

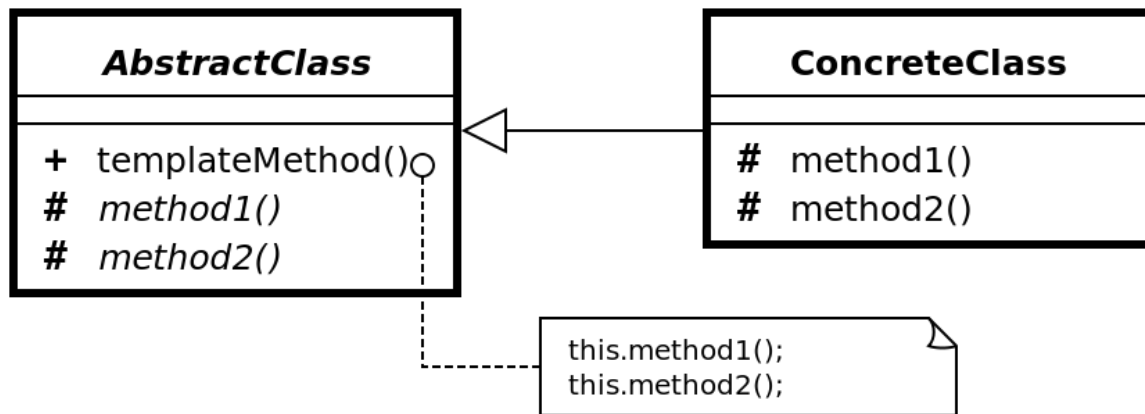
# The Template Method Pattern

## SSD Patterns & Pattern Languages

### Lesson 10.1

#### Introduction

The name is the “Template Method” Pattern, meaning that we create a template for a method. We do this by encapsulating aspects of an algorithm. An interface will define the steps needed to perform a method, but the implementation of those steps may vary between subclasses



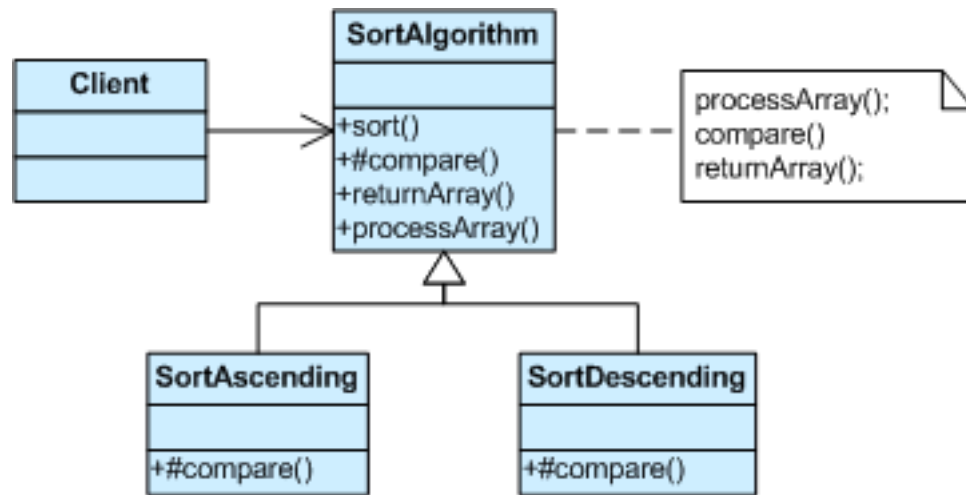
#### GENERIC UML FOR TEMPLATE METHOD PATTERN [1]

#### When to Use

The Template Method pattern should be used

- “To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary” [2]
- “When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is good example of "refactoring to generalize" as described by Opdyke and Johnson. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations” [2]
- “To control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points” [2]

Here is an example of swapping the comparison step of a sorting algorithm based on the functionality needed. The Template of sorting remains the same with only the compare step being called differently.



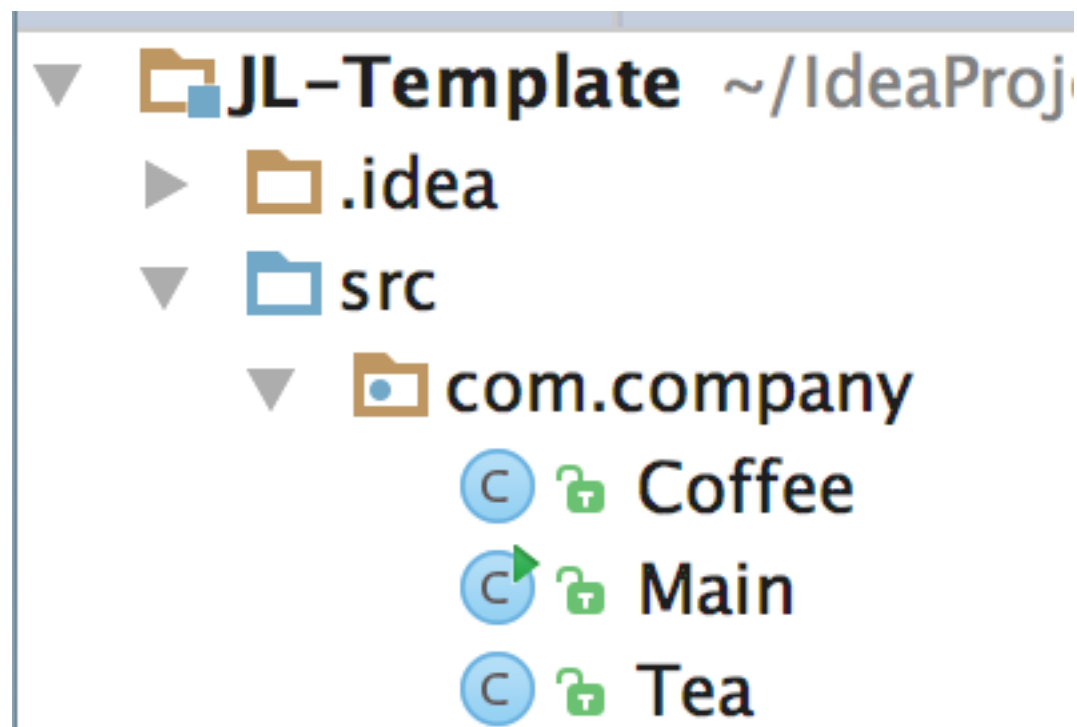
CONCRETE UML OF SORTALGORITHM TEMPLATE METHOD PATTERN [3]

### Project Description

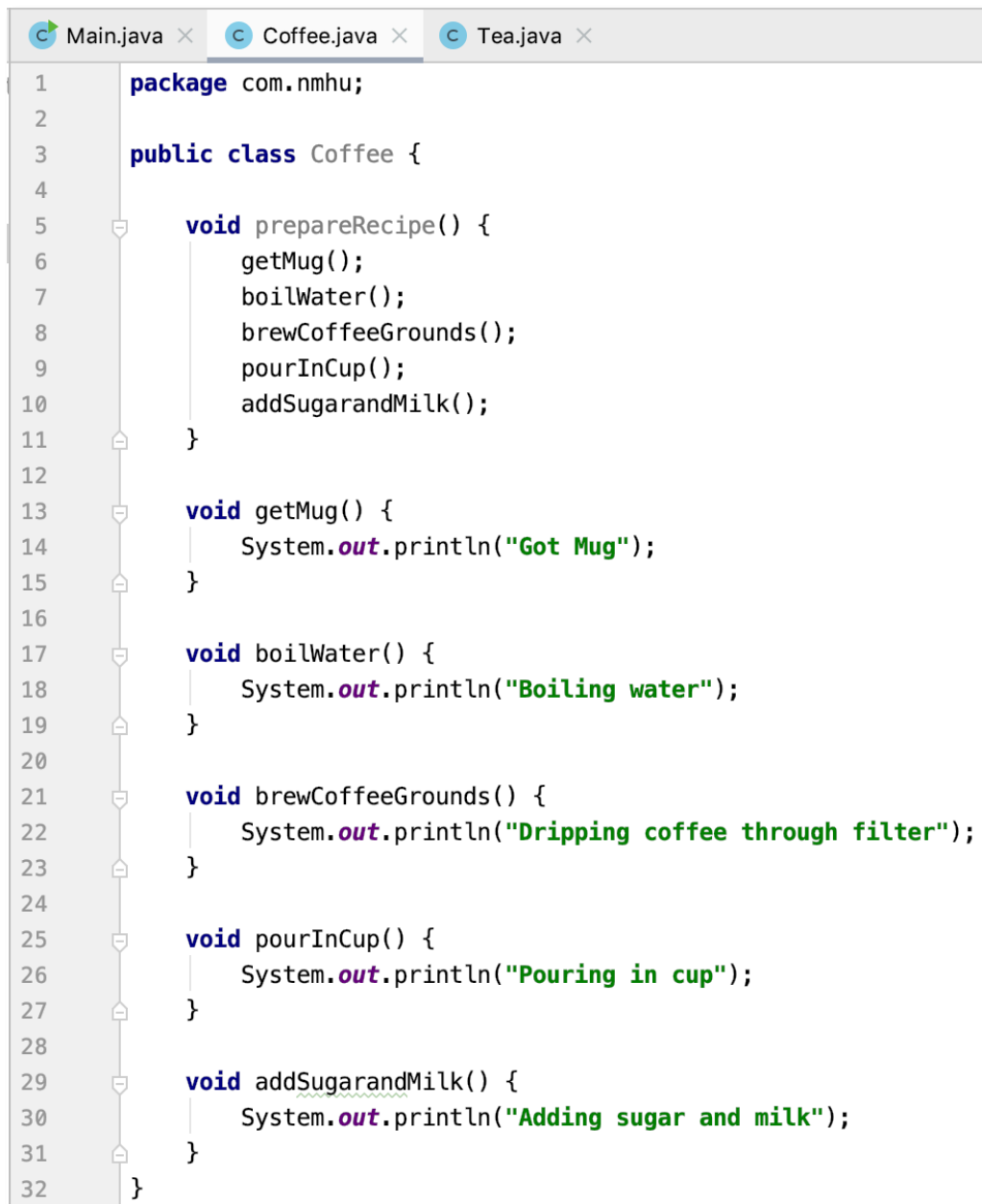
We will create a project based on the chapter in the Head First Design Patterns Book. It will model a drink creator robot where the drinks all share similar steps, but the steps are completed in different ways. We will then modify this project to use the Template Method pattern. After that we will add additional code to see this pattern's concept of *hooks* where certain drink implementations can choose to hook into or not hook into a particular step in the template.

### Step 1 - Creating the Simple Example

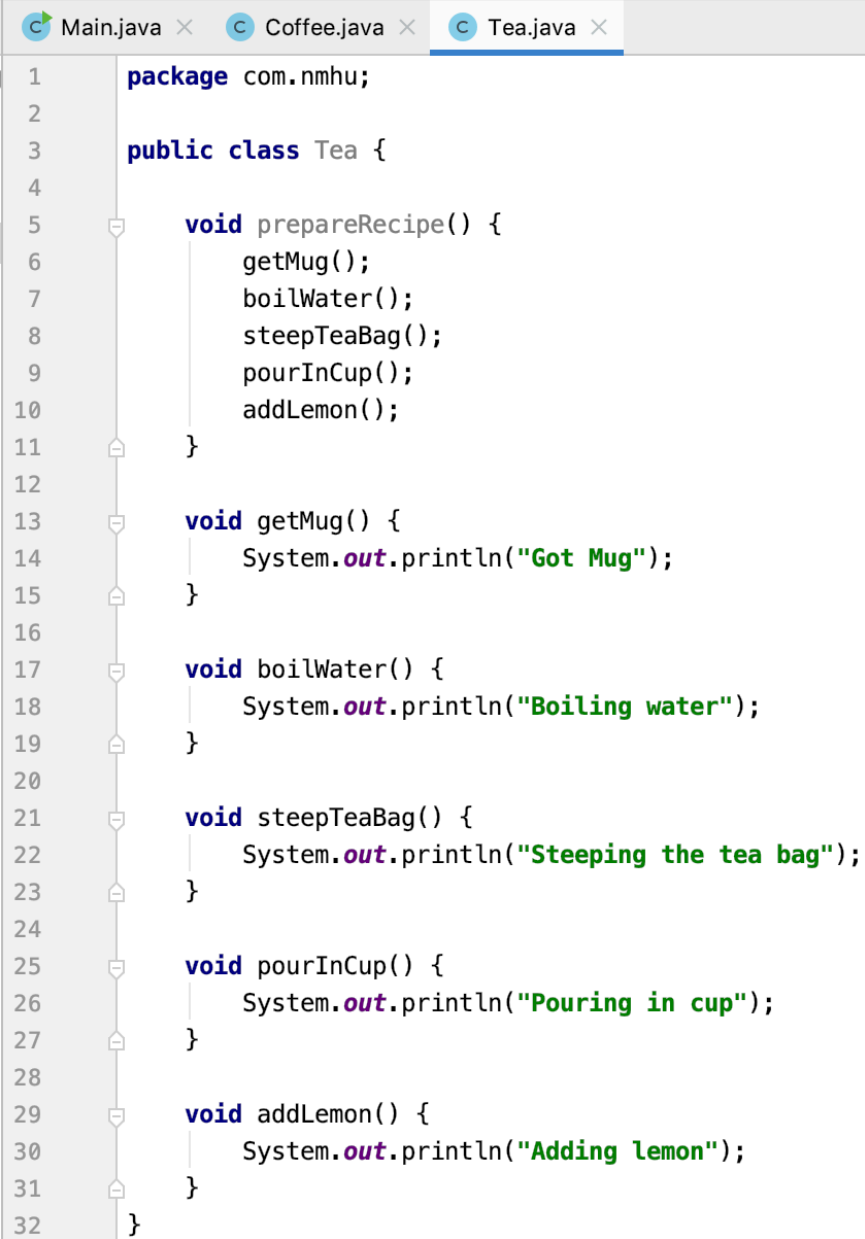
Create a new Java project with your initials called Template. Create the following files in your project:



Create the following code:

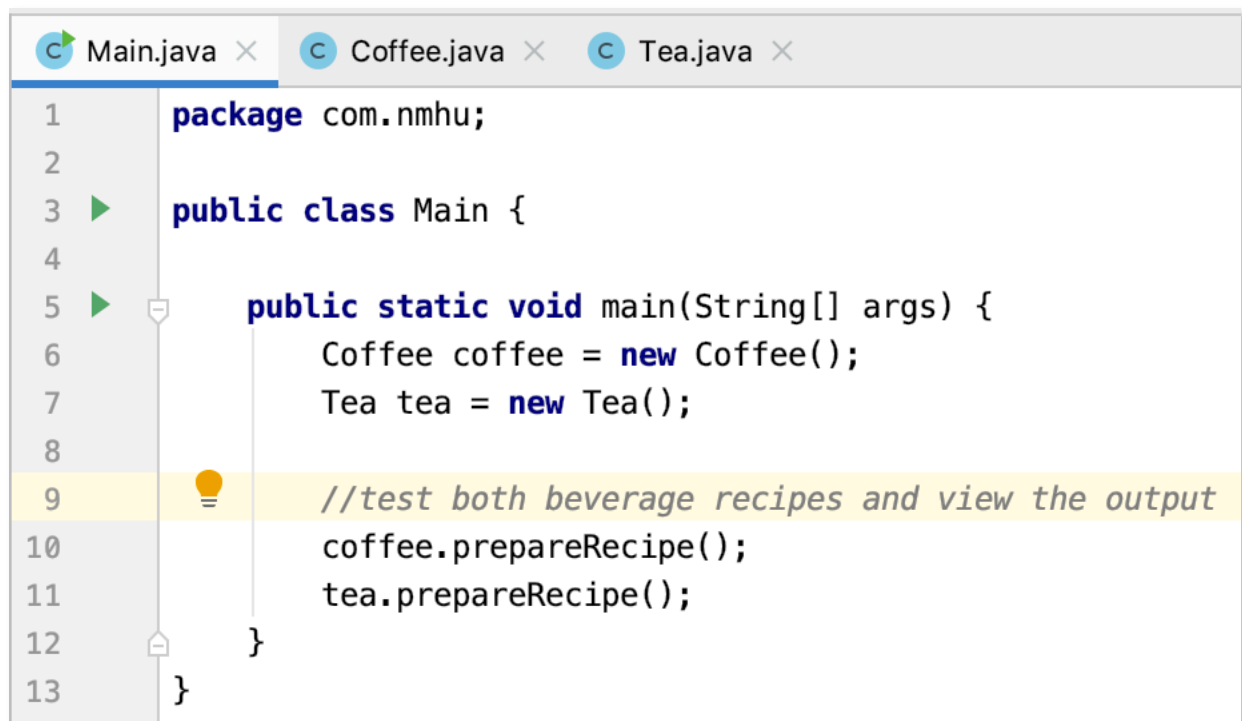


```
1 package com.nmhu;
2
3 public class Coffee {
4
5     void prepareRecipe() {
6         getMug();
7         boilWater();
8         brewCoffeeGrounds();
9         pourInCup();
10        addSugarandMilk();
11    }
12
13    void getMug() {
14        System.out.println("Got Mug");
15    }
16
17    void boilWater() {
18        System.out.println("Boiling water");
19    }
20
21    void brewCoffeeGrounds() {
22        System.out.println("Dripping coffee through filter");
23    }
24
25    void pourInCup() {
26        System.out.println("Pouring in cup");
27    }
28
29    void addSugarandMilk() {
30        System.out.println("Adding sugar and milk");
31    }
32 }
```



```
1 package com.nmhu;
2
3 public class Tea {
4
5     void prepareRecipe() {
6         getMug();
7         boilWater();
8         steepTeaBag();
9         pourInCup();
10        addLemon();
11    }
12
13    void getMug() {
14        System.out.println("Got Mug");
15    }
16
17    void boilWater() {
18        System.out.println("Boiling water");
19    }
20
21    void steepTeaBag() {
22        System.out.println("Steeping the tea bag");
23    }
24
25    void pourInCup() {
26        System.out.println("Pouring in cup");
27    }
28
29    void addLemon() {
30        System.out.println("Adding lemon");
31    }
32 }
```

Main and output of program.



```
1 package com.nmhu;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Coffee coffee = new Coffee();
7         Tea tea = new Tea();
8
9         //test both beverage recipes and view the output
10        coffee.prepareRecipe();
11        tea.prepareRecipe();
12    }
13 }
```

```
/Library/Java/JavaVirtualMachines/jd
Got Mug
Boiling water
Dripping coffee through filter
Pouring in cup
Adding sugar and milk
Got Mug
Boiling water
Steeping the tea bag
Pouring in cup
Adding lemon

Process finished with exit code 0
```

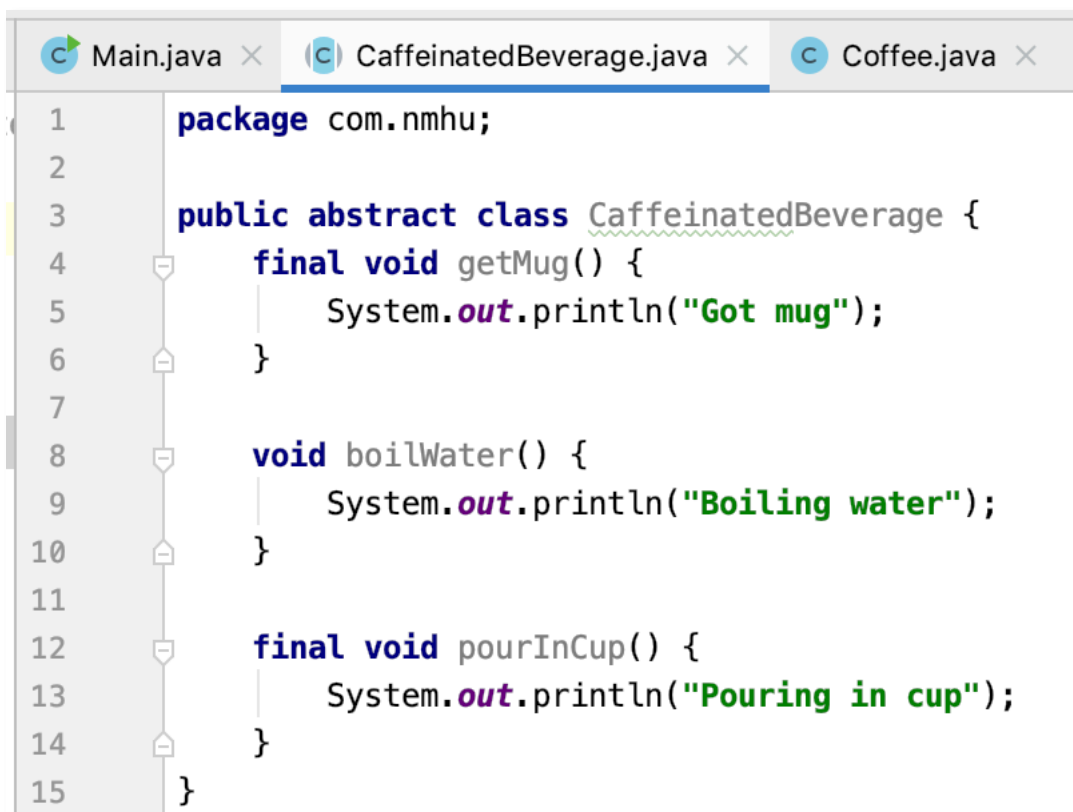
Note the shared code and with some renaming, everything in the prepareRecipe function could be shared code, conforming to an interface.

## Step 2 - Adding the Pattern Pt. 1

Make a new CaffeinatedBeverage abstract class to get rid of the duplicate code in Coffee and Tea.

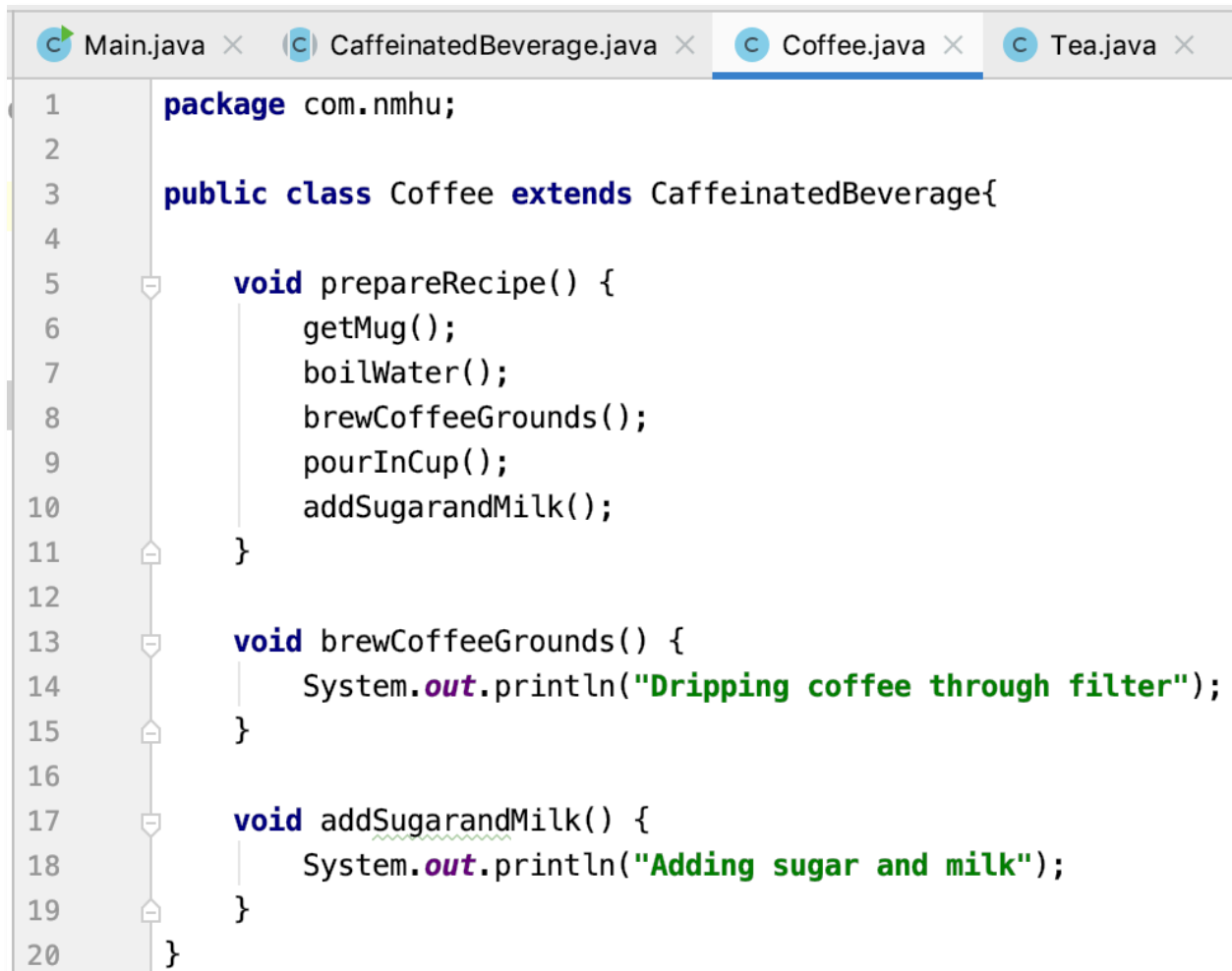
I made lines 4 and 12 final because we are requiring all beverages to be put into a cup. No class can ever change this methodology.

Read more about using final here if you need: <https://docs.oracle.com/javase/tutorial/java/land/final.html>



```
1 package com.nmhu;
2
3 public abstract class CaffeinatedBeverage {
4     final void getMug() {
5         System.out.println("Got mug");
6     }
7
8     void boilWater() {
9         System.out.println("Boiling water");
10    }
11
12    final void pourInCup() {
13        System.out.println("Pouring in cup");
14    }
15 }
```

Make Coffee extend this new class and delete what is implemented in the abstract.



```
1 package com.nmhu;
2
3 public class Coffee extends CaffeinatedBeverage{
4
5     void prepareRecipe() {
6         getMug();
7         boilWater();
8         brewCoffeeGrounds();
9         pourInCup();
10        addSugarandMilk();
11    }
12
13    void brewCoffeeGrounds() {
14        System.out.println("Dripping coffee through filter");
15    }
16
17    void addSugarandMilk() {
18        System.out.println("Adding sugar and milk");
19    }
20 }
```

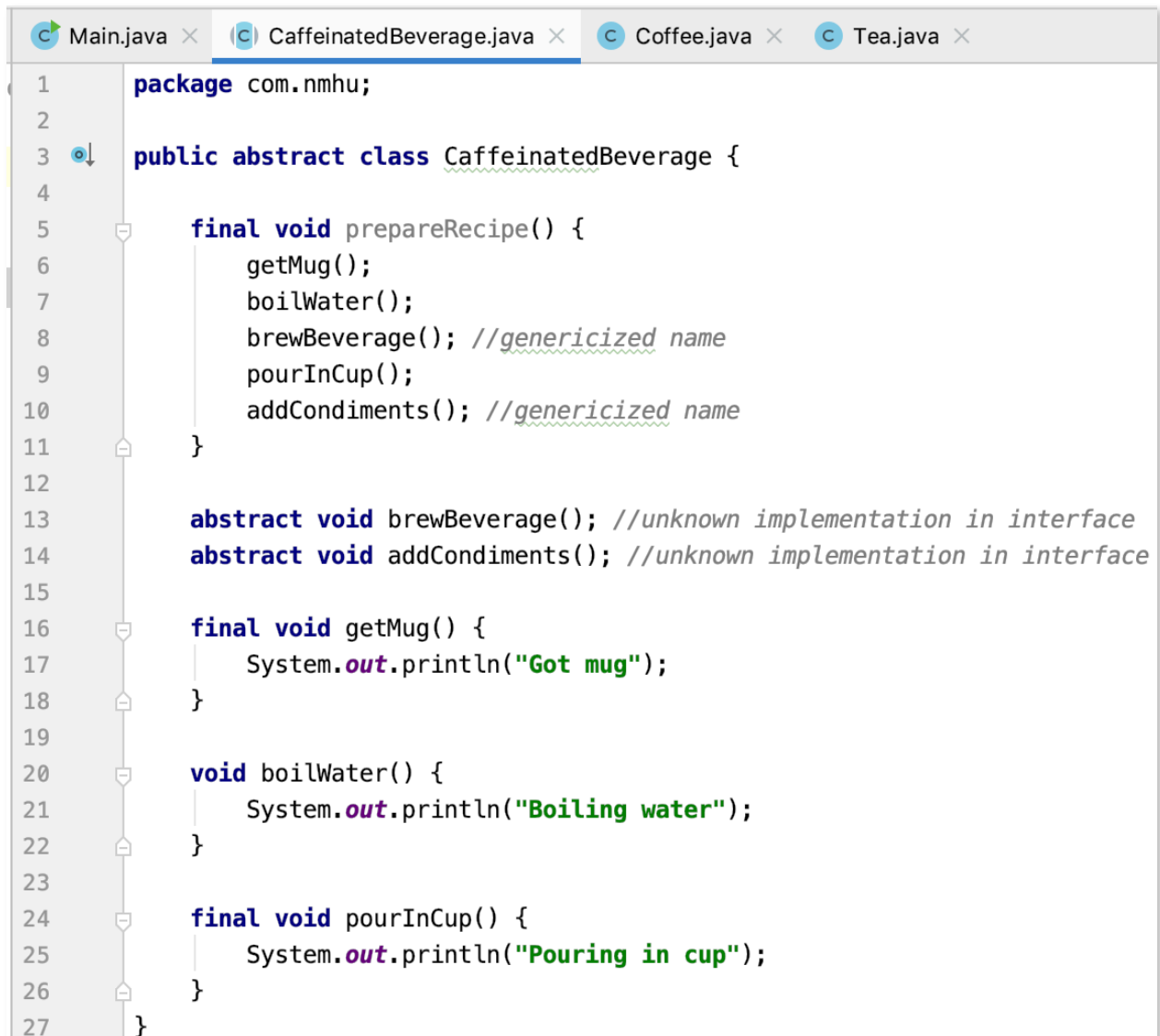
After doing this, make Tea extend the abstract class in the same way on your own. You should be able to test your program and get the same output from Main as we did in Step 1.

We have not used the Template Method pattern yet, however. We want prepareRecipe to be our Template Method and for both Tea and Coffee to inherit that method from the abstract class. Their implementations of it may be different, but we want to be certain they both respond to prepareRecipe as CaffeinatedBeverages.

Looking at the code, we can see that this is easily doable, but involves some renaming. This is reminiscent of what we had to do in the Command Pattern. We had to consider Volume and Speed to be the same thing so that both types of Electronic Device could conform to one interface. Since these two differing steps are essentially the same in the broad recipe algorithm, some renaming makes them good candidates for the Template Method Pattern. If they had been quite different recipes (like Coffee and Sandwich or Tea and Soup) then this pattern would not have worked.

### Step 3 - Adding the Pattern Pt. 2 (For real this time)

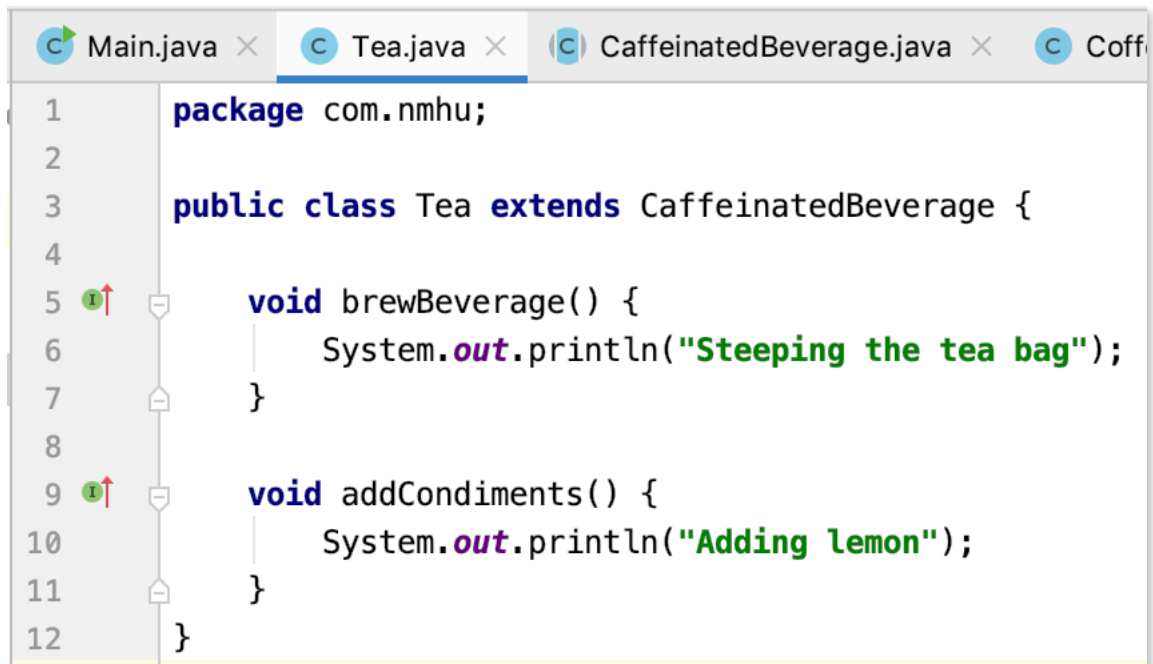
Modify CaffeinatedBeverage to now have prepareRecipe and everything prepareRecipe calls. Note that the implementation of some methods remains abstract. All methods could have been abstract. prepareRecipe is usually not abstract for the sake of the pattern though.



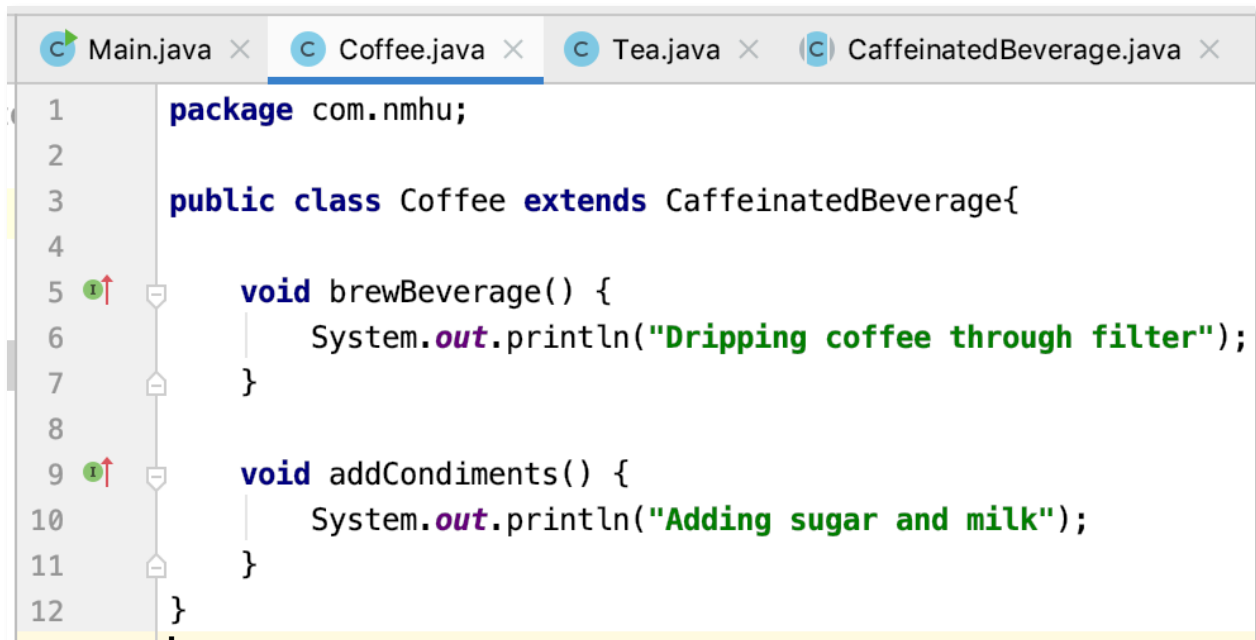
```
1 package com.nmhu;
2
3 public abstract class CaffeinatedBeverage {
4
5     final void prepareRecipe() {
6         getMug();
7         boilWater();
8         brewBeverage(); //genericized name
9         pourInCup();
10        addCondiments(); //genericized name
11    }
12
13    abstract void brewBeverage(); //unknown implementation in interface
14    abstract void addCondiments(); //unknown implementation in interface
15
16    final void getMug() {
17        System.out.println("Got mug");
18    }
19
20    void boilWater() {
21        System.out.println("Boiling water");
22    }
23
24    final void pourInCup() {
25        System.out.println("Pouring in cup");
26    }
27 }
```



Now implement the pieces of the template method or recipe that are different in our concrete classes, and we will have created a proper example of the Template Method Pattern.



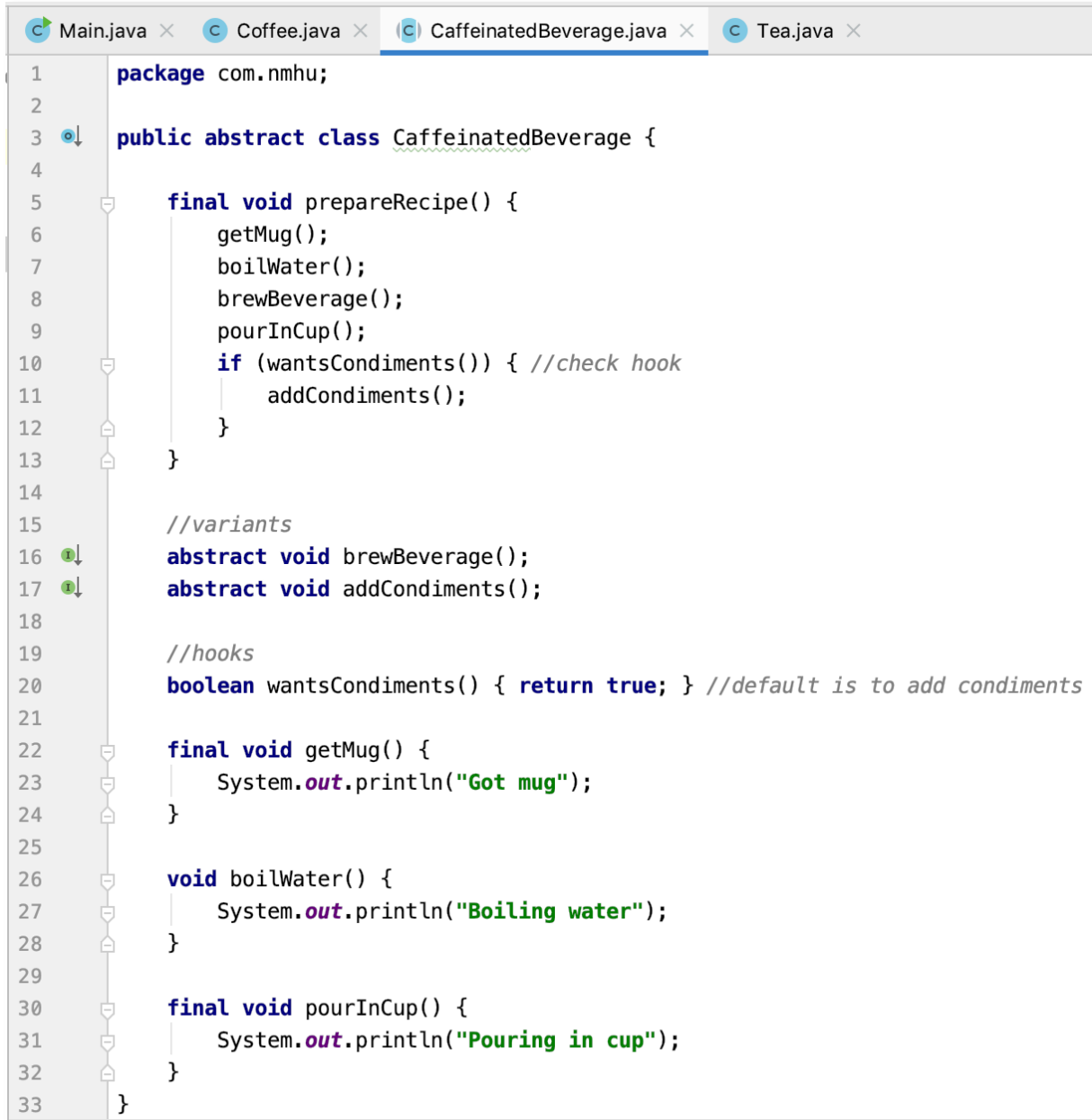
```
1 package com.nmhu;
2
3 public class Tea extends CaffeinatedBeverage {
4
5     void brewBeverage() {
6         System.out.println("Steeping the tea bag");
7     }
8
9     void addCondiments() {
10        System.out.println("Adding lemon");
11    }
12 }
```



```
1 package com.nmhu;
2
3 public class Coffee extends CaffeinatedBeverage{
4
5     void brewBeverage() {
6         System.out.println("Dripping coffee through filter");
7     }
8
9     void addCondiments() {
10        System.out.println("Adding sugar and milk");
11    }
12 }
```

## Step 4 - Adding Hooks to the Pattern

A hook allows the subclass to determine whether or not it wants to override certain behavior. It allows us to hook into the shared template, or opt out of a step.



```
1 package com.nmhu;
2
3 public abstract class CaffeinatedBeverage {
4
5     final void prepareRecipe() {
6         getMug();
7         boilWater();
8         brewBeverage();
9         pourInCup();
10        if (wantsCondiments()) { //check hook
11            addCondiments();
12        }
13    }
14
15    //variants
16    abstract void brewBeverage();
17    abstract void addCondiments();
18
19    //hooks
20    boolean wantsCondiments() { return true; } //default is to add condiments
21
22    final void getMug() {
23        System.out.println("Got mug");
24    }
25
26    void boilWater() {
27        System.out.println("Boiling water");
28    }
29
30    final void pourInCup() {
31        System.out.println("Pouring in cup");
32    }
33 }
```

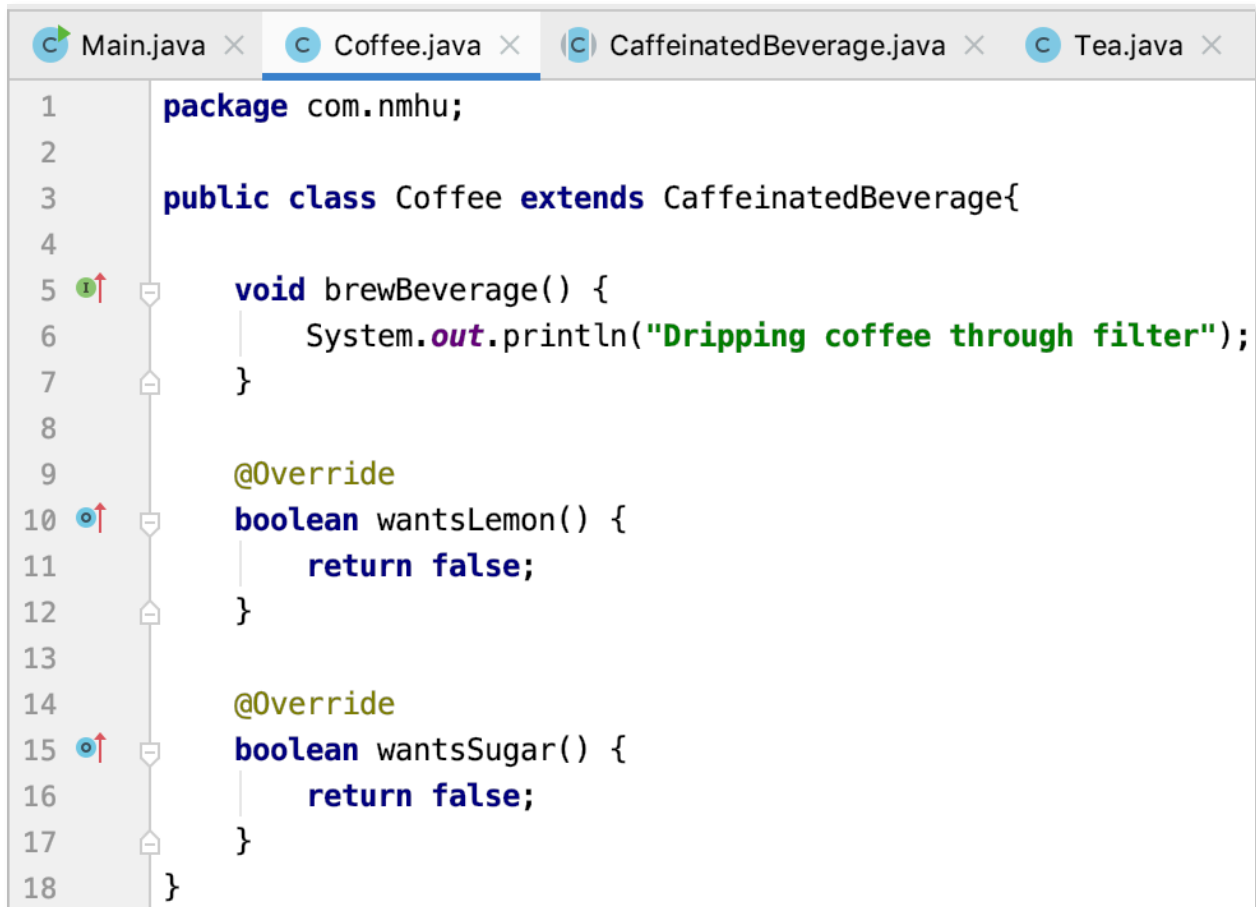
You can add multiple hooks if desired. Here I cover all my bases, though I do not think someone would want lemon in their coffee, but they could want sugar or milk in either, so I set defaults accordingly. Notice that I took out the addCondiments function completely and now these are more discrete steps in the interface.

```
1 package com.nmhu;
2
3 public abstract class CaffeinatedBeverage {
4
5     final void prepareRecipe() {
6         getMug();
7         boilWater();
8         brewBeverage();
9         pourInCup();
10        //check hooks
11        if (wantsLemon()) {
12            addLemon();
13        }
14        if (wantsSugar()) {
15            addSugar();
16        }
17        if (wantsMilk()) {
18            addMilk();
19        }
20    }
21
22    //variants
23    abstract void brewBeverage();
24    //abstract void addCondiments();
25
26    //hooks
27    boolean wantsLemon() { return false; }
28    boolean wantsMilk() { return true; }
29    boolean wantsSugar() { return true; }
30
31    //invariants
32    final void addLemon() {
33        System.out.println("Adding lemon");
```

*continues on next page*

```
34     }
35
36     final void addSugar() {
37         System.out.println("Adding sugar");
38     }
39
40     final void addMilk() {
41         System.out.println("Adding milk");
42     }
43
44     final void getMug() {
45         System.out.println("Got mug");
46     }
47
48     void boilWater() {
49         System.out.println("Boiling water");
50     }
51
52     final void pourInCup() {
53         System.out.println("Pouring in cup");
54     }
55 }
```

Utilizing the hooks now becomes a matter of overriding which ones you want and do not want. I did not have to override Lemon since the default is false, but I show you here anyway.

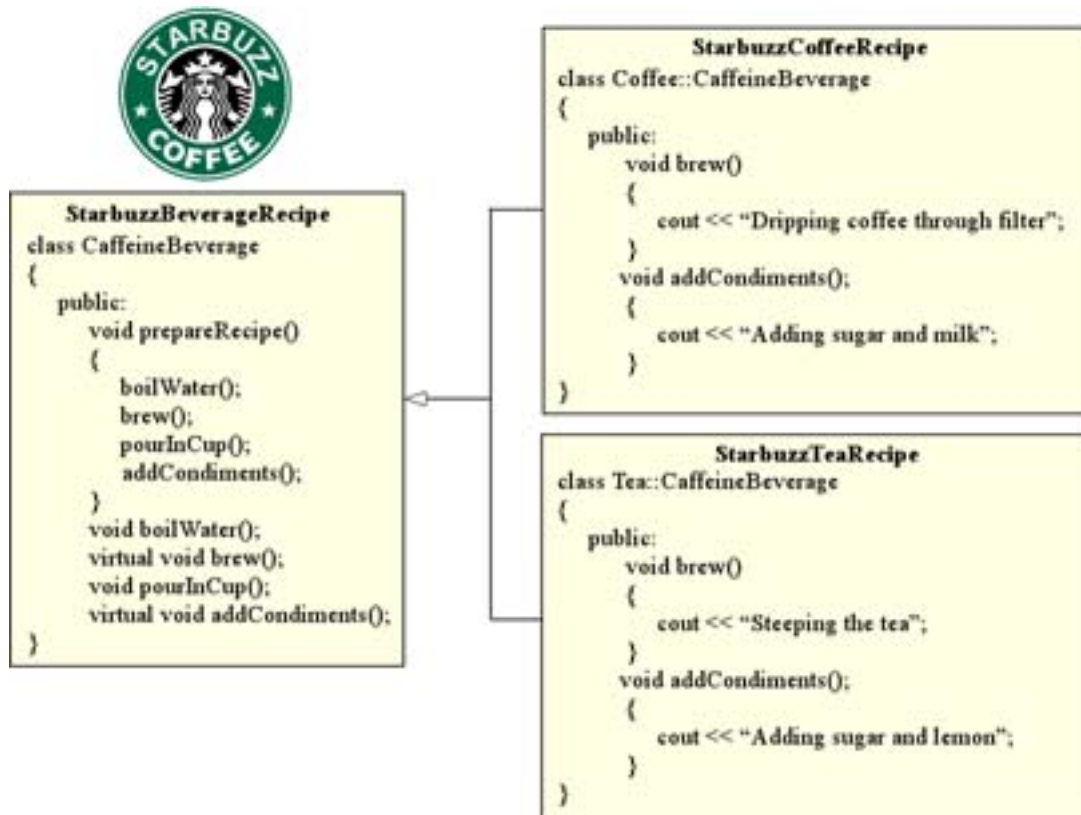


```
1 package com.nmhu;
2
3 public class Coffee extends CaffeinatedBeverage{
4
5     void brewBeverage() {
6         System.out.println("Dripping coffee through filter");
7     }
8
9     @Override
10    boolean wantsLemon() {
11        return false;
12    }
13
14    @Override
15    boolean wantsSugar() {
16        return false;
17    }
18 }
```

Complete Tea on your own. Your Main from Step 1 should still work with no changes.

## Conclusion

Here is a concrete UML similar to what we accomplished:



### STARBUZZ BEVERAGES SIMILAR TO OUR STEP 3 [4]

Here is a simple example in Python for this pattern:

<https://jpython.blogspot.com/2012/11/design-pattern-in-python-template-method.html>

## Assignment (Due before 10PM on Wed. March 1st)

### 1. Due via github

At the Dr. Pepper Museum in Texas, they server warm Dr. Pepper with lemon. Make a new class that extends CaffeineBeverage to make this new drink. Note that you are not heating water, you are heating Dr. Pepper, and you are not brewing anything.

### 2. Answer via email

Which other pattern could we add to this project that would make hooks more efficient?

## Sources

1. [https://ja.wikipedia.org/wiki/Template\\_Method\\_%E3%83%91%E3%82%BF%E3%83%BC%E3%83%B3](https://ja.wikipedia.org/wiki/Template_Method_%E3%83%91%E3%82%BF%E3%83%BC%E3%83%B3)
2. <https://java-design-patterns.com/patterns/template-method/>
3. <https://logannathan.blogspot.com/2011/02/behavioral-patterns-template-method.html>
4. [Defunct source. Archival reference] <https://web.archive.org/web/20150516150600/http://www.cs.uah.edu/~rcoleman/CS307/DesignPatterns/DP10-TemplateMethod.html>