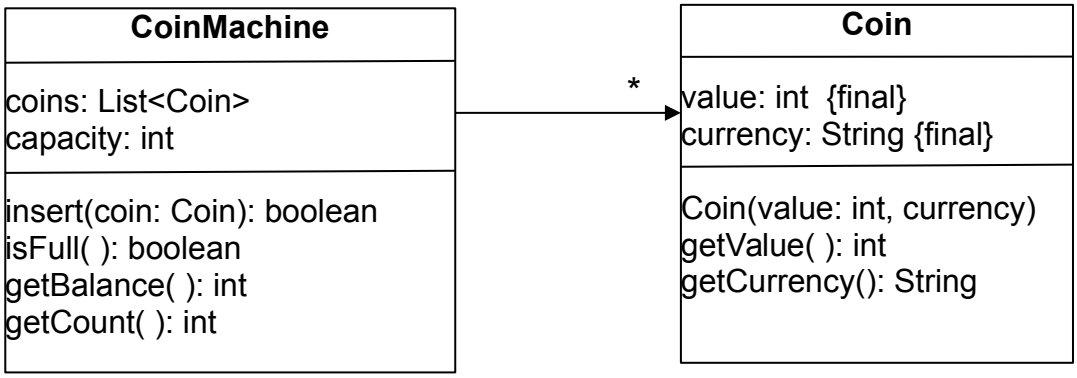
Objectives	Practice applying the Observer Pattern and Java Observer/Observable.
What to Submit	Commit your code to Github as a project named **`lab10`**. There is "starter" code for the project on Github. Fork (not clone) the project to your own Github account, then clone from your own account to your local computer.

Description of Coin Machine

A CoinMachine stores Coins. It knows how many of each type of coin have been inserted, and can compute the total value of the coins. A Coin Machine has a finite capacity (number of coins it can hold) and will refuse to accept coins when it is full.



Note: The JDK has a Currency class for displaying currencies and the values of money with correct formatting. For simplicity in this lab we use a String for currency, but using Currency would be better.

Here is a console demo for using the CoinMachine.

Problem 1. Answer these questions about the Coin Machine code

1.1 What methods can change the *state* of the CoinMachine? _____

1.2 For the purpose of inspection, the Coin Machine returns a "view" of the cons it contains, but it must be careful not to let the caller change the List of coins! For example, a caller could do this:

```
List<Coin> money = coinMachine.getCoins( );
money.remove( 0 );  // remove a coin!
```

How does Coin Machine prevent other code from secretly modifying its contents using getCoins()? Surrepticiously modifying attributes of an object is called "*breaking encapsulation*".

Problem 2. Write an equals( ) method for Coin

Write an equals method in the Coin class. Two Coins are equal if they have the same currency *and* the same value. You can assume the currency is not null.

Use the standard logic for equals methods. Write the code yourself. Don't use autogenerated code.

Problem 3. Make the CoinMachine Observable and add Observers

Java's Observer - Observable will be demonstrated in the lab.

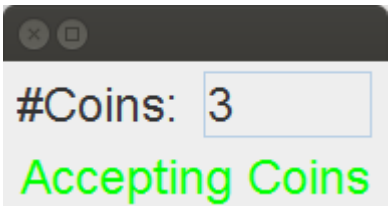
For more info, try the Java tutorial on Observers or read below.

3.1 Modify the CoinMachine class to be **`Observable`**. Notify Observers whenever coins are *successfully* inserted into the machine.

3.2 Create a class that implements *Observer* and just prints the balance of Coin Machine on the console each time that **`update( )`** is called, so you can see that it is working.

Each time you insert a coin (using console dialog) it should print the balance on the console.

3.3 Create at least 2 observers using graphical displays. They should all be updated whenever a coin is inserted. This should happen no matter how the coin is inserted (that is, insert via console dialog or insert via a GUI button press).



JButtons for inserting coins

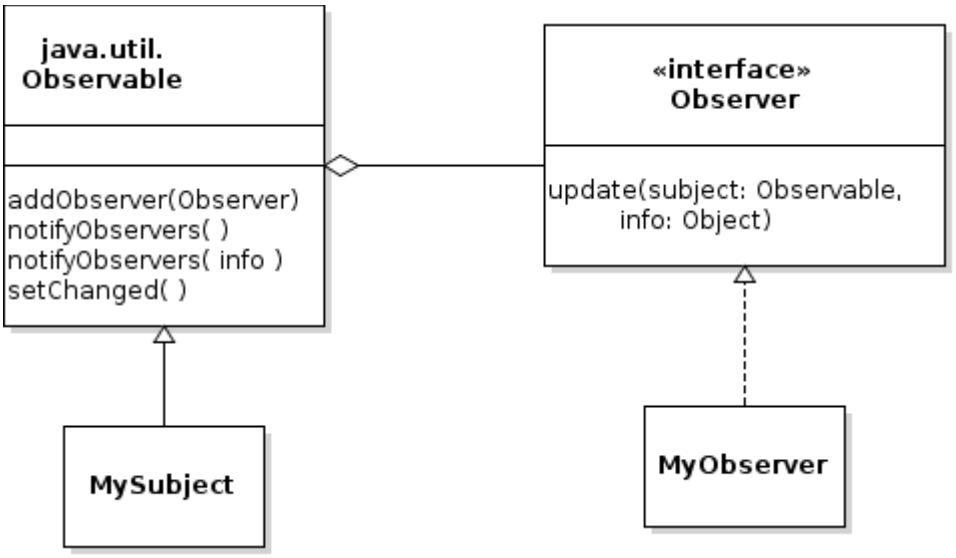
Introduction to Observer in Java

Use *Observers* when you want to notify objects of changes to a subject, or other "interesting" events. The *Subject* is the source of interesting information.

The *Observers* are objects that want to know when something happens to the *Subject.*

For example, using a GUI text editor, when you edit the text (the Subject) you want the formatted view of the text to update immediately (the observer). Some HTML editors do this.

In Java, the Observer pattern is implemented like this:



a) Define your subject class (MySubject) as a subclass of java.util.Observable.

b) In your subject class, when something "interesting" occurs that observers might want to know about, you tell the superclass (Observable) to notify them like this:

```
public void withdraw( int amount ) {
    withdraw the money
    // notify the observers that something happened
    super.setChanged();
    super.notifyObservers( );
}
```

You can optionally pass an object of any type to notifyObservers( ) and the parameter will be passed to the observers:

```
public void withdraw( int amount ) {
    withdraw the money
    // notify the observers that money was withdrawn
    super.setChanged();
    super.notifyObservers( "withdraw" );
    // the argument is passed to update( ) method of observers
}
```

c) The *Observers* implement the java.lang.Observer interface, which specifies one method:

```
public class MyObserver implements Observer {

  public void update(Observable subject, Object info ) {
    System.out.println("The subject just updated me!");
    // you can invoke methods of subject to get more info
    if (info != null)
      System.out.println("The subject sent: " + info );
}
```

d) Somewhere in your code you must "register" the *Observers* with the *Subject*. This is usually done in the Main or Application class. For example:

```
public static void main(String[] args) {
    MySubject subject = new MySubject( );
    MyObserver myobserver = new MyObserver( );
    subject.addObserver( myobserver );

    // rest of the code to initialize application
}
```

e) You can have as many *Observers* of a *Subject* as you want. This key step is to call subject.addObserver(observer) to "register" observers. If you forget this step, the observer will never be updated.

You can also unregister an Observer by calling subject.deleteObserver(observer);