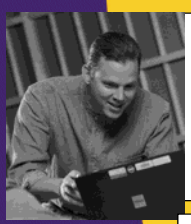
The Google Web Toolkit (GWT): Declarative Layout with UiBinder – Basics

(GWT 2.5 Version)

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/gwt.html

For live Ajax & GWT training, see training courses at http://courses.coreservlets.com/.



Taught by the author of *Core Servlets and JSP*, *More Servlets and JSP*, and this tutorial. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android development, Java 7 or 8 programming, custom mix of topics
  - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, GWT, Hadoop, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- Problems with *only* Java-based layout
- Main idea behind UiBinder approach
- Advantages of UiBinder
- Disadvantages of UiBinder
- Steps for HTML-based UiBinder GUIs
- Steps for Widget-based UiBinder GUIs
- Simplified UI event handling with UiBinder

Overview

Layout Strategies

- **HTML-based layout**
  - Write HTML by hand, designate places for individual controls
  - HTML body contains a lot of content
  - Best when GWT is used for
    - A traditional Web app with some pieces of Ajax-enabled content
    - Complex page where HTML layout does not change
    - You need the HTML content indexed by a search engine
- **Java-based layout**
  - Attaches main layout panel to HTML <body> element
  - Java uses Panels to build up overall layout
    - Similar to way LayoutManagers are used in desktop Java
  - Best when GWT is used to
    - Mimic a desktop application
    - Create an application where HTML layout changes on fly
- **Declarative layout with UiBinder (this tutorial)**

HTML-Based Layout

- **HTML**

```
<body>
Regular HTML <div id="id-1"></div>
Regular HTML <div id="id-2"></div>
Regular HTML <div id="id-3"></div>
Regular HTML
</body>
```

- **Java**

```
public void onModuleLoad() {
  SomeWidget w1 = …;
  RootPanel.get("id-1").add(w1);
  SomeWidget w2 = …;
  RootPanel.get("id-2").add(w2);
  SomeWidget w3 = …;
  RootPanel.get("id-3").add(w3);
}
```

Java-Based Layout

- **HTML**

```
<body>
<!-- Possibly history iframe, but few or no HTML elements -->
</body>
```

- **Java**

```
public void onModuleLoad() {
  SomePanel mainPanel = …;
  mainPanel.setVariousProperties();
  SomeOtherPanel subPanel = …;
  SomeWidget w1 = …;
  subPanel.add(w1);
  …
  mainPanel.add(subPanel);
  …
  RootLayoutPanel.get().add(mainPanel);
}
```


Problems with *Only* Java-Based Layout

- **Hard to format HTML in Java**
  - Think servlets vs. JSP
  - HTML-based layout is usually not dynamic enough
    - If it is, use it!
- **Hard to visualize the layout**
  - Not easy to see final result from looking at Java code
- **Hard to effectively involve a graphic UI Web designer**
  - They usually don't know Java
  - Even if you manage to get the design into Java, it is hard to maintain

UiBinder: Main Idea

- **Use XML file to lay out chunk of content**
  - Can represent HTML or a Widget
- **Make Java class that represents that chunk of content**
  - There are some overly-details steps involved, but Eclipse has shortcuts that automate making most of them
- **Use that Java class in your main application**
  - If class represents HTML, use the GWT DOM API to insert it
  - If class represents a Widget, use normal "add" method to insert it

Advantages of UiBinder

- **Can make complex page layouts using HTML**
  - Or HTML-like XML
    - Analogous to adding JSP to pure-servlet apps
    - More maintainable
- **Graphic Web designers can be involved from initial design through maintenance**
  - Easy to start with regular HTML and gradually "sprinkle in" GWT bindings
- **Separation of concerns**
  - Aesthetics and functionality of UI no longer mashed together
- **Compile-time check of cross references between XML & Java and even within XML itself**

Advantages of UiBinder (Continued)

- **Better browser performance**
  - Browsers are very fast at stuffing long strings into innerHTML of an element
    - Not so much when it comes to executing JavaScript APIs
  - Lesson: use regular HTML whenever possible!
  - Goal of UiBinder: Make the easier choice the right choice
- **Tooling support**
  - Eclipse, NetBeans

Disadvantages of UiBinder

- **Causes proliferation of files**
  - There are two files for *every* component
- **Unlike JSP, UiBinder is only declarative**
  - No rendering mechanism: no loops, conditionals, etc.
  - It's just a declaration of your layout
  - Still need Java to loop through data and generate output
- **Complex page layouts are done with HTML**
  - Wait! What? Didn't I say that was an advantage?
  - Depends on your background and why you chose GWT in the first place

HTML-Based UiBinder GUIs

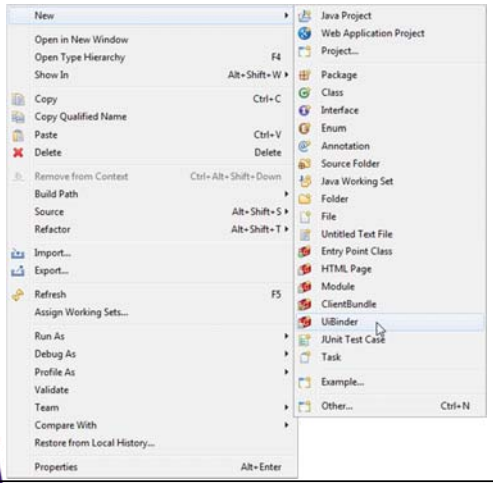
Basic Steps

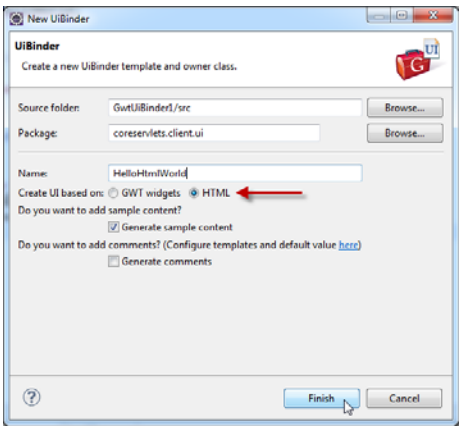
1. **Create XML file**
   - Top-level element is an HTML tag like
   - Put raw HTML inside
   - Use ui:field to mark places for later insertion
2. **Create Java class that represents the XML**
   - Extend UiObject
   - Use UiBinder interface and GWT.create to bind the Java class to the XML representation
   - Mark fields with @UiField (one for each ui:field in XML)
   - Call setElement
   - Insert content into those fields
3. **Use new Java class in EntryPoint class**
   - Use DOM API with getElement and appendChild

- **Note**
  - Eclipse wizard will automate much of this process

Example: HelloHtmlWorld UiBinder

- **Start with Eclipse wizard**
  - Created coreservlets.client.ui package (for convenience of organization)
  - Right-clicked on the ui package and chose New→Other→GWT→UiBinder
  - Chose Create UI based on HTML, clicked Finish
  - After creation, removed some generated content we'll learn later about
  - Creates 2 files: HelloHtmlWorld.java and HelloHtmlWorld.ui.xml





Step 1: XML File – HelloHtmlWorld.ui.xml

```
<!DOCTYPE ui:UiBinder SYSTEM
"http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder">
 <div>
   Hello,  &hearts; 
   <span ui:field="greetSpan"></span>
 </div>
</ui:UiBinder>
```

This is considered the root element of the UiBinder declaration, even though the root element of the XML file is ui:UiBinder.

This is what the Java counterpart needs to treat this declaration as, i.e., div element

This is a bound field. There must be a corresponding member of the same name in HelloHtmlWorld.java . It must have at least package private scope, i.e., cannot be private.

The type of the corresponding field must be a GWT DOM class that is equivalent to owning tag, i.e., . (SpanElement or its superclass)

Regular XML does not define named HTML character entities like   and ♥. GWT provides them through the DOCTYPE above. This is already baked into the GWT compiler, so don't need to have Internet access for this to work.

Yes, this is DTD and XML Schema declaration in the same file! This is completely legal!

Step 2A: Extend UiObject – HelloHtmlWorld.java

```
...
import com.google.gwt.core.client.GWT;
import com.google.gwt.dom.client.*;
import com.google.gwt.uibinder.client.*;
import com.google.gwt.user.client.ui.UIObject;

public class HelloHtmlWorld extends UIObject {
  interface HelloHtmlWorldUiBinder
            extends UiBinder<DivElement, HelloHtmlWorld> {}

  private static HelloHtmlWorldUiBinder uiBinder =
                              GWT.create(HelloHtmlWorldUiBinder.class);

  @UiField SpanElement greetSpan;

  public HelloHtmlWorld(String greeting) {
    setElement(uiBinder.createAndBindUi(this));
    greetSpan.setInnerText(greeting);
  }
}
```

Only if your root UiBinder element is HTML, i.e., it's an HTML-based UiBinder. In this case, the root UiBinder element is div.

In next section, we will use Widget-based UiBinders, so will extend Composite instead.

Step 2B: Use UiBinder & GWT.create to Bind Java to XML

Declare marker interface that will tell GWT which two types to attempt binding.

Must be GWT type equivalent to the type of root UiBinder element. (DivElement or Element type will work here).

Owner type whose @UiFields need to be bound, i.e., type of this class.

```
...
import com.google.gwt.user.client.ui.UIObject;

public class HelloHtmlWorld extends UIObject {
  interface HelloHtmlWorldUiBinder
            extends UiBinder<DivElement, HelloHtmlWorld> {}

  private static HelloHtmlWorldUiBinder uiBinder =
                              GWT.create(HelloHtmlWorldUiBinder.class);

  @UiField SpanElement greetSpan;

  public HelloHtmlWorld(String greeting) {
    setElement(uiBinder.createAndBindUi(this));
    greetSpan.setInnerText(greeting);
  }
}
```

GWT creates an instance of a factory that can generate the UI and glue it to the owning class, i.e., this class, associating all the fields with @UiField with its declared counterpart.

Steps 2C – 2E

```
...

public class HelloHtmlWorld extends UIObject {
 interface HelloHtmlWorldUiBinder
            extends UiBinder<DivElement, HelloHtmlWorld> {
  }

  private static HelloHtmlWorldUiBinder uiBinder =
                              GWT.create(HelloHtmlWorldUiBinder.class);
  @UiField SpanElement greetSpan;

  public HelloHtmlWorld(String greeting) {
    setElement(uiBinder.createAndBindUi(this));
    greetSpan.setInnerText(greeting);
  }
}
```

Step 2C: Mark fields with @UiField (on for each ui:field in the XML). Var name must match ui:field name in ui.xml file.

Creates the root UiBinder type defined in the interface, i.e., DivElement, passing it an instance of an owner class, i.e., HelloHtmlWorld.

Step 2D: call setElement. Before *anything* can be done with a UIObject instance, setElement method must be called. Tells GWT what HTML element to treat this object as.

Step 2E: insert content. Since this represents an HTML Element (SpanElement in this case), not a Widget, we use the lower-level Element methods like setInnerText.

Final Result: HelloHtmlWorld.java

```
...
import com.google.gwt.core.client.GWT;
import com.google.gwt.dom.client.*;
import com.google.gwt.uibinder.client.*;
import com.google.gwt.user.client.ui.UIObject;

public class HelloHtmlWorld extends UIObject {
  interface HelloHtmlWorldUiBinder
            extends UiBinder<DivElement, HelloHtmlWorld> {
  }

  private static HelloHtmlWorldUiBinder uiBinder =
                              GWT.create(HelloHtmlWorldUiBinder.class);

  @UiField SpanElement greetSpan;

  public HelloHtmlWorld(String greeting) {
    setElement(uiBinder.createAndBindUi(this));
    greetSpan.setInnerText(greeting);
  }
}
```

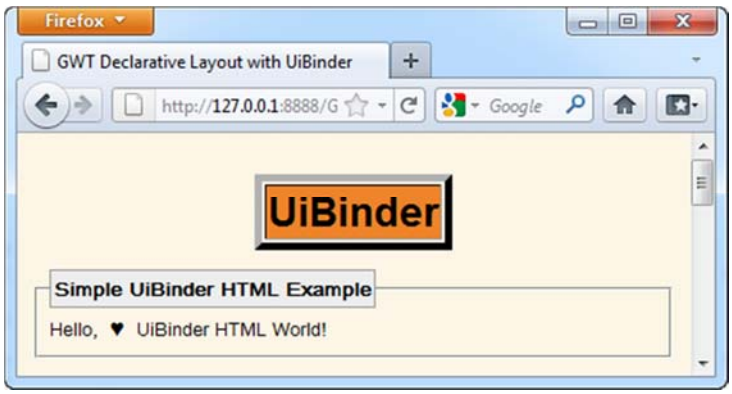
Step 3: Use New Java Class in EntryPoint Class

```
...
public class GwtUiBinder1 implements EntryPoint {
  public void onModuleLoad() {
    HelloHtmlWorld helloHtmlWorld =
      new HelloHtmlWorld("UiBinder HTML World!");
    Element attachTo =
      RootPanel.get("uibinder-html").getElement();
    attachTo.appendChild(helloHtmlWorld.getElement());

    ...
  }
}
```

Since the HelloHtmlWorld is a UIObject and not a Widget, we need to use the lower-level DOM API to append it to the browser DOM.

Example: Result

```
...
<fieldset>
<legend>Simple UiBinder HTML Example</legend>
<div id="uibinder-html"></div>
</fieldset>
<br/>
...
```



Widget-Based UiBinder GUIs

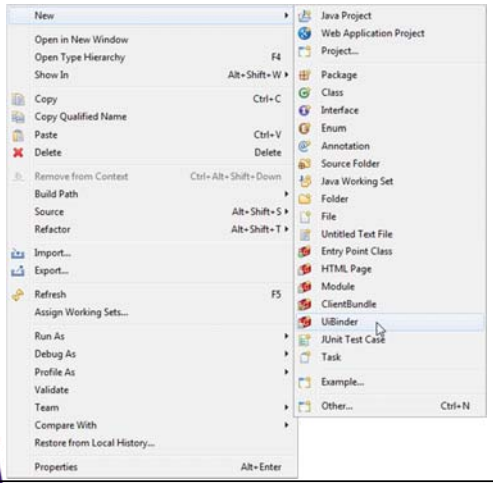
Basic Steps (Main Changes from HTML-Based in Red)

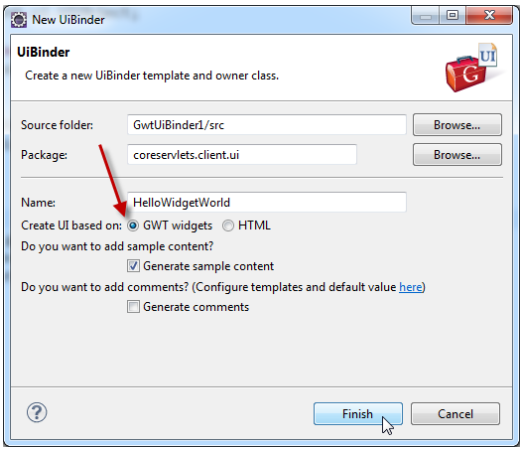
1. **Create XML file**
   - Top-level element represents a Widget: <g:someGwtWidget>
   - Put mixture of raw HTML and <g:otherGwtWidgets> inside
   - Use ui:field to mark places for later insertion
2. **Create Java class that represents the XML**
   - Extend Composite
   - Use UiBinder interface and GWT.create to bind the Java class to the XML representation
   - Mark fields with @UiField (one for each ui:field in XML)
   - Call initWidget
   - Set properties of those fields
3. **Use new Java class in EntryPoint class**
   - Use normal Widget methods (e.g., RootPanel.get(…).add(yourWidget))

- **Note**
  - Eclipse wizard will automate much of this process

Example: HelloWidgetWorld UiBinder

- **Start with Eclipse wizard**
  - Created coreservlets.client.ui package (for convenience of organization)
  - Right-clicked on the ui package and chose New-UiBinder
  - Chose Create UI based on GWT widgets, clicked Finish
  - After creation, removed some generated content we'll learn later about
  - Creates 2 files: HelloWidgetWorld.java and HelloWidgetWorld.ui.xml





Step 1: XML File

```
<!DOCTYPE ui:UiBinder SYSTEM
"http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
xmlns:g="urn:import:com.google.gwt.user.client.ui">
  <g:HTMLPanel>
    <span style="font-style: italic;">Hello,</span>
    <g:HTML ui:field="label" />
  </g:HTMLPanel>
</ui:UiBinder>
```

Imports everything in the com.google.gwt.user.client.ui package and gives it an XML namespace prefix 'g'.

Now, any class in that package can be referenced here with the prefix 'g'.

This is considered the root widget of this UiBinder declaration, even though the root element of the XML file is ui:UiBinder.

This is what the Java counterpart needs to treat this declaration as, i.e., HTMLPanel

This is a bound field. There must be a corresponding member of the same name in HelloWidgetWorld.java . It must have at least package private scope, i.e., can not be private.

The type of the corresponding field must be a GWT widget type that is either HTML or its superclass.

Step 2A: Extend Composite – HelloWidgetWorld.java

```
...
import com.google.gwt.core.client.GWT;
import com.google.gwt.uibinder.client.*;
import com.google.gwt.user.client.ui.*;

public class HelloWidgetWorld extends Composite {
  interface HelloWidgetWorldUiBinder
                    extends UiBinder<Widget, HelloWidgetWorld> {}

  private static HelloWidgetWorldUiBinder uiBinder =
                    GWT.create(HelloWidgetWorldUiBinder.class);

  @UiField HTML label;

  public HelloWidgetWorld(String greeting) {
    initWidget(uiBinder.createAndBindUi(this));
    label.setText(greeting);
  }
}
```

Only if your root UiBinder element is a GWT Widget, i.e., it's a GWT Widget-based UiBinder. In this case, the root UiBinder widget is HTMLPanel, whose superclass is Widget, but we could have used HTMLPanel here just the same.

Yes, this IS a custom widget!

Step 2B: Use UiBinder & GWT.create to Bind Java to XML

```
...
public class HelloWidgetWorld extends Composite {
  interface HelloWidgetWorldUiBinder
                    extends UiBinder<Widget, HelloWidgetWorld> {}

  private static HelloWidgetWorldUiBinder uiBinder =
                    GWT.create(HelloWidgetWorldUiBinder.class);

  @UiField HTML label;

  public HelloWidgetWorld(String greeting) {
    initWidget(uiBinder.createAndBindUi(this));
    label.setText(greeting);
  }
}
```

Declare marker interface that will tell GWT which two types to attempt binding.

Must be GWT type equivalent to the type of root UiBinder element. (HTMLPanel or Widget type will work here).

Owner type whose @UiFields need to be bound, i.e., type of this class.

GWT creates an instance of a factory that can generate the UI and glue it to the owning class, i.e., this class, associating all the fields with @UiField with its declared counterpart.

Steps 2C – 2E

```
...

public class HelloWidgetWorld extends Composite {
  interface HelloWidgetWorldUiBinder
                    extends UiBinder<Widget, HelloWidgetWorld> {}

  private static HelloWidgetWorldUiBinder uiBinder =
                    GWT.create(HelloWidgetWorldUiBinder.class);

  @UiField HTML label;

  public HelloWidgetWorld(String greeting) {
    initWidget(uiBinder.createAndBindUi(this));
    label.setText(greeting);
  }
}
```

Step 2C: mark fields with @UiField. Must match ui:field name in ui.xml file.

Creates the root UiBinder type defined in the interface, i.e., Widget, passing it an instance of an owner class, i.e., HelloWidgetWorld.

Step 2D: call initWidget. Before *anything* can be done with a Composite instance, initWidget method must be called. Tells GWT what GWT Widget to treat this object as.
(We showed this before when creating custom widgets).

Step 2E: Set properties of fields. This is a normal Widget (of type HTML in this case).

Final Result: HelloWidgetWorld.java

```
...
import com.google.gwt.core.client.GWT;
import com.google.gwt.uibinder.client.*;
import com.google.gwt.user.client.ui.*;

public class HelloWidgetWorld extends Composite {
  interface HelloWidgetWorldUiBinder
                    extends UiBinder<Widget, HelloWidgetWorld> {}

  private static HelloWidgetWorldUiBinder uiBinder =
                    GWT.create(HelloWidgetWorldUiBinder.class);

  @UiField HTML label;

  public HelloWidgetWorld(String greeting) {
    initWidget(uiBinder.createAndBindUi(this));
    label.setText(greeting);
  }
}
```

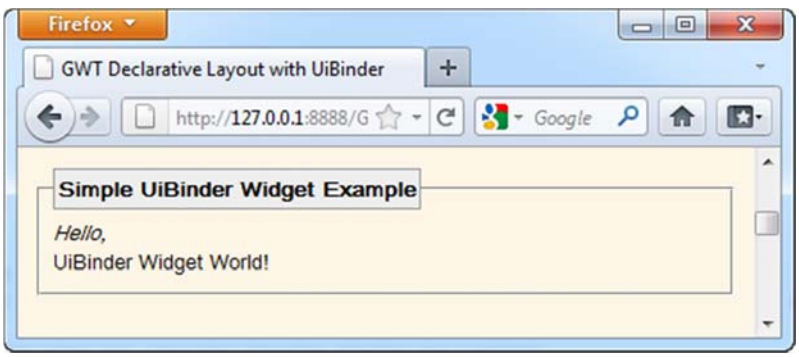
Step 3: Use New Java Class in EntryPoint Class

```
public class GwtUiBinder1 implements EntryPoint {

public void onModuleLoad() {
    HelloHtmlWorld helloHtmlWorld =
              new HelloHtmlWorld("UiBinder HTML World!");
    Element attachTo = RootPanel.get("uibinder-html").getElement();
    attachTo.appendChild(helloHtmlWorld.getElement());

    HelloWidgetWorld helloWidgetWorld =
               new HelloWidgetWorld("UiBinder Widget World!");
    RootPanel.get("uibinder-widget").add(helloWidgetWorld);
    ...
  }
}
```

Since the HelloWidgetWorld is a regular GWT widget, we can directly add it to RootPanel without resorting to GWT DOM API.

Example: Result

```
...
<fieldset>
<legend>Simple UiBinder Widget Example</legend>
<div id="uibinder-widget"></div>
</fieldset>
<br/>
...
```



Simplified UI Event Handling

Summary

- **UI event handling has a lot of "boilerplate" code**
  - Create a class that implements Some*Handler* interface
    - Can be done as an anonymous, inner, or standalone
    - Implement the on*Blah* method in that interface
  - Call add*SomeHandler* to the widget
- **In UiBinder, use shortcut:**

  @UiHandler("fieldName") void method(*Blah*Event e) {}
  - "fieldName" – name of declared member field in this class
  - *Blah*Event is any event type, e.g., ClickEvent, BlurEvent, etc.
  - Method name can be anything you want (*but can't be private*)
- **Disadvantages**
  - GWT-created handlers are not sharable with other components
    - You can still refactor and share the functionality
  - Can only be used on widgets, not DOM elements

Random Number and Linked ListBox Example Review

```
...
<fieldset>
<legend>Client-Side Data</legend>
<table border="1">
 <tr><th>User Control</th>
     <th>Result</th></tr>
 <tr><td id="randomNumberButton"></td>
     <td id="randomNumberResult"></td></tr>
 <tr><td>State: <span id="stateListHolder"></span></td>
     <td>City:  <span id="cityListHolder"></span></td>
 </tr>
</table><br/>
</fieldset>
...
```


Random Number and Linked ListBox Example Review (Cont.)

```
public class GwtApp1 implements EntryPoint {
  public void onModuleLoad() {  // Main entry point
    buttonSetup();
    listSetup();
  }

  private void buttonSetup() {
    Button randomNumberButton = new Button("Show Random Number");
    HTML randomNumberResult = new HTML("<i>Num will go here</i>");
    randomNumberButton.addClickHandler
      (new RanNumHandler(randomNumberResult));
    RootPanel.get("randomNumberButton").add(randomNumberButton);
    RootPanel.get("randomNumberResult").add(randomNumberResult);
  }
  ...
}
```

Random Number and Linked ListBox Example Review (Cont.)

```
private class RanNumHandler implements ClickHandler {
   private HTML resultRegion;

   public RanNumHandler(HTML resultRegion) {
     this.resultRegion = resultRegion;
   }

   public void onClick(ClickEvent event) {
     resultRegion.setText("Number: " + Math.random()*10);
   }
 }
```

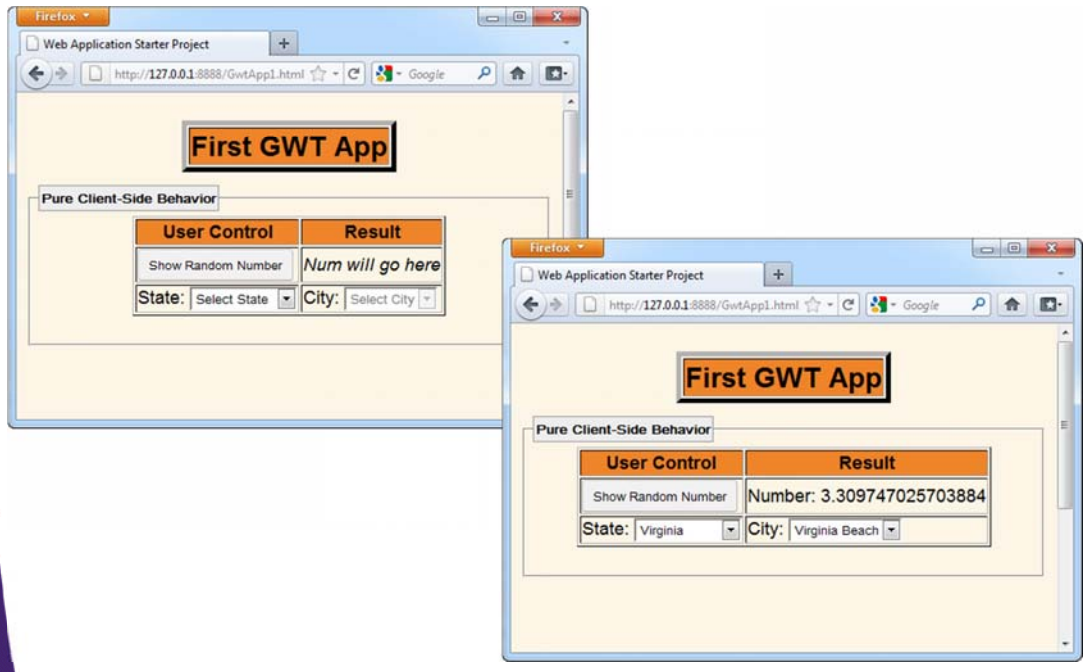

Random Number and Linked ListBox Example Review (cont)

```
private void listSetup() {
  ListBox stateList = new ListBox();
  populateStateList(stateList);
  stateList.setVisibleItemCount(1);
  ListBox cityList = new ListBox();
  cityList.addItem("Select City");
  cityList.setVisibleItemCount(1);
  cityList.setEnabled(false);
  stateList.addChangeHandler(new StateHandler(stateList, cityList));
  RootPanel.get("stateListHolder").add(stateList);
  RootPanel.get("cityListHolder").add(cityList);
}

private void populateStateList(ListBox stateList) {
  stateList.addItem("Select State");
  StateInfo[] nearbyStates =
    StateInfo.getNearbyStates();
  for(StateInfo state: nearbyStates) {
    stateList.addItem(state.getStateName());
  }
}
```

Random Number and Linked ListBox Example Review (cont)

```
private class StateHandler implements ChangeHandler {
    private ListBox stateList, cityList;

    public StateHandler(ListBox stateList, ListBox cityList) {
      this.stateList = stateList;
      this.cityList = cityList;
    }

    public void onChange(ChangeEvent event) {
      int index = stateList.getSelectedIndex();
      String state = stateList.getItemText(index);
      StateInfo[] nearbyStates =
        StateInfo.getNearbyStates();
      String[] cities =
        StateInfo.findCities(nearbyStates, state);
      cityList.clear();
      for(String city: cities) {
        cityList.addItem(city);
      }
      cityList.setEnabled(true);
    }
  }
```


Random Number and Linked ListBox Example Review: Result



Example: UI Event Handling *Without* UiBinder shortcut: HandleEvents1.ui.xml

```
<!DOCTYPE ui:UiBinder SYSTEM
"http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
      xmlns:g="urn:import:com.google.gwt.user.client.ui">
<g:HTMLPanel>
<table border="1">
  <tr><th>User Control</th><th>Result</th></tr>
  <tr>
     <td><g:Button ui:field="numberButton">Show Random Number
         </g:Button></td>
     <td><g:HTML ui:field="numberResult">Num will go here
         </g:HTML></td></tr>
  <tr><td>State: <g:ListBox ui:field="stateList"/></td>
      <td>City: <g:ListBox ui:field="cityList"/></td></tr>
</table><br/>
</g:HTMLPanel>
</ui:UiBinder>
```


Example: UI Event Handling *Without* UiBinder shortcut: HandleEvents1.java

```
...
public class HandleEvents1 extends Composite {
  private static HandleEvents1UiBinder uiBinder = GWT
      .create(HandleEvents1UiBinder.class);
  interface HandleEvents1UiBinder
              extends UiBinder<Widget, HandleEvents1> {
  }

  @UiField Button numberButton;
  @UiField HTML numberResult;
  @UiField ListBox stateList;
  @UiField ListBox cityList;

  public HandleEvents1() {
    initWidget(uiBinder.createAndBindUi(this));
    numberButton.addClickHandler(new RanNumHandler());
    listSetup();
  }
```

Example: UI Event Handling *Without* UiBinder shortcut: HandleEvents1.java (continued)

```
...
  private class RanNumHandler implements ClickHandler {
    public void onClick(ClickEvent event) {
      numberResult.setText("Number: " + Math.random() * 10);
    }
  }

  private void listSetup() {
    populateStateList();
    stateList.setVisibleItemCount(1);
    cityList.addItem("Select City");
    cityList.setVisibleItemCount(1);
    cityList.setEnabled(false);
    stateList.addChangeHandler(new StateHandler());
  }

  private void populateStateList() {
    stateList.addItem("Select State");
    StateInfo[] nearbyStates = StateInfo.getNearbyStates();
    for (StateInfo state : nearbyStates) {
      stateList.addItem(state.getStateName());
    }
  }
  ...
```


Example: UI Event Handling *Without* UiBinder shortcut: HandleEvents1.java (Continued)

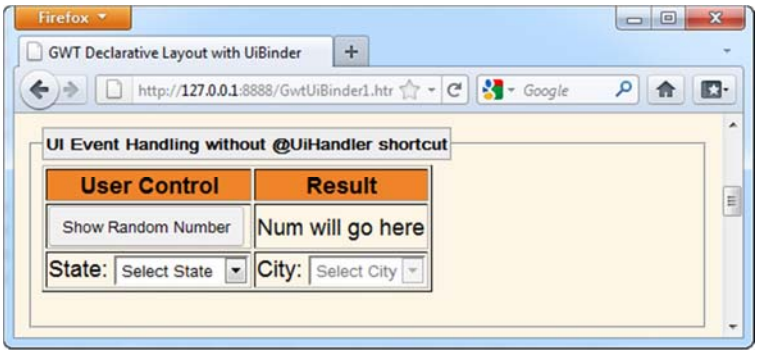
```
...
  private class StateHandler implements ChangeHandler {
    public void onChange(ChangeEvent event) {
      int index = stateList.getSelectedIndex();
      String state = stateList.getItemText(index);
      StateInfo[] nearbyStates = StateInfo.getNearbyStates();
      String[] cities = StateInfo.findCities(nearbyStates, state);
      cityList.clear();
      for (String city : cities) {
        cityList.addItem(city);
      }
      cityList.setEnabled(true);
    }
  }
}
```

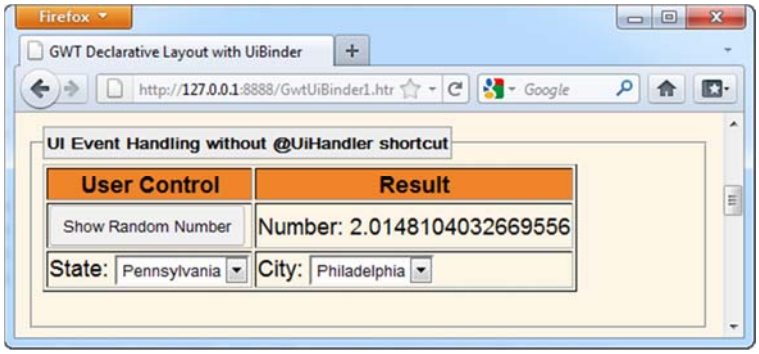
Example: UI Event Handling *Without* UiBinder shortcut: GwtUiBinder1.html & EntryPoint Class

```
...
<fieldset>
<legend>UI Event Handling without @UiHandler shortcut</legend>
<div id="uibinder-events-without"></div>
</fieldset>
<p/>
```

```
...
public class GwtUiBinder1 implements EntryPoint {

  public void onModuleLoad() {
    ...

    HandleEvents1 events1 = new HandleEvents1();
    RootPanel.get("uibinder-events-without").add(events1);

    ...
  }
}
```


Example: UI Event Handling *Without* UiBinder shortcut: Result





Example: UI Event Handling *With* UiBinder shortcut: HandleEvents2.ui.xml

```
<!DOCTYPE ui:UiBinder SYSTEM "http://dl.google.com/gwt/DTD/xhtml.ent">
<ui:UiBinder xmlns:ui="urn:ui:com.google.gwt.uibinder"
             xmlns:g="urn:import:com.google.gwt.user.client.ui">
<g:HTMLPanel>
<table border="1">
  <tr><th>User Control</th><th>Result</th></tr>
  <tr>
     <td><g:Button ui:field="numberButton">Show Random Number
         </g:Button></td>
     <td><g:HTML ui:field="numberResult">Num will go here
         </g:HTML></td></tr>
  <tr><td>State: <g:ListBox ui:field="stateList"/></td>
      <td>City: <g:ListBox ui:field="cityList"/></td></tr>
</table><br/>
</g:HTMLPanel>
</ui:UiBinder>
```

No Changes. Same as HandleEvents1.ui.xml.

Example: UI Event Handling *With* UiBinder shortcut: HandleEvents2.java

```
...
public class HandleEvents2 extends Composite {
  private static HandleEvents2UiBinder uiBinder =
                  GWT.create(HandleEvents2UiBinder.class);
  interface HandleEvents2UiBinder
                  extends UiBinder<Widget, HandleEvents2> {}

  @UiField Button numberButton;
  @UiField HTML numberResult;
  @UiField ListBox stateList;
  @UiField ListBox cityList;

  public HandleEvents2() {
    initWidget(uiBinder.createAndBindUi(this));
    listSetup();
  }
```

Example: UI Event Handling *With* UiBinder shortcut: HandleEvents2.java (continued)

```
...
  @UiField Button numberButton;
  @UiField HTML numberResult;
  @UiField ListBox stateList;
  @UiField ListBox cityList;
  ...
  @UiHandler("numberButton")
  void displayRandomNumber(ClickEvent event) {
    numberResult.setText("Number: " + Math.random() * 10);
  }
```

GWT matches field name with the value of @UiHandler annotation.

Must be either public or package private (i.e., default) scope. Can't be a private method.

In this example, it is void, but any return type is valid.

Based on the event type, this handler will respond to an onclick event.

Example: UI Event Handling *With* UiBinder shortcut: HandleEvents2.java (continued)

```
  private void listSetup() {
    populateStateList();
    stateList.setVisibleItemCount(1);
    cityList.addItem("Select City");
    cityList.setVisibleItemCount(1);
    cityList.setEnabled(false);
  }
  ...
  @UiHandler("stateList")
  void handleStateChange(ChangeEvent event) {
    int index = stateList.getSelectedIndex();
    String state = stateList.getItemText(index);
    StateInfo[] nearbyStates = StateInfo.getNearbyStates();
    String[] cities = StateInfo.findCities(nearbyStates, state);
    cityList.clear();
    for (String city : cities) {
      cityList.addItem(city);
    }
    cityList.setEnabled(true);
  }
}
```

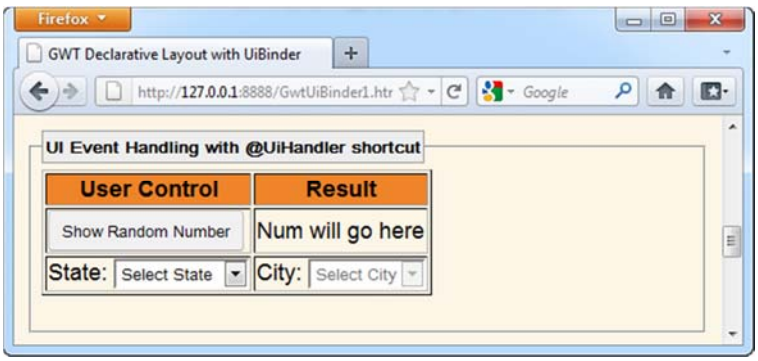
The contents of the method are the same as before, but now no boilerplate code!

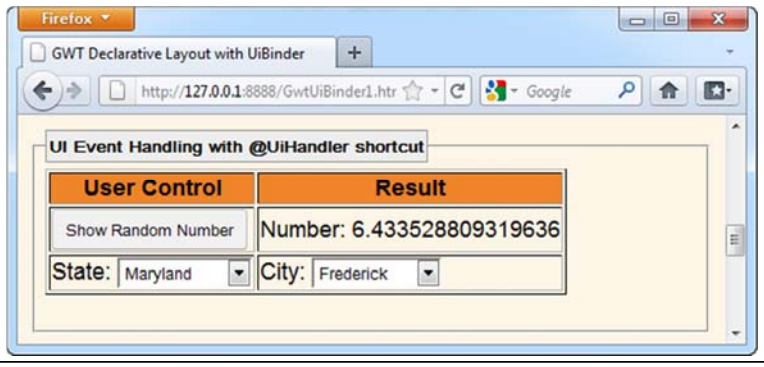
Example: UI Event Handling *Without* UiBinder shortcut: GwtUiBinder1.html & EntryPoint Class

```
...
<fieldset>
<legend>UI Event Handling with @UiHandler shortcut</legend>
<div id="uibinder-events-with"></div>
</fieldset>
<p/>
```

```
...
public class GwtUiBinder1 implements EntryPoint {

  public void onModuleLoad() {
    ...

    HandleEvents2 events2 = new HandleEvents2();
    RootPanel.get("uibinder-events-with").add(events2);
    ...
  }
}
```


Example: UI Event Handling *With* UiBinder shortcut: Result





Wrap-Up

Summary: HTML-Based (Eclipse Wizard Builds #1 & #2)

1. **Create XML file**
   - Top-level element is an HTML tag like
   - Put raw HTML inside
   - Use ui:field to mark places for later insertion
2. **Create Java class that represents the XML**
   - Extend UiObject
   - Use UiBinder interface and GWT.create to bind the Java class to the XML representation
   - Mark fields with @UiField (one for each ui:field in XML)
   - Call setElement
   - Insert content into those fields
3. **Use new Java class in EntryPoint class**
   - Use DOM API with getElement and appendChild

Summary: Widget-Based (Eclipse Wizard Builds #1 & #2)

1. **Create XML file**
   - Top-level element represents a Widget: <g:someGwtWidget>
   - Put mixture of raw HTML and <g:otherGwtWidgets> inside
   - Use ui:field to mark places for later insertion
2. **Create Java class that represents the XML**
   - Extend Composite
   - Use UiBinder interface and GWT.create to bind the Java class to the XML representation
   - Mark fields with @UiField (one for each ui:field in XML)
   - Call initWidget
   - Set properties of those fields
3. **Use new Java class in EntryPoint class**
   - Use normal Widget methods
     - E.g., RootPanel.get(…).add(yourWidget)

Questions?