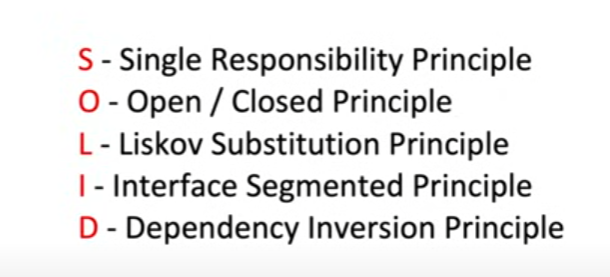
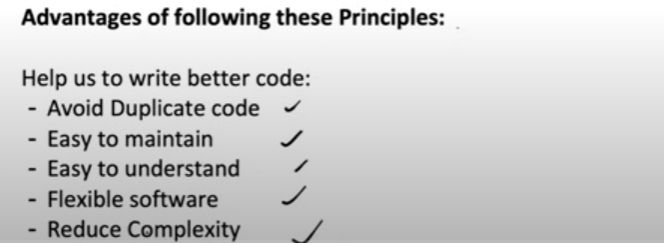
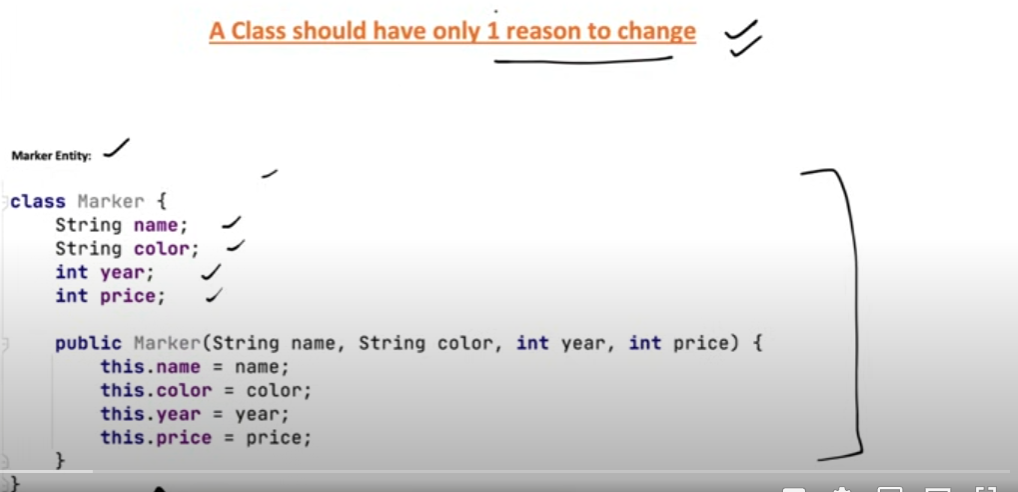
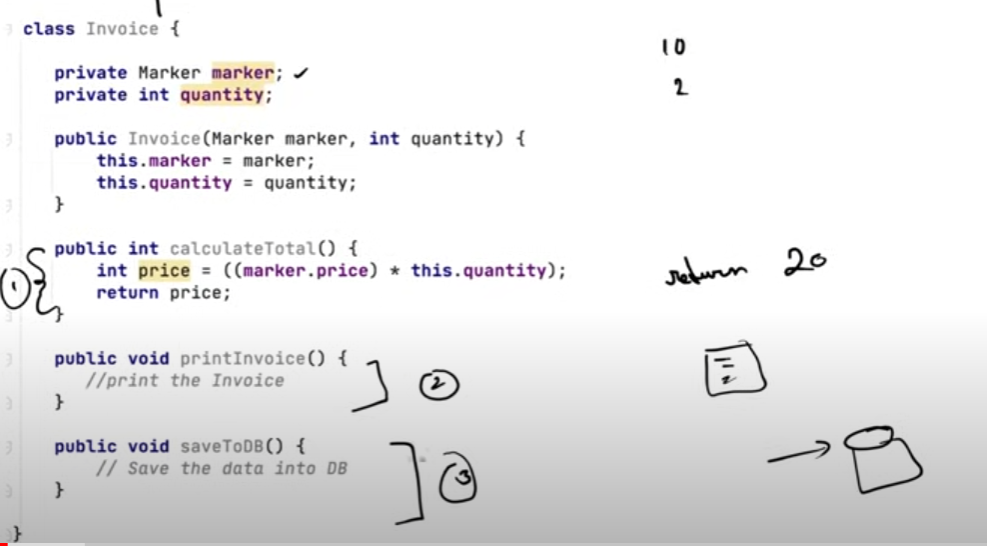
**SOLID PRINCIPLES**

****

****

**S Single Responsibility Principle**

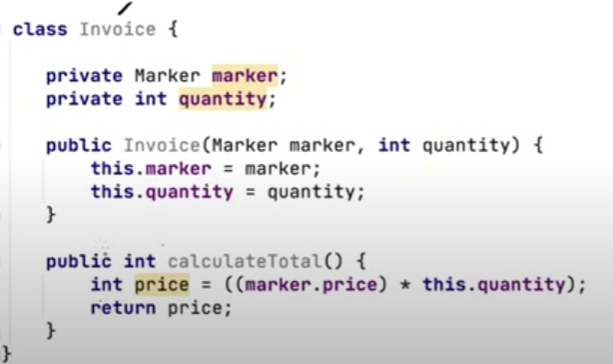
****

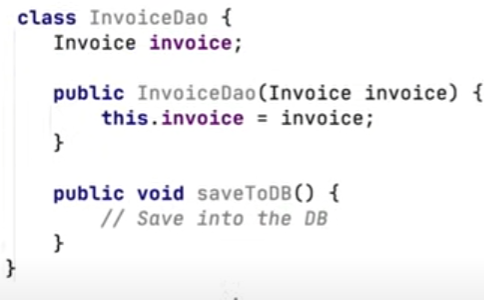
****Now in this case Invoice class can have 3 reasons to change ie:

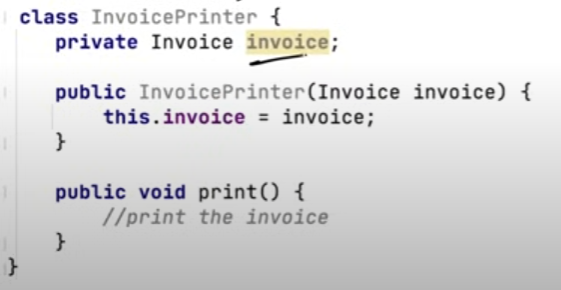
1. The calculateTotal method can have a change in logic where gst or some discount comes into picture.
2. The printInvoice impl or logic can change
3. In saveToDB() we might change the idea and save to a file

Therefore this Invoice class is not following Single Resp Principle.  
  
**Solution:**

**Make separate classes for each func**

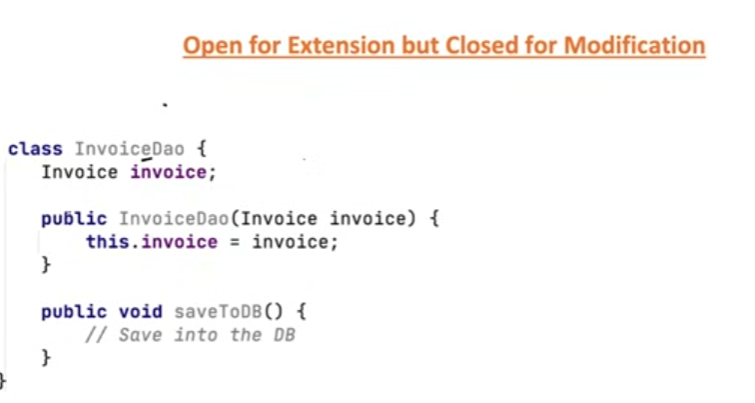




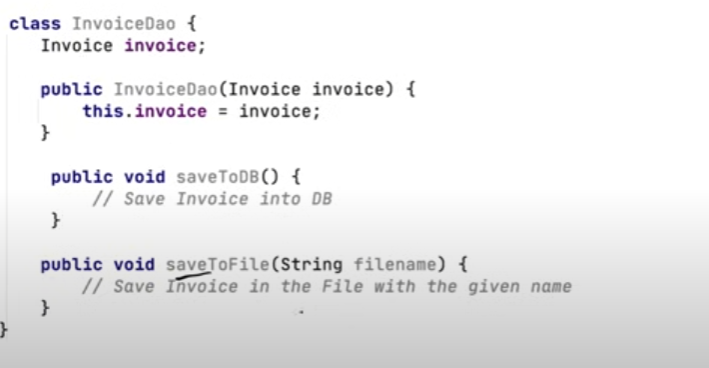


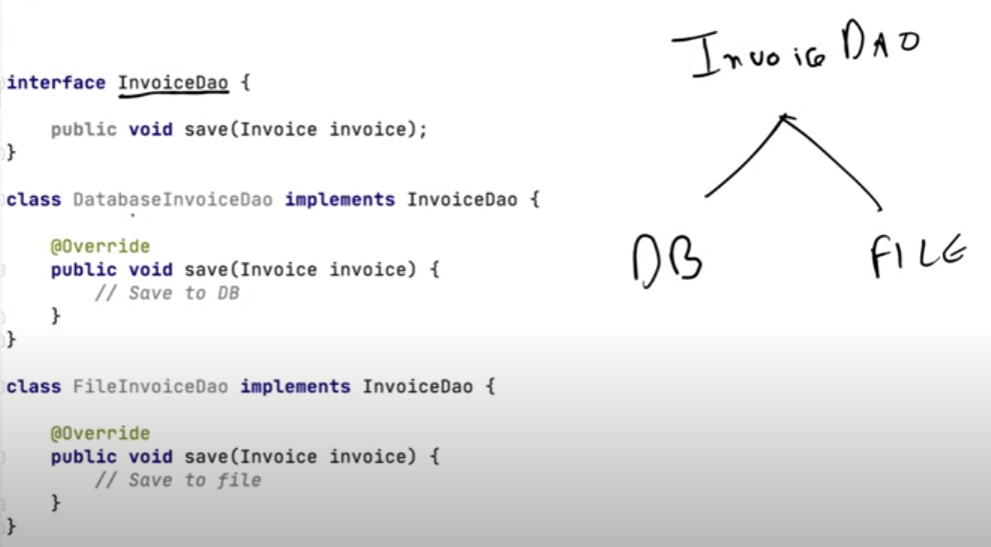
**Therefore now it is easy to maintain**

**O – Open/Closed Principle**

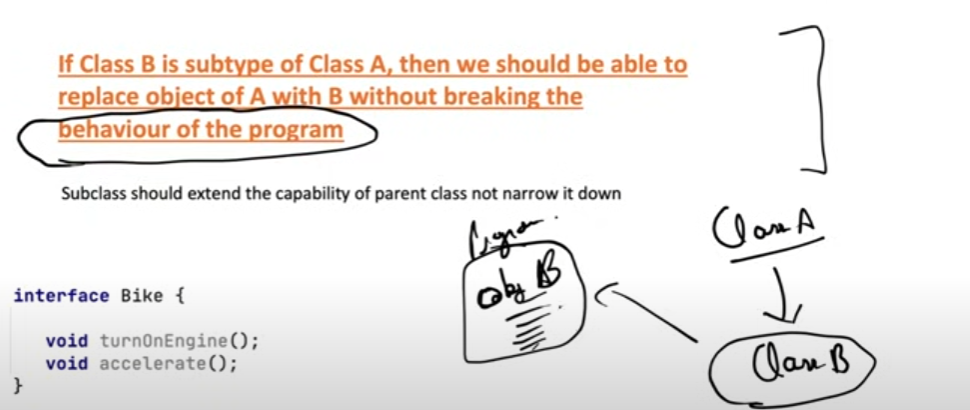
****

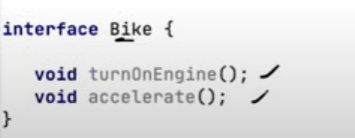
**Now this file we created previously and is tested and is being used in app. Now there is a new request to save it to file as well.**

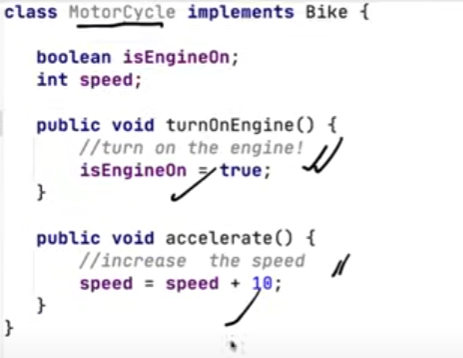
**Now if we solve this req as below:  
  
This will not follow the O principle as it is prone to bugs  
  
  
  
  
  
Soln  
  
Create a interface and then put both the impls**

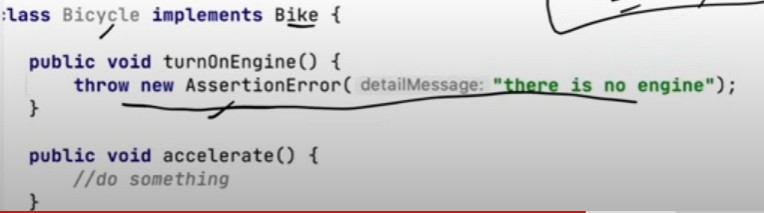
****

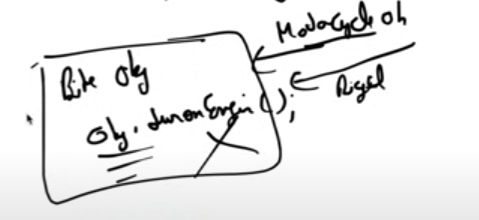
**L Liskov Substitution Principle**

****

****

****

**  
Now in this example we are limiting the capability by throwing an exception**

**  
Suppose for a piece of code we were getting a motorcycle object since for the existing code now, some day we pass Bicycle now wen turnOnEngine for Bicycle will be called it will throw an exception.**

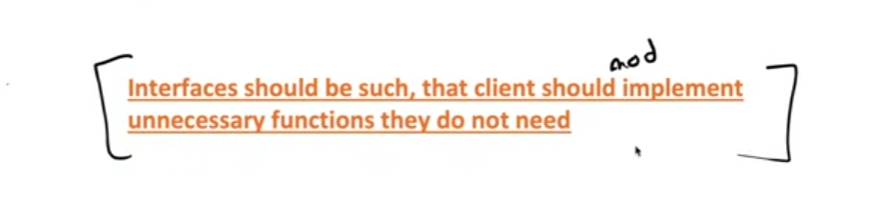
**Soln:**

**In order to resolve this create a separate class Bike with generic attributes and func accelerate () and a separate interface named checkEngine having method turnOnEngine()**

**Motorcycle extends Bike and implements checkEngine {}**

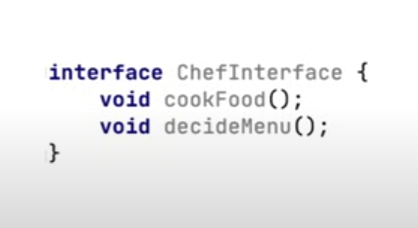
**Bicycle extends Bike{}**

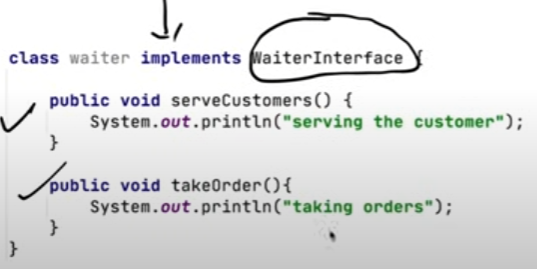
**I Interface Segregation Principle**

****

**  
Now when waiter class implements RestaurantEmployee it will have to implement all the methods. But it is not the job of a waiter to cookFood  
  
  
Therefore the interface RestaurantEmployee can be broken into :**

****

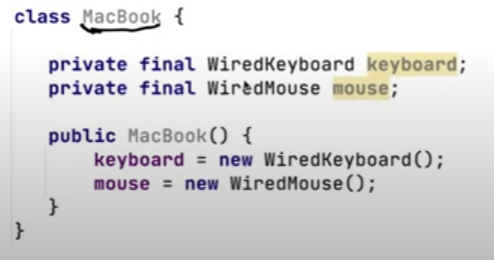
****

****

**D Dependency Inversion Principle**

****

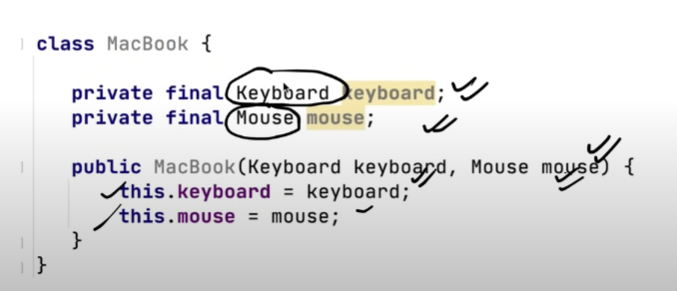
****

****

**Now suppose we have a class for MacBook. In this way of writing code the problem is in future if we want to create a obj of Mackbook where my keyboard and mouse would be wireless , it will be a prob as we are using a concrete class.**

**Soln**

**In order to resolve this we will use interface classes. And we will assign the type of keyboard or mouse using constructor injection**

****