Observable Pattern

* Use cases:
  + user interface elements are "observable"
    - buttons, scrollbars, textfields
  + application objects that must respond to user's interactions with the UI are "observers"
* Asynchronous Programming
  + also known as "event-based programming"
  + may have well-defined actions corresponding to events that may occur, but can't know in advance which event will occur or when it will occur
* building block/component in other design patterns

* defining observer as an interface allows any class to act as an observer for the Observable class
  + Observable can have more than on observer
  + the Observable object doesn't care what type of object is doing the observing
* Observable maintains a collection of Observers
  + at the very least, the Observable must:
    - add observers
    - remove observers
    - update observers through some sort of method/action when something happens
* **Example:** Game = observable, Duke fan and UNC fan are observers
  + main method changes score -> Game changes its internals, gives points to correct team -> notifies all observers by running a for in loop -> observers call their update method
* But the basic version of observer doesn't work very well, because what if one observers is linked to more than one obsevables?
  + the problem: when update is called on the observer, how does the observer know which observable called it?
  + to fix this:
    - pass in an observable object into the update method
    - then, the observer object doesn't even have to encapsulate an observable object at all, it just knows within the method
  + Ex: within the observable:

void notiftyObservers(){

for(Observer o: observers){

o.update(this);

}

}

* But also, the observable doesn't know WHAT changed
  + must refine our method again
  + in the update method, pass in another parameter that specifies the change

interface Observer{

void update(Observable o, info i);

}

* in our game example, the update method now takes a string called whoScored
  + this will prevent the Duke fan to cheer when UNC scores, because that's what happens now because
* Java provides Observer/Observable in java.util
  + don't have to use them; our examples didn't
  + the pattern is defined by relationship between objects and their interaction
  + if you DO use the java.util skeleton:
    - you no longer have to manage your own list since you inherit one from Observable
    - the Observers must implement java.util.Observer
    - parameters to update are no longer context specific
      * need contravariant cast
    - also, each observer has a setChanged() method, which will mark the observer as having been changed
      * call setChanged() before notifyObservers(String teamThatScored)
  + Problem with this:
    - what if your observable object already has a parent due to design or circumstance
      * solution lies with delegation:
        + create a helper class that does subclass Observable(not a public class, declared within same file)

this class implements same interface as original class via delegation

it is a decoration of the original class, with the additional capability of being observable

override notifyObservers to call setChanged() first

* + - * + provide methods of Observable via delegation to this instance
      * delegation goes both ways
        + allows the Observer object to use hidden observable instance as if it were the original object
        + allows outside code to interact with original object as if it were Observable