

תוכנה 1

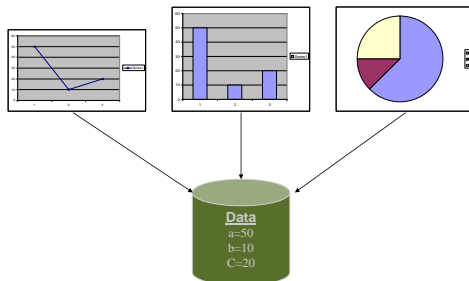
תרגול 10: Design Patterns
ומחלקות פנימיות
הדס צור ואסף זריצקי

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

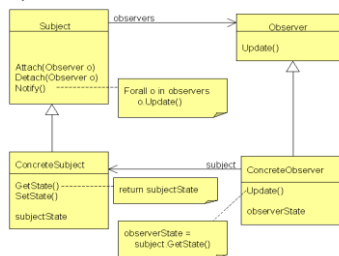


Different Views (cont.)

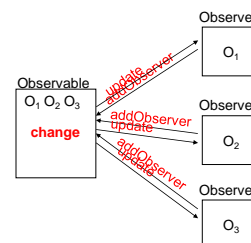
- When the data change all views should change
 - Views dependant on data
- Views may vary, more added in the future
- Data store implementation may changes
- We want:
 - Separate the data aspect from the view one
 - Notify views upon change in data

The Observer Design Pattern

- A.k.a publish/subscribe



Observable and Observer



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

7

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 **update**([Observable](#) o, [Object](#) arg)
This method is called whenever the observed object is changed.

8

Observable

Constructor Summary

[Observable\(\)](#)
Constructs an **Observable** with zero **Observers**.

Method Summary

void **addObserver**([Observer](#) o)
Add an observer to the set of observers for this object, provided that it is not the
protected void **clearChanged**()
Indicates that this object has no longer changed, or that it has already notified all
hasChanged method will now return false.
int **countObservers**()
Return the number of observers of this **Observable** object.
void **deleteObserver**([Observer](#) o)
Delete an observer from the set of observers of this object.
void **deleteObservers**()
Clear the observer list so that this object no longer has any observers.
boolean **hasChanged**()
Tests if this object has changed.
void **notifyObservers**()
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 **notifyObservers([Object](#) 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 **setChanged**()
Marks this **Observable** object as having been changed; the **hasChanged** method

9

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);
        }
    }
}
```

11

מחלקות פנימיות (מקוננות) Inner (Nested) Classes

12

Inner Classes

- מחלקה פנימית היא מחלקה שהוגדרה בתחום (Scope – בין המסולליים) של מחלקה אחרת

■ דוגמא:

```
public class House {  
    private String address;  
    public class Room {  
        private double width;  
        private double height;  
    }  
}
```

שימוש לב!

Room אינה שדה של
House המחלקה

13

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

14

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 ממשקים, לרשת ממחלקות אחרות וכדומה

15

"סוגי" מחלקות פנימיות

- מחלקה שמוגדרת בתוך מחלקה אחרת

1. סטטית (static member)
2. לא סטטית (nonstatic member)
3. אנונימית (anonymous)
4. מקומית (local)

Inner classes

16

מחלקות פנימיות

- הגדרת מחלקה כפנימית מרמזת על היחס בין המחלקה הפנימית והמחלקה העוטפת:

- למחלקה הפנימית יש משמעות רק בהקשר של המחלקה העוטפת
- למחלקה הפנימית יש הכרות אינטימית עם המחלקה העוטפת
- המחלקה הפנימית היא מחלקת עזר של המחלקה העוטפת

■ דוגמאות:

```
Collection -> Iterator  
Body -> Brain  
List -> Cell  
מבני נתונים המוגדרים ברקורסיה:
```

17

Inner Classes

- ב Java כל מופע של עצם מטיפוס המחלקה הפנימית משויך לעצם מטיפוס המחלקה העוטפת

■ השלכות

- תחביר מיוחד לבנאי
- לעצם מטיפוס המחלקה הפנימית יש שדה הפנייה שמיוצר אוטומטית לעצם מהמחלקה העוטפת
- כתוצאה לכך יש למחלקה הפנימית גישה לשדות ולשרותים (אפילו פרטיים!) של המחלקה העוטפת ולהיפך

18

Inner Classes

```
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;
        }
    }
}
```

19

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;
        }
    }
}
```

20

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

21

יצירת מופע ע"י המחלקה העוטפת

```
public class House {
    private String address;
    public void test() {
        Room r = new Room();
        System.out.println( r );
    }

    public class Room {
        ...
    }
}
```

22

יצירת מופע שלא ע"י המחלקה החיצונית

```
public class Test {
    public static void main(String[] args){
        House h = new House();
        House.Room r = h.new Room();
    }
}
```

`outerObject.new InnerClassConstructor`

23

Static Nested Classes

- ניתן להגדיר מחלקה פנימית כ `static` ובכך ליצין שהיא אינה קשורה למופע מסוים של המחלקה העוטפת
- הדבר אנלוגי למחלקה שכל שרותיה הגודרו כ-`static` והיא משמשת כספרייה עבור מחלקה מסוימת
- (בשפת C++ יחס זה מושג ע"י הגדרת יחס `friend`)

24

```
public class House {
    private String address;
    public static class Room {
        public String toString() {
            return "Room " + address;
        }
    }
}
```

Error: this room is not related to any house

Not related to any specific house

```
public class Test {
    public static void main(String[] args) {
        House.Room r = new House.Room();
        ...
    }
}
```

`new OuterClassName.InnerClassConstructor`

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;

        ...
    }
    ...
}
```

26

הגנה על מחלקות פנימיות

- אם המחלקה הפנימית אינה ציבורית (אינה מוגדרת public), הטיפוס שלה מוסתר, אבל עצמים מהמחלקה אינם מוסתרים אם יש התייחסות אליהם

```
public class Outer {
    private static class Inner implement Interface {
        ...
    }

    public static Interface getInner() {
        return new Inner ();
    }
}
```

```
Interface i = new Outer.Inner(); //error
Interface i = Outer.getInner(); // ok
```

27

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

28

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() {...}
    };
}
```

29

מחלקות מקומיות - מחלקות פנימיות בתוך מתודות

- ניתן להגדיר מחלקה פנימית בתוך שירות של המחלקה העוטפת
- הדבר מגביל את תחום ההכרה של אותה מחלקה לתחום השירות בלבד
- המחלקה הפנימית תוכל להשתמש במשתנים מקומיים של המתודה רק אם הם הוגדרו כ-final (מדוע?)

30

מחלקות מקומיות

```
public class Test {  
    ...  
    public void test () {  
        class Info {  
            private int x;  
            public Info(int x) {this.x=x;}  
            public String toString() {  
                return "** " + x + "**" ;  
            }  
        };  
        Info info = new Info(0);  
        System.out.println(info);  
    }  
}
```

31

שימוש במשתנים מקומיים

```
public class Test {  
    public void test (int x) {  
        final int y = x+3;  
        class Info {  
            public String toString(){  
                return "**" + y + "**";  
            }  
        };  
        System.out.println( new Info());  
    }  
}
```

32

הידור של מחלקות פנימיות

- המהדר (קומפיילר) יוצר קובץ class. עבור כל מחלקה. מחלקה פנימית אינה שונה במובן זה ממחלקה רגילה
- שם המחלקה הפנימית יהיה Outer\$Inner.class
- אם המחלקה הפנימית אנונימית, שם המחלקה שיוצר הקומפיילר יהיה Outer\$1.class

33