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

- 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
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

```java
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;
    }
}
```

**Observer**

```java
public interface Observer {
    public void update(Observable o, Object arg);
}
```

**Example Code - Subject**

```java
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);
        }
    }
    ...
}
```

**Example Code - Observer**

```java
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);
        }
    }
    ...
}
```

**Inner (Nested) Classes**
Inner Classes

Inner Classes are classes defined within another class.

**Example:**

```java
public class House {
    private String address;

    public class Room {
        private double width;
        private double height;
    }
}
```

Inner Classes are useful for:

1. **Building logical constructs**
   - When using a specific type within the context of another type, it is embedded to preserve the logical relationship.

2. **Enhanced visibility**
   - By embedding a type in another, we expose only the embedded type, not all others.

3. **Positioned declarations**
   - By positioning the type definition near where it is used.

4. **Common characteristics**
   - A class is often a helper class for another class.
   - Inner classes can be abstract, interfaces, or similarly.

Inner Classes can be:

- **Static** (static member)
- **Nonstatic** (nonstatic member)
- **Anonymous** (anonymous)
- **Local** (local)

Inner Classes can be:

- Used directly on types.
- Used to define simple, powerful mechanisms.
- Used to create new types from existing types.
- Used to create new types from existing types.
- Used to create new types from existing types.
- Used to create new types from existing types.

Translation:

**Inner Classes**

A class defined within another class.

**Example:**

```java
public class House {
    private String address;

    public class Room {
        private double width;
        private double height;
    }
}
```

Inner Classes are:

- **Static** (static member)
- **Nonstatic** (nonstatic member)
- **Anonymous** (anonymous)
- **Local** (local)

Translation:

**Inner Classes**

A class defined within another class.

**Example:**

```java
public class House {
    private String address;

    public class Room {
        private double width;
        private double height;
    }
}
```
Inner Classes

```java
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;
        }
    }
}
```

Static Nested Classes

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

**Creation of a nested class object:**
- A nested class object is created inside the enclosing class.
- A nested class object is created outside the enclosing class.

**Creation of a nested class object outside:**
- Use special syntax to create a nested class object outside the enclosing class.

**Static Nested Classes:**
- A nested class can be defined as static.
- The methods of the enclosing class can access static nested classes.
- In C++, you can use the static keyword to declare methods within a class.
- Use the `static` keyword to define static nested classes.

**Creation of a nested class object outside:**
- Use the `new` keyword to create a static nested class object.

```java
public class Test {
    public static void main(String[] args) {
        House h = new House();
        House.Room r = h.new Room();
    }
}
```
```java
public class House {
    private String address;
    public static class Room {
        public String toString()
            return "Room " + address;
    }
}

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

```
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;
        ...
    }
    ...
}
```

```
public class Outer {
    private static class Inner implement Interface {
        ...
    }
    public static Interface getInner()
        return new Inner();
}
```

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

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

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