Observer Pattern

Context:

An object (the *Subject*) is the source of events. Other objects (*Observers*) want to know when an event occurs.

Or several objects should be immediately updated when the state of one object changes, e.g. an editor with live preview.

Forces:

We don't want the observers to *poll* for changes, which is inefficient.

We don't want to complicate the *Subject* with a lot of code for event notification.
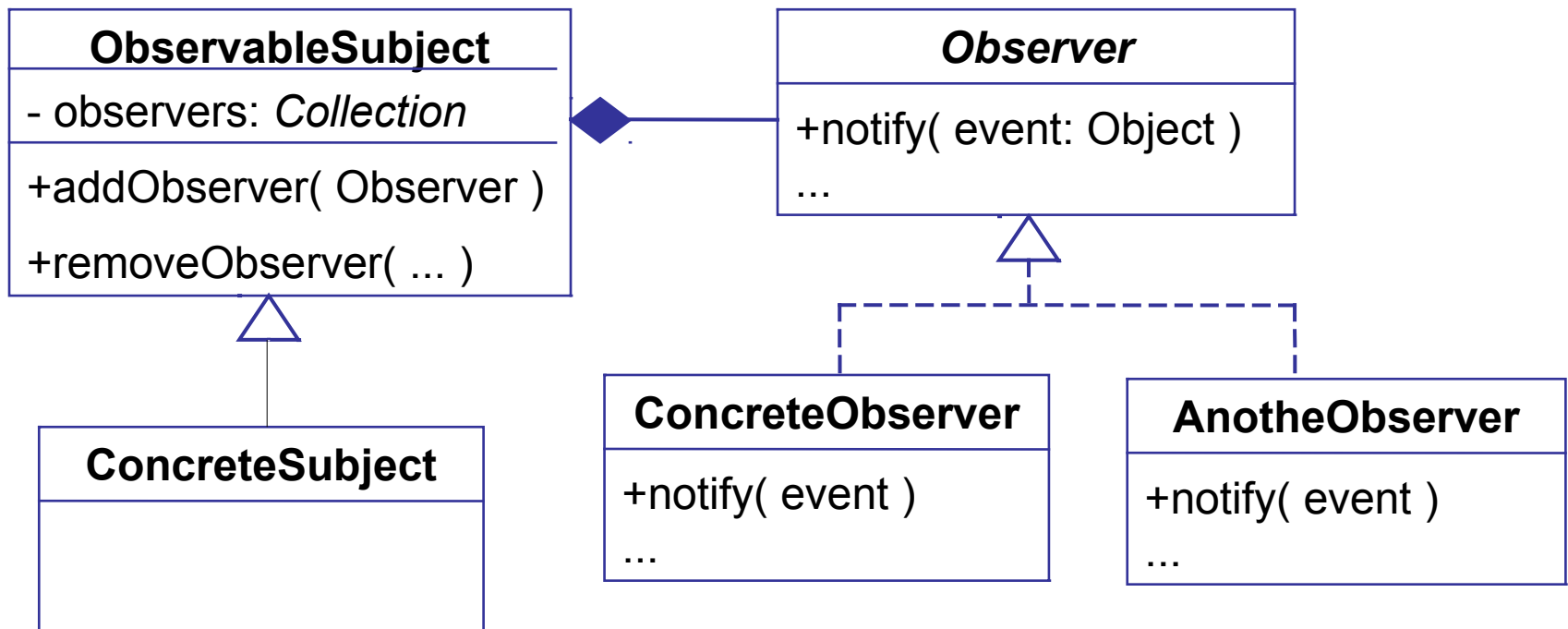
Observer Pattern

Solution:

(1) Subject provides a method for Observer to **register itself** as wanting to receive event notification.

(2) Subject calls a method when observers should be notified.

(3) To avoid complicating the subject, implement the registration and event notification code in a separate class.

This can be a superclass of the Subject, or another class that the Subject uses (delegate to it).

UML for Observer Pattern

(1) Subject provides a method for Observers to register themselves as wanting to be notified of events. Method: addObserver( )

(2) Each Observer implements a known method (*notify*) for the Subject to invoke when an event occurs.

ObservableSubject

- observers: *Collection*

+addObserver(Observer)

+removeObserver(...)

Observer

+notify(event: Object)
...

ConcreteSubject

ConcreteObserver

+notify(event)
...

AnotheObserver

+notify(event)
...

What are some examples of the Observer Pattern?

Button uses Observer

Subject: Button is the source of events.

Event: button press (an *ActionEvent*)

Observer: any object that want to know when the button is pressed.

How to implement:

1. Observer implements EventHandler, and defines a handle( ) method to receive notificatoins.

2. Observer registers itself by calling button.addEventHandler() or button.setOnAction()

Button Observers

This observer counts button presses.

```java
/** An observer that counts button presses */
public class ClickCounter
        implements EventHandler<ActionEvent> {
   private int count = 0;

   /** The event notification method. */
   public void handle(ActionEvent evt) {
      count += 1;
      System.out.println("Click number "+count);
   }
   public int getClickCount() { return count; }
}
```

Register the Observer

We must add ClickCounter as an observer of the Button.
This is called *registering an observer*.

```
Button button = new Button("Press Me");

ClickCounter counter = new ClickCounter();

// register the observer
button.addEventHandler(ActionEvent.ACTION,
                       counter );
```

Benefits of using Observers

1. Button is not *coupled* to the actual observer classes.

   Button depends only on the *interface* for observers.

2. We can define and add new observers any time (*extensible*).

3. We can *reuse* the same observer for many components.

Table for Identifying a Pattern

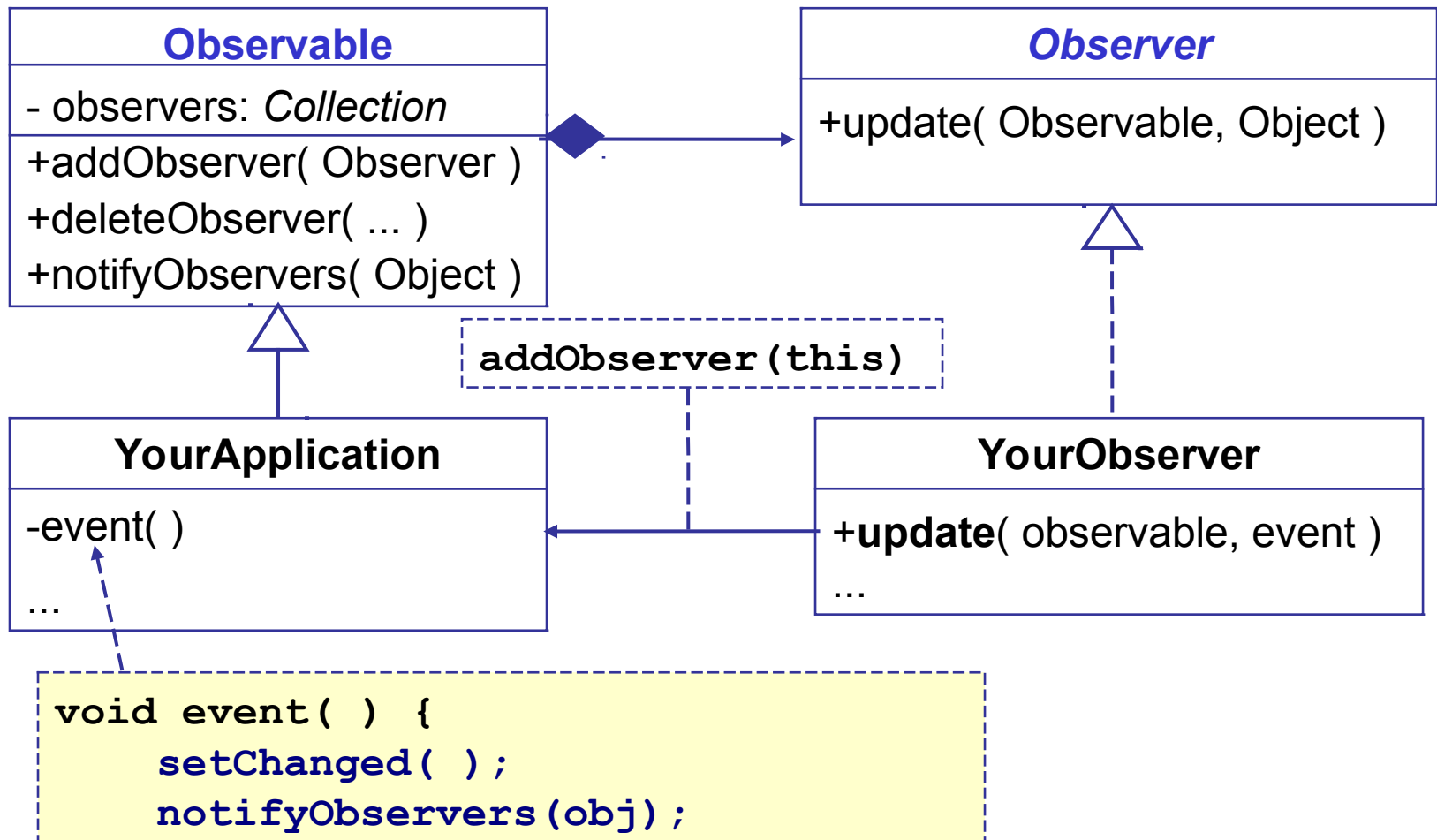| Name In Pattern	Name in Application: this is for a Button
Subject	Button
Observer	*EventHandler*
Concrete Observer	a class that implements *EventHandler*
addObserver(Observer)	addEventHandler( ) or setOnAction(this)
notify(Event) [in the observer]	handle(ActionEvent)
notifyObservers [in Subject]	fireEvent(ActionEvent)

Adding Observers to your App

How can we apply the Observer Pattern to our code?

Example: A UI for coin purse that tells us what the balance is.
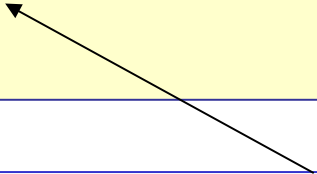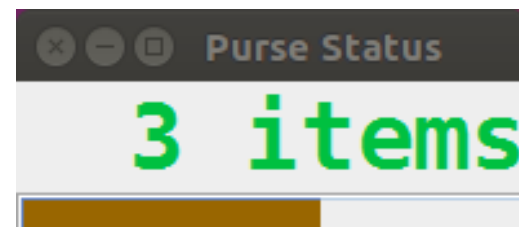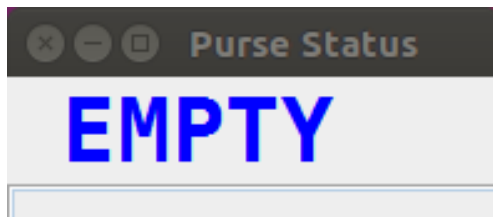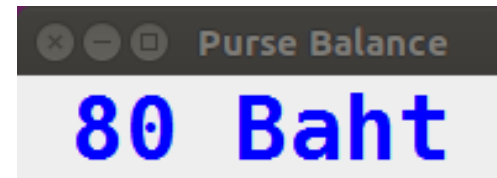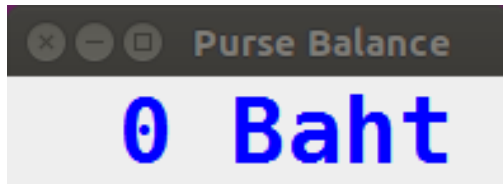
Observer Pattern in Java

Java provides an **Observable** class and **Observer** interface that make it *easy* to use the Observer pattern..

Observable
- observers: *Collection*
+addObserver(Observer)
+deleteObserver(...)
+notifyObservers(Object)

Observer
+update(Observable, Object)

`addObserver(this)`

YourApplication
-event( )
...

YourObserver
+**update**(observable, event)
...

```
void event( ) {
    setChanged( );
    notifyObservers(obj);
```

Using the Observable class

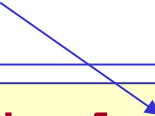(1) Declare that your Subject class extends **Observable**

```java
public class MySubject extends Observable
{
  /** An event the observers want to know about */
  public void somethingHappened( ) {
      doSomeWork( );
      // now notify the observers
      setChanged( );
      notifyObservers( ); // can include a parameter
  }
```

(2) When an event or change occurs, invoke `setChanged()`
and `notifyObservers()`

Writing an Observer

(3) Declare that observers *implement* the Observer interface.

```
public class MyObserver implements Observer {
  /* This method receives notification from the
   * subject (Observable) when something happens
   * @param subject Observable that caused notif.
   * @param message is value of parameter sent
   *    by subject. May be null.
   */
  public void update( Observable subject,
                      Object message ) {
    mySubject = (MySubject)subject;
```

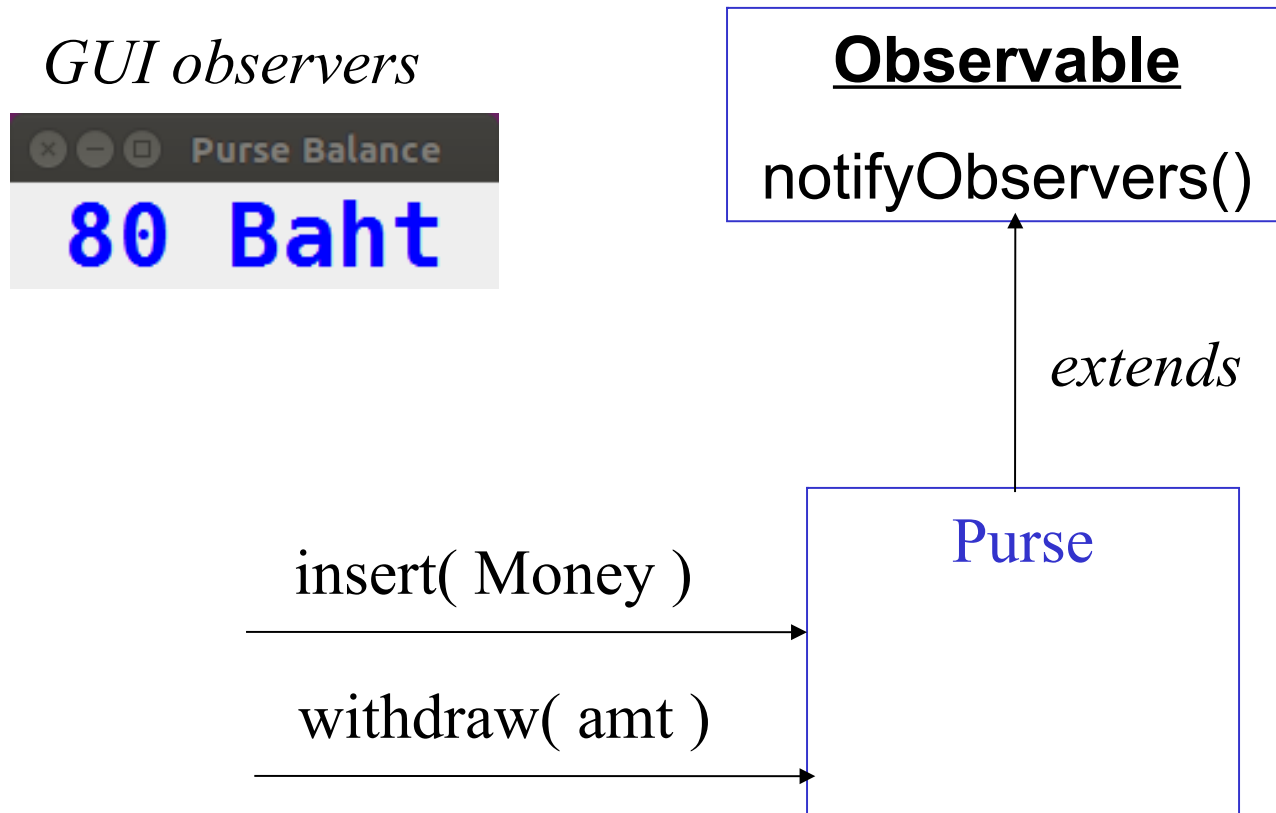(4) update takes action using notification from the Subject.

Last Step: add Observers to Subject

Call addObserver( ) to register the Observers with subject.

```java
public static void main(String [] args) {
    Observable subject = new MySubject( );
    MyObserver observer = new MyObserver( );

    subject.addObserver( observer );

    subject.run( );

}
```

Example for Coin Purse

What are the *interesting events?*

GUI observers

Purse Balance
80 Baht

Observable

notifyObservers()

extends

Purse

insert(Money)

withdraw(amt)

Purse with observer notification

The purse should notify observers when the state of the purse changes.

Draw a sequence diagram of what happens, using insert() as example.

C# Delegates as Observers

□ Delegate is a type in the C# type system.

□ It describes a group of functions with same parameters.

□ Delegate can act as a collection for observers.

```csharp
/** define a delegate that accepts a string **/
public delegate void WriteTo( string msg );
```

```csharp
/** create some delegates **/
WriteTo observers = new WriteTo( out.WriteLine );
observers += new WriteTo( button.setText );
observers += new WriteTo( textarea.append );
/** call all the observers at once! **/
observers("Wake Up!");
```