

## Lesson 04 Demo 04

### Creating and Using Design Patterns Factory and State

**Objective:** To implement the Factory and State design patterns

**Tools required:** Eclipse IDE

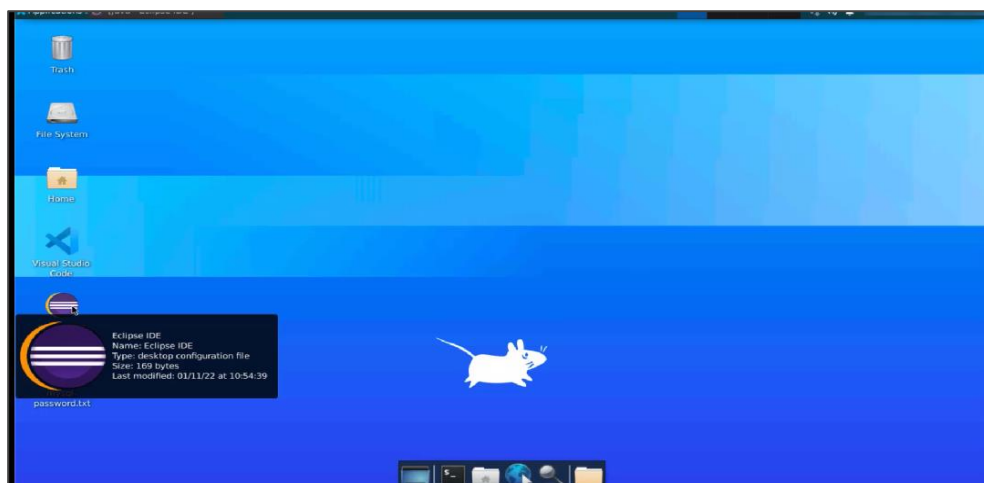
**Prerequisite:** None

**Steps to be followed:**

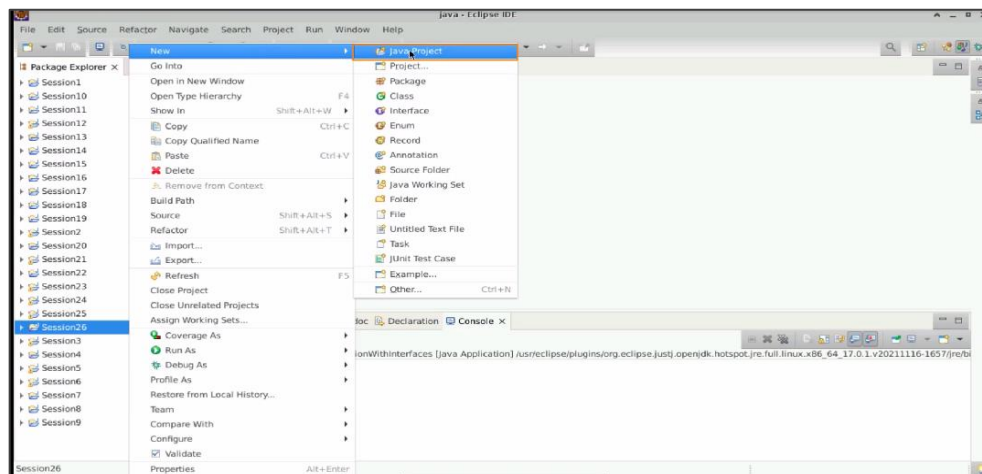
1. Implement and create Factory Design Pattern class and Plan interface
2. Create class Plan3G
3. Create class Plan4G and Plan5G
4. Create class Plan Factory
5. Implement the methods of the interface and classes
6. Create State Design Patterns

#### Step 1: Implement and create Factory Design Pattern class and Plan interface

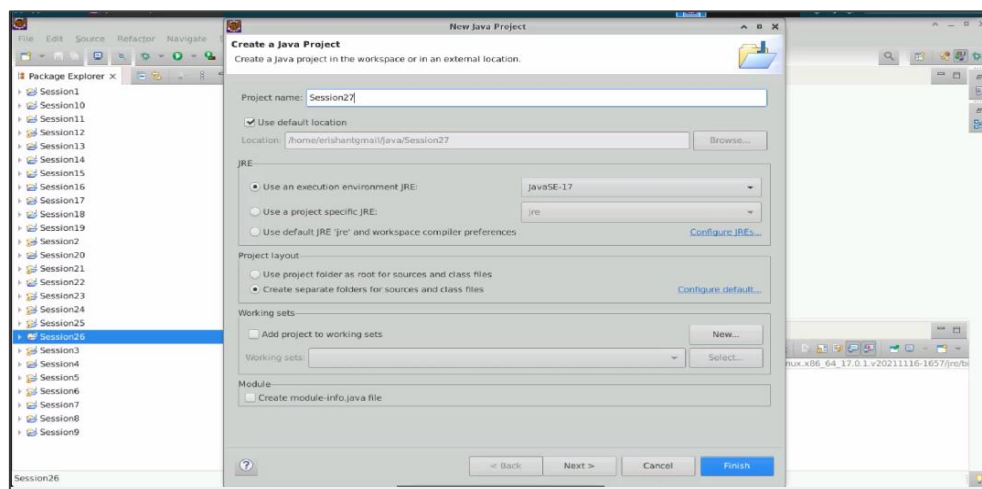
##### 1.1 Open the Eclipse IDE.



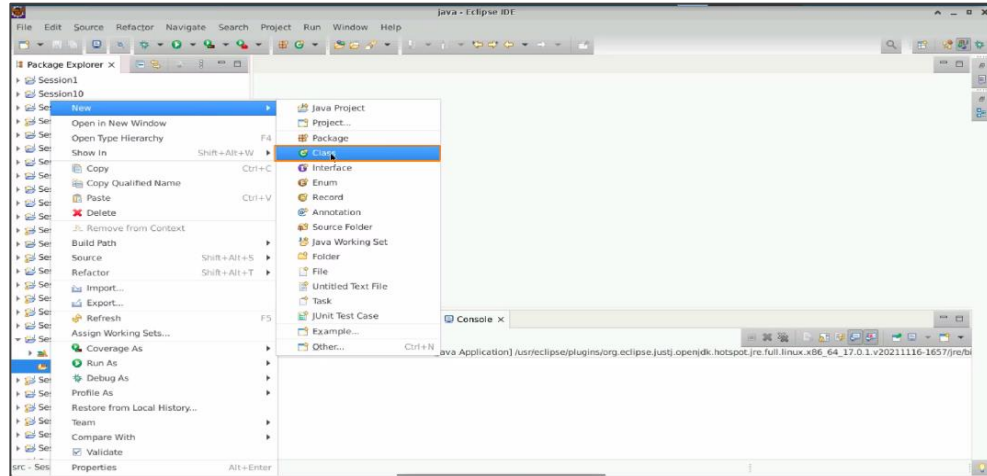
## 1.2 Select **File**, then **New**, and then **Java Project**.



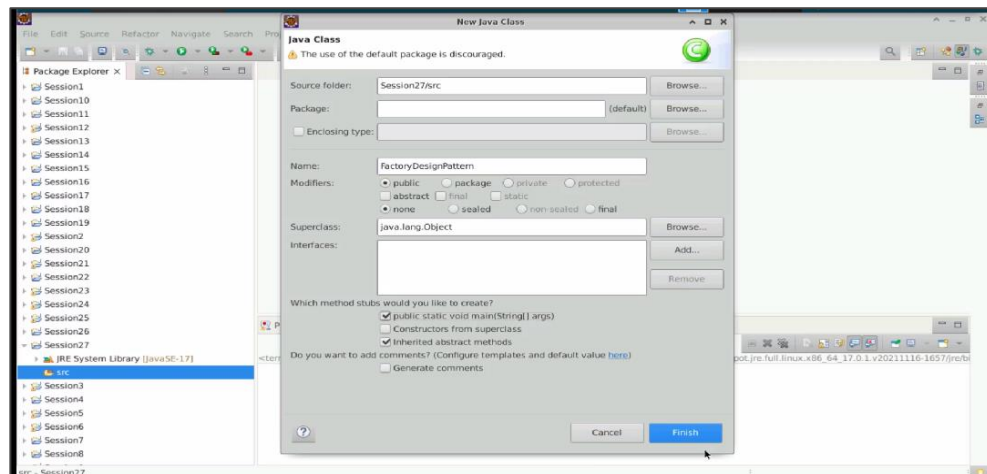
## 1.3 Name the project as **“Session27”** and select **Finish**.



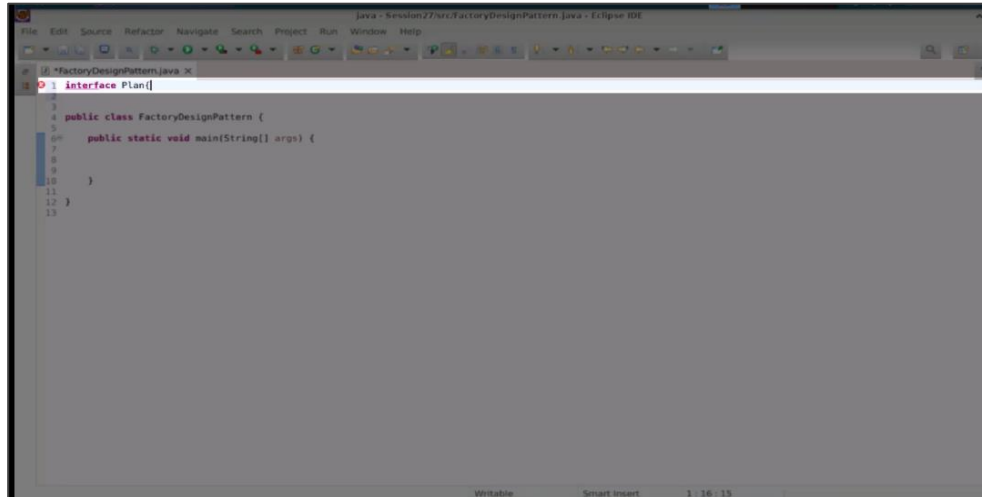
1.4 Right click on the **src** folder under the project name, select **New**, and then select **Class**.



1.5 Name the class as **FactoryDesignPattern** and select the option **public static void main(String[] args)**. Select **Finish**. This will create a java class with a **main()** method.



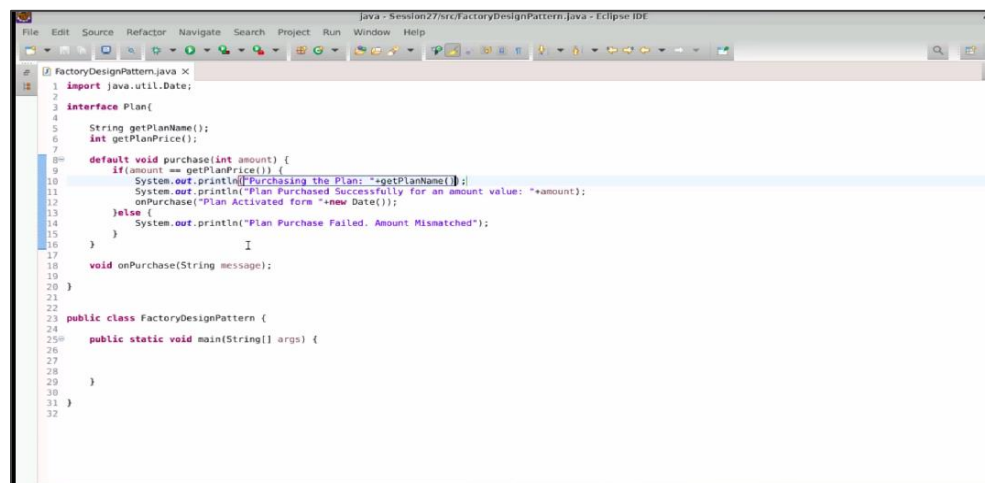
1.6 In the class **FactoryDesignPattern**, create an interface **Plan** by writing the **interface Plan**.



```
1 interface Plan {
2
3 }
4
5 public class FactoryDesignPattern {
6
7     public static void main(String[] args) {
8
9     }
10 }
11
12 }
13 }
```

1.7 Create four methods inside the interface. The methods are:

- **String getPlanName()** : This method will return the plan name as a string.
- **int getPlanPrice()** : This method will return the price of the selected plan as an integer value.
- **default void purchase(int amount)** : In the purchasing technique, you will first determine whether the amount is equal to determine the plan price.  
If your amount equals the plan price only then will you complete a transaction, regardless of the plan price and print the **Plan purchased successfully for the amount** and mention the amount using the **system.out.println()**. Else, print **Plan purchase failed. Amount Mismatched** in the **else** section.
- **void onPurchase(String message)** : This method will be executed in the **purchase()** method and will display a string message **Plan activated from** and the current date and time.



```

1  import java.util.Date;
2
3  interface Plan {
4      String getPlanName();
5      int getPlanPrice();
6
7      default void purchase(int amount) {
8          if (amount == getPlanPrice()) {
9              System.out.println("Purchasing the Plan: " + getPlanName());
10             System.out.println("Plan Purchased Successfully for an amount value: " + amount);
11             onPurchase("Plan Activated form " + new Date());
12         } else {
13             System.out.println("Plan Purchase Failed. Amount Mismatched");
14         }
15     }
16
17     void onPurchase(String message);
18 }
19
20
21
22
23 public class FactoryDesignPattern {
24
25     public static void main(String[] args) {
26
27     }
28
29 }
30
31
32

```

## Step 2: Create class Plan3G

2.1 Create a new class **Plan3G** which implements **Plan** interface as class **Plan3G** implements **Plan**.

```

1  import java.util.Date;
2
3  interface Plan {
4      String getPlanName();
5      int getPlanPrice();
6
7      default void purchase(int amount) {
8          if (amount == getPlanPrice()) {
9              System.out.println("Purchasing the Plan: " + getPlanName());
10             System.out.println("Plan Purchased Successfully for an amount value: " + amount);
11             onPurchase("Plan Activated form " + new Date());
12         } else {
13             System.out.println("Plan Purchase Failed. Amount Mismatched");
14         }
15     }
16
17     void onPurchase(String message);
18 }
19
20
21
22 class Plan3G implements Plan {
23 }
24
25
26
27 public class FactoryDesignPattern {
28
29     public static void main(String[] args) {
30
31     }
32
33 }
34
35
36

```

2.2 Click on the error bulb of **Plan3G** and select the option **Add unimplemented methods** to implement the methods of the plan interface.

```

1  import java.util.Date;
2
3  interface Plan {
4      String getPlanName();
5      int getPlanPrice();
6
7      default void purchase(int amount) {
8          if (amount == getPlanPrice()) {
9              System.out.println("Purchasing the Plan: " + getPlanName());
10             System.out.println("Plan Purchased Successfully for an amount value: " + amount);
11             onPurchase("Plan Activated form " + new Date());
12         } else {
13             System.out.println("Plan Purchase Failed. Amount Mismatched");
14         }
15     }
16
17     void onPurchase(String message);
18 }
19
20
21
22 class Plan3G implements Plan {
23 }
24
25
26
27 public class FactoryDesignPattern {
28
29     public static void main(String[] args) {
30
31     }
32
33 }
34
35
36

```

The type Plan3G must implement the inherited abstract method Plan.getPlanName()

3 methods to implement:

- Plan.getPlanName()
- Plan.getPlanPrice()
- Plan.onPurchase(...)

## 2.3 In `getPlanName()` method, write `return "Unlimited plan3G"`.

```

1  *FactoryDesignPattern.java X
2  3  String getPlanName();
3  4  int getPlanPrice();
4  5
5  6  default void purchase(int amount) {
6  7      if(amount == getPlanPrice()) {
7  8          System.out.println("Purchasing the Plan: "+getPlanName());
8  9          System.out.println("Plan Purchased Successfully for an amount value: "+amount);
9  10         onPurchase("Plan Activated form "+new Date());
10 11     } else {
11 12         System.out.println("Plan Purchase Failed. Amount Mismatched");
12 13     }
13 14 }
14 15
15 16 void onPurchase(String message);
16 17
17 18 }
18 19
19 20
20 21
21 22 class Plan3G implements Plan{
22 23
23 24     @Override
24 25     public String getPlanName() {
25 26         return "Unlimited Plan3G";
26 27     }
27 28
28 29     @Override
29 30     public int getPlanPrice() {
30 31         // TODO Auto-generated method stub
31 32         return 0;
32 33     }
33 34
34 35     @Override
35 36     public void onPurchase(String message) {
36 37         // TODO Auto-generated method stub
37 38     }
38 39 }
39 40
40 41
41 42
42 43

```

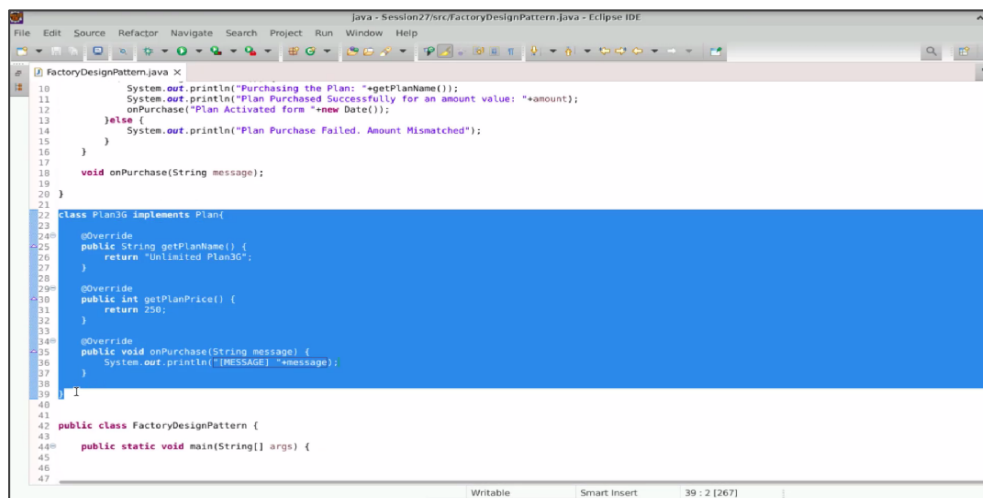
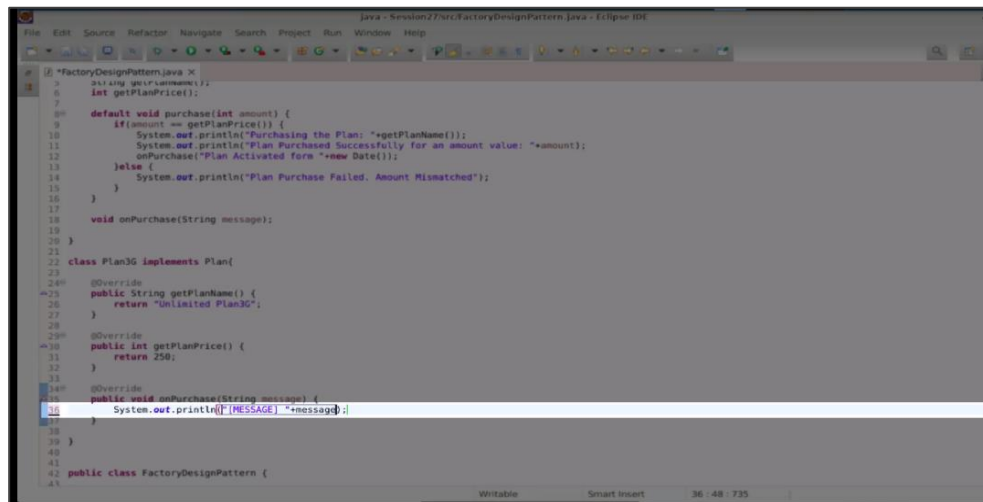
## 2.4 In the `getPlanPrice()` method, write `return 250`.

```

1  *FactoryDesignPattern.java X
2  3  String getPlanName();
3  4  int getPlanPrice();
4  5
5  6  default void purchase(int amount) {
6  7      if(amount == getPlanPrice()) {
7  8          System.out.println("Purchasing the Plan: "+getPlanName());
8  9          System.out.println("Plan Purchased Successfully for an amount value: "+amount);
9  10         onPurchase("Plan Activated form "+new Date());
10 11     } else {
11 12         System.out.println("Plan Purchase Failed. Amount Mismatched");
12 13     }
13 14 }
14 15
15 16 void onPurchase(String message);
16 17
17 18 }
18 19
19 20
20 21
21 22 class Plan3G implements Plan{
22 23
23 24     @Override
24 25     public String getPlanName() {
25 26         return "Unlimited Plan3G";
26 27     }
27 28
28 29     @Override
29 30     public int getPlanPrice() {
30 31         return 250;
31 32     }
32 33
33 34     @Override
34 35     public void onPurchase(String message) {
35 36         // TODO Auto-generated method stub
36 37     }
37 38 }
38 39
39 40
40 41
41 42
42 43

```

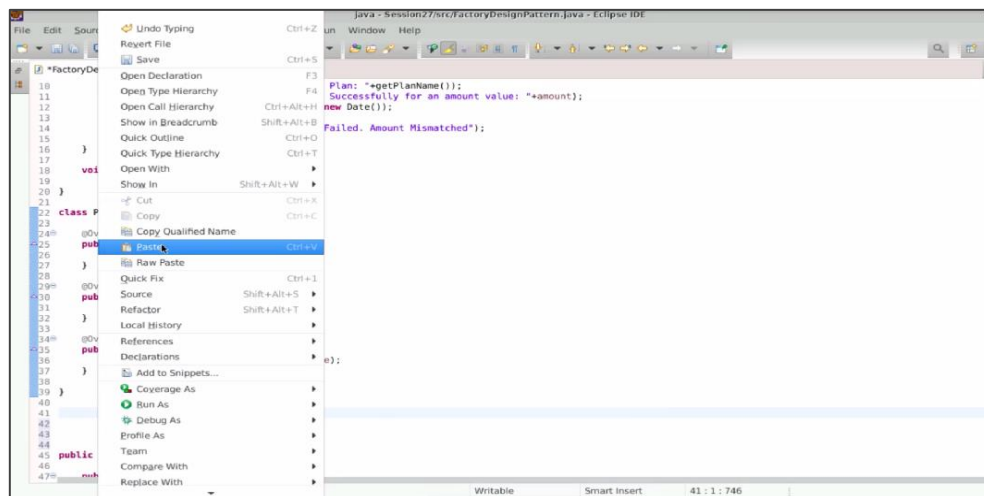
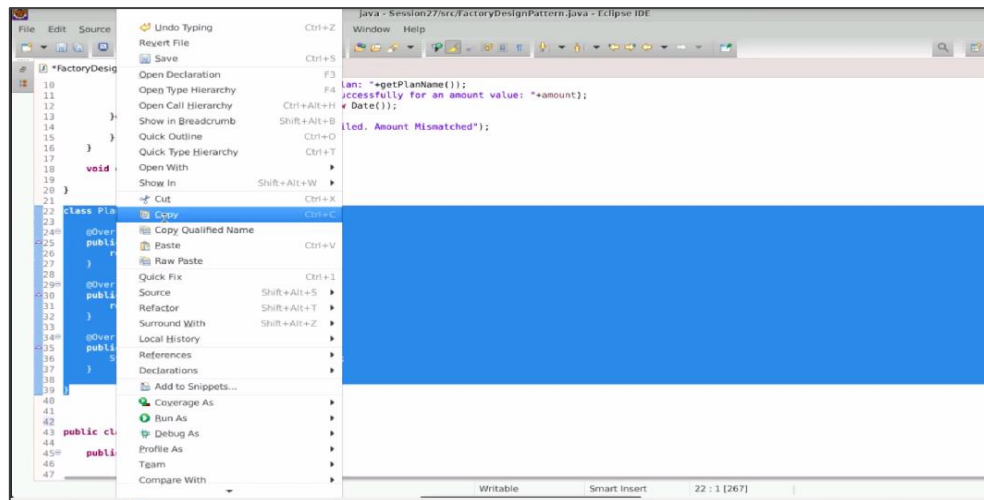
2.5 In the `onPurchase()` method, write `system.out.println("[MESSAGE] "+message)`

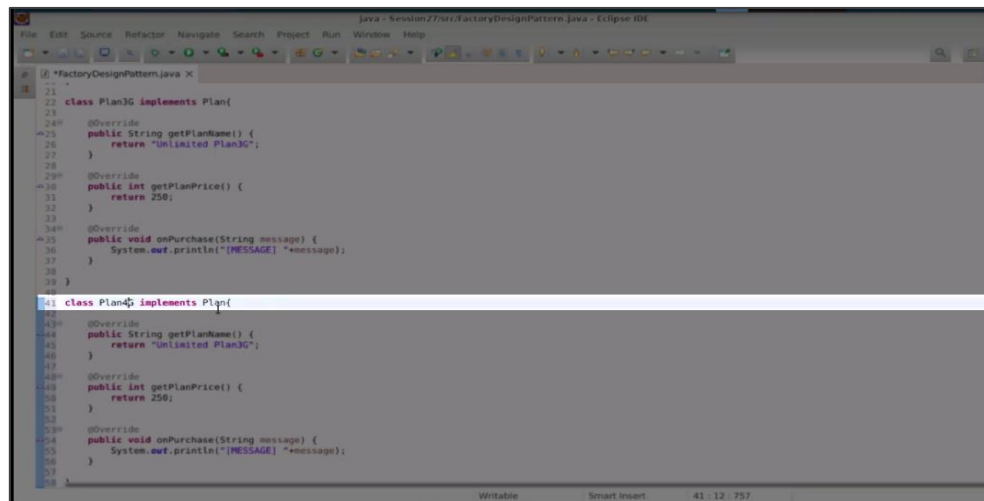




## Step 3: Create class Plan4G and Plan5G

### 3.1 Create class Plan4G by copying and pasting class Plan3G.



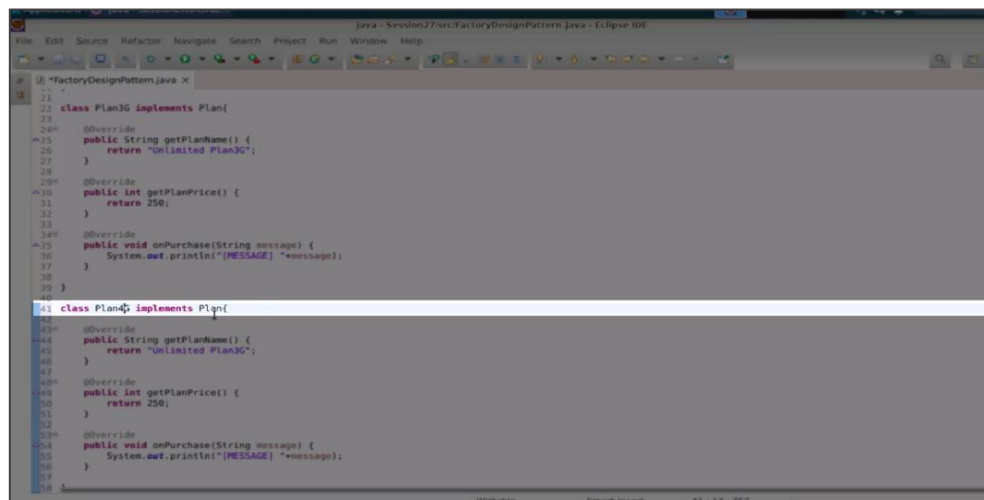


```

1  *FactoryDesignPattern.java
2
3  class Plan3G implements Plan{
4
5      @Override
6      public String getPlanName() {
7          return "Unlimited Plan3G";
8      }
9
10     @Override
11     public int getPlanPrice() {
12         return 250;
13     }
14
15     @Override
16     public void onPurchase(String message) {
17         System.out.println("[MESSAGE] "+message);
18     }
19 }
20
21 class Plan4G implements Plan{
22
23     @Override
24     public String getPlanName() {
25         return "Unlimited Plan3G";
26     }
27
28     @Override
29     public int getPlanPrice() {
30         return 250;
31     }
32
33     @Override
34     public void onPurchase(String message) {
35         System.out.println("[MESSAGE] "+message);
36     }
37 }
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

3.2 In `getPlanName()` method, write **return "Truly Unlimited plan4G"**.

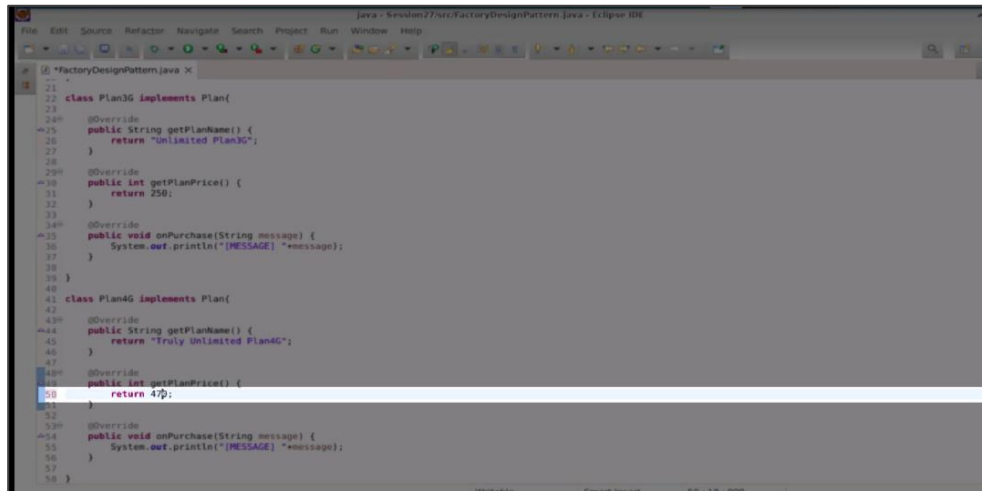


```

1  *FactoryDesignPattern.java
2
3  class Plan3G implements Plan{
4
5      @Override
6      public String getPlanName() {
7          return "Unlimited Plan3G";
8      }
9
10     @Override
11     public int getPlanPrice() {
12         return 250;
13     }
14
15     @Override
16     public void onPurchase(String message) {
17         System.out.println("[MESSAGE] "+message);
18     }
19 }
20
21 class Plan4G implements Plan{
22
23     @Override
24     public String getPlanName() {
25         return "Unlimited Plan3G";
26     }
27
28     @Override
29     public int getPlanPrice() {
30         return 250;
31     }
32
33     @Override
34     public void onPurchase(String message) {
35         System.out.println("[MESSAGE] "+message);
36     }
37 }
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

3.3 In the `getPlanPrice()` method, write **return 470**.

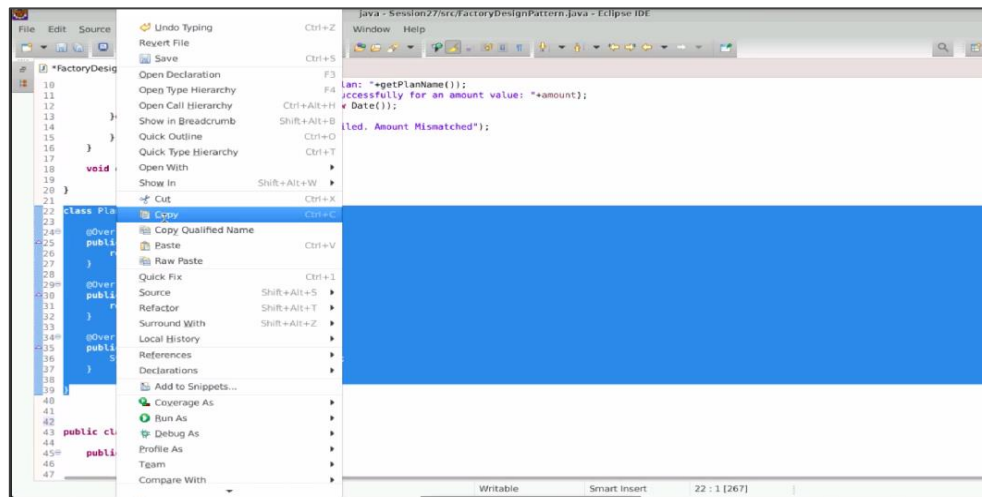


```

21
22 class Plan3G implements Plan {
23
24     @Override
25     public String getPlanName() {
26         return "Unlimited Plan3G";
27     }
28
29     @Override
30     public int getPlanPrice() {
31         return 250;
32     }
33
34     @Override
35     public void onPurchase(String message) {
36         System.out.println("[MESSAGE] " + message);
37     }
38 }
39
40 class Plan4G implements Plan {
41
42     @Override
43     public String getPlanName() {
44         return "Truly Unlimited Plan4G";
45     }
46
47     @Override
48     public int getPlanPrice() {
49         return 470;
50     }
51
52     @Override
53     public void onPurchase(String message) {
54         System.out.println("[MESSAGE] " + message);
55     }
56 }
57
58

```

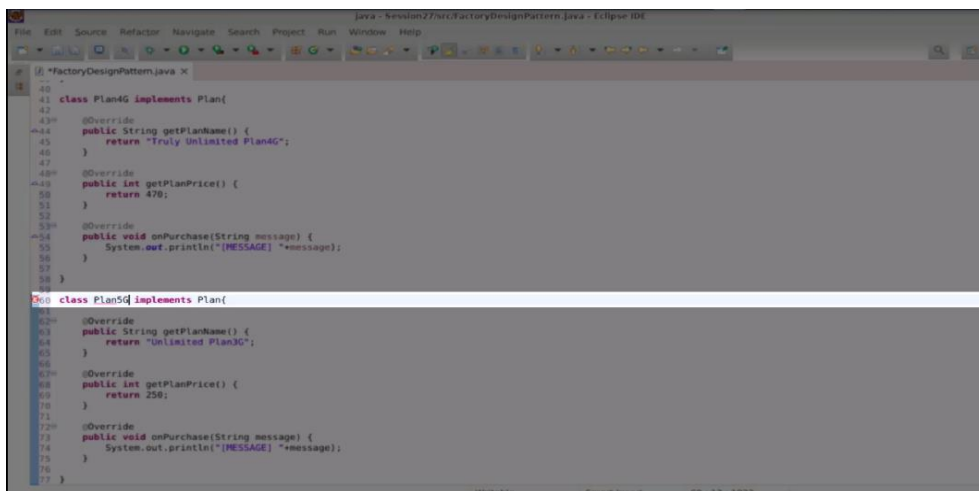
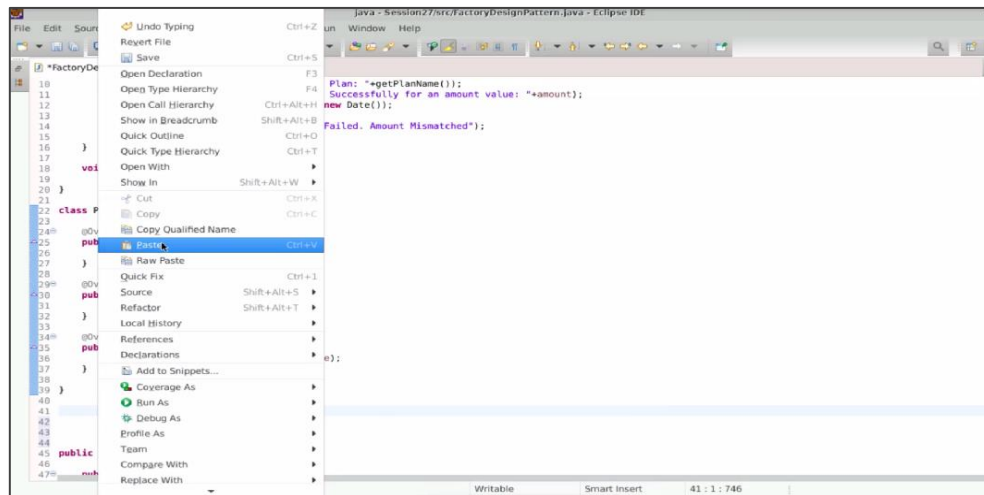
3.4 Create class **Plan5G** by copying and pasting class **Plan3G**.



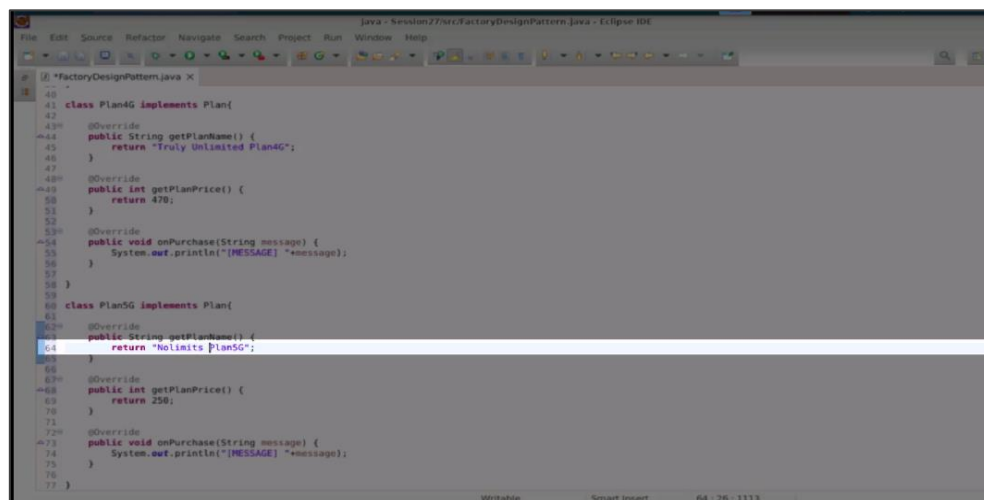
```

10
11
12
13
14
15
16
17
18 void
19
20
21
22 class Plan3G implements Plan {
23
24     @Override
25     public String getPlanName() {
26         return "Unlimited Plan3G";
27     }
28
29     @Override
30     public int getPlanPrice() {
31         return 250;
32     }
33
34     @Override
35     public void onPurchase(String message) {
36         System.out.println("[MESSAGE] " + message);
37     }
38 }
39
40
41
42 public class FactoryDesignPattern {
43
44     public static void main(String[] args) {
45
46
47

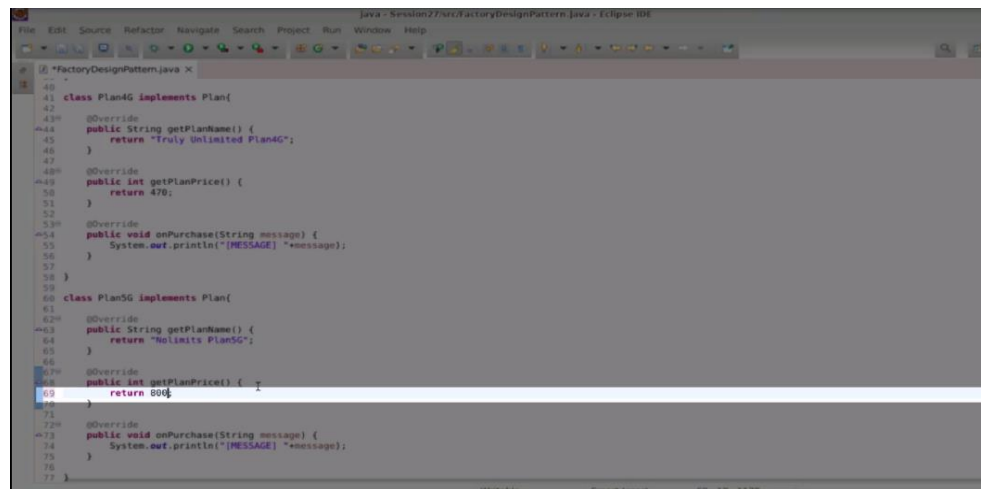
```



3.5 In `getPlanName()` method, write `return Nolimits plan5G`.



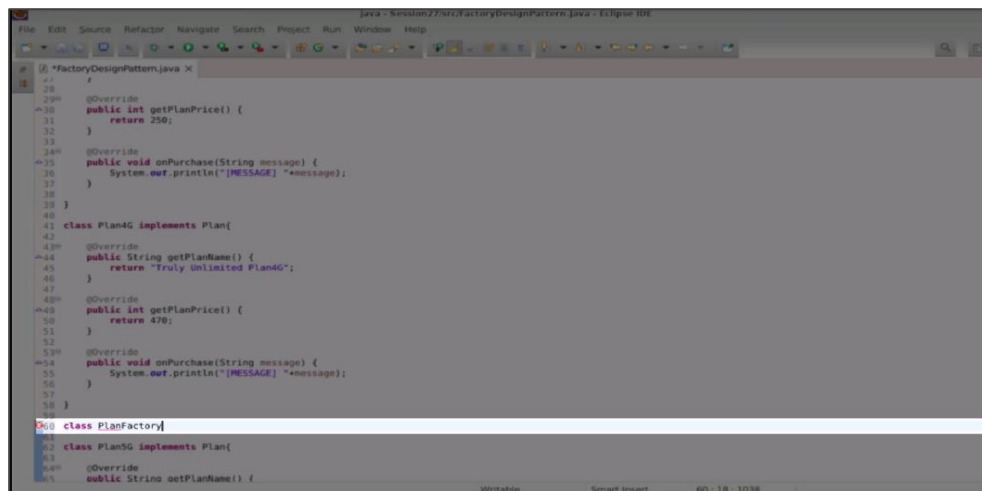
3.6 In the `getPlanPrice()` method, write `return 800`.



```
File Edit Source Refactor Navigate Search Project Run Window Help
FactoryDesignPattern.java x
40
41 class Plan4G implements Plan {
42
43     @Override
44     public String getPlanName() {
45         return "Truly Unlimited Plan4G";
46     }
47
48     @Override
49     public int getPlanPrice() {
50         return 470;
51     }
52
53     @Override
54     public void onPurchase(String message) {
55         System.out.println("[MESSAGE] " + message);
56     }
57 }
58
59 class Plan5G implements Plan {
60
61     @Override
62     public String getPlanName() {
63         return "Holixits Plan5G";
64     }
65
66     @Override
67     public int getPlanPrice() {
68         return 800;
69     }
70
71     @Override
72     public void onPurchase(String message) {
73         System.out.println("[MESSAGE] " + message);
74     }
75 }
76
77 }
```

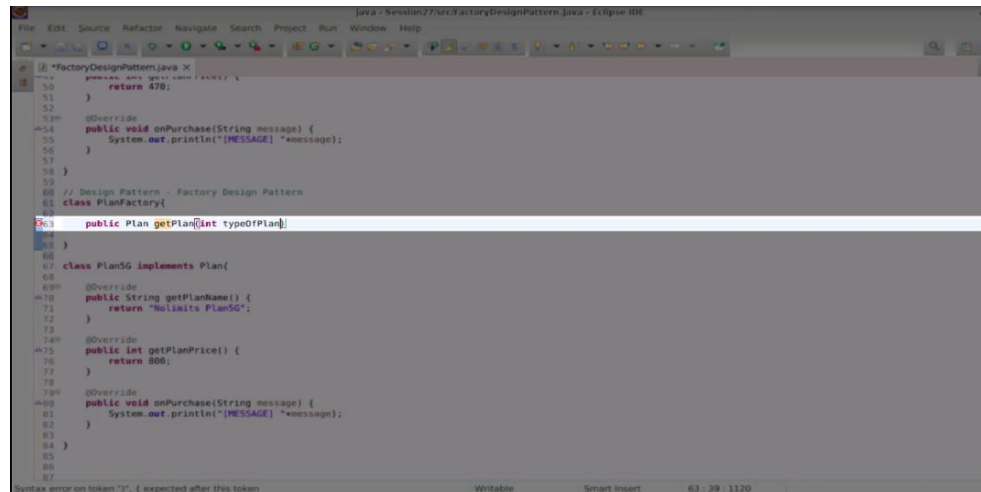
## Step 4: Create class PlanFactory

4.1 Create another class **PlanFactory** by writing the **class PlanFactory** to implement the factory design pattern.



```
File Edit Source Refactor Navigate Search Project Run Window Help
FactoryDesignPattern.java x
28
29
30     @Override
31     public int getPlanPrice() {
32         return 250;
33     }
34
35     @Override
36     public void onPurchase(String message) {
37         System.out.println("[MESSAGE] " + message);
38     }
39 }
40
41 class Plan4G implements Plan {
42
43     @Override
44     public String getPlanName() {
45         return "Truly Unlimited Plan4G";
46     }
47
48     @Override
49     public int getPlanPrice() {
50         return 470;
51     }
52
53     @Override
54     public void onPurchase(String message) {
55         System.out.println("[MESSAGE] " + message);
56     }
57 }
58
59
60 class PlanFactory {
61
62     class Plan5G implements Plan {
63
64         @Override
65         public String getPlanName() {
```

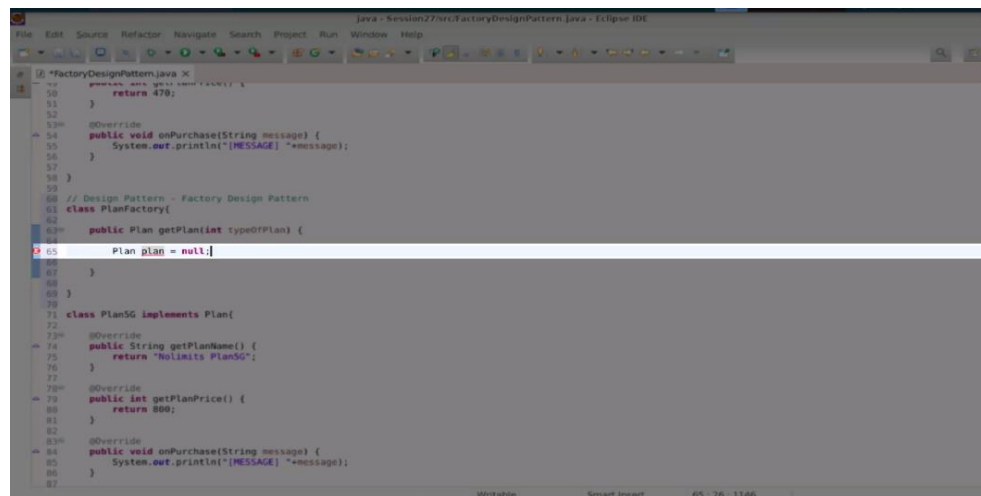
- 4.2 Create a public method which returns a **plan**. Code it as **public Plan getPlan(int typeOfPlan)** which takes typeOfPlan as input.



```
File Edit Source Refactor Navigate Search Project Run Window Help
java - Session27src/FactoryDesignPattern.java - Eclipse IDE

// FactoryDesignPattern.java
50 public void getPlan() {
51     return 470;
52 }
53
54 @Override
55 public void onPurchase(String message) {
56     System.out.println("[MESSAGE] " + message);
57 }
58 }
59
60 // Design Pattern - Factory Design Pattern
61 class PlanFactory{
62     public Plan getPlan(int typeOfPlan){
63     }
64 }
65
66 class Plan5G implements Plan{
67
68     @Override
69     public String getPlanName() {
70         return "NoLimits Plan5G";
71     }
72
73     @Override
74     public int getPlanPrice() {
75         return 800;
76     }
77
78     @Override
79     public void onPurchase(String message) {
80         System.out.println("[MESSAGE] " + message);
81     }
82 }
83
84 }
85
86
87
Syntax error on token "}", { expected after this token  Winable Smart Insert 63 : 39 : 1120
```

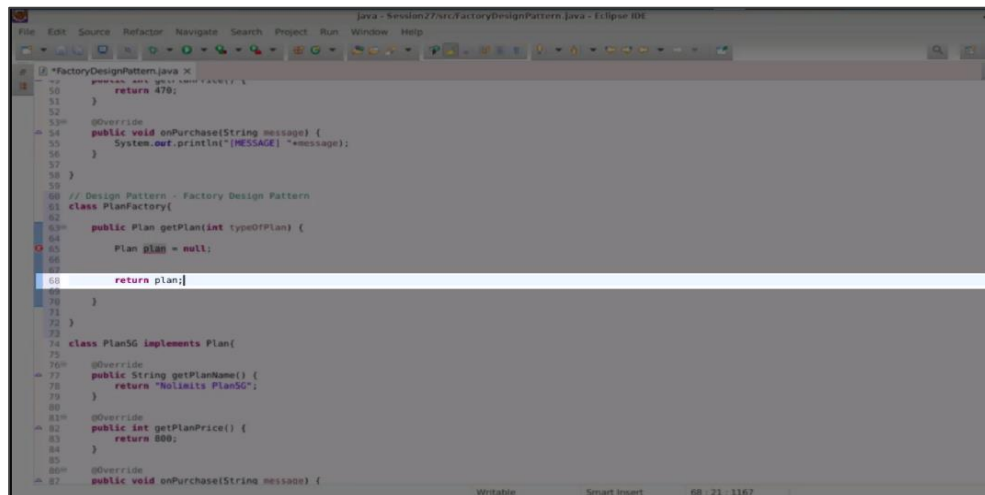
- 4.3 Create a reference variable of plan which is by default **null**.



```
File Edit Source Refactor Navigate Search Project Run Window Help
java - Session27src/FactoryDesignPattern.java - Eclipse IDE

// FactoryDesignPattern.java
50 public void getPlan() {
51     return 470;
52 }
53
54 @Override
55 public void onPurchase(String message) {
56     System.out.println("[MESSAGE] " + message);
57 }
58 }
59
60 // Design Pattern - Factory Design Pattern
61 class PlanFactory{
62     public Plan getPlan(int typeOfPlan) {
63         Plan plan = null;
64     }
65 }
66
67 class Plan5G implements Plan{
68
69     @Override
70     public String getPlanName() {
71         return "NoLimits Plan5G";
72     }
73
74     @Override
75     public int getPlanPrice() {
76         return 800;
77     }
78
79     @Override
80     public void onPurchase(String message) {
81         System.out.println("[MESSAGE] " + message);
82     }
83 }
84
85
86
87
Winable Smart Insert 65 : 28 : 1146
```

#### 4.4 Return the reference variable plan.



4.5 If the type of plan is equivalent to 3, you are attempting to make reference to a 3G plan. And, "4" refers to the 4G plan. The type of plan is equivalent to five then denotes a 5G plan, and in any other situation, you can inform the user that their plan is invalid.

```

// Design Pattern - Factory Design Pattern
class PlanFactory{

    public Plan getPlan(int typeOfPlan) {

        Plan plan = null;

        if(typeOfPlan == 3) {

        }else if(typeOfPlan == 4) {

        }else if(typeOfPlan == 5) {

        }else {
            System.out.println("Invalid Type of Plan");
        }

        return plan;

    }

}

```

- 4.6 According to the selected plan, create the reference objects of respective plans. This is called runtime polymorphism; the same reference variable plan can refer to any plan object. This is the polymorphic behavior with the interfaces.

```
// Design Pattern - Factory Design Pattern
class PlanFactory{

    public Plan getPlan(int typeOfPlan) {

        Plan plan = null;

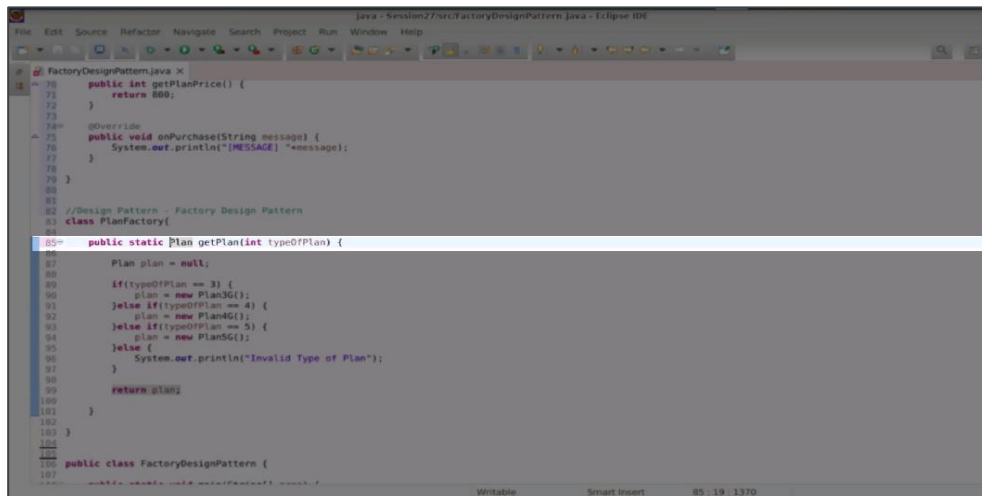
        if(typeOfPlan == 3) {
            plan = new Plan3G();
        } else if(typeOfPlan == 4) {
            plan = new Plan4G();
        } else if(typeOfPlan == 5) {
            plan = new Plan5G();
        } else {
            System.out.println("Invalid Type of Plan");
        }

        return plan;
    }

}
```

## Step 5: Implement the methods of the interface and classes

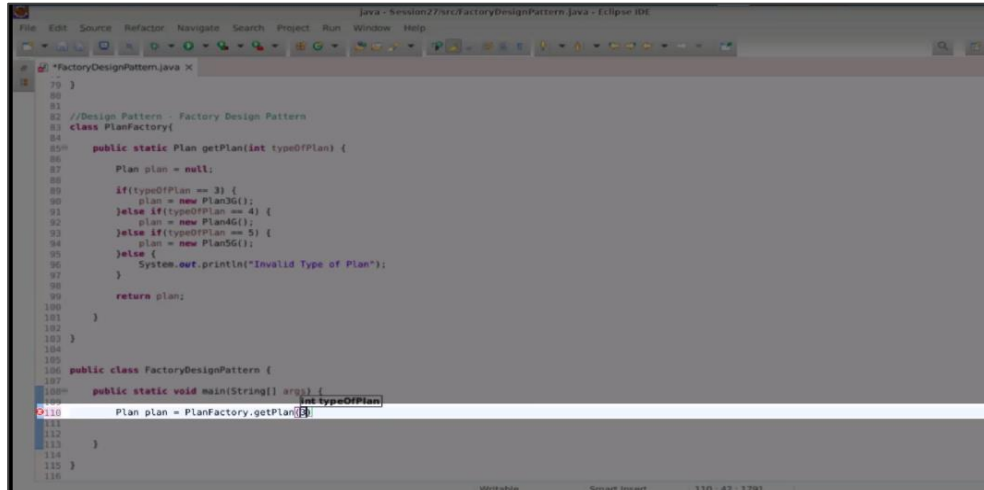
- 5.1 Make the **getPlan** method of class **PlanFactory** as **static** so that we need not create the object to implement this method.



```
FactoryDesignPattern.java
10 public int getPlanPrice() {
11     return 800;
12 }
13
14 @Override
15 public void onPurchase(String message) {
16     System.out.println("MESSAGE: " + message);
17 }
18
19 }
20
21 //Design Pattern - Factory Design Pattern
22 class PlanFactory{
23
24     public static Plan getPlan(int typeOfPlan) {
25
26         Plan plan = null;
27
28         if(typeOfPlan == 3) {
29             plan = new Plan3G();
30         } else if(typeOfPlan == 4) {
31             plan = new Plan4G();
32         } else if(typeOfPlan == 5) {
33             plan = new Plan5G();
34         } else {
35             System.out.println("Invalid Type of Plan");
36         }
37
38         return plan;
39     }
40 }
41
42 public class FactoryDesignPattern {
43
44     public static void main(String[] args) {
45
46         PlanFactory planFactory = new PlanFactory();
47         Plan plan = planFactory.getPlan(3);
48         System.out.println("Plan: " + plan);
49
50     }
51 }
```

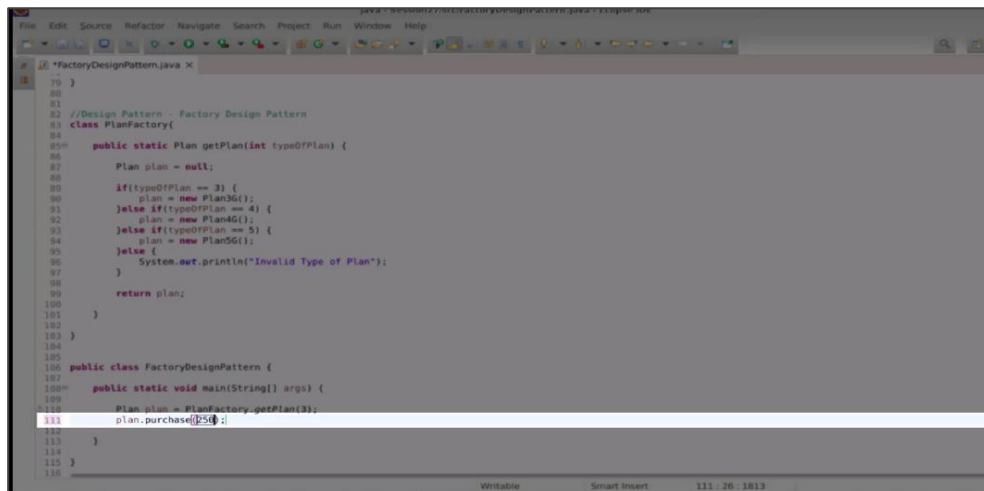


5.2 Now under the main of the class **FactoryDesignPattern**, call the **getPlan** method as **Plan plan = PlanFactory.getPlan()**. Pass the value 3 inside the function call.



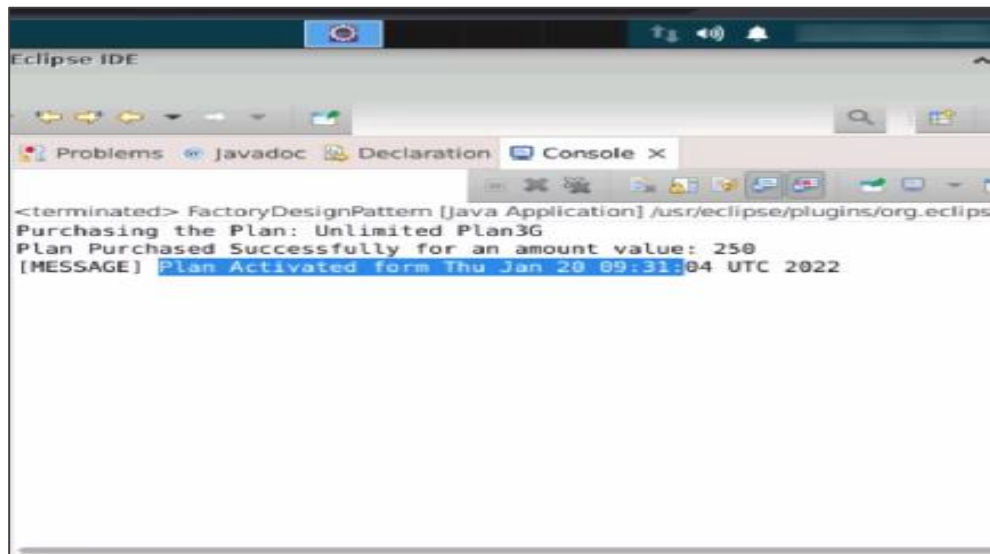
```
70 }
71
72 //Design Pattern - Factory Design Pattern
73 class PlanFactory{
74
75     public static Plan getPlan(int typeOfPlan) {
76
77         Plan plan = null;
78
79         if(typeOfPlan == 3) {
80             plan = new Plan3G();
81         } else if(typeOfPlan == 4) {
82             plan = new Plan4G();
83         } else if(typeOfPlan == 5) {
84             plan = new Plan5G();
85         } else {
86             System.out.println("Invalid Type of Plan");
87         }
88
89         return plan;
90     }
91 }
92
93 public class FactoryDesignPattern {
94
95     public static void main(String[] args) {
96
97         Plan plan = PlanFactory.getPlan(3);
98     }
99 }
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
```

5.3 Call the **purchase()** method as **plan.purchase()** and then enter the amount. This amount should match the type of plan amount.



```
70 }
71
72 //Design Pattern - Factory Design Pattern
73 class PlanFactory{
74
75     public static Plan getPlan(int typeOfPlan) {
76
77         Plan plan = null;
78
79         if(typeOfPlan == 3) {
80             plan = new Plan3G();
81         } else if(typeOfPlan == 4) {
82             plan = new Plan4G();
83         } else if(typeOfPlan == 5) {
84             plan = new Plan5G();
85         } else {
86             System.out.println("Invalid Type of Plan");
87         }
88
89         return plan;
90     }
91 }
92
93 public class FactoryDesignPattern {
94
95     public static void main(String[] args) {
96
97         Plan plan = PlanFactory.getPlan(3);
98         plan.purchase(3);
99     }
100 }
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
```

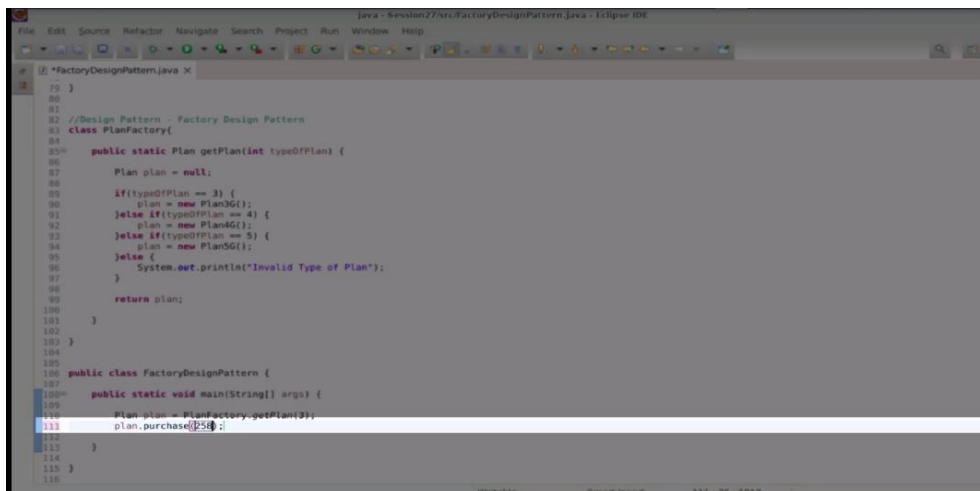
5.4 Run the code and the following output will be obtained.



```

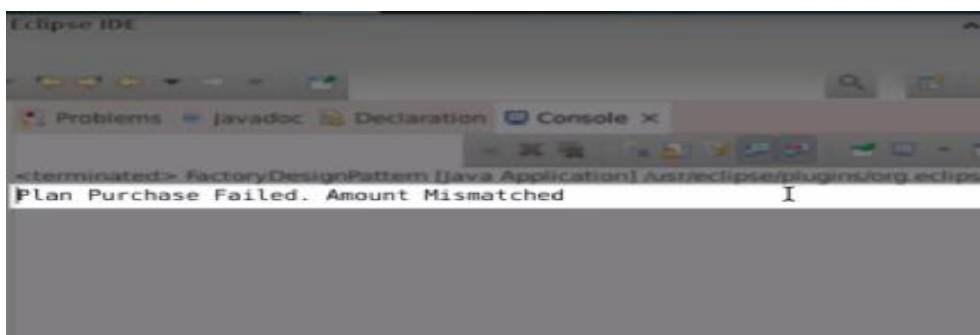
<terminated> FactoryDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse
Purchasing the Plan: Unlimited Plan3G
Plan Purchased Successfully for an amount value: 250
[MESSAGE] Plan Activated form Thu Jan 20 09:31:04 UTC 2022
  
```

5.5 Enter the wrong amount and you will get the message of the amount mismatched.



```

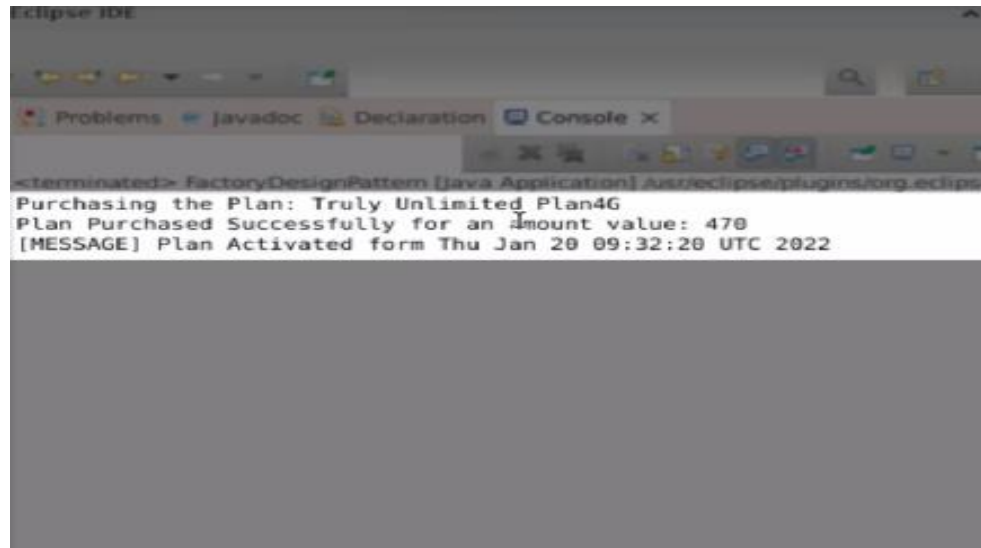
79 }
80
81 //Design Pattern - Factory Design Pattern
82 class PlanFactory{
83     public static Plan getPlan(int typeOfPlan) {
84         Plan plan = null;
85         if(typeOfPlan == 3) {
86             plan = new Plan3G();
87         } else if(typeOfPlan == 4) {
88             plan = new Plan4G();
89         } else if(typeOfPlan == 5) {
90             plan = new Plan5G();
91         } else {
92             System.out.println("Invalid Type of Plan");
93         }
94         return plan;
95     }
96 }
97
98 public class FactoryDesignPattern {
99     public static void main(String[] args) {
100         Plan plan = PlanFactory.getPlan(3);
101         plan.purchase(250);
102     }
103 }
104
105
106
107
108
109
110
111
112
113
114
115
116
  
```



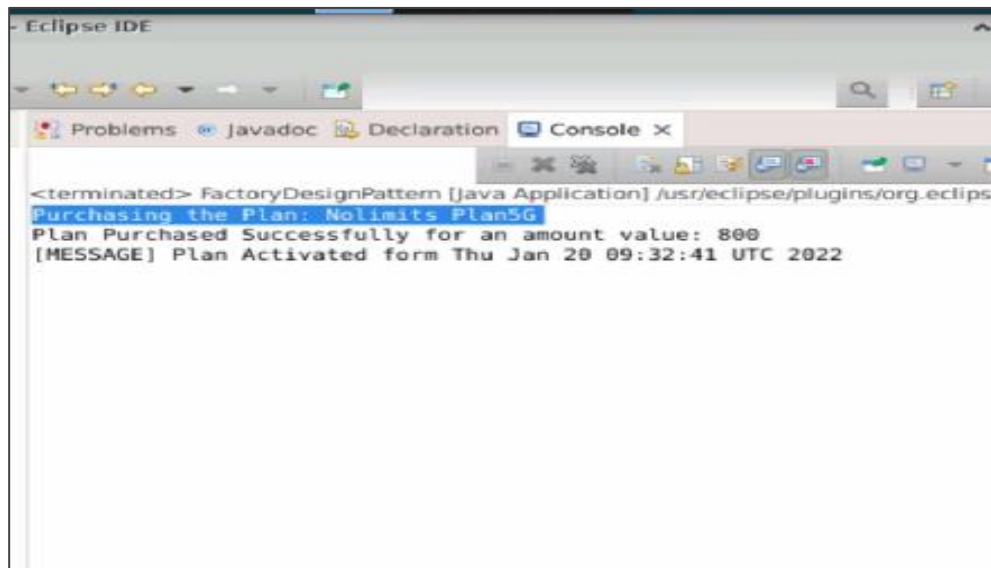
```

<terminated> FactoryDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse
Plan Purchase Failed. Amount Mismatched
  
```

- 5.6 Run the code for 4G and 5G plans also by changing the amount and **typeOfplan** value. The following outputs will be obtained.

A screenshot of the Eclipse IDE's console window. The console shows the output of a Java application. The text in the console is as follows:

```
<terminated> FactoryDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse...  
Purchasing the Plan: Truly Unlimited Plan4G  
Plan Purchased Successfully for an Amount value: 470  
[MESSAGE] Plan Activated form Thu Jan 20 09:32:20 UTC 2022
```

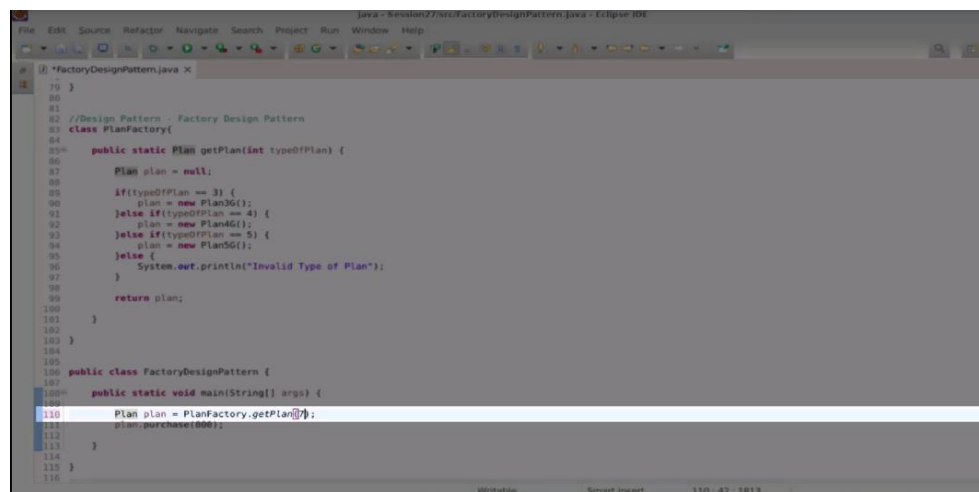


```

<terminated> FactoryDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse
Purchasing the Plan: Nolimits Plan5G
Plan Purchased Successfully for an amount value: 800
[MESSAGE] Plan Activated form Thu Jan 20 09:32:41 UTC 2022

```

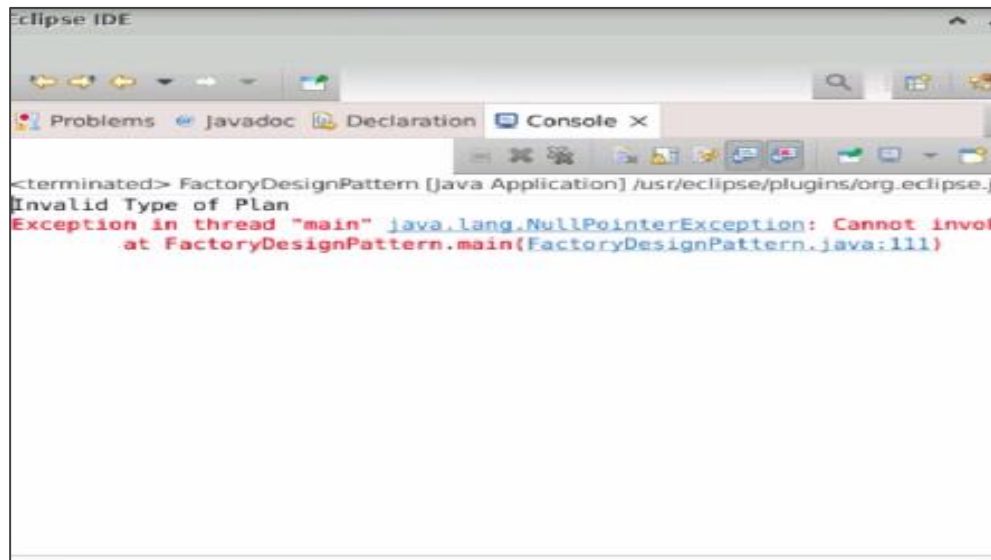
5.7 Pass the value as 7 in the getPlan method and run the code. The code will crash.



```

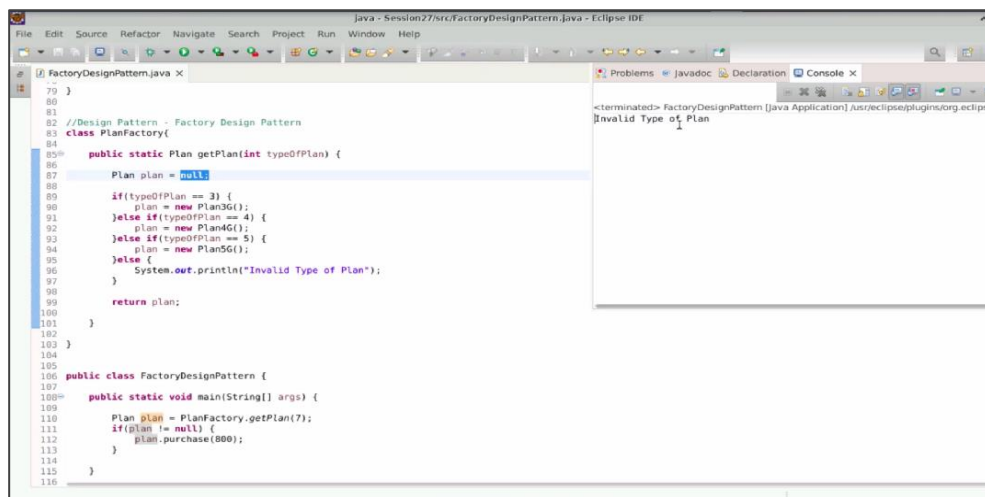
79 }
80
81 //Design Pattern - Factory Design Pattern
82 class PlanFactory{
83     public static Plan getPlan(int typeOfPlan) {
84         Plan plan = null;
85         if(typeOfPlan == 3) {
86             plan = new Plan3G();
87         } else if(typeOfPlan == 4) {
88             plan = new Plan4G();
89         } else if(typeOfPlan == 5) {
90             plan = new Plan5G();
91         } else {
92             System.out.println("Invalid Type of Plan");
93         }
94         return plan;
95     }
96 }
97
98 public class FactoryDesignPattern {
99     public static void main(String[] args) {
100         Plan plan = PlanFactory.getPlan(7);
101         plan.purchase(800);
102     }
103 }
104
105
106
107
108
109
110
111
112
113
114
115
116

```



```
<terminated> FactoryDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse.j
Invalid Type of Plan
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
    at FactoryDesignPattern.main(FactoryDesignPattern.java:111)
```

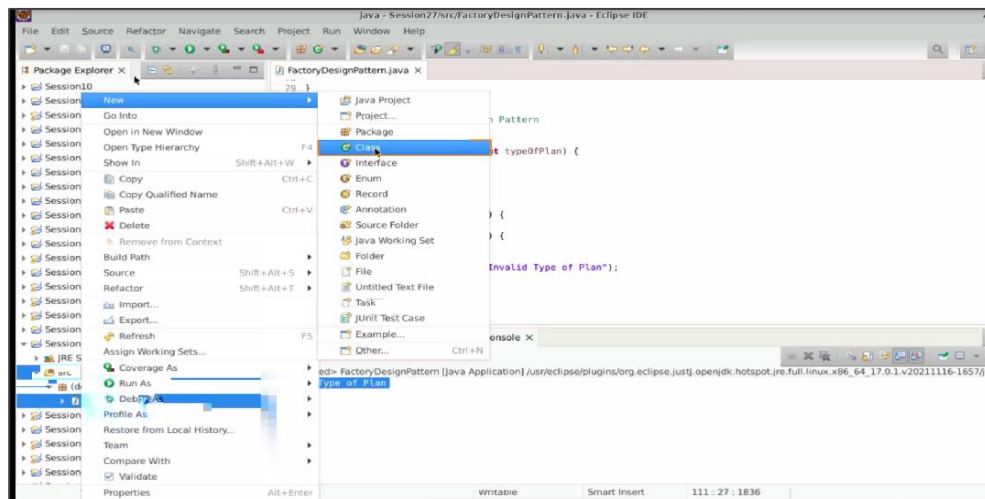
5.8 This is where you need to perform a check. The check involves verifying if your plan is not null. Only if the plan is not null, should you attempt to execute the method called plan.purchase. Initially, the plan was null. With this check in place, you won't encounter a crash and instead, you will receive a message indicating an invalid type of plan.



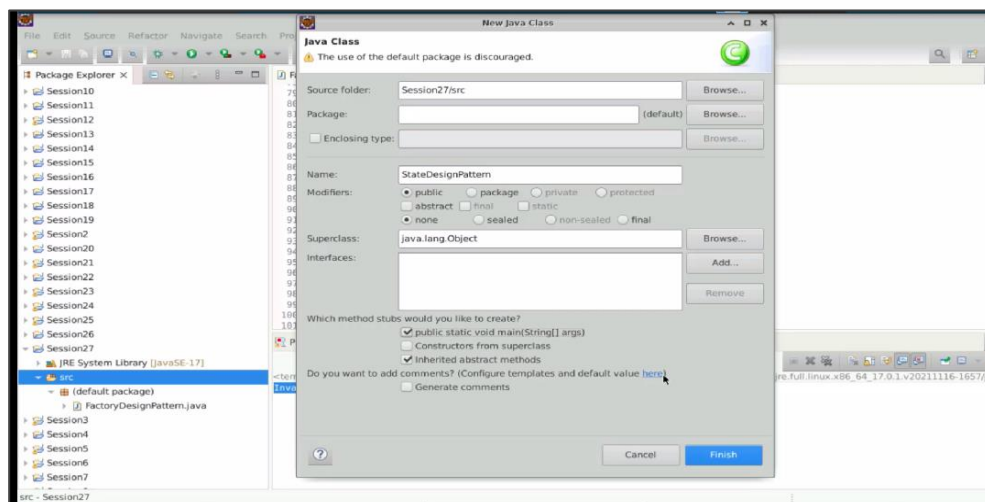
```
70 }
71
72 //Design Pattern - Factory Design Pattern
73 class PlanFactory{
74
75     public static Plan getPlan(int typeOfPlan) {
76
77         Plan plan = null;
78
79         if(typeOfPlan == 3) {
80             plan = new Plan3G();
81         }else if(typeOfPlan == 4) {
82             plan = new Plan4G();
83         }else if(typeOfPlan == 5) {
84             plan = new Plan5G();
85         }else {
86             System.out.println("Invalid Type of Plan");
87         }
88
89         return plan;
90     }
91 }
92
93 public class FactoryDesignPattern {
94
95     public static void main(String[] args) {
96
97         Plan plan = PlanFactory.getPlan(7);
98         if(plan != null) {
99             plan.purchase(800);
100         }
101     }
102 }
103
104
105
106
107
108
109
110
111
112
113
114
115
116
```

## Step 6: Create State Design Patterns

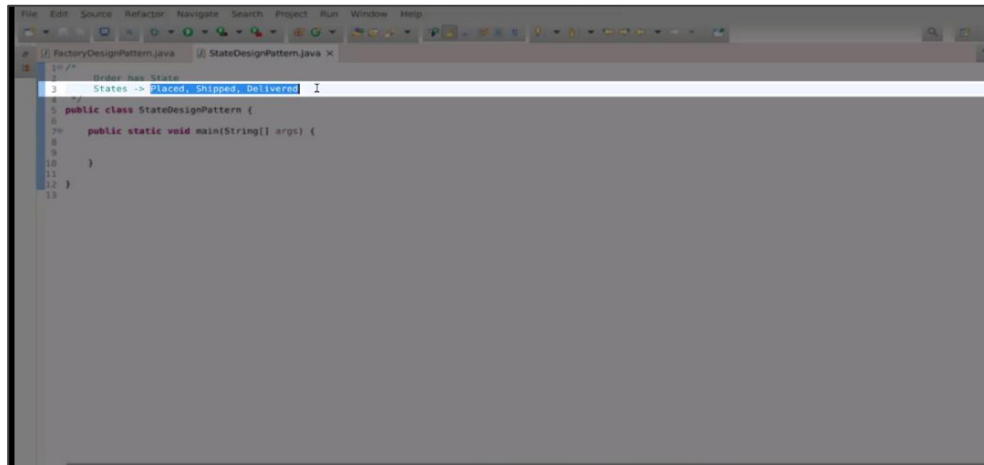
6.1 Right-click on the **src** folder under the project name, select **New**, and then select **Class**.



6.2 Name the class as **StateDesignPattern** and select the option **public static void main(String[] args)**. Select **Finish**. This will create a java class with a **main()** method.

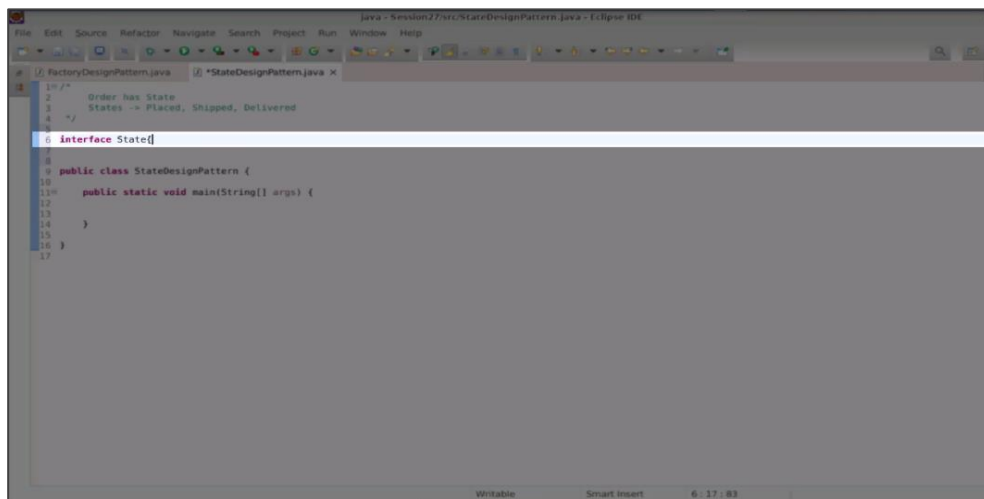


6.3 An order has various states. These states could be **placed**, **shipped**, and **delivered**.



```
1 //  
2 Order has State  
3 States -> Placed, Shipped, Delivered  
4  
5 public class StateDesignPattern {  
6  
7     public static void main(String[] args) {  
8  
9     }  
10  
11 }  
12  
13 }
```

6.4 In the class **StateDesignPattern**, create an interface **State** by writing **interface State**.



```
1 //  
2 Order has State  
3 States -> Placed, Shipped, Delivered  
4  
5  
6 interface State {  
7  
8 }  
9  
10 public class StateDesignPattern {  
11  
12     public static void main(String[] args) {  
13  
14     }  
15  
16 }  
17 }
```

6.5 Create three classes, class **Placed**, class **Shipped** and class **Delivered**. All these classes must implement the interface **State**.

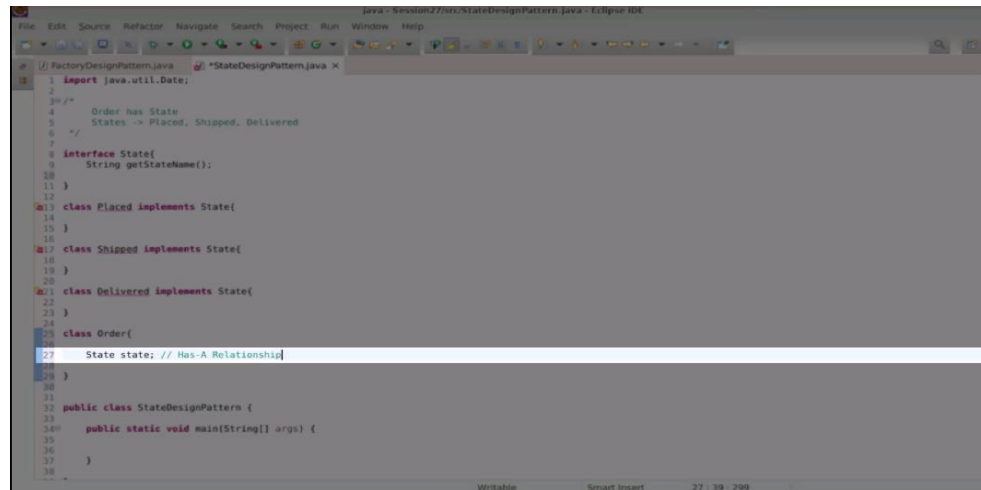
```
1 //
2 Order has State
3 States -> Placed, Shipped, Delivered
4 */
5
6 interface State{
7
8 }
9
10 class Placed implements State{
11
12 }
13
14 class Shipped implements State{
15
16 }
17
18 class Delivered implements State{
19
20 }
21
22 public class StateDesignPattern {
23
24     public static void main(String[] args) {
25
26     }
27
28 }
29
30 }
31
```

6.6 Create a method **String getStateName()** inside the interface **State**.

```
1 //
2 Order has State
3 States -> Placed, Shipped, Delivered
4 */
5
6 interface State{
7     String getStateName();
8 }
9
10 class Placed implements State{
11
12 }
13
14 class Shipped implements State{
15
16 }
17
18 class Delivered implements State{
19
20 }
21
22 public class StateDesignPattern {
23
24     public static void main(String[] args) {
25
26     }
27
28 }
29
30 }
31
32 }
33
```



6.7 Create a new class **Order** which would have a reference to the state. This is called a **has-a-relationship**.

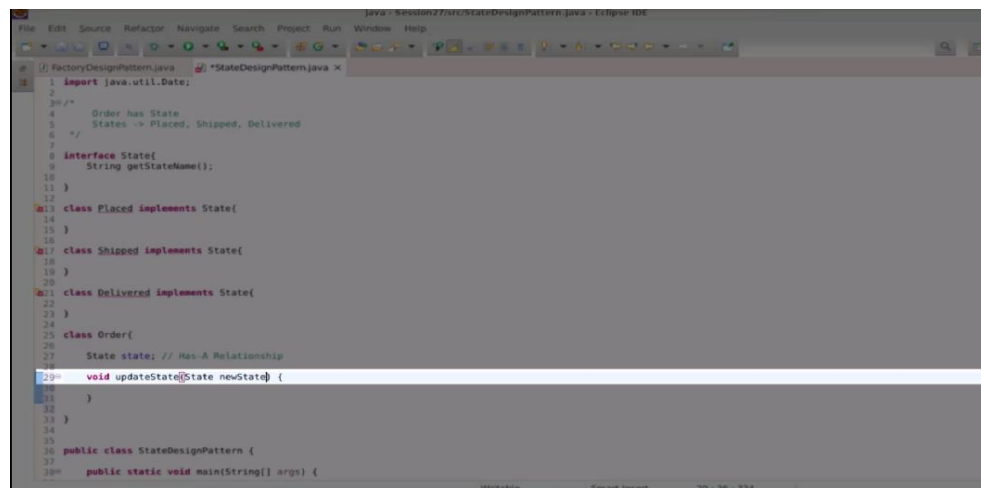


```

1  import java.util.Date;
2
3  /**
4   * Order has State
5   * States -> Placed, Shipped, Delivered
6   */
7
8  interface State{
9      String getStateName();
10 }
11
12
13 class Placed implements State{
14
15 }
16
17 class Shipped implements State{
18
19 }
20
21 class Delivered implements State{
22
23 }
24
25 class Order{
26
27     State state; // Has-A Relationship
28 }
29
30
31
32 public class StateDesignPattern {
33
34     public static void main(String[] args) {
35
36     }
37 }
38

```

6.8 There will be a change in the state. Therefore, create a method **void updateState(State newState)** and pass newState as an input. Make the state as newState.



```

1  import java.util.Date;
2
3  /**
4   * Order has State
5   * States -> Placed, Shipped, Delivered
6   */
7
8  interface State{
9      String getStateName();
10 }
11
12
13 class Placed implements State{
14
15 }
16
17 class Shipped implements State{
18
19 }
20
21 class Delivered implements State{
22
23 }
24
25 class Order{
26
27     State state; // Has-A Relationship
28
29     void updateState(State newState) {
30
31     }
32 }
33
34
35
36 public class StateDesignPattern {
37
38     public static void main(String[] args) {
39
40     }
41 }
42

```

```

1  import java.util.Date;
2
3  /**
4   * Order has State
5   * States -> Placed, Shipped, Delivered
6   */
7
8  interface State{
9      String getStateName();
10 }
11
12
13 class Placed implements State{
14 }
15
16 class Shipped implements State{
17 }
18
19 class Delivered implements State{
20 }
21
22
23 class Order{
24     State state; // Has-A Relationship
25
26     void updateState(State newState) {
27         state = newState;
28     }
29 }
30
31
32 public class StateDesignPattern {
33
34     public static void main(String[] args) {

```

6.9 Create the constructor of class **Order** and set the state as placed so that whenever an object of order is created, the state is set to placed by default.

```

1  /**
2   * States -> Placed, Shipped, Delivered
3   */
4
5  interface State{
6      String getStateName();
7  }
8
9
10 class Placed implements State{
11 }
12
13 class Shipped implements State{
14 }
15
16 class Delivered implements State{
17 }
18
19
20 class Order{
21     State state; // Has-A Relationship
22
23     Order(){
24         state = new Placed();
25     }
26
27     void updateState(State newState) {
28         state = newState;
29     }
30 }
31
32
33 public class StateDesignPattern {
34
35     public static void main(String[] args) {

```

6.10 Complete the unimplemented methods of classes **Placed**, **Shipped** and **Delivered**. Return the strings **PLACED**, **SHIPPED**, and **DELIVERED** in the **getStateName** method of the three classes.

```

1  import java.util.Date;
2
3  /**
4   * Order has State
5   * States -> Placed, Shipped, Delivered
6   */
7
8  interface State{
9      String getStateName();
10 }
11
12 class Placed implements State{
13     // Add unimplemented methods
14 }
15 // Make type 'Placed' abstract
16
17 class
18 // Rename in file (Ctrl+2 R)
19 // Rename in workspace
20
21 class
22
23
24 class
25
26
27 // Press 'Ctrl+Enter' to fix 3 problems of same category in file
28 // Press 'Tab' from proposal table or click for focus
29 // Whenever we create the object of Order, the state is Placed, by default :)
30 Order(){
31     state = new Placed();
32 }
33
34 void updateState(State newState) {
35     state = newState;
36 }
37
38 }

```

The type Placed must implement the inherited abstract method State.getStateName()

```

3  States -> Placed, Shipped, Delivered
4
5
6
7
8  interface State{
9      String getStateName();
10 }
11
12
13 class Placed implements State{
14
15     @Override
16     public String getStateName() {
17         return "PLACED";
18     }
19 }
20
21
22 class Shipped implements State{
23
24     @Override
25     public String getStateName() {
26         // TODO Auto-generated method stub
27         return "SHIPPED";
28     }
29 }
30
31
32 class Delivered implements State{
33
34     @Override
35     public String getStateName() {
36         // TODO Auto-generated method stub
37         return "DELIVERED";
38     }
39 }
40
41
42 class Order{
43
44
45

```

```

27     }
28     }
29
30     class Delivered implements State{
31
32     @Override
33     public String getStateName() {
34         return "DELIVERED";
35     }
36     }
37
38     }
39
40     class Order{
41
42         State state; // Has-A Relationship
43
44         // Whenever we create the object of Order, the state is Placed, by default :)
45         Order(){
46             state = new Placed();
47         }
48
49         State getState() {
50             return state;
51         }
52
53         void updateState(State newState) {
54             state = newState;
55         }
56     }
57
58
59     public class StateDesignPattern {
60
61         public static void main(String[] args) {
62
63             Order order = new Order();

```

6.13 Print the state of the order as `order.getState().getStateName()`.

```
32  
33  
34 public String getStateName() {  
35     return "DELIVERED";  
36 }  
37  
38 }  
39  
40 class Order {  
41  
42     State state; // Has-A Relationship  
43  
44     // whenever we create the object of Order, the state is Placed, by default :)  
45     Order() {  
46         state = new Placed();  
47     }  
48  
49     State getState() {  
50         return state;  
51     }  
52  
53     void updateState(State newState) {  
54         state = newState;  
55     }  
56  
57 }  
58  
59 public class StateDesignPattern {  
60  
61     public static void main(String[] args) {  
62  
63         Order order = new Order();  
64         System.out.println("State of Order as of now is: " + order.getState().getStateName());  
65     }  
66  
67 }  
68  
69
```

6.14 Run the code and implement the basic code of **StateDesignPattern** and obtain the following output.

```
<terminated> StateDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse  
State of Order as of now is: PLACED
```

6.15 Write the statement **Date dateTimeStamp** in the three classes to record the date timestamp associated with these classes.

```

12
13 class Placed implements State{
14
15     Date dateTimeStamp;
16
17     @Override
18     public String getStateName() {
19         return "PLACED";
20     }
21 }
22
23 class Shipped implements State{
24
25     Date dateTimeStamp;
26
27     @Override
28     public String getStateName() {
29         return "SHIPPED";
30     }
31 }
32
33
34 class Delivered implements State{
35
36     Date dateTimeStamp;
37
38

```

6.16 Create two more methods in the state interface. The methods are: **void updateDateTimeStamp()** and **Date getDateTimeStamp()**.

```

File Edit Source Refactor Navigate Search Project Run Window Help
FactoryDesignPattern.java StateDesignPattern.java x
1 import java.util.Date;
2
3 /**
4  * Order has State
5  * States -> Placed, Shipped, Delivered
6  */
7
8 interface State{
9     String getStateName();
10    Date getDateTimeStamp()
11 }
12
13 class Placed implements State{
14
15     Date dateTimeStamp;
16
17     @Override
18     public String getStateName() {
19         return "PLACED";
20     }
21 }
22
23
24 class Shipped implements State{
25
26     Date dateTimeStamp;
27
28     @Override
29     public String getStateName() {
30         return "SHIPPED";
31     }
32 }
33
34
35 class Delivered implements State{
36
37     Date dateTimeStamp;
38

```

```

1  import java.util.Date;
2
3  /**
4   * Order has State
5   * States -> Placed, Shipped, Delivered
6   */
7
8  interface State {
9      String getStateName();
10     void updateDateTimeStamp(Date date);
11     Date getDateTimeStamp();
12 }
13
14 class Placed implements State {
15     Date dateTimeStamp;
16
17     @Override
18     public String getStateName() {
19         return "PLACED";
20     }
21
22 }
23
24 class Shipped implements State {
25     Date dateTimeStamp;
26
27     @Override
28     public String getStateName() {
29         return "SHIPPED";
30     }
31
32 }
33
34 class Delivered implements State {
35     Date dateTimeStamp;
36
37 }
38

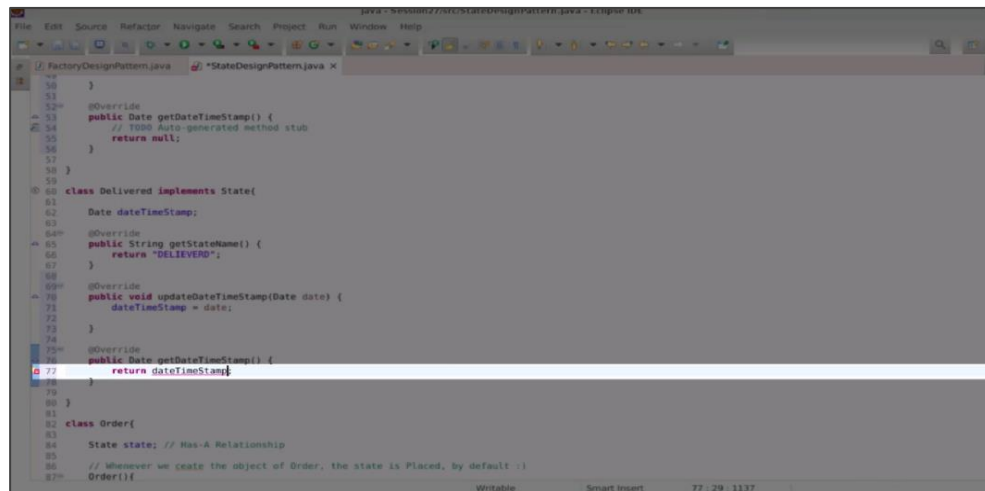
```

6.17 Implement these two methods inside the three classes. In the updateDateTimeStamp method, copy the date to dateTimeStamp. And in the getDateTimeStamp method, simply return the dateTimeStamp attribute.

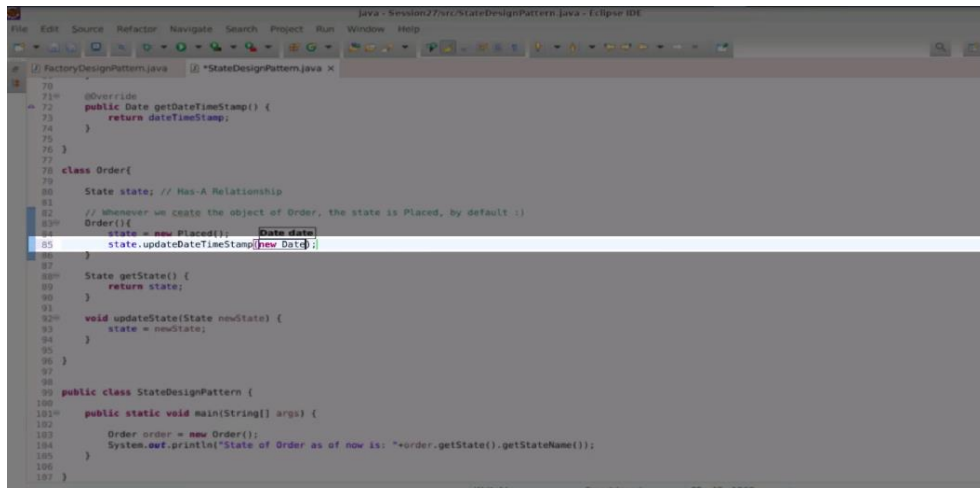
```

39
40 }
41
42 @Override
43 public Date getDateTimeStamp() {
44     // TODO Auto-generated method stub
45     return null;
46 }
47
48 }
49
50 class Delivered implements State {
51     Date dateTimeStamp;
52
53     @Override
54     public String getStateName() {
55         return "DELIVERED";
56     }
57
58     @Override
59     public void updateDateTimeStamp(Date date) {
60         dateTimeStamp = date;
61     }
62
63     @Override
64     public Date getDateTimeStamp() {
65         // TODO Auto-generated method stub
66         return null;
67     }
68 }
69
70 class Order {
71     State state; // Has-A Relationship
72
73     // Whenever we create the object of Order, the state is Placed, by default.
74 }
75

```

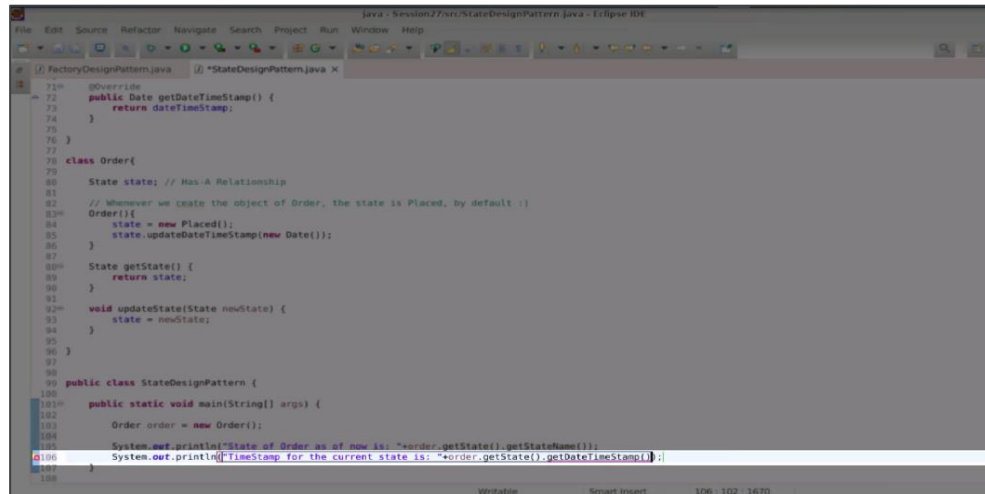


6.18 Create an object of date to display the current date when you place an order by writing **state.updateDateTimeStamp(new Date)** in the constructor **Order**.





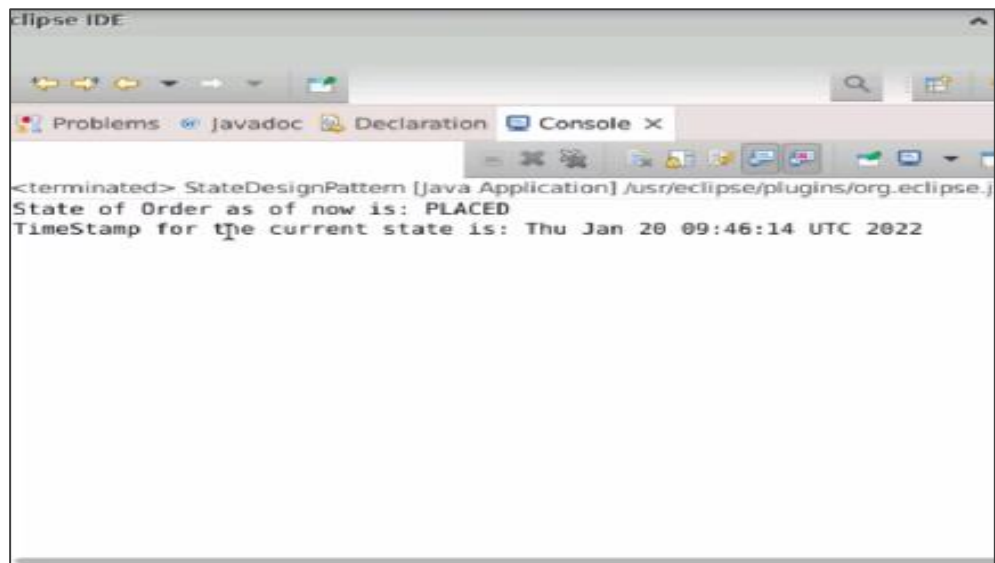
6.19 Check the state of your order and date timestamp by calling `order.getState().getDateTimeStamp()`.



```
File Edit Source Refactor Navigate Search Project Run Window Help
jira - bin\src\main\java\StateDesignPattern.java - Eclipse IDE

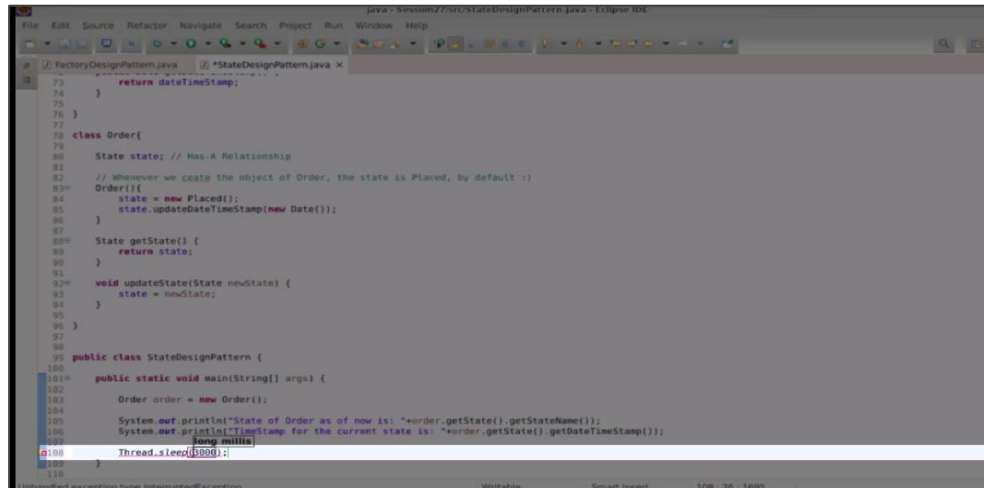
j FactoryDesignPattern.java j *StateDesignPattern.java x
71 @Override
72 public Date getDateTimeStamp() {
73     return dateTimestamp;
74 }
75 }
76 }
77 }
78 class Order {
79     State state; // Has-A Relationship
80     // Whenever we create the object of Order, the state is Placed, by default :)
81     Order() {
82         state = new Placed();
83         state.updateDateTimeStamp(new Date());
84     }
85     State getState() {
86         return state;
87     }
88     void updateState(State newState) {
89         state = newState;
90     }
91 }
92 public class StateDesignPattern {
93     public static void main(String[] args) {
94         Order order = new Order();
95         System.out.println("State of Order as of now is: " + order.getState().getStateName());
96         System.out.println("Time Stamp for the current state is: " + order.getState().getDateTimeStamp());
97     }
98 }
```

6.20 Run the code and the following output will be obtained.



```
clipse IDE
Problems Javadoc Declaration Console x
<terminated> StateDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse.j
State of Order as of now is: PLACED
Time Stamp for the current state is: Thu Jan 20 09:46:14 UTC 2022
```

6.21 Introduce a delay of 3 seconds by writing `Thread.sleep(3000)`. Surround this by try catch block by clicking on the error bulb.



```

173 return dateTimeStamp;
174 }
175 }
176 }
177
178 class Order {
179     State state; // Has-A Relationship
180
181     // Whenever we create the object of Order, the state is Placed, by default :)
182     Order() {
183         state = new Placed();
184         state.updateDateTimeStamp(new Date());
185     }
186
187     State getState() {
188         return state;
189     }
190
191     void updateState(State newState) {
192         state = newState;
193     }
194 }
195
196 public class StateDesignPattern {
197
198     public static void main(String[] args) {
199
200         Order order = new Order();
201
202         System.out.println("State of Order as of now is: "+order.getState().getStateName());
203         System.out.println("TimeStamp for the current state is: "+order.getState().getDateTimeStamp());
204         long millis =
205         Thread.sleep(3000);
206     }
207 }
  
```

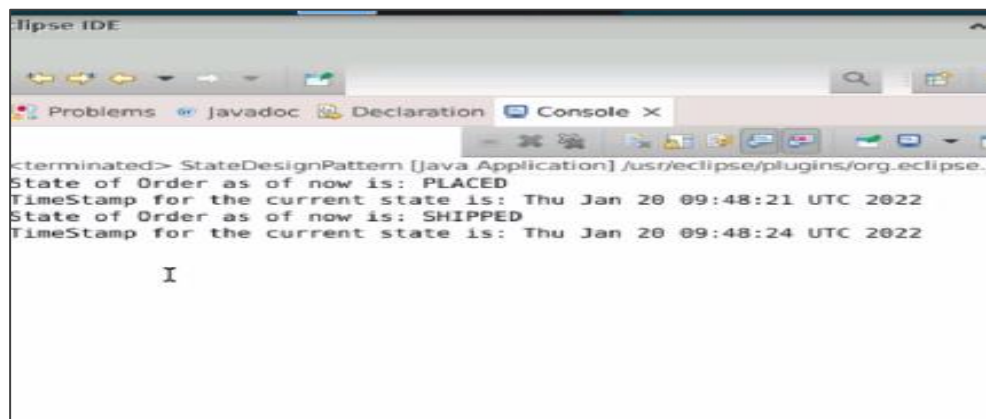
6.22 Create a new state called shipped state and update the date timestamp and give once again a new object of date. Then update the state to this new state by writing `order.updateState(state)`. Copy paste the two print statements above.



```

114
115     State state;
116     state = new Shipped();
117     state.updateDateTimeStamp(new Date());
118
119     order.updateState(state);
120
121     System.out.println("State of Order as of now is: "+order.getState().getStateName());
122     System.out.println("TimeStamp for the current state is: "+order.getState().getDateTimeStamp());
123
124 }
125
  
```

### 6.23 Run the code.



```

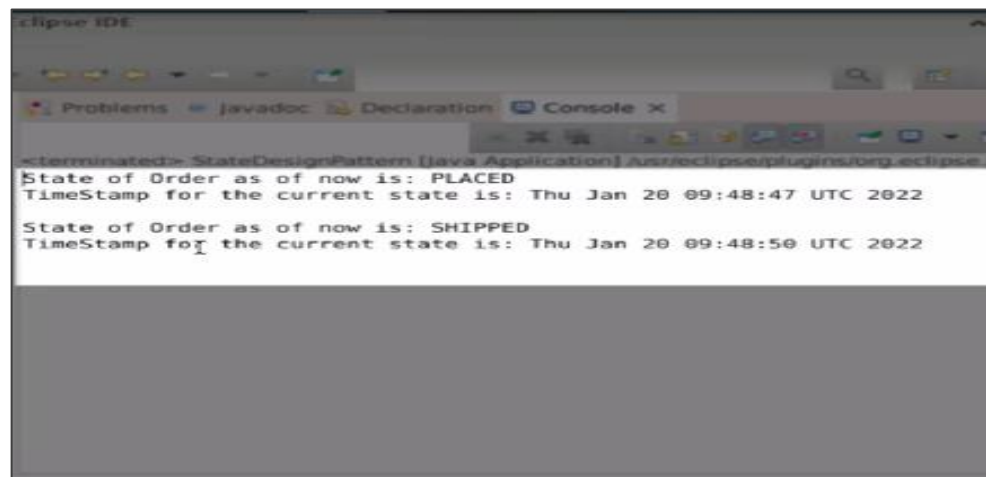
Eclipse IDE

Problems Javadoc Declaration Console X

<terminated> StateDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse.j
State of Order as of now is: PLACED
TimeStamp for the current state is: Thu Jan 20 09:48:21 UTC 2022
State of Order as of now is: SHIPPED
TimeStamp for the current state is: Thu Jan 20 09:48:24 UTC 2022

I

```



```

Eclipse IDE

Problems Javadoc Declaration Console X

<terminated> StateDesignPattern [Java Application] /usr/eclipse/plugins/org.eclipse.j
State of Order as of now is: PLACED
TimeStamp for the current state is: Thu Jan 20 09:48:47 UTC 2022
State of Order as of now is: SHIPPED
TimeStamp for the current state is: Thu Jan 20 09:48:50 UTC 2022

```

By following the above steps, you have successfully implemented the Factory and State design patterns. These patterns enhance flexibility, reusability, and scalability in your software, creating a robust and efficient solution.