The Singleton Pattern

"One of a Kind Objects"

Singleton: What is this?

- How to instantiate just one object - one and only one!
- Why?
  - Many objects we need only one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, etc.
  - If more than one instantiated: Incorrect program behavior, overuse of resources, inconsistent results.
- Alternatives:
  - Use a global variable: assign an object to a global variable, then that object might be created when application begins.
    - Downside: If application never ends up using it and object is resource intensive --> waste!
  - Use a static variable
    - How do you prevent creation of more than one class object?

The Little Singleton

How would you create a single object?	`new MyClass();`
And what if another object wanted to create a `MyClass`? Could it call `new` on `MyClass` again?	Yes, why not.
Can we always instantiate a class one or more times?	Yes. Caveat: Only if it is **public** class
And if not?	If it's not a **public** class, only **classes in the same package** can instantiate it, but they can instantiate it more than once.

Is this possible?
`public class MyClass {`
`  private MyClass() { }`
`}`
	Yes. It is a legal definition

What does it mean?	A class that can't be instantiated because it has a private constructor

The Little Singleton (con't)

Is there any class that could use a private constructor?	`MyClass` is the only code that could call it.

What does this mean?
`public class MyClass {`
`  public static MyClass getInstance() { }`
`}`
	We can call the `static` method:
	`MyClass.getInstance()`

Now, can I instantiate a MyClass?
`public class MyClass {`
`  private MyClass() { }`
`  public static MyClass getInstance() {`
`    return new MyClass();`
`  }`
`}`
	Yes

How you would create a `MyClass` object now?	`MyClass.getInstance()`
Can you create now more than one `MyClass`?	Yes, why not?

The Little Singleton (con't)

But I would like to have only one **MyClass**. How you do it?

```
public MyClass {
   private static MyClass oneClass;

   private MyClass() { }

   public static MyClass getInstance() {
      if (oneClass == null) {
          oneClass = new MyClass();
      }
      return oneClass;
   }
}
```

The Classic Singleton Pattern

```
public class Singleton {
   private static Singleton uniqueInstance;
   // other useful instance variables

   private Singleton() {
   }

   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton();
      }
      return uniqueInstance;
   }

   // other useful methods
}
```

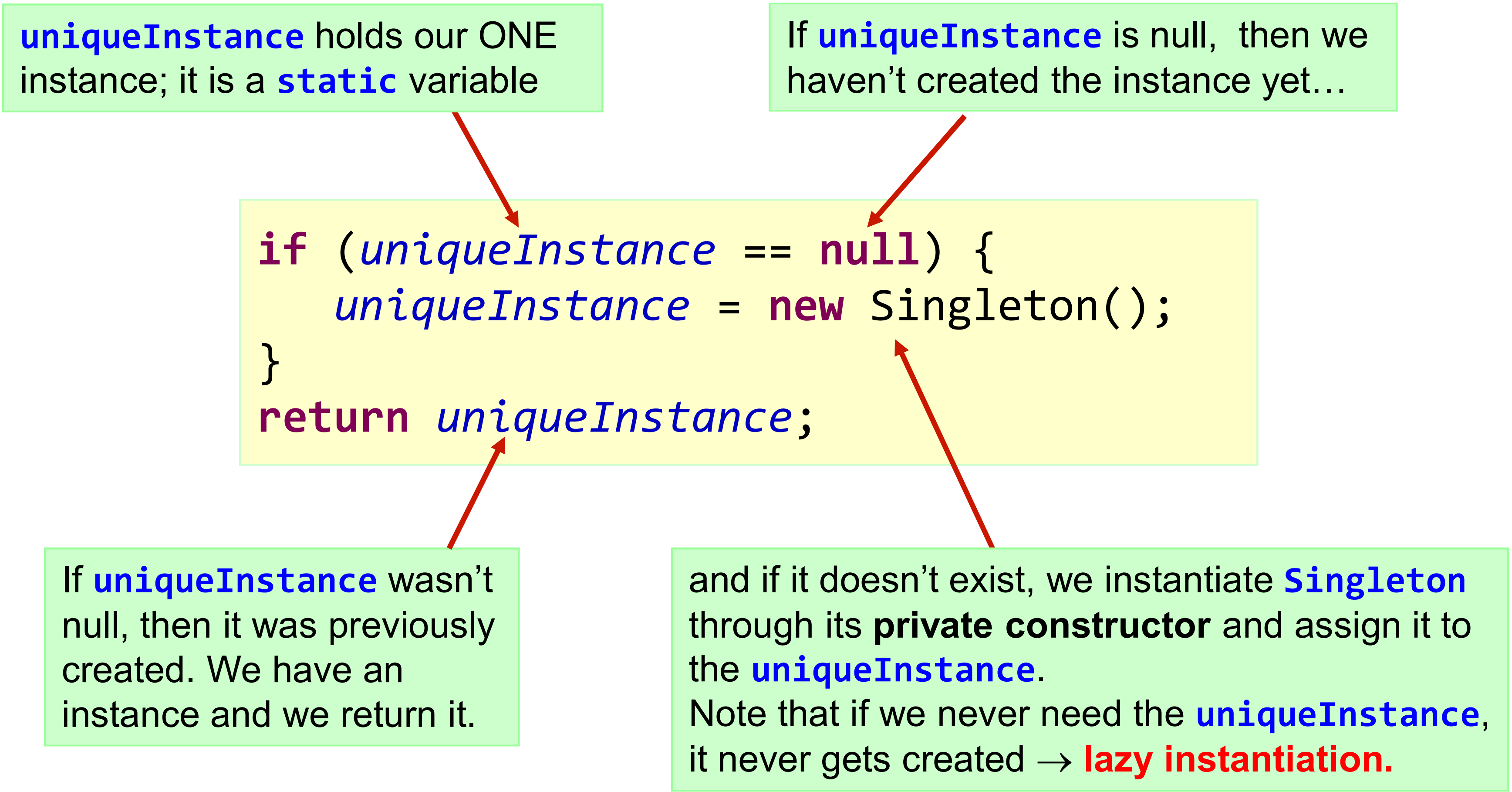
static variable to hold our one instance of the class **Singleton**.

Constructor is declared **`private`**; only singleton can instantiate this class!

The method instantiate the class and return an instance of it.

Singleton is a regular class, so it has other useful instances and methods.

Code Up Close



```
if (uniqueInstance == null) {
   uniqueInstance = new Singleton();
}
return uniqueInstance;
```

uniqueInstance holds our ONE instance; it is a **static** variable

If **uniqueInstance** is null, then we haven't created the instance yet…

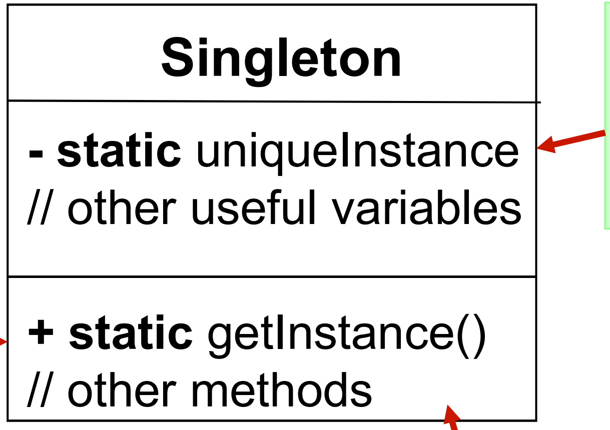
If **uniqueInstance** wasn't null, then it was previously created. We have an instance and we return it.

and if it doesn't exist, we instantiate **Singleton** through its **private constructor** and assign it to the **uniqueInstance**.
Note that if we never need the **uniqueInstance**, it never gets created → **lazy instantiation.**

Singleton Pattern Defined

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

The `getInstance()` method is `static`, which means it is a class method, so you can conveniently access this method anywhere using `Singleton.getInstance()`



Singleton

- static uniqueInstance
// other useful variables

+ static getInstance()
// other methods

The `uniqueInstance` class variable holds our one and only one instance of `Singleton`.

A class implementing a **Singleton Pattern** is more than a Singleton; it is a general purpose class with its own set of data and methods.

Computer controled chocolate boiler
The job of boiler is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phrase of making the bars

```java
public class ChocolateBoiler {
   private boolean empty;
   private boolean boiled;
   public ChocolateBoiler() {
      empty = true; boiled = false;
   }
   public void fill() {
      if (isEmpty()) {
         empty = false;
         boiled = false;
         // fill the boiler with a milk/chocolate mixture
      }
   }
   public void drain() {
      if (!isEmpty() && isBoiled()) {
         // drain the boiled milk and chocolate
         empty = true;
      }
   }
   public void boil() {
      if (!isEmpty() && !isBoiled()) {
         // bring the contents to a boil
         boiled = true;
      }
   }
   public boolean isEmpty() { return empty; }
   public boolean isBoiled() { return boiled; }
}
```

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags

To drain the boiler it must be full (not empty) and also boiled. Once it is drained we set empty back to true

To boil the mixture, the boiler has to be full and not already boiled. Once it is boiled we set boiled flag to true

Turning ChocolateBoiler into singleton

```
public class ChocolateBoiler {
   private boolean empty;
   private boolean boiled;
   private static ChocolateBoiler uniqueInstance;

   private ChocolateBoiler(){
      empty = true;
      boiled = false;
   }
   public static ChocolateBoiler getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new ChocolateBoiler();
      }
      return uniqueInstance;
   }
   public void fill() { ... }
```

Houston, we have a problem….

- We improved the `Chocolate Boiler` code with the `Singleton` pattern and we added some optimizations to the `Chocolate Boiler Controller` that makes use of multiple threads

- Ugh… the `Chocolate Boiler`’s `fill()` method was able to start filling the boiler even though a batch of milk and chocolate was already boiling! That’s 500 gallons of milk and chocolate spilled.

- Could the addition of threads have caused this?

Be the JVM

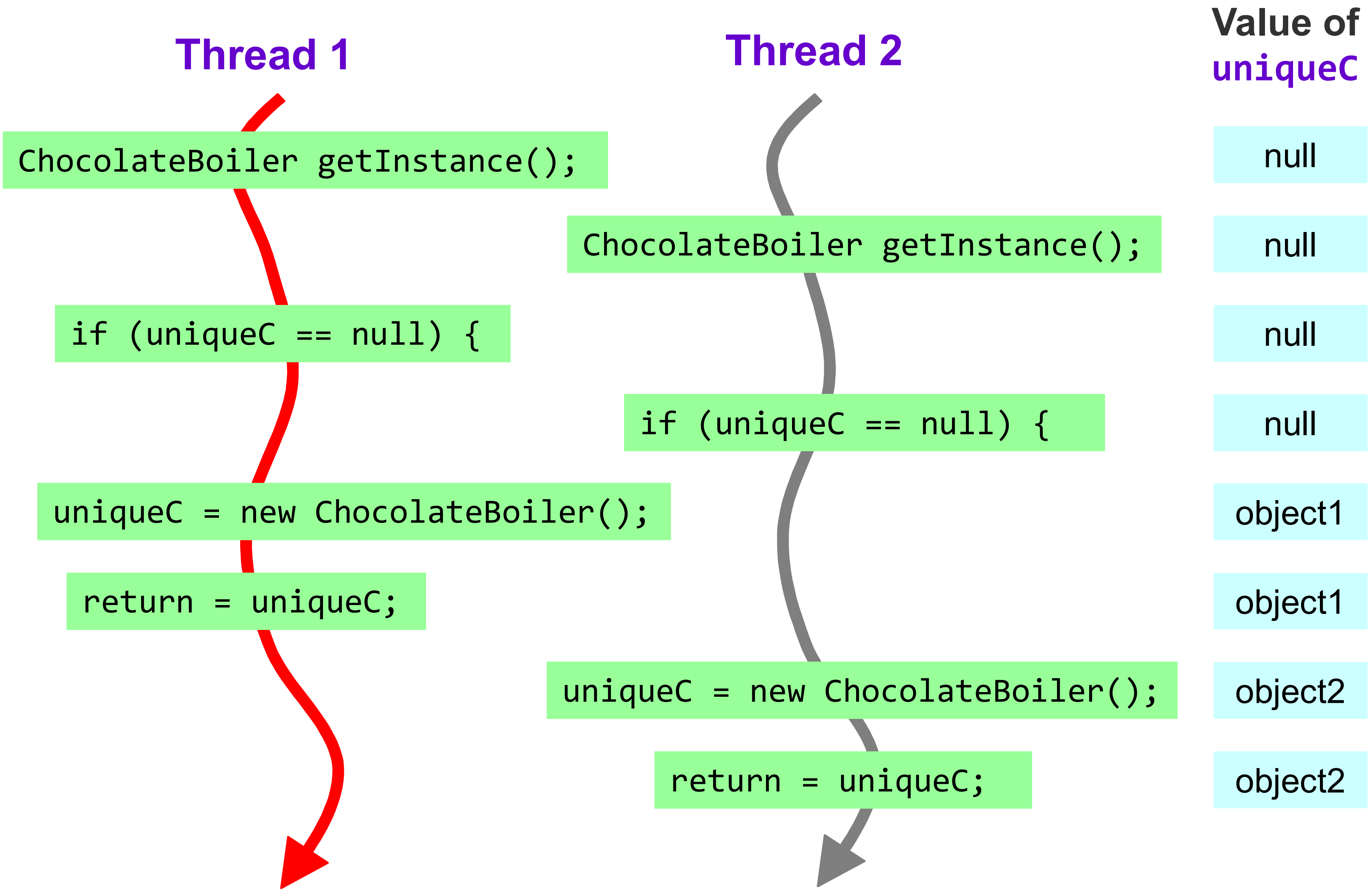
- We have two threads each executing this code:

```
ChocolateBoiler boiler = ChocolateBoiler.getInstance();
boiler.fill();
boiler.boil();
boiler.drain();
```

- Could the two threads get hold of different boiler objects?

```
public static ChocolateBoiler getInstance() {
  if (uniqueInstance == null) {
    uniqueInstance = new ChocolateBoiler();
  }
  return uniqueInstance;
}
```

What happened?



Thread 1	Thread 2	Value of uniqueC
`ChocolateBoiler getInstance();`		null
	`ChocolateBoiler getInstance();`	null
`if (uniqueC == null) {`		null
	`if (uniqueC == null) {`	null
`uniqueC = new ChocolateBoiler();`		object1
`return = uniqueC;`		object1
	`uniqueC = new ChocolateBoiler();`	object2
	`return = uniqueC;`	object2

Dealing with Multi-threading

- Easy fix: make **getInstance()** a **synchronized** method

```java
public class Singleton {
   private static Singleton uniqueIn
   // other useful instance variable

   private Singleton() {}
   public static synchronized Singleton getInstance() {
      if (uniqueInstance == null) {
         uniqueInstance = new Singleton();
      }
      return uniqueInstance;
   }
   // other useful methods here
}
```

Adding the **synchronized** keyword force every thread to wait its turn before it can enter the method. That is, **no two threads may enter the method at the same time**.

- This fixes the problem, but synchronization is expensive.
- Synchronization is really only needed the first time through this method. Once we have created the first **Singleton** instance, we have no further need to synchronize this method.

Can we improve multithreading?

1. Do nothing if the performance of **`getInstance()`** isn’t critical to your application. (remember that synchronizing can decrease performance by a factor of 100)
2. Move to an ***eagerly created instance*** rather than a lazily created one.

Go ahead and create an instance of **Singleton** in a **static** initializer.
This code is guaranteed to be thread safe!

```
public class Singleton {
  private static Singleton uniqueInstance = new Singleton();

  private Singleton() {}

  public static Singleton getInstance() {
    return uniqueInstance;
  }
}
```

We’ve already got an instance, so just return it.

Can we improve multithreading?

- Use “double-checked locking” to reduce the use of synchronization in **getInstance()**

If performance is an issue then this method can drastically reduce overhead!

```
public class Singleton {
   private volatile static Singleton uniqueInstance;

   private Singleton() {}
   public static Singleton getInstance() {
      if (uniqueInstance == null) {
         synchronized (Singleton.class) {
            if (uniqueInstance == null) {
               uniqueInstance = new Singleton();
            }
         }
      }
      return uniqueInstance;
   }
```

Check for an instance and if there isn’t one, enter the synchronized block.

we only **synchronize** the first time through.

The ***volatile*** keyword ensures that multiple threads handle **uniqueInstance** variable correctly when it is being initialized to the **Singleton** instance.

Once in the block, check again if null. If so create instance.

Summary

- The Singleton Pattern ensures you have at most one instance of a class in your application
- The Singleton Pattern also provides a global access point to that instance.
- Java's implementation of the Singleton Pattern makes use of a private constructor, a static method combined with a static variable
- Examine your performance and resource constraints and carefully choose an appropriate Singleton implementation for multi-threaded applications.
- Be careful if you are using multiple class loaders -- this can defeat the purpose of the Singleton implementation

The Template Method Pattern

Encapsulating Algorithms

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

The recipe for coffee and tea are very similar!

Whipping up some Coffee in Java

```
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");
  }
}
```

Recipe for coffee - each step is implemented as a separate method.

Each of these methods implements one step of the algorithm.

And now for 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 pourInCup() {
    System.out.println("Pouring into cup");
  }
  public void addLemon() {
    System.out.println("Adding Lemon");
  }
}
```

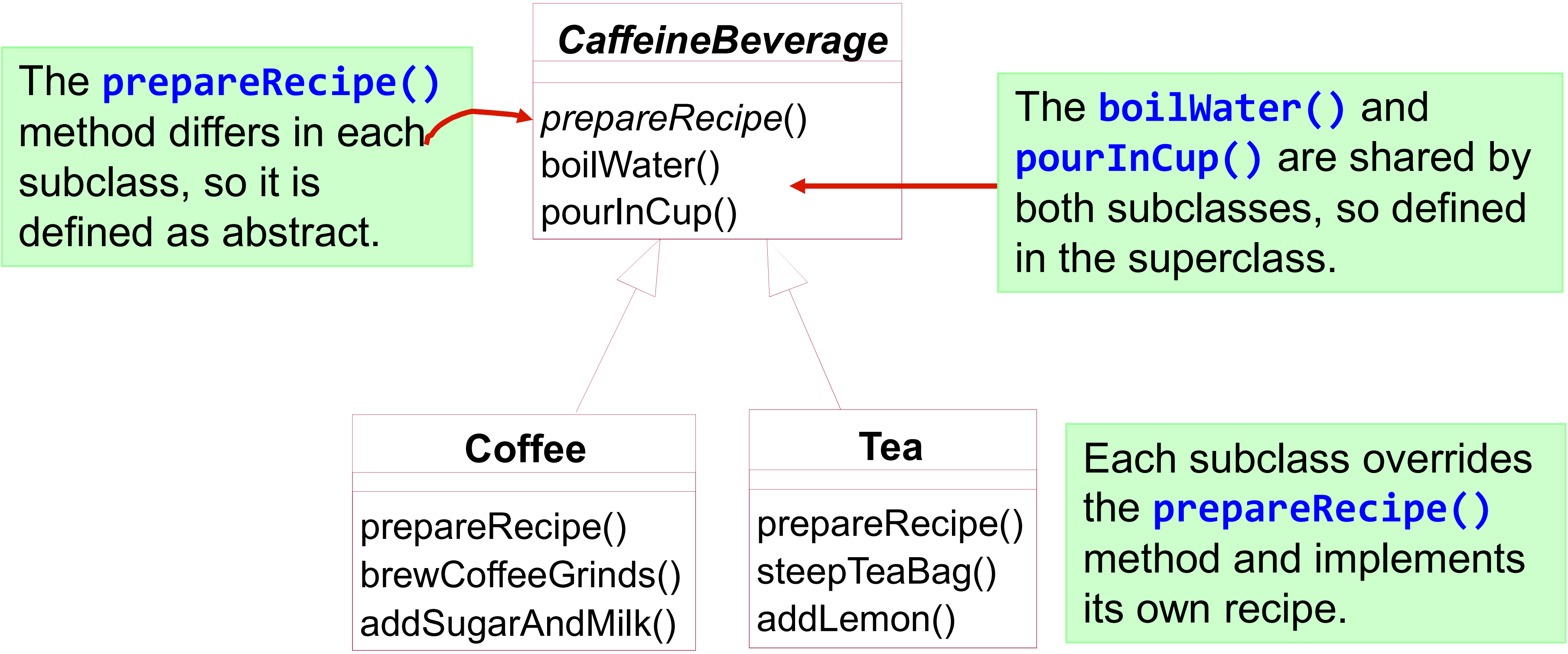
Very similar to the coffee – 2nd and 4th steps are different.

These methods are exactly the same as the are in `Coffee`

These methods are specialized to `Tea`

We have **code duplication** - that’s a good sign that we need to clean up the design. We should abstract the commonality into a base class since coffee and tea are so similar, right?

Sir, may I abstract your Coffee, Tea?



CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()

Coffee

prepareRecipe()
brewCoffeeGrinds()
addSugarAndMilk()

Tea

prepareRecipe()
steepTeaBag()
addLemon()

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

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

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

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

Is this a good redesign? Are we overlooking some other commonality? What are other ways that Coffee and Tea are similar?

What else do they have in common?

- Both the recipes follow the same algorithm:
  1. Boil some water
  2. Use hot water to extract the tea or coffee
  3. Pour the resulting beverage into a cup
  4. Add the appropriate condiments to the beverage.

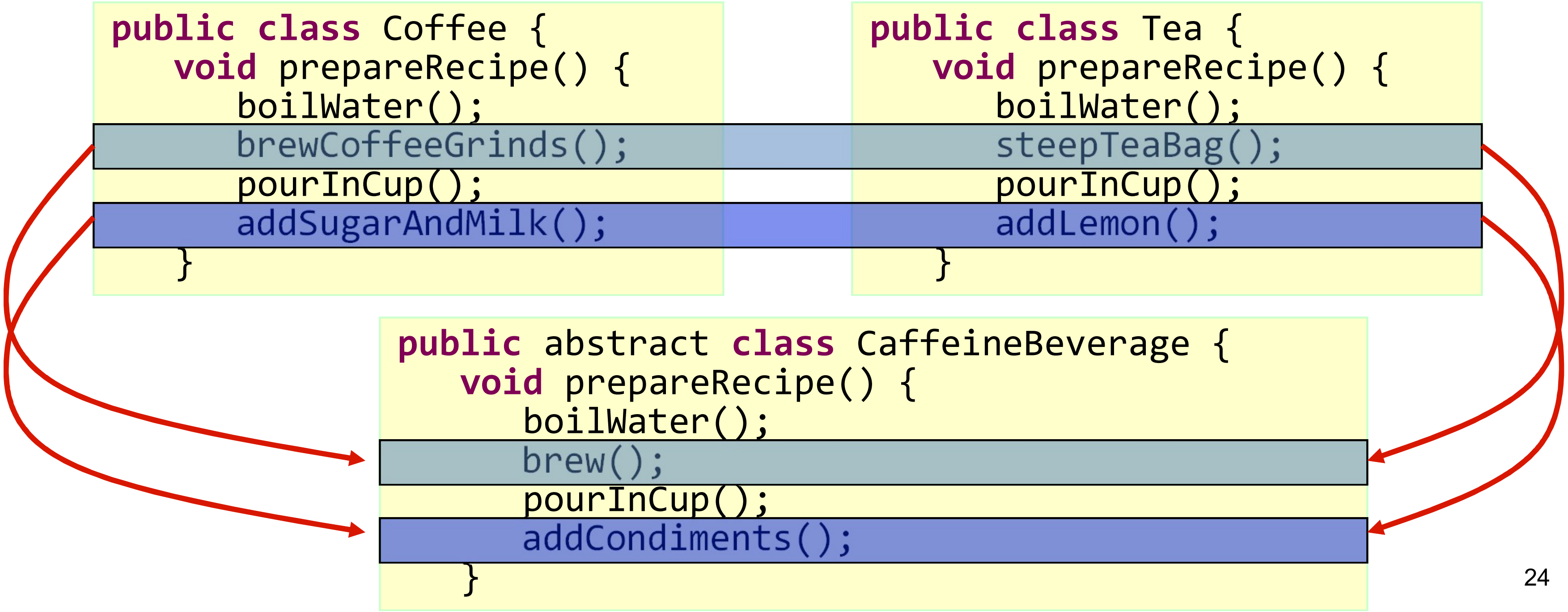
These two are already abstracted into the base class

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

Can we abstract `prepareRecipe()` too? Yes ...

Abstracting PrepareRecipe()

- Provide a common interface for the different methods
  - Problem : Coffee uses brewCoffeeGrinds() and addSugarAndMilk() methods while Tea uses steepTeaBag() and addLemon() methods
  - Steeping and brewing are pretty analogous – so a common interface may be the ticket: brew() and addCondiments()

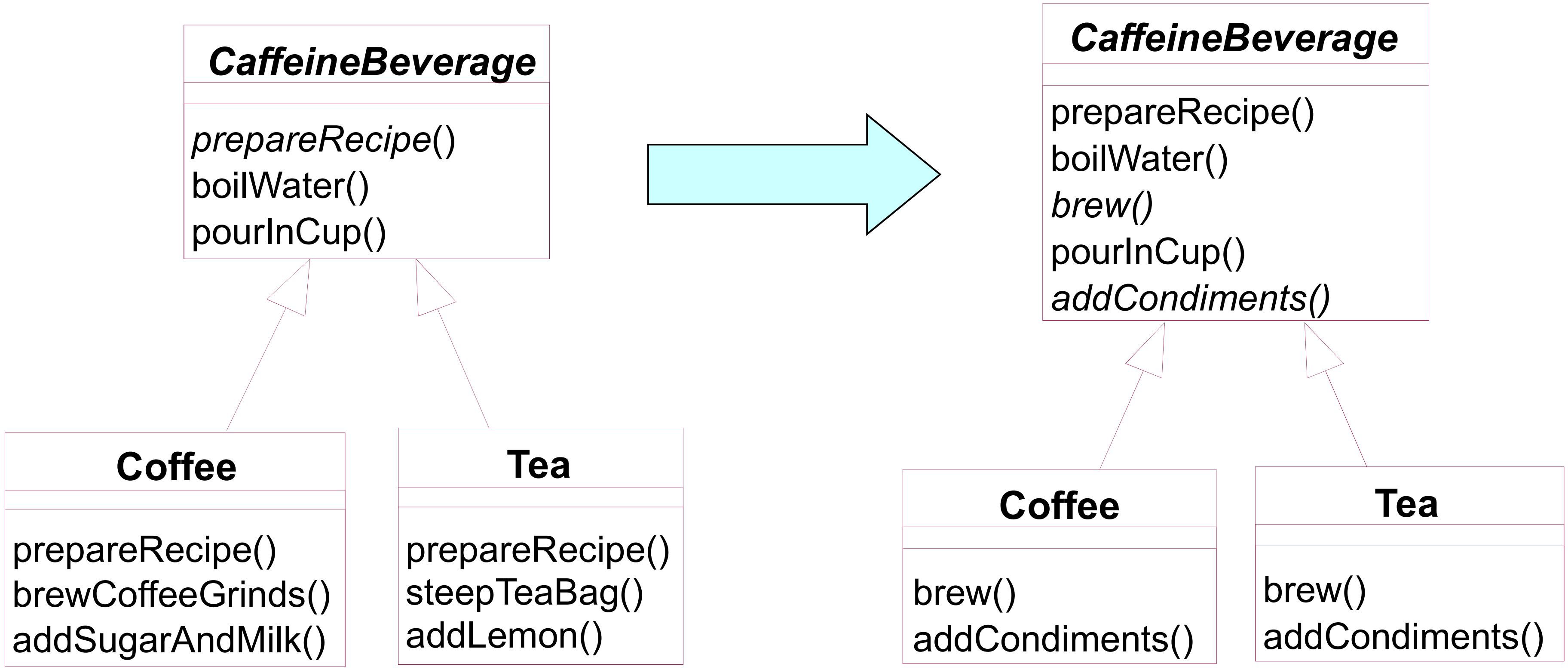


```
public class Coffee {
   void prepareRecipe() {
      boilWater();
      brewCoffeeGrinds();
      pourInCup();
      addSugarAndMilk();
   }
```

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

```
public abstract class CaffeineBeverage {
   void prepareRecipe() {
      boilWater();
      brew();
      pourInCup();
      addCondiments();
   }
```

Abstracting `prepareRecipe()`



CaffeineBeverage

prepareRecipe()
boilWater()
pourInCup()

Coffee

prepareRecipe()
brewCoffeeGrinds()
addSugarAndMilk()

Tea

prepareRecipe()
steepTeaBag()
addLemon()

CaffeineBeverage

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

Coffee

brew()
addCondiments()

Tea

brew()
addCondiments()

The New Java Classes….

```java
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");
  }
}
```

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

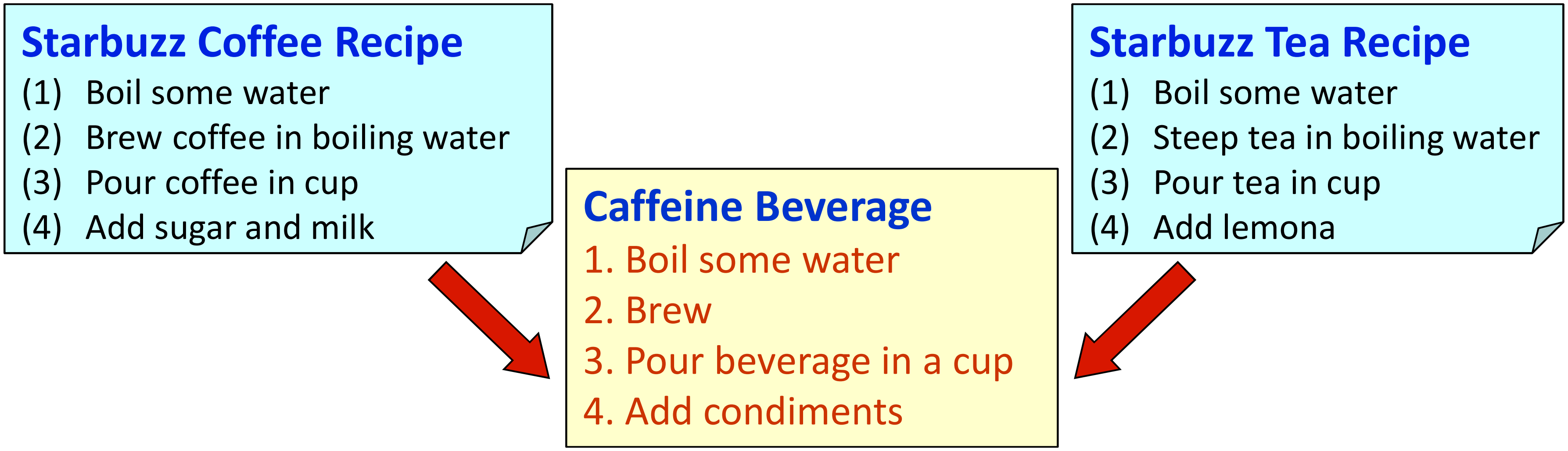
The New Java Classes….

```
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");
  }
}
```

What have we done?

- We have 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.
  - We’ve made it so that some of the steps in the recipe rely on the subclass implementations.



Starbuzz Coffee Recipe

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

Caffeine Beverage

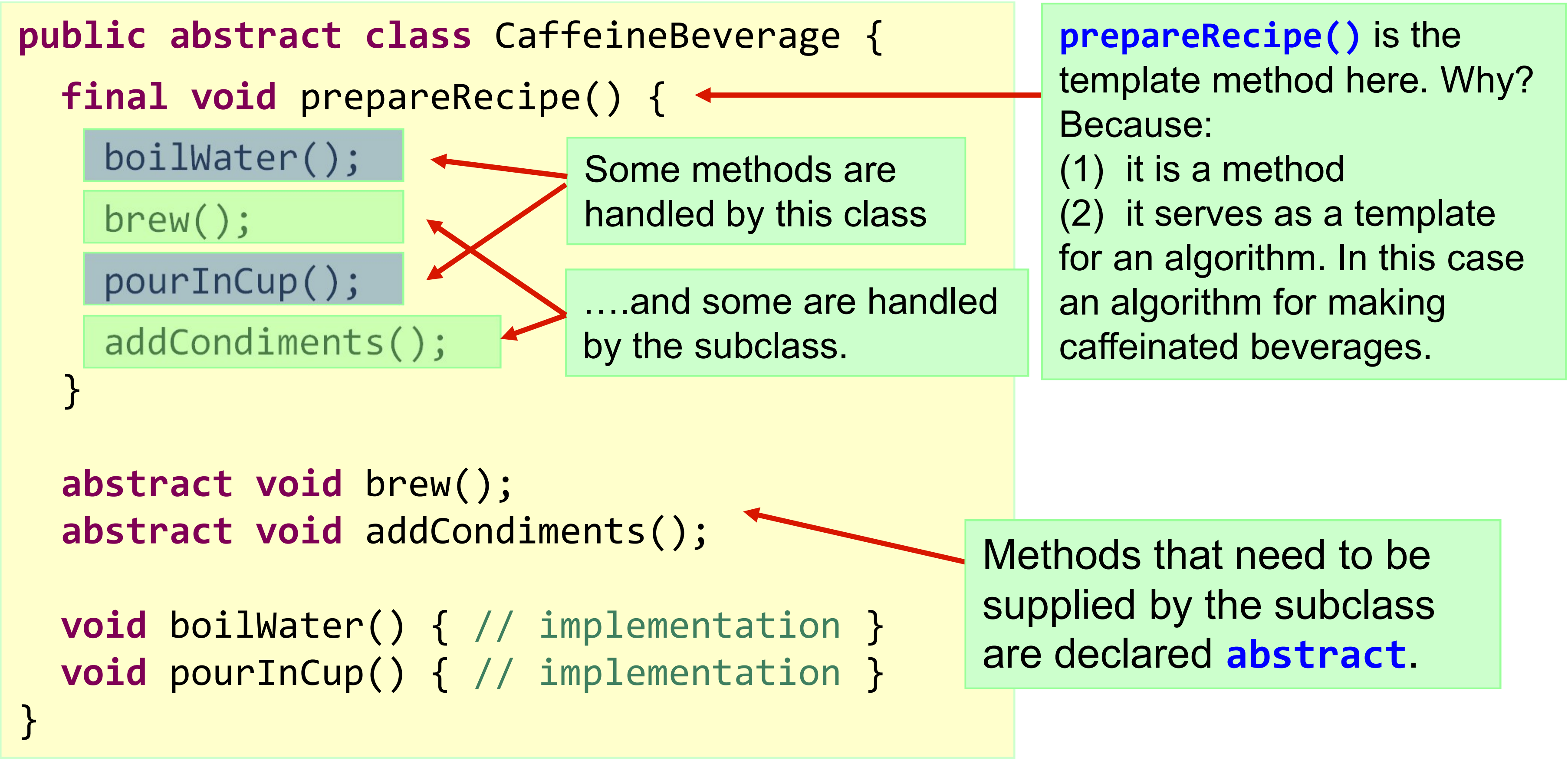
1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

Starbuzz Tea Recipe

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

Essentially - we have implemented the Template Method Pattern!

The New Java Classes….



```
public abstract class CaffeineBeverage {
  final void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
  }

  abstract void brew();
  abstract void addCondiments();

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

Some methods are handled by this class

….and some are handled by the subclass.

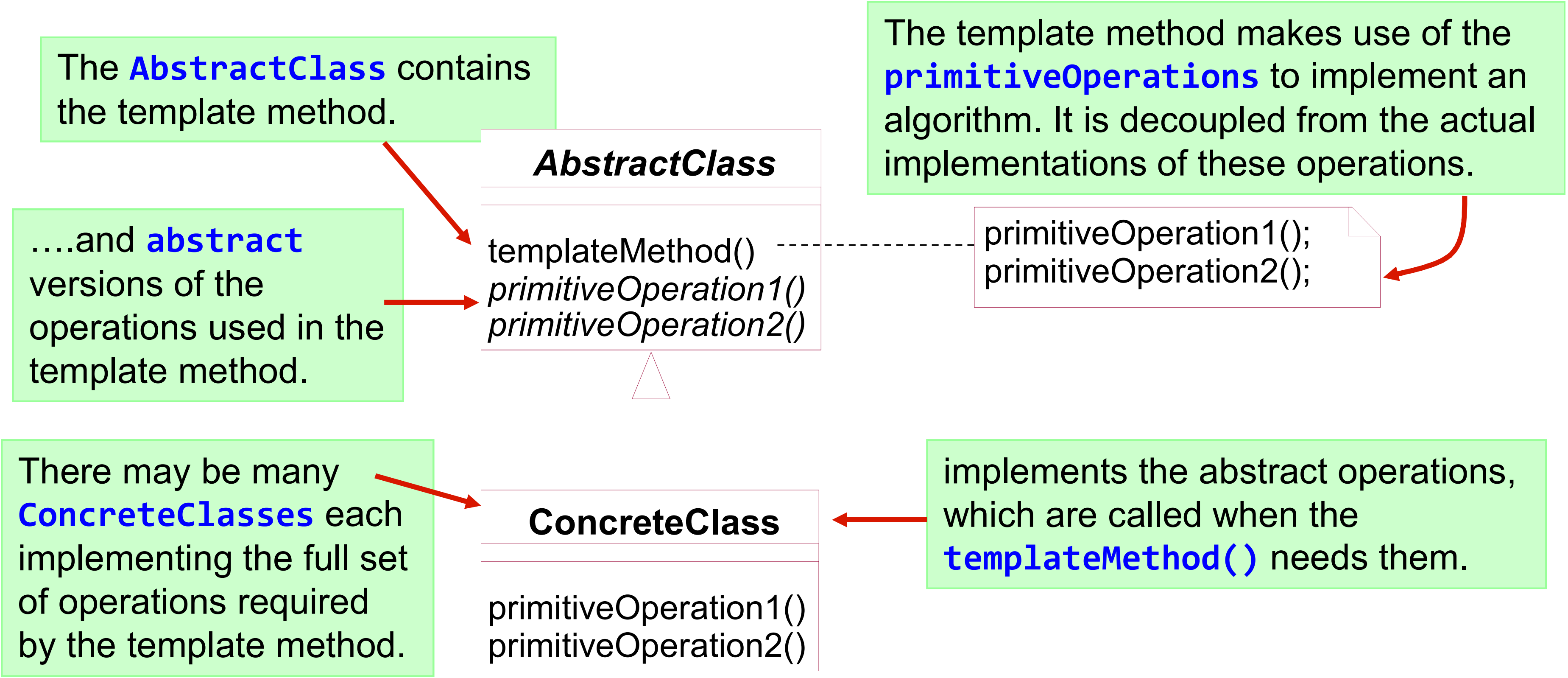
prepareRecipe() is the template method here. Why? Because:
(1) it is a method
(2) it serves as a template for an algorithm. In this case an algorithm for making caffeinated beverages.

Methods that need to be supplied by the subclass are declared **abstract**.

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

The Template Method 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.



The `AbstractClass` contains the template method.

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

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

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

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

Hooked on the Template Method

- A hook is a method that is declared in the abstract class, but **only given an empty or default implementation**.
  - Gives the subclasses the ability to “**hook into**” the algorithm at various points, if they wish; they can ignore the hook as well.

```
public abstract class CaffeineBeverageWithHook {
  void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    if (customerWantsCondiments()){
      addCondiments();
    }
  }
  abstract void brew();
  abstract void addCondiments();
  void boilWater() { // implementation }
  void pourInCup() { // implementation }
  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()`

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

If subclasses want to use the hook they simply override it!

Using hook: Overide it in our subclass

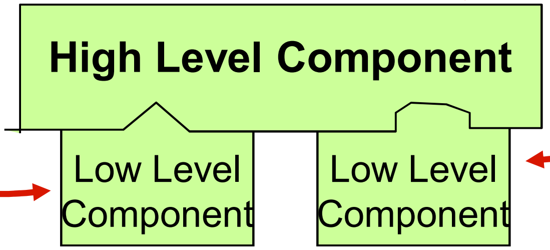
```
public class CoffeeWithHook extends CaffeineBeverageWithHook {
   public void brew() {
      System.out.println("Dripping Coffee through filter");
   }
   public void addCondiments() {
      System.out.println("Adding Sugar and Milk");
   }
   @Override
   public boolean customerWantsCondiments() {
      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 (y/n)? ");
      Scanner in = new Scanner(System.in));
      answer = in.readLine();
      if (answer == null) { return "no"; }
      return answer;
   }
}
```

DP: The Hollywood 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….
- With the Hollywood principle
  - We allow low level components to hook themselves into a system
  - But high level components determine when they are needed and how.
  - High level components give the low-level components a “don’t call us, we’ll call you” treatment.



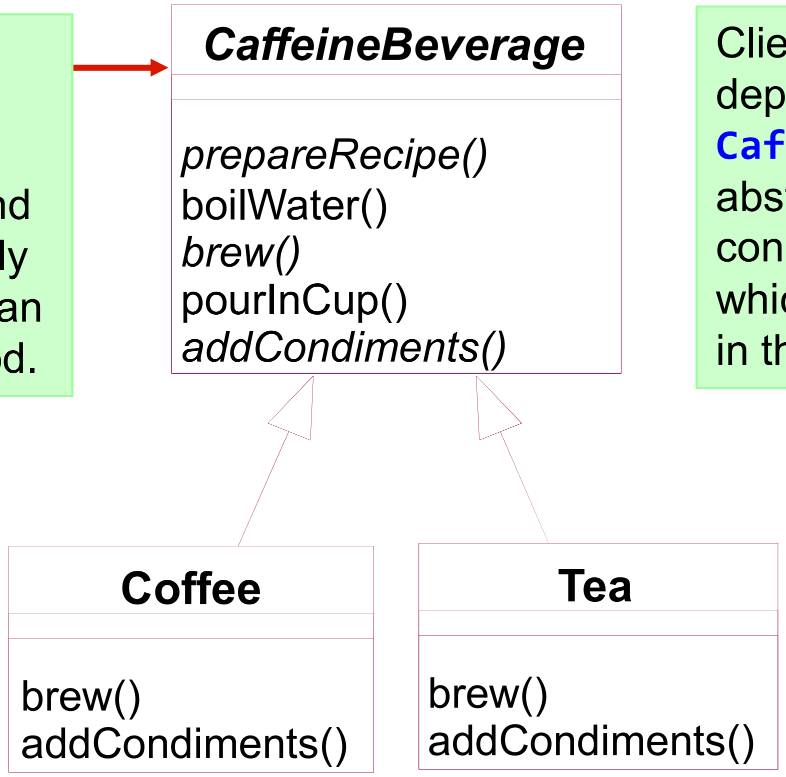
But high-level components control when and how.

Low-level components can participate in the computation.

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

The Hollywood Principle and the 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 are 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.

What other patterns make use of the Hollywood Principle?

Template Methods in the Wild

- Template method is a very common pattern and you’re going to find lots in the wild!
- Some examples:
  - Sorting with Template Method
  - Swinging with Frames

Sorting with Template Method

- **Arrays** - common operation - sort them!
- Designers of Java **Arrays** provide a handy template method for sorting.

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

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;
}
```

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**

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.

Sorting some Ducks…

- Assume that you have an array of ducks to sort. How would you do it?
- Use **the `sort()` template method** in the `Array` to do the sort
  - Need to implement the `compareTo()` method – to tell the `sort()` how to compare ducks.
- Any issues here?

In the `Template` method we typically would subclass something.
An array doesn't subclass anything.
Reason: 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.
One more detail - because we are not using subclassing, the `sort()` method needs to know that you have implemented the `compareTo()` method.
To handle this there is the `Comparable` interface that your class needs to implement. It has just one method – `compareTo()`

Comparing Ducks and 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;
      }
   }
}
```

Comparing Ducks and 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("\n After Sorting");
    display(ducks);
  }

  public static void display(Duck[] ducks) {
    for (int j=0; j < ducks.length; j++)
      System.out.println(ducks[j]);
  }
}
```

Is this really a Template Method Pattern?

- The pattern calls for implementing an algorithm, and letting subclasses supply the implementation of the steps.
- `Array.sort()` is doing just that!
- Patterns in real life are often adaptations of the book patterns `Arrays.sort()`
- Designers of the method had some constraints:
  - In general you can’t subclass Java Array and they wanted sort to be useful for all arrays
  - For `Array sort()` implementation:
    - `static` method was defined
    - Comparison was deferred to the items being sorted.
  - So yes this is a `Template Method`!

Swingin' with Frames

- JFrame - most basic Swing container and inherits a paint() method.
- Default behavior of paint() method - does nothing because it is a hook.
- By overriding the paint() method, you can insert yourself into JFrame's algorithm for displaying its area of screen and have your own graphic output incorporated into the JFrame.

- Next up -- a simple example using JFrame to override the paint() method.

Simple JFrame Example

```
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");
   }
}
```

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()`** method.

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

Applets example

- Applets provide numerous hooks!

```
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() {
      message = "Goodbye, cruel world";
      repaint();
   }
   public void paint(Graphics g) {
      g.drawString(message, 5, 15);
   }
}
```

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 that the applet needs to be redrawn.

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.

used when the applet is going to be destroyed, say, when the browser pane is closed.

Well looky here! **`Applet`** also makes use of the **`paint()`** 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.

Summary (1/2)

- 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 subclasses.

Summary (2/2)

- To prevent the 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 the 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 **Factory Method** is a specialization of the **Template Method**!