THE TEMPLATE METHOD PATTERN

Chandan R. Rupakheti

Week 6-1

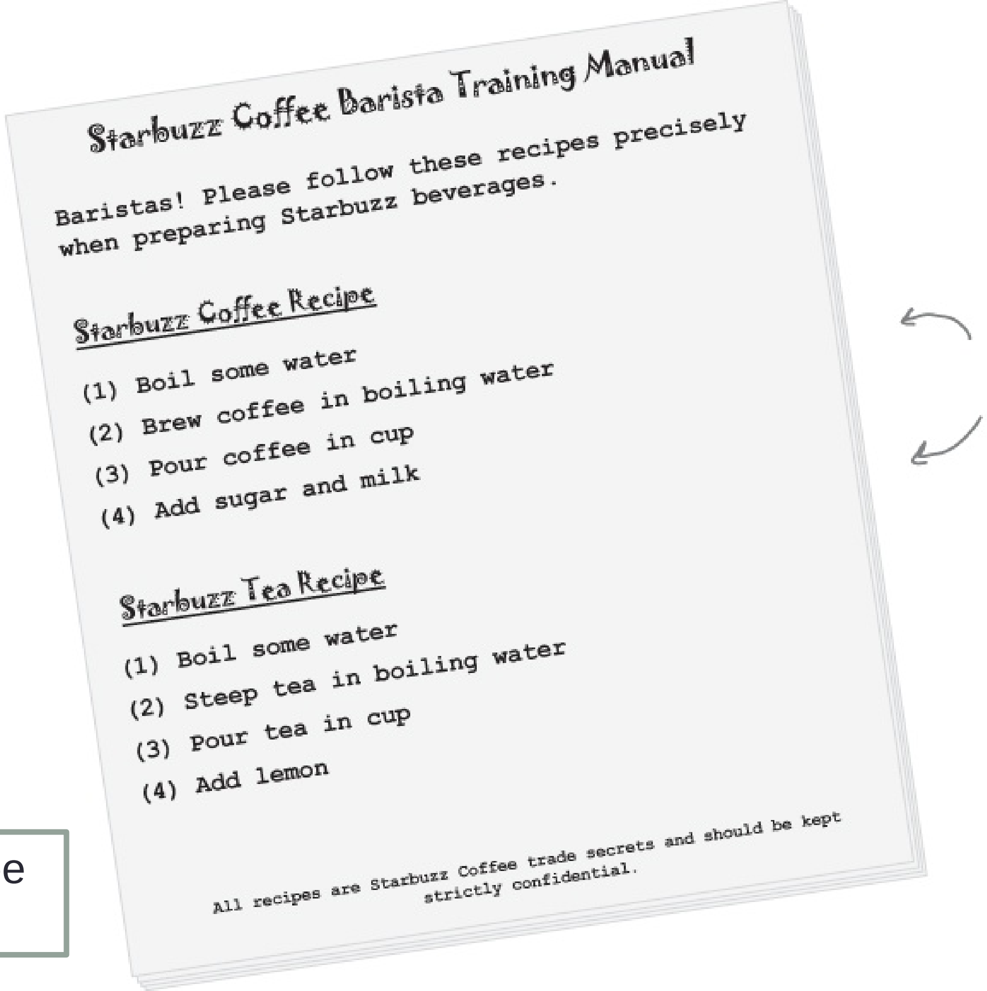
Today …

We're going to learn about a design principle inspired by Hollywood

We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want

MEN WORKING
Yeah, he's a great boss until it comes to getting down in this hole, then it ALL becomes MY job. See what I mean? He's nowhere in sight!

It’s time for some more caffeine



Starbuzz Coffee Barista Training Manual

Baristas! Please follow these recipes precisely when preparing Starbuzz beverages.

Starbuzz Coffee Recipe

(1) Boil some water
(2) Brew coffee in boiling water
(3) Pour coffee in cup
(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water
(2) Steep tea in boiling water
(3) Pour tea in cup
(4) Add lemon

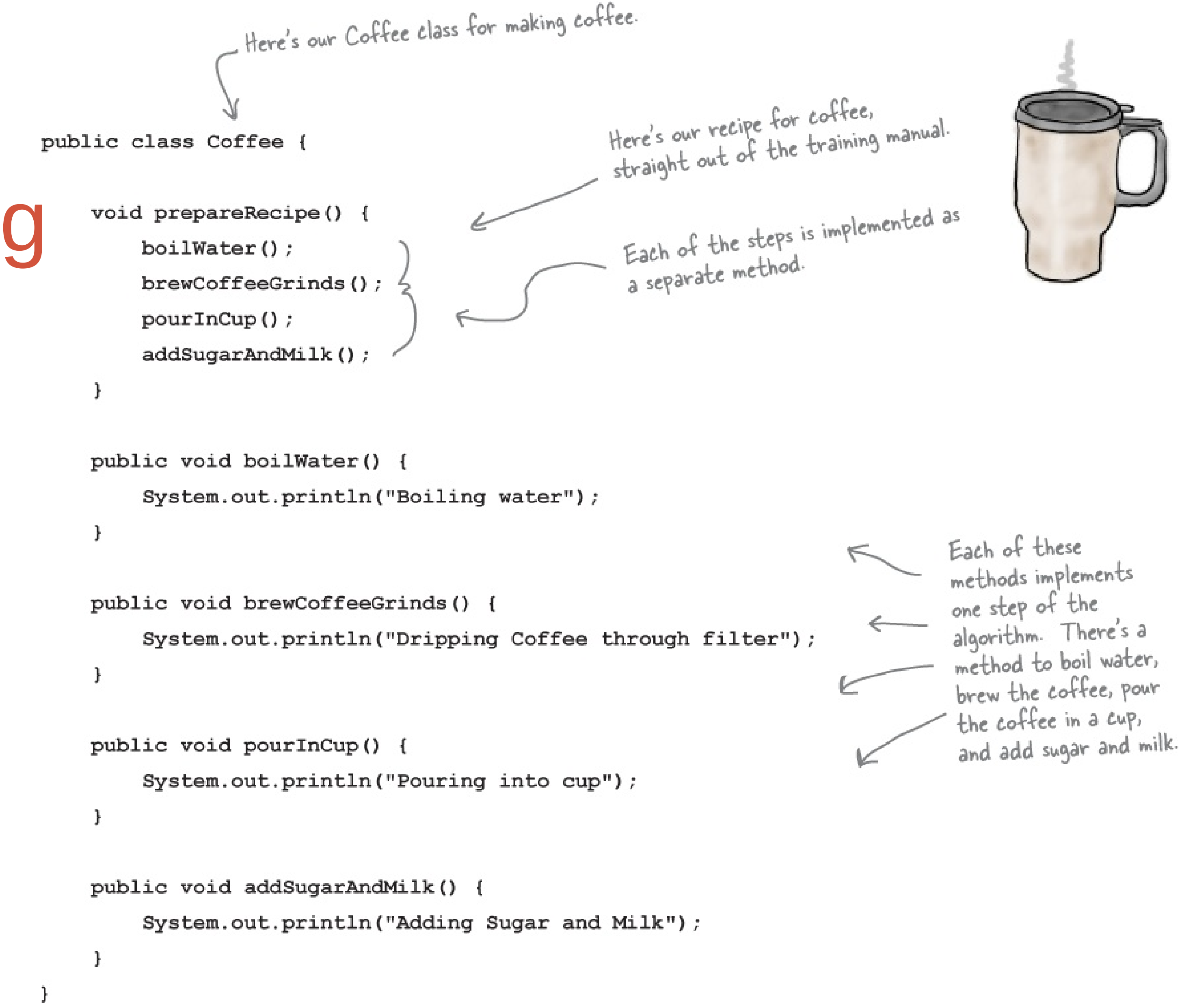
All recipes are Starbuzz Coffee trade secrets and should be kept strictly confidential.

The recipe for coffee looks a lot like the recipe for tea, doesn’t it?

Some people can’t live without their coffee; some people can’t live without their tea. The common ingredient? **Caffeine**, of course!

But there’s more; tea and coffee are made in very similar ways.

Whipping up the coffee class



Here's our Coffee class for making coffee.

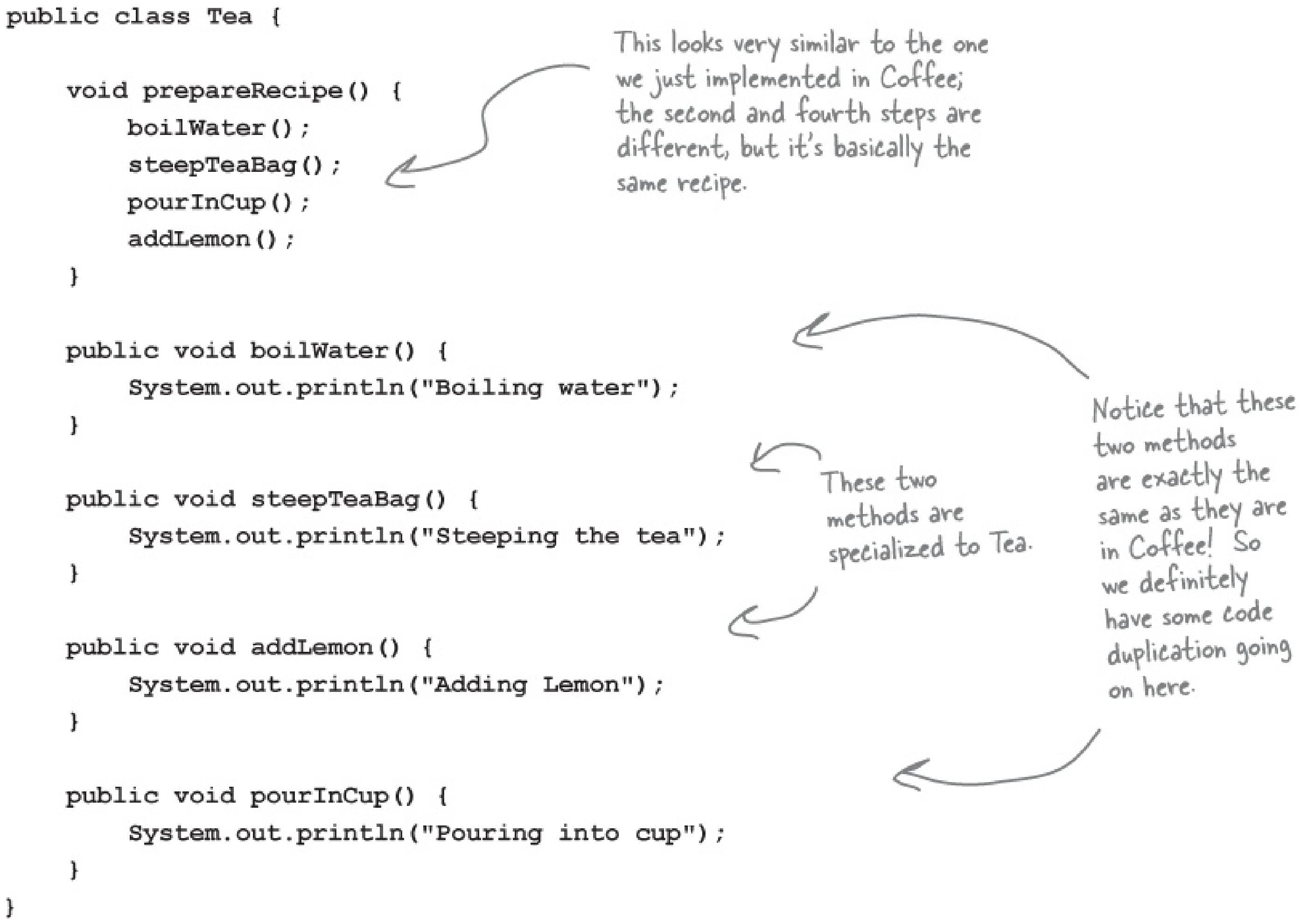
```
public class Coffee {

    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through filter");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }

    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Here's our recipe for coffee, straight out of the training manual.

Each of the steps is implemented as a separate method.

Each of these methods implements one step of the algorithm. There's a method to boil water, brew the coffee, pour the coffee in a cup, and add sugar and milk.

And now the Tea …



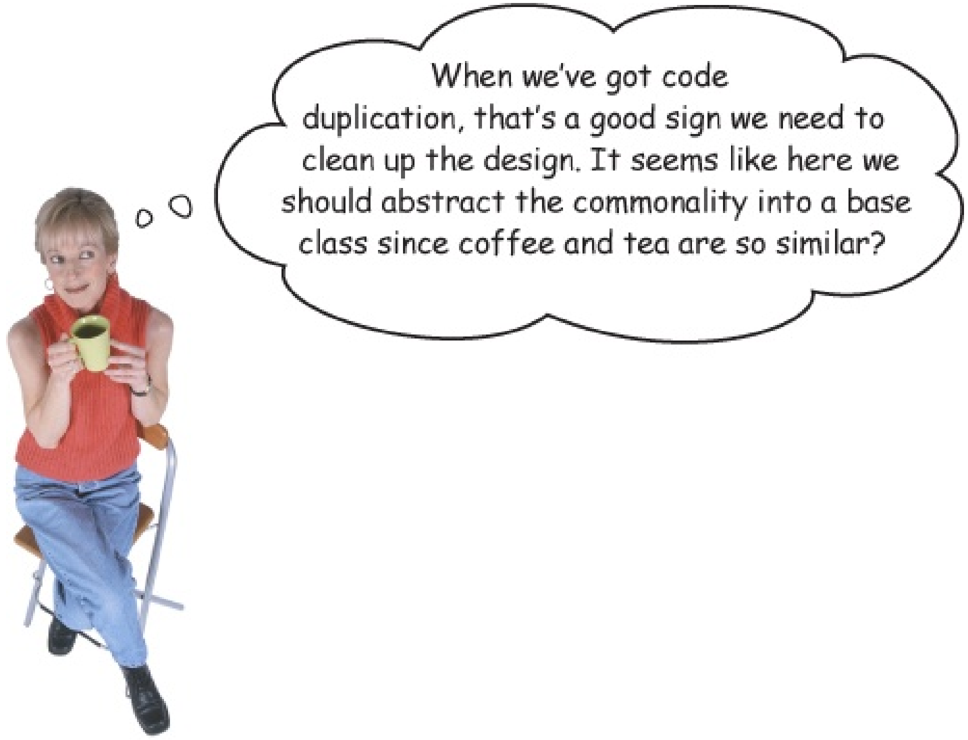
```
public class Tea {

    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling water");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

This looks very similar to the one we just implemented in Coffee; the second and fourth steps are different, but it's basically the same recipe.

These two methods are specialized to Tea.

Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

Observations

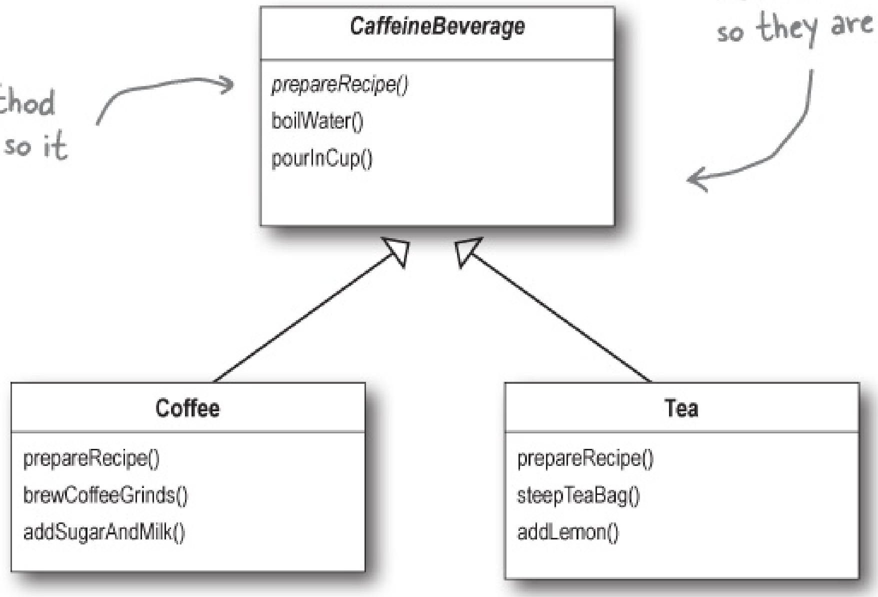

When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?

You've seen that the Coffee and Tea classes have a fair bit of code duplication

Redesign

The prepareRecipe() method differs in each subclass, so it is defined as abstract.

The boilWater() and pourInCup() methods are shared by both subclasses, so they are defined in the superclass.



CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()

Coffee

prepareRecipe()
brewCoffeeGrinds()
addSugarAndMilk()

Tea

prepareRecipe()
steepTeaBag()
addLemon()

Each subclass implements its own recipe.

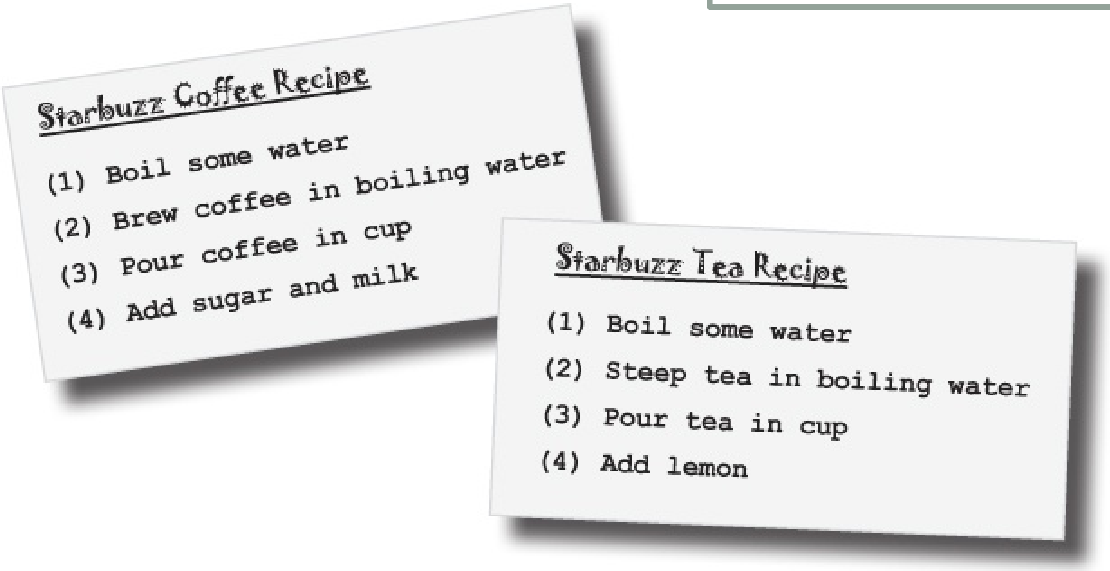
Each subclass overrides the prepareRecipe() method and implements its own recipe.

The methods specific to Coffee and Tea stay in the subclasses.

Did we do a good job here?

Taking the design further …

So what else do Coffee and Tea have in common? Let’s start with the recipes.



Starbuzz Coffee Recipe

(1) Boil some water

(2) Brew coffee in boiling water

(3) Pour coffee in cup

(4) Add sugar and milk

Starbuzz Tea Recipe

(1) Boil some water

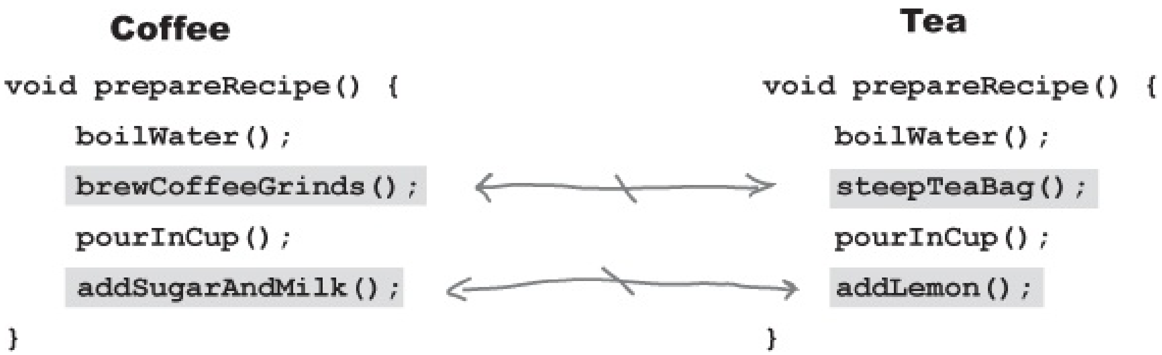
(2) Steep tea in boiling water

(3) Pour tea in cup

(4) Add lemon

Notice that both recipes follow the same algorithm

Abstracting prepareRecipe()



Coffee

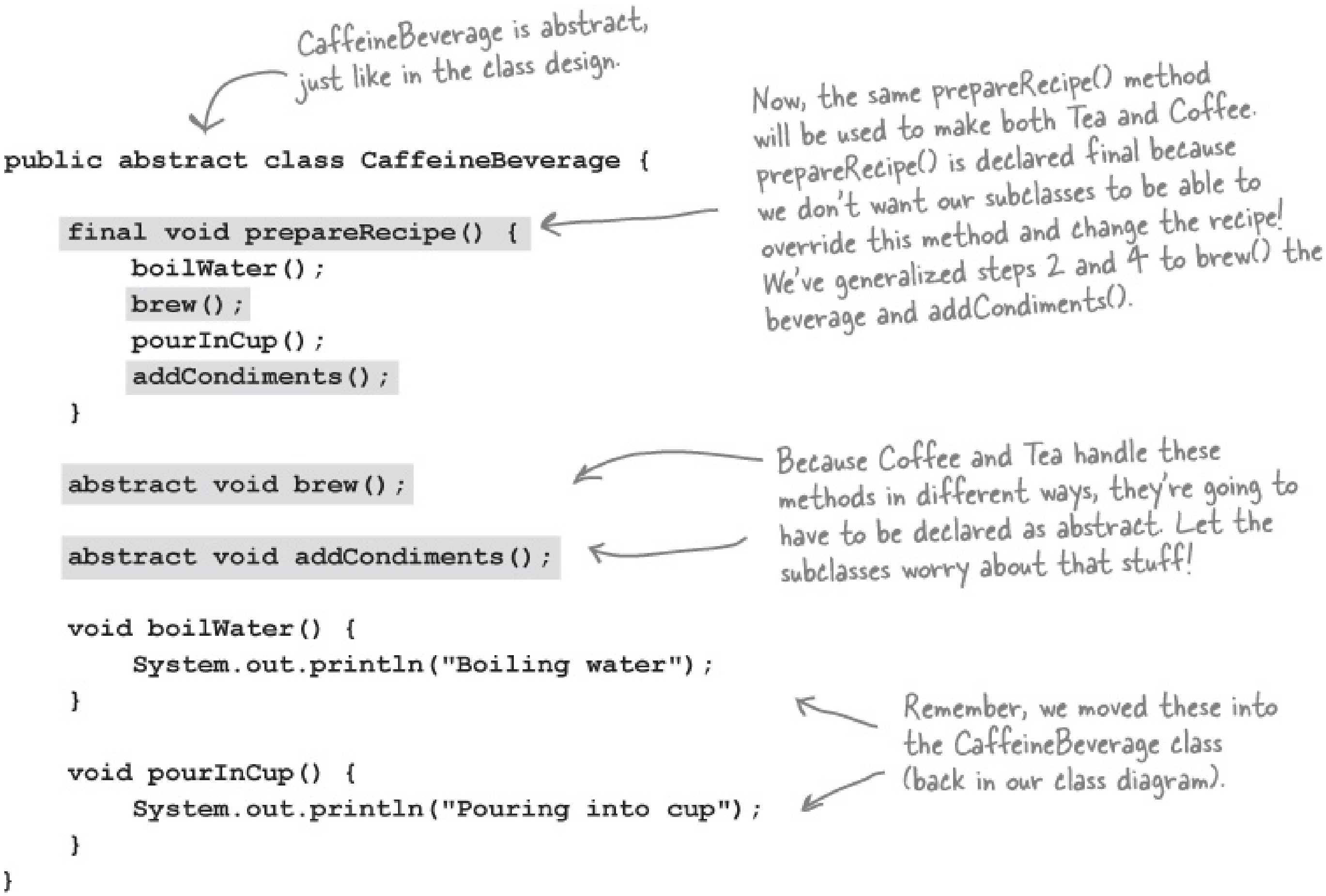
```
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}
```

Tea

```
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

The first problem we have is that Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods, while Tea uses steepTeaBag() and addLemon() methods.

Updating the super class …



CaffeineBeverage is abstract, just like in the class design.

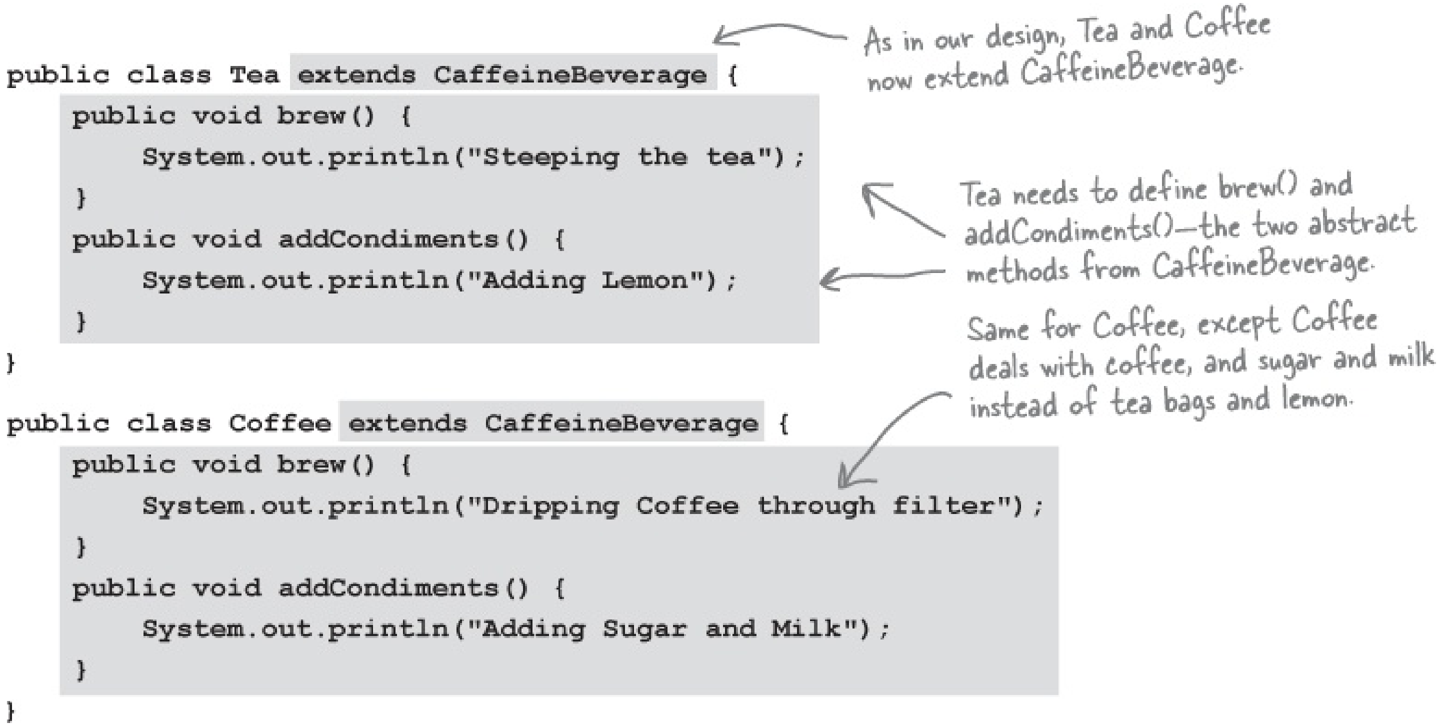
```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

Now deal with Coffee and Tea classes



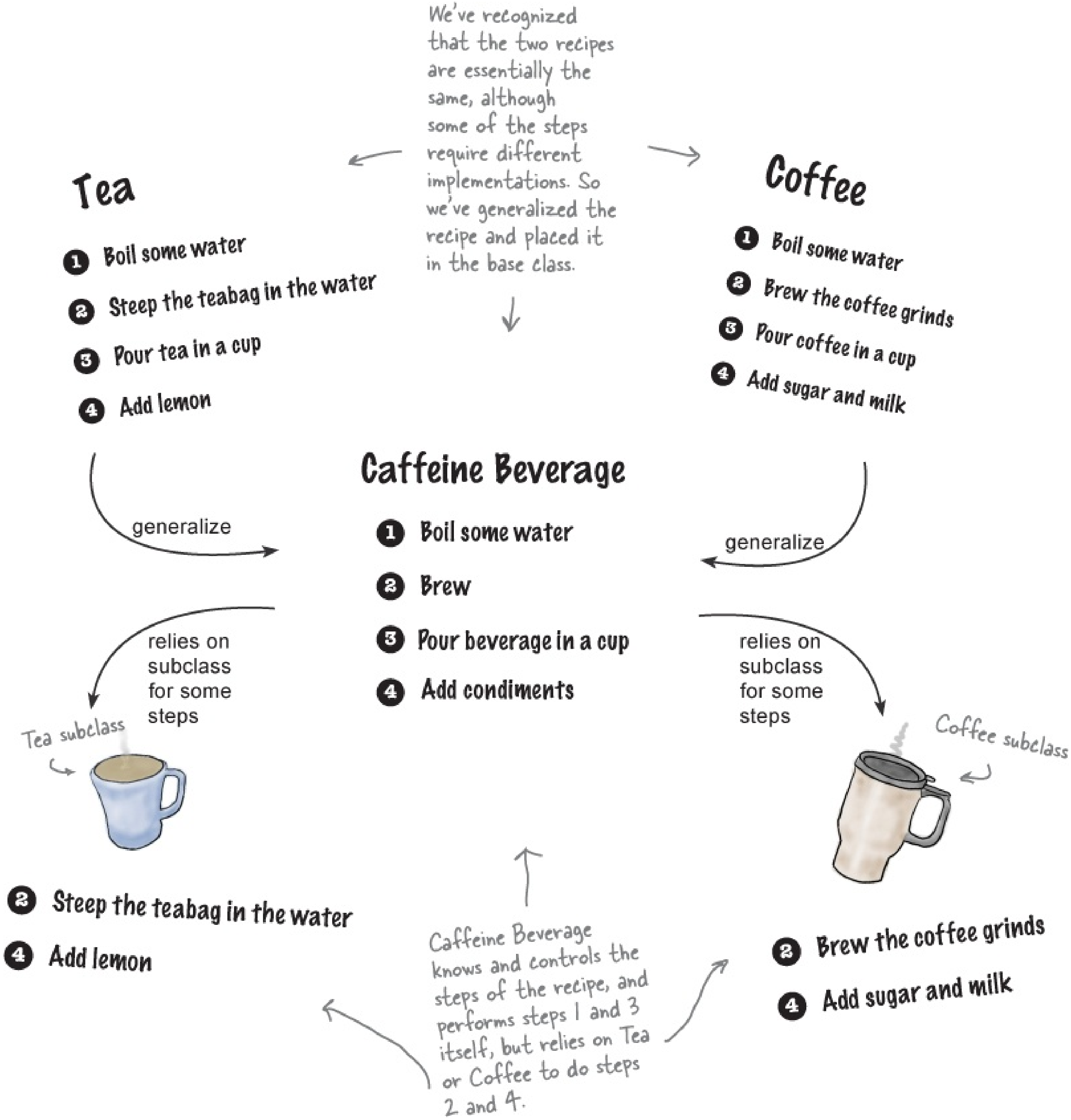
```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

As in our design, Tea and Coffee now extend CaffeineBeverage.

Tea needs to define brew() and addCondiments()—the two abstract methods from CaffeineBeverage.

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

What have we done?



We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea

1. Boil some water
2. Steep the teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

generalize

generalize

Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

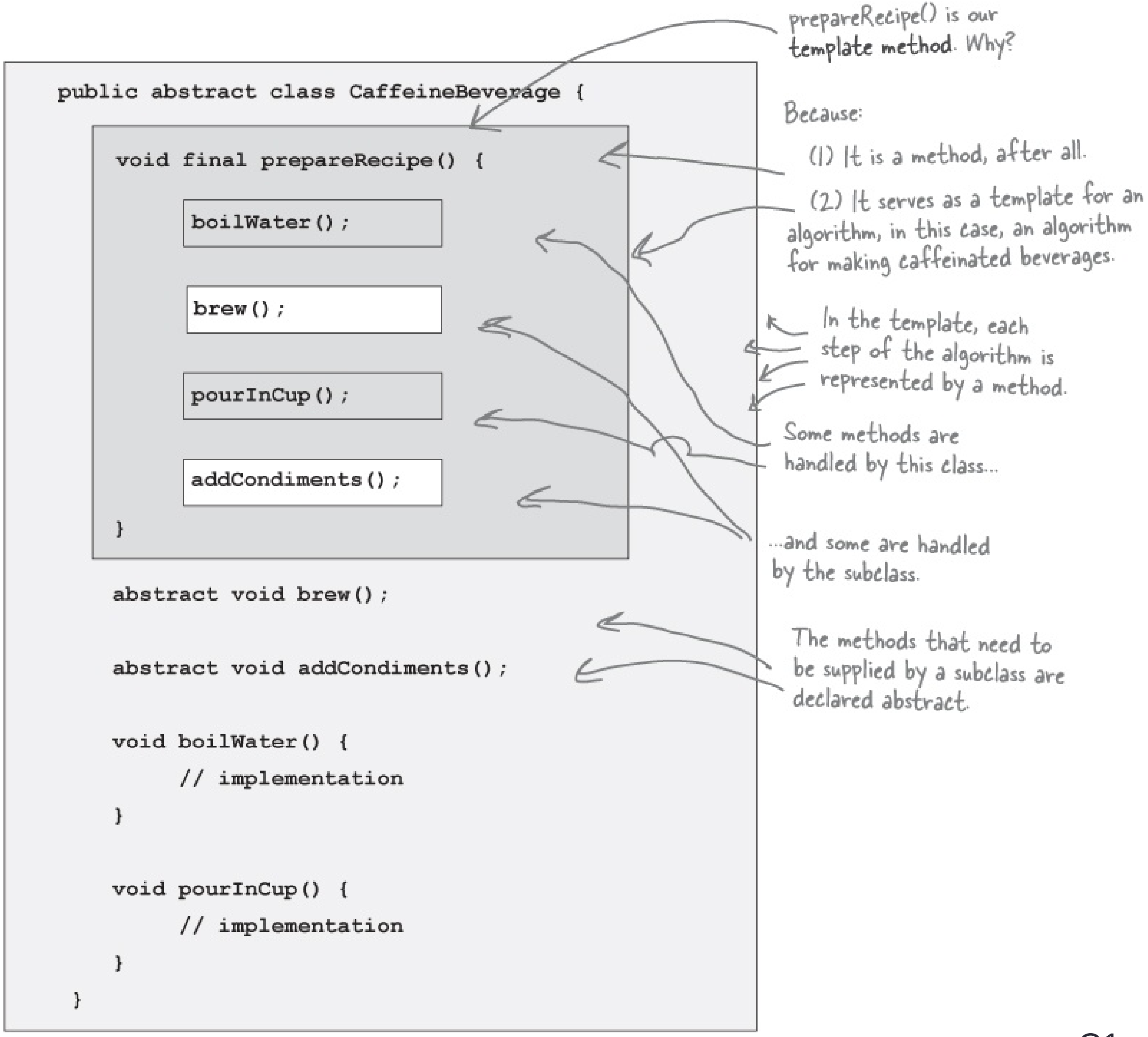
2. Steep the teabag in the water
4. Add lemon

Coffee subclass

2. Brew the coffee grinds
4. Add sugar and milk

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

Meet the Template Method



```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }
}
```

prepareRecipe() is our **template method**. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

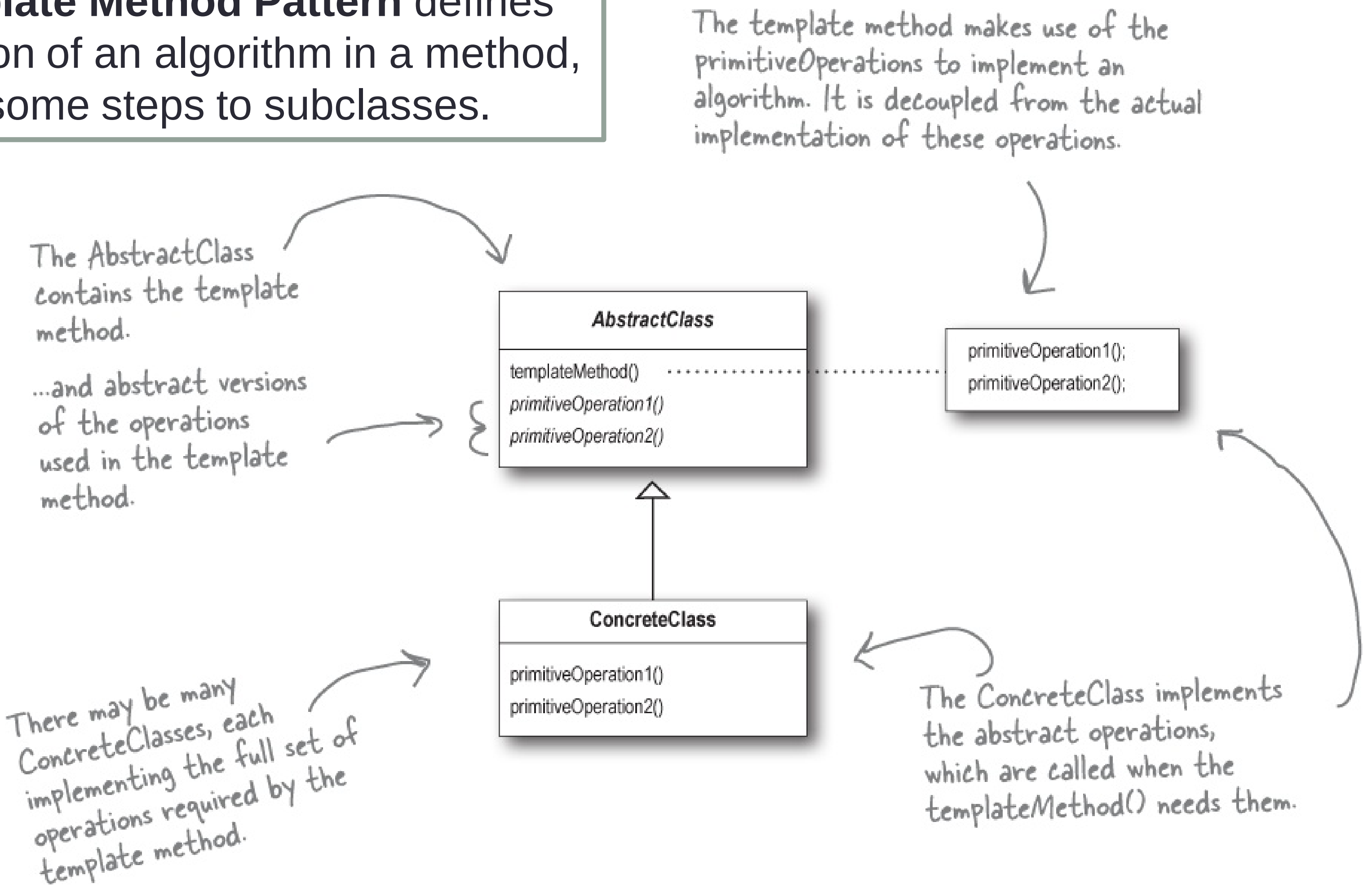
...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

Q1

Template Method Pattern Defined

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses.



The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.

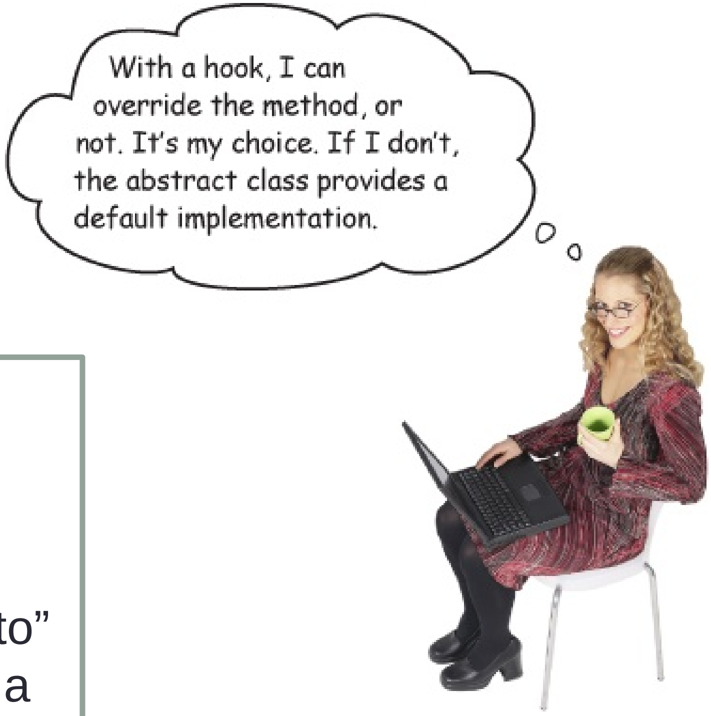
The AbstractClass contains the template method.

...and abstract versions of the operations used in the template method.

There may be many ConcreteClasses, each implementing the full set of operations required by the template method.

The ConcreteClass implements the abstract operations, which are called when the templateMethod() needs them.

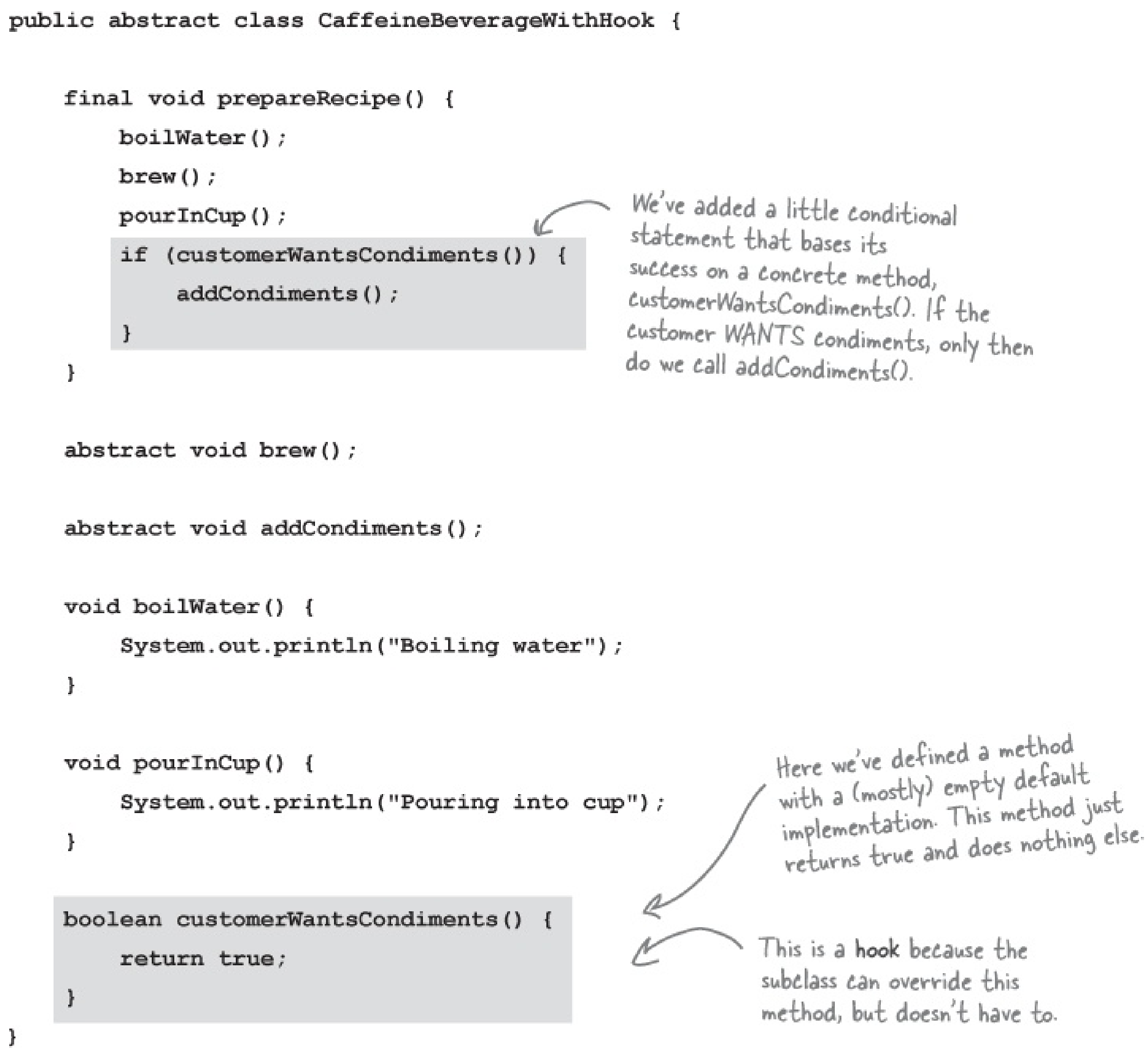
Hooked on Template Method …


With a hook, I can override the method, or not. It's my choice. If I don't, the abstract class provides a default implementation.

A hook is a method that is declared in the abstract class, but only given an empty or default implementation

This gives subclasses the ability to “hook into” the algorithm at various points, if they wish; a subclass is also free to ignore the hook

A simple hook



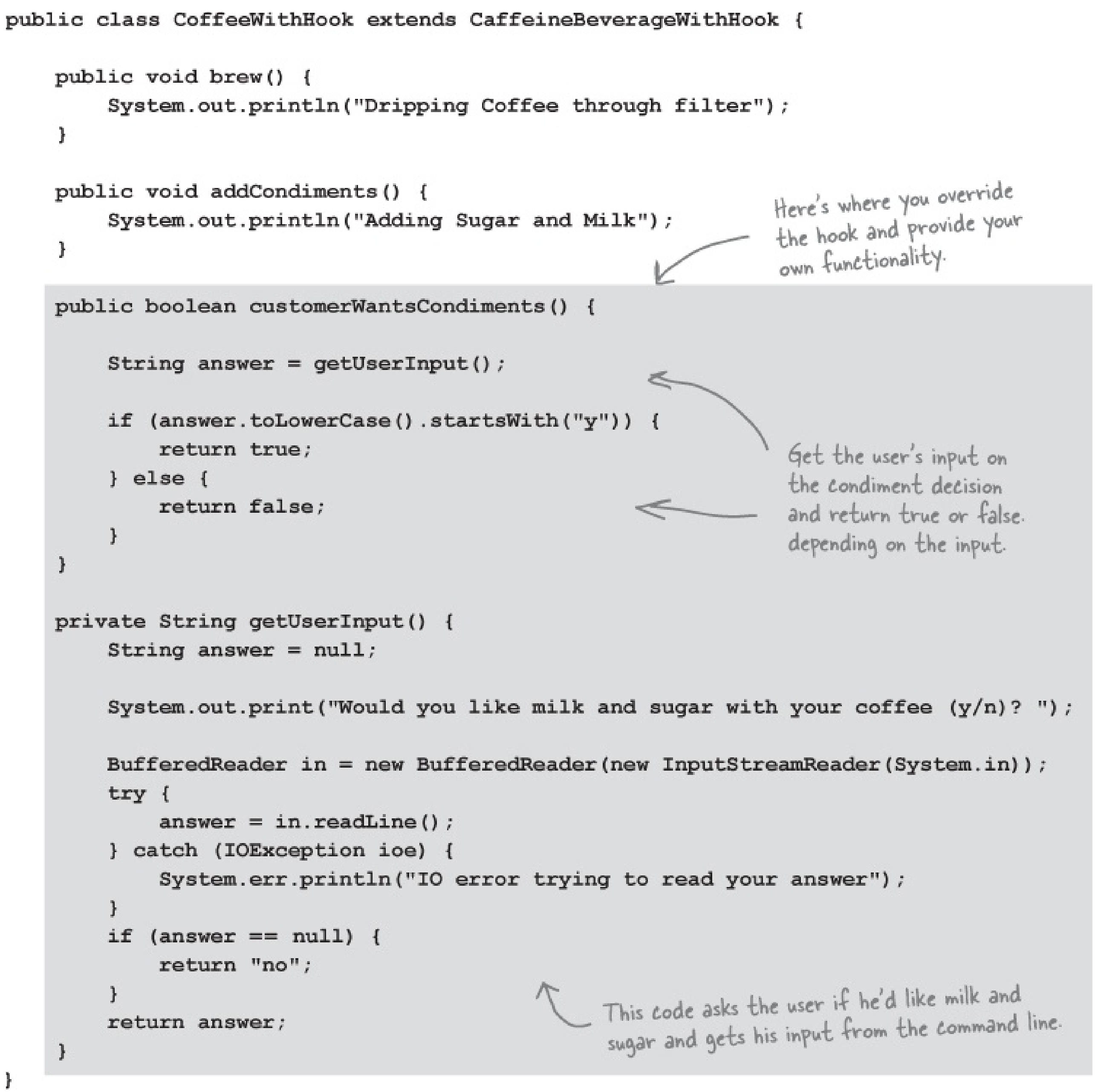
```
public abstract class CaffeineBeverageWithHook {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

Using the hook



```
public class CoffeeWithHook extends CaffeineBeverageWithHook {

    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }

    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }

    public boolean customerWantsCondiments() {

        String answer = getUserInput();

        if (answer.toLowerCase().startsWith("y")) {
            return true;
        } else {
            return false;
        }
    }

    private String getUserInput() {
        String answer = null;

        System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            answer = in.readLine();
        } catch (IOException ioe) {
            System.err.println("IO error trying to read your answer");
        }
        if (answer == null) {
            return "no";
        }
        return answer;
    }
}
```

Here's where you override the hook and provide your own functionality.

Get the user's input on the condiment decision and return true or false. depending on the input.

This code asks the user if he'd like milk and sugar and gets his input from the command line.

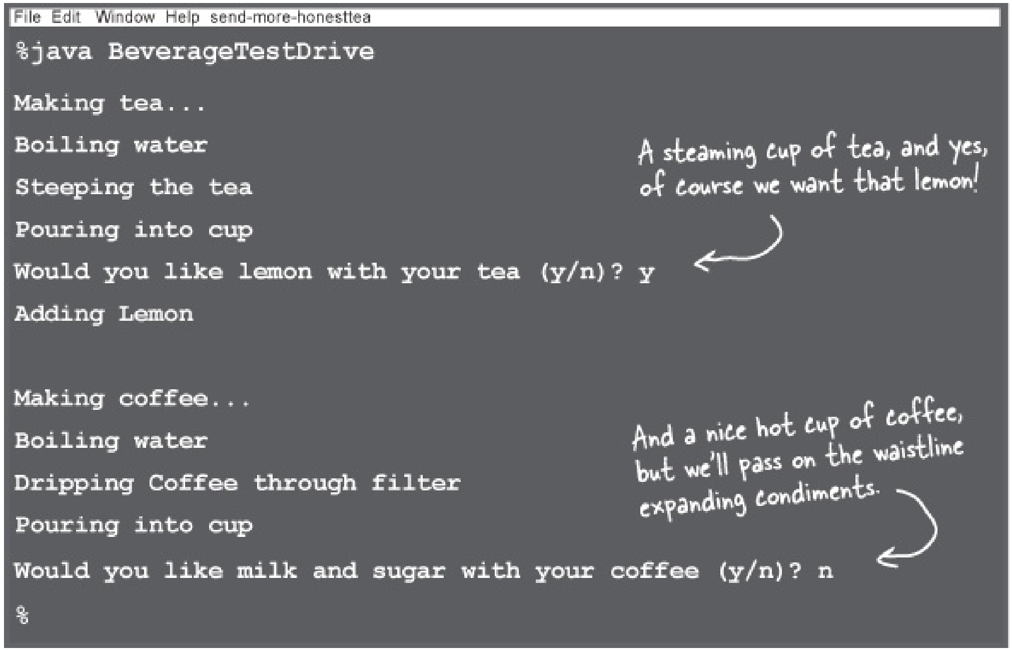
Let’s run the Test Drive

```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();
        CoffeeWithHook coffeeHook = new CoffeeWithHook();

        System.out.println("\nMaking tea...");
        teaHook.prepareRecipe();

        System.out.println("\nMaking coffee...");
        coffeeHook.prepareRecipe();
    }
}
```

Create a tea.

A coffee.

And call prepareRecipe() on both!



```
%java BeverageTestDrive
Making tea...
Boiling water
Steeping the tea
Pouring into cup
Would you like lemon with your tea (y/n)? y
Adding Lemon

Making coffee...
Boiling water
Dripping Coffee through filter
Pouring into cup
Would you like milk and sugar with your coffee (y/n)? n
%
```

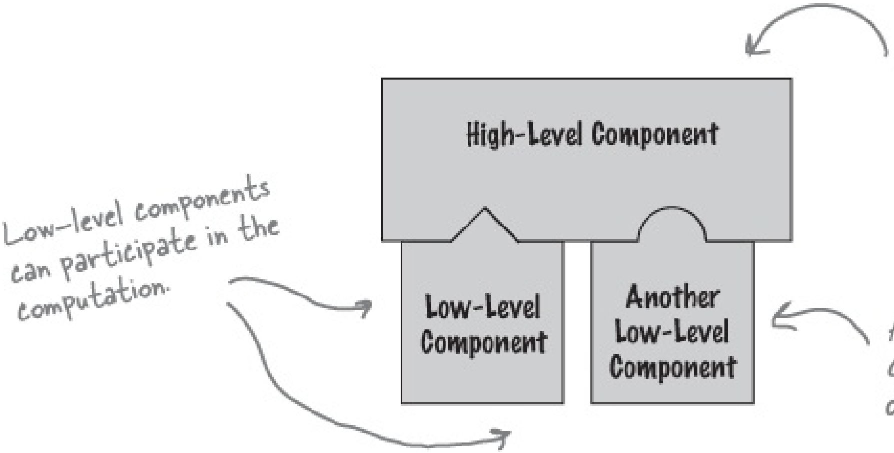
A steaming cup of tea, and yes, of course we want that lemon!

And a nice hot cup of coffee, but we’ll pass on the waistline expanding condiments.

Q2

The Hollywood Principle

Don’t call us, we’ll call you!


High-Level Component
Low-Level Component
Another Low-Level Component
Low-level components can participate in the computation.

But the high-level components control when and how.

A low-level component never calls a high-level component directly.

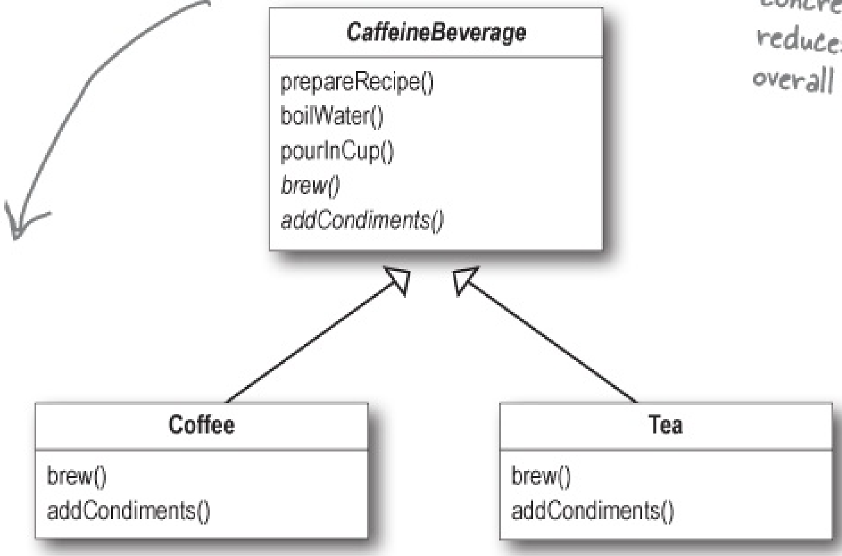
You’ve heard me say it before, and I’ll say it again: don’t call me, I’ll call you!

The Hollywood Principle gives us a way to prevent **dependency rot**

Hollywood Principle and Template Method

CaffeineBeverage is our high-level component. It has control over the algorithm for the recipe, and calls on the subclasses only when they're needed for an implementation of a method.

Clients of beverages will depend on the CaffeineBeverage abstraction rather than a concrete Tea or Coffee, which reduces dependencies in the overall system.



CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()
brew()
addCondiments()

Coffee

brew()
addCondiments()

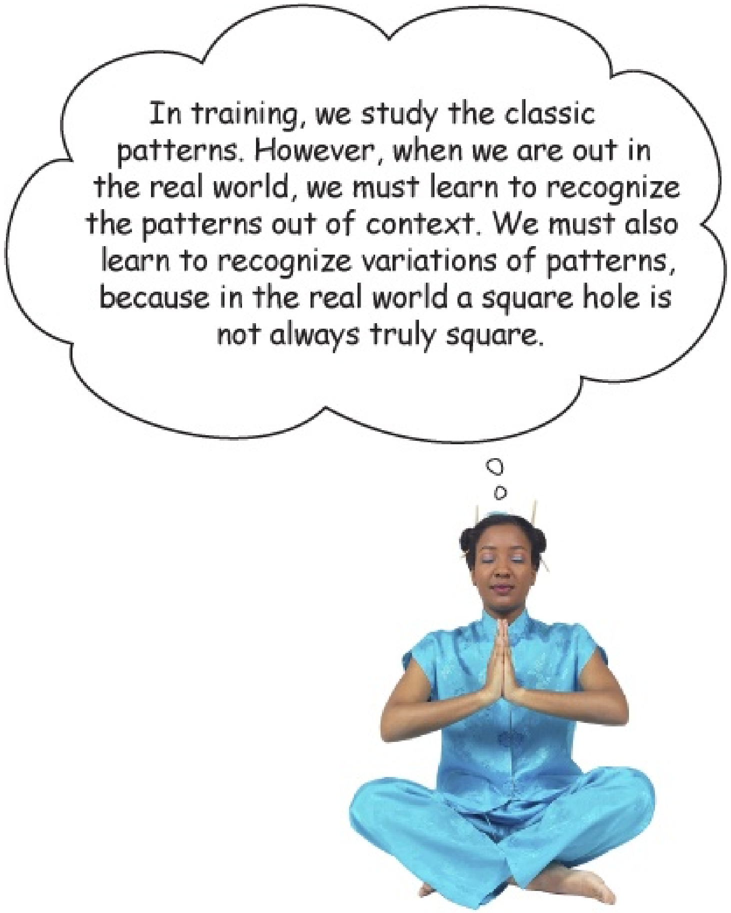
Tea

brew()
addCondiments()

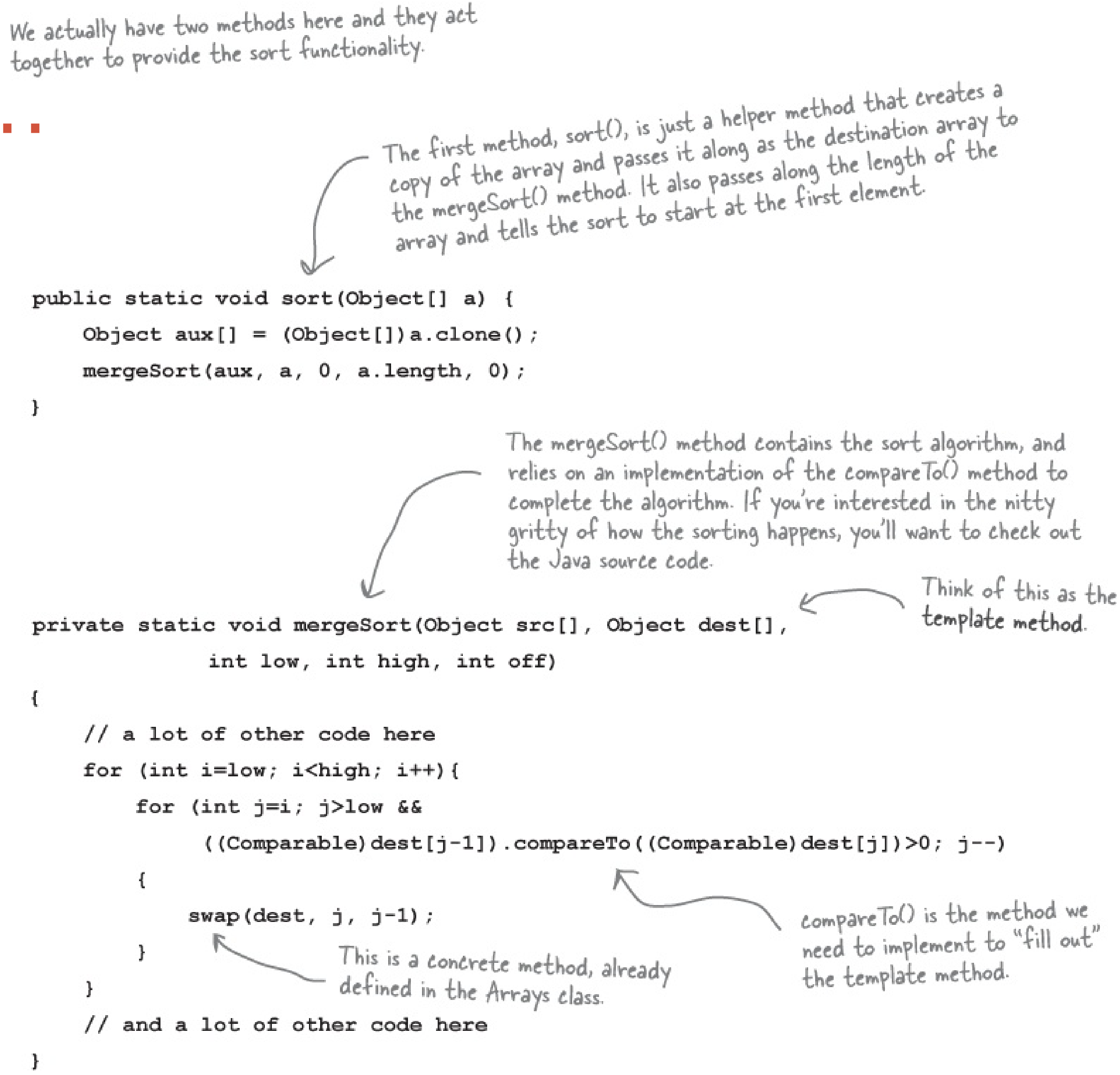
The subclasses are used simply to provide implementation details.

Tea and Coffee never call the abstract class directly without being "called" first.

Template Methods in the Wild


In training, we study the classic patterns. However, when we are out in the real world, we must learn to recognize the patterns out of context. We must also learn to recognize variations of patterns, because in the real world a square hole is not always truly square.

Sorting …



We actually have two methods here and they act together to provide the sort functionality.

The first method, sort(), is just a helper method that creates a copy of the array and passes it along as the destination array to the mergeSort() method. It also passes along the length of the array and tells the sort to start at the first element.

```
public static void sort(Object[] a) {
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

The mergeSort() method contains the sort algorithm, and relies on an implementation of the compareTo() method to complete the algorithm. If you're interested in the nitty gritty of how the sorting happens, you'll want to check out the Java source code.

Think of this as the template method.

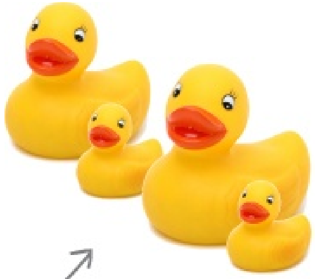
```
private static void mergeSort(Object src[], Object dest[],
                  int low, int high, int off)
{
    // a lot of other code here
    for (int i=low; i<high; i++){
        for (int j=i; j>low &&
             ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    // and a lot of other code here
}
```

compareTo() is the method we need to implement to "fill out" the template method.

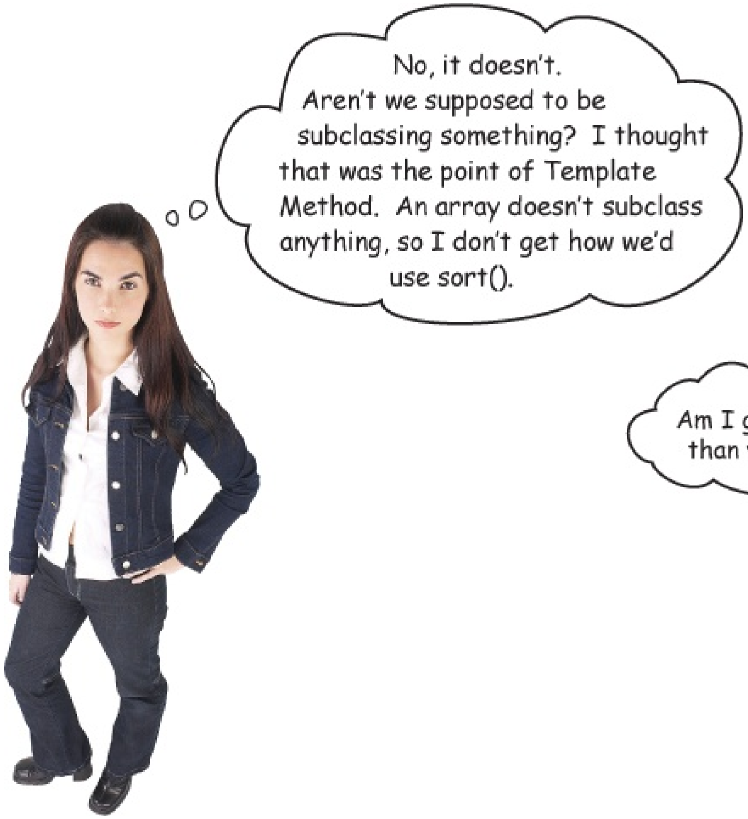
This is a concrete method, already defined in the Arrays class.

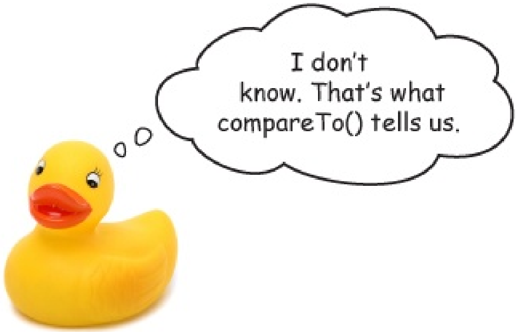
We’ve got some duck to sort …

The sort() template method in Arrays gives us the algorithm, but you need to tell it how to compare ducks, which you do by implementing the compareTo() method. Make sense?

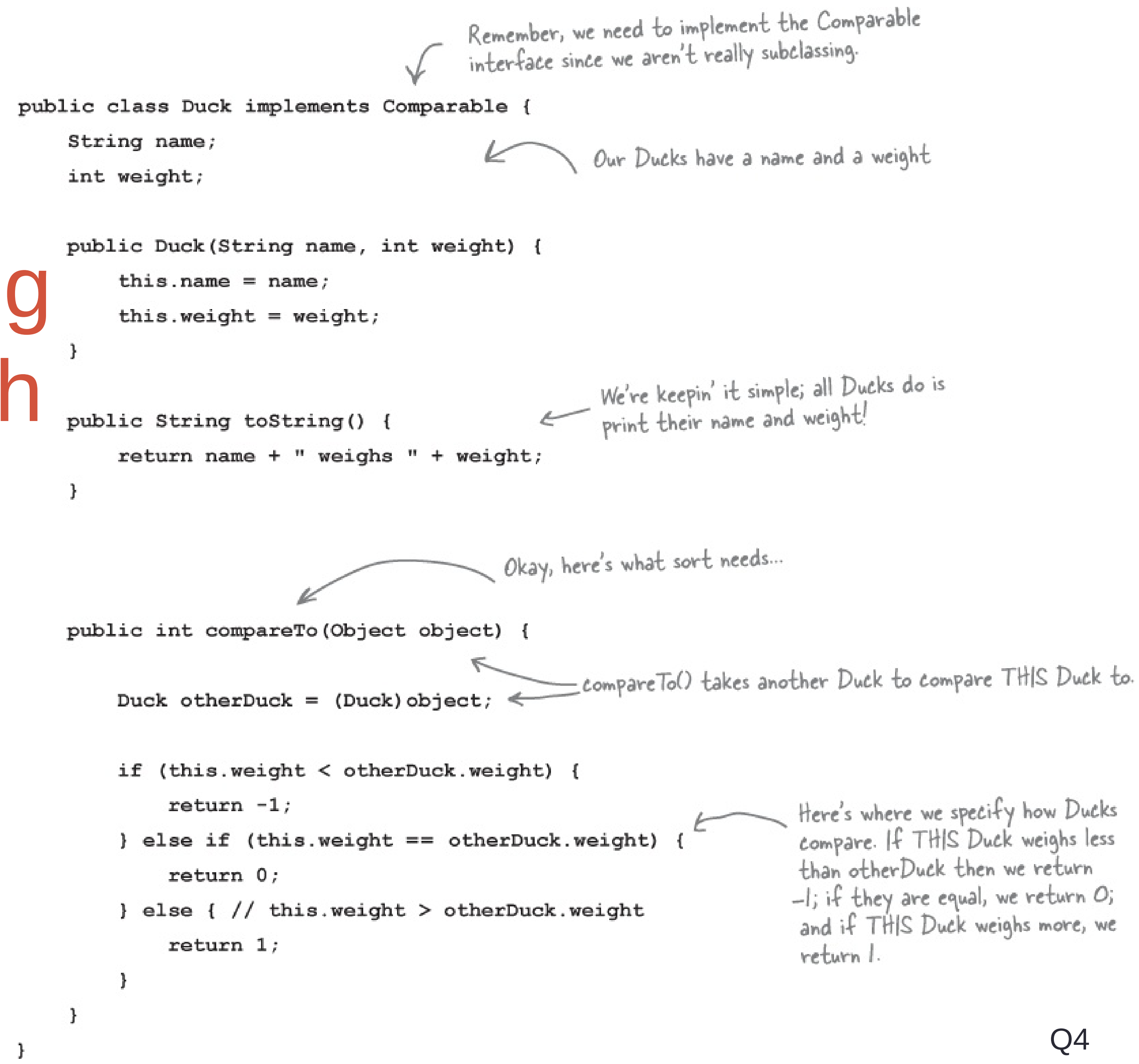


We’ve got an array of Ducks we need to sort.


No, it doesn’t. Aren’t we supposed to be subclassing something? I thought that was the point of Template Method. An array doesn’t subclass anything, so I don’t get how we’d use sort().



Comparing Ducks with Ducks



```
public class Duck implements Comparable {
    String name;
    int weight;

    public Duck(String name, int weight) {
        this.name = name;
        this.weight = weight;
    }

    public String toString() {
        return name + " weighs " + weight;
    }


    public int compareTo(Object object) {

        Duck otherDuck = (Duck)object;

        if (this.weight < otherDuck.weight) {
            return -1;
        } else if (this.weight == otherDuck.weight) {
            return 0;
        } else { // this.weight > otherDuck.weight
            return 1;
        }
    }
}
```

Remember, we need to implement the Comparable interface since we aren't really subclassing.

Our Ducks have a name and a weight

We're keepin' it simple; all Ducks do is print their name and weight!

Okay, here's what sort needs...

compareTo() takes another Duck to compare THIS Duck to.

Here's where we specify how Ducks compare. If THIS Duck weighs less than otherDuck then we return -1; if they are equal, we return 0; and if THIS Duck weighs more, we return 1.

Q4

Recap

A “template method” defines the steps of an algorithm, deferring to subclasses for the implementation of those steps

The template method’s abstract class may define concrete methods, abstract methods, and hooks

The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules