

# Lecture 09: The Template Method Pattern

---

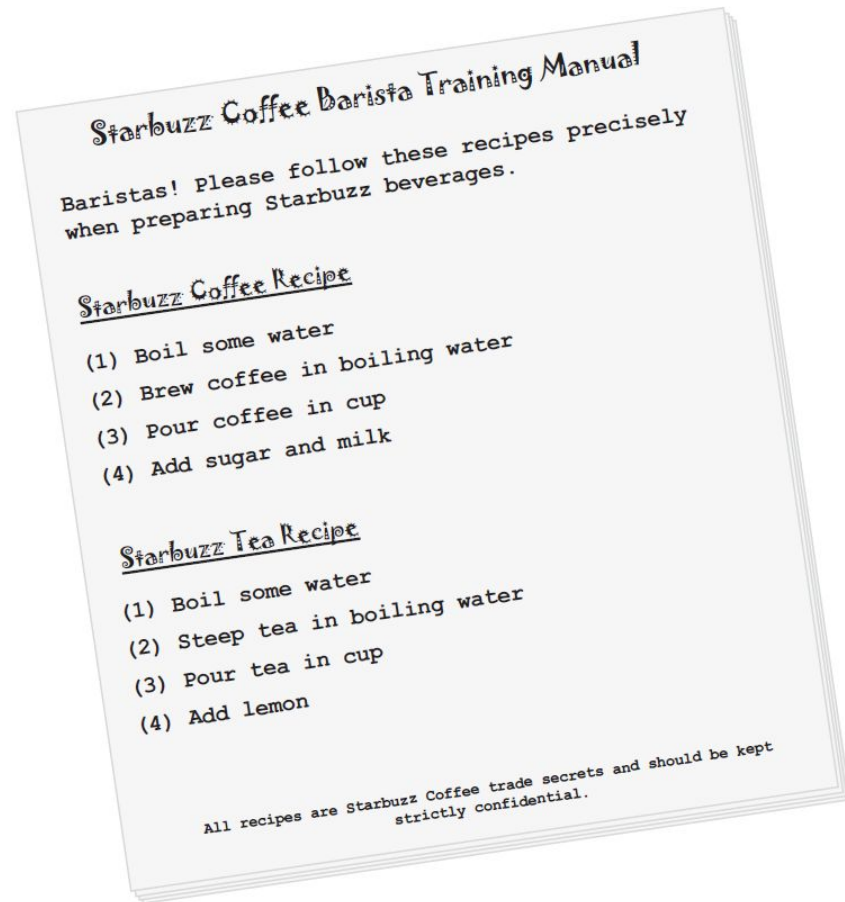
SE313, Software Design and Architecture  
Damla Oguz

# Chapter 8: The Template Method Pattern

- We've encapsulated object creation, method invocation, complex interfaces...
- In this lecture, we're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want.
- We're even going to learn about a design principle inspired by Hollywood.

# It's time for some more caffeine

- 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. Let's check it out:



Let's write some code for creating coffee and tea.  
Here's the 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.

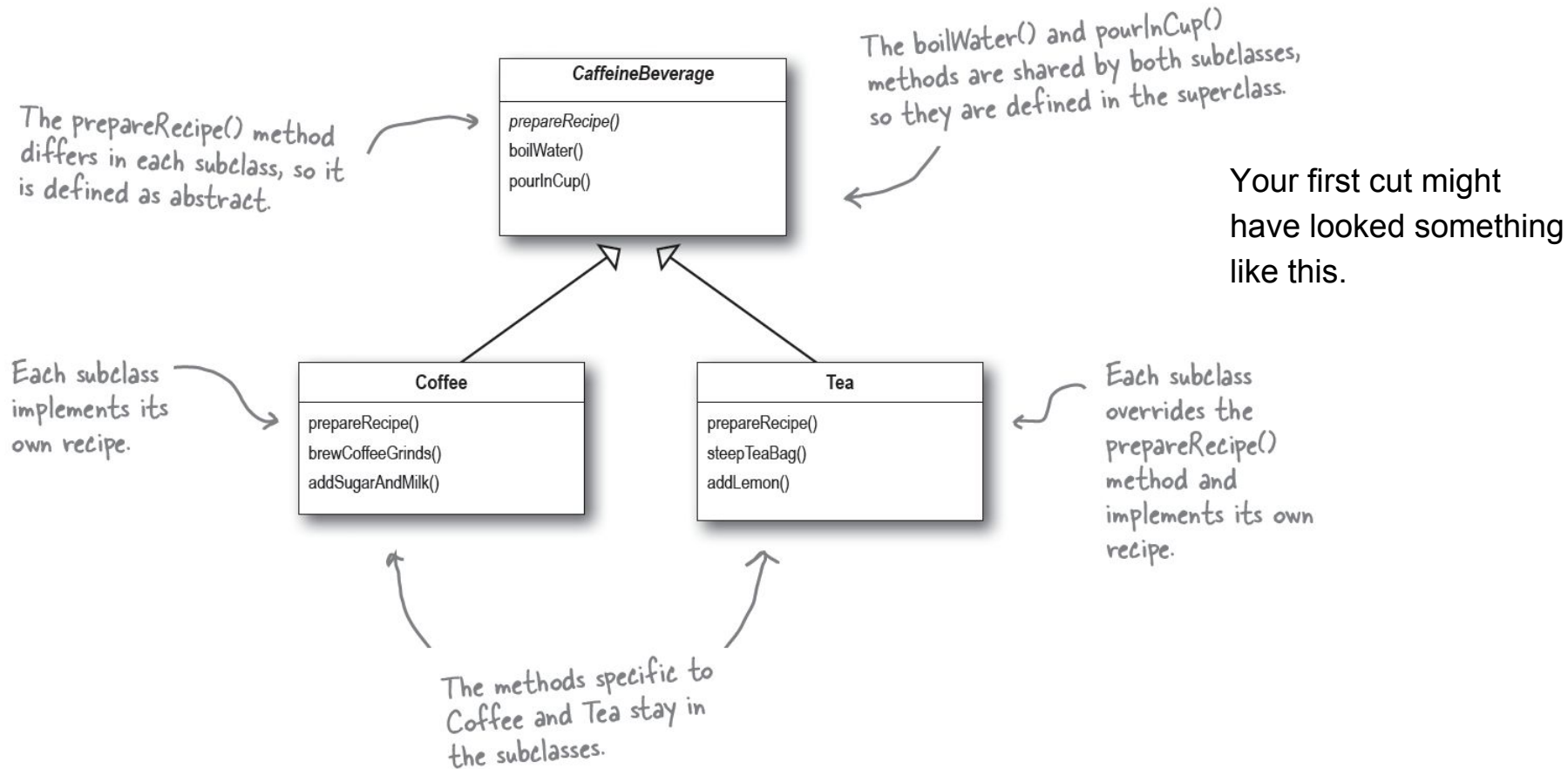
Notice that these two methods are exactly the same as they are in Coffee! So we definitely have some code duplication going on here.

# Is everything fine?



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?

# Sir, may I abstract your Coffee, Tea?



# 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

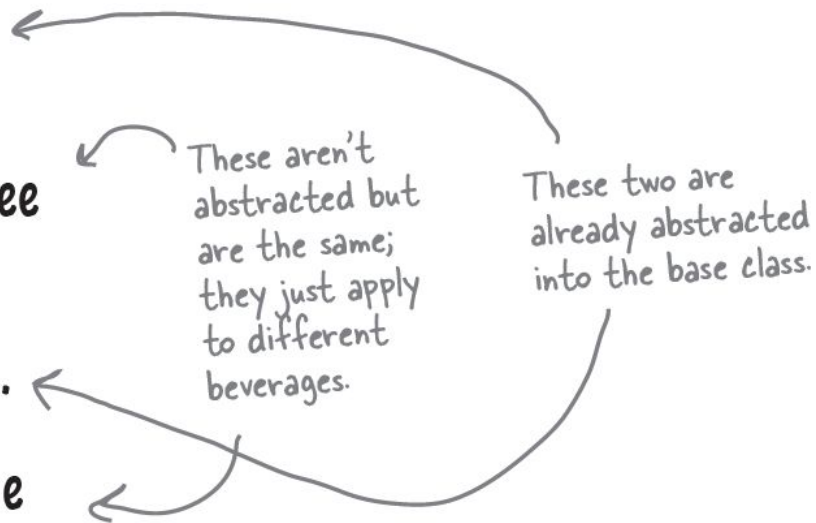
- (1) Boil some water
- (2) Steep tea in boiling water
- (3) Pour tea in cup
- (4) Add lemon



# Taking the design further... (cont.)

- Notice that both recipes follow the same algorithm:

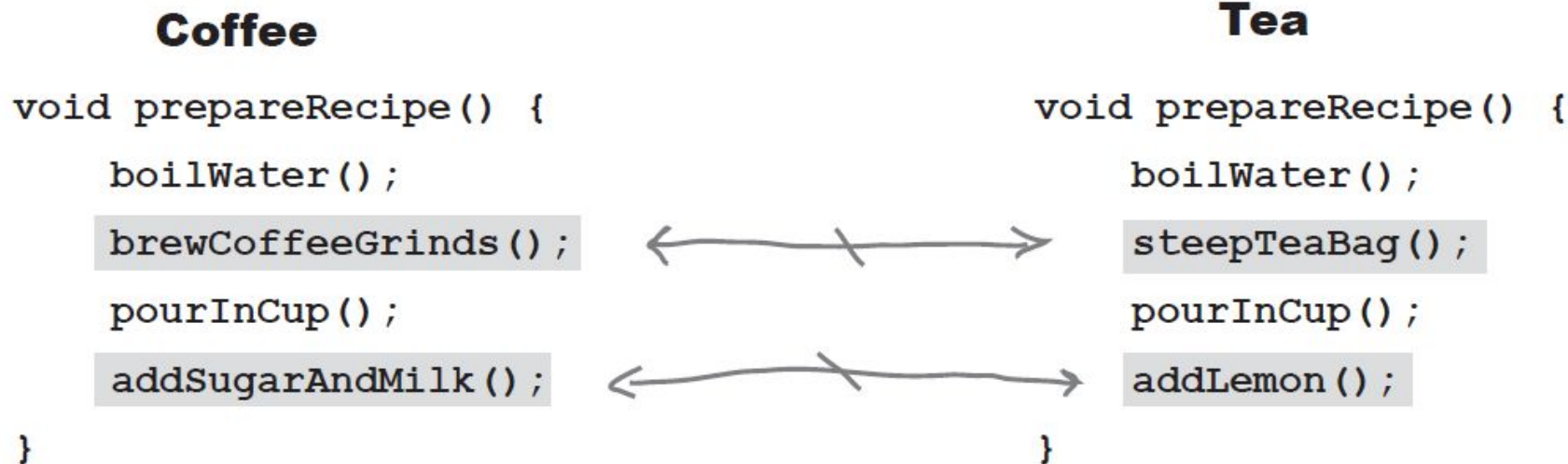
- ❶ Boil some water.
- ❷ Use the hot water to extract the coffee or tea.
- ❸ Pour the resulting beverage into a cup.
- ❹ Add the appropriate condiments to the beverage.



- So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...

# Abstracting prepareRecipe()

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



## Abstracting prepareRecipe() (cont.)

- Let's think through this: steeping and brewing aren't so different; they're pretty analogous.
- So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.
- Likewise, adding sugar and milk is pretty much the same as adding a lemon: both are adding condiments to the beverage.
- Let's also make up a new method name, addCondiments(), to handle this.

## Abstracting prepareRecipe() (cont.)

- So, our new prepareRecipe() method will look like this:

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

## Abstracting prepareRecipe() (cont.)

2. Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass...

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).

## Abstracting prepareRecipe() (cont.)

3. Finally, we need to deal with the Coffee and Tea classes. They now rely on CaffeineBeverage to handle the recipe, so they just need to handle brewing and condiments...

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

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

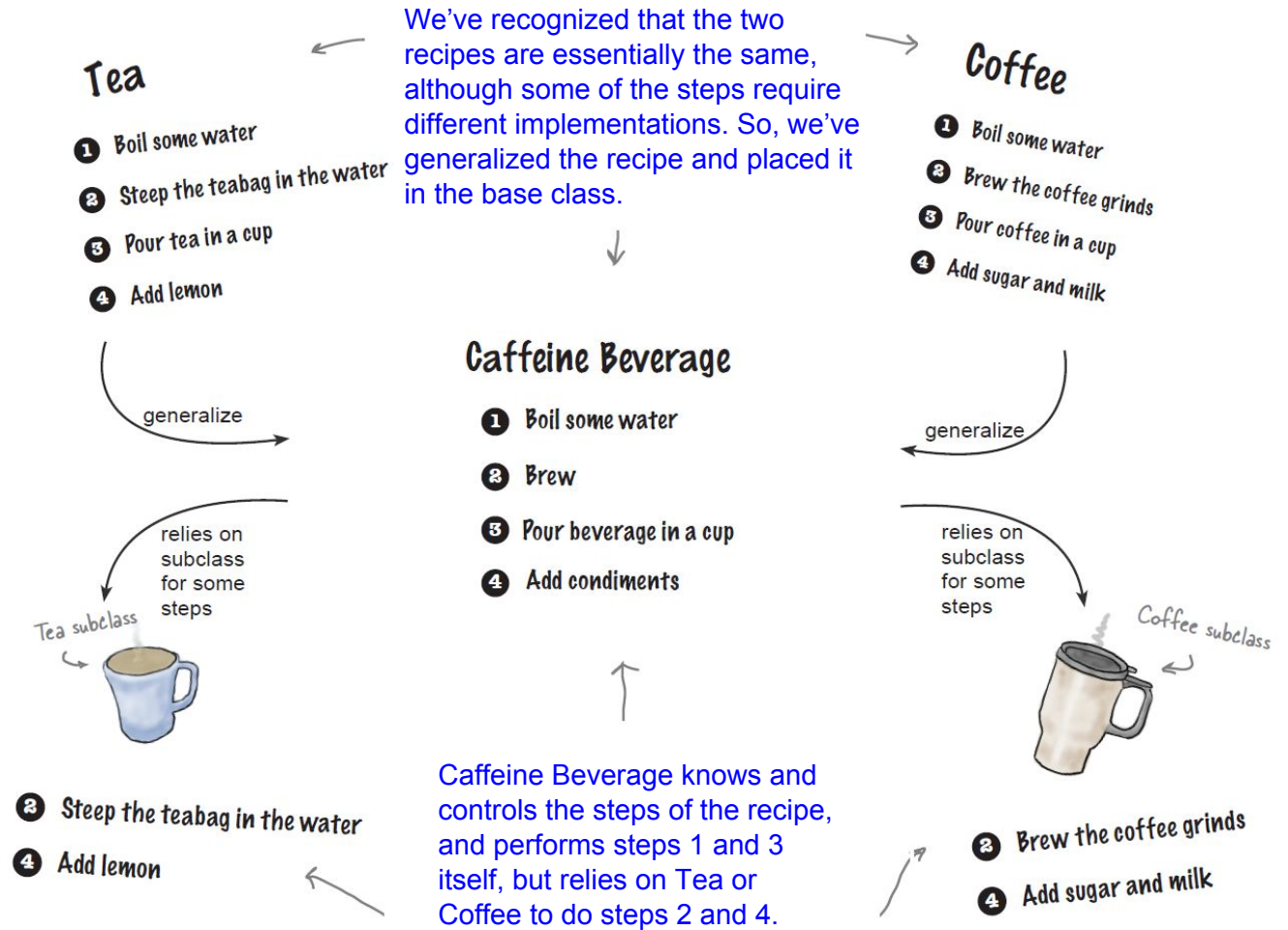
Tea needs to define brew() and addCondiments()—the two abstract methods from CaffeineBeverage.

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

```
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");  
    }  
}
```



What have we done?



# Meet the Template Method

- We've basically just implemented the Template Method Pattern.
- What's that?
  - The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.
- Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method"....

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.

```
public abstract class CaffeineBeverage {
```

```
    void final prepareRecipe() {
```

```
        boilWater();
```

```
        brew();
```

```
        pourInCup();
```

```
        addCondiments();
```

```
    }
```

```
    abstract void brew();
```

```
    abstract void addCondiments();
```

```
    void boilWater() {
```

```
        // implementation
```

```
    }
```

```
    void pourInCup() {
```

```
        // implementation
```

```
    }
```

```
}
```

# Let's make some tea...

- Let's step through making a tea and trace through how the template method works. You'll see that the template method controls the algorithm; at certain points in the algorithm, it lets the subclass supply the implementation of the steps...

- 1 Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

- 2 Then we call the template method:

```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

```
boilWater();  
brew();  
pourInCup();  
addCondiments();
```

The prepareRecipe() method controls the algorithm. No one can change this, and it counts on subclasses to provide some or all of the implementation.

**3**

First we boil water:

```
boilWater();
```

which happens in CaffeineBeverage.

**4**

Next we need to brew the tea, which only the subclass knows how to do:

```
brew();
```

**5**

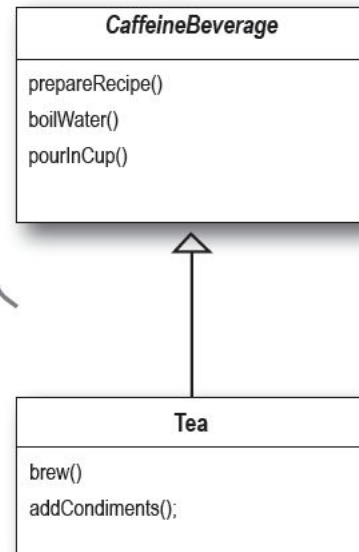
Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

```
pourInCup();
```

**6**

Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addCondiments();
```



# What did the Template Method get us?

## **Underpowered Tea & Coffee implementation**

- Coffee and Tea are running the show; they control the algorithm.
- Code is duplicated across Coffee and Tea.
- Code changes to the algorithm require opening the subclasses and making multiple changes.

## **CaffeineBeverage powered by Template Method**

- The CaffeineBeverage class runs the show; it has the algorithm, and protects it.
- The CaffeineBeverage class maximizes reuse among the subclasses.
- The algorithm lives in one place and code changes only need to be made there.

# What did the Template Method get us? (cont.)

## **Underpowered Tea & Coffee implementation**

- Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.
- Knowledge of the algorithm and how to implement it is distributed over many classes.

## **CaffeineBeverage powered by Template Method**

- The Template Method version provides a framework that other caffeine beverages can be plugged into. New caffeine beverages only need to implement a couple of methods.
- The CaffeineBeverage class concentrates knowledge about the algorithm and relies on subclasses to provide complete implementations.

# Template Method Pattern defined

**The Template Method Pattern** defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

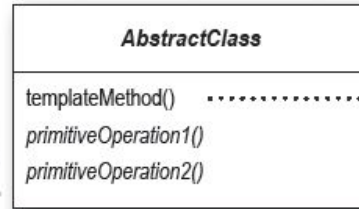
- This pattern is all about creating a template for an algorithm. What's a template?
  - As you've seen it's just a method; more specifically, it's a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass.
  - This ensures the algorithm's structure stays unchanged, while subclasses provide some part of the implementation.



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.



`primitiveOperation1();`  
`primitiveOperation2();`

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.

# Code up close

Here we have our abstract class; it is declared abstract and meant to be subclassed by classes that provide implementations of the operations.



```
abstract class AbstractClass {
```

```
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
    }  
}
```

Here's the template method. It's declared final to prevent subclasses from reworking the sequence of steps in the algorithm.



The template method defines the sequence of steps, each represented by a method.



## Code up close (cont.)

```
abstract void primitiveOperation1();
```

```
abstract void primitiveOperation2();
```

```
void concreteOperation() {  
    // implementation here  
}
```

In this example, two of the primitive operations must be implemented by concrete subclasses.

We also have a concrete operation defined in the abstract class. More about these kinds of methods in a bit...

# Code way up close

```
abstract class AbstractClass {
```

```
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
        hook();  
    }
```


```
    abstract void primitiveOperation1();
```

```
    abstract void primitiveOperation2();
```

We've changed the `templateMethod()` to include a new method call.



We still have our primitive methods; these are abstract and implemented by concrete subclasses.



## Code way up close (cont.)

```
final void concreteOperation() {  
    // implementation here  
}
```

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

```
void hook() {}
```

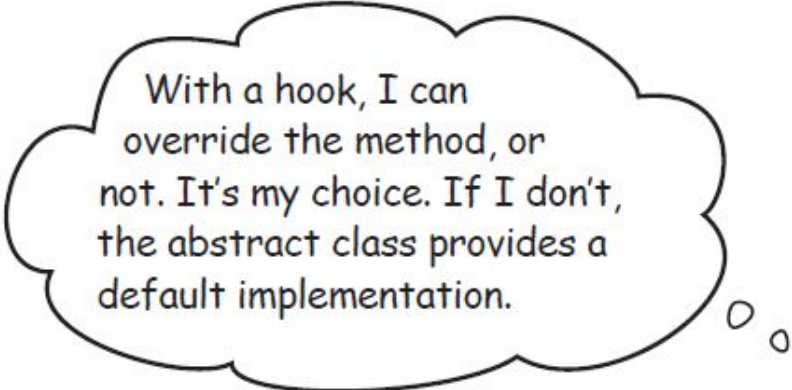
```
}
```

A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

# Hooked on Template Method...

- 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.



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.



```
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");
```

```
    }
```

 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()`.



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

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

- To use the hook, we override it in our subclass.
- Here, the hook controls whether the CaffeineBeverage evaluates a certain part of the algorithm; that is, whether it adds a condiment to the beverage.
- How do we know whether the customer wants the condiment? Just ask!

```
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;
```

```
}
```

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.

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

# 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!

# Let's run the Test Drive (cont.)

```
File Edit Window Help send-more-honesttea
```

```
%java BeverageTestDrive
```

```
Making tea...
```

```
Boiling water
```

```
Steeping the tea
```

```
Pouring into cup
```

```
Would you like lemon with your tea (y/n)? y
```

```
Adding Lemon
```

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

```
Making coffee...
```

```
Boiling water
```

```
Dripping Coffee through filter
```

```
Pouring into cup
```

```
Would you like milk and sugar with your coffee (y/n)? n
```

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

```
%
```

# Some questions

Q: When I'm creating a template method, how do I know when to use abstract methods and when to use hooks?

- Use abstract methods when your subclass **MUST** provide an implementation of the method or step in the algorithm.
- Use hooks when that part of the algorithm is optional.
- With hooks, a subclass may choose to implement that hook, but it doesn't have to.

## Some questions (cont.)

Q: What are hooks really supposed to be used for?

- There are a few uses of hooks. As we just said, a hook may provide a way for a subclass to implement an optional part of an algorithm, or if it isn't important to the subclass's implementation, it can skip it.
- Another use is to give the subclass a chance to react to some step in the template method that is about to happen, or just happened.
  - For instance, a hook method like `justReOrderedList()` allows the subclass to perform some activity (such as redisplaying an onscreen representation) after an internal list is reordered. As you've seen, a hook can also provide a subclass with the ability to make a decision for the abstract class.

## Some questions (cont.)

Q: Does a subclass have to implement all the abstract methods in the `AbstractClass`?

- Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

# The Hollywood Principle

- We've got another design principle for you; it's called the Hollywood Principle:

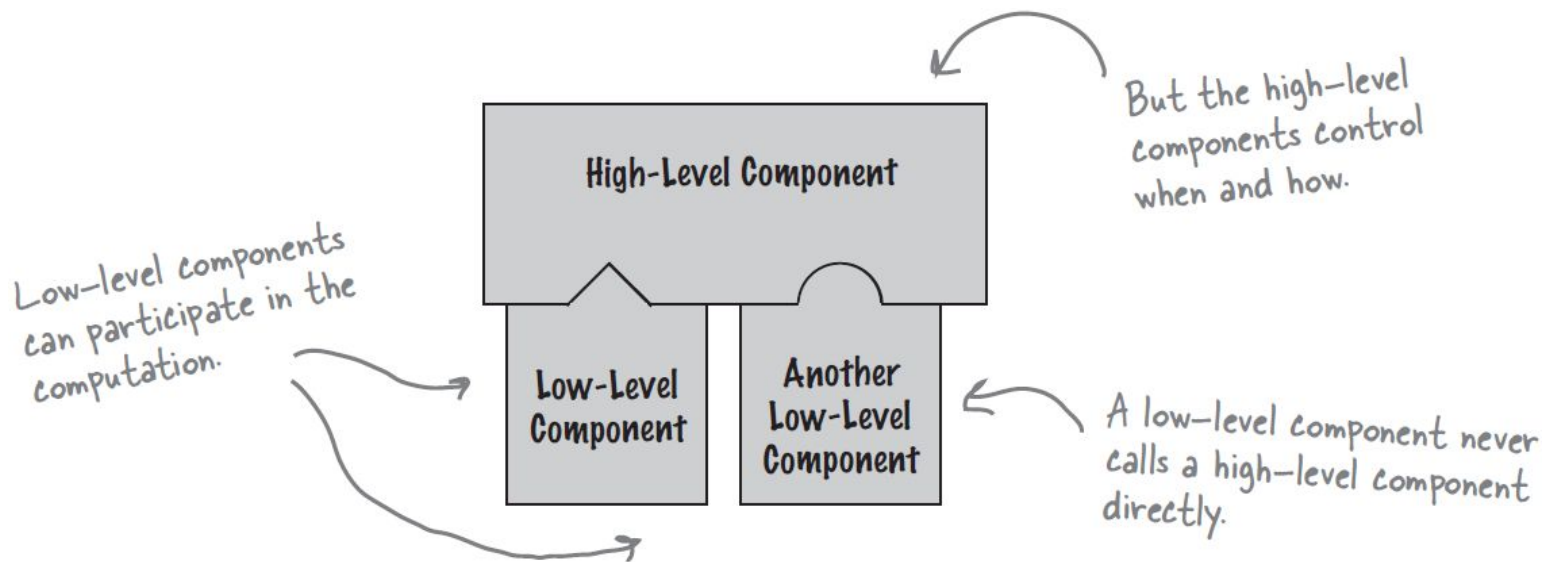
**Design principle: Don't call us, we'll call you.**

- The Hollywood Principle gives us a way to prevent “dependency rot.”
- Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on.
- When rot sets in, no one can easily understand the way a system is designed.



# The Hollywood Principle (cont.)

- With the principle, we allow low-level components to hook themselves into a system, but the high-level components determine when they are needed, and how. In other words, the high-level components give the low-level components a “don’t call us, we’ll call you” treatment.

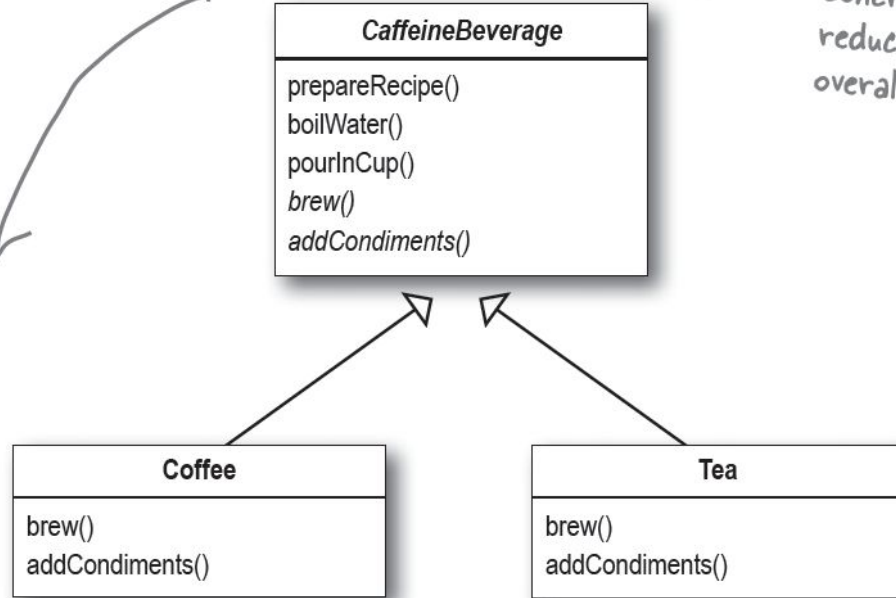


# The Hollywood Principle and Template Method

- The connection between the Hollywood Principle and the Template Method Pattern is probably somewhat apparent: when we design with the Template Method Pattern, we're telling subclasses, "don't call us, we'll call you." How?
- Let's take another look at our CaffeineBeverage design...

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.



The subclasses are used simply to provide implementation details.

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

# A question

Q. What other patterns make use of the Hollywood Principle?

- The Observer Pattern
- The Factory Method Pattern

# Who does what?

- Match each pattern with its description:

<b>Pattern</b>	<b>Description</b>
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Factory Method	Subclasses decide which concrete classes to instantiate.

# Who does what?

- Match each pattern with its description:

Pattern	Description
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use.
Strategy	Subclasses decide how to implement steps in an algorithm.
Factory Method	Subclasses decide which concrete classes to instantiate.

# Template Methods in the Wild

- The Template Method Pattern is a very common pattern and you're going to find lots of it in the wild.
- You've got to have a keen eye, though, because there are many implementations of the template methods that don't quite look like the textbook design of the pattern.
- This pattern shows up so often because it's a great design tool for creating frameworks, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.

# Sorting with Template Method

- The designers of the Java Arrays class have provided us with a handy template method for sorting. Let's take a look at how this method operates:
  - 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
}
```

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

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

# We've got some ducks to sort...

- Let's say you have an array of ducks that you'd like to sort. How do you do it?
- Well, 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.

# We've got some ducks to sort... (cont.)



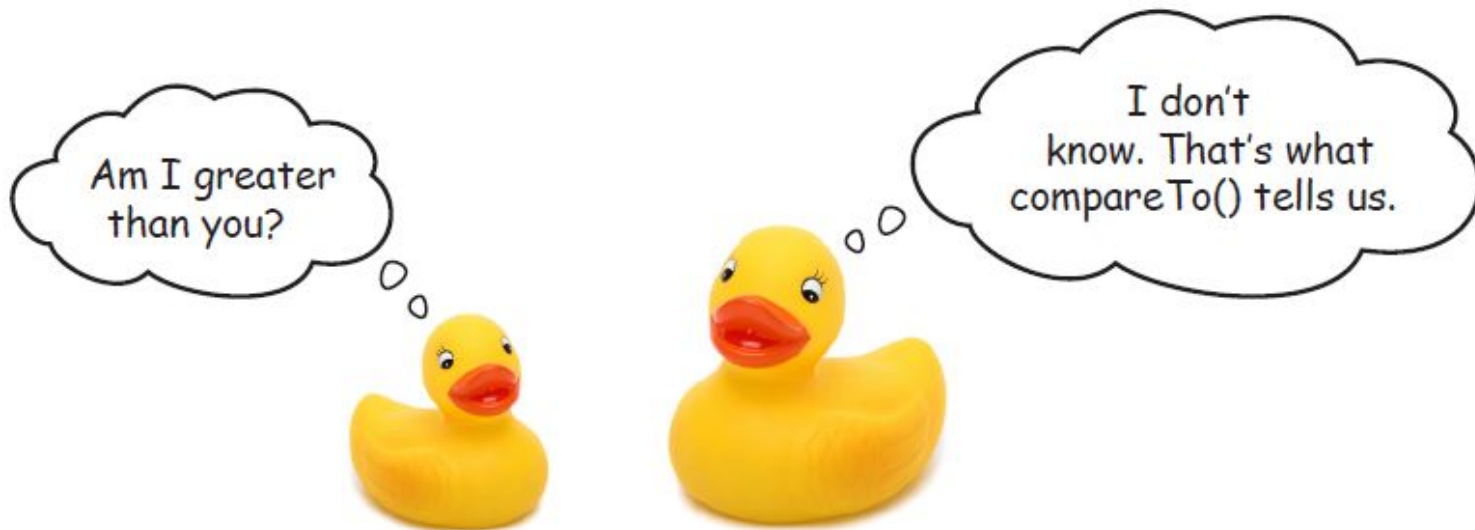
No, it doesn't.  
Aren't we supposed to be subclassing something? I thought that was the point of Template Method. An array doesn't subclass anything, so I don't get how we'd use `sort()`.

- Good point. Here's the deal: the designers of `sort()` wanted it to be useful across all arrays, so they had to make `sort()` a static method that could be used from anywhere.

- But that's okay, it works almost the same as if it were in a superclass.
- Now, here is one more detail: because `sort()` really isn't defined in our superclass, the `sort()` method needs to know that you've implemented the `compareTo()` method, or else you don't have the piece needed to complete the sort algorithm.
- To handle this, the designers made use of the `Comparable` interface. All you have to do is implement this interface, which has one method (surprise): `compareTo()`.

# What is compareTo()?

- The compareTo() method compares two objects and returns whether one is less than, greater than, or equal to the other. sort() uses this as the basis of its comparison of objects in the array.



# Comparing Ducks and Ducks

- Okay, so you know that if you want to sort Ducks, you're going to have to implement this `compareTo()` method; by doing that you'll give the `Arrays` class what it needs to complete the algorithm and sort your ducks.
- Here's the duck implementation...

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

```
public class Duck implements Comparable {
```

```
    String name;
```

```
    int weight;
```

Our Ducks have a name and a weight

```
    public Duck(String name, int weight) {
```

```
        this.name = name;
```

```
        this.weight = weight;
```

```
    }
```

```
    public String toString() {
```

```
        return name + " weighs " + weight;
```

```
    }
```

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

Okay, here's what sort needs...

```
public int compareTo(Object object) {
```

```
    Duck otherDuck = (Duck) object;
```

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

```
    if (this.weight < otherDuck.weight) {
```

```
        return -1;
```

```
    } else if (this.weight == otherDuck.weight) {
```

```
        return 0;
```

```
    } else { // this.weight > otherDuck.weight
```

```
        return 1;
```

```
    }
```

```
}
```

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.

# Let's sort some Ducks

- Here's the test drive for sorting Ducks...



```
public class DuckSortTestDrive {
```

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

```
        Duck[] ducks = {
```

```
            new Duck("Daffy", 8),  
            new Duck("Dewey", 2),  
            new Duck("Howard", 7),  
            new Duck("Louie", 2),  
            new Duck("Donald", 10),  
            new Duck("Huey", 2)
```

```
        };
```

```
        System.out.println("Before sorting:");  
        display(ducks);
```

```
        Arrays.sort(ducks);
```

```
        System.out.println("\nAfter sorting:");  
        display(ducks);
```

```
    }
```

```
    public static void display(Duck[] ducks) {
```

```
        for (Duck d : ducks) {
```

```
            System.out.println(d);
```

```
        }
```

```
    }
```

```
}
```

We need an array of Ducks; these look good.

Let's print them to see their names and weights.

It's sort time!

Let's print them (again) to see their names and weights.

```
File Edit Window Help DonaldNeedsToGoOnADiet
%java DuckSortTestDrive
Before sorting:
Daffy weighs 8
Dewey weighs 2
Howard weighs 7
Louie weighs 2
Donald weighs 10
Huey weighs 2
The unsorted Ducks

After sorting:
Dewey weighs 2
Louie weighs 2
Huey weighs 2
Howard weighs 7
Daffy weighs 8
Donald weighs 10
%
The sorted Ducks
```

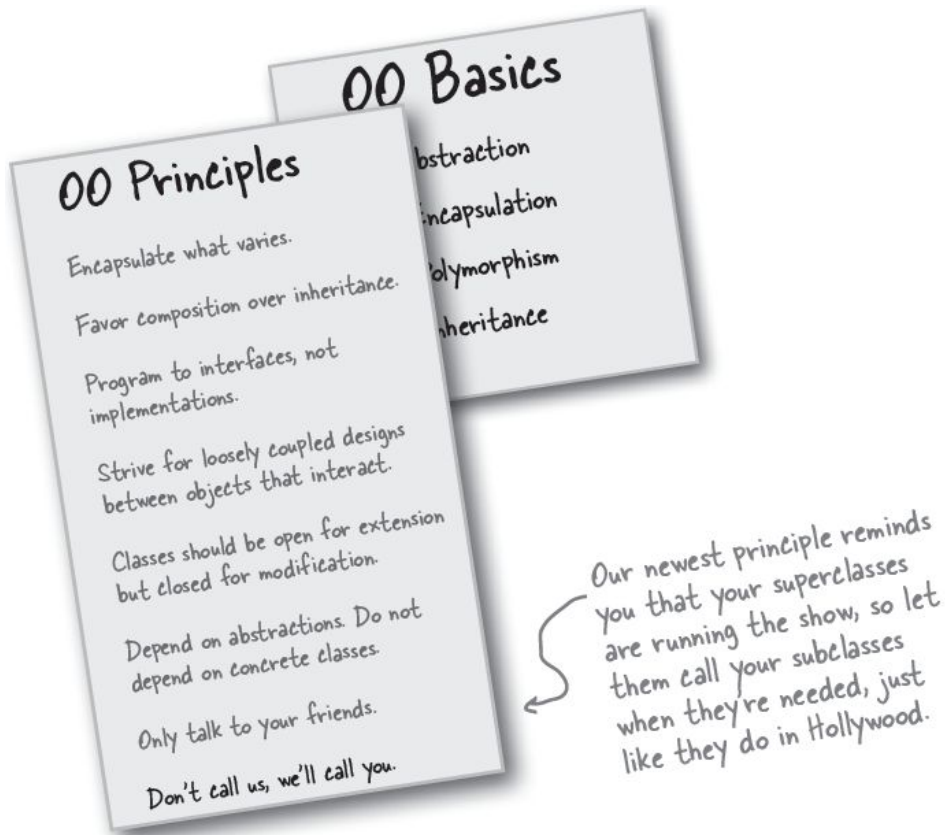
# Recommendation

- Read the “Swingin’ with Frames” and “Applets” examples from the book and make sure you study their codes.

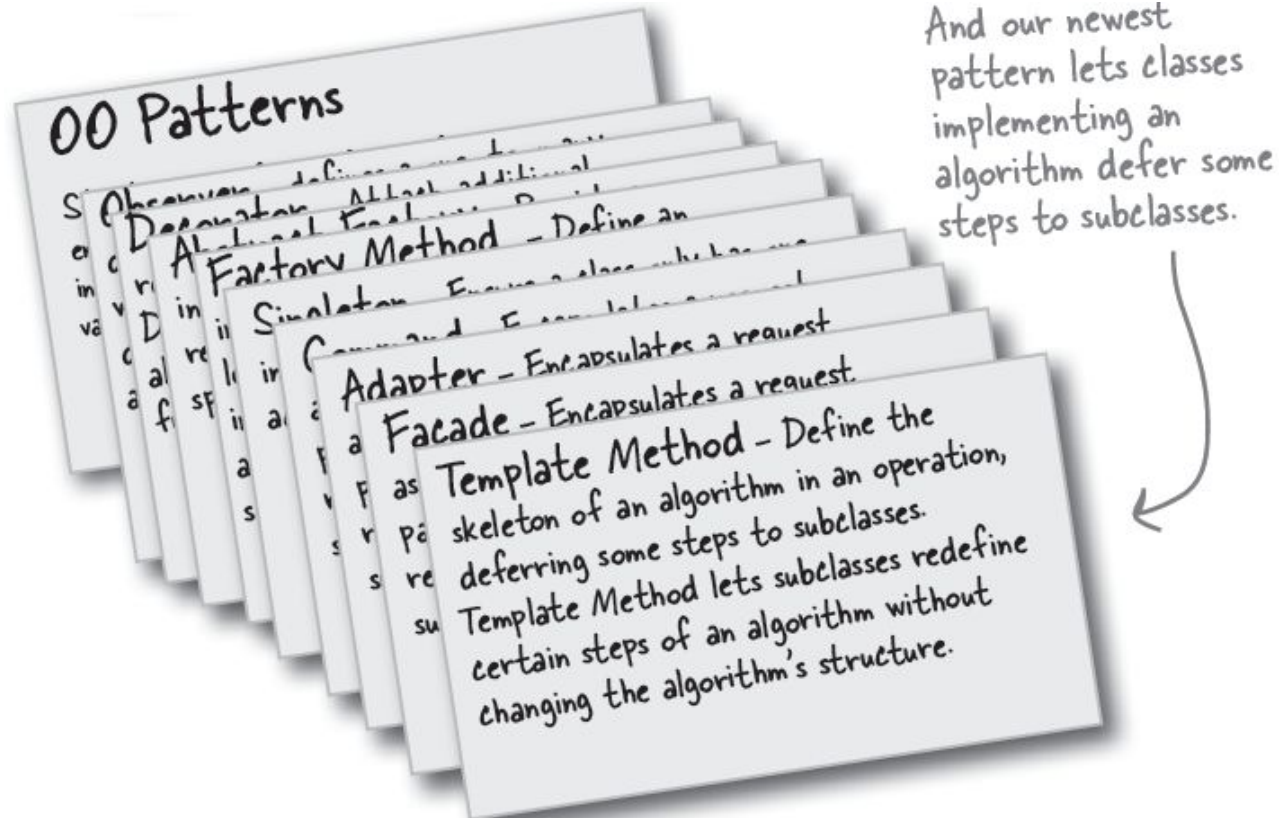
# Some Bullet Points

- A “template method” defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.
- The Template Method Pattern gives us an important technique for code reuse.
- The Template Method’s abstract class may define concrete methods, abstract methods, and hooks.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- The Hollywood Principle guides us to put decision making in high-level modules that can decide how and when to call low-level modules.
- The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.
- The Factory Method is a specialization of Template Method.

# Tools for your Design Toolbox



# Tools for your Design Toolbox (cont.)



# References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2014.