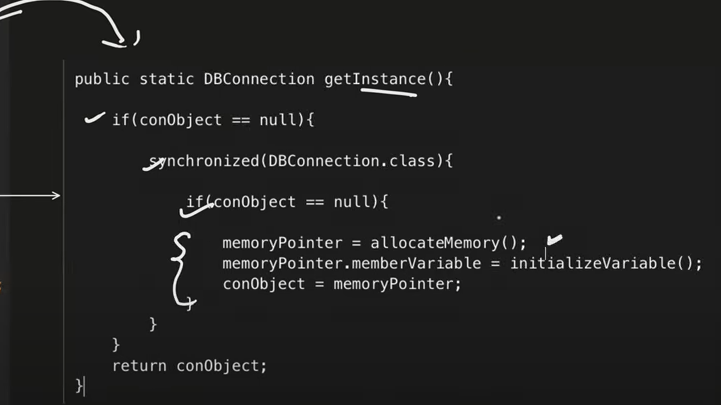
There are certain memory issues because of double locking

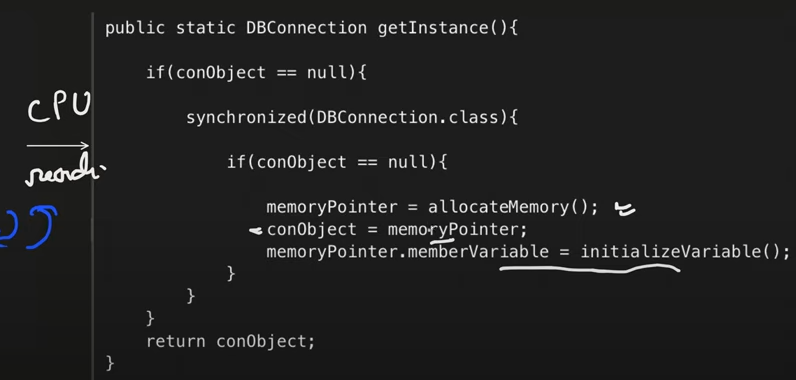
Suppose we introduce a new variable memberVariable



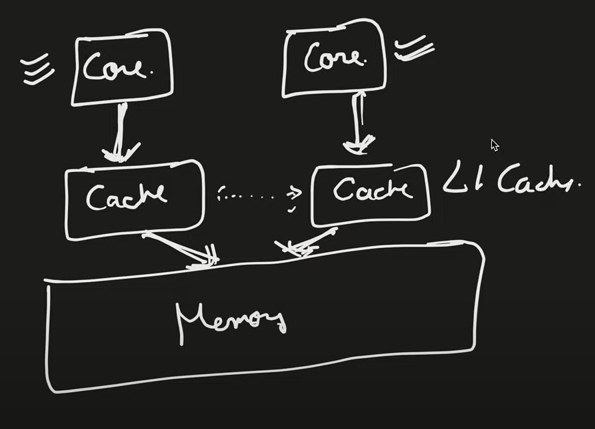
**Issue 1: Reordering of Instruction**

**  
  
The above block of code can be represented as below where we have broken down the steps when an obj is created using the new DBConnection()**

**  
  
Since CPU manages the order of instructions this can also be represented as**

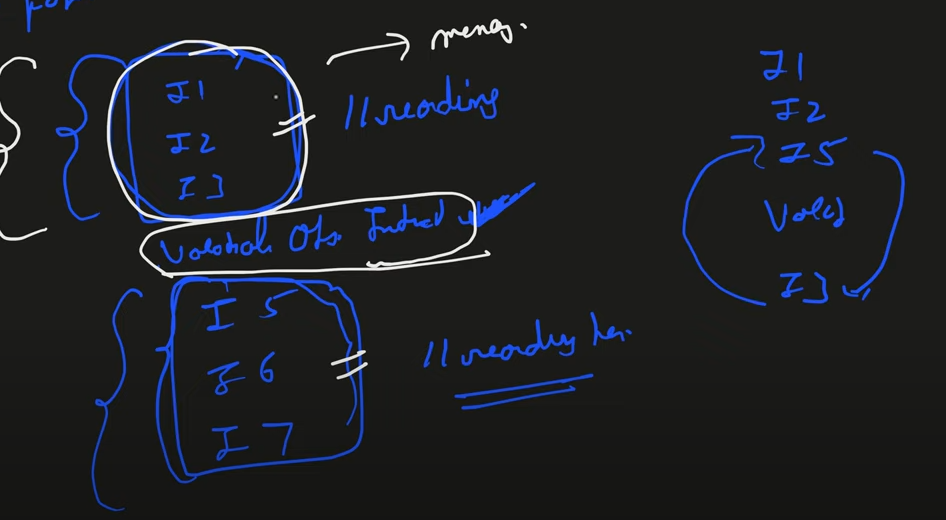
****

**Now conObject would be pointing to object for which memory is allocated. And at this moment memberVariable is assigned with a default value. Now this is before initializeVariable() being called.  
  
Therefore now when T2 comes in this block conObject== null is false and it will use the existing conObject where memberVariable is assigned with a default value as initializeVariable() has not been called yet.  
  
  
  
Issue 2: L1 Caching**

**  
  
CPU has multiple cores, now each core has its own cache where object is present and after some time it is written to the memory**

**Now suppose Thread T1 writes an object into the L1 cache of first core but it is not stored into the memory yet and meanwhile another thread T2 comes in a different core now in that case conObject == null would try to read from memory and would be true and this would go ahead creating a new object again.**

**  
  
VOLATILE -> reads/writes from memory directly  
  
With volatile in case of scenario defined in Issue 2, data is always read from and written to the memory.  
  
It also handles Issue 1 by( the left rep shows internally instructions and reorder but not with the volatile instruction therefore the representation on right is not allowed)**

****

**Final code:**

***public class DBConnection {***

***private static volatile DBConnection conObject; // Use volatile to prevent caching and reordering***

***int memberVariable;***

***private DBConnection(int memberVariableValue) {***

***this.memberVariable = memberVariableValue;***

***}***

***public static DBConnection getInstance() {***

***if (conObject == null) { // First check without locking***

***synchronized (DBConnection.class) { // Synchronized block***

***if (conObject == null) { // Second check with locking***

***conObject = new DBConnection(10); // Fully initialize the object***

***}***

***}***

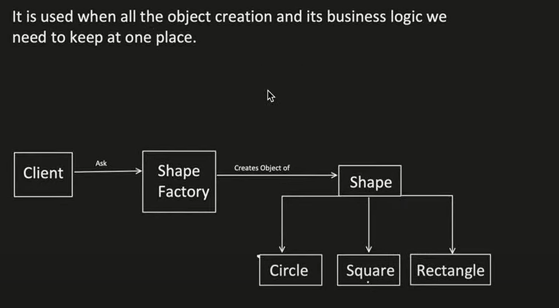
***}***

***return conObject;***

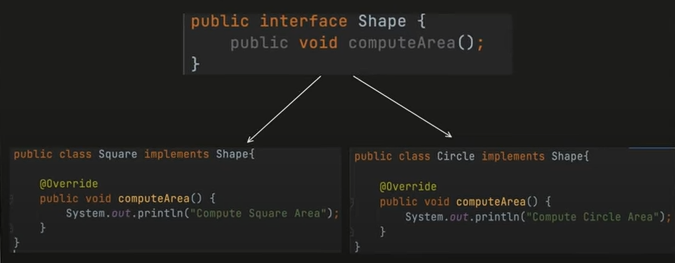
***}***

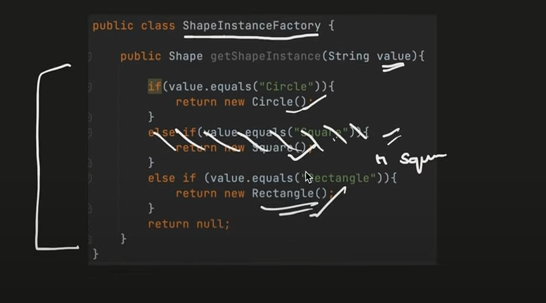
***}***

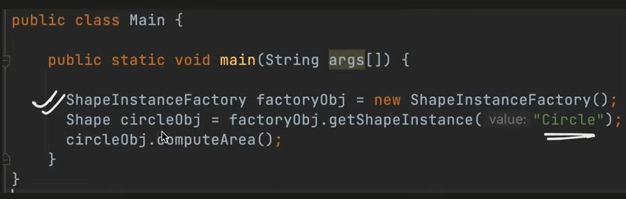
**Factory Pattern  
  
  
Simple Factory**

****

**This says that if there is any change in the logic of creation of an object we need to change it only at one place and not in 100s of other classes**

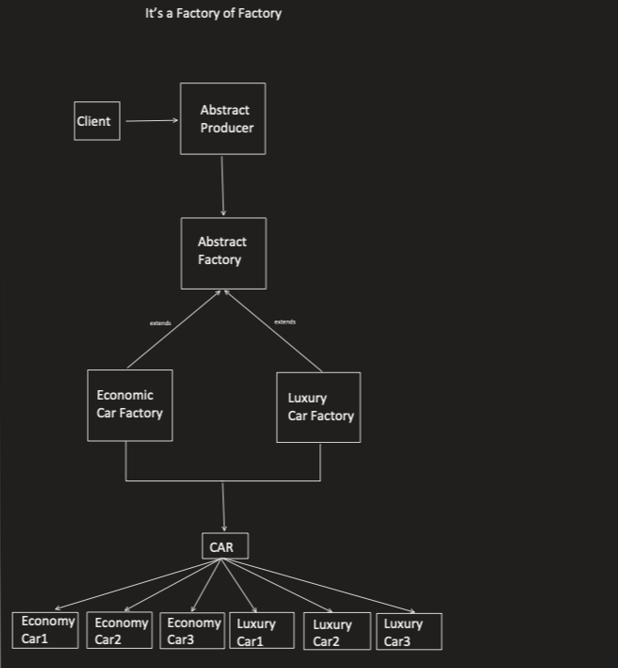
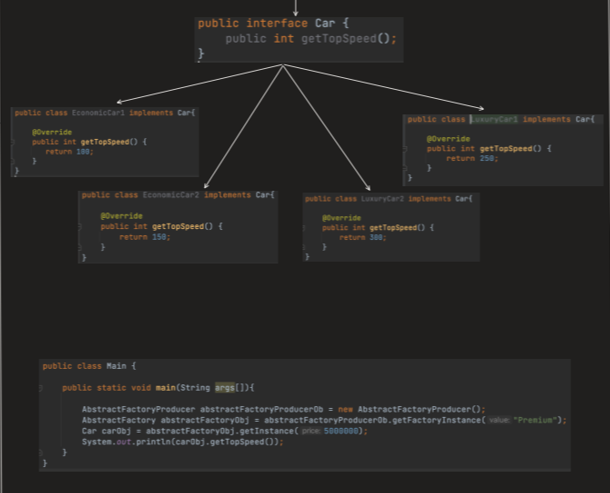
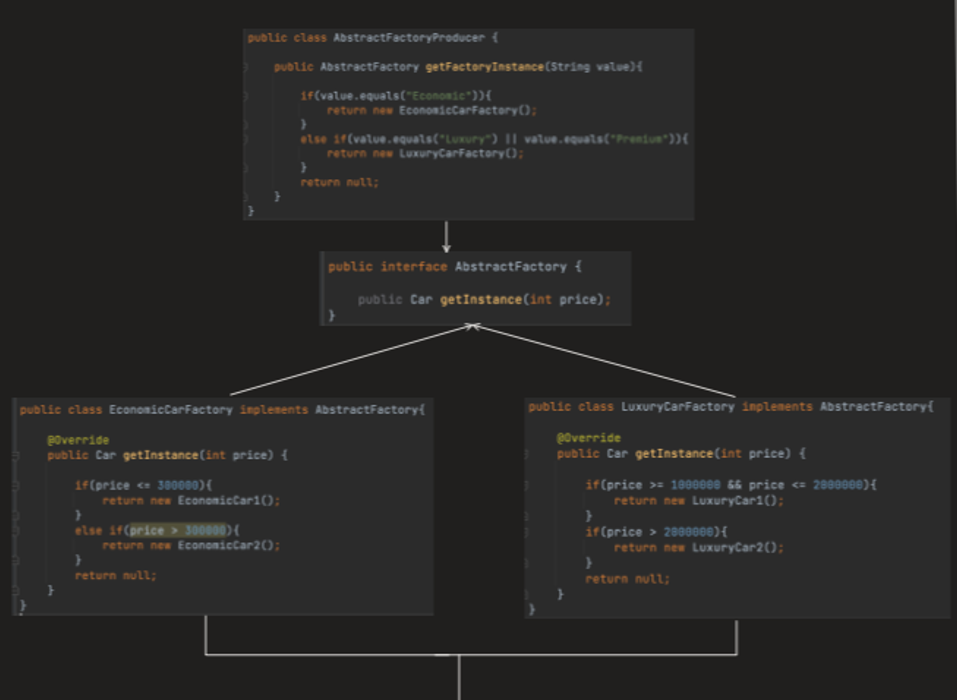
****

**  
Now as we can see if there is a change in logic say suppose if value ==”Rectangle” or “Square” then we have to return Rectangle only. In this case we just need to change this logic at one place and not in every place that is using a square or rectangle.**

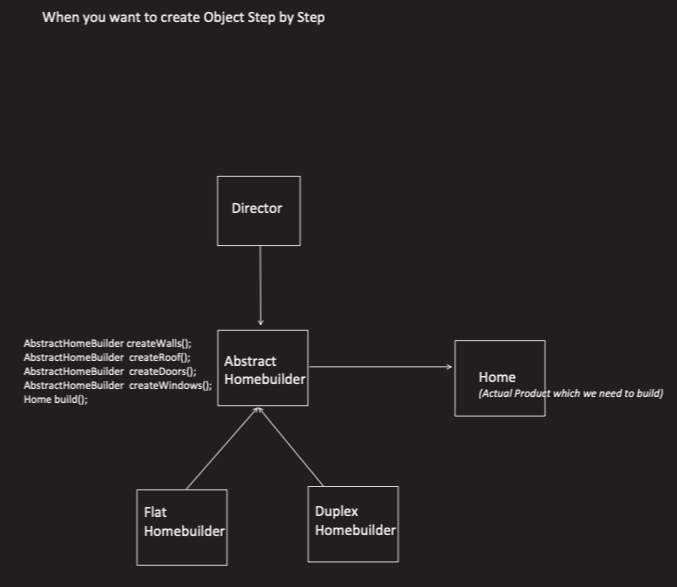
****

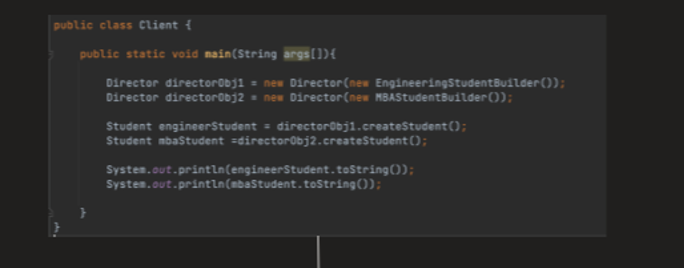
**Abstract Factory**

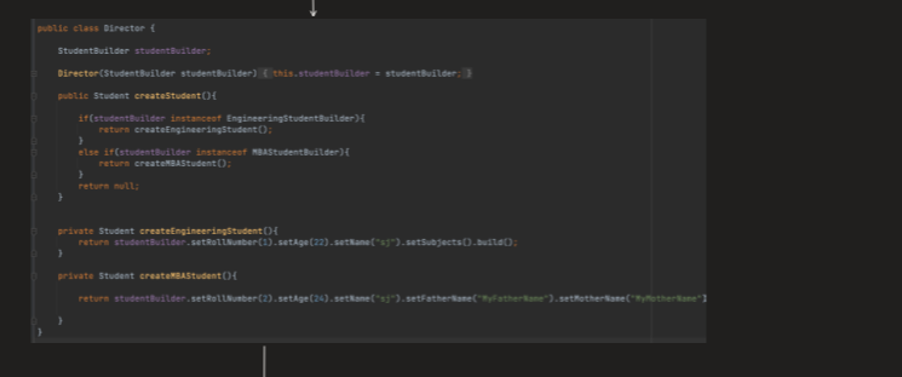
**It’s a Factory of Factory**

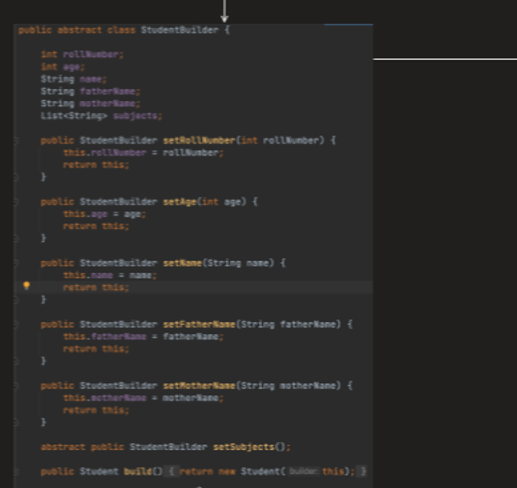
**  
  
**

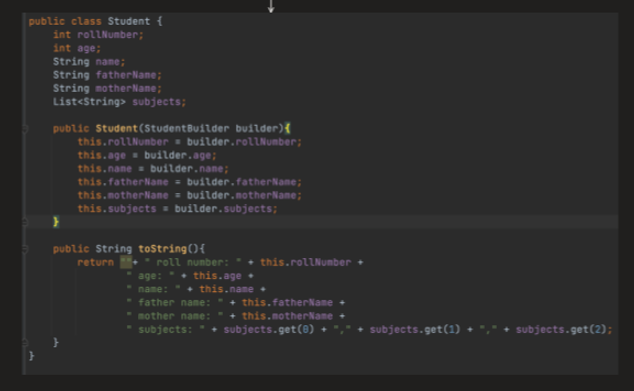
**Builder Pattern  
  
When we want to create an obj step by step**

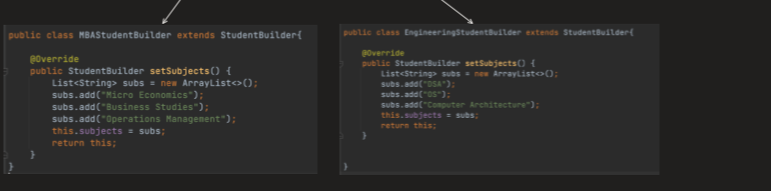
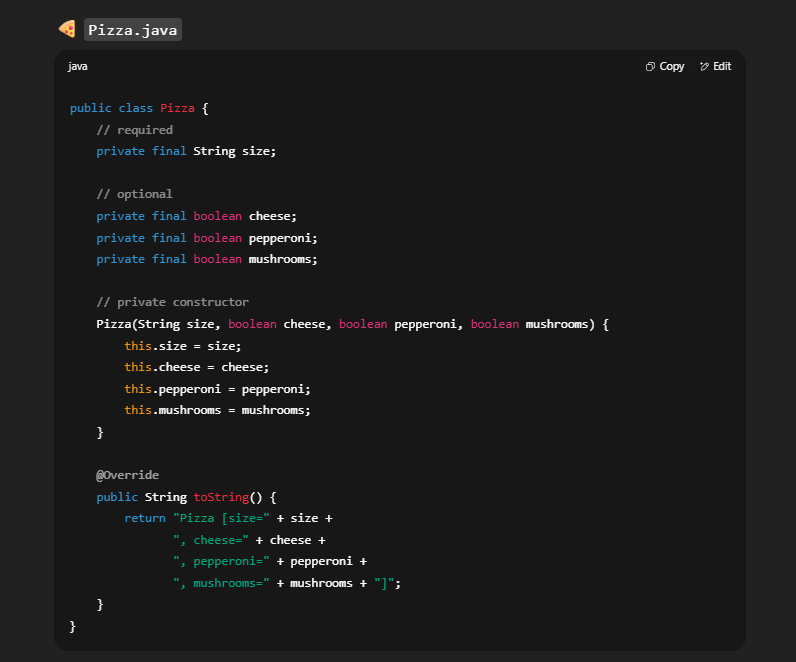
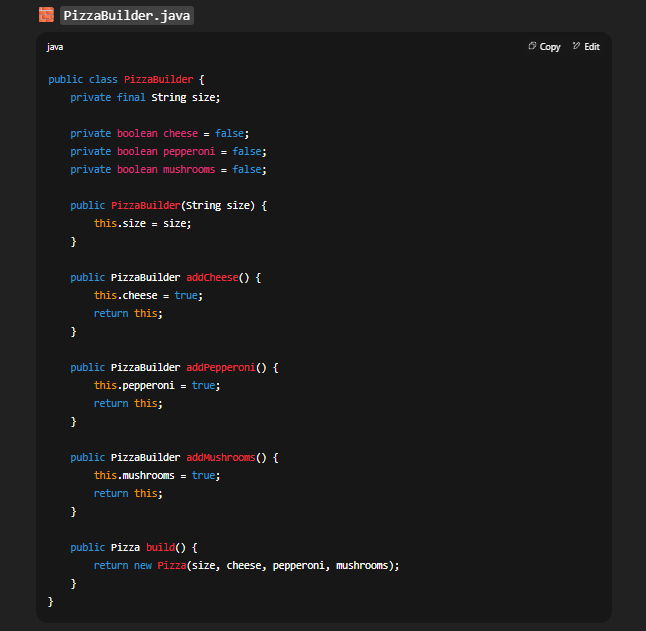
****

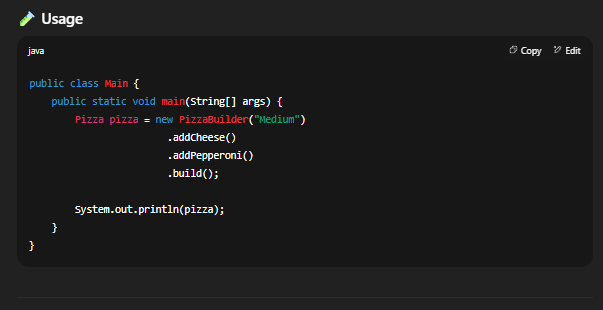
****

****

****

****

**  
  
Simpler example  
  
  
  
  
**

****