The Observer Pattern Using Java 8

Learn about the observer pattern using modern Java: What it is, specializations, common naming conventions, and more!

The Observer Pattern, or the Publish-Subscribe (Pub-Sub) Pattern, is a classic design pattern codified by the Gang of Four (GoF) Design Patterns book in 1994 (pg. 293-313). Although this pattern has quite a long and storied history, it is still applicable in a wide range of contexts and scenarios, and has even become an integral part in the Standard Java Library. While there are numerous useful articles on the topic of the Observer Pattern, and its implementation in Java, many focus on the strict implementation of the pattern in Java, rather than on the idiosyncrasies and common issues developers using the Observer Pattern in Java will experience.

This article is written with the intent of filling this gap: This article illustrates a simple implementation of the Observer Pattern using modern Java 8 constructs and uses this basic implementation to explore some of the more sophisticated problems associated with this classic pattern, including the use of anonymous inner classes and lambdas, thread safety, and non-trivial and temporally-expensive observer implementations. This article is by no means comprehensive and many of the intricacies of this pattern are beyond the scope of a single article, but when complete, the reader will understand not only the Observer Pattern, but its common peculiarities in modern Java and how to approach some of the most common issues that arise when implementing the Observer Pattern in Java.

The Observer Pattern

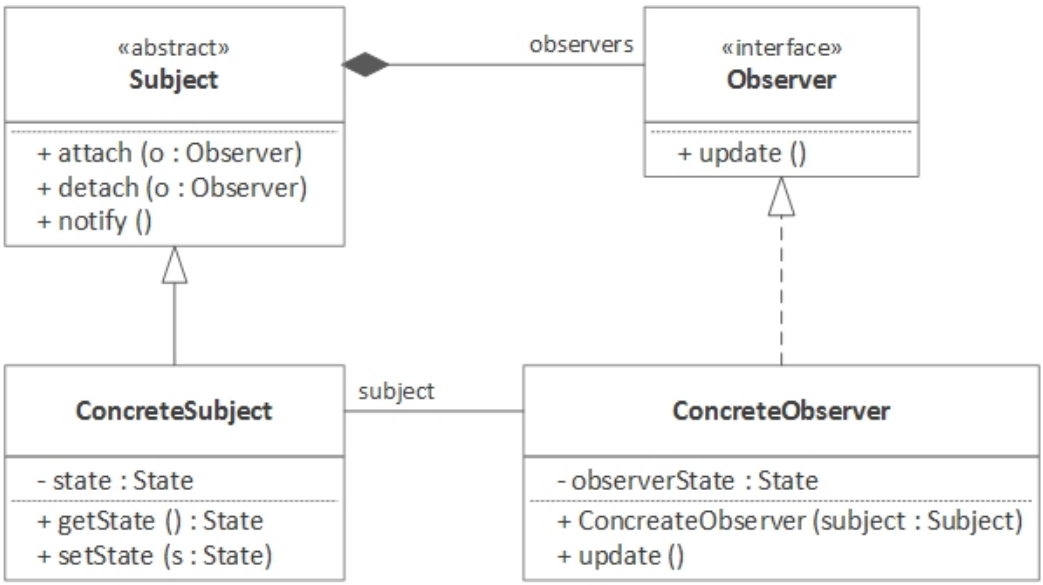
According to the classic definition coined by the GoF, the intent of the Observer Pattern is to

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

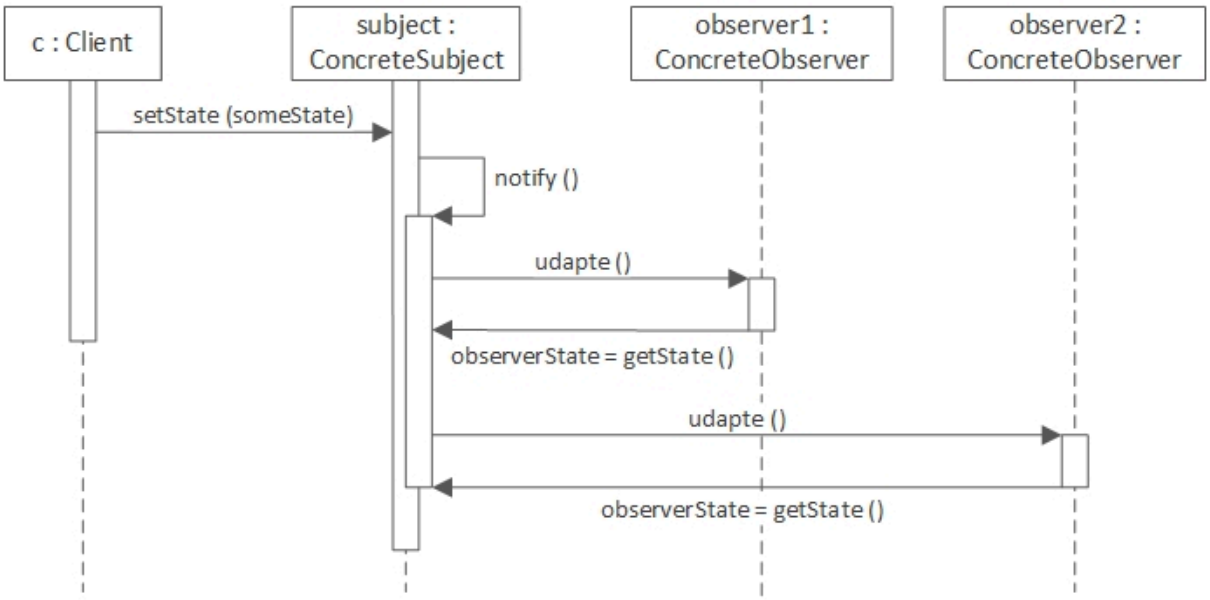
What does this mean? In many software applications, the state of one object is dependent on another. For example, if an application focuses on numeric data processing, this data may be displayed on a Graphical User Interface (GUI) using a spreadsheet or a graph, or both: When the underlying numeric data is updated, the corresponding GUI components should be updated accordingly. The crux of this problem is how to allow the GUI components to be updated when a change is made to the underlying numerical data without coupling the GUI components directly to the underlying numeric data.

A naive and inextensible solution would be to provide the object that manages the underlying numerical data with a reference to the spreadsheet and graph GUI components, allowing the object to notify the GUIs components when the numeric data changes. This simple solution quickly proves its inability to cope with more complex applications when more GUI components are added. For example, if there are 20 GUI components that depend on the numeric data, the object that manages this numeric data would be required to maintain a reference to these 20 GUI components. When the number of objects that depend on the data of interest grows, the coupling between the manager of the data and the objects dependent on this data becomes more unrulely.

A better solution would be to allow objects to **register** for updates to the data of interest, allowing the manager of this data to **notify** these registered objects when the data of interest changes. Informally, the objects interested in the data tell the manager, "Let me know when a change occurs to the data." Furthermore, after an object has been registered for updates to the data of interest, it can be **unregistered**, ensuring that the manager no longer notifies the object of changes that occur to the data of interest. In the context of the original GoF definition, the object that is registered for updates is called an **observer**, the manager of the data of interest is called a **subject**, the data of interest is called the **state** of the subject, the process of registering an observer is called **attaching**, and the process of unregistering an observer is called **detaching**. As previously stated, this pattern is also called the Publish-Subscribe pattern, since the clients subscribe observers to a subject and when updates are made to the state of the subject, the subject publishes these updates to the subscribers (this design pattern is expanded to a common architecture, called the Publish-Subscribe Architecture). Pulling these concepts together into a single picture, we obtain the following class diagram:



In order to receive updates to the change in state, a `ConcreteObserver` is created, and a reference to the `ConcreteSubject` which it is observing is passed to its constructor. This provides the `ConcreteObserver` with a reference to the `ConcreteSubject`, from which the `State` will be obtained when the observer is notified of a change to the state. Simply, the `ConcreteObserver` will be notified of an update to the subject, at which time, the `ConcreteObserver` will use the reference to the `ConcreteSubject` obtained through its constructor to obtain the `State` of the `ConcreteSubject`, ultimately storing the retreived `State` object under the `observerState` attribute of the `ConcreteObserver`. This process is visualized in the sequence diagram below:



Specializations of the Classic Pattern

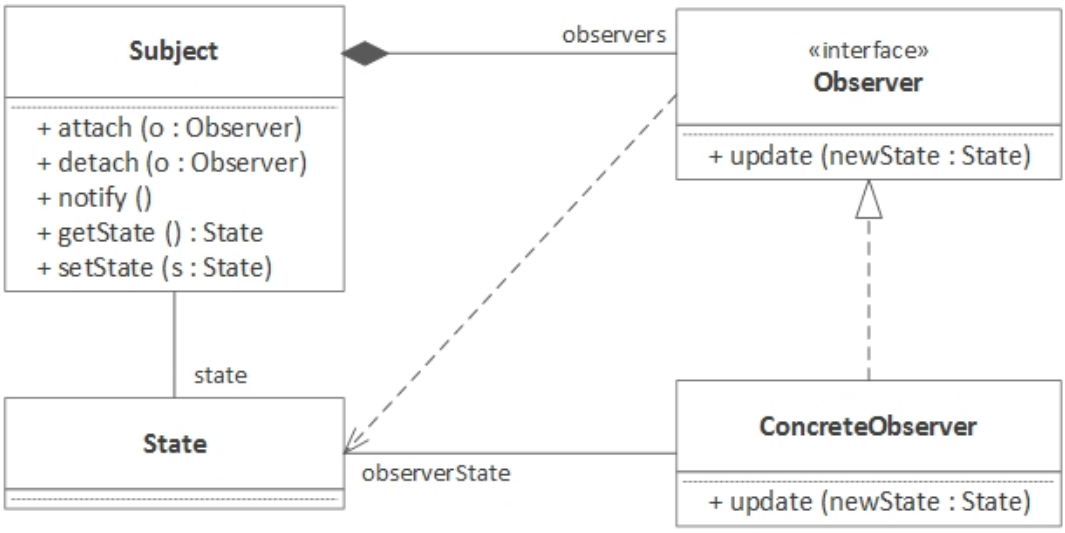
While the classic Observer Pattern provides a general case, there are common specializations made for this pattern. Two of the most common specializations are

1. Providing the `State` object as a parameter to the `update` method call made to the `Observer`. In the classic case, when an observer is notified of a change to the state of the subject, the observer is responsible for obtaining the state of the subject directly from the subject. This requires the observer to maintain a reference to the subject from whom the state will be obtained (a circular reference, where the `ConcreteSubject` maintains a reference to the `ConcreteObserver` in its list of registered listeners, and the `ConcreteObserver` maintains a reference to the `ConcreteSubject` that can be used to obtain the `Subject` of the `ConcreteSubject`). Apart from obtaining the updated state, the observer has no real association with the subject to which it is registered (all the observer cares about is the `State` object, not the `Subject` itself). In most cases, this means that we are imposing an unnatural association between a `ConcreteObserver` and some `ConcreteSubject`. Instead, by passing the `State` object to the

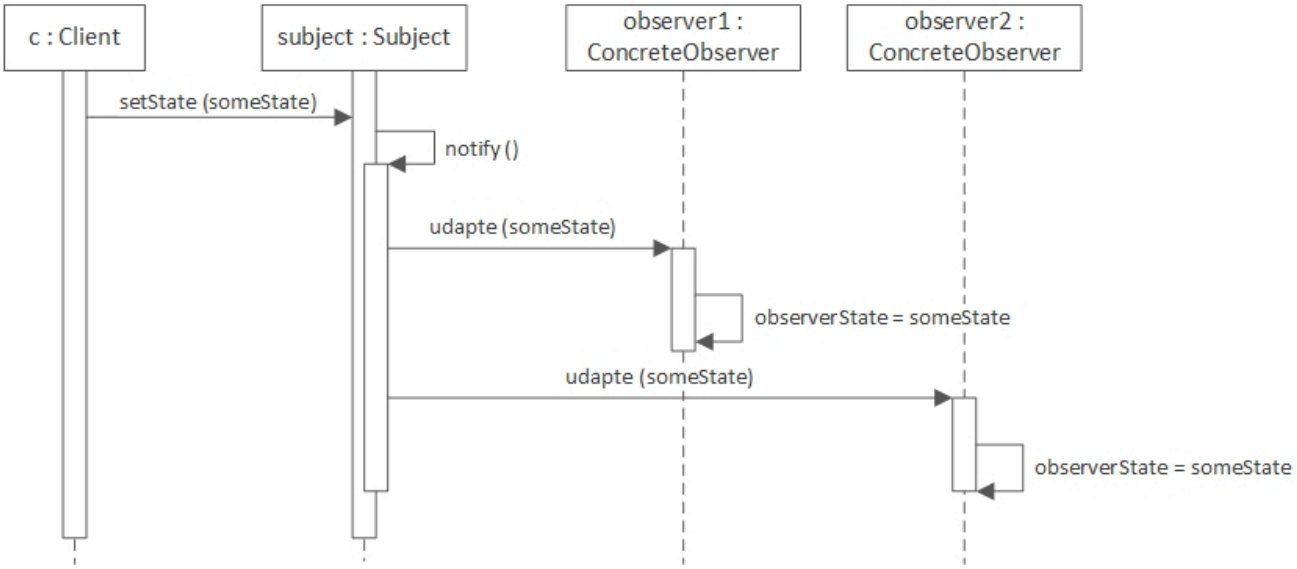
`ConcreteObserver` when a `ConcreteSubject` calls the `update` method, a `ConcreteObserver` is no longer required to maintain an association with the `ConcreteSubject`. This association is changed to an association between the `ConcreteObserver` and the `State` and to a weaker dependency between the `Observer` and the `State` (for more information on the differences between associations and dependencies, see Martin Fowler's article on the topic). For a visualization of this concept, see the `ConcreteObserver` in the illustration below.

2. Collapsing the `Subject` abstract class and the `ConcreteSubject` into a single`Subject` class. In many cases, the use of an abstract class for the `Subject` does not provide extra flexibility or extensibility and therefore, it simplifies a design to collapse this abstract class and concrete class into a single, concrete class.

The combination of these two specializations results in the following, simplified class diagram:



Not only is the static class structure of this pattern simplified by these specializations, but the interaction between these classes is also simplified, as well. The resulting sequence diagram is illustrated below:



Another oft made specialization is the removal of the `observerState` member variable from the `ConcreteObserver` class. In some cases, a concrete observer does not need to store the new state of the `Subject`, but rather, simply view the state of the subject at the time the state is updated. For example, if an observer prints member variables of the updated state to standard output, the `observerState` member variable can be removed. The removal of this member variable removes the association between the `ConcreteObserver` and the `State` class.

More Common Naming Conventions

Although the classic pattern (even with the above specialiazations) uses the terms **attach**, **detach**, and **observer**, many Java implementations of this pattern use a different lexicon. Instead, the terms **register**, **unregister**, and **listener**, respectively, are often used. It is important to note that the term **state** is a general term used for any object

that an listener should be notified of upon change. The actual name of this state object will depend on the context in which the Observer Pattern is used. For example, if the Observer Pattern is used in a context where listeners are notified of events occuring, the term **event** will commonly be used: "The registered listeners were notified of the button press event." The signature of the update method would resemble `update (event : Event)` instead.

The name of the subject rarely contains the `Subject` when used in practice. For example, suppose we were to create a zoo application, where a set of listeners were to be registered with a `Zoo` class and notified each time a new animal is added to the zoo. The subject in this example would be the `Zoo` class, and in keeping with the terms used in the problem domain, would not include the term `Subject` (i.e., would *not* be called `ZooSubject`, since a collection of animals is simply called a zoo in the problem domain).

In many cases, listeners are named according to their purpose, suffixed by the `Listener`. For example, the listener described in the previous paragraph would commonly be named `AnimalAddedListener`. Similarly, the `register`, `unregister`, and `notify` methods are often suffixed by the type of the listener that is registered, unregistered, or notified, respectively. For example, a register method for the `AnimalAddedListener` will commonly be called `registerAnimalAddedListener` and have the signature `registerAnimalAddedListener (listener : AnimalAddedListener)`. Likewise, the name of the unregister and notify methods will resemble `unregisterAnimalAddedListener` and `notifyAnimalAddedListeners` (note the trailing *s*, since the notify method is dealing with a collection of listeners, not a single listener).

At first, the inclusion of the listener type in the register, unregister, and notify method names may appear bloated, but it is often the case that a single subject will register more than one type of listener. For example, a `Zoo` may also allow clients to register listeners to be notified when an animal is removed from the `Zoo`. In this case, there would need to be two register methods: (1) `registerAnimalAddedListener` and (2) `registerAnimalRemovedListener`. In this context, the type of the listener acts as a qualifier that denotes the type of the observer that is being registered (a single `registerListener` method could be created and overloaded to accept each type of listener, but using named qualification provides a more explicit illustration of the type of listener being registered; in any case, overloading of the register method is *not* a common convention).

Although not always used, another common idiom is to use the `on` prefix, rather than the `update` prefix in the `Observer` interface. For example, instead of `updateAnimalAdded`, the update method may be named `onAnimalAdded`. This more common when a listener is expecting to receive notifications in a sequence, such as the addition of an animal to a list, rather than the updating of a single data point, such as the name of an animal (i.e., notification of new objects rather updates to a single piece of data).

Throughout the remainder of this article, the Java-style notation described above will be used. Although the notation used does not change the actual design or implementation of the system, it is important to use a lexicon that is commonly known by other developers, and therefore, it is important to become accustomed to the Java-style Observer Pattern notation described in this section. In the following section, we will take the concepts described above and implement them in a basic Java 8 example.

A Simple Implementation

In keeping with the zoo example described in the previous section, we will implement a simple system that illustrates the fundamentals of the Observer Pattern using modern Java (Java 8 collection and stream Application Programmer Interfaces, or APIs). The problem description is

> *Create a zoo system that allows a client to register and unregister listeners that will be notified when an animal is added to the zoo. In addition, create a concrete listener that will print the name of the animal that is added to the zoo.*

Using the knowledge we obtained in the above description of the Observer Pattern, we need to create 4 classes, plus a main class that will drive the application:

1. A `Zoo` class that will act as the subject, responsible for storing a list of the animals in the zoo and notifying a collection of registered listeners each time a new animal is added to the zoo
2. An `Animal` class that represents an animal with a name
3. An `AnimalAddedListener` class that acts as the `Observer` interface
4. A `PrintNameAnimalAddedListener` concrete observer class that prints the name of the animal that is added to the zoo

Starting with the Animal class, we create a simple Plain Old Java Object (POJO) that contains a single name member variable, and corresponding constructor, getter, and setter:

```
public class Animal {

    private String name;

    public Animal (String name) {
        this.name = name;
    }

    public String getName () {
        return this.name;
    }

    public void setName (String name) {
        this.name = name;
    }
}
```

Using this class to represent an animal, we can now create the `AnimalAddedListener` interface:

```
public interface AnimalAddedListener {
    public void onAnimalAdded (Animal animal);
}
```

Due to the simplicity of the two previous classes, a detail description is unwarranted. Using these foundational classes, the `Zoo` class can be created:

```
public class Zoo {

    private List<Animal> animals = new ArrayList<>();
    private List<AnimalAddedListener> listeners = new ArrayList<>();

    public void addAnimal (Animal animal) {
        // Add the animal to the list of animals
        this.animals.add(animal);

        // Notify the list of registered listeners
        this.notifyAnimalAddedListeners(animal);
    }

    public void registerAnimalAddedListener (AnimalAddedListener listener) {
```

```
15
        // Add the listener to the list of registered listeners
16
        this.listeners.add(listener);
17
    }
18

19
    public void unregisterAnimalAddedListener (AnimalAddedListener listener) {
20
        // Remove the listener from the list of the registered listeners
21
        this.listeners.remove(listener);
22
    }
23

24
    protected void notifyAnimalAddedListeners (Animal animal) {
25
        // Notify each of the listeners in the list of registered listeners
26
        this.listeners.forEach(listener -> listener.onAnimalAdded(animal));
27
    }
28
}
```

While this class is more complex than the previous two, it simply follows the structure and convention described in the previous section. This class contains 2 lists: (1) a list to store the animals contained in the zoo and (2) and a list to store the listeners to be notified when a new animal is added. Due to the simplified nature of both the animals and listener collections, an `ArrayList` is used. If, instead, the listeners were to be stored with some priority, another type of collection may be used (or the algorithm for registering listeners can be changed accordingly). The concrete type of the collection used to store the listeners will depend on the nature of the application and problem being solved.

The implementation of the register and unregister methods are simple delegation methods: The listener provided as an argument is added to or removed from the list of listeners, respectively. The implementation of the notify method deviates slightly from the standard signature of the Observer Pattern. Instead of accepting zero arguments, the notify method accepts a single argument: The animal object that was added. This allows the notify method to provide the listeners with a reference to the animal that was just added. Iterating through each of the listeners is performed using the `forEach` method of the streams API, which executes the`onAnimalAdded` method of each listener, supplying the newly added animal object.

In the `addAnimal` method, the supplied animal object is added to the list of animals associated with the zoo and the registered listeners are notified of the new addition. The benefit of encapsulating the logic for notifying the registered listeners within the `notifyAnimalAddedListeners` method is illustrated in the implementation of the `addAnimal` method: Regardless of the complexity of the notification process (e.g., if the animal state must be changed before notifying the registered listeners, a counter must be incremented, etc.), this logic is contained in a single, easy-to-call method, requiring only a reference to the newly added animal object.

Apart from the logic of the notify method, there is some controversy about the visibility of the notify method that must be addressed. In the classic Observer Pattern, the notify method is marked public (GoF Patterns book, pg. 301). Although public visibility is used in the classic implementation, this does not necessarily mean that public visibility should always be used. The visibility selected should reflect the intent of the application. For example, in the case of the `Zoo` implementation above, protected visibility is used, since it is not desired that any object should be able to initiate a notification of the registered observers, but it this functionality should be available to subclasses that may inherit from this class. This is not always the case: It is important to undestand who should be able to initiate a notification and set the visibility of the notify method accordingly.

Next, the `PrintNameAnimalAddedListener` class must be implemented. This class simply prints the name of the animal that was added using the standard `System.out.println` method:

```
8
1
public class PrintNameAnimalAddedListener implements AnimalAddedListener {
2

3
    @Override
```

```
4
    public void onAnimalAdded (Animal animal) {
5
        // Print the name of the newly added animal
6
        System.out.println("Added a new animal with name '" + animal.getName() + "'");
7
    }
8
}
```

Lastly, we must implement the main method that will drive the application:

```
13
1
public class Main {
2

3
    public static void main (String[] args) {
4
        // Create the zoo to store animals
5
        Zoo zoo = new Zoo();
6

7
        // Register a listener to be notified when an animal is added
8
        zoo.registerAnimalAddedListener(new PrintNameAnimalAddedListener());
9

10
        // Add an animal notify the registered listeners
11
        zoo.addAnimal(new Animal("Tiger"));
12
    }
13
}
```

This driver class simply creates a zoo object, registers a listener that prints the name of the animal, and adds a new animal, triggering the registered listener. The resulting output is:

```
1
Added a new animal with name 'Tiger'
```

Adding New Listeners

The capability of the Observer Pattern is truly demonstrated when new listeners are created and added to the subject. For example, if we wanted to create a new listener that counts the number of animals added to the zoo, *no changes* would be required for the `Zoo` class. Instead of adding a reference to the new listener in the `Zoo` class, a new concrete listener class is simply created and registered with the `Zoo` (no changes are made to the source code of the `Zoo` class). An implementation of the counting listener, `CountingAnimalAddedListener`, may resemble the follow:

```
13
1
public class CountingAnimalAddedListener implements AnimalAddedListener {
2

3
    private static int animalsAddedCount = 0;
4

5
    @Override
6
    public void onAnimalAdded (Animal animal) {
7
        // Increment the number of animals
8
        animalsAddedCount++;
9
```

```
        // Print the number of animals
        System.out.println("Total animals added: " + animalsAddedCount);
    }
}
```

An updated main method is as follows:

```
public class Main {

    public static void main (String[] args) {
        // Create the zoo to store animals
        Zoo zoo = new Zoo();

        // Register listeners to be notified when an animal is added
        zoo.registerAnimalAddedListener(new PrintNameAnimalAddedListener());
        zoo.registerAnimalAddedListener(new CountingAnimalAddedListener());

        // Add an animal notify the registered listeners
        zoo.addAnimal(new Animal("Tiger"));
        zoo.addAnimal(new Animal("Lion"));
        zoo.addAnimal(new Animal("Bear"));
    }
}
```

The resulting output is

```
Added a new animal with name 'Tiger'
Total animals added: 1
Added a new animal with name 'Lion'
Total animals added: 2
Added a new animal with name 'Bear'
Total animals added: 3
```

Any number of listeners can be created in the same manner without changing the existing logic (apart from the code where the listeners are registered). This extensibility is a direct result of the association between the `Subject` class and the `Observer` interface, rather than between the `Subject` and a `ConcreteObserver`. So long as this interface does not change, the `Subject` that interacts with the `Observer` interface is not required to change.

Anonymous Inner Classs, Lambda & `this` Listener Registration

One of the major improvements of Java 8 is the addition of functional characteristics, such as lambda functions. Prior to lambda functions (and still used in many legacy applications), Java provided similar functionality with anonymous inner classes. In the context of the Observer Pattern, each allows for new listeners to be created without

explicitly creating new concrete observer classes. For example, the `PrintNameAnimalAddedListener` class can be implemented using an anonymous inner class in the main method as follows:

```
public class Main {

    public static void main (String[] args) {
        // Create the zoo to store animals
        Zoo zoo = new Zoo();

        // Register listeners to be notified when an animal is added
        zoo.registerAnimalAddedListener(new AnimalAddedListener() {
            @Override
            public void onAnimalAdded (Animal animal) {
                // Print the name of the newly added animal
                System.out.println("Added a new animal with name '" + animal.getName() + "'");
            }
        });

        // Add an animal notify the registered listeners
        zoo.addAnimal(new Animal("Tiger"));
    }
}
```

Similarly, a lambda function can be used to accomplish the same functionality:

```
public class Main {

    public static void main (String[] args) {
        // Create the zoo to store animals
        Zoo zoo = new Zoo();

        // Register listeners to be notified when an animal is added
        zoo.registerAnimalAddedListener(
            (animal) -> System.out.println("Added a new animal with name '" + animal.getName() + "'")
        );

        // Add an animal notify the registered listeners
        zoo.addAnimal(new Animal("Tiger"));
    }
```

```
15
}
```

Note that a lambda function can only be used if the listener interface has a single method (see this Stackoverflow post for more information). Although this constraint may appear to be restrictive, a large number of the listeners used in pratical cases use a single method (such as the `AnimalAddedListener` used in this example). If a listener interface has more than a single method, then an anonymous inner class can still be used.

The ability to register implicitly created listeners introduces a problem: Since the object is created within the scope of the registration call, storing a reference to the concrete listener may not be possible. This means that a listener registered using a lambda function or anonymous inner class may not be unregistered (since the unregister method requires a reference to a listener that has been previously registered). A simple solution to this problem is to return a reference to the registered listener in the `registerAnimalAddedListener` method. Using this technique, client code may store a refernce to the listener created in place which can be used to unregister the listener at a later time. This new method would resemble the following:

```
6
1
public AnimalAddedListener registerAnimalAddedListener (AnimalAddedListener listener) {
2
    // Add the listener to the list of registered listeners
3
    this.listeners.add(listener);
4

5
    return listener;
6
}
```

The client code interacting with this redesigned method would resemble the following:

```
22
1
public class Main {
2

3
    public static void main (String[] args) {
4
        // Create the zoo to store animals
5
        Zoo zoo = new Zoo();
6

7
        // Register listeners to be notified when an animal is added
8
        AnimalAddedListener listener = zoo.registerAnimalAddedListener(
9
            (animal) -> System.out.println("Added a new animal with name '" + animal.getName() + "'")
10
        );
11

12
        // Add an animal notify the registered listeners
13
        zoo.addAnimal(new Animal("Tiger"));
14

15
        // Unregister the listener
16
        zoo.unregisterAnimalAddedListener(listener);
17

18
        // Add another animal, which will not print the name, since the listener
19
        // has been previously unregistered
20
        zoo.addAnimal(new Animal("Lion"));
```

```
    }
}
```

The resulting output will only print `Added a new animal with name 'Tiger'`, since the listener was unregistered prior to the second animal being added:

```
Added a new animal with name 'Tiger'
```

If a more complex registration technique is required, the registration method may also return a receipt class that can be used to unregister the listener. For example

```
public class AnimalAddedListenerReceipt {

    private final AnimalAddedListener listener;

    public AnimalAddedListenerReceipt (AnimalAddedListener listener) {
        this.listener = listener;
    }

    public final AnimalAddedListener getListener () {
        return this.listener;
    }
}
```

The subject code would now use a registration method that returns a receipt and an unregistration method that accepts a receipt as a parameter. The new zoo implementation would resemble the following:

```
public class ZooUsingReceipt {

    // ...Existing attributes and constructor...

    public AnimalAddedListenerReceipt registerAnimalAddedListener (AnimalAddedListener listener) {
        // Add the listener to the list of registered listeners
        this.listeners.add(listener);

        return new AnimalAddedListenerReceipt(listener);
    }

    public void unregisterAnimalAddedListener (AnimalAddedListenerReceipt receipt) {
        // Remove the listener from the list of the registered listeners
        this.listeners.remove(receipt.getListener());
```

```
    }

    // ...Existing notification method...
}
```

While the receipt implementation provided above is trivial, this mechanism allows other information to be stored that can later be used for unregistering a listener (i.e., if the unregistration algorithm depends on the state of the subject *at the time the listener was registered*). If the unregistration algoritm simple requires a reference to the previously registered listener, the receipt technique may be burdensome and should be avoided.

Except for the most complex concrete listeners, the most common means of registering a listener is either through a lambda function or anonymous inner class. One exception to this rule is an idiosyncrasy where a class that contains the subject implements the observer interface and registers a listener with the contained subject using the `this` reference. For example:

```
public class ZooContainer implements AnimalAddedListener {

    private Zoo zoo = new Zoo();

    public ZooContainer () {
        // Register this object as a listener
        this.zoo.registerAnimalAddedListener(this);
    }

    public Zoo getZoo () {
        return this.zoo;
    }

    @Override
    public void onAnimalAdded (Animal animal) {
        System.out.println("Added animal with name '" + animal.getName() + "'");
    }

    public static void main (String[] args) {
        // Create the zoo container
        ZooContainer zooContainer = new ZooContainer();

        // Add an animal notify the innerally notified listener
        zooContainer.getZoo().addAnimal(new Animal("Tiger"));
    }
}
```

This method is often used for brevity and is arguably a code smell, but nonetheless, a modern Java developer is likely to come across this method of registering a listener during his or her tenure; therefore, it is important to understand how this example works. Since `ZooContainer` implements the `AnimalAddedListener` interface, an instance (object) of the `ZooContainer` class is eligable to be registered as a `AnimalAddedListener`. Within the `ZooContainer` class, the `this` reference represents the current object (and therefore, an object of type `ZooContainer`), and thus, can be used as a `AnimalAddedListener`.

In general, it is not a requirement that a container class be used to accomplish this technique. Instead, the class that implements the listener interface need only have access to the registration method of the subject and simply provide a `this` reference as the listener object to register. In the following section, we will address the difficulties that arise in the common mutlithreaded environment and how to address some of the most common multithreaded use cases.

Thread-Safe Implementation

While the previous section presents a complete, albeit simple, implementation of the Observer Pattern in moden Java, it lacks a key characteristic: Thread-safety. A large portion of applications built using Java are multithreaded applications, and what's more, many of the contexts in which the Observer Pattern is used pair closely with multithreaded or asynchronous systems. For example, if an external service updates its database and a notification is asynchronously sent to an application, the application can use the Observer Pattern to notify local components of the update, rather than having each component directly register with the external service (a local component can act as a proxy for the external service).

Thread-safety for the Observer Pattern focuses largely on the subject of the pattern, since thread contention can occur when altering the collection containing the registered listeners. For example, if one thread attempts to add a new listener while another thread is adding a new animal (which will trigger a notification of all registered listeners). Depending on the order, the first thread may or may not register the new listener prior to each of the registered listeners being notified of the new animal. This is a classic case of a race-condition and tips us off as developers that some mechanism is required to ensure thread-safety.

A naive implementation of thread-safety would make each of the methods that access or alter the list of registered listeners `synchronized`, as such:

```
3
1
public synchronized AnimalAddedListener registerAnimalAddedListener (AnimalAddedListener listener) { /*...*/ }
2
public synchronized void unregisterAnimalAddedListener (AnimalAddedListener listener) { /*...*/ }
3
public synchronized void notifyAnimalAddedListeners (Animal animal) { /*...*/ }
```

While this implementation would successfully remove the race-condition, since only one thread at a time could alter or access the list of registered listeners, it is too restrictive (for more information on the `synchronized` keyword and the Java concurrency model, see the official page for method and statement synchronization). By using method synchronization, we are overlooking the nature of the concurrent access to the list of registered listeners: Registering and unregistering a listener alters the list of listeners as a writer, while the notification of listeners accesses the list of listeners as a reader. Since the access through notification is a reading action, multiple notifications can occurrently be performed.

As such, so long as no listener is registered or unregistered, any number of concurrent notifications can be executed without introducing a race-condition on the list of registered listeners (there still exist other race-conditions that will be discussed in detail and rectified shortly). To accomplish this, a `ReadWriteLock` is used to perform separate read and write locking. A thread-safe implemention of the `Zoo` class, `ThreadSafeZoo`, is depicted below:

```
1
public class ThreadSafeZoo {
2

3
    private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
4
    protected final Lock readLock = readWriteLock.readLock();
5
    protected final Lock writeLock = readWriteLock.writeLock();
6

7
    private List<Animal> animals = new ArrayList<>();
8
    private List<AnimalAddedListener> listeners = new ArrayList<>();
```

```
9
10  public void addAnimal (Animal animal) {
11      // Add the animal to the list of animals
12      this.animals.add(animal);
13
14      // Notify the list of registered listeners
15      this.notifyAnimalAddedListeners(animal);
16  }
17
18  public AnimalAddedListener registerAnimalAddedListener (AnimalAddedListener listener) {
19      // Lock the list of listeners for writing
20      this.writeLock.lock();
21
22      try {
23          // Add the listener to the list of registered listeners
24          this.listeners.add(listener);
25      }
26      finally {
27          // Unlock the writer lock
28          this.writeLock.unlock();
29      }
30
31      return listener;
32  }
33
34  public void unregisterAnimalAddedListener (AnimalAddedListener listener) {
35      // Lock the list of listeners for writing
36      this.writeLock.lock();
37
38      try {
39          // Remove the listener from the list of the registered listeners
40          this.listeners.remove(listener);
41      }
42      finally {
43          // Unlock the writer lock
44          this.writeLock.unlock();
```

```
45
        }
46
    }
47

48
    public void notifyAnimalAddedListeners (Animal animal) {
49
        // Lock the list of listeners for reading
50
        this.readLock.lock();
51

52
        try {
53
            // Notify each of the listeners in the list of registered listeners
54
            this.listeners.forEach(listener -> listener.onAnimalAdded(animal));
55
        }
56
        finally {
57
            // Unlock the reader lock
58
            this.readLock.unlock();
59
        }
60
    }
61
}
```

With this, the *subject implementation* is thread-safe and multiple threads are capable of concurrently notifying the subject. Even so, there are still two noticeable race-conditions:

1. The concurrent access of each listener. Since multiple threads may notify a the list of registered listeners of a new animal, each listener may be concurrently called by multiple threads.

2. The concurrent access of the list of animals. Multiple threads may add to the list of animals, requiring some concurrency mechanism to guard against a race-condition. This also leads to an issue with the ordering of notifications. For example, a race-condition can occur where the list of registered listeners is notified of the addition of animal 1 *after* being notified of the addition of animal 2, even if animal 1 is added before animal 2 if the addition of animal 1 is performed in a separate thread from the addition of animal 2 (i.e., thread 1 adds animal 1 and blocks before notifying the listeners; thread 2 adds animal 2 and notifies the listeners, completing execution; thread 1 resumes and notifies the listeners that animal 1 was added). While this is not always an issue (if the ordering of notifications does not matter), in the general case, this presents an issue (in the case where ordering does not matter, we simple ignore the race-condition, but it nonetheless exists).

Concurrent Access to Listeners

The issue presented by concurrent access to listeners is rectified by making each individual listener thread-safe. In the spirit of assigning responsibility to the class with the information neccessary to fulfill the responsibility, it is responsibility of each listener to ensure that it is thread-safe. For example, in the counting listener example before, multiple threads incrementing or decrementing the count of animals can cause thread-safety issues. In order to rectify this issue, we must ensure that the count of animals is altered in an atomic manner, either through atomic variables or through method synchronization. The former solution can resemble the following:

```
13
1
public class ThreadSafeCountingAnimalAddedListener implements AnimalAddedListener {
2

3
    private static AtomicLong animalsAddedCount = new AtomicLong(0);
4
```

```
5
    @Override
6
    public void onAnimalAdded (Animal animal) {
7
        // Increment the number of animals
8
        animalsAddedCount.incrementAndGet();
9

10
        // Print the number of animals
11
        System.out.println("Total animals added: " + animalsAddedCount);
12
    }
13
}
```

The latter technique, using method synchronization, would resemble the following:

```
13
1
public class CountingAnimalAddedListener implements AnimalAddedListener {
2

3
    private static int animalsAddedCount = 0;
4

5
    @Override
6
    public synchronized void onAnimalAdded (Animal animal) {
7
        // Increment the number of animals
8
        animalsAddedCount++;
9

10
        // Print the number of animals
11
        System.out.println("Total animals added: " + animalsAddedCount);
12
    }
13
}
```

It is important to note that is the responsibility of each listener to ensure thread-safety. First, it is not simple for the subject to ensure that access to and alteration of a listener is performed in a thread-safe manner: The subject must understand the underlying logic used by the listener. Thus, the listener itself has the knowledge needed to acheive thread-safety, not the subject. Moreover, if another subject were to use the same listener, the thread-safety logic would need to reimplemented in the second subject (and all other subjects using the listener). Thus, it is desirable that a listener take on the responsibility of ensuring its own thread-safety.

Ordered Notification of Listeners

When ordered execution of listeners is required, the read-write lock implementation alone is insufficient for acheiving total ordering of listener invocation. Instead, some mechanism must be used to ensure that invocations of the notify method are executed in the order in which the animals are added to the zoo. Although it is tempting to use method synchronization to acheive this ordering, according to the Oracle documentation for method synchronization, method synchronization *does not* provide ordering of execution. Rather, it ensures that methods are executed atomically (i.e., execution will not be interrupted, but it does not ensure a first-come-first-out (FIFO) ordering of executing threads). To accomplish this ordering, a `ReentrantReadWriteLock` is used, with fair-ordering enabled:

```
61
1
public class OrderedThreadSafeZoo {
2
```

```
3
    private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock(true);
4
    protected final Lock readLock = readWriteLock.readLock();
5
    protected final Lock writeLock = readWriteLock.writeLock();
6
7
    private List<Animal> animals = new ArrayList<>();
8
    private List<AnimalAddedListener> listeners = new ArrayList<>();
9
10
    public void addAnimal (Animal animal) {
11
        // Add the animal to the list of animals
12
        this.animals.add(animal);
13
14
        // Notify the list of registered listeners
15
        this.notifyAnimalAddedListeners(animal);
16
    }
17
18
    public AnimalAddedListener registerAnimalAddedListener (AnimalAddedListener listener) {
19
        // Lock the list of listeners for writing
20
        this.writeLock.lock();
21
22
        try {
23
            // Add the listener to the list of registered listeners
24
            this.listeners.add(listener);
25
        }
26
        finally {
27
            // Unlock the writer lock
28
            this.writeLock.unlock();
29
        }
30
31
        return listener;
32
    }
33
34
    public void unregisterAnimalAddedListener (AnimalAddedListener listener) {
35
        // Lock the list of listeners for writing
36
        this.writeLock.lock();
37
38
        try {
```

```
39
            // Remove the listener from the list of the registered listeners
40
            this.listeners.remove(listener);
41
        }
42
        finally {
43
            // Unlock the writer lock
44
            this.writeLock.unlock();
45
        }
46
    }
47

48
    public void notifyAnimalAddedListeners (Animal animal) {
49
        // Lock the list of listeners for reading
50
        this.readLock.lock();
51

52
        try {
53
            // Notify each of the listeners in the list of registered listeners
54
            this.listeners.forEach(listener -> listener.onAnimalAdded(animal));
55
        }
56
        finally {
57
            // Unlock the reader lock
58
            this.readLock.unlock();
59
        }
60
    }
61
}
```

With fair-ordering enabled, all threads accessing the register, unregister, and notify methods will approximate a FIFO ordering of read and write lock acquisition. For example, one thread (thread 1) registers a listener and another thread (thread 2), after the execution of the registration starts, attempts to notify the registered listeners. Additionally, thread 3 attempts to notify the registered listeners after thread 2 is waiting on the read lock. Using the fair-ordering approach, thread 1 will first register the listener; thread 2 will then notify the listeners; lastly, thread 3 will notify the listeners. This ensures that the order in which the actions were *initiated* is the order in which the actions were taken.

In the method synchronization technique, there is no guarantee that thread 2 will notify the listener prior to thread 3: Thread 3 may be selected for execution by the lock prior to thread 2, even though thread 2 obtained the lock first. This last portion is the crux of the problem: The fairness policy ensure that threads are executed in the order in which they lock. There are instricacies in the ordering or readers and writers, and the official documentation for the `ReentrantReadWriteLock` should be consulted to ensure that the logic of the lock is sufficient for the task at hand.

With thread-safety acheived, in the next section, we will look at the advantages and disavantages of extracting the logic of the subject into its own mixin class and capturing this logic as a repeatable unit of code.

Observer Pattern Subjects as Mixins

Based on the designs used above for our Observer Pattern implementation, it may be tempting to extract this functionality into its own mixin. In general, an Observer Pattern subject will contain a collection of registered listeners, a method to register a new listener, a method to unregister previously registered listeners, and a method to

notify the registered listeners. As seen in the zoo example above, the only portion of the zoo that deals with the domain knowledge of a zoo is the list of animals; otherwise, the remaining portion of the code in the class deals with the logic surrounding the subject implementation.

An example of this mixin is depicted below (note the thread-safe code is removed for the sake of brevity):

```
public abstract class ObservableSubjectMixin<ListenerType> {

    private List<ListenerType> listeners = new ArrayList<>();

    public ListenerType registerListener (ListenerType listener) {
        // Add the listener to the list of registered listeners
        this.listeners.add(listener);

        return listener;
    }

    public void unregisterAnimalAddedListener (ListenerType listener) {
        // Remove the listener from the list of the registered listeners
        this.listeners.remove(listener);
    }

    public void notifyListeners (Consumer<? super ListenerType> algorithm) {
        // Execute some function on each of the listeners
        this.listeners.forEach(algorithm);
    }
}
```

Since we do not have any interface information about the type of listeners being registered (the generic type parameter is not qualified), we are unable to make specific notification calls on the listeners. Instead, we make the notification function more general, allowing client code to supply some function, which accepts an argument matching the type of the generic parameter, to act on each of the listeners. With this implementation, our subject implementation becomes

```
public class ZooUsingMixin extends ObservableSubjectMixin<AnimalAddedListener> {

    private List<Animal> animals = new ArrayList<>();

    public void addAnimal (Animal animal) {
        // Add the animal to the list of animals
        this.animals.add(animal);
```

```
9
        // Notify the list of registered listeners
10
        this.notifyListeners((listener) -> listener.onAnimalAdded(animal));
11
    }
12
}
```

The main advantage to this mixin technique is that it provides a reusable, encapsulated class that represents an observable subject, rather than sprinkling the logic for an Observer Pattern subject throughout multiple subjects. What's more, this method clears up the zoo implementation, which now consists only of domain knowledge dealing with the storage of animals (rather than knowledge of how to store and notify listeners).

While these advantages may be of use in certain contexts, it should also be noted that there are some disadvantages that may preclude the use of the mixin technique. First, what if we wanted to store multiple types of listeners? For example, what if we also wanted to store a new listener type called `AnimalRemovedListener`, that will be executed when an animal is removed? Since the mixin is represented as an abstract class, we cannot use multiple inheritance in Java to subclass multiple classes. Furthermore, we cannot transform the mixin into an `interface` and use `default` implementations for the methods, since interfaces, by definition, may not contain state, and state is required to store the list containing the registered listeners.

A possible solution is to create a new listener type, `ZooListener`, that contains methods for notifying when an animal is both added and removed, as follows:

```
4
1
public interface ZooListener {
2
    public void onAnimalAdded (Animal animal);
3
    public void onAnimalRemoved (Animal animal);
4
}
```

Now, we can use this listener interface in our zoo implementation, allowing a single listener type to be used to notify listeners of changes to the state of the zoo:

```
20
1
public class ZooUsingMixin extends ObservableSubjectMixin<ZooListener> {
2

3
    private List<Animal> animals = new ArrayList<>();
4

5
    public void addAnimal (Animal animal) {
6
        // Add the animal to the list of animals
7
        this.animals.add(animal);
8

9
        // Notify the list of registered listeners
10
        this.notifyListeners((listener) -> listener.onAnimalAdded(animal));
11
    }
12

13
    public void removeAnimal (Animal animal) {
14
        // Remove the animal from the list of animals
15
        this.animals.remove(animal);
16

17
        // Notify the list of registered listeners
```

```
18
        this.notifyListeners((listener) -> listener.onAnimalRemoved(animal));
19
    }
20
}
```

Although combining multiple listener types into a single listener interface does solve the problem of using multiple listener types with the observable mixin, it does have some disadvantages that are explored in the following section.

Multi-Method Listeners & Adapters

What if a listener's interface includes so many methods to implement that it becomes cumbersome? For example, the Swing `MouseListener` includes 5 methods that must be implemented, even if we are only interested in one of the events. If we wish to listen for a mouse clicked event, we are forced to implement *all* 5 of the methods; more than likely, we are also going to implement the remaining methods with blank method bodies. Not only is this tedious, it also pollutes our code with needless clutter.

One solution to this problem is to create an adapter (derived from another GoF pattern, the Adapter Pattern), where a concrete class is created that implements the abstract methods in the listener interface, from which all future concrete listener classes inherit. Therefore, the concrete listener classes may pick and choose the methods to implement, leaving the unimplemented methods to the default behavior of the adapter. Using our `ZooListener` example above, we create a `ZooAdapter` (it is a common practice to replace the `Listener` in the listener interface with `Adapter` in the adapter class name), which resembles the following:

```
8
1
public class ZooAdapter implements ZooListener {
2

3
    @Override
4
    public void onAnimalAdded (Animal animal) {}
5

6
    @Override
7
    public void onAnimalRemoved (Animal animal) {}
8
}
```

At first glance, this adapter class may seem trivial, but the benefits it produces are very practical. Now a concrete class, such as the one below, need only implement the methods of interest:

```
8
1
public class NamePrinterZooAdapter extends ZooAdapter {
2

3
    @Override
4
    public void onAnimalAdded (Animal animal) {
5
        // Print the name of the animal that was added
6
        System.out.println("Added animal named " + animal.getName());
7
    }
8
}
```

It is important to note that there are two alternatives to the adapter class technique that provide the same functionality: (1) using default method implementations and (2) collapsing the listener interface and adapter into a concrete class. The first alternative is new to Java 8, which allows developers to provide default (defender) implementations for methods in an interface.

The purpose of this capability was to allow the developers of the standard Java library to extend the collections, streams, and other APIs without breaking existing code that did not include these extensions. Therefore, this approach should be used with caution. Some may go so far as to claim that the use of the default method implementations is a code smell, while others see it as an acceptable part of Java 8. In either case, it is important to understand the purpose of this technique, and understand that it may be the best design decision, but that depends on the context of the problem at hand. Using default method implementations, the `ZooListener` interface becomes

```
public interface ZooListener {
    default public void onAnimalAdded (Animal animal) {}
    default public void onAnimalRemoved (Animal animal) {}
}
```

By simply adding the default qualifier, concrete classes that implement this interface are no longer required to implement *all* methods in the interface, but rather, the selected methods of interest. While this is a clean and simple solution to the bloated interface problem, developers should remember to use this technique with care.

The second technique is a simplification of the Observer Pattern, where the listener interface is bypassed, and a concrete class, with *do nothing* implementations are used for each of the methods in the listener, is used instead. For example, the `ZooListener` interface becomes

```
public class ZooListener {
    public void onAnimalAdded (Animal animal) {}
    public void onAnimalRemoved (Animal animal) {}
}
```

Although this technique simplifies the inheritance hierarchy of the Observer Pattern, it is not appropriate in all cases. First, if a concrete listener intends to implement multiple listener interfaces, it cannot do so if the listener interface is collapsed into a concrete class. For example, if a single concrete listener were to implement the `AnimalAddedListener` interface and `AnimalRemovedListener` interface, it would be unable to do so if those interfaces were collapsed into concrete classes. Second, the intent of the listener interface is much clearer than that of the listener concrete class. In the former case, it is clear that we are attempting to provide an interface for other classes to implement (with the functionality that suits the desired listener), but in the latter case, this is not so evident.

Without proper documentation, it is not overtly clear that we have created a class that is intented to act like an interface, but simply has default *do nothing* implementations for each of the methods. Additionally, the name of the class does not include *adapter*, since it is not adapting an interface, and therfore, the name of the class does not particularly hint at this intent either. For these reasons, it is important to select the best technique based on the problem set at hand. There is no silver bullet and the choice made will depend on the needs of the design for each problem.

Before moving on to the next section, it is important to note that Observer Pattern adapters are common, especially in legacy Java code. As seen above, the Swing API is fond of this method, as are many older uses of the Observer Pattern in Java 5 and 6 libraries. While there may not be a need to create an adapter for the listeners we create, it is important to understand their purpose and applications, as we may come upon their use in existing code. In the following section, we will deal with temporally complex listeners, where listeners may perform time-consuming computations or make asynchronous calls that may not return immediately.

Complex & Blocking Listeners

There is an assumption made with the Observer Pattern that listeners will execute is some managable time. Unbeknownst to the caller of a method in a subject, a series of listeners will be executed, and it is assumed that this action will be transparent to the caller. For example, the client code that adds a new animal to the zoo does not know that a list of listeners will be executed before control is returned to the client code. If the execution of the listeners requires a noticeable period of time (due to the number of listeners, the execution time of each listener, or a combination of the two), then client code may view the execution of a simple method (such as adding an animal) as having temporal side-effects.

While the full converage of this topic is outside the scope of this article, the following are a list of a few items that developers should consider if a series of listeners are noticeably complex:

1. Dispatch a new thread *within the listener*. Instead of executing the logic of the listener sequentially, have the listener dispatch a new thread, off-loading the execution of the listener logic to the new thread. After the thread has been dispatched, return from the listener method, allowing other listeners to begin execution, while concurrently allowing the dispatched thread to execute the logic of the listener.

2. Dispatch a new thread *within the subject*. Instead of iterating through the list of registered listeners sequentially, have the notification method of a subject dispatch a new thread and iterate through the list of registered listeners. This allows the notification method to return immediately, while allowing the concurrent execution of each listener. Note that some thread-safety mechanism is needed to ensure that concurrent modification of the list of registered listeners does not occur.

3. Queue the listener function invocations and have a set of threads execute the listener functions. Instead of simply iterating through each listener in the list of registered listeners, have the execution of the listener methods encapsulated in some functor and queue these functors. Once these functors have been queued, a thread or set of threads (possibly from a thread pool) can pop each functor from the queue and execute the listener logic. This amounts to a Producer-Consumer problem, where the notification process produces a set of executable functors that are queued, while the threads consume these functors from the queue and execute each. The functor must store the parameters to be provided to the listener method *at the time of creation*, not at the time of execution (which may be some indeterminate time after creation). For example, the functor is created to store the state of the listener method execution at the time the functor is created, allowing the consumer threads to execute the functor at a later time as if it were being executed at the time of creation. This functionality can be approximated in Java as follows

```
public class AnimalAddedFunctor {

    private final AnimalAddedListener listener;
    private final Animal parameter;

    public AnimalAddedFunctor (AnimalAddedListener listener, Animal parameter) {
        this.listener = listener;
        this.parameter = parameter;
    }

    public void execute () {
        // Execute the listener with the parameter provided during creation
        this.listener.onAnimalAdded(this.parameter);
    }
}
```

   Instead of iterating through each listener and executing the listeners immediately, a functor is created and queued for later execution. Once a functor has been queued for each of the listeners that would have been invoked, the notification method returns control to the client code. At some later time, the consumer threads execute these functors, which in turn executes each listener as if it had been executed when the notification method was invoked. This technique is call **paramerter binding** in other languages and is approximated in the above example (essentially, the single parameter of the above listener is stored, or **bound**, allowing the `execute()` method to be called with zero arguments). If the listener accepted more than one parameter, each parameter would be stored in the functor in a manner similar to the single parameter above.

It is important to note that if the order of execution of each listener must be maintained, then some overall ordering mechanism must be used. In the first case, the listeners are executed in their normal order, and the logic is dispatched to a new thread, ensure that execution of each listener is *initiated* in the order in which the listeners were

registered. In the second case, the queue provides some level of ordering that ensures that the functors are executed in the order in which they were placed on the queue. In general, the introduction of multithreaded listener execution introduces a level of complexity that must be handled with care to ensure the desired functionality is acheived.

Conclusion

The Observer Pattern has been a staple of software design patterns even before it was codified in 1994, and continues to provide a clever solution to a recurring set of problems in software. In particular, Java has been a leader in incorporating this pattern in its standard libraries, but as Java has advanaced to version 8, it is important that this classic pattern be revisited. With the advent of lambdas and other new constructs, this "old" pattern has taken on a new look. Whether dealing with legacy code or using this time-honored solution, the Observer Pattern is a mainstay in any developer's toolbox, especially the experienced Java developer.