

# Introduction to Java (cs2514)

## Lecture 11: Nested Classes

M. R. C. van Dongen

March 5, 2018

## Outline

[Observer Pattern](#)[Windows](#)[Events](#)[Nested Classes](#)[Nested Interfaces](#)[Question Time](#)[For Thursday](#)[Acknowledgements](#)[References](#)[About this Document](#)

- This lecture is about *nested classes*:
  - Classes inside other classes.
- Nested classes are frequently used with GUI applications.
- We study the famous *observer design pattern*:
  - We have a *subject* and one or more *observers*.
  - The subject may be viewed as a newspaper.
  - The observers are subscribed to the newspaper.
  - The subscribed observers are waiting for news from the subject.
  - The subject broadcasts news items to its subscribed observers.
    - Done by calling observer's `notify()` method.
- The observer pattern is frequently used with GUIs.
  - The GUIs are used only to show how the observer pattern is used.
  - There won't be any questions about GUIs for the exam.

# The Observer Design Pattern

- The *observer pattern* is a commonly used design pattern.
- It defines a one-to-many object dependency.
- The dependency ensures that the object's dependents are automatically updated when the object's state changes [Gamma et al. 2008].
- AKA Dependents, Publish-Subscribe [Eric Freeman, and Elisabeth Freeman 2005, Pages 44–78], and Event-Listener.

# The Observer Pattern (Continued)

## The Source of the News: A Newspaper

- There is one Subject.
- There are zero or more Observers.
- An Observer can be attached to the Subject.
- An Observer can be detached from the Subject.
- If the Subject's state changes it updates all its Observers.
  - This is done by calling each Subject's `update()` method.

# The Observer Pattern (Continued)

## Potential Readers

- There is one Subject.
- There are zero or more Observers.
- An Observer can be attached to the Subject.
- An Observer can be detached from the Subject.
- If the Subject's state changes it updates all its Observers.
  - This is done by calling each Subject's `update()` method.

# The Observer Pattern (Continued)

Subscribe as Reader to the Newspaper

- There is one Subject.
- There are zero or more Observers.
- An Observer can be attached to the Subject.
- An Observer can be detached from the Subject.
- If the Subject's state changes it updates all its Observers.
  - This is done by calling each Subject's `update()` method.

## The Observer Pattern (Continued)

## Unsubscribe as Reader to the Newspaper

- ❑ There is one Subject.
- ❑ There are zero or more Observers.
- ❑ An Observer can be attached to the Subject.
- ❑ An Observer can be detached from the Subject.
- ❑ If the Subject's state changes it updates all its Observers.
  - ❑ This is done by calling each Subject's update( ) method.

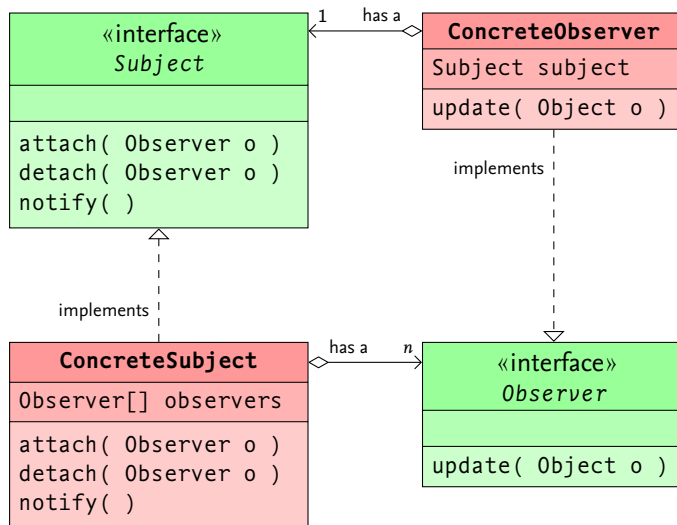
# The Observer Pattern (Continued)

Inform the Readers about News

- There is one Subject.
- There are zero or more Observers.
- An Observer can be attached to the Subject.
- An Observer can be detached from the Subject.
- If the Subject's state changes it updates all its Observers.
  - This is done by calling each Subject's `update()` method.



# Class Diagram of the Observer Pattern



# Case Study

- We have a newspaper and readers of the newspaper.
- The readers can subscribe and unsubscribe.
- The newspaper informs the subscribers about new newsitems.

# Implementing the Interfaces

## Java

```
public interface Subject {  
    // Subscribe to this newspaper.  
    public void attach( Observer subscriber );  
    // Unsubscribe from this newspaper.  
    public void detach( Observer subscriber );  
    // Notify this newspaper of a news event.  
    public void notify( String event );  
}
```

## Java

```
public interface Observer {  
    // Inform this subscriber about a published event.  
    public void update( String event );  
}
```

# Implementing the Interfaces

## Java

```
public interface Subject {  
    // Subscribe to this newspaper.  
    public void attach( Observer subscriber );  
    // Unsubscribe from this newspaper.  
    public void detach( Observer subscriber );  
    // Notify this newspaper of a news event.  
    public void notify( String event );  
}
```

## Java

```
public interface Observer {  
    // Inform this subscriber about a published event.  
    public void update( String event );  
}
```

# Implementing the Interfaces

## Java

```
public interface Subject {  
    // Subscribe to this newspaper.  
    public void attach( Observer subscriber );  
    // Unsubscribe from this newspaper.  
    public void detach( Observer subscriber );  
    // Notify this newspaper of a news event.  
    public void notify( String event );  
}
```

## Java

```
public interface Observer {  
    // Inform this subscriber about a published event.  
    public void update( String event );  
}
```

# Implementing the Interfaces

## Java

```
public interface Subject {  
    // Subscribe to this newspaper.  
    public void attach( Observer subscriber );  
    // Unsubscribe from this newspaper.  
    public void detach( Observer subscriber );  
    // Notify this newspaper of a news event.  
    public void notify( String event );  
}
```

## Java

```
public interface Observer {  
    // Inform this subscriber about a published event.  
    public void update( String event );  
}
```

# A Concrete Observer

## Java

```
public class ConcreteObserver implements Observer {
    // The name of the subscriber.
    final String name;

    public ConcreteObserver( final String name ) {
        this.name = name;
    }

    // Inform this subscriber about a published event.
    @Override
    public void update( final String event ) {
        System.out.println( name + " reading: " + event );
    }

    @Override
    public String toString( ) {
        return name;
    }
}
```

# A Concrete Subject

## Java

```
public class ConcreteSubject implements Subject {  
  
    // The name of this newspaper.  
    private final String name;  
    // The subscribers of this newspaper.  
    private final ArrayList<Observer> subscribers;  
  
    public ConcreteSubject( final String name ) {  
        subscribers = new ArrayList<Observer>( );  
        this.name = name;  
    }  
  
    @Override  
    public String toString( ) {  
        return name;  
    }  
  
    // omitted  
  
}
```



# A Concrete Subject (Continued)

## Java

```
public class ConcreteSubject implements Subject {

    // omitted

    @Override // Subscribe a new customer.
    public void attach( final Observer subscriber ) {
        System.out.println( subscriber + " subscribes to " + this );
        subscribers.add( subscriber );
    }

    @Override // Unsubscribe an existing customer.
    public void detach( final Observer subscriber ) {
        System.out.println( subscriber + " unsubscribes from " + this );
        subscribers.remove( subscriber );
    }

    @Override // Inform this newspaper about hot news item.
    public void notify( final String news ) {
        // Inform all subscribers about the news item.
        System.out.println( this + " got news item: " + news );
        for( Observer subscriber : subscribers ) {
            subscriber.update( news );
        }
    }
}
```

# The Main Class

## Java

```
public class Main {  
    public static void main( String[] args ) {  
        final Subject eolas = new ConcreteSubject( "Eolas" );  
        final Subject examiner = new ConcreteSubject( "Examiner" );  
  
        final Observer john = new ConcreteObserver( "John" );  
        final Observer jane = new ConcreteObserver( "Jane" );  
        final Observer eoin = new ConcreteObserver( "Eoin" );  
  
        examiner.attach( john );  
        examiner.attach( eoin );  
        examiner.attach( jane );  
        eolas.attach( jane );  
  
        eolas.notify( "Assignment 1 to be handed back on Thursday." );  
        examiner.notify( "Eolas still reporting fake news." );  
  
        examiner.detach( jane );  
  
        examiner.notify( "No news today." );  
    }  
}
```

[Outline](#)[Observer Pattern](#)[Case Study](#)[Windows](#)[Events](#)[Nested Classes](#)[Nested Interfaces](#)[Question Time](#)[For Thursday](#)[Acknowledgements](#)[References](#)[About this Document](#)

# Sample Output

## Unix Session

\$

# Sample Output

## Unix Session

```
$ java Main
```

# Sample Output

## Unix Session

```
$ java Main
John subscribes to Examiner
Eoin subscribes to Examiner
Jane subscribes to Examiner
Jane subscribes to Eolas
Eolas got news item: Assignment 1 to be handed back on Thursday.
Jane reading: Assignment 1 to be handed back on Thursday.
Examiner got news item: Eolas still reporting fake news.
John reading: Eolas still reporting fake news.
Eoin reading: Eolas still reporting fake news.
Jane reading: Eolas still reporting fake news.
Jane unsubscribes from Examiner
Examiner got news item: Sorry folks: No news today.
John reading: Sorry folks: No news today.
Eoin reading: Sorry folks: No news today.
$
```

- Without a window you can't write a GUI application.
- In Java a window is represented as a JFrame object.
- The JFrame is where you put your window's *widgets* in.
- Possible widgets are
  - Buttons,
  - Checkboxes,
  - Sliders,
  - Dialogue boxes,
  - Text fields,
  - And so on.
- The appearance of a JFrame may differ from os to os.

# Open that Window

- Create a JFrame.

## Java

```
JFrame frame = new JFrame( <title string> );
```

- Set the JFrame's closing operation.

## Java

```
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

- Make one or several widgets and add them to the JFrame.

## Java

```
JButton button = new JButton( "Click me" );  
frame.getContentPane( ).add( button );
```

- Give the JFrame a size and make it visible.

## Java

```
frame.setSize( 300, 300 );  
frame.setVisible( true );
```



# A Full Program

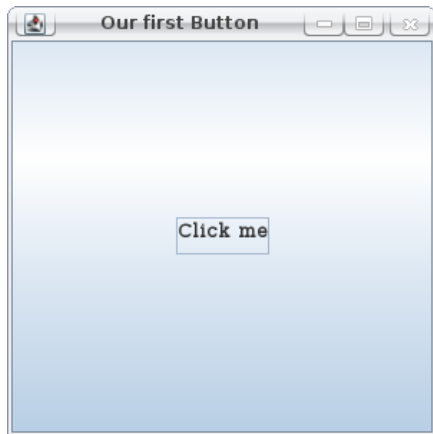
## Java

```
import javax.swing.*;

public class DummyButton {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "Our second Button" );
        JButton button = new JButton( "Click me" );
        frame.getContentPane( ).add( button );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize( 300, 300 );
        frame.setVisible( true );
    }
}
```

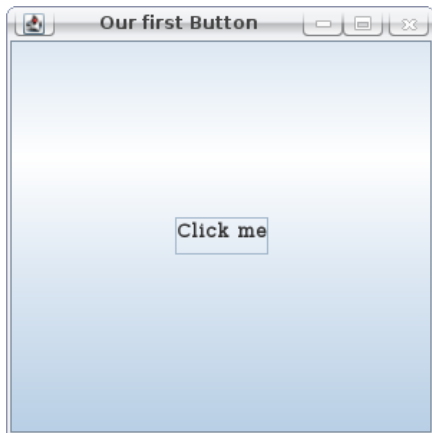


# Our First Button



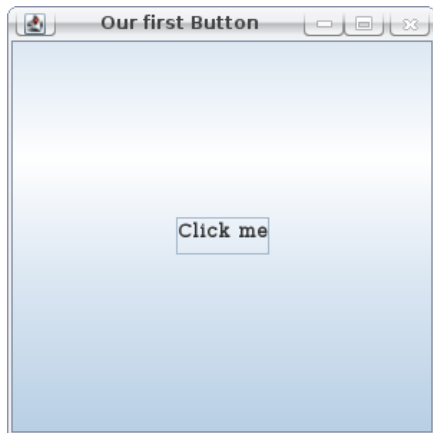
# Our First Button

But ...



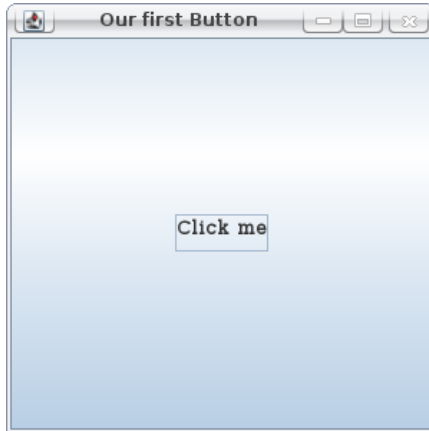
# Our First Button

But ..., When We Click the Button



# Our First Button

But ..., When We Click the Button, Nothing Happens:-)



- It is quite obvious our button did nothing when we clicked it.
  - After all, we didn't tell it what to do.
  - Let alone, *how*, and *when*.
- The JButton class knows *when* its buttons are clicked:
  - Event:** Clicking the button generates a button event.
- To let the button do something *when* it's clicked we need:
  - Listener:** Listener to button events.
  - Handler:** Listener instance method that is called for each event.

# Summary

- 1 The button event is activated when the button is clicked.
- 2 The button event triggers the button event listener.
- 3 The button event listener carries out the button event handler.

# That Sounds Familiar: The Observer Pattern

- The JButton is the Subject.
- Clicking the JButton is a *user action*.
- The JButton turns the user action into a button event object.
  - It may be thought of as a call to `notify( event )`.
- The button event is broadcast to all button event listeners.
- The Observers are the button event listeners.
- Each Observer implements its button event handler.
  - Each event handler is a dedicated `update( )` method.
  - The call `update( event )` sends the event to the listener.
    - The button sends the event by calling `update( )`.
    - By doing things in `update( )`'s body, the listener responds.

# Creating an Event Listener

- An event listener class implements an event listener interface.
  - Button event listeners implement the button listener interface,
  - Mouse event listeners implement the mouse listener interface,
  - And so on.
- Some interfaces have more than one `notify( )` method.
- For buttons you usually only want to know when it's clicked.
  - However, it is possible to distinguish between events pressing and releasing a button.
- The “click events” for JButtons are **ActionEvent** objects.
- So our listener must implement the **ActionListener** interface.
  - The method `actionPerformed( ActionEvent event )` in the interface is equivalent to the Observer's `update( )` method.



# Example

## Java

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.Color;

public class SimpleGUI implements ActionListener {
    private final JButton button;
    private boolean alert;

    public static void main( String[] args ) {
        JFrame frame = <Create JFrame>
        SimpleGUI gui = new SimpleGUI( );
        <Remaining JFrame-related statements.>
    }

    public SimpleGUI( ) {
        button = new JButton( "Click me" );
        button.addActionListener( this );
    }

    @Override
    public void actionPerformed((ActionEvent event) {
        button.setText( alert ? "Alarm" : "No panic" );
        button.setBackground( alert ? Color.red : Color.green );
        alert = !alert;
    }
}
```



# A Button with a Counter

## Java

```
import javax.swing.*;
import java.awt.event.*;

public class CountingButton implements ActionListener {
    private int clicks;
    private final JButton button;

    public static void main( String[] args ) {
        JFrame frame = <Create JFrame>
        SimpleGUI gui = new SimpleGUI( );
        <Remaining JFrame-related statements.>
    }

    public CountingButton( ) {
        clicks = 0;
        button = new JButton( "Click me" );
        button.addActionListener( this );
    }

    @Override
    public void actionPerformed((ActionEvent event) {
        String text = "# clicks = " + ++ clicks + ". Try again.";
        button.setText( text );
    }
}
```

# Nested Classes

- Classes defined in other classes are called *nested classes*.
- There are two kinds of nested classes.
  - Static classes: these are called *static (nested) classes*.
  - Non-static classes: these are called *inner classes*.
- Both kinds of classes are part of the enclosing (defining) class.
- The enclosing class is also referred to as the *outer* class.
- The differences between static and nested classes are subtle.

# Proper Inner Classes

- Defined at top level of its outer class.
- An inner class instance *depends on an instance of the outer class*.
  - The inner instance can see its outer instance's instance attributes.
  - Implicitly, the inner instance owns its outer instance's reference.
  - Inner classes cannot have class attributes and class methods.
- You may create inner class instances in two kinds of methods.
  - 1 **An instance method or constructor of the outer class.**
    - The new instance depends on the `this` of the method/constructor.
  - 2 **An instance method or constructor of the inner class.**
    - The new instance depends on the instance that the current inner class instance depends on.

# Example: Inner Class

## Java

```
public class Outer {  
    private final int value;  
  
    public void outerMethod( ) {  
        Inner inner = new Inner( );  
    }  
  
    private class Inner {  
        private Inner( ) {  
            System.out.println( value );  
        }  
        ...  
    }  
}
```

# Inner Class: Second Example

## Java

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.Color;

public class InnerClassExample {
    private final JButton button;
    private boolean alert;

    public static void main( String[] args ) {
        final InnerClassExample gui = new InnerClassExample( );
        gui.run( );
    }

    private InnerClassExample( ) {
        button = new JButton( "click me" );
        alert = false;
    }

    ...
}
```

# Inner Class: Second Example (Continued)

## Java

```
public class InnerClassExample {
    private final JButton button;
    private boolean alert;

    ...

    private void run( ) {
        JFrame frame = new JFrame( "Inner Class Example" );
        final JPanel panel = new JPanel( );
        final Listener listener = new Listener( );
        frame.getContentPane( ).add( panel );
        panel.add( button );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize( 300, 100 );
        frame.setVisible( true );
    }

    ...
}
```

# Inner Class: Second Example (Continued)

## Java

```
public class InnerClassExample {
    private final JButton button;
    private boolean alert;

    ...

    private class Listener implements ActionListener {

        private Listener( ) {
            button.addActionListener( this );
        }

        @Override
        public void actionPerformed((ActionEvent event) {
            button.setText( alert ? "Alarm" : "No panic" );
            button.setBackground( alert ? Color.red : Color.green );
            alert = !alert;
        }
    }
}
```



# Several Inner Classes

- You can have zero, one, or more inner classes.
- Very useful for GUI applications.

**Outer class** Owns attributes that represent GUI state.

**Inner class** Listens to the events.

- Has access to the attributes.
- Can modify them when an event occurs.

# Third Inner Class Example

## Java

```
public class EditorGUI {  
    private final ButtonGroup fontStyleGroup;  
    private final ButtonGroup sizeGroup;  
    ...  
  
    public EditorGUI( ) { ... }  
  
    private class FontGroupListener implements ActionListener { ... }  
  
    private class SizeGroupListener implements ActionListener { ... }  
  
    ...  
}
```

# Local Classes

- Java also lets you define classes in methods.
- These classes are called *local* classes.
  - A local class defined in instance method is an inner class.
  - A local class defined in a class method is a static class.
- Local classes may have names or not.
  - With name:** These are called *local (inner) classes*.
  - Without name:** These are called *anonymous classes*.
- Only use them when classes are really short.
  - With long classes, you usually can't see the wood from the trees.

# Example: Local Inner Class

## Java

```
public class Outer {  
    public static void classMethod( ) {  
        private class LocalInner {  
            ...  
        }  
        final LocalInner inner = new LocalInner( );  
        ...  
    }  
}
```

# Static Classes

- A static class is defined at the top level of some other class.
- It has no access to outer class instance methods.
- It has no access to outer class instance attributes.

# Static Classes

## Java

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.Color;

public class StaticDoubleListener {

    public static void main( String[] args ) {
        JFrame frame = new JFrame( "Two Listeners" );
        final JButton firstButton = new JButton( "first" );
        final JButton secondButton = new JButton( "second" );
        final JPanel panel = new JPanel( );
        final Listener first = new Listener( firstButton, secondButton );
        final Listener second = new Listener( secondButton, firstButton );
        frame.getContentPane( ).add( panel );
        panel.add( firstButton );
        panel.add( secondButton );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize( 300, 100 );
        frame.setVisible( true );
    }

    private static class Listener implements ActionListener {
        ...
    }
}
```

## Java

```
private static class Listener implements ActionListener {
    private final JButton button;
    private boolean alert;

    private Listener( final JButton thisButton,
                     final JButton thatButton ) {
        button = thisButton;
        thatButton.addActionListener( this );
    }

    @Override
    public void actionPerformed((ActionEvent event) {
        button.setText( alert ? "Alarm" : "No panic" );
        button.setBackground( alert ? Color.red : Color.green );
        alert = !alert;
    }
}
```

[Outline](#)[Observer Pattern](#)[Windows](#)[Events](#)[Nested Classes](#)[Inner Classes](#)[Local Classes](#)[Static Classes](#)[Anonymous Classes](#)[Nested Interfaces](#)[Question Time](#)[For Thursday](#)[Acknowledgements](#)[References](#)[About this Document](#)

# Anonymous Classes

- An anonymous class is a class without name.
- It extends a single class or implements a single interface.
- It combines class definition & instance creation.
  - It cannot have an explicit constructor.
  - Its body should override all necessary methods.



# Example: Anonymous Class

## Java

```
public class Matrimony {  
    ...  
    private static void unite( ) {  
        final Man john = new Man( ) {  
            @Override public void marry( final Woman wife ) { ... }  
        };  
        final Woman mary = new Woman( ) {  
            @Override public void marry( final Man husband ) { ... }  
        };  
        john.marry( mary );  
    }  
}
```

# Interfaces can be “Nested” Too

## Java

```
public class Matrimony {  
    ...  
  
    private interface Unitable { }  
  
    private interface Woman extends Unitable {  
        public void marry( final Man husband );  
    }  
  
    private interface Man extends Unitable {  
        public void marry( final Woman wife );  
    }  
}
```

Outline

Observer Pattern

Windows

Events

Nested Classes

Nested Interfaces

Question Time

For Thursday

Acknowledgements

References

About this Document

# Questions Anybody?

# For Thursday

- Study the presentation;
- Re-implement the newspaper example from scratch;
- Read [Sierra, and Bates 2004, Chapter 12] if you have the book;
- Study [Bloch 2008, Item 30] if you have the book.

# Acknowledgements

- This lecture corresponds to [Sierra, and Bates 2004].
- [Eric Freeman, and Elisabeth Freeman 2005, Pages 44–78];
- Gamma et al. [2008].
- Some material is based on the Oracle tutorials.

## Bibliography I

- 📖 Bloch, Joshua [2008]. *Effective Java*. Addison–Wesley. ISBN: 978-0-321-35668-0.
- 📖 Freeman, Eric, and Elisabeth Freeman [2005]. *Head First Design Patterns*. O'Reilly. ISBN: 978-0-596-00920-5.
- 📖 Gamma, Erich et al. [2008]. *Design Patterns Elements of Reusable Object-Oriented Software*. 36th Printing. Addison–Wesley. ISBN: 0-201-63361-2.
- 📖 Sierra, Kathy, and Bert Bates [2004]. *Head First Java*. O'Reilly. ISBN: 978-0-596-00712-6.

# About this Document

- This document was created with pdf $\text{\LaTeX}$ atex.
- The  $\text{\LaTeX}$  document class is beamer.