

# תוכנה 1

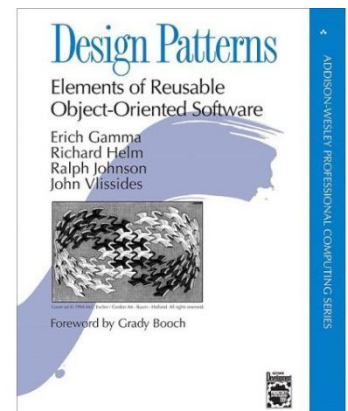
תרגול 11: Design Patterns

ומחלקות פנימיות

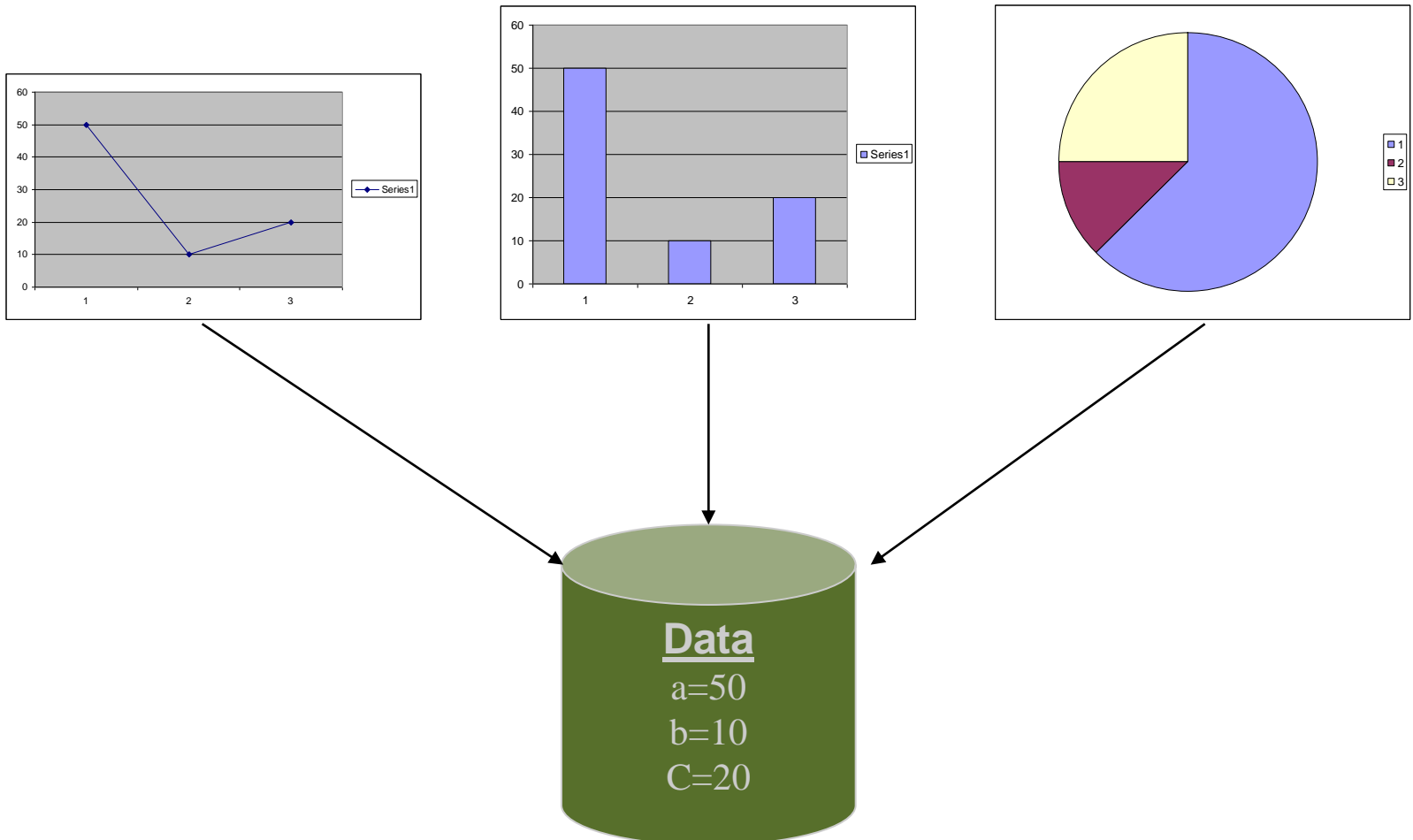
רובי בוים ומתי שמרת

# Design Patterns

- A general reusable solution to recurring design problems.
  - Not a recipe
- A higher level language for design
  - Factory, Singleton, Observer and not “this class inherits from that other class”
- *Design Patterns: Elements of Reusable Object-Oriented Software*
- Lots of information online



# Different Views



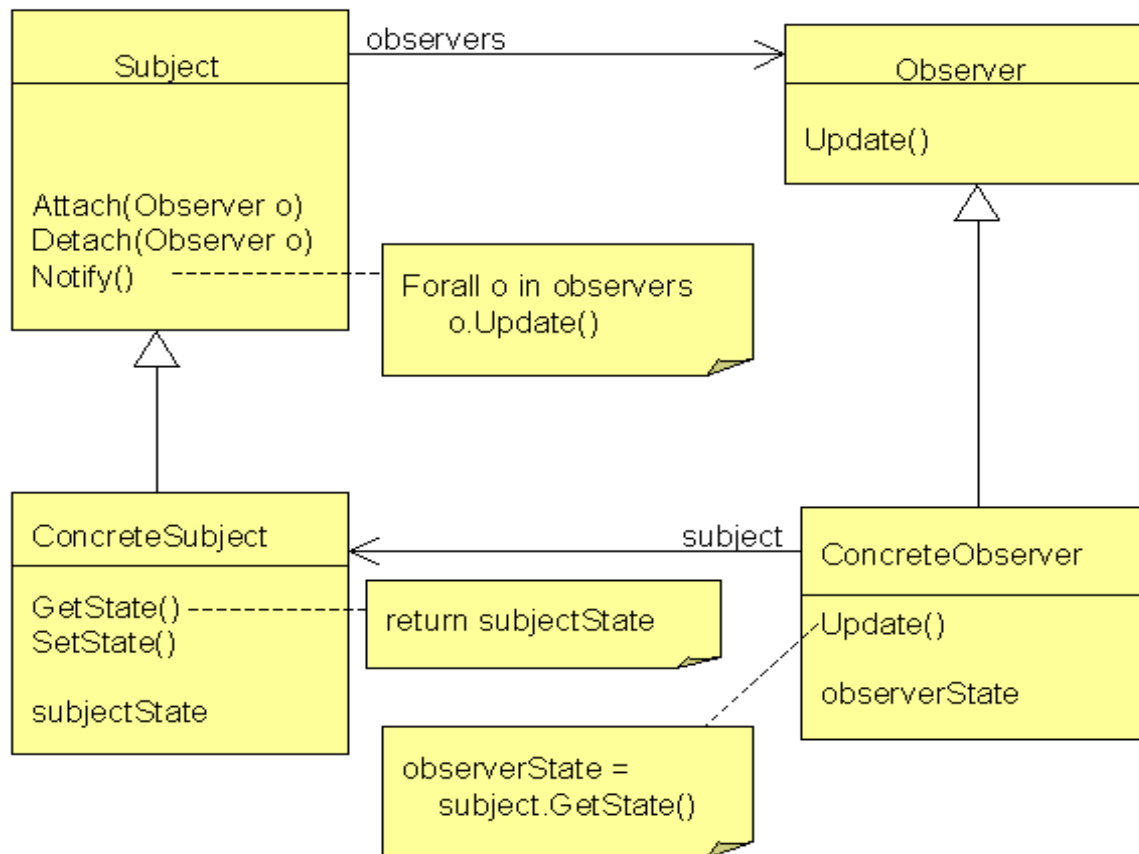
# Different Views (cont.)

---

- When the data change all views should change
  - Views dependant on data
- Views may vary, more added in the future
- Data store implementation may changes
- We want:
  - Separate the data aspect from the view one
  - Notify views upon change in data

# The Observer Design Pattern

- A.k.a publish/subscribe



# Observer and Java

---

- Java provides an Observer interface and an Observable class
- Subclass Observable to implement your own subject
  - registration and removal of observers
  - notification
- Implement Observer
- Other uses of this pattern throughout the JDK

# Observer

java.util

## Interface Observer

---

```
public interface Observer
```

A class can implement the `Observer` interface when it wants to be informed of changes in obser

**Since:**

JDK1.0

**See Also:**

[Observable](#)

---

### Method Summary

void	<a href="#">update</a> ( <a href="#">Observable</a> o, <a href="#">Object</a> arg)
------	--

This method is called whenever the observed object is changed.

# Observable

## Constructor Summary

[Observable](#) ()

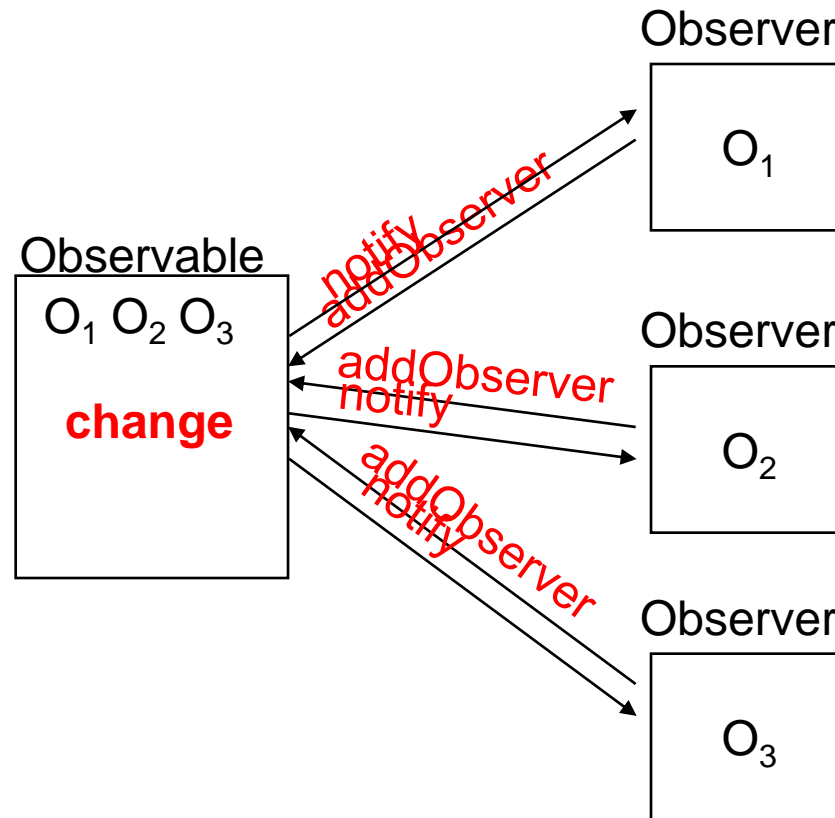
Construct an Observable with zero Observers.

## Method Summary

void	<a href="#">addObserver</a> ( <a href="#">Observer</a> o) Adds an observer to the set of observers for this object, provided that it is not the
protected void	<a href="#">clearChanged</a> () Indicates that this object has no longer changed, or that it has already notified all c hasChanged method will now return false.
int	<a href="#">countObservers</a> () Returns the number of observers of this Observable object.
void	<a href="#">deleteObserver</a> ( <a href="#">Observer</a> o) Deletes an observer from the set of observers of this object.
void	<a href="#">deleteObservers</a> () Clears the observer list so that this object no longer has any observers.
boolean	<a href="#">hasChanged</a> () Tests if this object has changed.
void	<a href="#">notifyObservers</a> () If this object has changed, as indicated by the hasChanged method, then notify all method to indicate that this object has no longer changed.
void	<a href="#">notifyObservers</a> ( <a href="#">Object</a> arg) If this object has changed, as indicated by the hasChanged method, then notify all method to indicate that this object has no longer changed.
protected void	<a href="#">setChanged</a> () Marks this Observable object as having been changed; the hasChanged method v



# Observable and Observer



# Example Code - Subject

```
public class IntegerDataBag extends Observable
    implements Iterable<Integer> {

    private ArrayList<Integer> list = new ArrayList<Integer>();

    public void add( Integer i ) {
        list.add(i);
        setChanged();
        notifyObservers();
    }

    public Iterator<Integer> iterator() {
        return list.iterator();
    }

    public Integer remove( int index ) {
        if( index < list.size() ) {
            Integer i = list.remove( index );
            setChanged();
            notifyObservers();
            return i;
        }
        return null;
    }
}
```

# Example Code - Observer

```
public class IntegerAdder implements Observer {  
  
    private IntegerDataBag bag;  
  
    public IntegerAdder( IntegerDataBag bag ) {  
        this.bag = bag;  
        bag.addObserver( this );  
    }  
  
    public void update(Observable o, Object arg) {  
        if (o == bag) {  
            println("The contents of the IntegerDataBag have changed.");  
            int sum = 0;  
            for (Integer i : bag) {  
                sum += i;  
            }  
            println("The new sum of the integers is: " + sum);  
        }  
    }  
  
    ...  
}
```



# Nested Classes

# מחלקות מקוננות

# מחלקה מקוננת

■ מחלקה שמוגדרת בתוך מחלקה אחרת

.1 סטטית (static member)

.2 לא סטטית (nonstatic member)

.3 אנונימית (anonymous)

.4 מקומית (local)

Inner classes

```
class Outer {  
    static class NestedButNotInner {  
        ...  
    }  
    class Inner {  
        ...  
    }  
}
```

# בשביל מה זה טוב

## ■ קיבוץ לוגי

אם עושים שימוש בטיפוס רק בהקשר של טיפוס מסוים נטמיע את הטיפוס כדי לשמר את הקשר הלוגי

## ■ הכמסה מוגברת

על ידי הטמעת טיפוס אחד באחר אנו חושפים את המידע הפרטי רק לטיפוס המוטמע ולא לכולם

## ■ קריאות

מיקום הגדרת טיפוס בסמוך למקום השימוש בו

# תכונות משותפות

- למחלקה מקוננת יש גישה לשדות הפרטיים של המחלקה העוטפת ולהפך
- הנראות של המחלקה היא עבור "צד שלישי"
- אלו מחלקות (כמעט) רגילות לכל דבר ועניין
- יכולות להיות אבסטרקטיות, לממש מנשקים, לרשת ממחלקות אחרות וכדומה

# Static Member Class

■ מחלקה רגילה ש"במקרה" מוגדרת בתוך מחלקה אחרת

■ החוקים החלים על איברים סטטיים אחרים חלים גם על מחלקות סטטיות

■ גישה לשדות / פונקציות סטטיים בלבד

■ גישה לאיברים לא סטטיים רק בעזרת הפניה לאובייקט

■ גישה לטיפוס בעזרת שם המחלקה העוטפת

`OuterClass.StaticNestedClass`

■ יצירת אובייקט

`OuterClass.StaticNestedClass nested =`

`new OuterClass.StaticNestedClass ();`

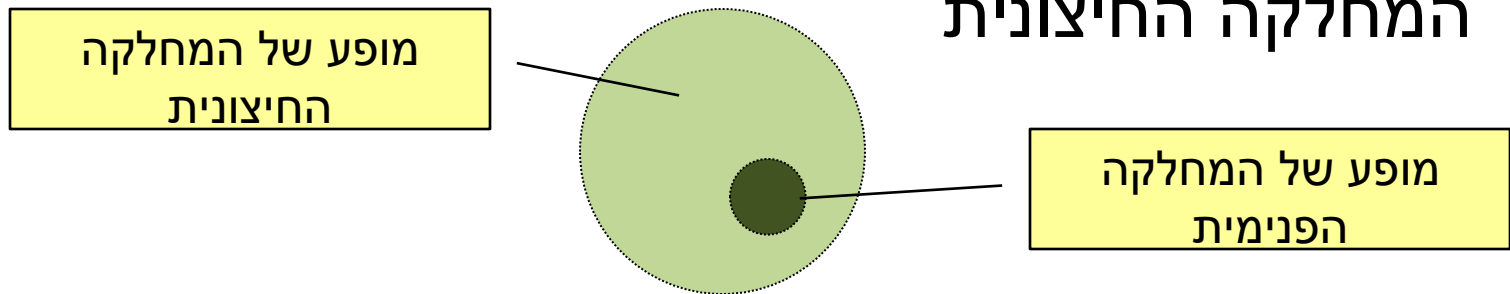


# AbstractMap

```
public abstract class AbstractMap<K,V> implements Map<K,V> {  
  
    public static class SimpleEntry<K,V>  
        implements Entry<K,V>, java.io.Serializable {  
        private final K key;  
        private V value;  
  
        ...  
    }  
  
    ...  
}
```

# Non-static Member Class

- כל מופע של המחלקה הפנימית משויך למופע של המחלקה החיצונית



- השיוך מבוצע בזמן יצירת האובייקט ואינו ניתן לשינוי
  - באובייקט הפנימי קיימת הפניה לאובייקט החיצוני (qualified this)

# House Example

```
public class House {
    private String address;

    public class Room {
        // implicit reference to a House
        private double width;
        private double height;

        public String toString(){
            return "Room inside: " + address;
        }
    }
}
```

גישה למשתנה פרטי לא סטטי

# Inner Classes

```
public class House {  
    private String address;  
    private double height;  
    public class Room {  
        // implicit reference to a House  
        private double height;  
        public String toString() {  
            return "Room height: " + height  
                + " House height: " + House.this.height;  
        }  
    }  
}
```

Height of *House*

Height of *Room*

Height of *Room*  
Same as *this.height*

# AbstractList

```
public abstract class AbstractList<E> extends
    AbstractCollection<E> implements List<E> {
    public Iterator<E> iterator() {
        return new Itr();
    }

    private class Itr implements Iterator<E> {
        ...
    }

    private class ListItr extends Itr implements
        ListIterator<E> {
        ...
    }
}
```

# יצירת מופעים

■ כאשר המחלקה העוטפת יוצרת מופע של עצם מטיפוס המחלקה הפנימית אזי העצם נוצר בהקשר של העצם היוצר

■ כאשר עצם מטיפוס המחלקה הפנימית נוצר מחוץ למחלקה העוטפת, יש צורך בתחביר מיוחד

```
outerObject.new InnerClassConstructor
```

# מחלקות אנונימיות

- מחלקה ללא שם
- הגדרה ויצירת מופע בנקודת השימוש
- מגבלות:
  - חייבת לרשת מטיפוס קיים (מנשק או מחלקה)
  - לא ניתן להגדיר איברים סטטים, לא ניתן להשתמש בהקשר שדורש שם (instanceof), לא ניתן לרשת ממספר טיפוסים, לקוחות מוגבלים לממשק של טיפוס האב
- מחלקה אנונימית צריכה להיות קצרה כדי לא לפגוע בקריאות של הקוד

# דוגמאות שימוש

## Function object (functor) ■

### מיון מחרוזות לפי אורך ■

```
Arrays.sort(stringArray, new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

## מימוש איטרטור ■

```
public Iterator<E> iterator() {  
    return new Iterator<E>() {  
        boolean hasNext() {...}  
        E next() {...}  
        void remove() {...}  
    };  
}
```



# המרה ממערך לרשימה

```
static List<Integer> intArrayAsList(final int[] a) {  
    if (a == null)  
        throw new IllegalArgumentException();  
    return new AbstractList<Integer>() {  
        public Integer get(int i) {  
            return a[i];  
        }  
        public Integer set(int i, Integer val) {  
            int oldVal = a[i];  
            a[i] = val;  
            return oldVal;  
        }  
        public int size() {  
            return a.length;  
        }  
    };  
}
```

גישה למשתנים מקומיים  
שהוגדרו final

# מחלקות מקומיות

## ■ מוגדרות בתוך מתודות

■ יש להם שם וניתן להשתמש בהם מספר פעמים

■ אובייקט עוטף רק אם הוגדרו בהקשר לא סטטי; לא ניתן להגדיר משתני סטטיים

■ המחלקה הפנימית תוכל להשתמש גם במשתנים

מקומיים של המתודה אבל רק אם הם הוגדרו כ-  
**final**

# המרה ממערך לרשימה

```
static List<Integer> intArrayAsList(final int[] a) {
    if (a == null)
        throw new IllegalArgumentException();
    class IntegerList extends AbstractList<Integer> {
        public Integer get(int i) {
            return a[i];
        }
        public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val;
            return oldVal;
        }
        public int size() {
            return a.length;
        }
    }
    return new IntegerList();
}
```