# Overview:

* 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.
* Many objects need to be notified whenever an event occurs and want to be notified automatically.
  + However, I do not want to change the broadcasting object every time there is a change to the set of objects listening to the broadcast.
    - Would be like having to change a radio transmitter every time a new car radio comes to town.
    - Want to decouple the notify-ers and the notify-ees.
    - This is what the observer pattern solves.
* In this pattern, there are many observers (objects) which are observing a particular subject (object).
  + Observers register themselves to a subject and are automatically notified when the subject changes
    - When they lose interest in the subject, they simply unregister from subject.
* Provides a loosely coupled design between objects that interact.
  + More flexibility with changing requirements
* One of the most common patterns that is used in software development.

# Examples:

* A great example is how newspapers or magazines subscriptions work.
* A newspaper publisher goes into business and begins publishing newspapers
* You subscribe to a particular publisher, and every time there is a new edition, it is delivered to you.
  + As long as you remain a subscriber, you get new newspapers.
* While the publisher remains in business, people, hotels, airlines, and other businesses constantly subscribe and unsubscribe to the newspaper.
* How about a celebrity who has many fans:
  + Each of these fans wants to get all the latest updates of his/her favorite celebrity.
  + He/she can follow the celebrity as long as his/her interest persist.
  + When they lose interest, they simply stop following that celebrity.
  + The fan is an observer and celebrity is the subject.
* Consider a simple UI-based example, where this UI is connected with some database:
  + A user can execute some query through that UI and after searching the database, the result is reflected back in the UI.
  + If a change occurs in the database, the UI should be notified so that it can update its display according to the change.
* Heavily used in GUI toolkits and event listeners
  + The button (subject) and onClickListner (observer) are modelled with observer pattern.
* Social media, RSS feeds, email subscriptions in which you have the option to follow or subscribe and you receive latest notification.
* All users of an app on the google play store are notified if there is an update.

# One-to-many relationship:

* The Observer pattern defines a one-to-many relationship between a set of objects.
  + When the state of one object changes, all of its dependents are notified.
* The subject and observers define the one-to-many relationship:
  + Observers are dependent on the subject such that when the subject’s state changes, the observers get notified.

# When to use this pattern?

* When a change to one object requires changing others, and you do not know how many objects need to be changed.
* When multiple objects are dependent on the state of one object.
* When an object should be able to notify other objects without making assumptions about who these objects are:
  + You do not want these objects tightly coupled.

# Implementation:

# Overview:

* To understand observer pattern implementation, first you need to understand the subject and observer objects.
* The relation between subject and observer can easily be understood as an analogy to magazine subscription.
* A magazine publisher (subject) is in the business and publishes magazines (data).
* If you (user of data/observer) are interested in the magazine, you subscribe (register), and if a new edition is published it is delivered to you.
* If you unsubscribe (unregister) you stop getting new editions.
* Publisher does not know who you are and how you use the magazine, it just delivers it to you because you are a subscriber (loose coupling).
* Observer and Subject are interfaces or abstract classes.

# Participants:

* **Subject:**
  + Knows its observers (contains a list of observers to notify)
  + Any number of Observer objects may observe a subject
  + Provides an interface for attaching and detaching Observer objects
    - Methods allowing observers to register and unregister themselves
    - Also contains a method to notify all the observers of any change.
* **Observer:**
  + Defines an updating interface for objects that should be notified of changes in a subject.
  + All potential observers need to implement this interface.
* **ConcreteSubject:**
  + Stores state of interest to ConcreteSubject objects.
  + Sends a notification to its observers when its state changes.
* **ConcreteObserver:**
  + Maintains a reference to a ConcreteSubject object
  + Stores state that should stay consistent with the subject’s
  + Implements the Observer updating interface to keep its state consistent with the subject’s.

# Advantages of this implementation:

* The Observer Pattern provides an object design where subjects and observers are loosely coupled.
* The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface)
  + Does not need to know the concrete class of the observer, what it does, or anything else about it,
* We can add new observers at any time
  + The only thing the subject depends on is a list of objects that implement the Observer interface.
  + We can replace any observer at runtime with another observer and the subject will keep purring along.
* We never need to modify the subject to add new types of observers:
  + All we have to do is implement the Observer interface in the new class and register as an observer:
    - Subject does not care; it will deliver notifications to any object that implements the Observer interface.
* We can reuse subjects or observers independently of each other
  + If we have another use for a subject or an observer, we can easily reuse them because the two are not tightly coupled.
* Changes to either the subject or an observer will not affect the other:
  + As long as the objects still meet their obligations to implement the subject or observer interfaces.

# Java implementation:

* The Observer pattern is so useful that Java contains an implementation of it in its packages.
* These are quite similar to our Subject and Observer interfaces, but give you a lot of functionality out of the box.
  + You can also either implement a push or pull style of update to your observers.
* The Observable class and the Observer interface make up the pattern
  + Observable class plays the role of the subject.
* Instead of the methods attach, detach, and notify, Java uses addObserver, deleteObserver, and notifyObservers.
* For an Object to become an observer, you will need to implement the Observer interface.
  + Call addObserver() on any Observable object
  + To remove yourself as an observer, just call deleteObserver()
* For the Observable to send notifications you, need to be Observable by extending the java.util.Observable super class.
  + You first must call the setChanged () method to signify that the state has changed in your object.
  + Then call one of two notifyObservers () methods (notifyObservers () or notifyObservers (Object arg)).
* For an Observer to receive notifications, It must implement the update method
  + The signature of the method is a bit different
    - If you want to “push” data to the observers, you can pass the data as a data object to the notifyObservers (arg) method.
    - If not, then the Observer has to “pull” the data it wants from the Observable object passed to it.
* The setChanged () method is used to signify that the state has changed:
  + When notifyObservers () is called it should update its observers.
  + If notifyObservers () is called without first calling setChanged (), the observers will not be notified.
* You need to call setChanged () for notifications to work:
  + You may also want to use the clearChanged () method, which sets the changed state back to false.
  + In addition, the hasChanged () method, which tells you the current state of the changed flag.
* The java.util.Observable implementation has a number of problems that limit its usefulness and reuse:
  + Not widely used
  + Implementation is really simple
* Observable is a class
  + You have to subclass it
  + Means you cannot add on the Observable behavior to an existing class that already extends another superclass.
  + Limits its reuse potential (and is not that why we are using patterns in the first place?).
* There is no Observable interface
  + You cannot create your own implementation that plays well with Java’s built-in Observer API.
  + You do not have the option of swapping out the java.util implementation for another (say, a new, multithreaded implementation)
* Observable may serve your needs if you can extend java.util.Observable.
* On the other hand, more often than not, you will need to create your own implementation.