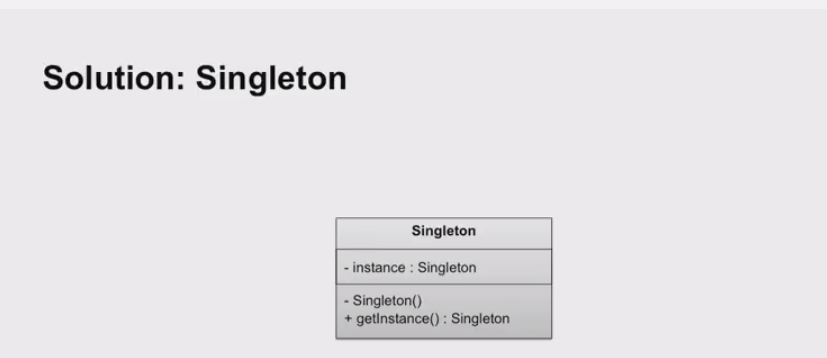
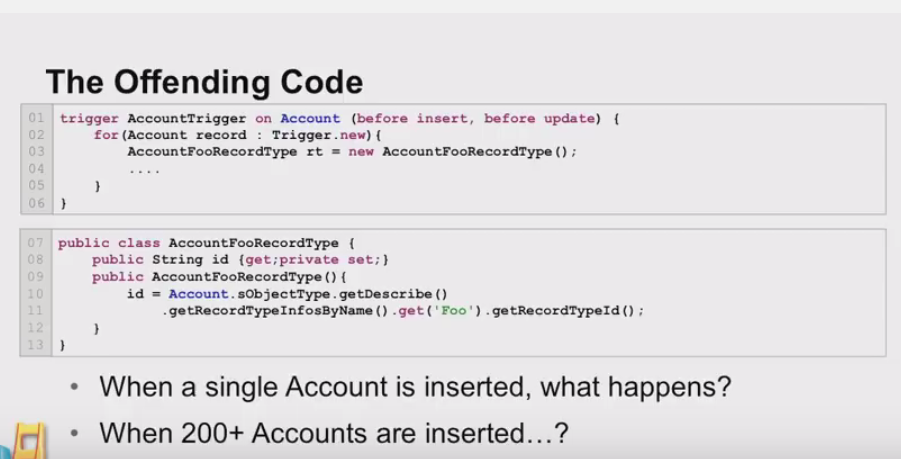
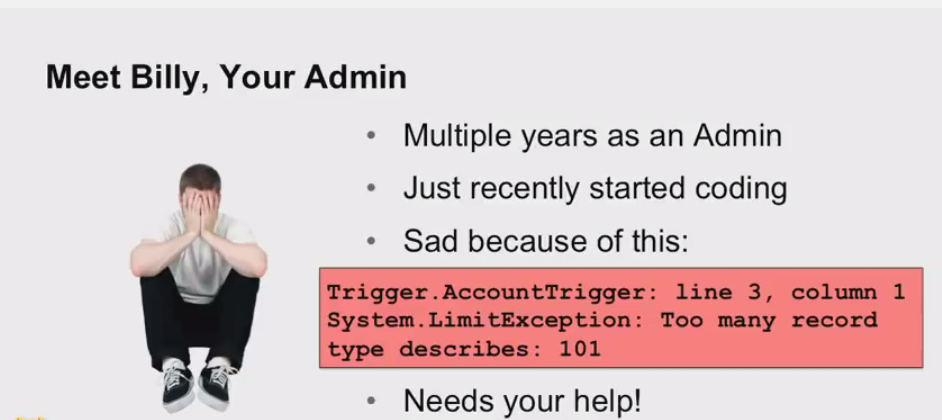
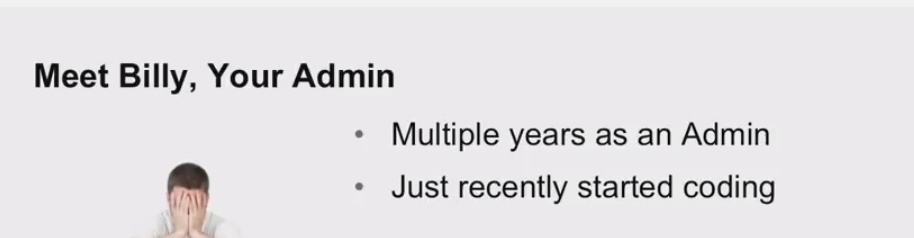
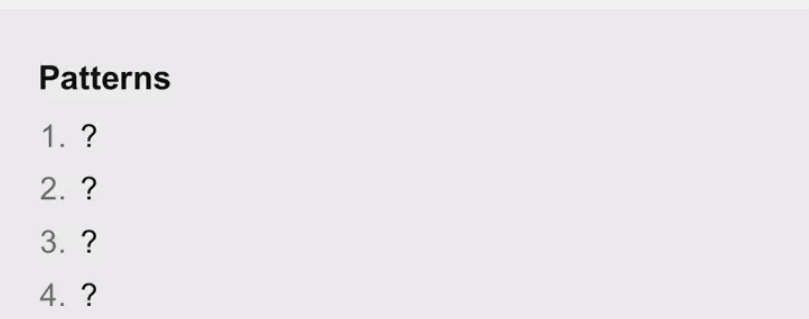
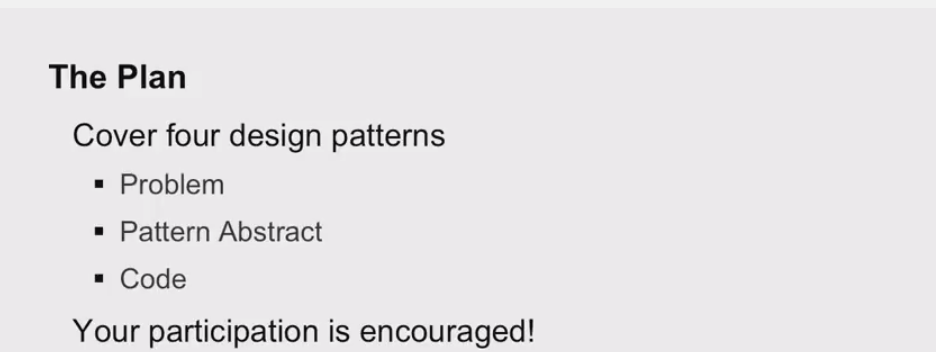
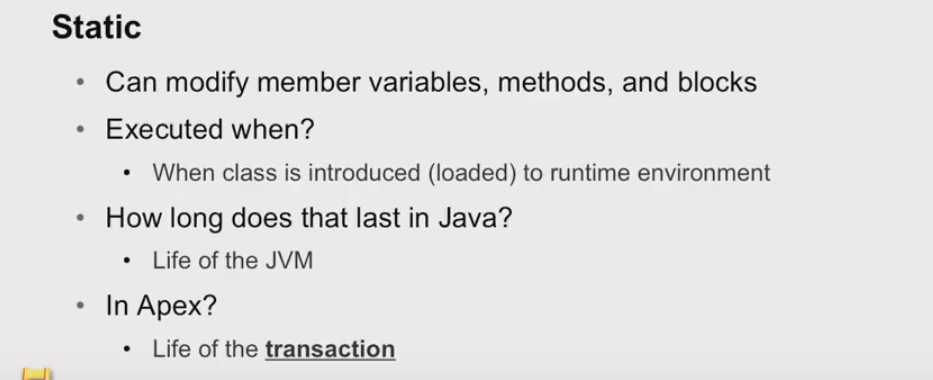
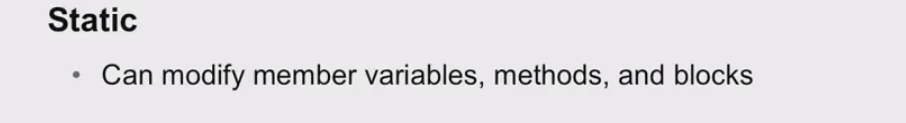
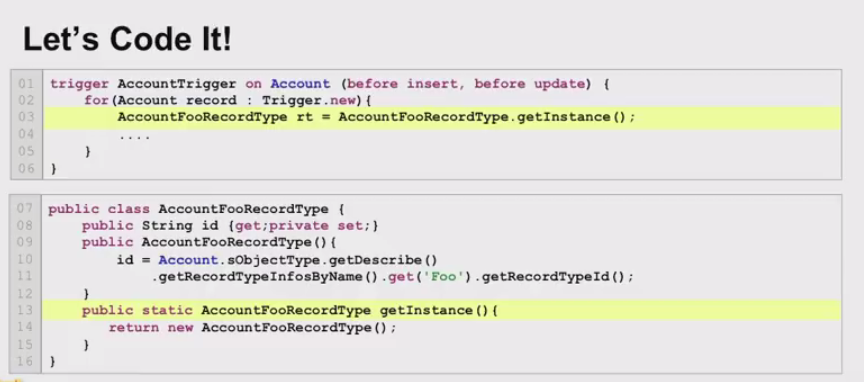
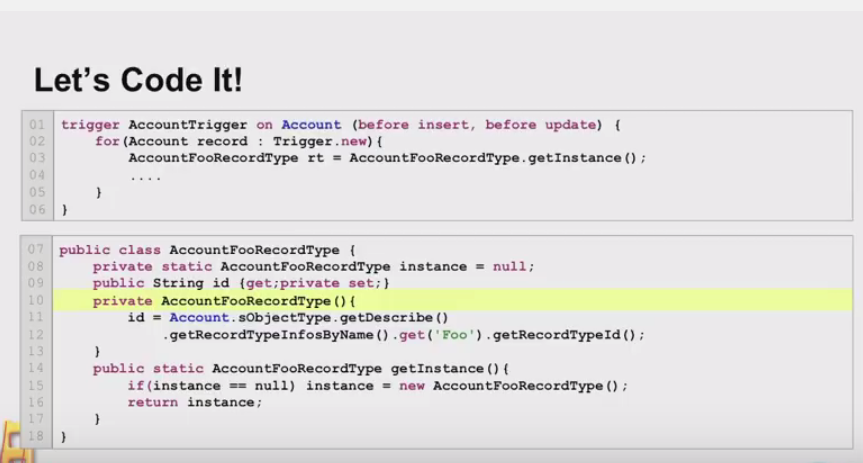
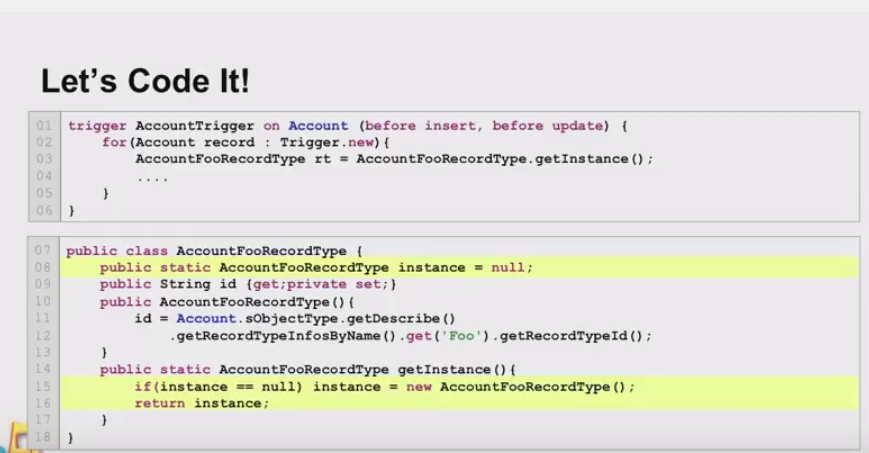
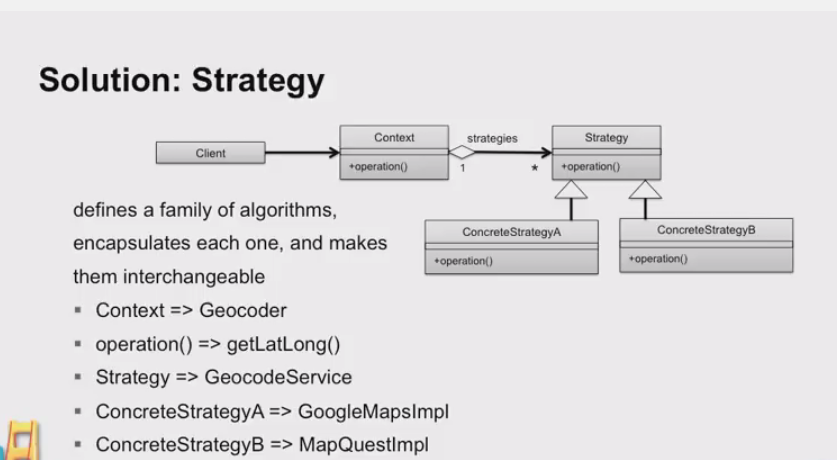
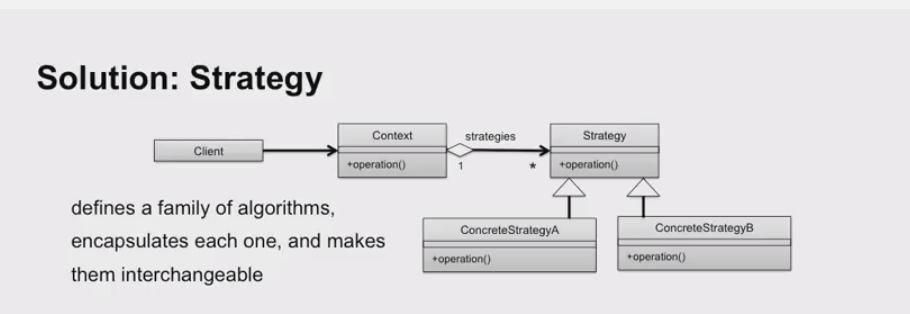
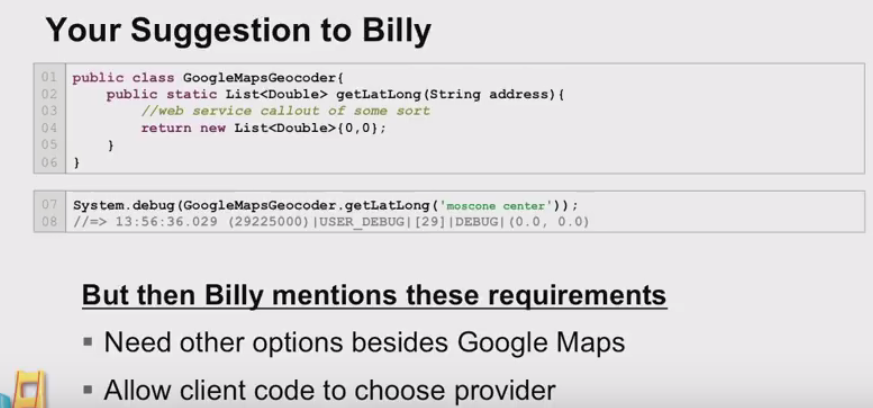
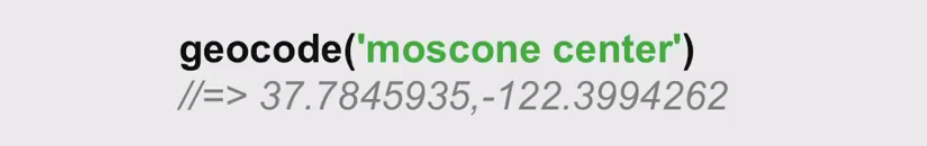
**APEX Design Patterns:**

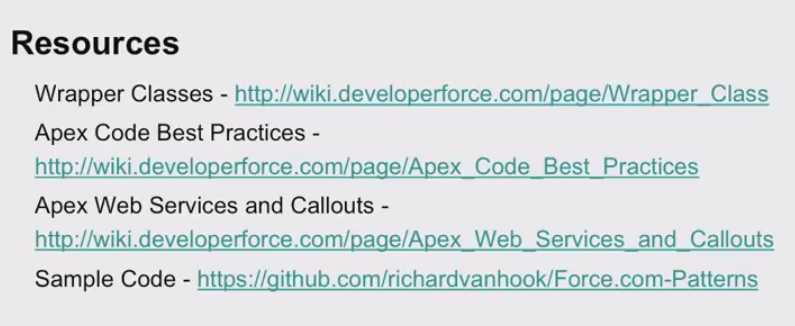
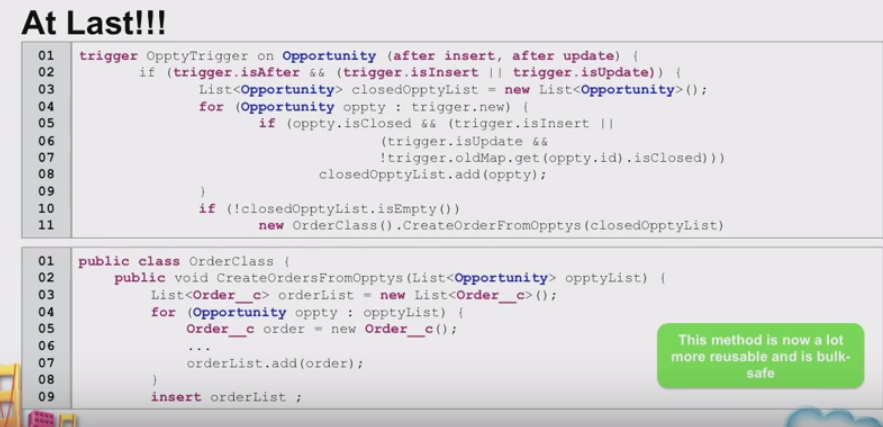
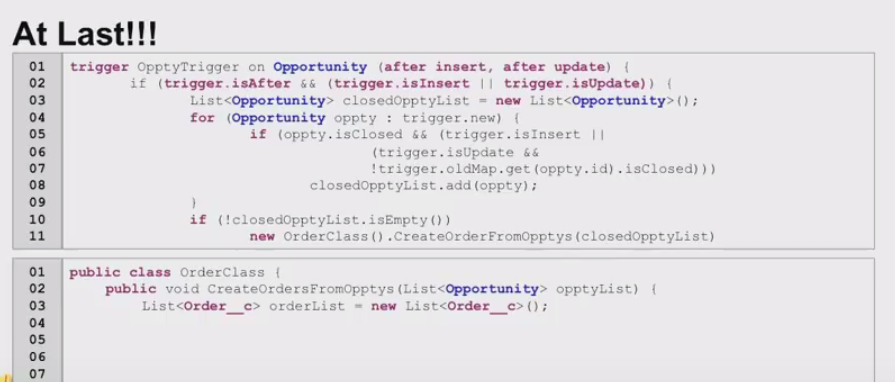
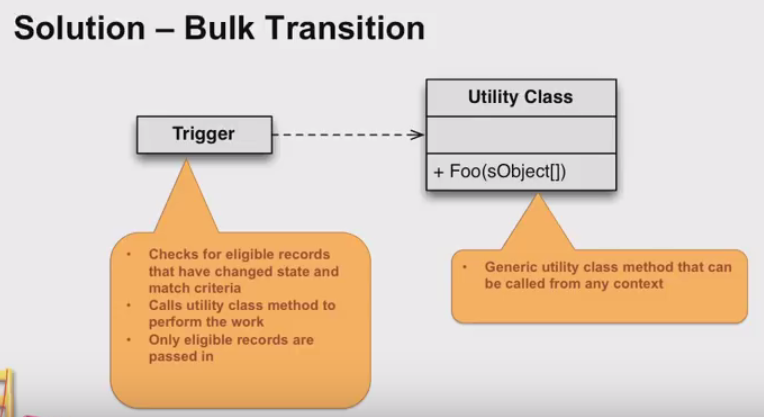
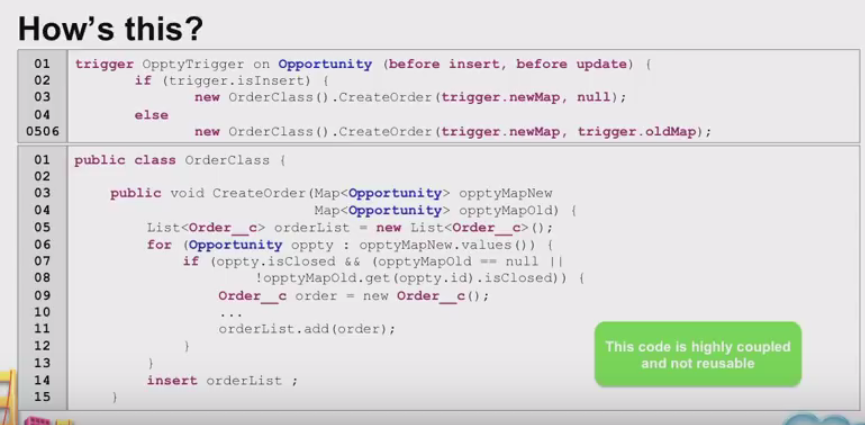
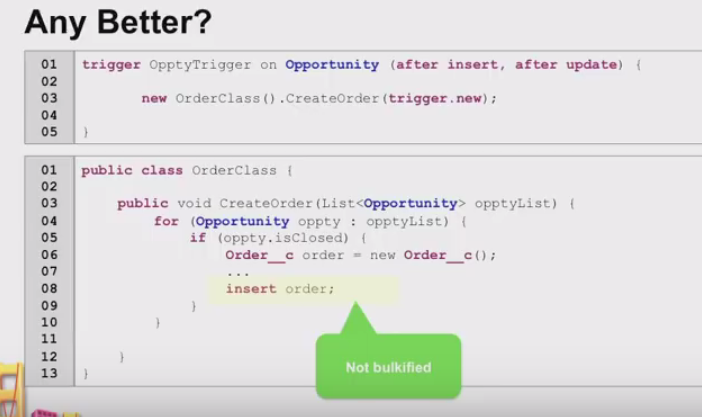
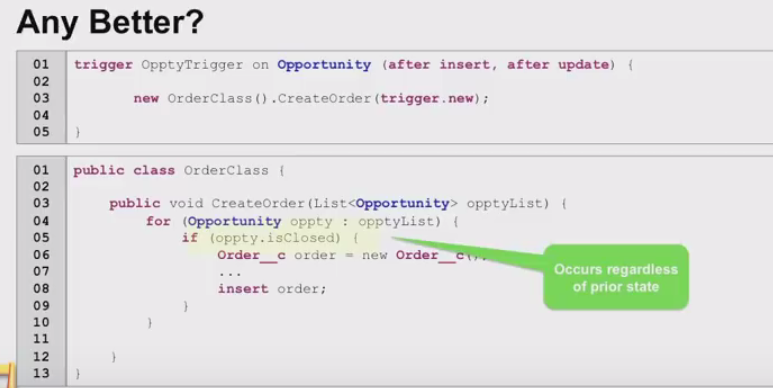
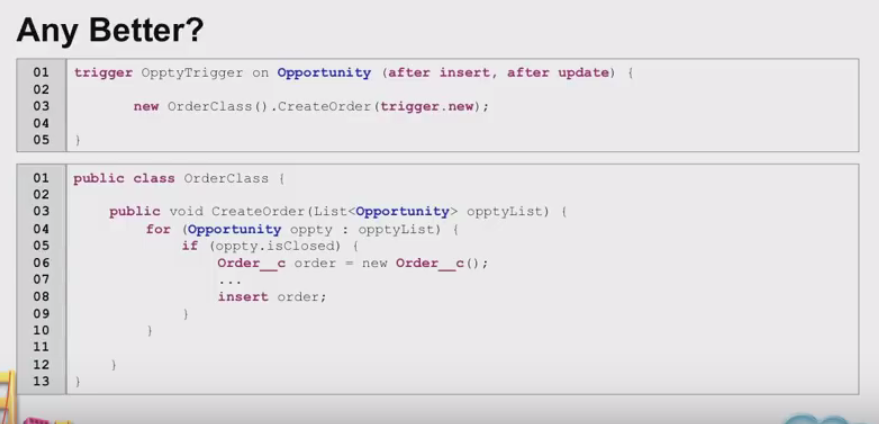
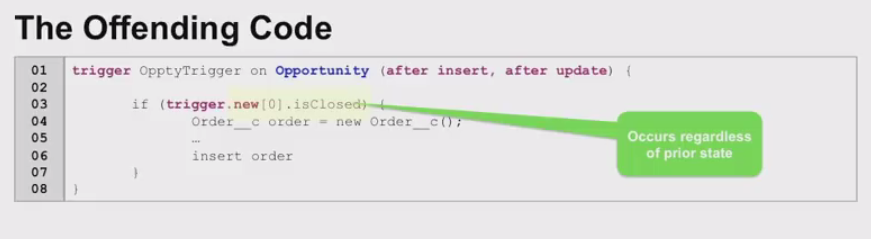
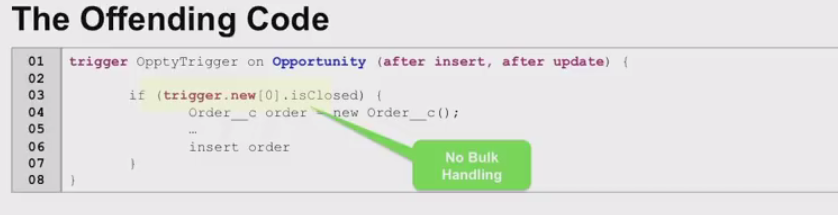
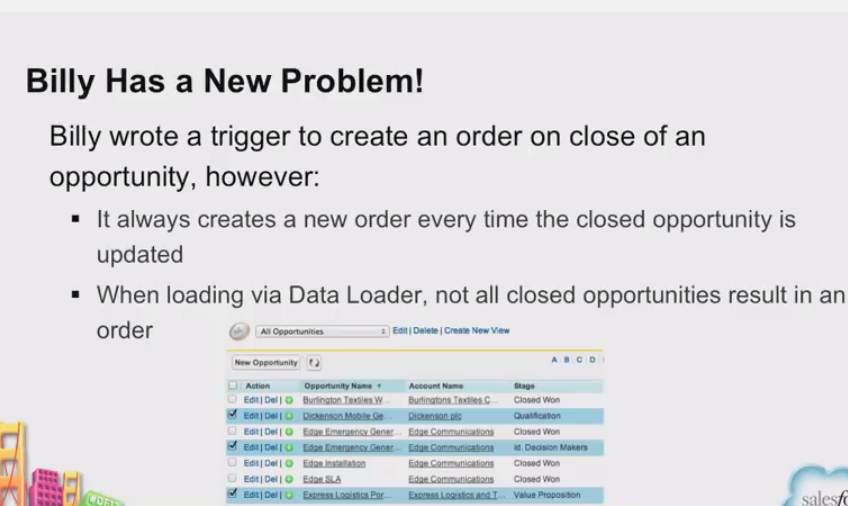
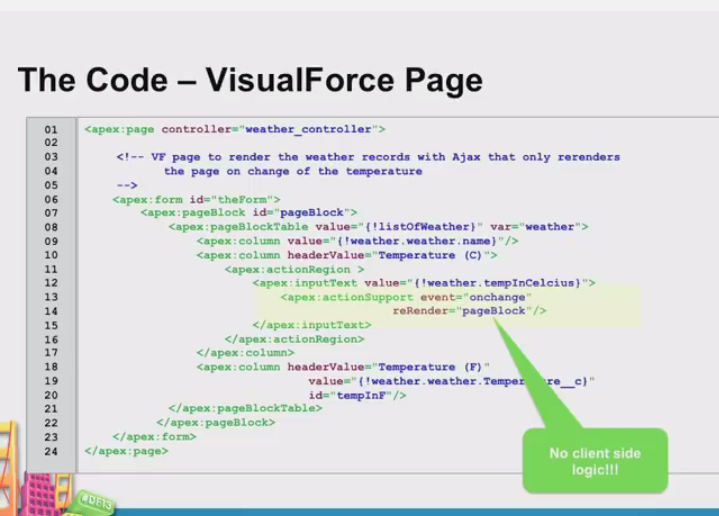
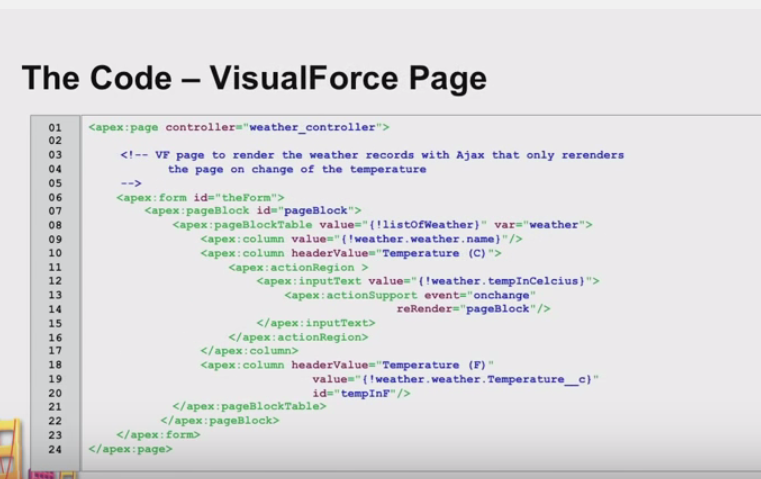
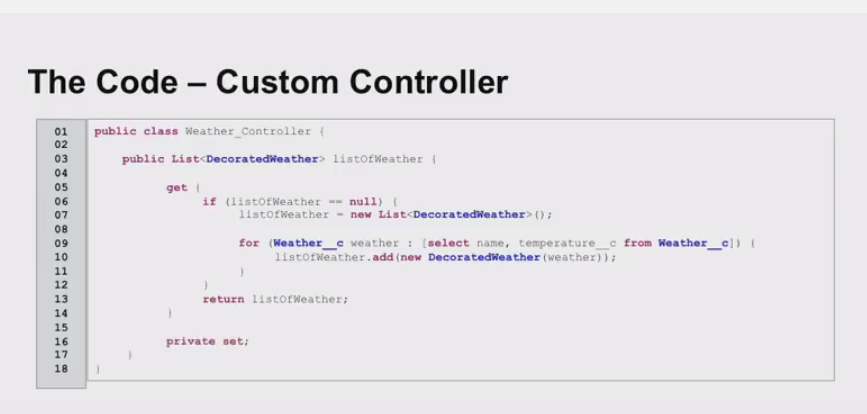
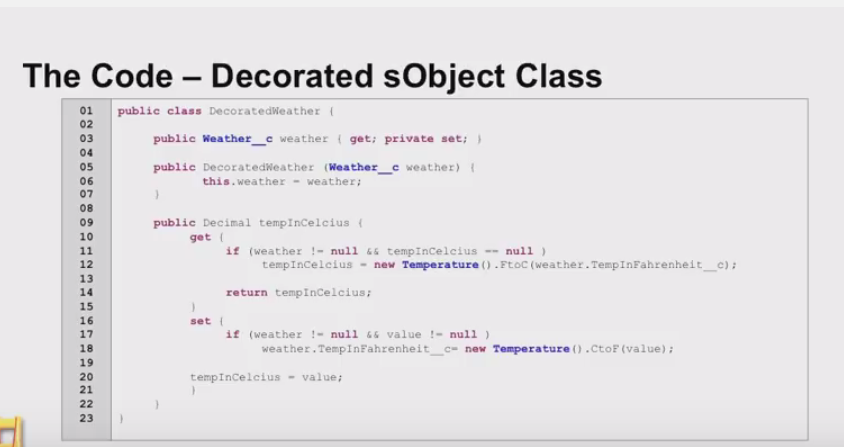
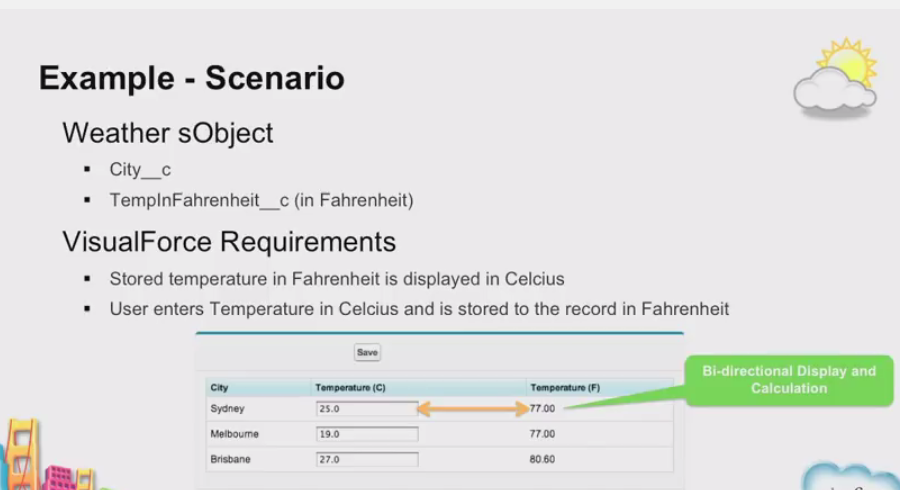
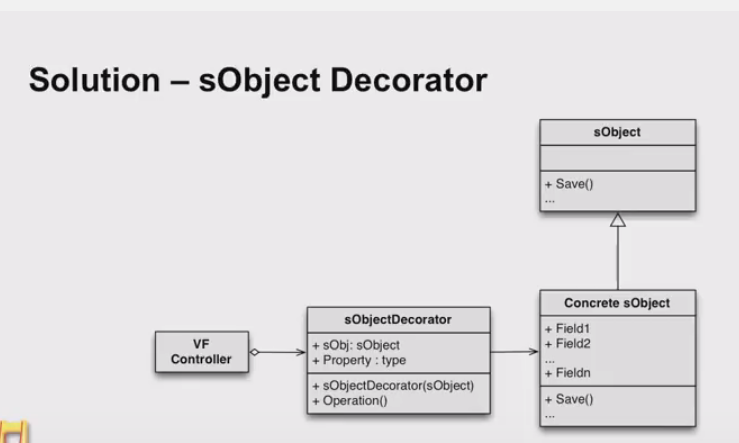
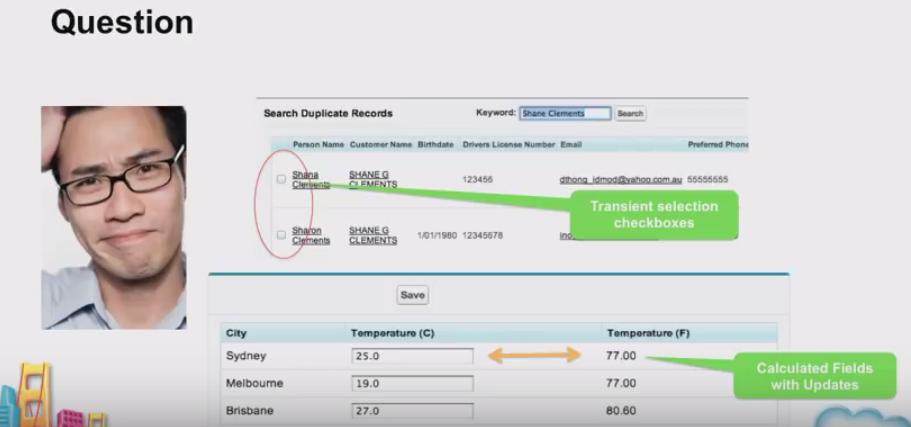
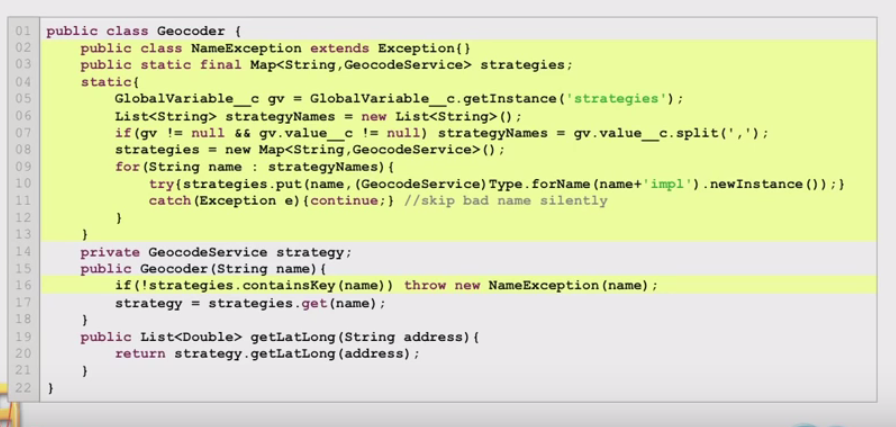
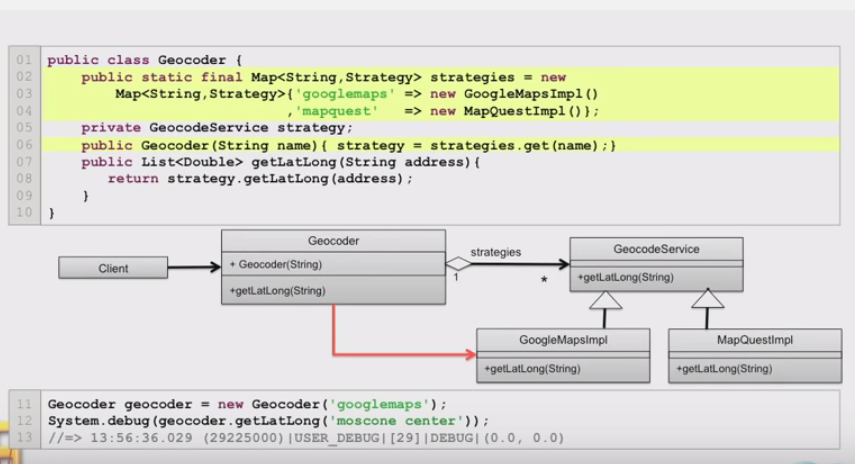
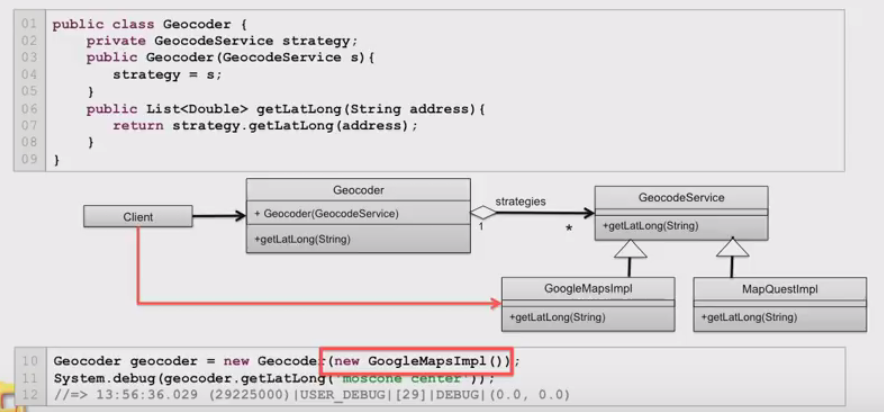
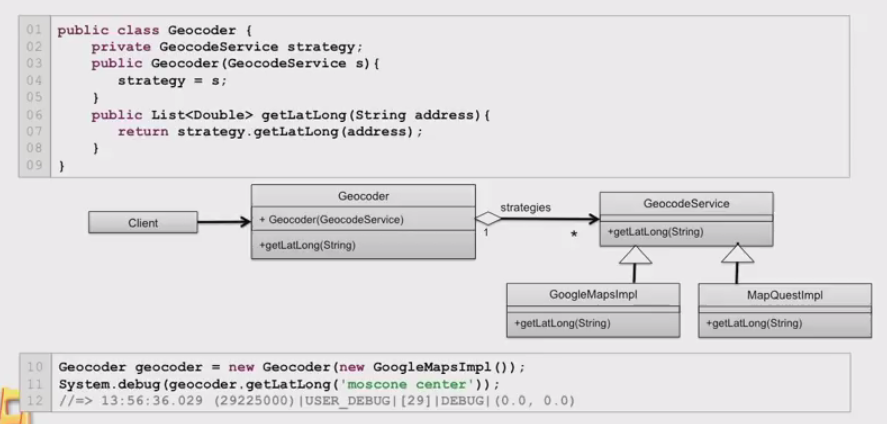
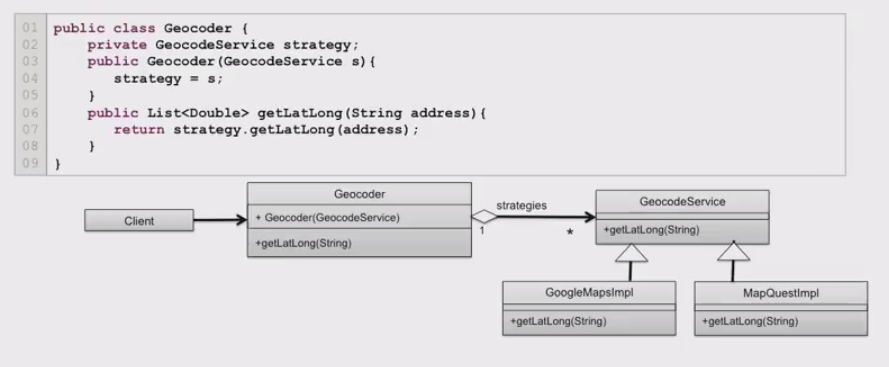


Singleton: Make sure you have one instance of the class any time in a transaction









------------------------------------------------------------##########################---------------------------------------

# Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

## **1. Creational Design Pattern**

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.

## **2. Structural Design Pattern**

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

## **3. Behavioral Design Pattern**

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

### Do you know?

* **Christopher Alexander** was the first person who invented all the above Design Patterns in 1977.
* But later the **Gang of Four - Design patterns, elements of reusable object-oriented software** book was written by a group of four persons named as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in 1995.
* That's why all the above 23 Design Patterns are known as **Gang of Four (GoF) Design Patterns.**

Strategy Design Pattern in Java – Example Tutorial

Posted by: [Pankaj Kumar](https://www.javacodegeeks.com/author/Pankaj-Kumar/) in [Core Java](https://www.javacodegeeks.com/category/java/core-java/) August 13th, 2013

**Strategy pattern** is one of the **behavioral design pattern**. Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

Strategy pattern is also known as **Policy Pattern**. We defines multiple algorithms and let client application pass the algorithm to be used as a parameter. One of the best example of this pattern is Collections.sort()method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways, check this post for sorting objects in java using [Java Comparable and Comparator](http://www.journaldev.com/780/java-comparable-and-comparator-example-to-sort-objects).

For our example, we will try to implement a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.

First of all we will create the interface for our strategy, in our case to pay the amount passed as argument.

|  |  |
| --- | --- |
| 1 | package com.journaldev.design.strategy; |
| 2 |  |

|  |  |
| --- | --- |
| 3 | public interface PaymentStrategy { |
| 4 |  |

|  |  |
| --- | --- |
| 5 | public void pay(int amount); |
| 6 | } |

Now we will have to create concrete implementations of algorithms for payment using credit/debit card or through paypal.

|  |  |
| --- | --- |
| 01 | package com.journaldev.design.strategy; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class CreditCardStrategy implements PaymentStrategy { |
| 04 |  |

|  |  |
| --- | --- |
| 05 | private String name; |
| 06 | private String cardNumber; |

|  |  |
| --- | --- |
| 07 | private String cvv; |
| 08 | private String dateOfExpiry; |

|  |  |
| --- | --- |
| 09 |  |
| 10 | public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){ |

|  |  |
| --- | --- |
| 11 | this.name=nm; |
| 12 | this.cardNumber=ccNum; |

|  |  |
| --- | --- |
| 13 | this.cvv=cvv; |
| 14 | this.dateOfExpiry=expiryDate; |

|  |  |
| --- | --- |
| 15 | } |
| 16 | @Override |

|  |  |
| --- | --- |
| 17 | public void pay(int amount) { |
| 18 | System.out.println(amount +" paid with credit/debit card"); |

|  |  |
| --- | --- |
| 19 | } |
| 20 |  |

|  |  |
| --- | --- |
| 21 | } |
| 01 | package com.journaldev.design.strategy; |

|  |  |
| --- | --- |
| 02 |  |
| 03 | public class PaypalStrategy implements PaymentStrategy { |

|  |  |
| --- | --- |
| 04 |  |
| 05 | private String emailId; |

|  |  |
| --- | --- |
| 06 | private String password; |
| 07 |  |

|  |  |
| --- | --- |
| 08 | public PaypalStrategy(String email, String pwd){ |
| 09 | this.emailId=email; |

|  |  |
| --- | --- |
| 10 | this.password=pwd; |
| 11 | } |

|  |  |
| --- | --- |
| 12 |  |
| 13 | @Override |

|  |  |
| --- | --- |
| 14 | public void pay(int amount) { |
| 15 | System.out.println(amount + " paid using Paypal."); |

|  |  |
| --- | --- |
| 16 | } |
| 17 |  |

|  |  |
| --- | --- |
| 18 | } |

Now our algorithms are ready and we can implement Shopping Cart and payment method will require input as Payment strategy.

|  |  |
| --- | --- |
| 01 | package com.journaldev.design.strategy; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class Item { |
| 04 |  |

|  |  |
| --- | --- |
| 05 | private String upcCode; |
| 06 | private int price; |

|  |  |
| --- | --- |
| 07 |  |
| 08 | public Item(String upc, int cost){ |

|  |  |
| --- | --- |
| 09 | this.upcCode=upc; |
| 10 | this.price=cost; |

|  |  |
| --- | --- |
| 11 | } |
| 12 |  |

|  |  |
| --- | --- |
| 13 | public String getUpcCode() { |
| 14 | return upcCode; |

|  |  |
| --- | --- |
| 15 | } |
| 16 |  |

|  |  |
| --- | --- |
| 17 | public int getPrice() { |
| 18 | return price; |

|  |  |
| --- | --- |
| 19 | } |
| 20 |  |

|  |  |
| --- | --- |
| 21 | } |
| 01 | package com.journaldev.design.strategy; |

|  |  |
| --- | --- |
| 02 |  |
| 03 | import java.text.DecimalFormat; |

|  |  |
| --- | --- |
| 04 | import java.util.ArrayList; |
| 05 | import java.util.List; |

|  |  |
| --- | --- |
| 06 |  |
| 07 | public class ShoppingCart { |

|  |  |
| --- | --- |
| 08 |  |
| 09 | //List of items |

|  |  |
| --- | --- |
| 10 | List<Item> items; |
| 11 |  |

|  |  |
| --- | --- |
| 12 | public ShoppingCart(){ |
| 13 | this.items=new ArrayList<Item>(); |

|  |  |
| --- | --- |
| 14 | } |
| 15 |  |

|  |  |
| --- | --- |
| 16 | public void addItem(Item item){ |
| 17 | this.items.add(item); |

|  |  |
| --- | --- |
| 18 | } |
| 19 |  |

|  |  |
| --- | --- |
| 20 | public void removeItem(Item item){ |
| 21 | this.items.remove(item); |

|  |  |
| --- | --- |
| 22 | } |
| 23 |  |

|  |  |
| --- | --- |
| 24 | public int calculateTotal(){ |
| 25 | int sum = 0; |

|  |  |
| --- | --- |
| 26 | for(Item item : items){ |
| 27 | sum += item.getPrice(); |

|  |  |
| --- | --- |
| 28 | } |
| 29 | return sum; |

|  |  |
| --- | --- |
| 30 | } |
| 31 |  |

|  |  |
| --- | --- |
| 32 | public void pay(PaymentStrategy paymentMethod){ |
| 33 | int amount = calculateTotal(); |

|  |  |
| --- | --- |
| 34 | paymentMethod.pay(amount); |
| 35 | } |

|  |  |
| --- | --- |
| 36 | } |

Notice that payment method of shopping cart requires payment algorithm as argument and doesn’t store it anywhere as instance variable.

Let’s test our setup with a simple program.

|  |  |
| --- | --- |
| 01 | package com.journaldev.design.strategy; |
| 02 |  |

|  |  |
| --- | --- |
| 03 | public class ShoppingCartTest { |
| 04 |  |

|  |  |
| --- | --- |
| 05 | public static void main(String[] args) { |
| 06 | ShoppingCart cart = new ShoppingCart(); |

|  |  |
| --- | --- |
| 07 |  |
| 08 | Item item1 = new Item("1234",10); |

|  |  |
| --- | --- |
| 09 | Item item2 = new Item("5678",40); |
| 10 |  |

|  |  |
| --- | --- |
| 11 | cart.addItem(item1); |
| 12 | cart.addItem(item2); |

|  |  |
| --- | --- |
| 13 |  |
| 14 | //pay by paypal |

|  |  |
| --- | --- |
| 15 | cart.pay(new PaypalStrategy("myemail@example.com", "mypwd")); |
| 16 |  |

|  |  |
| --- | --- |
| 17 | //pay by credit card |
| 18 | cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786","12/15")); |

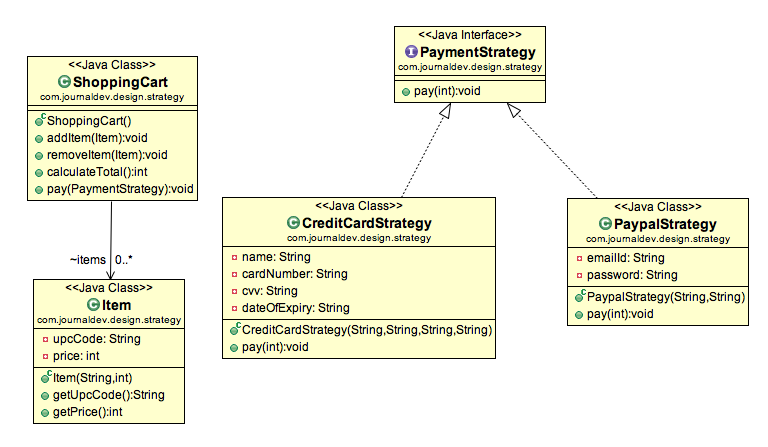
|  |  |
| --- | --- |
| 19 | } |
| 20 |  |

|  |  |
| --- | --- |
| 21 | } |

Output of above program is:

|  |  |
| --- | --- |
| 1 | 50 paid using Paypal. |
| 2 | 50 paid with credit/debit card |

Strategy Pattern Class Diagram

[](http://www.javacodegeeks.com/wp-content/uploads/2013/08/Strategy-Pattern.png)

Important Points

* We could have used composition to create instance variable for strategies but we should avoid that as we want the specific strategy to be applied for a particular task, same is followed in Collections.sort() and Arrays.sort() method that take comparator as argument.
* Strategy Pattern is very similar to [**State Pattern**](http://www.javacodegeeks.com/2013/08/state-design-pattern-in-java-example-tutorial.html). One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state whereas in strategy pattern strategy is passed as argument to the method and context object doesn’t have any variable to store it.
* Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to chose any of the algorithm at runtime for specific task.

That’s all for Strategy Pattern in java, I hope you liked it.