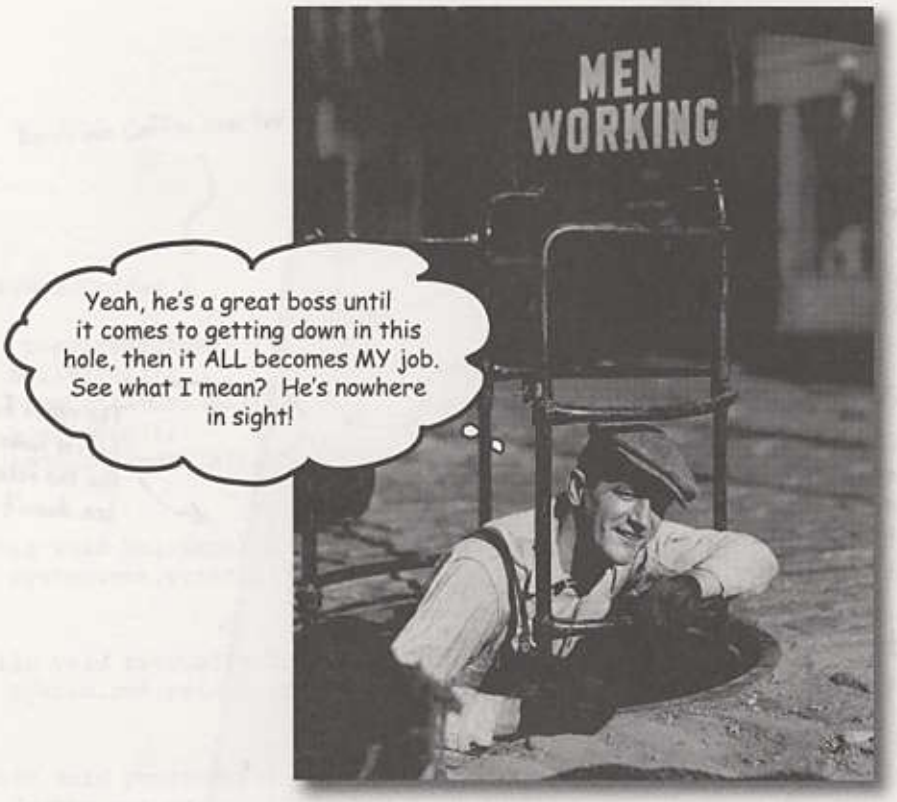
finished 1st round this whole book on 14 - July - 09 3:49 AM

This "Template Method" design pattern used in Spring JDBC Module.

8 the Template Method Pattern

Encapsulating Algorithms

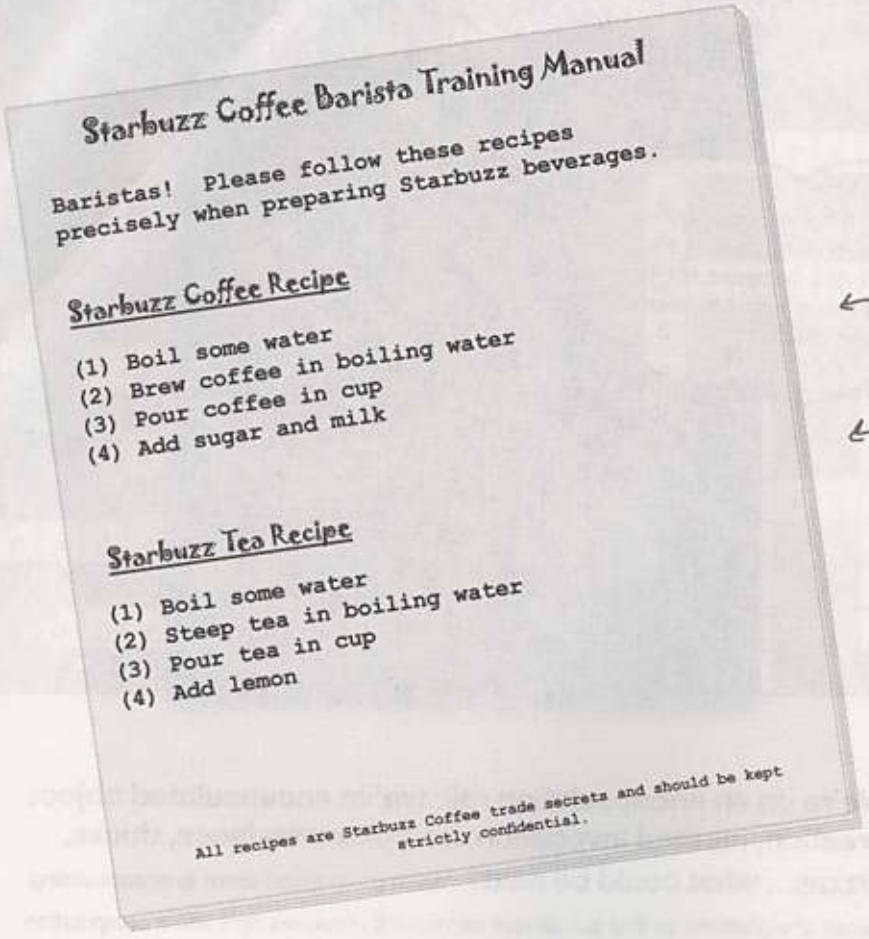


We're on an encapsulation roll; we've encapsulated object creation, method invocation, complex interfaces, ducks, pizzas... what could be next? We're going to get down to encapsulating pieces of algorithms so that subclasses can hook themselves right into a computation anytime they want. We're even going 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:



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

Whipping up some coffee and tea classes (in Java)

Let's play "coding barista" and write some code for creating coffee and tea.

Here's the coffee:

Here's our Coffee class for making coffee.

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

Each of the steps is implemented as a separate method.

```
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");
    }
}
```

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 forth 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.

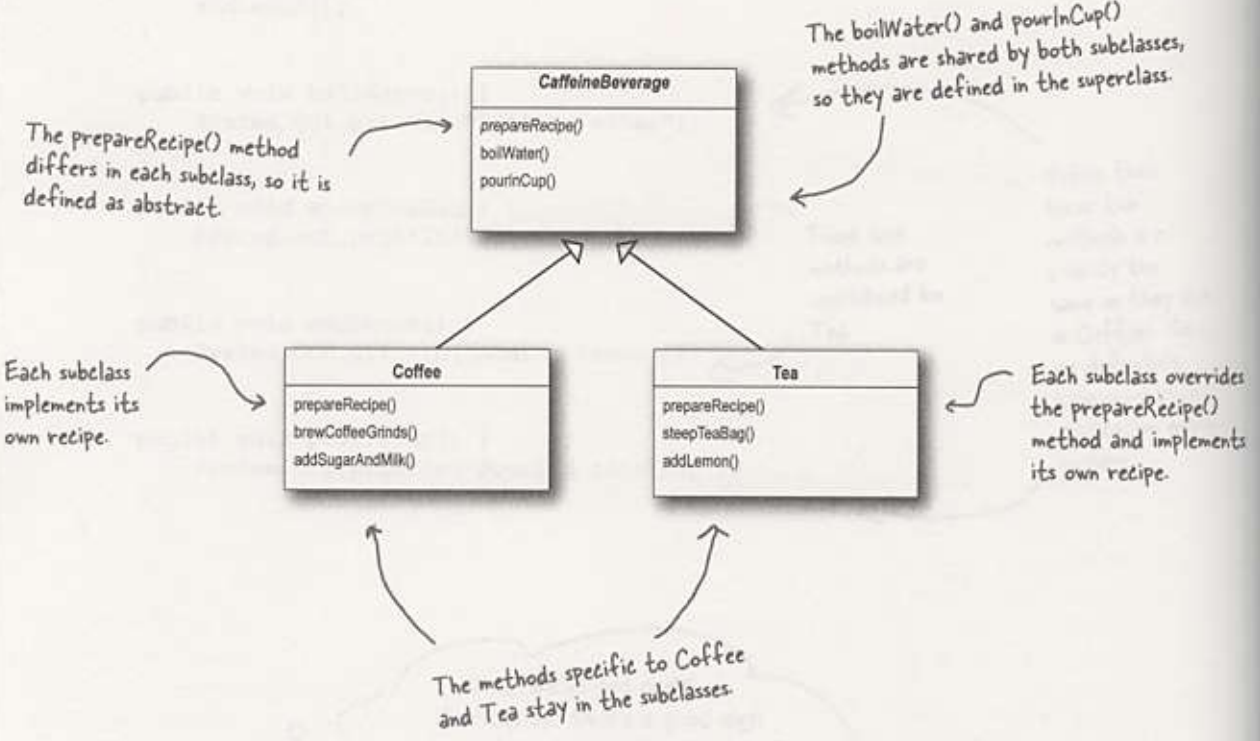
When we've got code duplication, that's a good sign we need to clean up the design. It seems like here we should abstract the commonality into a base class since coffee and tea are so similar?

Design Puzzle

You've seen that the Coffee and Tea classes have a fair bit of code duplication. Take another look at the Coffee and Tea classes and draw a class diagram showing how you'd redesign the classes to remove redundancy:

Sir, may I abstract your Coffee, Tea?

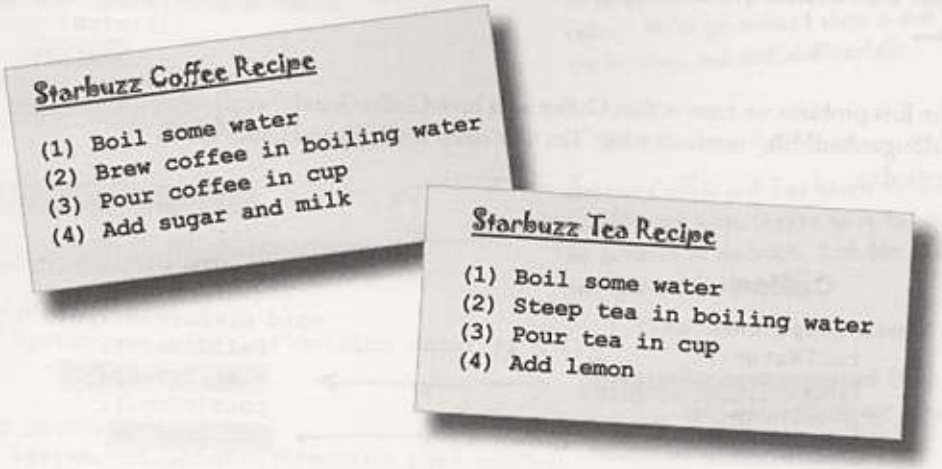
It looks like we've got a pretty straightforward design exercise on our hands with the Coffee and Tea classes. Your first cut might have looked something like this:



Did we do a good job on the redesign? Hmmmm, take another look. Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

Taking the design further...

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



Notice that both recipes follow the same algorithm:

1. Boil some water.
2. Use the hot water to extract the coffee or tea.
3. Pour the resulting beverage into a cup.
4. Add the appropriate condiments to the beverage.

These aren't abstracted, but are the same, they just apply to different beverages.

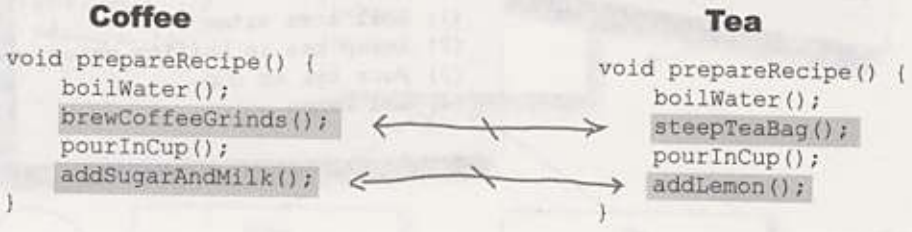
These two are already abstracted into the base class.

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

Abstracting prepareRecipe()

Let's step through abstracting prepareRecipe() from each subclass (that is, the Coffee and Tea classes)...

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



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

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

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).

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:

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

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

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

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");
    }
}
```

Sharpen your pencil

Draw the new class diagram now that we've moved the implementation of prepareRecipe() into the CaffeineBeverage class:

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

Caffeine Beverage

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

generalize

generalize

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

Coffee subclass

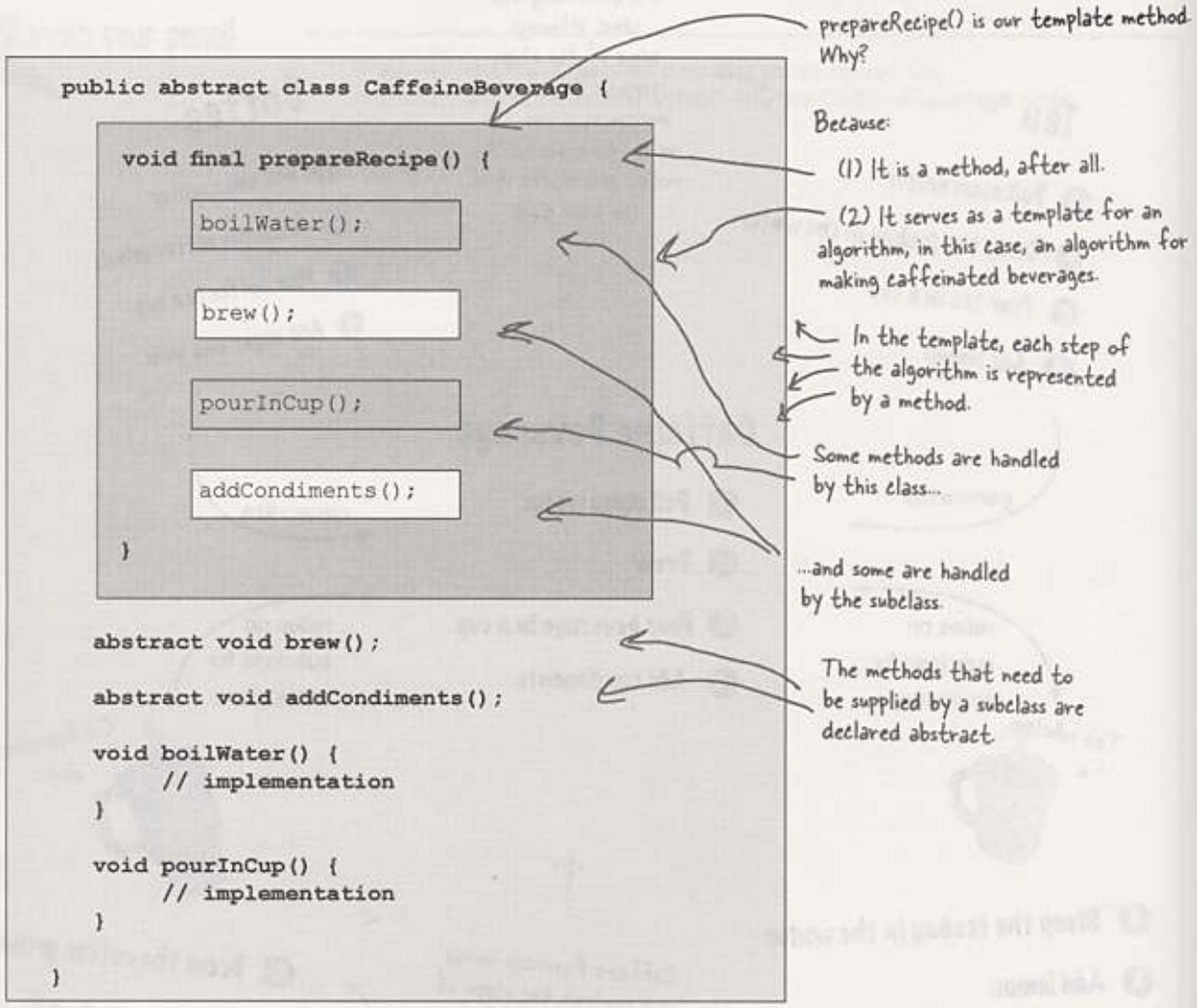
2. Steep the teabag in the water
4. Add lemon

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

We've basically just implemented the Template Method Pattern. What's that? Let's look at the structure of the CaffeineBeverage class; it contains the actual "template method:"



The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Let's make some tea...

Behind the Scenes

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...

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:

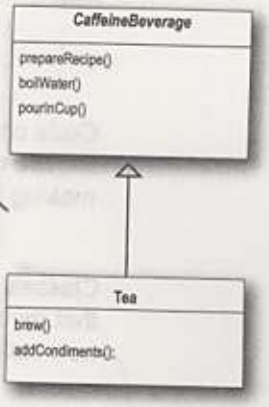
```
pourInCup();
```

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

```
addCondiments();
```

```
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.



What did the Template Method get us?

Underpowered Tea & Coffee implementation

New, hip CaffeineBeverage powered by Template Method

Coffee and Tea are running the show; they control the algorithm.	The CaffeineBeverage class runs the show; it has the algorithm, and protects it.
Code is duplicated across Coffee and Tea.	The CaffeineBeverage class maximizes reuse among the subclasses.
Code changes to the algorithm require opening the subclasses and making multiple changes.	The algorithm lives in one place and code changes only need to be made there.
Classes are organized in a structure that requires a lot of work to add a new caffeine beverage.	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.
Knowledge of the algorithm and how to implement it is distributed over many classes.	The CaffeineBeverage class concentrates knowledge about the algorithm and relies on subclasses to provide complete implementations.

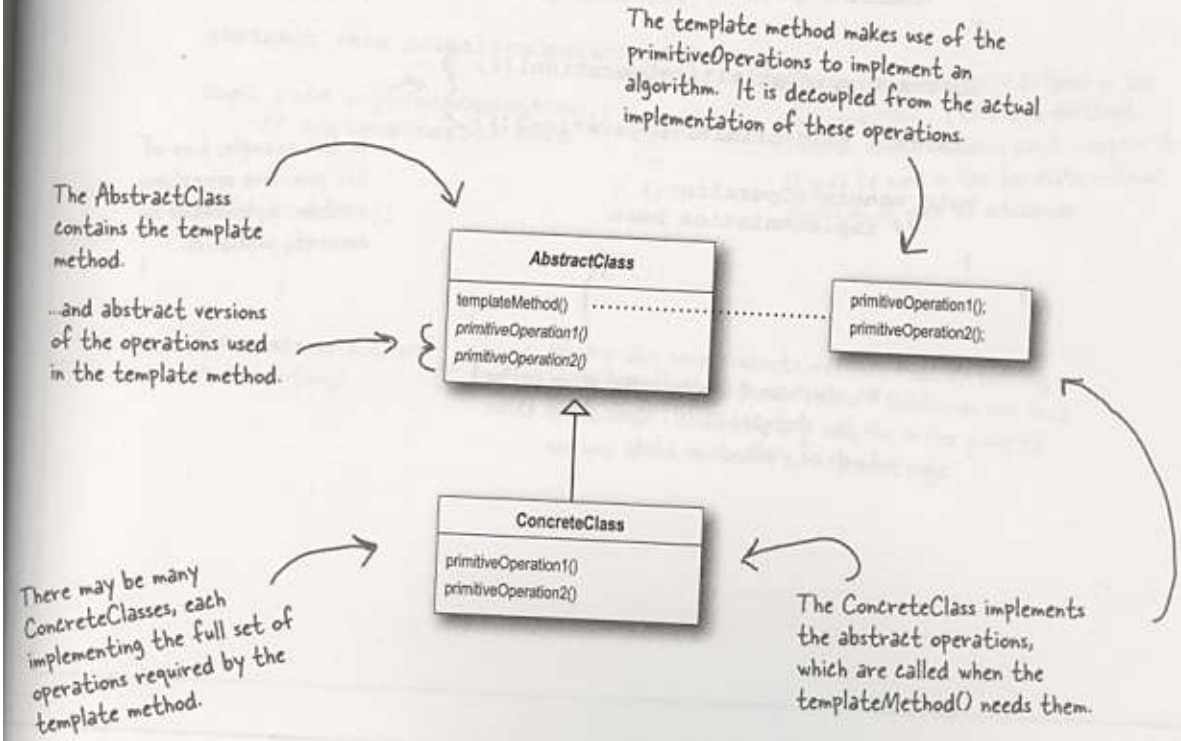
Template Method Pattern defined

You've seen how the Template Method Pattern works in our Tea and Coffee example; now, check out the official definition and nail down all the details:

> **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.

Let's check out the class diagram:



Code Up Close

Let's take a closer look at how the AbstractClass is defined, including the template method and primitive operations.

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

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

```
abstract class AbstractClass {

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

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

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

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

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

Now we're going to look even closer at the types of method that can go in the abstract class:

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

```
abstract class AbstractClass {

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

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

abstract class can have final methods (i tested)

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

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.

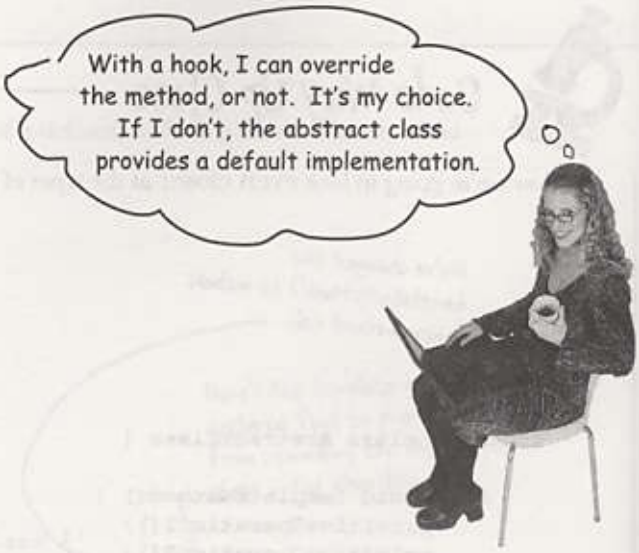
A concrete method, but it does nothing!

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.

There are several uses of hooks; let's take a look at one now. We'll talk about a few other uses later:




```
public abstract class CaffeineBeverageWithHook {

    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 a 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.

hook method has to be defined or declared in super class itself

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;
    }
}
```

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 TestDrive

Okay, the water's boiling... Here's the test code where we create a hot tea and a hot coffee

```
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

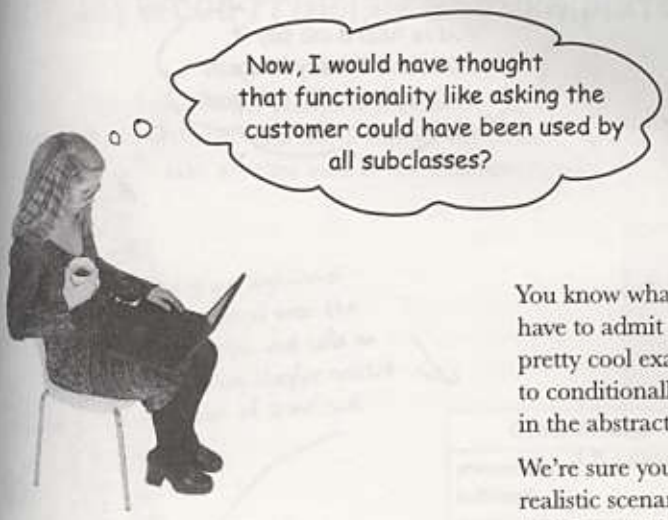
And call prepareRecipe() on both!

And let's give it a run...

```
File Edit Window Help send-more-honestlea
%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.



You know what? We agree with you. But you have to admit before you thought of that it was a pretty cool example of how a hook can be used to conditionally control the flow of the algorithm in the abstract class. Right?

We're sure you can think of many other more realistic scenarios where you could use the template method and hooks in your own code.

there are no Dumb Questions

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

A: 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.

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

A: 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' 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.

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

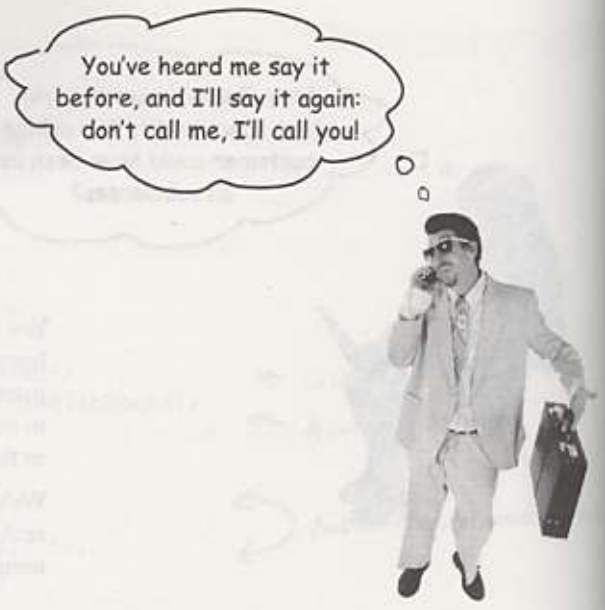
A: Yes, each concrete subclass defines the entire set of abstract methods and provides a complete implementation of the undefined steps of the template method's algorithm.

Q: It seems like I should keep my abstract methods small in number, otherwise it will be a big job to implement them in the subclass.

A: That's a good thing to keep in mind when you write template methods. Sometimes this can be done by not making the steps of your algorithm too granular. But it's obviously a trade off: the less granularity, the less flexibility.

Remember, too, that some steps will be optional; so you can implement these as hooks rather than abstract classes, easing the burden on the subclasses of your abstract class.

The Hollywood Principle



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

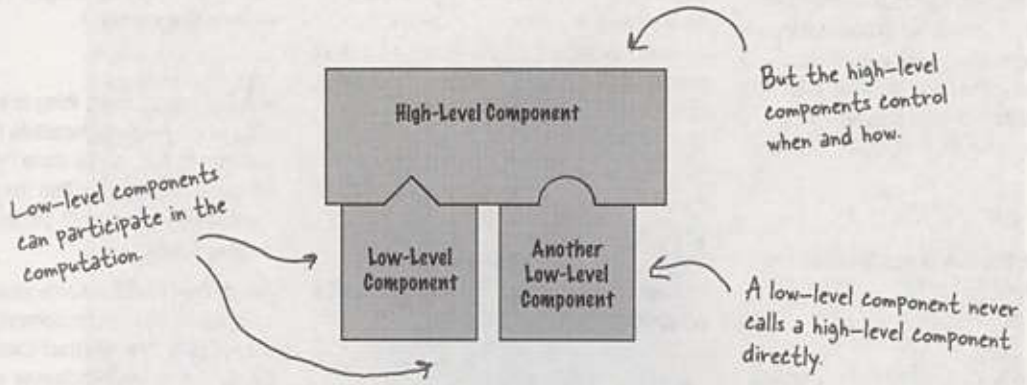
The Hollywood Principle

Don't call us, we'll call you.

Easy to remember, right? But what has it got to do with OO design?

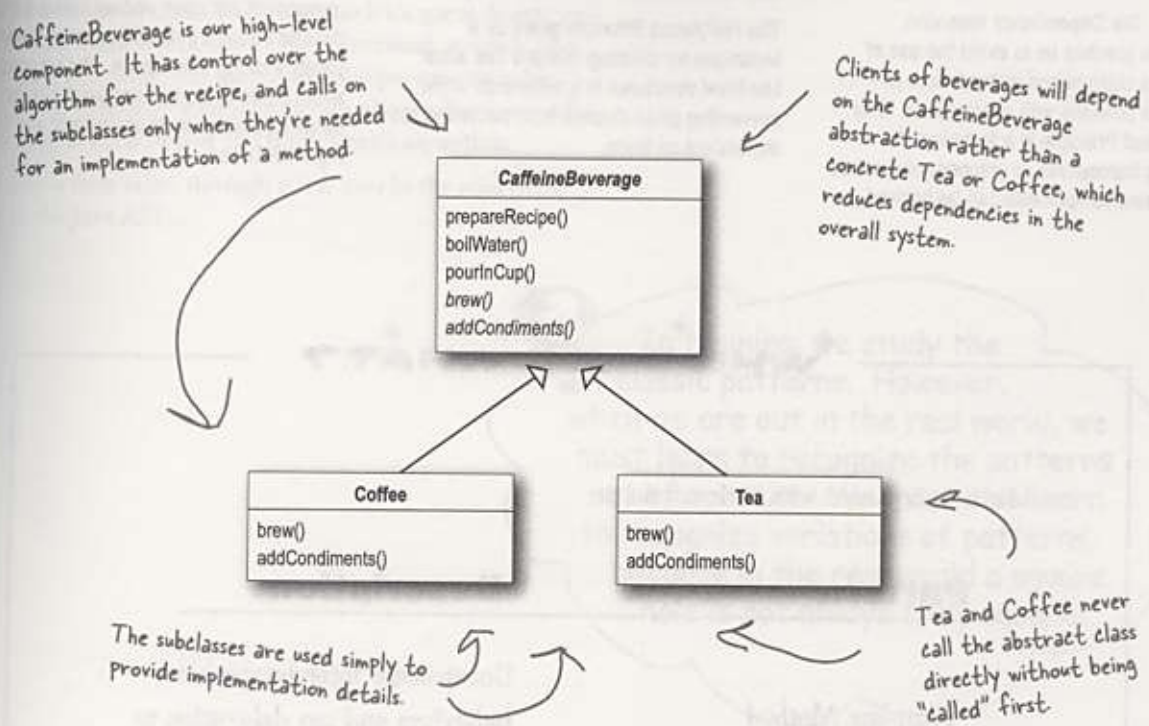
The Hollywood principle gives us a way to prevent "dependency rot." Dependency rot happens when you have high-level components depending on low-level components depending on high-level components depending on sideways components depending on low-level components, and so on. When rot sets in, no one can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

With the Hollywood 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:



BRAIN POWER

What other patterns make use of the Hollywood Principle?

The Factory Method, Observer, any others?

there are no Dumb Questions

Q: How does the Hollywood Principle relate to the Dependency Inversion Principle that we learned a few chapters back?

A: The Dependency Inversion Principle teaches us to avoid the use of concrete classes and instead work as much as possible with abstractions. The Hollywood Principle is a technique for building frameworks or components so that lower-level components can be hooked into the computation, but without creating dependencies between the lower-level components and the higher-level layers. So, they both have the goal of decoupling, but the Dependency Inversion Principle makes a much stronger and general statement about how to avoid dependencies in design.

The Hollywood Principle gives us a technique for creating designs that allow low-level structures to interoperate while preventing other classes from becoming too dependent on them.

Q: Is a low-level component disallowed from calling a method in a higher-level component?

A: Not really. In fact, a low level component will often end up calling a method defined above it in the inheritance hierarchy purely through inheritance. But we want to avoid creating explicit circular dependencies between the low-level component and the high-level ones.

WHO DOES WHAT?

Match each pattern with its description:

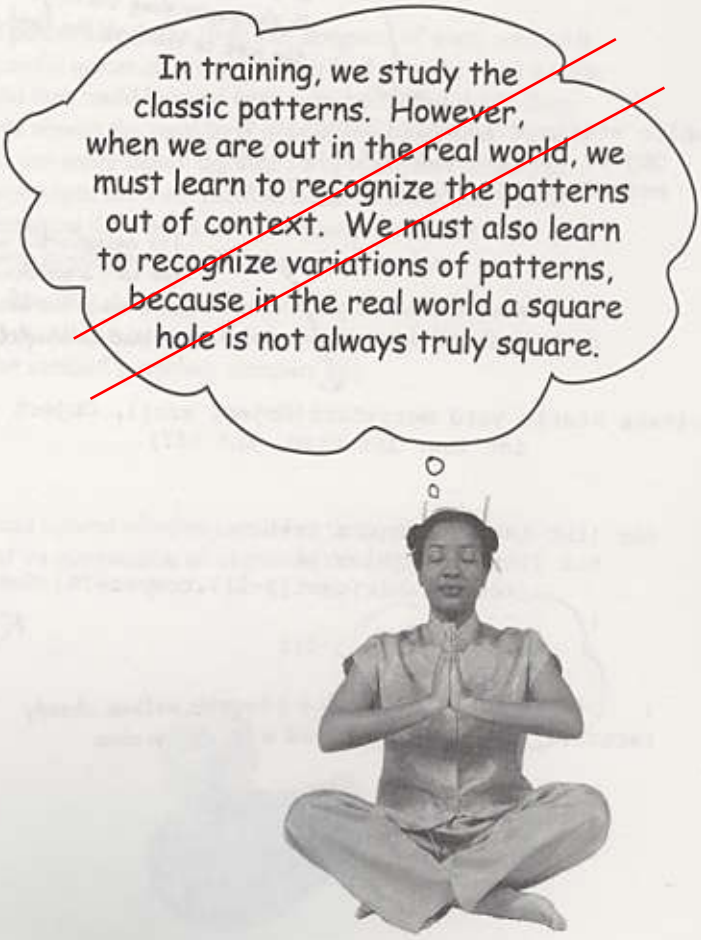
Pattern	Description
Template Method	Encapsulate interchangeable behaviors and use delegation to decide which behavior to use
Strategy	Subclasses decide how to implement steps in an algorithm
Factory Method	Subclasses decide which concrete classes to create

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.

Let's take a little safari through a few uses in the wild (well, okay, in the Java API)...



Sorting with Template Method

What's something we often need to do with arrays? Sort them!

Recognizing that, 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've pared down this code a little to make it easier to explain. If you'd like to see it all, grab the source from Sun and check it out...

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.

Think of this as the template method.

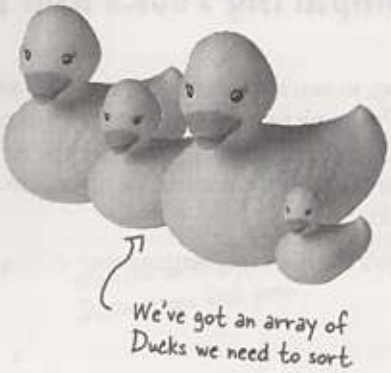
```
private static void mergeSort(Object src[], Object dest[],
                              int low, int high, int off)
{

    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);
        }
    }
    return;
}
```

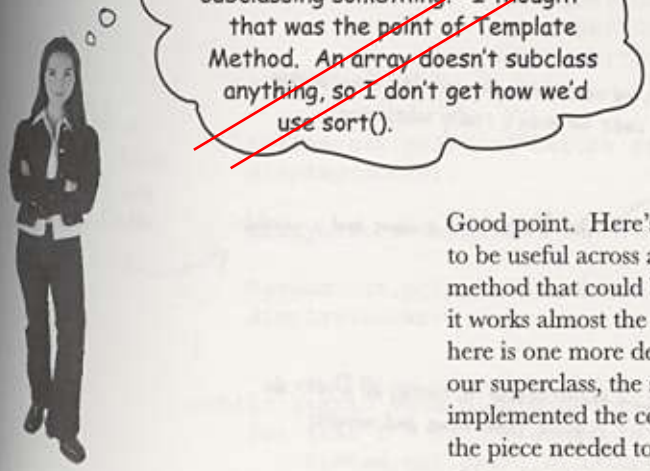
This is a concrete method, already defined in the Arrays class.

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?



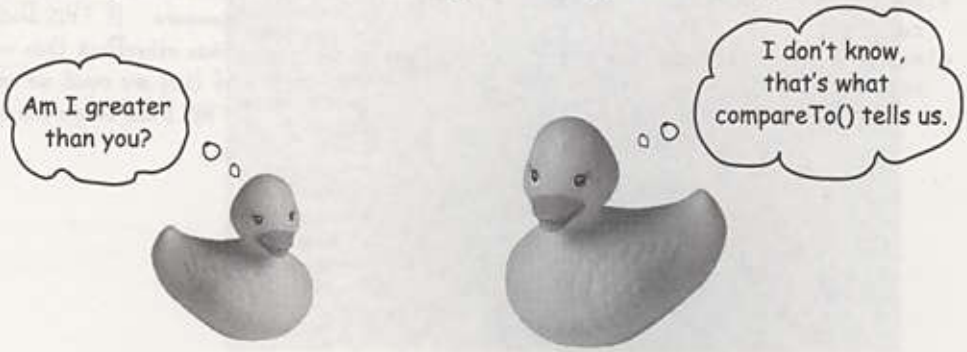


Good point. Here's the deal: the designers of sort() wanted it to be useful across all arrays, so they had to make sort() a static method that could be used from anywhere. But that's okay, it works almost the same as if it were in a superclass. Now, here is one more detail: because sort() really isn't 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.

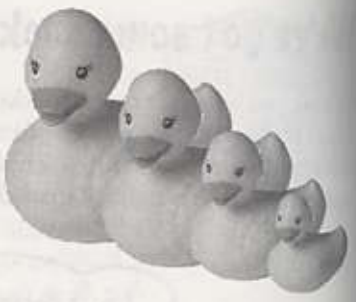
To handle this, the designers made use of the Comparable interface. All you have to do is implement this interface, which has one method (surprise): compareTo().

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;

    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;
        }
    }
}
```

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.

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 (int i = 0; i < ducks.length; i++) {
            System.out.println(ducks[i]);
        }
    }
}
```

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.

Notice that we call Arrays' static method sort, and pass it our Ducks.

Let the sorting commence!

```
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

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

The unsorted Ducks

The sorted Ducks

The making of the sorting duck machine

Let's trace through how the Arrays sort() template method works. We'll check out how the template method controls the algorithm, and at certain points in the algorithm, how it asks our Ducks to supply the implementation of a step...

```
for (int i=low; i<high; i++){
        ... compareTo() ...
        ... swap() ...
}
```

The sort() method controls the algorithm, no class can change this. sort() counts on a Comparable class to provide the implementation of compareTo()

1. First, we need an array of Ducks:

```
Duck[] ducks = {new Duck("Daffy", 8), ... };
```

2. Then we call the sort() template method in the Array class and pass it our ducks:

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

The sort() method (and its helper mergeSort()) control the sort procedure.

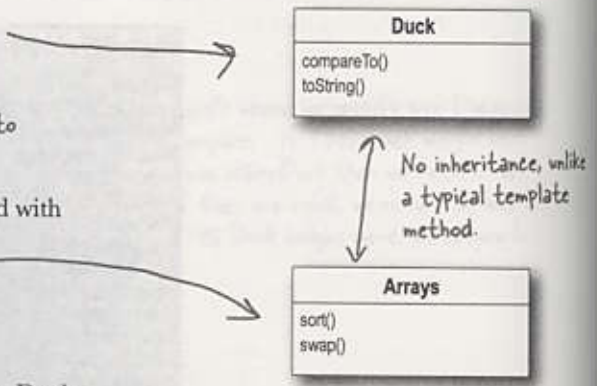
3. To sort an array, you need to compare two items one by one until the entire list is in sorted order.

When it comes to comparing two ducks, the sort method relies on the Duck's compareTo() method to know how to do this. The compareTo() method is called on the first duck and passed the duck to be compared to:

```
ducks[0].compareTo(ducks[1]);
```

First Duck

Duck to compare it to



4. If the Ducks are not in sorted order, they're swapped with the concrete swap() method in Arrays:

```
swap()
```

5. The sort method continues comparing and swapping Ducks until the array is in the correct order!

there are no Dumb Questions

Q: Is this really the Template Method Pattern, or are you trying too hard?

A: The pattern calls for implementing an algorithm and letting subclasses supply the implementation of the steps – and the Arrays sort is clearly not doing that! But, as we know, patterns in the wild aren't always just like the textbook patterns. They have to be modified to fit the context and implementation constraints.

The designers of the Arrays sort() method had a few constraints. In general, you can't subclass a Java array and they wanted the sort to be used on all arrays (and each array is a different class). So they defined a static method and deferred the comparison part of the algorithm to the items being sorted.

So, while it's not a textbook template method, this implementation is still in the spirit of the Template Method Pattern. Also, by eliminating the requirement that you have to subclass Arrays to use this algorithm, they've made sorting in some ways more flexible and useful.

Q: This implementation of sorting actually seems more like the Strategy Pattern than the Template Method Pattern. Why do we consider it Template Method?

A: You're probably thinking that because the Strategy Pattern uses object composition. You're right in a way – we're *using* the Arrays object to sort our array, so that's similar to Strategy. But remember, in Strategy, the class that you compose with implements the *entire* algorithm. The algorithm that Arrays implements for sort is incomplete; it needs a class to fill in the missing compareTo() method. So, in that way, it's more like Template Method.

Q: Are there other examples of template methods in the Java API?

A: Yes, you'll find them in a few places. For example, java.io has a read() method in InputStream that subclasses must implement and is used by the tempate method read(byte b[], int off, int len).

BRAIN POWER

We know that we should favor composition over inheritance, right? Well, the implementers of the sort() template method decided not to use inheritance and instead to implement sort() as a static method that is composed with a Comparable at runtime. How is this better? How is it worse? How would you approach this problem? Do Java arrays make this particularly tricky?

BRAIN² POWER

Think of another pattern that is a specialization of the template method. In this specialization, primitive operations are used to create and return objects. What pattern is this?

Swingin' with Frames

Up next on our Template Method safari... keep your eye out for swinging JFrames!

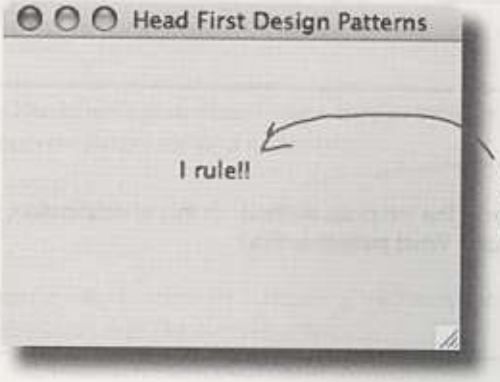
If you haven't encountered JFrame, it's the most basic Swing container and inherits a paint() method. By default, paint() does nothing because it's a *hook*! By overriding paint(), you can insert yourself into JFrame's algorithm for displaying its area of the screen and have your own graphic output incorporated into the JFrame. Here's an embarrassingly simple example of using a JFrame to override the paint() hook method:

We're extending JFrame, which contains a method update() that controls the algorithm for updating the screen. We can hook into that algorithm by overriding the paint() hook method.

```
public class MyFrame extends JFrame {

    public MyFrame(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "I rule!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MyFrame myFrame = new MyFrame("Head First Design Patterns");
    }
}
```

Don't look behind the curtain! Just some initialization here...

JFrame's update algorithm calls paint(). By default, paint() does nothing... it's a hook. We're overriding paint(), and telling the JFrame to draw a message in the window.



Here's the message that gets painted in the frame because we've hooked into the paint() method.

Applets

Our final stop on the safari: the applet.

You probably know an applet is a small program that runs in a web page. Any applet must subclass Applet, and this class provides several hooks. Let's take a look at a few of them:

```
public class MyApplet extends Applet {
    String message;

    public void init() {
        message = "Hello World, I'm alive!";
        repaint();
    }

    public void start() {
        message = "Now I'm starting up...";
        repaint();
    }

    public void stop() {
        message = "Oh, now I'm being stopped...";
        repaint();
    }

    public void destroy() {
        // applet is going away...
    }

    public void paint(Graphics g) {
        g.drawString(message, 5, 15);
    }
}
```

The init hook allows the applet to do whatever it wants to initialize the applet the first time.

repaint() is a concrete method in the Applet class that lets upper-level components know the applet needs to be redrawn.

The start hook allows the applet to do something when the applet is just about to be displayed on the web page.

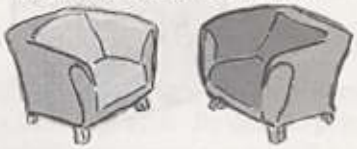
If the user goes to another page, the stop hook is used, and the applet can do whatever it needs to do to stop its actions.

And the destroy hook is used when the applet is going to be destroyed, say, when the browser pane is closed. We could try to display something here, but what would be the point?

Well looky here! Our old friend the paint() method! Applet also makes use of this method as a hook.

Concrete applets make extensive use of hooks to supply their own behaviors. Because these methods are implemented as hooks, the applet isn't required to implement them.

Fireside Chats



Tonight's talk: **Template Method and Strategy compare methods.**

Template Method

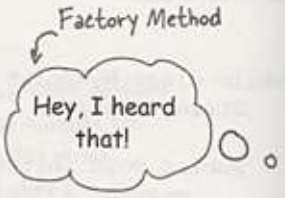
Hey Strategy, what are you doing in my chapter? I figured I'd get stuck with someone boring like Factory Method

I was just kidding! But seriously, what are you doing here? We haven't heard from you in eight chapters!

You might want to remind the reader what you're all about, since it's been so long.

Hey, that does sound a lot like what I do. But my intent's a little different from yours; my job is to define the outline of an algorithm, but let my subclasses do some of the work. That way, I can have different implementations of an algorithm's individual steps, but keep control over the algorithm's structure. Seems like you have to give up control of your algorithms.

Strategy



Nope, it's me, although be careful – you and Factory Method are related, aren't you?

I'd heard you were on the final draft of your chapter and I thought I'd swing by to see how it was going. We have a lot in common, so I thought I might be able to help...

I don't know, since Chapter 1, people have been stopping me in the street saying, "Aren't you that pattern..." So I think they know who I am. But for your sake: I define a family of algorithms and make them interchangeable. Since each algorithm is encapsulated, the client can use different algorithms easily.

I'm not sure I'd put it quite like *that*... and anyway, I'm not stuck using inheritance for algorithm implementations. I offer clients a choice of algorithm implementation through object composition.

Template Method

I remember that. But I have more control over my algorithm and I don't duplicate code. In fact, if every part of my algorithm is the same except for, say, one line, then my classes are much more efficient than yours. All my duplicated code gets put into the superclass, so all the subclasses can share it.

Yeah, well, I'm *real* happy for ya, but don't forget I'm the most used pattern around. Why? Because I provide a fundamental method for code reuse that allows subclasses to specify behavior. I'm sure you can see that this is perfect for creating frameworks.

How's that? My superclass is abstract.

Like I said Strategy, I'm *real* happy for you. Thanks for stopping by, but I've got to get the rest of this chapter done.

Got it. Don't call us, we'll call you...

Strategy

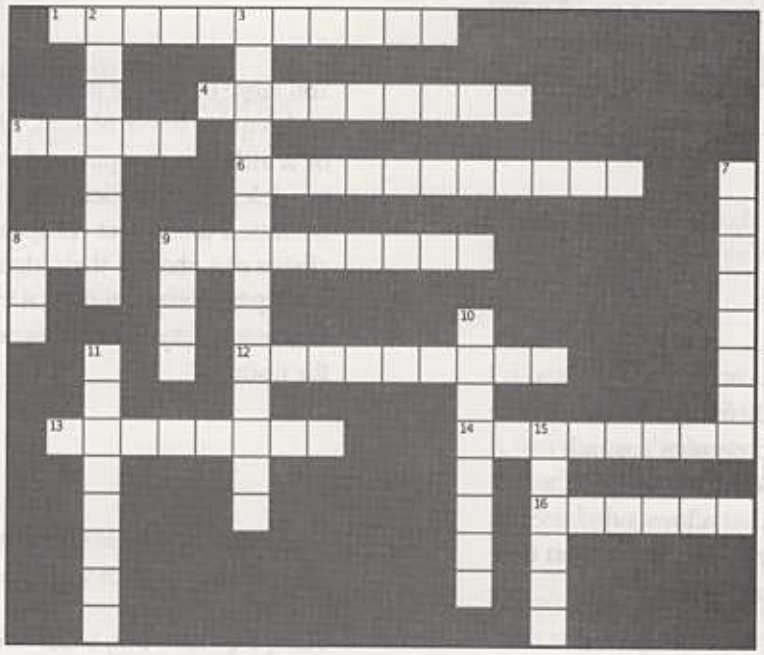
You *might* be a little more efficient (just a little) and require fewer objects. *And* you might also be a little less complicated in comparison to my delegation model, but I'm more flexible because I use object composition. With me, clients can change their algorithms at runtime simply by using a different strategy object. Come on, they didn't choose *me* for Chapter 1 for nothing!

Yeah, I guess... but, what about dependency? You're way more dependent than me.

But you have to depend on methods implemented in your superclass, which are part of your algorithm. I don't depend on anyone; I can do the entire algorithm myself!

Okay, okay, don't get touchy. I'll let you work, but let me know if you need my special techniques anyway, I'm always glad to help.

It's that time again....



Across

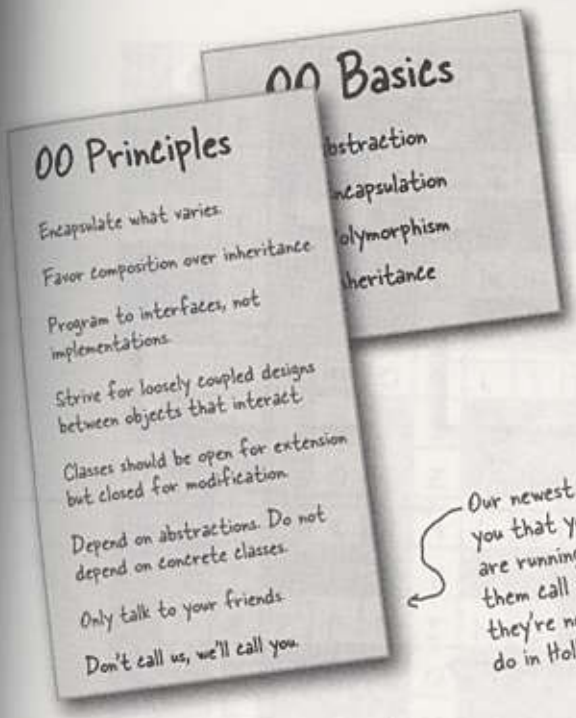
1. Strategy uses _____ rather than inheritance
4. Type of sort used in Arrays
5. The JFrame hook method that we overrode to print "I Rule"
6. The Template Method Pattern uses _____ to defer implementation to other classes
8. Coffee and _____
9. Don't call us, we'll call you is known as the _____ Principle
12. A template method defines the steps of an _____
13. In this chapter we gave you more _____
14. The template method is usually defined in an _____ class
16. Class that likes web pages

Down

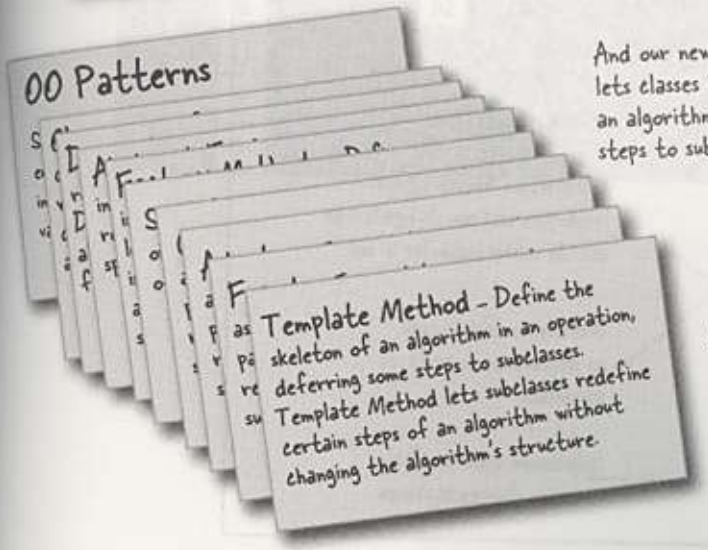
2. _____ algorithm steps are implemented by hook methods
3. Factory Method is a _____ of Template Method
7. The steps in the algorithm that must be supplied by the subclasses are usually declared _____
8. Huey, Louie and Dewey all weigh _____ pounds
9. A method in the abstract superclass that does nothing or provides default behavior is called a _____ method
10. Big headed pattern
11. Our favorite coffee shop in Objectville
15. The Arrays class implements its template method as a _____ method

Tools for your Design Toolbox

We've added Template Method to your toolbox. With Template Method you can reuse code like a pro while keeping control of your algorithms.



Our newest principle reminds you that your superclasses are running the show, so let them call your subclasses when they're needed, just like they do in Hollywood.



And our newest pattern lets classes implementing an algorithm defer some steps to subclasses.

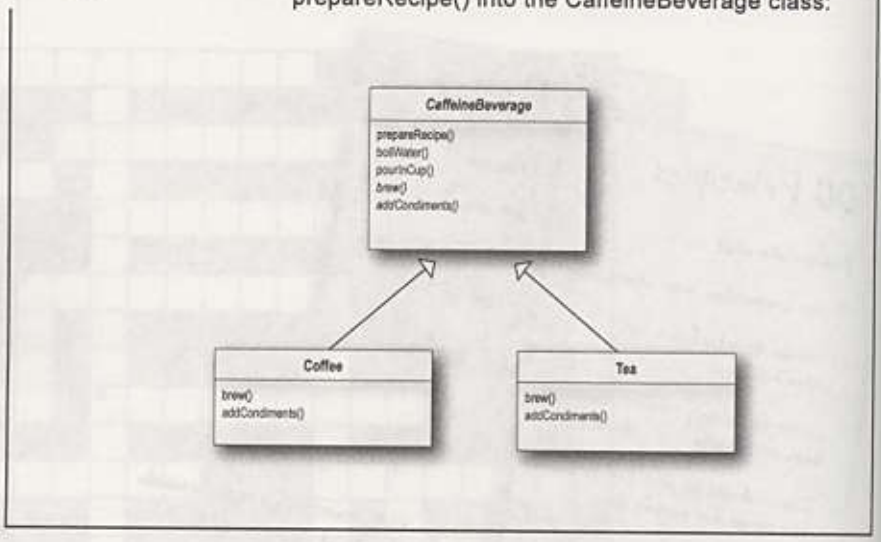
BULLET POINTS

- 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.
- Abstract methods are implemented by subclasses.
- Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.
- To prevent subclasses from changing the algorithm in the template method, declare the template method as final.
- The Hollywood Principle guides us to put decision-making in high-level modules that can decide how and when to call low level modules.
- You'll see lots of uses of the Template Method Pattern in real world code, but don't expect it all (like any pattern) to be designed "by the book."
- 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.

Exercise solutions

Sharpen your pencil

Draw the new class diagram now that we've moved prepareRecipe() into the CaffeineBeverage class:



WHO DOES WHAT?

Match each pattern with its description:

Pattern	Description
Template Method	Encapsulate interchangable 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 create

Exercise solutions