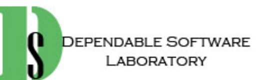
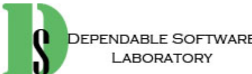
Design Patterns

건국대학교
유준범

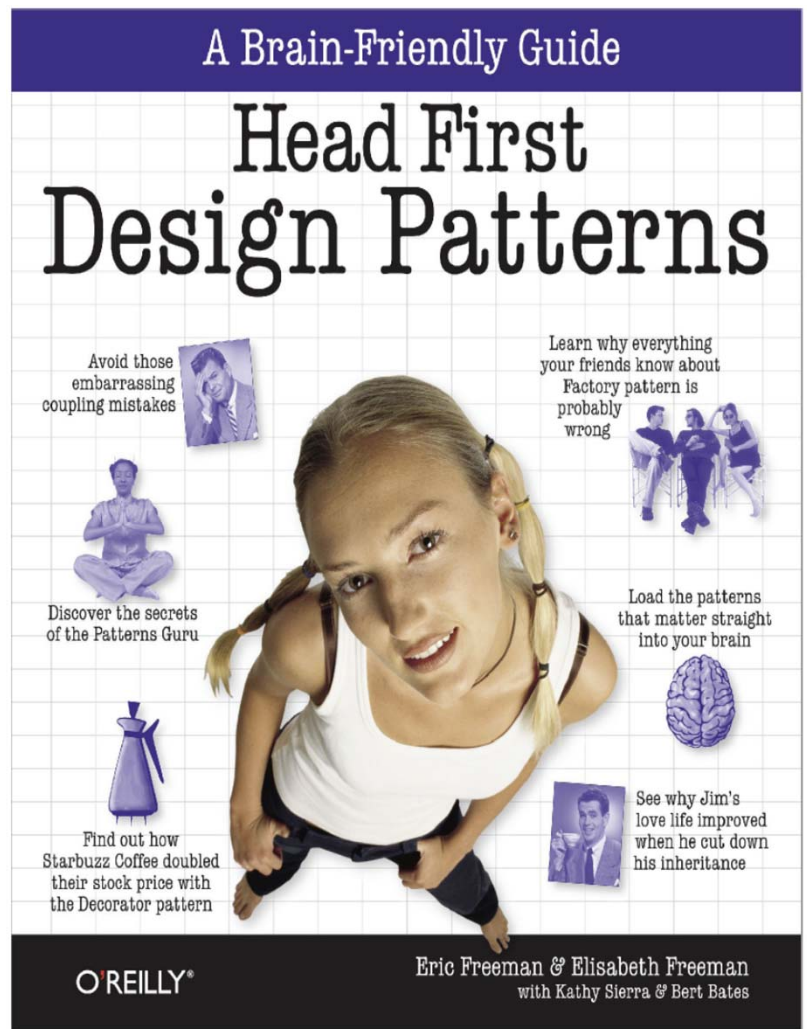


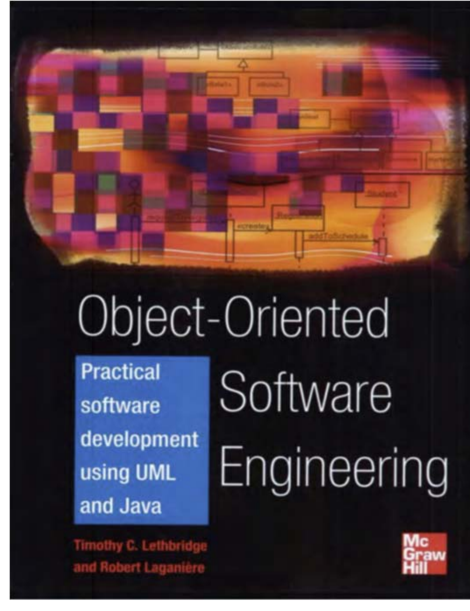
Contents

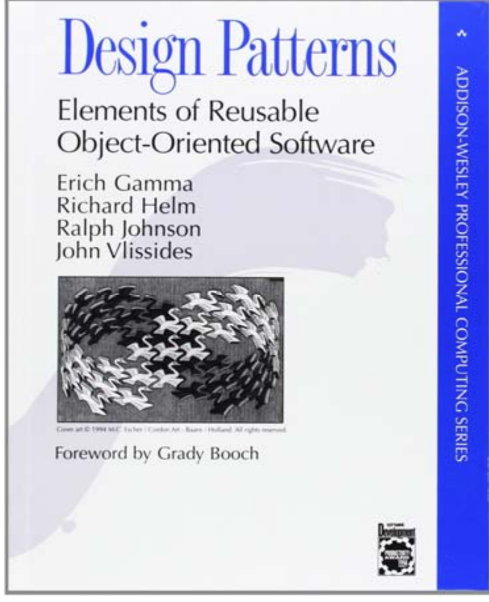
1. Intro to Design Patterns (The Strategy Pattern)
2. The Observer Pattern
3. The Decorator Pattern
4. The Factory Pattern
5. The Singleton Pattern
6. The Command Pattern
7. The Adapter and Façade Patterns
8. The Template Method Pattern
9. The Iterator and Composite Patterns
10. The State Pattern
11. The Proxy Pattern
12. Composed Patterns
13. Better Living with Patterns



Text and References







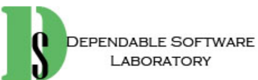
KU KONKUK
UNIVERSITY

“Keeping your Objects in the know”

2. The Observer Pattern

The Observer Pattern

- **Don't miss out when something interesting happens!**
- We've got a pattern that keeps your objects in the know when something they might care about happens. Objects can even decide at runtime whether they want to be kept informed.
- The Observer Pattern is one of **the most heavily used patterns in the JDK**, and it's incredibly useful. Before we're done, we'll also look at **one-to-many relationships** and loose coupling (yeah, that's right, we said coupling).



- **Congratulations! Your team has just won the contract to build Weather-O-Rama, Inc.'s next-generation, Internet-based Weather Monitoring Station.**

Weather-O-Rama, Inc.
100 Main Street
Tornado Alley, OK 45021

Statement of Work

Congratulations on being selected to build our next generation Internet-based Weather Monitoring Station!

The weather station will be based on our patent pending WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We'd like for you to create an application that initially provides three display elements: current conditions, weather statistics and a simple forecast, all updated in real time as the WeatherData object acquires the most recent measurements.

Further, this is an expandable weather station. Weather-O-Rama wants to release an API so that other developers can write their own weather displays and plug them right in. We'd like for you to supply that API!

Weather-O-Rama thinks we have a great business model: once the customers are hooked, we intend to charge them for each display they use. Now for the best part: we are going to pay you in stock options.

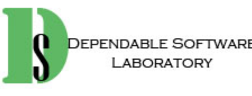
We look forward to seeing your design and alpha application.

Sincerely,

Johnny Hurricane

Johnny Hurricane, CEO

P.S. We are overnighting the WeatherData source files to you.



The Weather Monitoring Application overview

- **Our job is to create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.**

Current Conditions is one of three different displays. The user can also get weather stats and a forecast.

Humidity sensor device

Temperature sensor device

Pressure sensor device

Weather Station

pulls data

WeatherData object

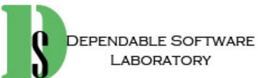
displays

Current Conditions
Temp: 72°
Humidity: 60
Pressure:

Display device

Weather-O-Rama provides

What we implement



Unpacking the WeatherData class

- As promised, the next morning the WeatherData source files arrive. When we peek inside the code, things look pretty straightforward:

WeatherData
getTemperature() getHumidity() getPressure() measurementsChanged() // other methods

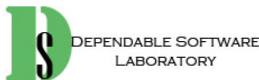
These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

The developers of the WeatherData object left us a clue about what we need to add...

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

WeatherData.java



- **Our job is to implement measurementsChanged() so that it updates the three displays for current conditions, weather stats, and forecast.**

Remember, this Current Conditions is just ONE of three different display screens.

Display device

Display Two

Display Three

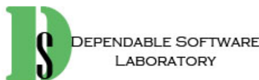
What do we know so far?

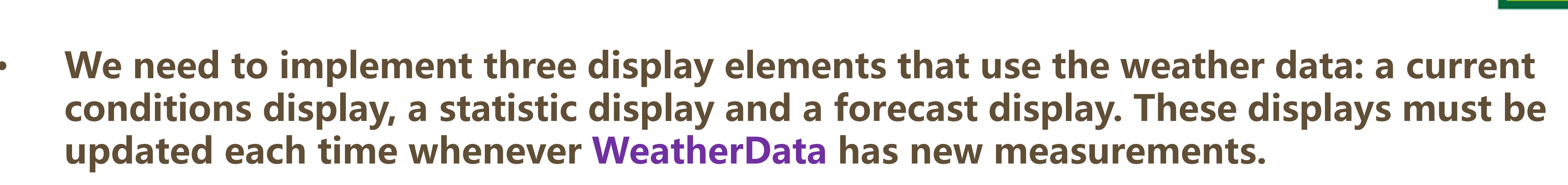
- The WeatherData class has getter methods for three measurement values: temperature, humidity and barometric pressure.

```
getTemperature()
getHumidity()
getPressure()
```

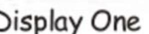
- The measurementsChanged() method is called any time new weather measurement data is available. We don't know or care how this method is called; we just know that it is.

```
measurementsChanged()
```





- We need to implement three display elements that use the weather data: a current conditions display, a statistic display and a forecast display. These displays must be updated each time whenever WeatherData has new measurements.



Display Three

- The system must be expandable - other developers can create new custom display elements and users can add or remove as many display elements as they want to the application. Currently, we know about only the initial three display.

Future displays

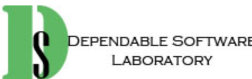
Taking a first attempt - misguided

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

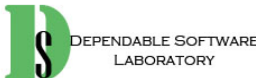
Call each display element to update its display, passing it the most recent measurements.



Sharpen your pencil

Based on our first implementation, which of the following apply? (Choose all that apply.)

- ❑ A. We are coding to concrete implementations, not interfaces.
- ❑ B. For every new display element we need to alter code.
- ❑ C. We have no way to add (or remove) display elements at run time.
- ❑ D. The display elements don't implement a common interface.
- ❑ E. We haven't encapsulated the part that changes.
- ❑ F. We are violating encapsulation of the WeatherData class.



What's wrong with our implementation?

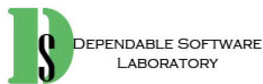
```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Area of change, we need to encapsulate this.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure values.

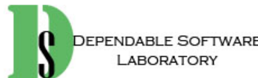
By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

Umm, I know I'm new here, but given that we are in the Observer Pattern chapter, maybe we should start using it?



Meet the Observer Pattern

- **You know how newspaper or magazine subscriptions work:**
  ① A newspaper publisher goes into business and begins publishing newspapers.
  ② You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
  ③ You unsubscribe when you don't want papers anymore, and they stop being delivered.
  ④ While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.



Publishers + Subscribers = Observer Pattern

The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.

When data in the Subject changes, the observers are notified.

Subject object manages some bit of data.

2

int

Subject Object

2

2

2

Dog Object

Cat Object

Mouse Object

Observer Objects

New data values are communicated to the observers in some form when they change.

Duck Object

This object isn't an observer, so it doesn't get notified when the Subject's data changes.

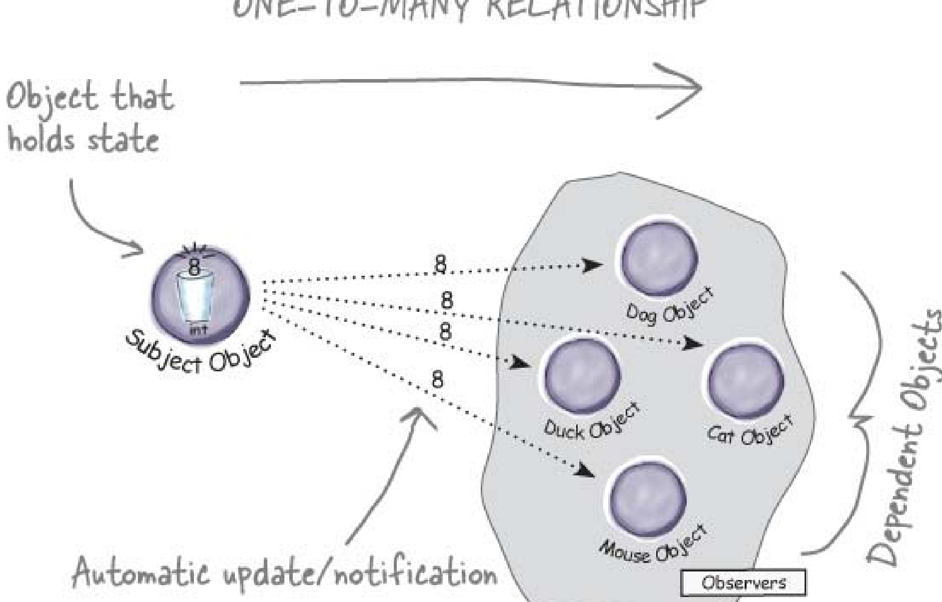
The Observer Pattern defined

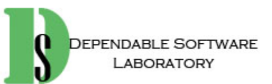
- The subject and observers define the **one-to-many relationship**. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with *new values*.

> **The Observer Pattern** defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

License

- There are a few different ways to implement the Observer Pattern, but most revolve around a class design that includes Subject and Observer interfaces.





The Observer Pattern defined: The class diagram

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.

<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()

observers

<<interface>>
Observer
update()

ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}

getState()
setState()

subject

ConcreteObserver
update()
// other Observer specific methods

A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

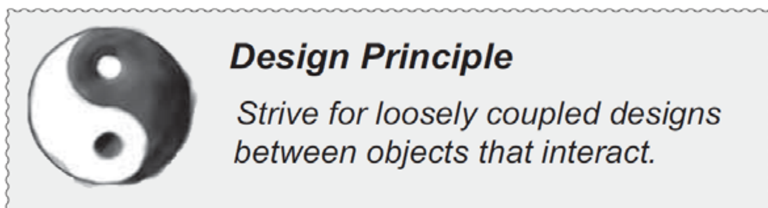
The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

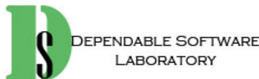
The power of Loose Coupling

- **When two objects are loosely coupled, they can interact, but have very little knowledge of each other.**

- The Observer Pattern provides an object design, where subjects and observers are loosely coupled. Why?
  - The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).
  - We can add new observers at any time.
  - We never need to modify the subject to add new types of observers.
  - We can reuse subjects or observers independently of each other.
  - Changes to either the subject or an observer will not affect the other.



- **Loosely coupled designs allow us to build flexible OO systems that can handle change, because they minimize the interdependency between objects.**



Designing the Weather Station

Here's our subject interface, this should look familiar.

All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.

Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.

<<interface>> **Subject**
registerObserver()
removeObserver()
notifyObservers()

observers

<<interface>> **Observer**
update()

<<interface>> **DisplayElement**
display()

CurrentConditions
update()
display() { // display current measurements }

subject

WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()

ThirdPartyDisplay
update()
display() { // display something else based on measurements }

StatisticsDisplay
update()
display() { // display the average, min and max measurements }

ForecastDisplay
update()
display() { // display the forecast }

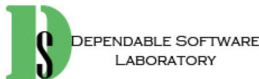
This display element shows the current measurements from the WeatherData object.

WeatherData now implements the Subject interface.

This one keeps track of the min/avg/max measurements and displays them.

Developers can implement the Observer and Display interfaces to create their own display element.

This display shows the weather forecast based on the barometer.



<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()

observers

<<interface>>
Observer
update()

<<interface>>
DisplayElement
display()

CurrentConditions
update()
display() { // display current measurements }

subject

WeatherData
registerObserver()
removeObserver()
notifyObservers()

getTemperature()
getHumidity()
getPressure()
measurementsChanged()

ThirdPartyDisplay
update()
display() { // display something else based on measurements }

StatisticsDisplay
update()
display() { // display the average, min and max measurements }

ForecastDisplay
update()
display() { // display the forecast }

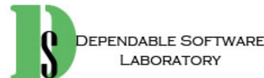
This display element shows the current measurements from the WeatherData object.

WeatherData now implements the Subject interface.

This one keeps track of the min/avg/max measurements and displays them.

Developers can implement the Observer and Display interfaces to create their own display element.

다른 Display들도 register를 위해
WeatherData를 subject로 받아야 합니다.



Implementing the Weather Station

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

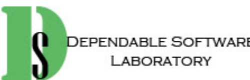
```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.



```
public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

Here we implement the Subject interface.

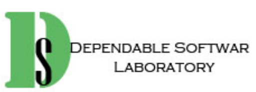
When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the Web.



Implementing the display elements

This display implements Observer so it can get changes from the WeatherData object.

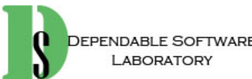
It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.



Power up the Weather Station

```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

First, create the WeatherData object.

If you don't want to download the code, you can comment out these two lines and run it.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

File Edit Window Help StormyWeather

```
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

Using Java's built-in Observer Pattern (~ Java 8)

The Observable class keeps track of all your observers and notifies them for you.

This should look familiar. In fact, it's exactly the same as our previous class diagram!

We left out the DisplayElement interface, but all the displays still implement it too.

Observable is a CLASS not an interface, so WeatherData extends Observable.

Observable
- addObserver()
- deleteObserver()
- notifyObservers()
- setChanged()

observers

<<interface>>
Observer
- *update()*

subject

GeneralDisplay
- update()
- display()

StatisticsDisplay
- update()
- display()

ForecastDisplay
- update()
- display()

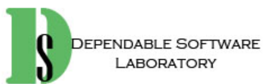
WeatherData
- getTemperature()
- getHumidity()
- getPressure()

This doesn't look familiar! Hold tight, we'll get to this in a sec...

Here's our Subject, which we can now also call the Observable. We don't need the register(), remove() and notifyObservers() methods anymore; we inherit that behavior from the superclass.

There will be a few changes to make to the update() method in the concrete Observers, but basically it's the same idea... we have a common Observer interface, with an update() method that's called by the Subject.

With Java's built-in support, all you have to do is extend Observable and tell it when to notify the Observers. The API does the rest for you.



KU KONKUK UNIVERSITY

How Java's built-in Observer Pattern works

- **For an Object to become an observer…**
  - As usual, implement the Observer interface (this time the **java.util.Observer** interface) and call addObserver() on any Observable object. Likewise, to remove yourself as an observer, just call deleteObserver().

- **For the Observable (Subject) to send notifications…**
  - You need to be Observable by extending the **java.util.Observable** superclass. From there it is a two-step process:
    ① You first must call the setChanged() method to signify that the state has changed in your object.
    ② Then, call one of two notifyObservers() methods:

either `notifyObservers()` **or** `notifyObservers(Object arg)`

This version takes an arbitrary data object that gets passed to each Observer when it is notified.

data object

- **For an Observer to receive notifications…**
  - It implements the update method, as before, but the signature of the method is a bit different:

`update(Observable o, Object arg)`

The Subject that sent the notification is passed in as this argument.

This will be the data object that was passed to notifyObservers(), or null if a data object wasn't specified.

Wait, before we get to that, why do we need this setChanged() method? We didn't need that before.

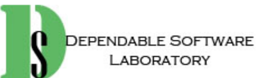
```
setChanged() {
   changed = true
}

notifyObservers(Object arg) {
   if (changed) {
      for every observer on the list {
         call update (this, arg)
      }
      changed = false
   }
}

notifyObservers() {
   notifyObservers(null)
}
```

Pseudocode for the Observable class.

The setChanged() method sets a changed flag to true.

notifyObservers() only notifies its observers if the changed flag is TRUE.

And after it notifies the observers, it sets the changed flag back to false.



Reworking the Weather Station with the built-in support

1. Make sure we are importing the right Observable.
2. We are now subclassing Observable.
3. We don't need to keep track of our observers anymore, or manage their registration and removal (the superclass will handle that), so we've removed the registerObserver(), removeObserver() and notifyObservers() methods.
4. Our constructor no longer needs to create a data structure to hold Observers.

```
public WeatherData() {
    observers = new ArrayList<Observer>();
}
```

* Notice we aren't sending a data object with the notifyObservers() call. That means we're using the PULL model.

5. We now first call setChanged() to indicate the state has changed before calling notifyObservers().
6. These methods aren't new, but because we are going to use "pull" we thought we'd remind you they are here. The Observers will use them to get at the WeatherData object's state.

```
import java.util.Observable;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers(); *
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

- **Now, let's rework the CurrentConditionsDisplay**

① Again, make sure we are importing the right Observer/Observable.

② We now are implementing the Observer interface from java.util.

③ Our constructor now takes an Observable and we use this to add the current conditions object as an Observer.

④ We've changed the update() method to take both an Observable and the optional data argument.

⑤ In update(), we first make sure the observable is of type WeatherData and then we use its getter methods to obtain the temperature and humidity measurements. After that we call display().

```
private Subject weatherData;
```

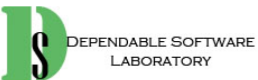
```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

Running the new code

- **Hmm, do you notice anything different? Look again...**
  - You'll see all the same calculations, but mysteriously, the order of the text output is different. Why might this happen? Think for a minute before reading on...
- **Never depend on order of evaluation of the Observer notifications**
  - The **java.util.Observable** has implemented its notifyObservers() method such that the Observers are notified in a *different* order than our own implementation. Who's right? Neither; we just chose to implement things in different ways.

```
File Edit Window Help StormyWeather
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

```
File Edit Window Help TryTihisAtHome
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: More of the same
Avg/Max/Min temperature = 80.0/82.0/78.0
Current conditions: 78.0F degrees and 90.0% humidity
%
```



The dark side of java.util.Observable

Doesn't java.util.Observable violate our OO design principle of programming to interfaces, not implementations?

- **Yes, good catch.**
- As you've noticed, Observable is a **class**, not an ***interface*** , and worse, it doesn't even *implement* an interface.
- Unfortunately, the **java.util.Observable** implementation has a number of problems that limit its usefulness and reuse.
- That's not to say it doesn't provide some utility, but there are some large potholes to watch out for.

The Observer Pattern in the JDK

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

Simple Swing application that just creates a frame and throws a button in it.

Makes the devil and angel objects listeners (observers) of the button.

Code to set up the frame goes here.

Here are the class definitions for the observers, defined as inner classes (but they don't have to be).

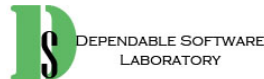
Rather than update(), the actionPerformed() method gets called when the state in the subject (in this case the button) changes.

And here's the output when we click on the button.

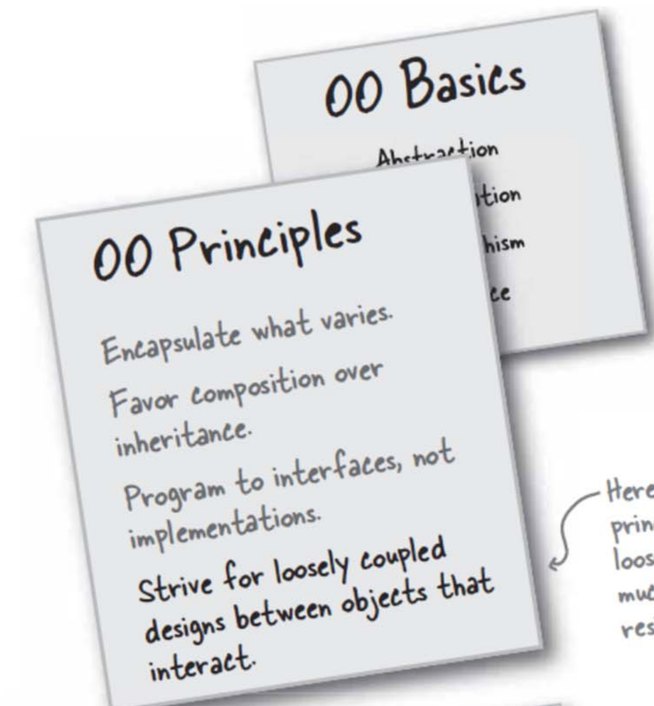
```
File Edit Window Help HeMadeMeDoIt
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

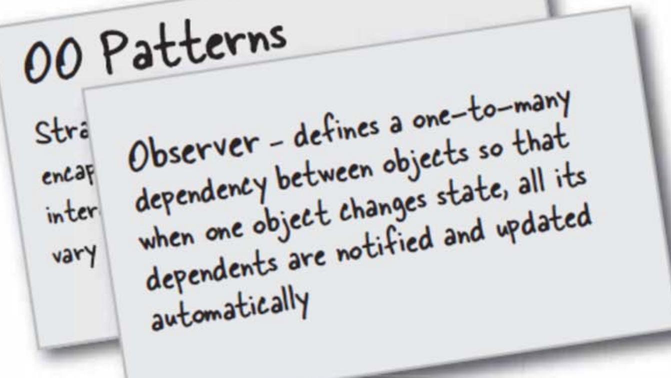
Devil answer

Angel answer



Tools for your design toolbox

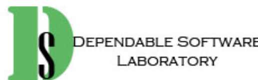




A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern - just wait until we talk about MVC!

BULLET POINTS

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects, or as we also know them, Observables, update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer Interface.
- You can push or pull data from the Observable when using the pattern (pull is considered more "correct").
- Don't depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose java.util. Observable.
- Watch out for issues with the java.util.Observable implementation.
- Don't be afraid to create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.
- You'll also find the pattern in many other places, including JavaBeans and RMI.



“One of a Kind Objects”

5. The Singleton Pattern

The Singleton Pattern

- **Our next stop is the Singleton Pattern, our ticket to creating one-of-a-kind objects for which there is only one instance.**

- You might be happy to know that of all patterns, the Singleton is the simplest in terms of its class diagram; in fact, the diagram holds just a single class!
- But don't get too comfortable; despite its simplicity from a class design perspective, we are going to encounter quite a few bumps and potholes in its implementation.

The Little Singleton

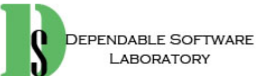
- **A small Socratic exercise in the style of The Little Lisper**

```
public MyClass {

    private MyClass() {}

}
```

What does it mean?

I suppose it is a class that can't be instantiated because it has a private constructor.

Because I'd have to have an instance of the class to call it, but I can't have an instance because no other class can instantiate it. It's a chicken and egg problem: I can use the constructor from an object of type MyClass, but I can never instantiate that object because no other object can use "new MyClass()".



Okay. It was just a thought.

What does this mean?

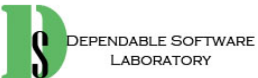
```
public MyClass {

   public static MyClass getInstance() {
   }
}
```

MyClass is a class with a static method. We can call the static method like this:

```
MyClass.getInstance();
```

Why did you use MyClass, instead of some object name?

Well, getInstance() is a static method; in other words, it is a CLASS method. You need to use the class name to reference a static method.



Now can I instantiate a MyClass?

```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

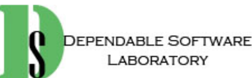
So, now can you think of a second way to instantiate an object?

`MyClass.getInstance();`

Can you finish the code so that only ONE instance of MyClass is ever created?

Yes, I think so...

(You'll find the code on the next page.)



Dissecting the classic Singleton Pattern implementation

Let's rename MyClass to Singleton.

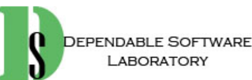
```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

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

    // other useful methods here
}
```

We have a static variable to hold our one instance of the class Singleton.

Our constructor is declared private; only Singleton can instantiate this class!

The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.



The Chocolate Factory

- The job of the **boiler** is to take in chocolate and milk, bring them to a boil, and then pass them on to the next phase of making chocolate bars.

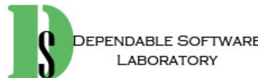
```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    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;
    }
}
```

This code is only started when the boiler is empty!

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

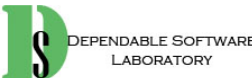
To drain the boiler, it must be full (non-empty) and also boiled. Once it is drained we set empty back to true.

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



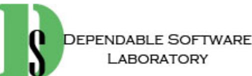
- Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a singleton?

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

    [                              ]

    [      ] ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    [                              ]

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```



Applying the Singleton Patten

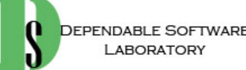
```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    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() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```



```java
package headfirst.singleton.chocolate;

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) {
            System.out.println("Creating unique instance of Chocolate Boiler");
            uniqueInstance = new ChocolateBoiler();
        }
        else {
        System.out.println("Returning instance of Chocolate Boiler");
        }

        return uniqueInstance;
    }

    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;
    }
}
```

```java
package headfirst.singleton.chocolate;

public class ChocolateController {
    public static void main(String args[]) {
        ChocolateBoiler boiler = ChocolateBoiler.getInstance();
        boiler.fill();
        boiler.boil();
        boiler.drain();

        // will return the existing instance
        ChocolateBoiler boiler2 = ChocolateBoiler.getInstance();
    }
}
```

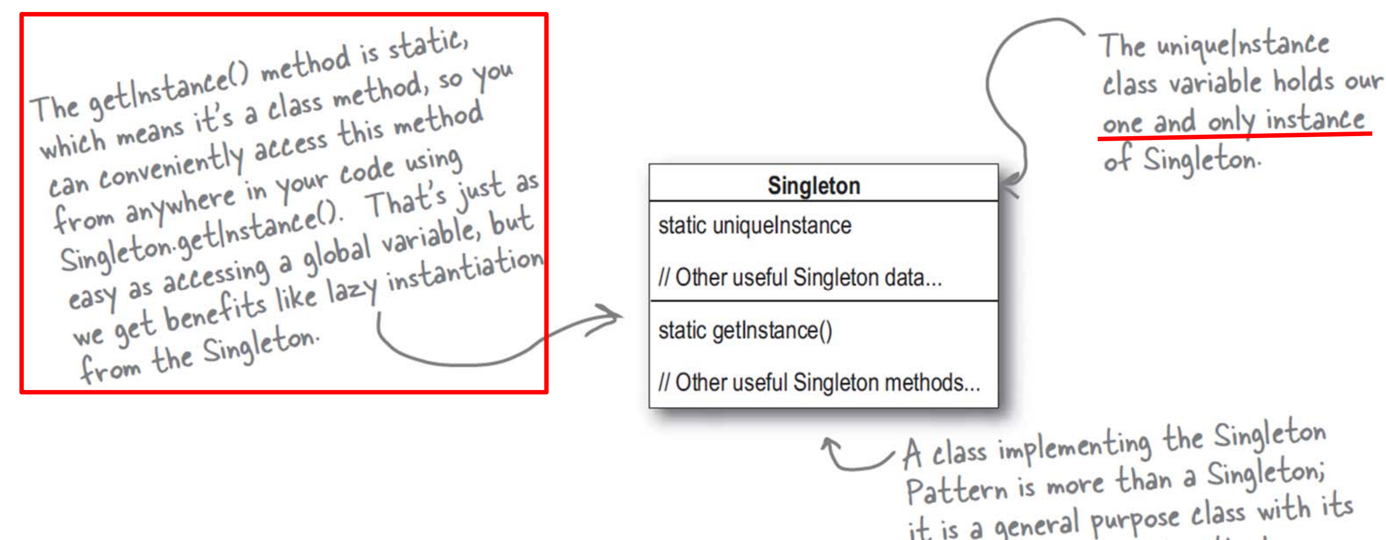
```
Creating unique instance of Chocolate Boiler
Returning instance of Chocolate Boiler
```

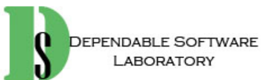


Singleton Pattern defined

- **No big surprises there. But what's really going on here?**
- We're taking a class and letting it manage a single instance of itself. We're also preventing any other class from creating a new instance on its own. To get an instance, you've got to go through the class itself.
- We're also providing a global access point to the instance: whenever you need an instance, just query the class and it will hand you back the single instance. As you've seen, we can implement this so that the Singleton is created in a lazy manner, which is especially important for resource-intensive objects.

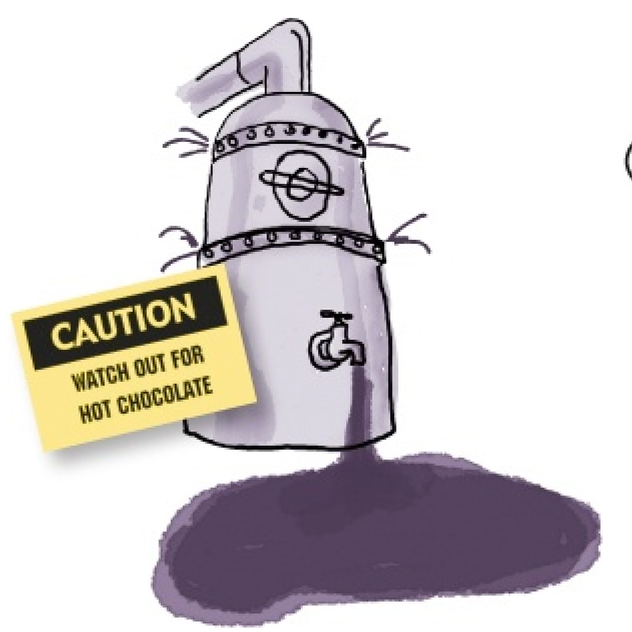
The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.





We have a problem…

- **It looks like the Chocolate Boiler has let us down;** despite the fact we improved the code using Classic Singleton, somehow the ChocolateBoiler's fill() method was able to start filling the boiler even though a batch of milk and chocolate was already boiling!



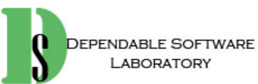
We don't know what happened! The new Singleton code was running fine. The only thing we can think of is that we just added some optimizations to the Chocolate Boiler Controller that makes use of multiple threads.

A multithreading problem

BE the JVM

We have two threads, each executing this code. Your job is to play the JVM and determine whether there is a case in which two threads might get ahold of different boiler objects. Hint: you really just need to look at the sequence of operations in the getInstance() method and the value of uniqueInstance to see how they might overlap. Use the code Magnets to help you study how the code might interleave to create two boiler objects.

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

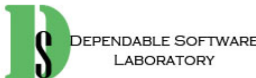


BE the JVM

Thead One	Thead Two	Value of uniqueInstance
`public static ChocolateBoiler getInstance() {`		null
	`public static ChocolateBoiler getInstance() {`	null
`if (uniqueInstance == null) {`		
	`if (uniqueInstance == null) {`	null
`uniqueInstance = new ChocolateBoiler();`		<object1>
`return uniqueInstance;`		<object1>
	`uniqueInstance = new ChocolateBoiler();`	<object2>
	`return uniqueInstance;`	<object2>

Uh oh, this doesn't look good!

Two different objects are returned! We have two ChocolateBoiler instances!!!



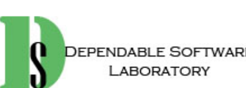
Dealing with multithreading

- Our multithreading woes are almost trivially fixed by making getInstance() a synchronized method:

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

    // other useful instance variables here

    private Singleton() {}

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

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we 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.

I agree this fixes the problem. But synchronization is expensive; is this an issue?



Can we Improve multithreading?

- **For most Java applications, we obviously need to ensure that the Singleton works in the presence of multiple threads.** But it is expensive to synchronize the getInstance() method, so what do we do?
- Well, we have a few options...

1. Do nothing if the performance of getInstance() isn't critical to your application.

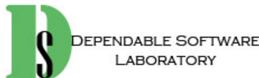
2. Move to an eagerly created instance rather than a lazily created one.

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

    private Singleton() {}

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

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

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

  - Using this approach, we rely on the JVM to create the unique instance of the Singleton when the class is loaded.
  - The JVM guarantees that the instance will be created before any thread accesses the static uniqueInstance variable.



3. Use "double-checked locking" to reduce the use of synchronization in getInstance().

- With double-checked locking, we first check to see if an instance is created, and if not, THEN we synchronize. This way, we only synchronize the first time through, just what we want.

```
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 a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

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

Watch it!

Double-checked locking doesn't work in Java 1.4 or earlier!

Unfortunately, in Java version 1.4 and earlier, many JVMs contain implementations of the volatile keyword that allow improper synchronization for double-checked locking. If you must use a JVM other than Java 5, consider other methods of implementing your Singleton.

there are no Dumb Questions

Q: For such a simple pattern consisting of only one class, Singletons sure seem to have some problems.

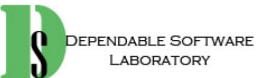
A: Well, we warned you up front! But don't let the problems discourage you; while implementing Singletons *correctly* can be tricky, after reading this chapter you are now well informed on the techniques for creating Singletons and should use them wherever you need to control the number of instances you are creating.

Q: Can't I just create a class in which all methods and variables are defined as static? Wouldn't that be the same as a Singleton?

A: Yes, if your class is self-contained and doesn't depend on complex initialization. However, because of the way static initializations are handled in Java, this can get very messy, especially if multiple classes are involved. Often this scenario can result in subtle, hard to find bugs involving order of initialization. Unless there is a compelling need to implement your "singleton" this way, it is far better to stay in the object world.

Q: What about class loaders? I heard there is a chance that two class loaders could each end up with their own instance of Singleton.

A: Yes, that is true as each class loader defines a namespace. If you have two or more classloaders, you can load the same class multiple times (once in each classloader). Now, if that class happens to be a Singleton, then since we have more than one version of the class, we also have more than one instance of the Singleton. So, if you are using multiple classloaders and Singletons, be careful. One way around this problem is to specify the classloader yourself.



Q: I've always been taught that a class should do one thing and one thing only. For a class to do two things is considered bad OO design. Isn't a Singleton violating this?

A: You would be referring to the "One Class, One Responsibility" principle, and yes, you are correct, the Singleton is not only responsible for managing its one instance (and providing global access), it is also responsible for whatever its main role is in your application. So, certainly it can be argued it is taking on two responsibilities. Nevertheless, it isn't hard to see that there is utility in a class managing its own instance; it certainly makes the overall design simpler. In addition, many developers are familiar with the Singleton pattern as it is in wide use. That said, some developers do feel the need to abstract out the Singleton functionality.

Q: I wanted to subclass my Singleton code, but I ran into problems. Is it okay to subclass a Singleton?

A: One problem with subclassing Singleton is that the constructor is private. You can't extend a class with a private constructor. So, the first thing you'll have to do is change your constructor so that it's public or protected. But then, it's not *really* a Singleton anymore, because other classes can instantiate it.

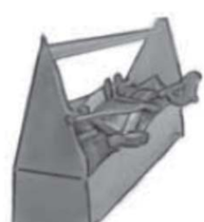
If you do change your constructor, there's another issue. The implementation of Singleton is based on a static variable, so if you do a straightforward subclass, all of your derived classes will share the same instance variable. This is probably not what you had in mind. So, for subclassing to work, implementing registry of sorts is required in the base class.

Before implementing such a scheme, you should ask yourself what you are really gaining from subclassing a Singleton. Like most patterns, the Singleton is not necessarily meant to be a solution that can fit into a library. In addition, the Singleton code is trivial to add to any existing class. Last, if you are using a large number of Singletons in your application, you should take a hard look at your design. Singletons are meant to be used sparingly.

Q: I still don't totally understand why global variables are worse than a Singleton.

A: In Java, global variables are basically static references to objects. There are a couple of disadvantages to using global variables in this manner. We've already mentioned one: the issue of lazy versus eager instantiation. But we need to keep in mind the intent of the pattern: to ensure only one instance of a class exists and to provide global access. A global variable can provide the latter, but not the former. Global variables also tend to encourage developers to pollute the namespace with lots of global references to small objects. Singletons don't encourage this in the same way, but can be abused nonetheless.

Tools for your design toolbox



OO Basics

straction

capsulation

ymorphism

eritance

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Classes should be open for extension but closed for modification.

Depend on abstractions. Do not depend on concrete classes.

OO Patterns

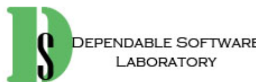
Singleton – Ensure a class only has one instance and provide a global point of access to it.

When you need to ensure you only have one instance of a class running around your application, turn to the Singleton.

BULLET POINTS

- 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 multithreaded applications (and we should consider all applications multithreaded!).
- Beware of the double-checked locking implementation; it is not thread-safe in versions before Java 2, version 5.
- Be careful if you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.
- If you are using a JVM earlier than 1.2, you'll need to create a registry of Singletons to defeat the garbage collector.

KU KONKUK UNIVERSITY

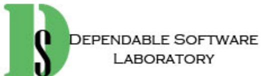

Dependable Software Laboratory

“Encapsulating Algorithms”

8. The Template Method Pattern

The Template Method Pattern

- **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

Starbuzz Coffee Barista Training Manual

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

Starbuzz Coffee Recipe

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

Starbuzz Tea Recipe

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

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

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

Whipping up some coffee and tea classes

- Here's the coffee:

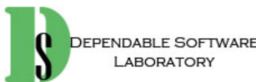
Here's our Coffee class for making coffee.

```
public class Coffee {

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

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

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

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

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

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

Each of the steps is implemented as a separate method.

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



- And now the tea…

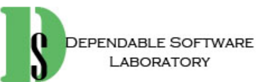
```
public class Tea {

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

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

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

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

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

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

These two methods are specialized to Tea.

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

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?



May I abstract your Coffee and Tea?

- Your first cut might have looked something like this:

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

CaffeineBeverage
- *prepareRecipe()*
- boilWater()
- pourInCup()

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

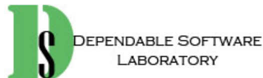
Each subclass implements its own recipe.

Coffee
- prepareRecipe()
- brewCoffeeGrinds()
- addSugarAndMilk()

Tea
- prepareRecipe()
- steepTeaBag()
- addLemon()

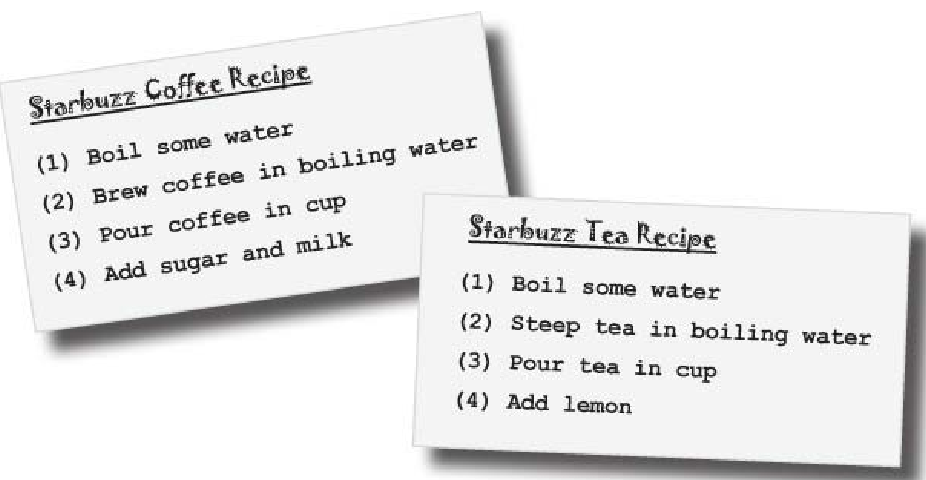
Each subclass overrides the prepareRecipe() method and implements its own recipe.

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



Taking the design further…

- What **else** do Coffee and Tea have in common? Let's start with the recipes.

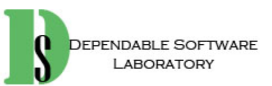


- Notice that both recipes follow **the same algorithm**: **prepareRecipe()**

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.



Abstracting prepareRecipe( )

Coffee

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

Tea

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

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

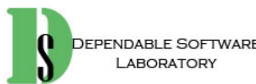
CaffeineBeverage is abstract, just like in the class design.

```
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).

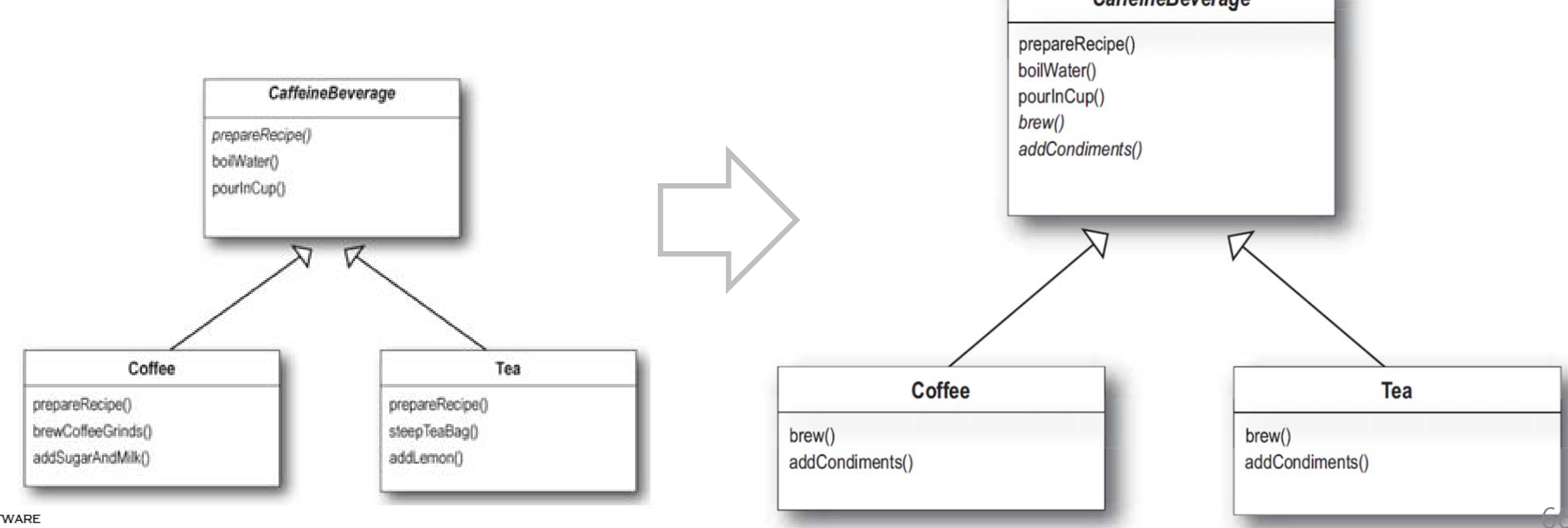


```java
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}

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

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

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

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



What have we done?

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

Tea

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

Coffee

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

generalize

generalize

Caffeine Beverage

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

relies on subclass for some steps

relies on subclass for some steps

Tea subclass

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

Coffee subclass

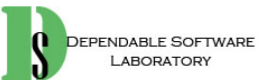
2. Brew the coffee grinds
4. Add sugar and milk

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

Meet the Template Method

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

```
public abstract class CaffeineBeverage {

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

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }

}
```

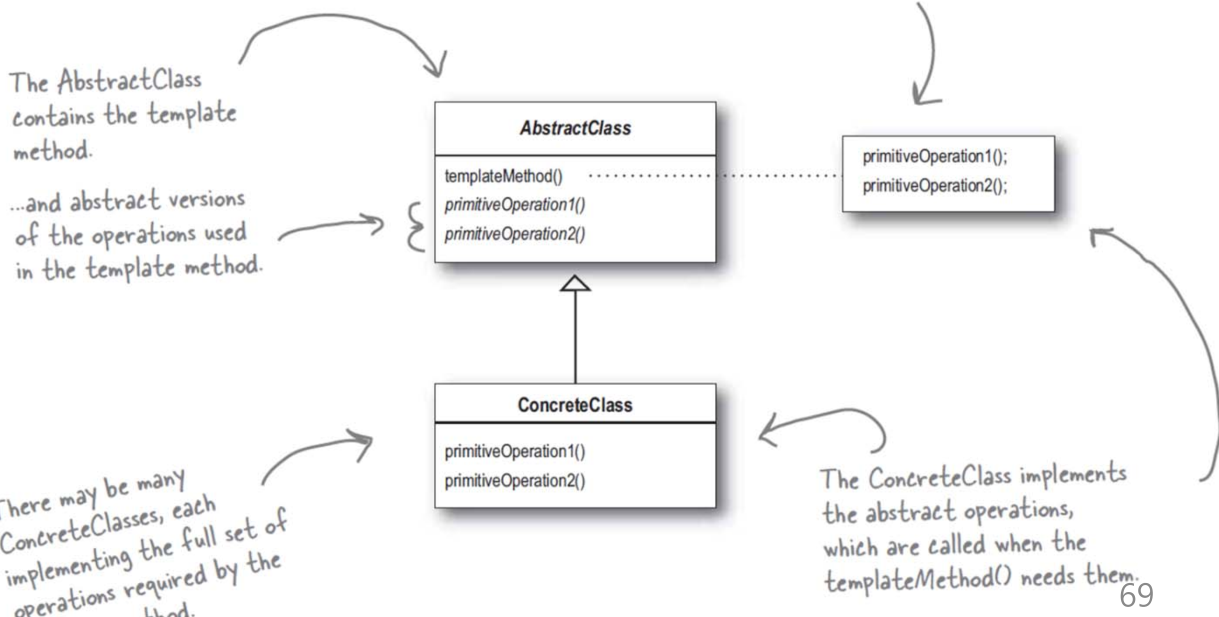



Template Method Pattern defined

- This pattern is all about creating a template for an algorithm.

> **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.

- 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.



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...

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() {}

}
```

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.

```
public abstract class CaffeineBeverageWithHook {

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

    abstract void brew();

    abstract void addCondiments();

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

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

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

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

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

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

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.

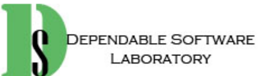
Using the hook

```java
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

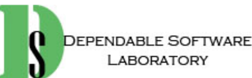
```
public class BeverageTestDrive {
    public static void main(String[] args) {

        TeaWithHook teaHook = new TeaWithHook();          // Create a tea.
        CoffeeWithHook coffeeHook = new CoffeeWithHook(); // A coffee.

        System.out.println("\nMaking tea...");            // And call prepareRecipe()
        teaHook.prepareRecipe();                          // on both!

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

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

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.



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.

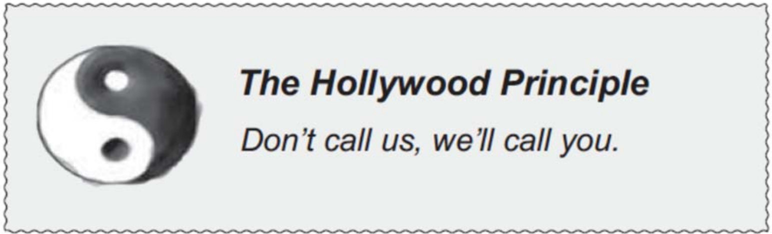
Q: It seems like I should keep my abstract methods small in number, otherwise it will be a big job to implement them in the subclass.

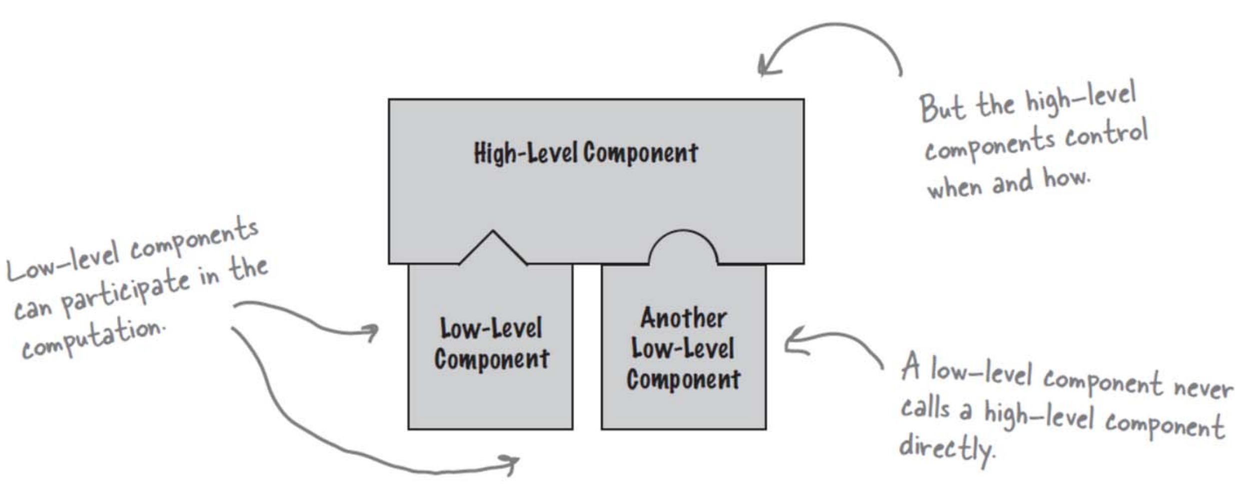
A: That's a good thing to keep in mind when you write template methods. Sometimes this can be done by not making the steps of your algorithm too granular. But it's obviously a trade off: the less granularity, the less flexibility.

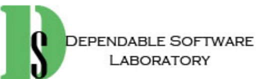
Remember, too, that some steps will be optional; so you can implement these as hooks rather than abstract classes, easing the burden on the subclasses of your abstract class.

The Hollywood Principle

- 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.
- 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

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

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

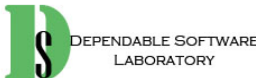
CaffeineBeverage
prepareRecipe() boilWater() pourInCup() *brew()* *addCondiments()*

Coffee
brew() addCondiments()

Tea
brew() addCondiments()

The subclasses are used simply to provide implementation details.

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



there are no Dumb Questions

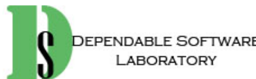
Q: How does the Hollywood Principle relate to the Dependency Inversion Principle that we learned a few chapters back?

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.

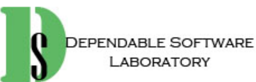
The Hollywood Principle gives us a technique for creating designs that allow low-level structures to interoperate while preventing other classes from becoming too dependent on them.

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.



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

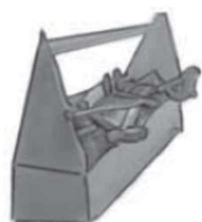


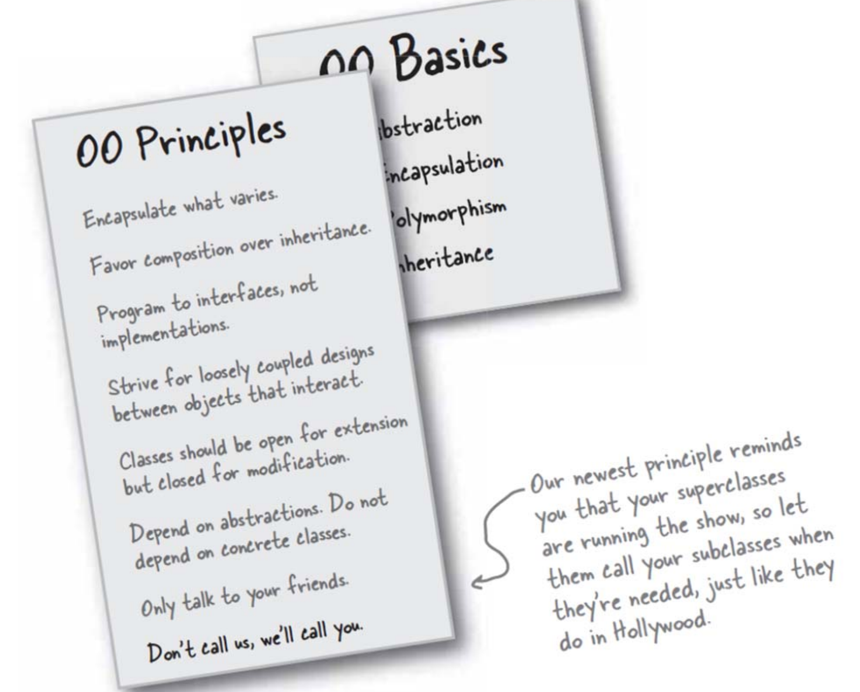
Template Methods in the Wild

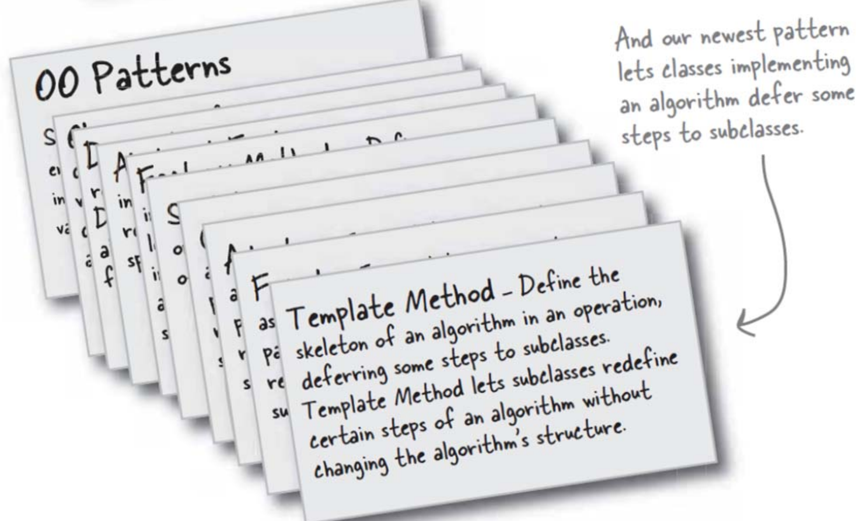
- This pattern shows up so often because it's a great design tool for creating **frameworks**, where the framework controls how something gets done, but leaves you (the person using the framework) to specify your own details about what is actually happening at each step of the framework's algorithm.
  - Sorting
  - Java JFrame
  - Applet

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

Tools for your design toolbox







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.