Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Design Patterns (DPs) II

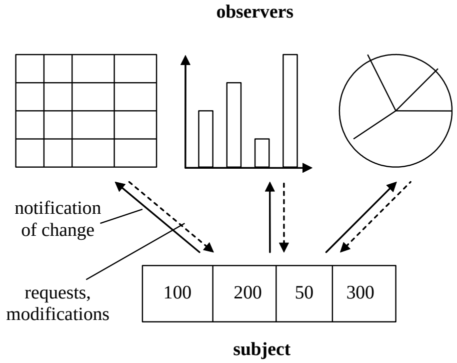
Observer based techniques and event models

I. Salomie, C.Pop
2017

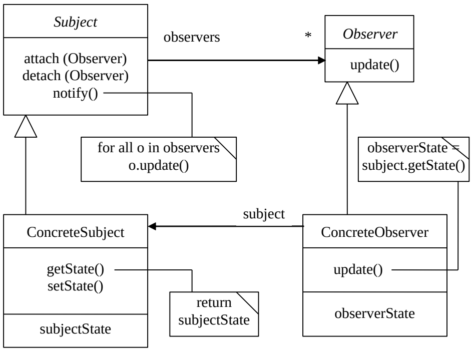
Observer [GoF, Grand]

Behavioral pattern

- Intention
  - Defines 1 : n dependency between objects
  - When the one side object changes the state, all its n dependent objects are automatically informed and updated
- Alternative name
  - Publish – Subscribers

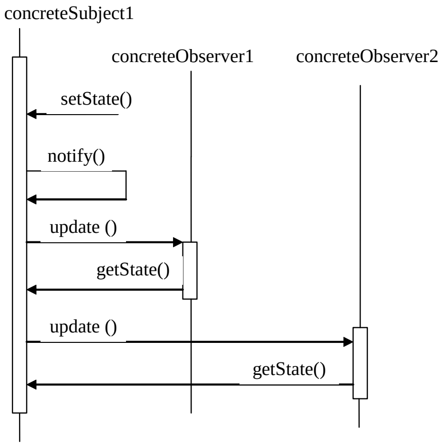


Observer
Structure and Participants



- Subject
  - Is associated with many dependant Observers or,
  - many Observer objects may observe the Subject object
- Observer
  - Specifies an interface that should be implemented by ConcreteObservers that should be notified whenever a change occurs in the observed subjects
- ConcreteSubject
  - Stores a state in which the observers are interested
  - Notifies their observers whenever the state changes
- ConcreteObserver
  - Implements the Observer's updating interface
  - Are notified whenever a subject changes its state
  - Manages a reference to a ConcreteSubject in which it shows interest
  - Stores a state that should be consistent with the associated ConcreteSubject

Observer
Colaborating Objects



- concreteSubject
  - Notifies its concrete observers whenever its state changes
- concreteObservers
  - When notified, queries the concrete subject for information
- Variations
  - Notify is not always invoked by the subject
  - It can be invoked by the observer or by other object

Observer

When to use the pattern

- When a certain abstraction has two dependent components
  - The two components should be implemented as two dependable separate classes
- When changes to an object determines changes into other dependable objects
  - the number of dependable objects is unknown)
- When an object must notify other objects without tightly coupling these objects

Main consequences

- Because of weak coupling between subjects and observers
  - Observers can be added / removed without modifying their subjects
  - Subjects and Observers can belong to different abstraction layers
- Support for event broadcasting
  - Subject sends notifications to all their subscribed observers
  - Observers can be added and / or removed at any time

Observer
Implementation problems to be considered

- How the subjects represent and keep track of their observers?
- An observers is interested in more subjects
  - How it identifies the subject who sends the notification?
- How the update is triggered? Who is responsible triggering?
  - Subject
  - Observer
  - Third party component
- How much info is passed from the subject to the observers when the state changes
  - Push model – all changes (the big picture)
  - Pull model – little info (what is needed)
    - The observers subscribe for specific event

Observer
Implementation in Java

- Java build-in support for the Observer pattern
- java.util.Observable
  - Class that plays the role of the Subject superclass
  - The ConcreteSubjects classes should inherit from java.Util.Observable
  - Uses a Vector object to store its Observers
- java.util.Observer
  - Interface that plays the role of Observer in the pattern structure
  - This interface must be implemented by any Observer class

Observer
Implementation in Java – Class Observable

- Instance variable *state*
- Default constructor – builds an Observable object with no observers
- addObserver(Observer o)
- deleteObserver(Observer o)
- notifyObservers(Object o)
  - In method implementation a call to **update()** is invoked for each subscribed Observer object. Two parameters are passed to the **update** method:
    - this Observable object
    - An argument (same as o) that indicated which attribute of the Observable object has changed
- notifyObservers()
  - No parameter, no indication about the attribute that changed
- hasChanged()
  - public boolean **hasChanged**()
  - Tests if this object has changed (i.e. variable *state*)
    - » True if and only if the **setChanged()** has been called more recently than the **clearChanged()** on this object;
- setChanged()
  - **protected** void **setChanged**()
  - Indicates that this object has changed
  - **hasChanged()** will return true after executing setChanged()
- clearChanged()
  - **protected** void **clearChanged**()
  - Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the **hasChanged** method will now return false. This method is called automatically by the **notifyObservers** methods [*Java API docs*]

Observer

Implementation in Java – Interface Observer

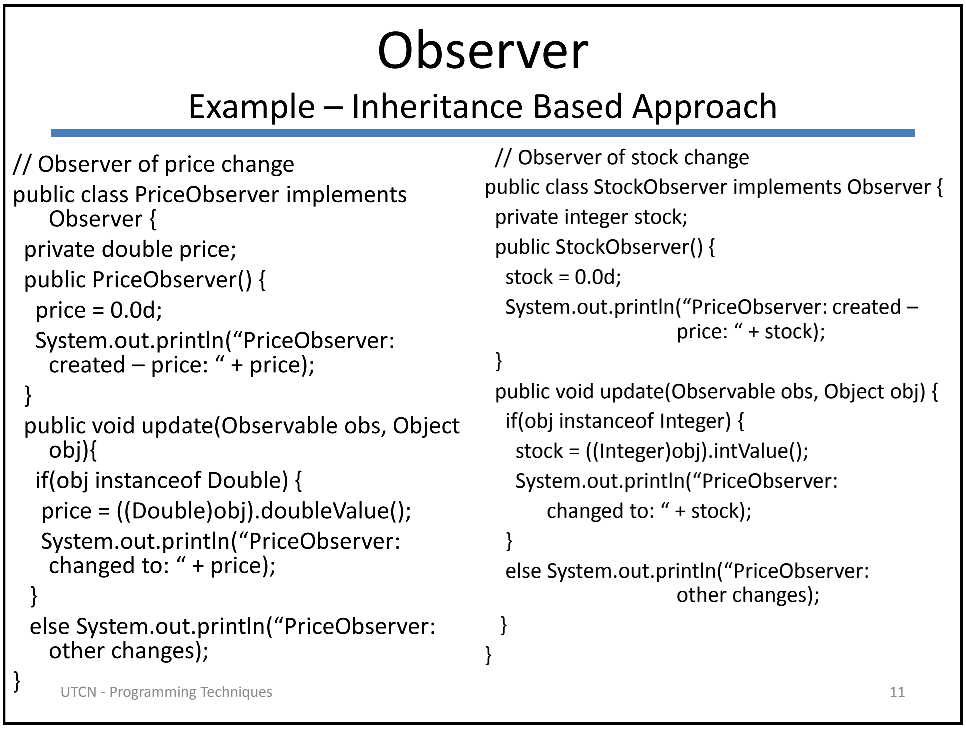
public abstract void update(Observable obj, Object arg)

- Parameters
  - *obj* is the observable object
  - *arg* is an object passed to notifyObservers() method
- The method is invoked whenever the Observed object has changed
- The application calls the observable object’s notifyObservers() to notify of the change all object’s observers

Observer

Example – Inheritance Based Approach

```
// Item is a Subject to be observed
public class Item extends Observable {
     private String name;
     private int stock;
     private double price;
     public Item (String s, int q, double p) {
          this.name = s;
          this.stock = q;
          this.price = p;
     }
     public String getName() {return name;}
     public int getStock() { return stock;}
     public double getPrice() { return price;}
     public void setStock(int q) {
       this.stock = q;
       setChanged();
       notifyObservers(new Integer(q));
     }
     public void setPrice(int p) {
       this.price = p;
       setChanged();
       notifyObservers(new Double(p));
     }
}// end class
```

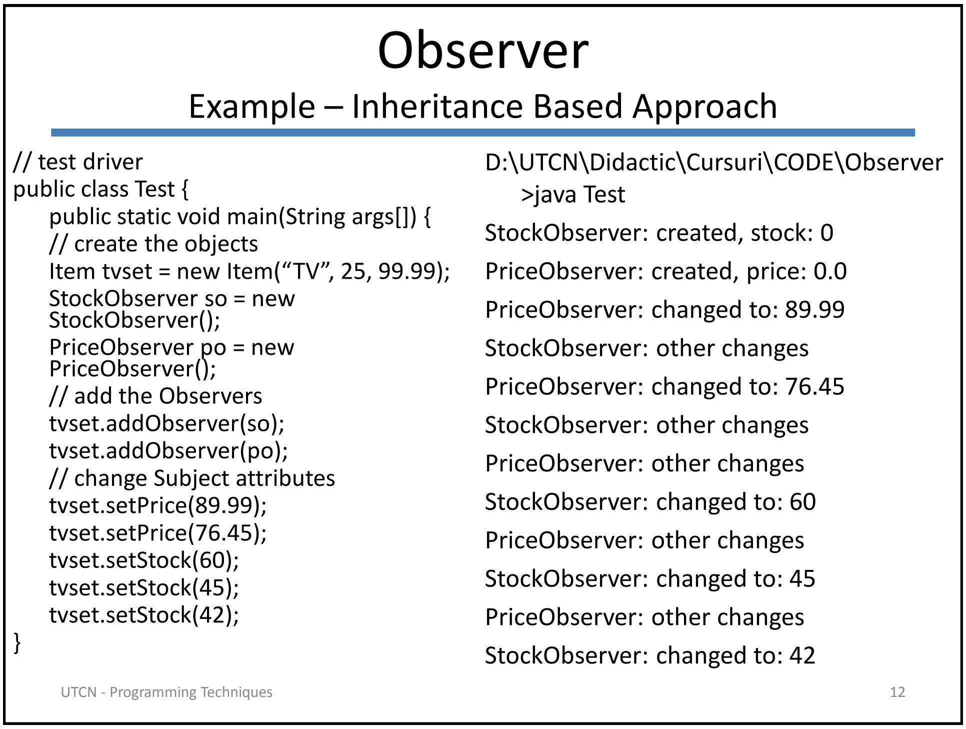


Observer

Example – Inheritance Based Approach

```
// Observer of price change
public class PriceObserver implements
    Observer {
 private double price;
 public PriceObserver() {
  price = 0.0d;
  System.out.println(“PriceObserver:
    created – price: “ + price);
 }
 public void update(Observable obs, Object
    obj){
  if(obj instanceof Double) {
   price = ((Double)obj).doubleValue();
   System.out.println(“PriceObserver:
    changed to: “ + price);
  }
  else System.out.println(“PriceObserver:
    other changes);
}
```

```
 // Observer of stock change
public class StockObserver implements Observer {
 private integer stock;
 public StockObserver() {
  stock = 0.0d;
  System.out.println(“PriceObserver: created –
                price: “ + stock);
 }
 public void update(Observable obs, Object obj) {
  if(obj instanceof Integer) {
   stock = ((Integer)obj).intValue();
   System.out.println(“PriceObserver:
      changed to: “ + stock);
  }
  else System.out.println(“PriceObserver:
                other changes);
 }
}
```

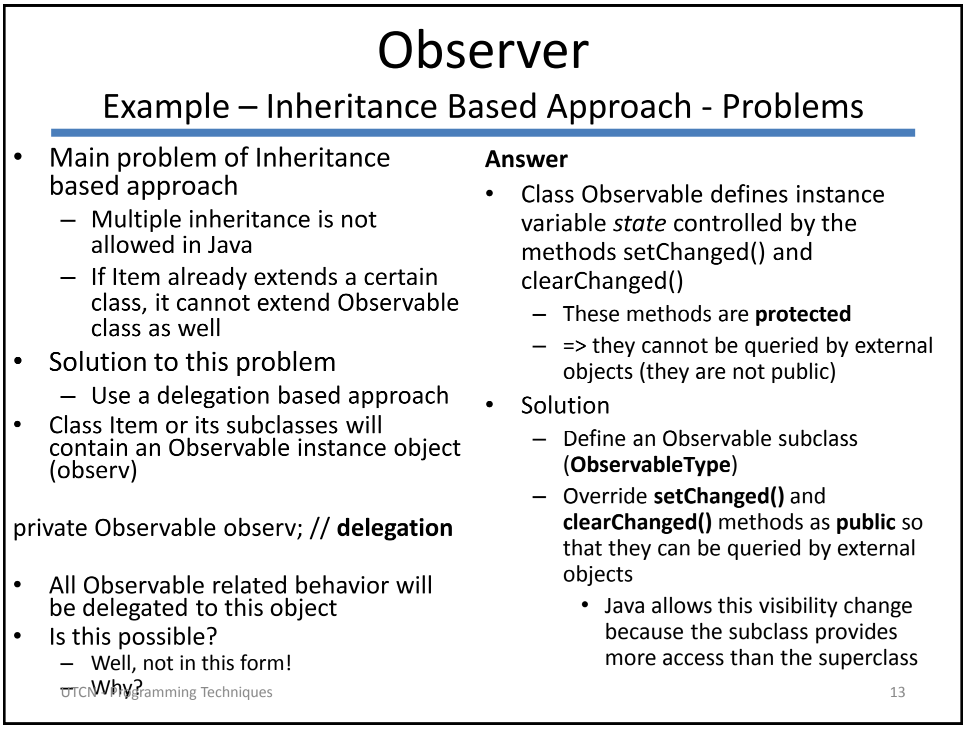


Observer

Example – Inheritance Based Approach

```
// test driver
public class Test {
    public static void main(String args[]) {
    // create the objects
    Item tvset = new Item(“TV”, 25, 99.99);
    StockObserver so = new
    StockObserver();
    PriceObserver po = new
    PriceObserver();
    // add the Observers
    tvset.addObserver(so);
    tvset.addObserver(po);
    // change Subject attributes
    tvset.setPrice(89.99);
    tvset.setPrice(76.45);
    tvset.setStock(60);
    tvset.setStock(45);
    tvset.setStock(42);
}
```

```
D:\UTCN\Didactic\Cursuri\CODE\Observer
   >java Test
StockObserver: created, stock: 0
PriceObserver: created, price: 0.0
PriceObserver: changed to: 89.99
StockObserver: other changes
PriceObserver: changed to: 76.45
StockObserver: other changes
PriceObserver: other changes
StockObserver: changed to: 60
PriceObserver: other changes
StockObserver: changed to: 45
PriceObserver: other changes
StockObserver: changed to: 42
```



Observer

Example – Inheritance Based Approach - Problems

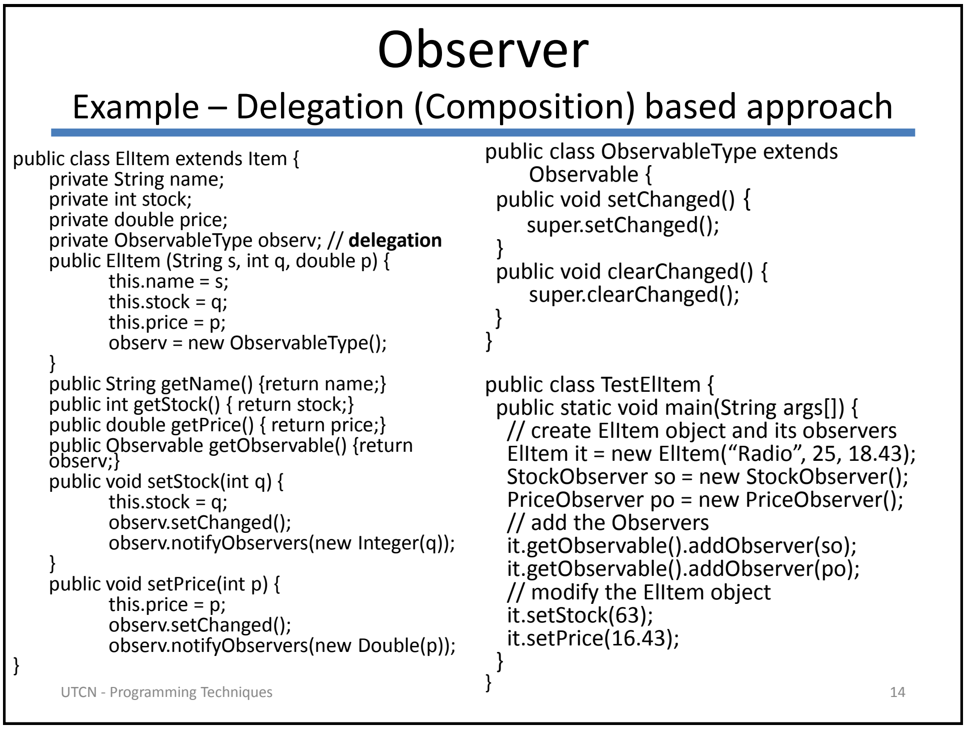
- Main problem of Inheritance based approach
  - Multiple inheritance is not allowed in Java
  - If Item already extends a certain class, it cannot extend Observable class as well
- Solution to this problem
  - Use a delegation based approach
- Class Item or its subclasses will contain an Observable instance object (observ)

private Observable observ; // **delegation**

- All Observable related behavior will be delegated to this object
- Is this possible?
  - Well, not in this form!
  - Why?

Answer

- Class Observable defines instance variable *state* controlled by the methods setChanged() and clearChanged()
  - These methods are **protected**
  - => they cannot be queried by external objects (they are not public)
- Solution
  - Define an Observable subclass (**ObservableType**)
  - Override **setChanged()** and **clearChanged()** methods as **public** so that they can be queried by external objects
    - Java allows this visibility change because the subclass provides more access than the superclass



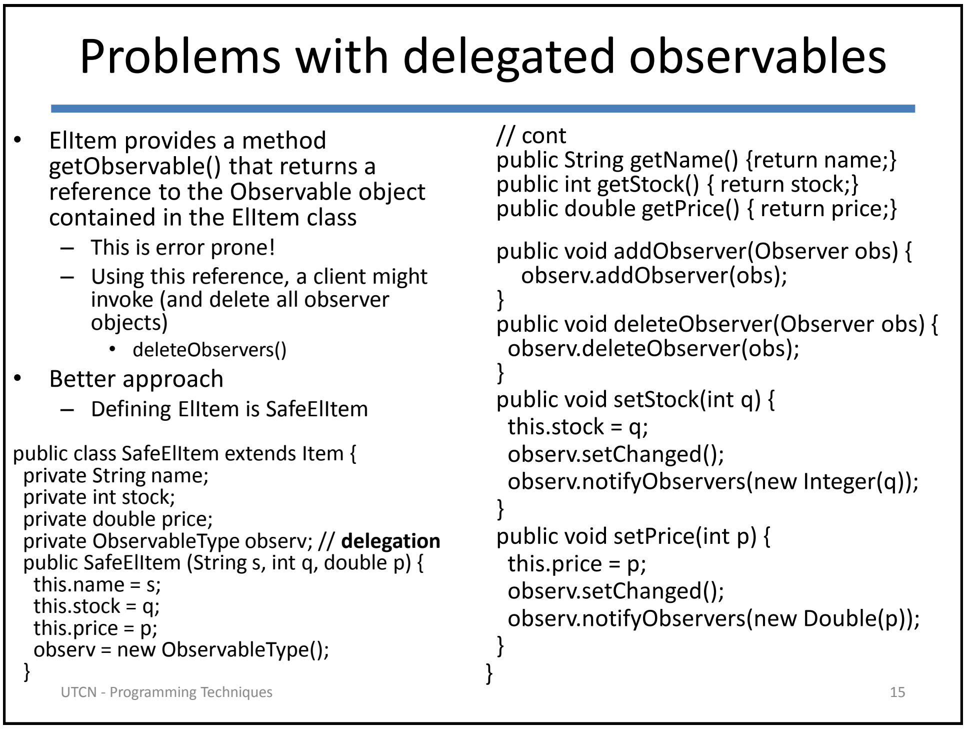
Observer

Example – Delegation (Composition) based approach

```
public class ElItem extends Item {
      private String name;
      private int stock;
      private double price;
      private ObservableType observ; // delegation
      public ElItem (String s, int q, double p) {
                this.name = s;
                this.stock = q;
                this.price = p;
                observ = new ObservableType();
      }
      public String getName() {return name;}
      public int getStock() { return stock;}
      public double getPrice() { return price;}
      public Observable getObservable() {return
      observ;}
      public void setStock(int q) {
                this.stock = q;
                observ.setChanged();
                observ.notifyObservers(new Integer(q));
      }
      public void setPrice(int p) {
                this.price = p;
                observ.setChanged();
                observ.notifyObservers(new Double(p));
}
```

```
public class ObservableType extends
       Observable {
  public void setChanged() {
     super.setChanged();
  }
  public void clearChanged() {
     super.clearChanged();
  }
}

public class TestElItem {
  public static void main(String args[]) {
    // create ElItem object and its observers
    ElItem it = new ElItem(“Radio”, 25, 18.43);
    StockObserver so = new StockObserver();
    PriceObserver po = new PriceObserver();
    // add the Observers
    it.getObservable().addObserver(so);
    it.getObservable().addObserver(po);
    // modify the ElItem object
    it.setStock(63);
    it.setPrice(16.43);
  }
}
```



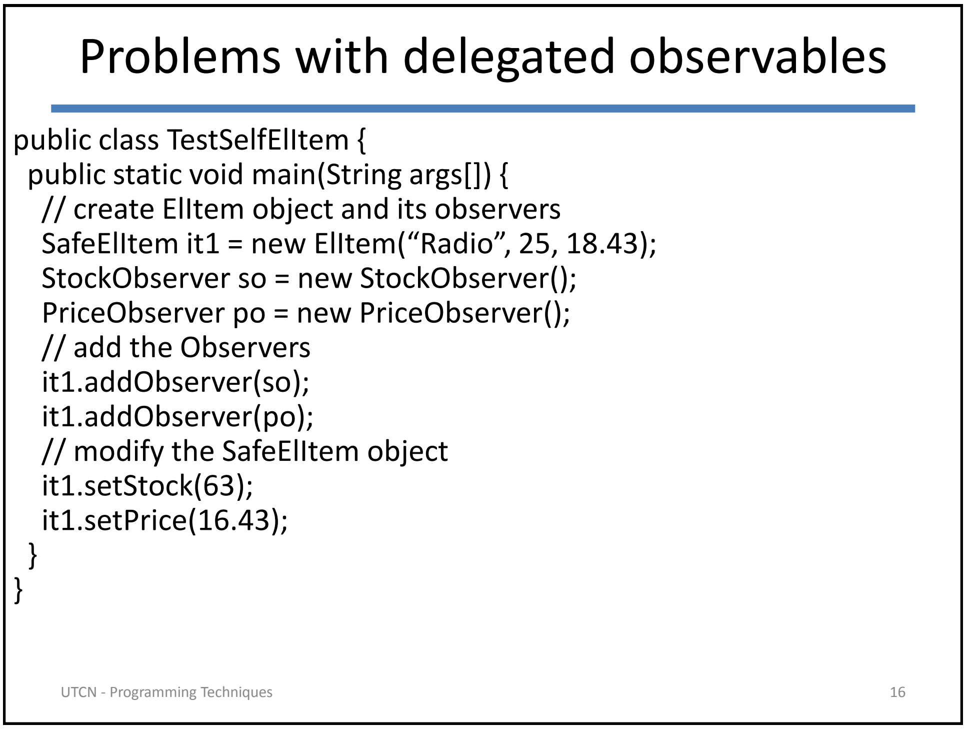
Problems with delegated observables

- ElItem provides a method getObservable() that returns a reference to the Observable object contained in the ElItem class
  - This is error prone!
  - Using this reference, a client might invoke (and delete all observer objects)
    - deleteObservers()
- Better approach
  - Defining ElItem is SafeElItem

```
public class SafeElItem extends Item {
 private String name;
 private int stock;
 private double price;
 private ObservableType observ; // delegation
 public SafeElItem (String s, int q, double p) {
   this.name = s;
   this.stock = q;
   this.price = p;
   observ = new ObservableType();
 }
```

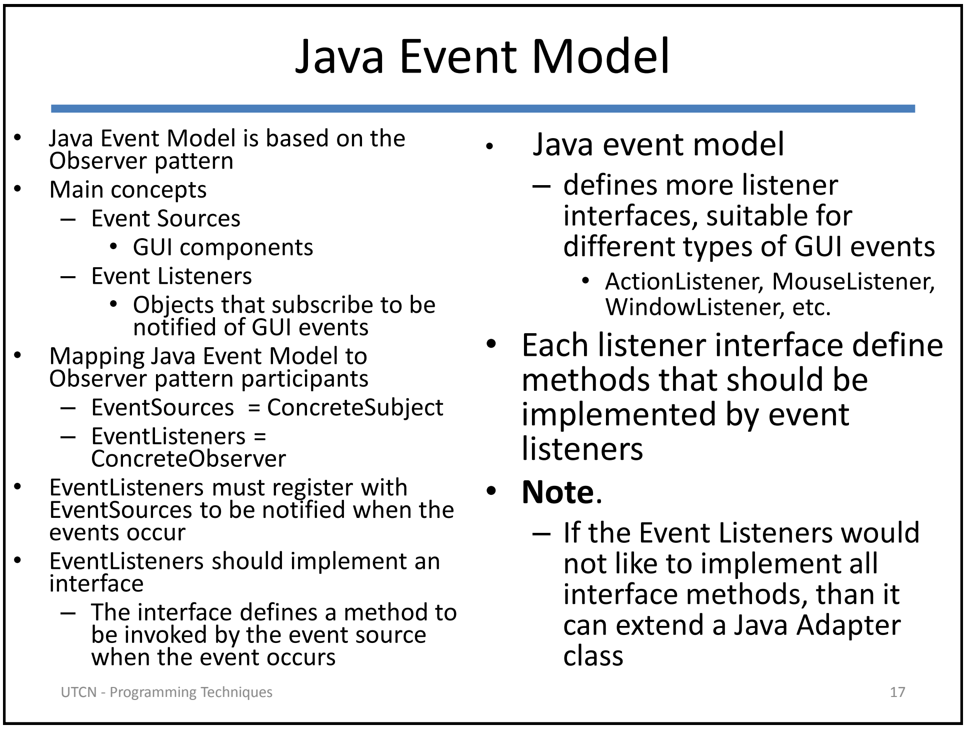
```
// cont
public String getName() {return name;}
public int getStock() { return stock;}
public double getPrice() { return price;}

public void addObserver(Observer obs) {
   observ.addObserver(obs);
}
public void deleteObserver(Observer obs) {
  observ.deleteObserver(obs);
}
public void setStock(int q) {
  this.stock = q;
  observ.setChanged();
  observ.notifyObservers(new Integer(q));
}
public void setPrice(int p) {
  this.price = p;
  observ.setChanged();
  observ.notifyObservers(new Double(p));
 }
}
```



Problems with delegated observables

```
public class TestSelfElItem {
 public static void main(String args[]) {
   // create ElItem object and its observers
   SafeElItem it1 = new ElItem(“Radio”, 25, 18.43);
   StockObserver so = new StockObserver();
   PriceObserver po = new PriceObserver();
   // add the Observers
   it1.addObserver(so);
   it1.addObserver(po);
   // modify the SafeElItem object
   it1.setStock(63);
   it1.setPrice(16.43);
 }
}
```



Java Event Model

- Java Event Model is based on the Observer pattern
- Main concepts
  - Event Sources
    - GUI components
  - Event Listeners
    - Objects that subscribe to be notified of GUI events
- Mapping Java Event Model to Observer pattern participants
  - EventSources = ConcreteSubject
  - EventListeners = ConcreteObserver
- EventListeners must register with EventSources to be notified when the events occur
- EventListeners should implement an interface
  - The interface defines a method to be invoked by the event source when the event occurs

- Java event model
  - defines more listener interfaces, suitable for different types of GUI events
    - ActionListener, MouseListener, WindowListener, etc.
- Each listener interface define methods that should be implemented by event listeners
- **Note**.
  - If the Event Listeners would not like to implement all interface methods, than it can extend a Java Adapter class