

Enjoying Web Development with Wicket

By

Kent Ka lok Tong

Copyright © 2007

TipTec Development

Publisher: TipTec Development

Author's email: freemant2000@yahoo.com

Book website: <http://www.agileskills2.org>

Notice: All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

ISBN: 978-99937-929-0-1

Edition: First edition Nov. 2007

Foreword

How to create AJAX web-based application easily?

If you'd like to create AJAX web-based applications easily, then this book is for you. More importantly, it shows you how to do that with joy and feel good about your own work! You don't need to know servlet or JSP while your productivity will be much higher than using servlet or JSP directly. This is possible because we're going to use a library called Wicket that makes complicated stuff simple and elegant.

How does it do that? First, it allows the web designer to work on the static contents and design of a page while allowing the developer to work on the dynamic contents of that page without stepping on each other's toes; Second, it allows developers to work with high level concepts such as objects and properties instead of HTTP URLs, query parameters or HTML string values; Third, it comes with powerful components such as calendar, tree and data grid and it allows you to create your own components for reuse in your own project.

However, don't take our word for it! This book will quickly walk you through real world use cases to show you how to use Wicket and leave it up to you to judge.

How this book can help you learn Wicket?

- It has a tutorial style that walks you through in a step-by-step manner.
- It is concise. There is no lengthy, abstract description.
- Many diagrams are used to show the flow of processing and high level concepts so that you get a whole picture of what's happening.
- Free sample chapters are available on <http://www.agileskills2.org>. You can judge it yourself.

Unique contents in this book

This book covers the following topics not found in other books on Wicket:

- How to create robust, scalable and maintainable enterprise applications with Wicket.
- How to do test-driven development (TDD) with Wicket.
- How to use JPA, Hibernate and Spring with Wicket.

Target audience and prerequisites

This book is suitable for those learning how to develop web-based applications and those who are experienced in servlet, JSP, Struts and would like to see if Wicket can make their jobs easier.

In order to understand what's in the book, you need to know Java, HTML and some simple SQL. However, you do NOT need to know servlet, JSP, Tomcat, Spring, JPA or Hibernate.

Acknowledgments

I'd like to thank:

- The Wicket team for creating Wicket.
- Helena Lei for proofreading this book.
- Eugenia Chan Peng U for doing book cover and layout design.

Table of Contents

Foreword.....	3
How to create AJAX web-based application easily?.....	3
How this book can help you learn Wicket?.....	3
Unique contents in this book.....	3
Target audience and prerequisites.....	4
Acknowledgments.....	4
Chapter 1 Getting Started with Wicket.....	11
What's in this chapter?.....	12
Developing a Hello World application with Wicket.....	12
Installing Eclipse.....	12
Installing Tomcat.....	12
Installing Wicket.....	14
Creating a Hello Word application.....	14
Generating dynamic content.....	23
Common errors in Wicket applications.....	25
More rigorous version of the template.....	27
Simpler version of the template.....	27
Page objects are serializable.....	28
Debugging a Wicket application.....	29
Summary.....	32
Chapter 2 Using Forms.....	35
What's in this chapter?.....	36
Developing a stock quote application.....	36
Mismatch in component hierarchy.....	42
Using a combo box.....	42
Inputting a date.....	46
Displaying feedback messages.....	49
Marking input as required.....	51
Using the DatePicker.....	55
Summary.....	57
Chapter 3 Validating Input.....	59
What's in this chapter?.....	60
Postage calculator.....	60
Using an object to represent the request.....	62
Making sure the page is serializable.....	67
What if the input is invalid?.....	69

Null input and validators.....	73
Validating the patron code.....	75
Displaying the error messages in red.....	77
Displaying invalid fields in red.....	78
Creating a feedback label component.....	80
Validating a combination of multiple input values.....	81
Pattern validator.....	84
Summary.....	84
Chapter 4 Creating an e-Shop.....	87
What's in this chapter?.....	88
Creating an e-shop.....	88
Listing the products.....	88
Using a model for the Labels.....	93
Showing the product details.....	94
Implementing a shopping cart.....	97
How Tomcat and the browser maintain the session.....	103
The checkout function.....	106
Implementing the login function.....	108
Implementing the checkout function.....	113
Protecting a bunch of pages.....	118
Implementing logout.....	120
Summary.....	121
Chapter 5 Building Interactive Pages with AJAX.....	123
What's in this chapter?.....	124
A sample AJAX application.....	124
Refreshing the question only.....	126
Refreshing the answer itself.....	129
Giving rating to a question.....	131
A common mistake with models.....	135
Making the form reusable.....	137
Using a modal window to get the rating.....	140
Having multiple questions.....	142
Falling back if Javascript is disabled.....	144
Summary.....	146
Chapter 6 Supporting Other Languages.....	147
What's in this chapter.....	148
A sample application.....	148
Supporting Chinese.....	148
An easier way to insert a localized message.....	154

Internationalize the page content.....	156
Letting the user change the locale.....	158
Localizing the full stop.....	166
Displaying a logo.....	168
Localizing the logo.....	170
Creating Image components automatically.....	172
Creating a license page.....	173
Creating PageLink components automatically.....	177
Observing the output encoding.....	178
Eliminating the Change button.....	178
Summary.....	180
Chapter 7 Using the DataTable Component.....	183
What's in this chapter?.....	184
Creating a phone book.....	184
Listing the entries in alternating colors.....	186
Storing the styles in a file.....	188
Displaying the entries in pages.....	189
Sorting the entries.....	193
Setting the styles.....	195
Making the first name a link.....	196
Adding a delete button.....	198
Moving the page links to the bottom.....	199
Customizing the message in the NavigationToolbar.....	201
Summary.....	202
Chapter 8 Handling File Downloads and Uploads.....	203
What's in this chapter?.....	204
Downloading a photo.....	204
Reading the bytes from an arbitrary source.....	208
Reading the bytes from a file.....	209
Displaying a photo.....	210
Allowing users to bookmark a page.....	211
Stateless vs stateful pages.....	215
setResponsePage() treating pages as stateful?.....	216
Making the View link bookmarkable.....	217
Using nice URL.....	219
Uploading a photo.....	221
<wicket:link> for bookmarkable pages.....	225
Summary.....	225
Chapter 9 Providing a Common Layout.....	227

What's in this chapter?.....	228
Providing a common layout.....	228
Using components in the abstract part.....	230
Turning the menu into a component.....	233
Using the Border component.....	234
Two varying parts?.....	237
Summary.....	242
Chapter 10 Using Javascript.....	243
What's in this chapter?.....	244
Are you sure to delete it?.....	244
Reusing the confirm button.....	246
Generating the call to Javascript at runtime.....	248
Using a namespace for the Javascript.....	250
Putting the Javascript into a file.....	251
Summary.....	254
Chapter 11 Unit Testing Wicket Pages.....	255
What's in this chapter?.....	256
Developing a calculator.....	256
Creating the Home page.....	257
Using setUp().....	265
Providing a list of operators.....	266
Implementing minus.....	267
Unit testing the History page.....	268
Serialization error.....	272
Implementing the Clear link.....	274
Creating the default CalculationSource.....	277
Logging each calculation.....	278
Refactoring.....	281
Creating the default CalculationSink.....	282
Running all the tests.....	283
Implementing validation.....	284
Integration testing.....	286
Testing AJAX functions.....	290
Summary.....	296
Chapter 12 Developing Robust, Scalable & Maintainable 3-tier Applications.....	299
What's in this chapter?.....	300
Developing a banking application.....	300
Setting up PostgreSQL.....	300

Hard coding some bank accounts.....	308
Transferring some money.....	309
Using a transaction.....	312
Connection pooling.....	315
Concurrency issues.....	319
Business transaction.....	337
Dividing the application into layers.....	348
Reducing the size of the session.....	358
Summary.....	361
Chapter 13 Using Spring in Wicket.....	363
What's in this chapter?.....	364
Examining the gluing code.....	364
Using Spring to manage dependencies.....	365
Using the class of the field to look up the bean.....	371
Will a Spring bean be serialized?.....	372
Using Spring to simplify transaction handling.....	373
Setting the default transaction isolation level.....	381
Unit testing a page that uses Spring beans.....	382
Stateful Spring beans.....	384
Summary.....	384
Chapter 14 Using JPA & Hibernate in Wicket.....	387
What's in this chapter?.....	388
Setting up Hibernate.....	388
Using JPA to access the database.....	388
Power of layering.....	395
Summary.....	395
Chapter 15 Deploying a Wicket Application.....	397
What's in this chapter?.....	398
Development mode.....	398
Distributing your application.....	400
Summary.....	401
References.....	403
Alphabetical Index.....	404

Chapter 1

Getting Started with Wicket

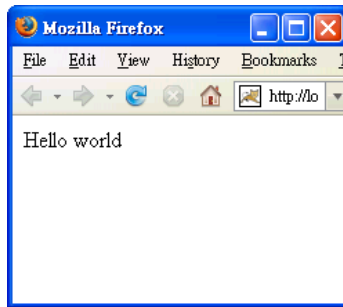


What's in this chapter?

In this chapter you'll learn how to set up a development environment and develop a Hello World application with Wicket.

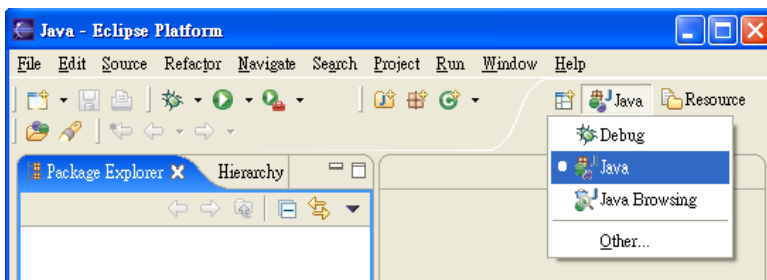
Developing a Hello World application with Wicket

Suppose that you'd like to develop an application like this:



Installing Eclipse

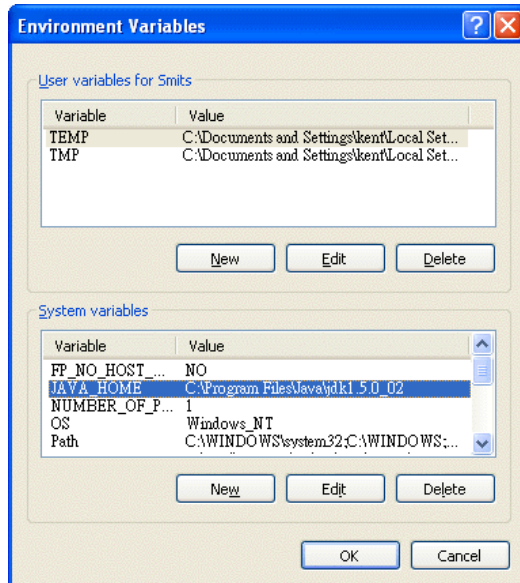
First, you need to make sure you have Eclipse installed. If not, go to <http://www.eclipse.org> to download the Eclipse platform (e.g., `eclipse-platform-3.1-win32.zip`) and the Eclipse Java Development Tool (`eclipse-JDT-3.1.zip`). Unzip both into `c:\eclipse`. Then, create a shortcut to run "`c:\eclipse\eclipse -data c:\workspace`". This way, it will store your projects under the `c:\workspace` folder. To see if it's working, run it and then you should be able to switch to the Java perspective:



Installing Tomcat

Next, you need to install Tomcat. Go to <http://tomcat.apache.org> to download a binary package of Tomcat. Download the zip version instead of the Windows exe version. Suppose that it is `apache-tomcat-6.0.13.zip`. Unzip it into a folder, say `c:\tomcat`. Note that Tomcat 6.x works with JDK 5 or above.

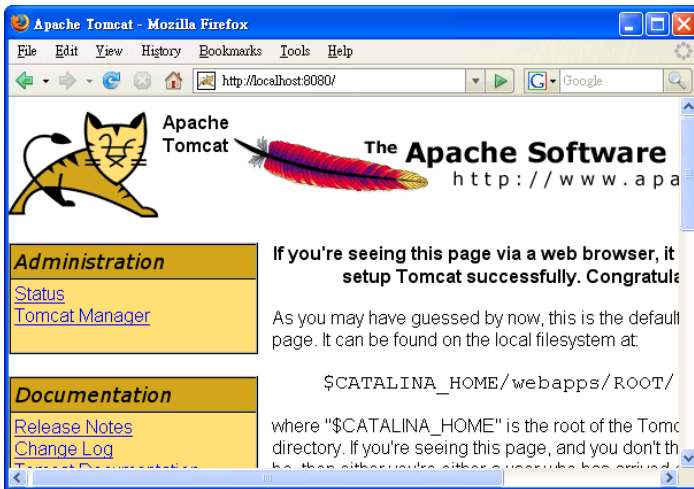
Before you can run it, make sure the environment variable `JAVA_HOME` is defined to point to your JDK folder (e.g., `C:\Program Files\Java\jdk1.5.0_02`):



If you don't have it, define it now. Now, open a command prompt, change to `c:\tomcat\bin` and then run `startup.bat`. If it is working, you should see:

```
Tomcat
Jun 19, 2007 12:18:42 PM org.apache.catalina.core.AprLifecycleListener init
INFO: The Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: C:\Program Files\Java\jdk1.5.0_02\bin;.;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\system32;C:\WINDOWS\system32\Xbin;C:\WINDOWS\System32\Wbin;C:\PROGRAM FILES\BORLAND\DELPHI6\BIN;C:\PROGRAM FILES\BORLAND\DELPHI6\PROJECTS\BPL;C:\PROGRAM FILES\COMMON FILES\GTK2.0\bin;c:\program files\jdk\software\gpl\ghostscript\gs-8.15\bin;c:\program files\jdk\software\gpl\ghostscript\gs-8.15\lib;C:\Subversion\bin;c:\naven-2.0.4\bin
Jun 19, 2007 12:18:42 PM org.apache.coyote.http11.Http11Protocol init
INFO: Initializing Coyote HTTP/1.1 on http-8080
Jun 19, 2007 12:18:42 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 1884 ms
Jun 19, 2007 12:18:42 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
Jun 19, 2007 12:18:42 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.13
Jun 19, 2007 12:18:45 PM org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
Jun 19, 2007 12:18:45 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Jun 19, 2007 12:18:45 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/32 config=null
Jun 19, 2007 12:18:45 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 2861 ms
```

Open a browser and go to `http://localhost:8080` and you should see:



Let's shut it down by changing to `c:\tomcat\bin` and running `shutdown.bat`.

Installing Wicket

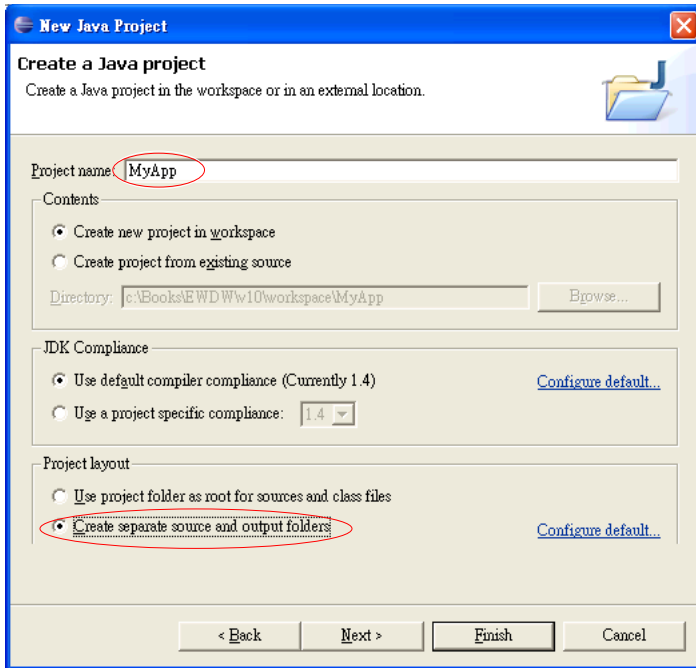
Next, go to <http://wicket.apache.org> to download a binary package of Wicket. Suppose that it is `apache-wicket-1.3.0.zip`. Unzip it into a folder, say `c:\wicket`.

In addition, Wicket uses a few jar files from other projects. So, go to <http://www.agileskills2.org/EWDW/wicket/lib>, download the jar files there and put them into `c:\wicket\lib`.

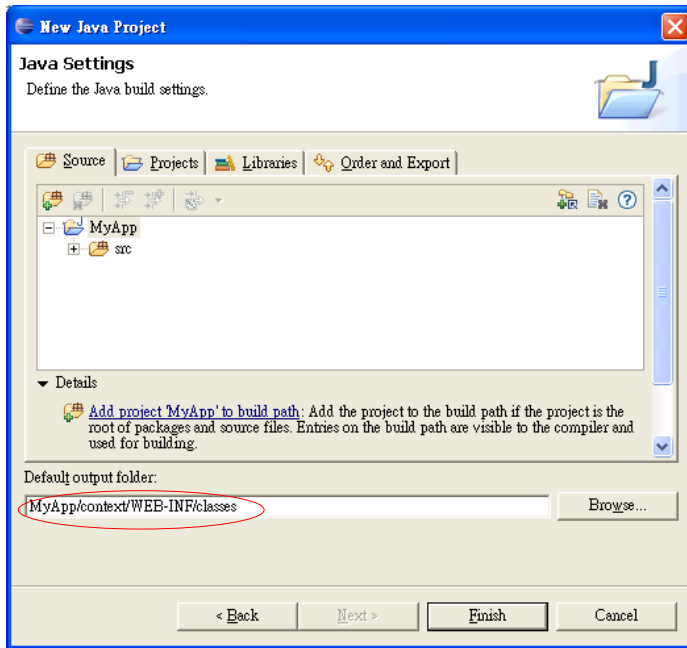
That's it. You can't run it yet because Wicket is a library, not an application.

Creating a Hello Word application

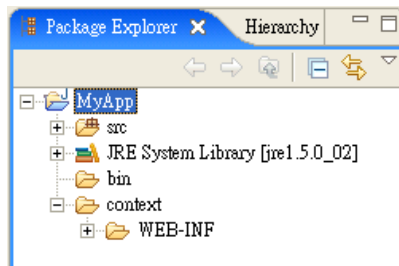
Now, create a new Java project. Name it "MyApp" and make sure it uses a separate output folder:



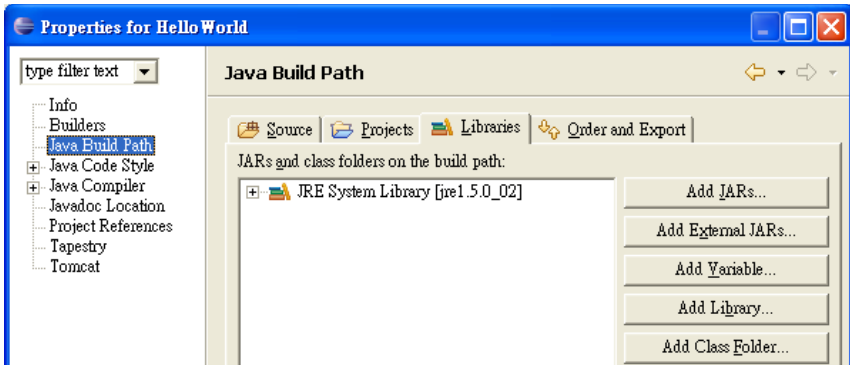
Set the output folder as shown below:



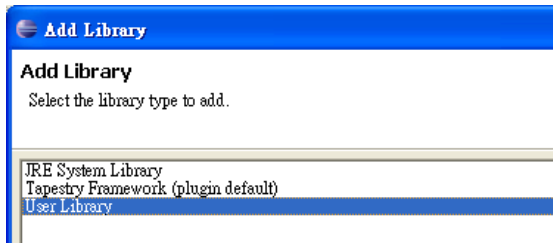
Finally, you should see the project structure:



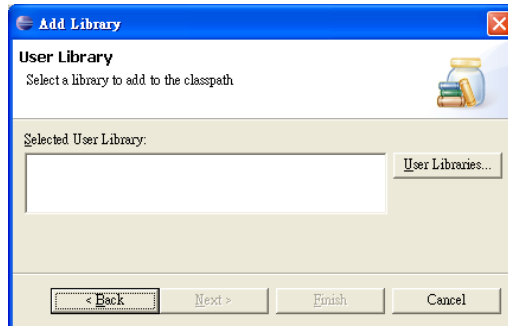
The bin folder is useless so you can delete it. Then right click the project and choose "Properties", choose "Java Build Path" on the left hand side, choose the "Libraries" tab:



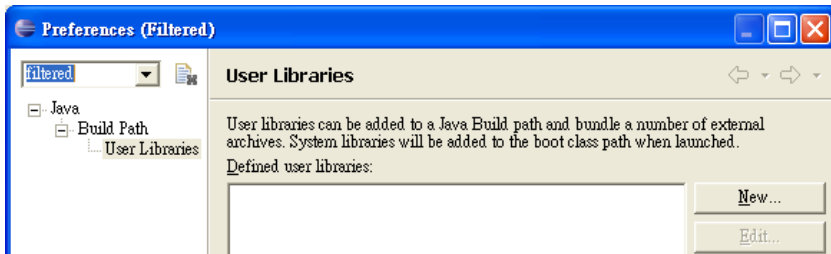
Click "Add Library" and choose "User Library":



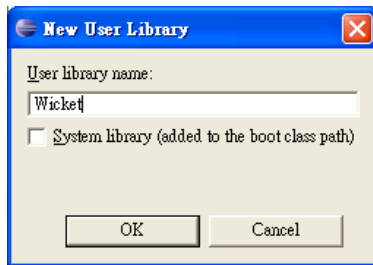
Click "Next":



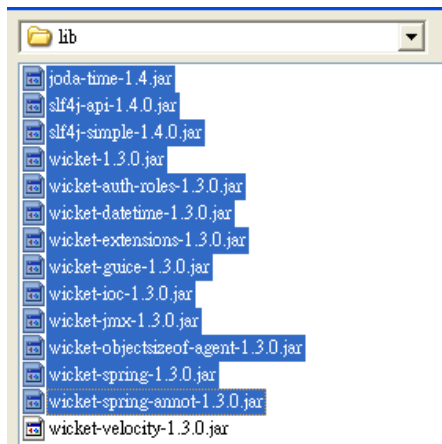
Click "User Libraries" to define your own Wicket library:



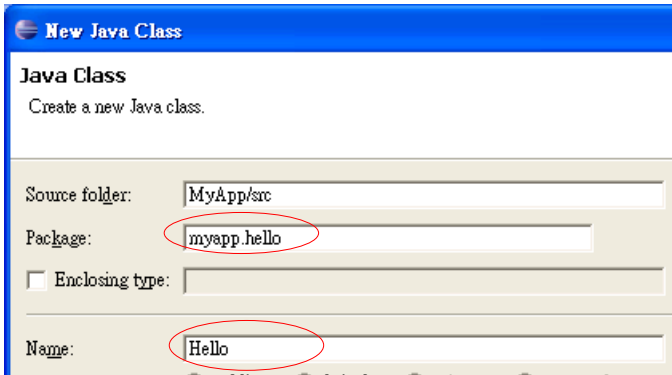
Click "New" to define a new one and enter "Wicket" as the name of the library:



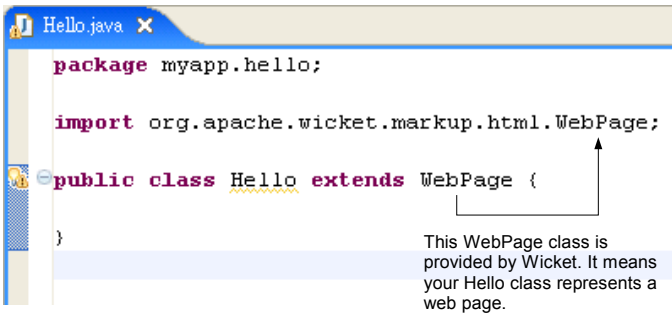
Click "Add JARs", browse to `c:\wicket\lib` and add all the jar files there except `wicket-velocity-1.3.0.jar`:



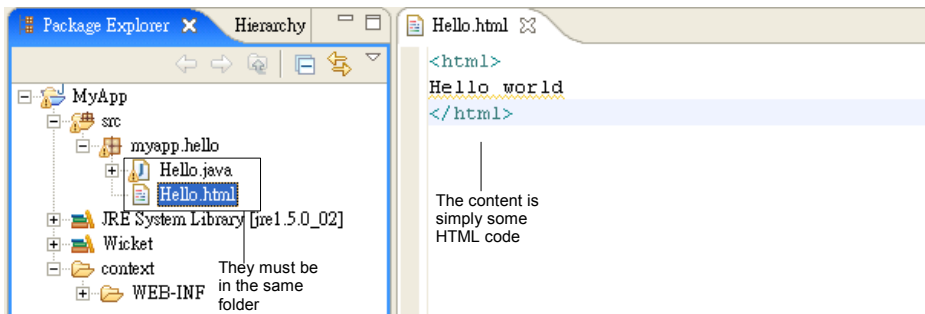
Then close all the dialog boxes. Next, create a new class named Hello in the `myapp.hello` package:



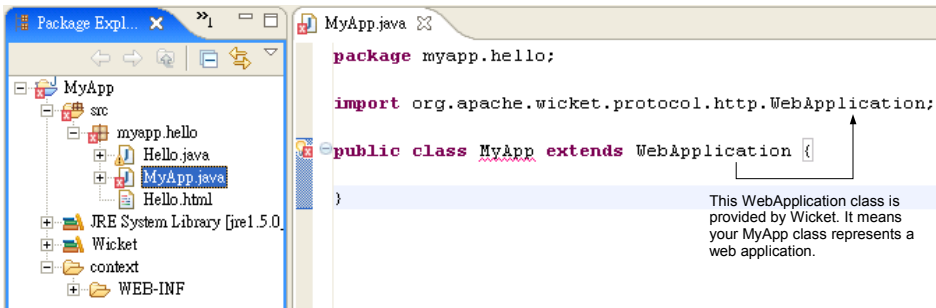
Input the content like this:



This class represents a web page in your application. Here you'd like it to display "Hello world". To do that, create a file Hello.html in the same folder as Hello.java and input the content:



Now your page is done! How to display it? Create a class MyApp in the same package:



You may notice that `MyApp` is marked as in error. This is because it must implement an abstract method `getHomePage()`. Define it now:

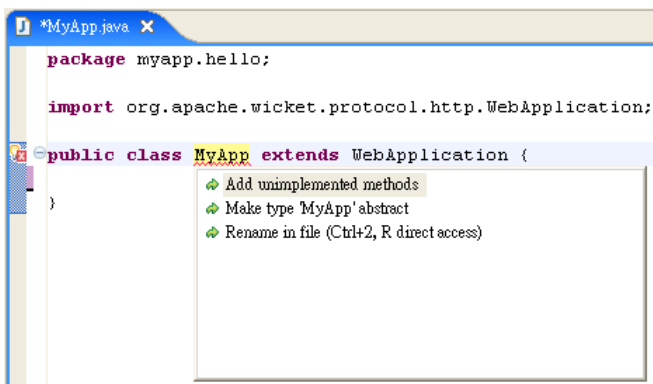
```
package myapp.hello;

import org.apache.wicket.protocol.http.WebApplication;

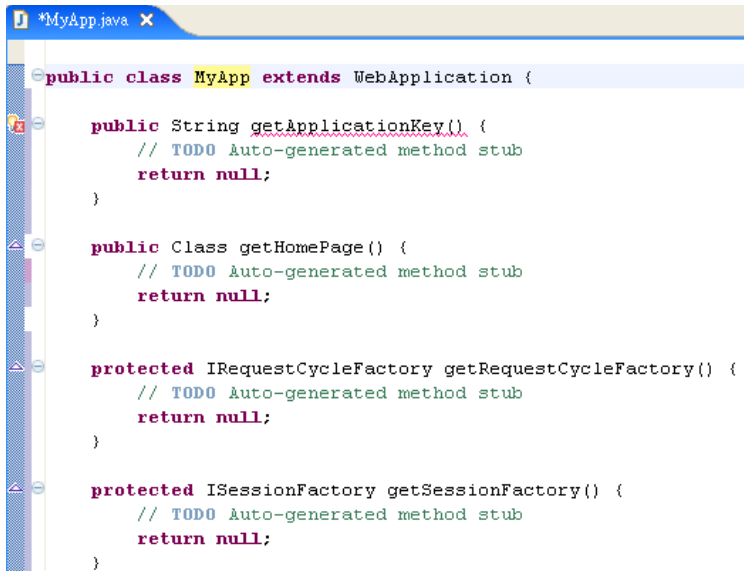
public class MyApp extends WebApplication {
    public Class getHomePage() {
        return Hello.class;
    }
}
```

The `Hello` class (page) is the home page of this application, i.e., it is the default page of the application.

If you're familiar with Eclipse, you may be tempted to use the "Add unimplemented methods" quick fix to add the method:



But due to a bug probably in Eclipse, you will get a lot more methods:



```
public class MyApp extends WebApplication {

    public String getApplicationKey() {
        // TODO Auto-generated method stub
        return null;
    }

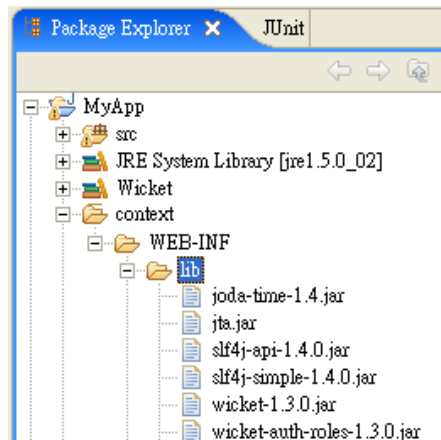
    public Class getHomePage() {
        // TODO Auto-generated method stub
        return null;
    }

    protected IRequestCycleFactory getRequestCycleFactory() {
        // TODO Auto-generated method stub
        return null;
    }

    protected ISessionFactory getSessionFactory() {
        // TODO Auto-generated method stub
        return null;
    }
}
```

So, don't do that. This problem only happens with the `WebApplication` class in Wicket. Other classes are fine.

Next, you need to make the Wicket jar files available to this application at runtime. To do that, create a `lib` folder under your `context/WEB-INF` folder and then copy all the jar files in `c:\wicket\lib` except `wicket-velocity-1.3.0.jar` into there:



Next, create a file `web.xml` in `context/WEB-INF` with the following content. This file is called the "deployment descriptor":

```

<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/TR/xmlschema-1/"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">
  <display-name>MyApp</display-name>
  <filter>
    <filter-name>WicketFilter</filter-name>
    <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
    <init-param>
      <param-name>applicationClassName</param-name>
      <param-value>myapp.hello.MyApp</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>WicketFilter</filter-name>
    <url-pattern>/app/*</url-pattern>
  </filter-mapping>
</web-app>

```

Tell Wicket that myapp.hello.MyApp is the class of your web application. This way, it will create an instance of this class and launch it.

Apart from the `applicationClassName` parameter, you can ignore the meaning of the rest for now. To make this application run in Tomcat, you must register it with Tomcat. To do that, create a file `MyApp.xml` in `c:\tomcat\conf\Catalina\localhost` (create this folder if it doesn't yet exist):

This file is called the "context descriptor". It tells Tomcat that you have a web application (yes, a web application is called a "context").

MyApp.xml

```

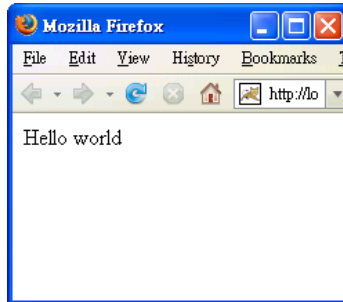
<Context
  docBase="c:/workspace/MyApp/context"
  reloadable="true"/>

```

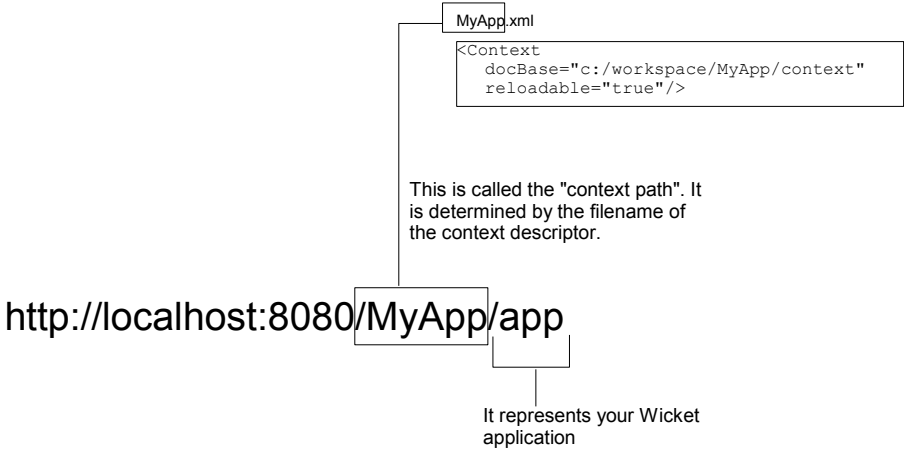
It tells Tomcat to reload this web application if any of its class files is changed

Tell Tomcat that the application's files can be found in `c:\workspace\MyApp\context`

Now, start Tomcat (by running `startup.bat`). To run your application, run a browser and try to go to `http://localhost:8080/MyApp/app`. You should see:



What does this URL mean? It is interpreted this way:

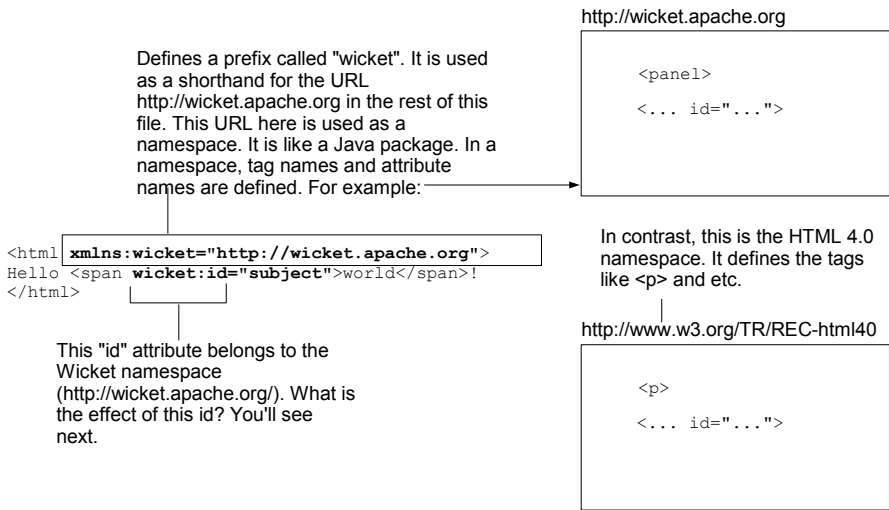


Generating dynamic content

Displaying "Hello World" is not particularly interesting. Next, you'll generate the message dynamically in Java. First, modify Hello.html as:

```
<html>
Hello <span>world</span>
</html>
```

`` is just a regular HTML element. It is used to enclose a section of HTML code. Next, add an attribute to this span:



Next, modify `Hello.java`:

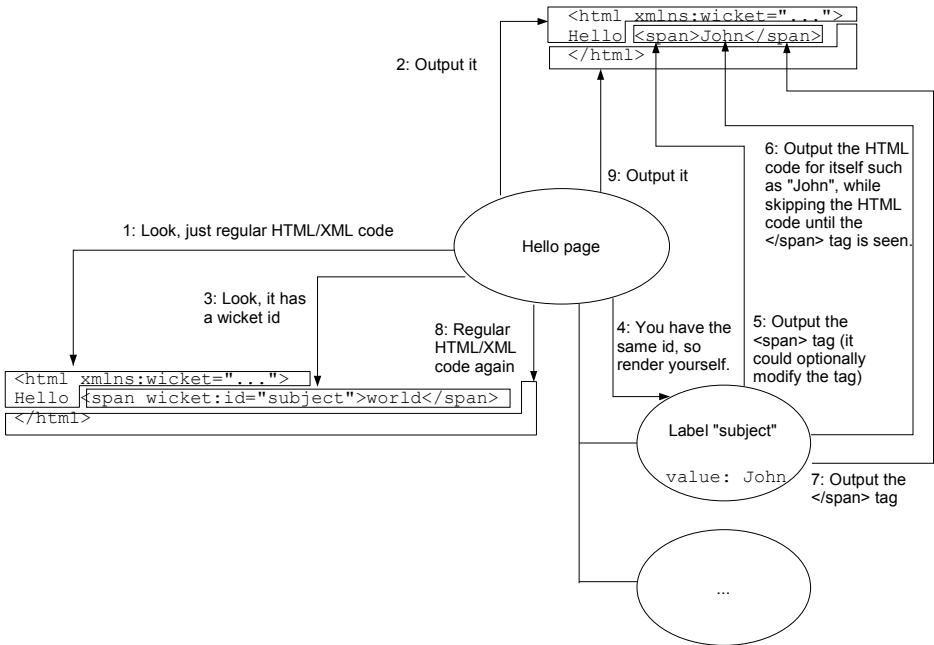
```
import org.apache.wicket.markup.html.basic.Label;

public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

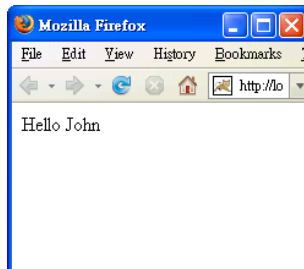
Add the Label component to the Hello page

Create a Label component whose id is "subject". It will output the string "John".

When Wicket displays (i.e., renders) the Hello page, it will create a Hello page object and its constructor will create the Label component and add it as a child. Then the Hello page will basically output the code in `Hello.html` (check the diagram below). When it finds that the span has a Wicket id of "subject", it will look up its children to find a child with this id. Here it will find the Label. Then it will ask the Label to render. The Label will print "John", while skipping the HTML code in `Hello.html` until the `` tag is passed:



Now run the application and you'll see:



As you can see, Hello.html is acting as a template for the Hello page. Each dynamic part in the page is like a blank to be filled in and you just mark each one using a Wicket id. So Hello.html is called the "template" or "markup" for the Hello page. In addition, the `...` element is said to be "associated" with the "subject" component.

Common errors in Wicket applications

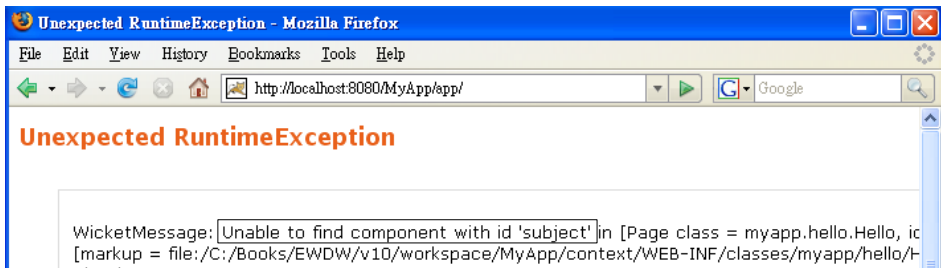
A very common error in Wicket applications is that for example, you have `wicket:id="subject"` in the template and have created a "subject" component in Java but forget to add it to the page:

```
<html xmlns:wicket="...">
Hello <span wicket:id="subject">world</span>
</html>
```

```
public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

Forget to add it to the page!

Then when you run the application, you'll get an exception (shown below). Whenever you see an exception saying it can't find a component with a certain id, check if you have really added the component to the page.



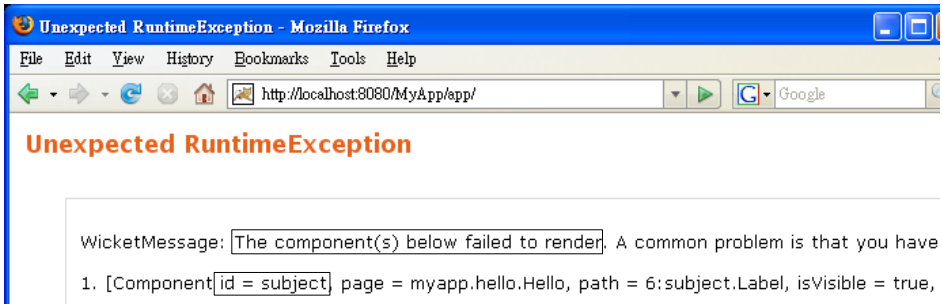
Another very common error is the opposite: You have added the component to the page but forget to add wicket:id to the template:

```
<html xmlns:wicket="...">
Hello <span wicket:id="subject">world</span>
</html>
```

Forget to mark it as a component!

```
public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

Now when you run it, you'll get another exception (shown below). Whenever you see an exception saying that a component failed to render, check if you have really a wicket:id in the template.



Now, undo the changes to make sure the code still works.

More rigorous version of the template

Strictly speaking, Hello.html should really be:

<p>As no prefix is provided, it means you'll use the following namespace as the default namespace.</p>	<p>It declares that this document is supposed to conform to the XHTML standard</p>
--	--

```

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:wicket="http://wicket.apache.org">
<head>
  <title>hello</title>
</head>
<body>
<p>Hello <span wicket:id="subject">world</span></p>
</body>
</html>

```

This is the XHTML namespace

This "strict" version complies with the so-called XHTML standard. In XHTML people can introduce tags and attributes from foreign namespaces such as those from the Wicket namespace, without making the document invalid.

Simpler version of the template

For simplicity, in this book we will not adhere to the strict XHTML. In fact, we will strive to make the code as simple as possible. For example, we will even omit the Wicket prefix declaration:

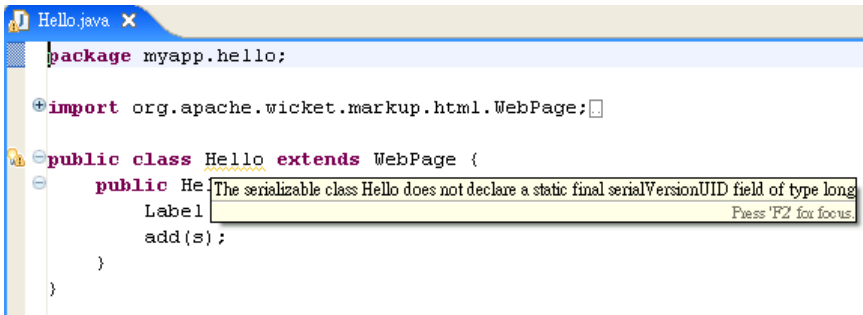
```
<html xmlns:wicket="http://wicket.apache.org">
Hello <span wicket:id="subject">world</span>
</html>
```

As long as you use "wicket" as the prefix, Wicket will assume it means the Wicket namespace.

The application will continue to work.

Page objects are serializable

You may have noticed that there is a warning in Hello.java:

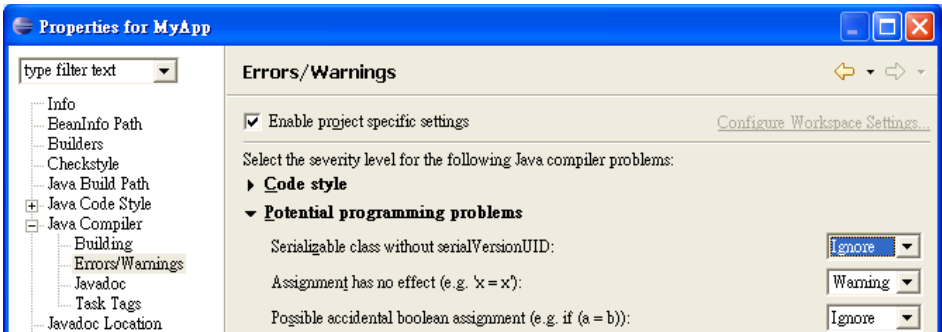


What does it mean? In Wicket page objects may be stored into the hard disk. To do that, the WebPage class implements the Serializable interface. Therefore your Hello class is also implementing Serializable. A rule in Java 5 states that such a serializable class should define a version id like this:

```
public class Hello extends WebPage {
    private static final long serialVersionUID = 1L;

    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

We won't explain the exact purpose of such a version id. However, you need to know that such a version id is not strictly required. Therefore, for simplicity, we will configure Eclipse to ignore this warning:



Debugging a Wicket application

To debug your application in Eclipse, you need to set two more environment variables for Tomcat and launch it in a special way:

```
fygdrive/c/tomcat
C:\tomcat\bin>set JPDA_ADDRESS=8000
C:\tomcat\bin>set JPDA_TRANSPORT=dt_socket
C:\tomcat\bin>catalina jpda start_
```

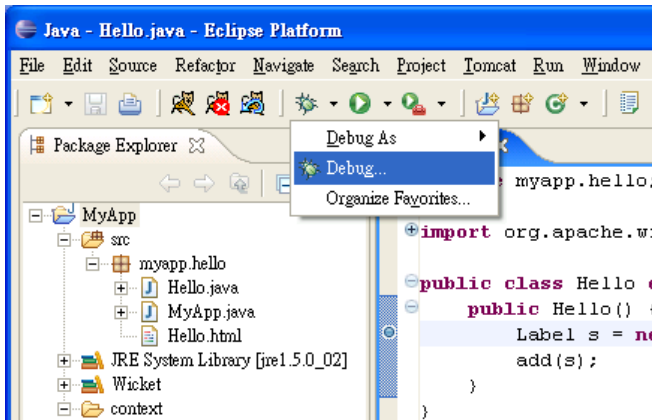
Note that you're now launching it using `catalina.bat` instead of `startup.bat`. This way Tomcat will run the JVM in debug mode and the JVM will listen for connections on port 8000. Later you'll tell Eclipse to connect to this port. Now, set a breakpoint here:

```
Hello.java x
package myapp.hello;

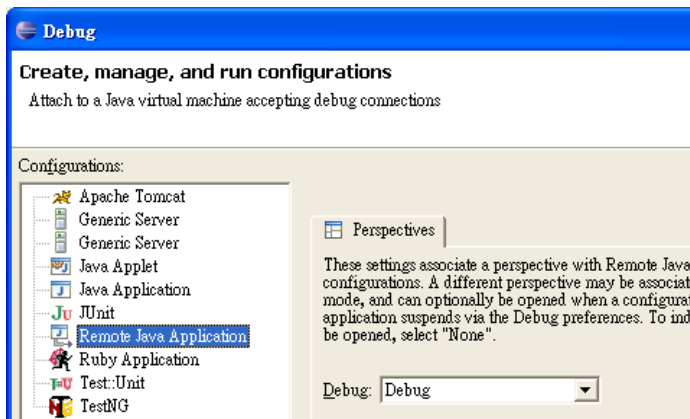
import org.apache.wicket.markup.html.WebPage;

public class Hello extends WebPage {
    public Hello() {
        Label s = new Label("subject", "John");
        add(s);
    }
}
```

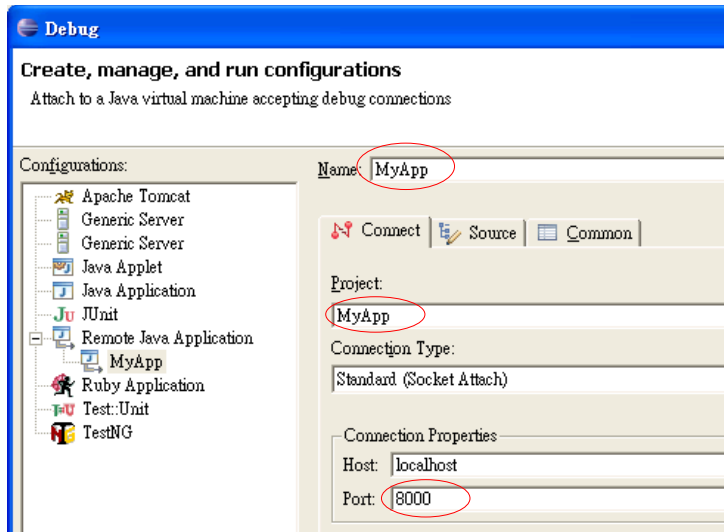
Choose "Debug":



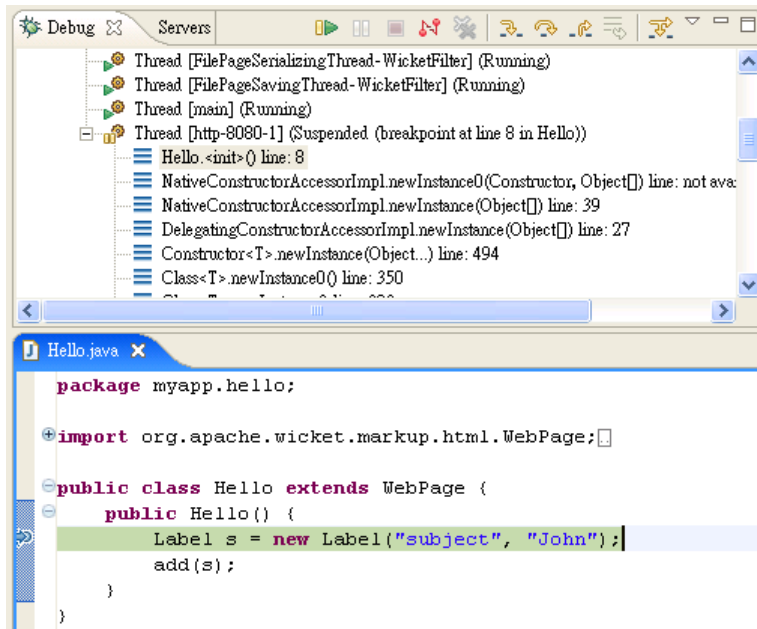
The following window will appear:



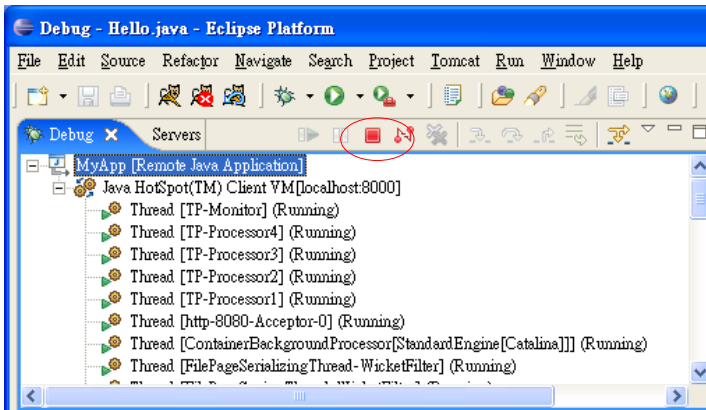
Right click "Remote Java Application" and choose "New". Browse to select your MyApp project and make sure the port is 8000:



Click "Debug" to connect to the JVM in Tomcat. Now go to the browser to load the page again. Eclipse will stop at the breakpoint:



Then you can step through the program, check the variables and whatever. To stop the debug session, choose the process and click the Stop icon:



Having to set all those environment variables every time is not fun. So, you may create a batch file `c:\tomcat\bin\debug.bat`:

```
set JPDA_ADDRESS=8000
set JPDA_TRANSPORT=dt_socket
catalina jpda start
```

Then in the future you can just run `debug.bat` to start Tomcat in debug mode.

Summary

To develop a Wicket application, you can install Tomcat and Eclipse.

To install Wicket, just unzip it into a folder. It is just a bunch of jar files. Copy the jar files into your `context/WEB-INF/lib` so that they are available to your web application.

Each page in a Wicket application is implemented by two files: a Java class and its template. They must be in the same folder. A Wicket application must contain an application class. Its major purpose is to tell Wicket which is the Java class for the home page. How does Wicket know which class is the application class? You do it in `web.xml`.

To register a web application with Tomcat, you need to create a `web.xml` file and a context descriptor to tell Tomcat where the application's files can be found.

To use a Wicket application, you can enter a URL to ask Wicket to display the home page.

When displaying a certain page, Wicket will create the page object and ask it to render. The page object will read its HTML file (the template) and basically output what's in the HTML file. But if there is a tag with a wicket id in the HTML file, it will locate a child component with that id in the page (the component it is associated with) and ask it to output HTML for itself.

A Label component will output the start tag as in the HTML file, then some plain

text as HTML code to replace the element body and finally the end tag.

A common error in Wicket applications is that you have `wicket:id` in the template and have indeed created the component but forget to add the component to the page. Another common error is the opposite: You have added the component to the page but forget to add `wicket:id` in the template.

To debug a Wicket application, tell Tomcat to run the JVM in debug mode, set a breakpoint in the Java code and make a Debug configuration in Eclipse to connect to that JVM.

Chapter 2

Using Forms

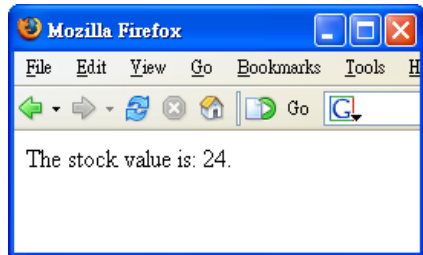
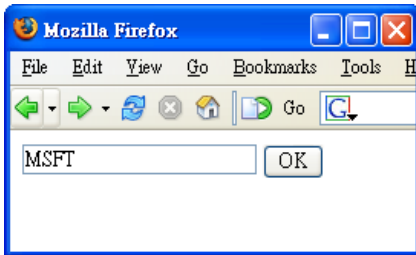


What's in this chapter?

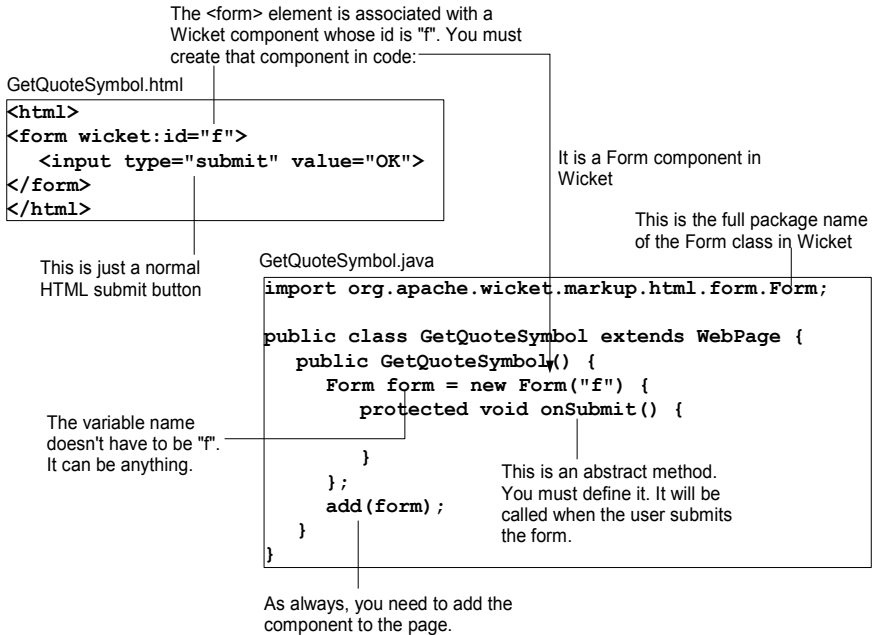
In this chapter you'll learn how to use forms to get input from the user.

Developing a stock quote application

Suppose that you'd like to develop an application like this:



That is, the user can enter the stock id and click OK, then the stock value will be displayed. As a first step, you'll create the input page without the text field first. Let's call this page `GetQuoteSymbol`. To do that, in your existing `MyApp` project, create a `GetQuoteSymbol` class and `GetQuoteSymbol.html` in the `myapp.stockquote` package:



The next step is to display the result page. Let's call it QuoteResult. Create QuoteResult.html and QuoteResult.java in the same package. QuoteResult.html is:

```
<html>
The stock value is: <span wicket:id="v">100</span>.
</html>
```

QuoteResult.java is:

```
public class QuoteResult extends WebPage {
  public QuoteResult(int stockValue) {
    add(new Label("v", Integer.toString(stockValue)));
  }
}
```

You need to pass the stock value to the constructor

The id used in the template

Convert the stock value to a string

Now, to display this page on form submission, modify GetQuoteSymbol.java:

```

public class GetQuoteSymbol extends WebPage {
    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                QuoteResult quoteResult = new QuoteResult(123);
                setResponsePage(quoteResult);
            }
        };
        add(form);
    }
}

```

Create a QuoteResult page and pass it an arbitrary stock value for the moment

Display this page to the browser

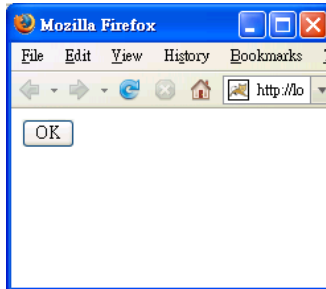
Now, you're about to run the application. However, before that, you need to modify MyApp.java to use GetQuoteSymbol as the home page:

```

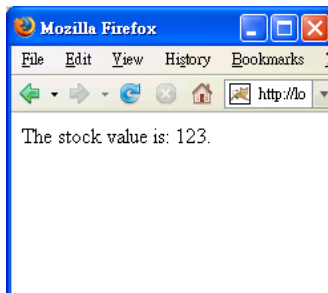
public class MyApp extends WebApplication {
    public Class getHomePage() {
        return GetQuoteSymbol.class;
    }
}

```

Now, run the application by going to <http://localhost:8080/MyApp/app>, you should see:



Clicking OK will display:



So, it's working. Next, you'll add the text field. Modify GetQuoteSymbol.html and GetQuoteSymbol.java:

```
<html>
<form wicket:id="f">
  <input type="text" wicket:id="sym">
  <input type="submit" value="OK">
</form>
</html>
```

It is a TextField component in Wicket

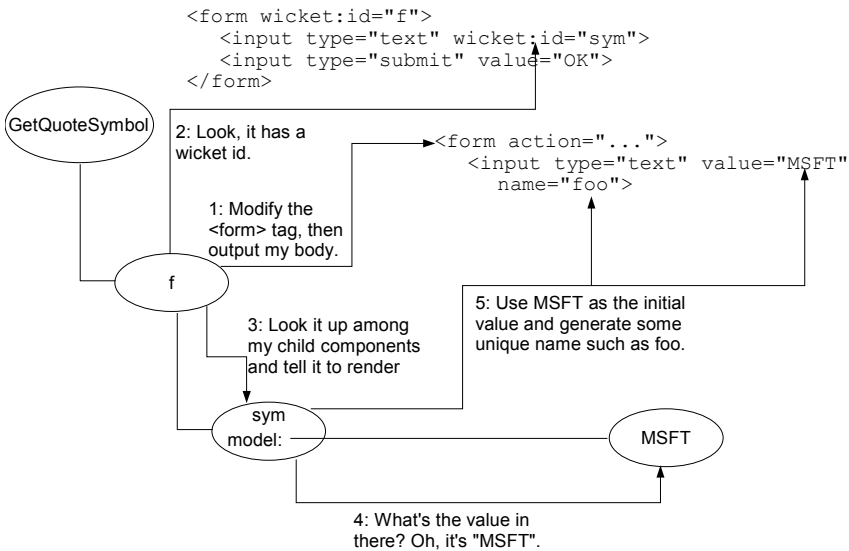
```
import org.apache.wicket.markup.html.form.TextField;
import org.apache.wicket.model.Model;

public class GetQuoteSymbol extends WebPage {
    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                QuoteResult quoteResult = new QuoteResult(123);
                setResponsePage(quoteResult);
            }
        };
        Model model = new Model("MSFT");
        TextField symbol = new TextField("sym", model);
        form.add(symbol);
        add(form);
    }
}
```

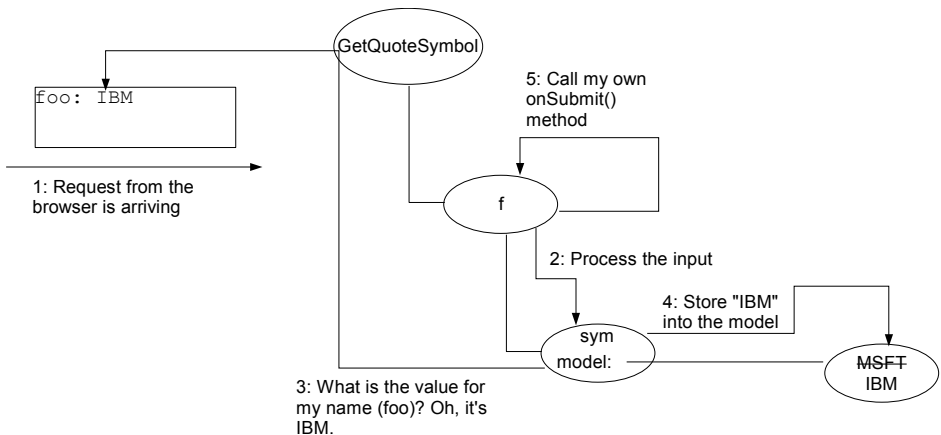
Use this object as the "model". What does it mean?

Add the text field to the form, not to the page! Why?

How does it work? When the form renders itself (see the diagram below), after modifying the `<form>` tag, it will render its body. When it sees the `wicket:id="sym"`, it will look up a child with id "sym" in itself, not in the page. Therefore, you must add the text field to the form, not to the page. It will find the symbol text field and ask it to render. To render itself, the text field will use its model. A model is just a container holding a value. In this case, initially the value is a string "MSFT". So, the text field gets the string "MSFT" from the model and uses it as the initial value of the text field. In addition, it will also generate a unique name for the `<input>` element:



Suppose that the user changes the symbol from "MSFT" to "IBM" and clicks OK, then the form component will get control. It will ask each of its children to process the input. The text field will see what is the value of its unique name (foo). Here it will find the string "IBM". Then it will store the string "IBM" into the model. Finally, the form component will call its `onSubmit()` method as mentioned earlier:



Now the symbol is stored into the model, you need to use it in `onSubmit()`. To do that, you can store the model into a field:


```

public class GetQuoteSymbol extends WebPage {
    private Model model;

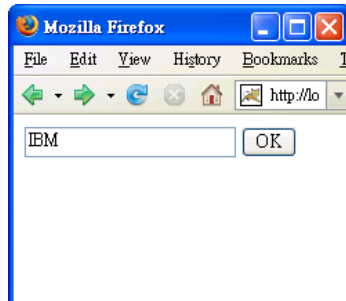
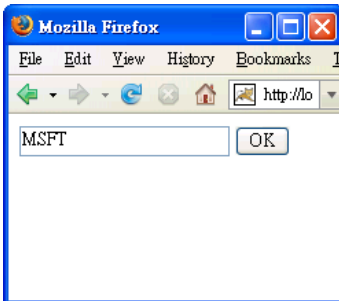
    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                int stockValue = sym.hashCode() % 100;
                QuoteResult quoteResult = new QuoteResult(stockValue);
                setResponsePage(quoteResult);
            }
        };
        Model model = new Model("MSFT");
        TextField symbol = new TextField("sym", model);
        form.add(symbol);
        add(form);
    }
}

```

Get the value from it. You know it has to be a string because a text field treats the input as a string by default.

Normally you should find out the stock value for the given symbol. Here, you just get a fake value: the hash code of the symbol modulo 100.

Now run it and it should work:

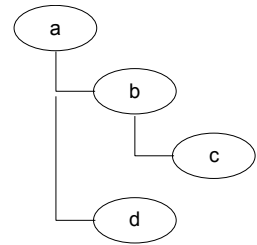
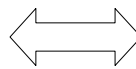


Mismatch in component hierarchy

As said before, because the the text field is in the body of the form component in the template, you must add the text field to the form component in Java code. That is, the component hierarchy in the template must match that in the Java code:

```
<span wicket:id="a">
  <span wicket:id="b">
    <span wicket:id="c">
    </span>
  </span>
  <span wicket:id="d">
  </span>
</span>
```

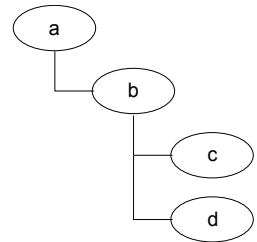
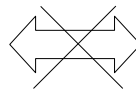
Component hierarchies
must match



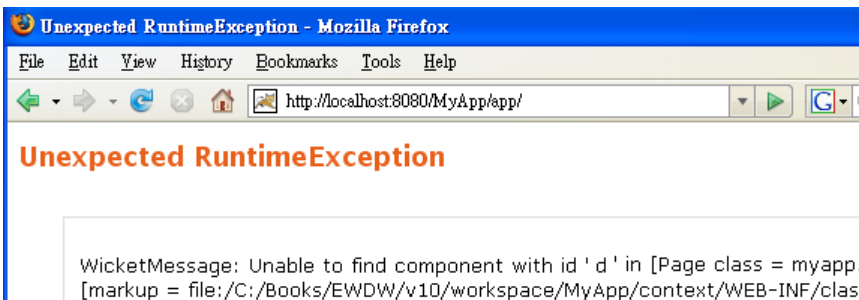
What if you make a mistake:

```
<span wicket:id="a">
  <span wicket:id="b">
    <span wicket:id="c">
    </span>
  </span>
  <span wicket:id="d">
  </span>
</span>
```

Component hierarchies
don't match

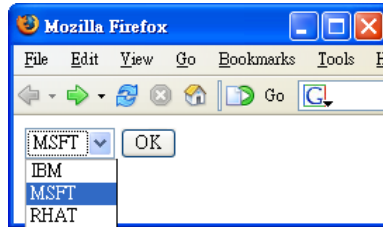


Then when "a" tries to find a child "d", it will fail. Then you'll see the exception (shown below) saying a component can't be found. It is the same exception you saw in the previous chapter when you forgot to add a component to the page. This time it is not because you forgot to add the component; it's because you added it to the wrong parent.



Using a combo box

Suppose that you'd like to change the application so that the user will choose from a list of stock symbols instead of typing in one:



To do that, modify GetQuoteSymbol.html:

```

Use a <select>
instead of <input>
<html>
<form wicket:id="f">
  <select wicket:id="sym">
    <option>MSFT</option>
    <option>IBM</option>
  </select>
  <input type="submit" value="OK">
</form>
</html>

```

These options are here just for preview only, e.g., when your designer is designing this HTML file using a tool like dreamweaver. They will be completely replaced by the output of the "sym" component.

Now the "sym" component should no longer be a TextField, but a DropDownChoice:

```

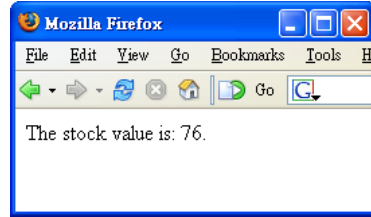
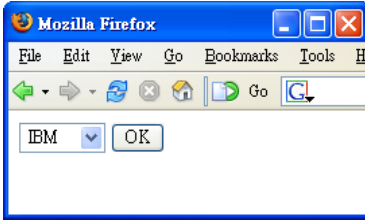
public class GetQuoteSymbol extends WebPage {
    private Model model;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                int stockValue = sym.hashCode() % 100;
                QuoteResult quoteResult = new QuoteResult(stockValue);
                setResponsePage(quoteResult);
            }
        };
        model = new Model("MSFT");
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        TextField symbol = new TextField("sym", model);
        form.add(symbol);
        add(form);
    }
}

```

Specify the available options as a java.util.List

Note that the DropDownChoice will actually check if it is associated with a <select> tag. If not, it will throw an exception. Now run the application and it should work:

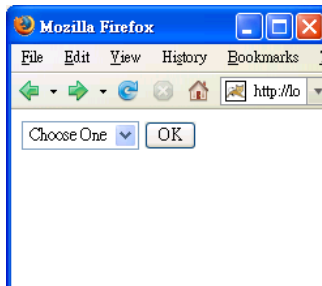


Currently you're using "MSFT" as the default symbol. What if there is no sensible default? You can just create the Model object without specifying any value:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                int stockValue = sym.hashCode() % 100;
                QuoteResult quoteResult = new QuoteResult(stockValue);
                setResponsePage(quoteResult);
            }
        };
        model = new Model("MSFT");
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        add(form);
    }
}
```

Then the initial value in the Model will be null. In that case, what will the initial selected item in the DropDownChoice? When the value is null or is not one of those on the list, it will show an extra item "Choose one":



If the user just clicks OK without choosing any value, the symbol will be null. So, you need to check for it:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;

    public GetQuoteSymbol() {
        Form form = new Form("f") {
```

```

protected void onSubmit() {
    String sym = (String) model.getObject();
    if (sym != null) {
        int stockValue = sym.hashCode() % 100;
        QuoteResult quoteResult = new QuoteResult(stockValue);
        setResponsePage(quoteResult);
    }
}
};
model = new Model();
List symbols = new ArrayList();
symbols.add("MSFT");
symbols.add("IBM");
symbols.add("RHAT");
DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
form.add(symbol);
add(form);
}
}

```

If the symbol is indeed null, you will not call `setResponsePage()`. What will happen then? Wicket will redisplay the page that handled the form submission, i.e., your `GetQuoteSymbol` page. This is good because you'd like the user to input the data again.

What if you'd like to display "Pick a symbol" instead of "Choose One"? You can create a text file `GetQuoteSymbol.properties` in the same folder as `GetQuoteSymbol.java`. Its content should be:

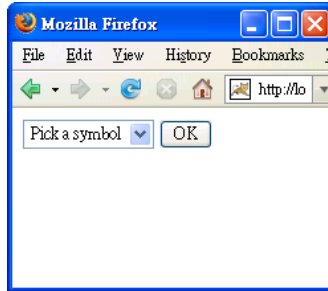
resource key	string value
null=Pick	a symbol

Here we say that "null" is the resource key. However, if you had two or more `DropDownChoice` components in the `GetQuoteSymbol` page, this line would affect all of them. To limit it to only the "sym" component, do it this way:

This is the id path from the page to the "sym" component. First, it goes to the form "f", then go to "sym".

└──┬──
f.sym.null=Pick a symbol

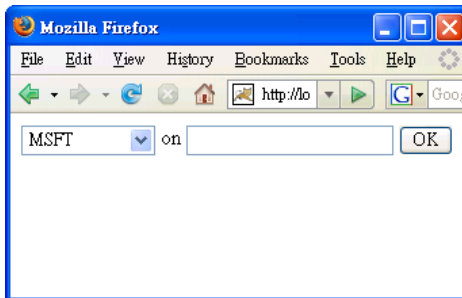
That is, you're qualifying the resource key using the id path to the component. What if both entries are present? The unqualified entry will serve as the default; if a qualified entry exists, it will override the unqualified one. Anyway, now run it and it should work:



If it doesn't work, make sure the application has been reloaded. For example, you may make some trivial changes to a Java class to trigger a reload.

Inputting a date

Suppose that you'd like to allow the user to query the stock value on a particular date:



To do that, modify GetQuoteSymbol.html:

```
<html>
<form wicket:id="f">
  <select wicket:id="sym">
    <option>MSFT</option>
    <option>IBM</option>
  </select>
  on <input type="text" wicket:id="quoteDate">
  <input type="submit" value="OK">
</form>
</html>
```

Define the "quoteDate" component in GetQuoteSymbol.java:

```

import java.util.Date;

public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

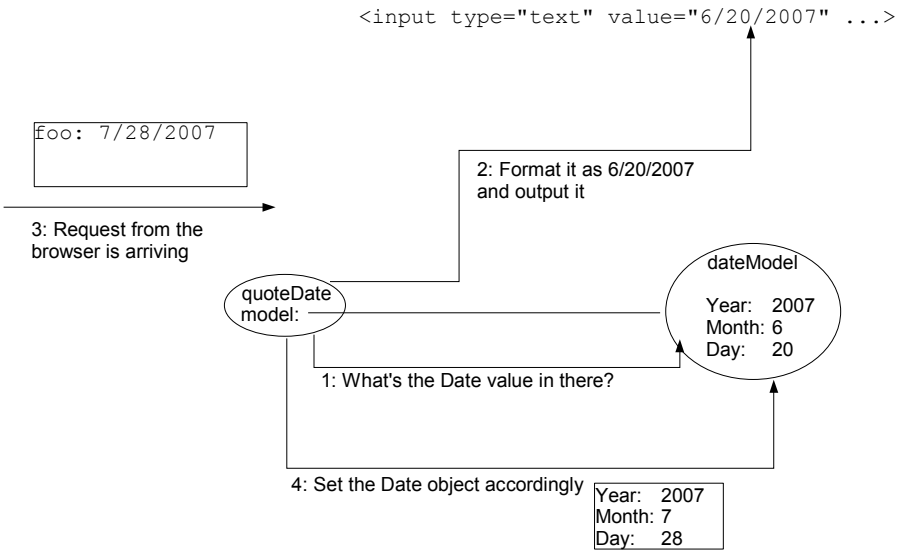
    public GetQuoteSymbol() {
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate =
            new TextField("quoteDate", dateModel, Date.class);
        form.add(quoteDate);
        add(form);
    }
}

```

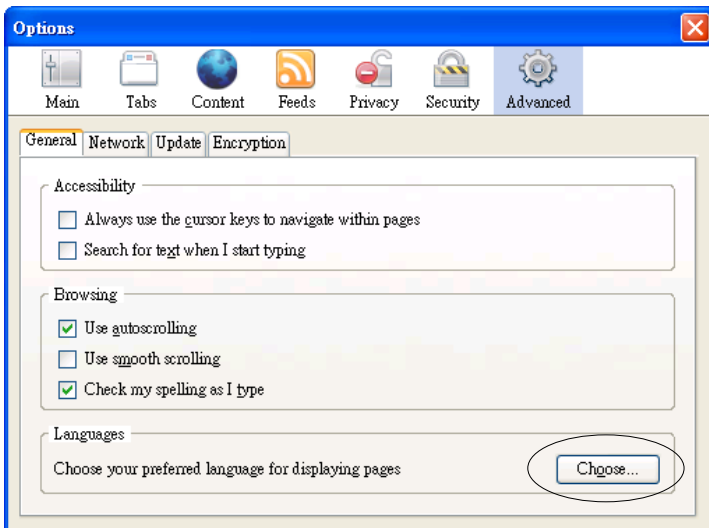
The value in the model is a java.util.Date object, not a string.

It tells the text field that the value stored in the model is a java.util.Date, not a string.

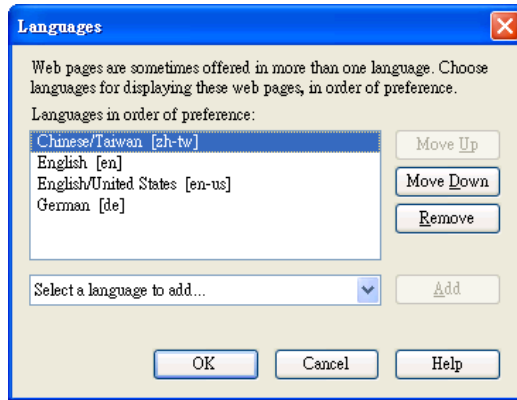
The `TextField` component knows about a few common types such as `java.util.Date`, `java.lang.Integer` and `java.lang.Double`. When its type parameter is specified as `Date.class`, on rendering (see the diagram below) it will try to get a `Date` object from the model, format it as a string and display it as the value in the `<input>` field. When the user submits the form, it will get the `<input>` field value (a string) and try to convert it back to a `Date` object and store it into the model:



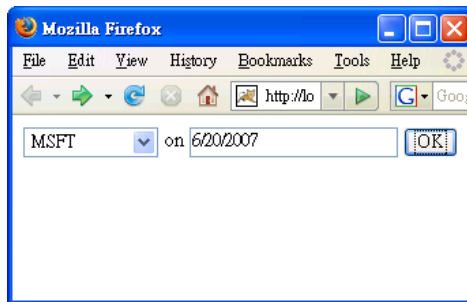
Will it format a Date as mm/dd/yyyy or dd/mm/yyyy or something else? First, it finds out the most preferred locale (language) of the browser. For example, in Firefox, it is set in "Tools | Options | Advanced":



Click "Choose":

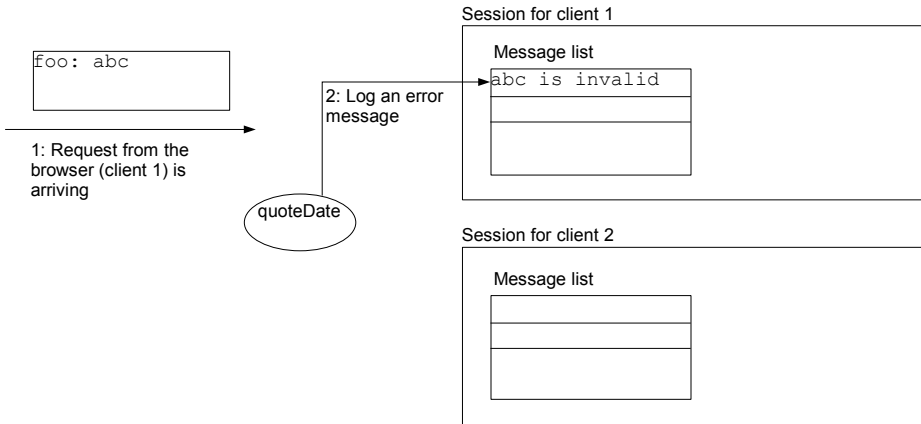


This information is sent in every request. Then the TextField will ask Java for the default date format for that locale. Now, run it and it should work:



Displaying feedback messages

What if the user enters some garbage like "abc" as the date (see the diagram below)? The TextField will fail to convert it back to a Date object. In that case, it will log an error message into a global list of messages. This list is stored in a memory area allocated for each currently connected client. Such a memory area is called the "session" for that client:



To display the list of messages to the user, modify `GetQuoteSymbol.html`:

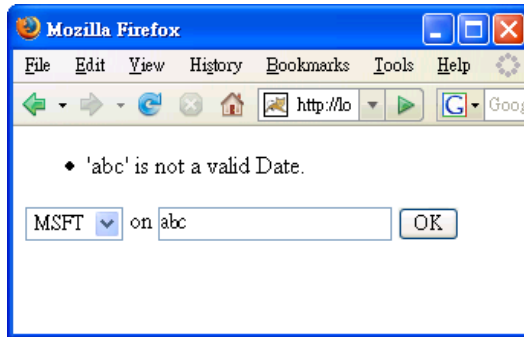
```
<html>
<span wicket:id="msgs"/>
<form wicket:id="f">
  <select wicket:id="sym">
    <option>MSFT</option>
    <option>IBM</option>
  </select>
  on <input type="text" wicket:id="quoteDate">
  <input type="submit" value="OK">
</form>
</html>
```

Modify `GetQuoteSymbol.java`:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        form.add(quoteDate);
        add(form);
    }
}
```

The FeedbackPanel class is coming from Wicket. It will display all the messages in the list (if there is no message, it will render nothing). Now, run the application, enter "abc" as the date and click OK, you'll see:



In addition, note that the original values of the form fields are redisplayed ("MSFT" and "abc"), no matter they are valid or not. Besides, the onSubmit() method of the form is not called. This happens when any of the form fields fails to update the value in its model.

Will the error message hang around forever? No. Once it is rendered, it will be deleted. To verify, just reload the page and the message will be gone.

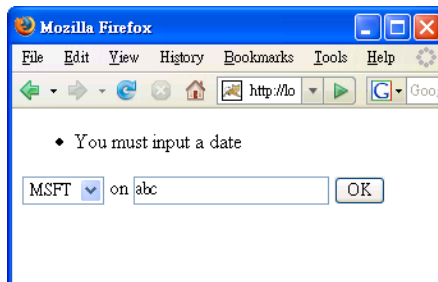
What if you don't like the default error message? For example, you'd like to display "You must input a date". To do that, modify GetQuoteSymbol.properties:

```
f.sym.null=Pick a symbol  
f.quoteDate.IConverter.Date=You must input a date
```

Diagram illustrating the mapping of the error message:

- The text "Pick a symbol" is connected by a vertical line to the label "Id path to the form component".
- The text "You must input a date" is connected by a vertical line to the label "Resource key".
- Horizontal lines connect the two lines above to the respective labels below.

Now run it and it will work:



Marking input as required

Now the TextField will not accept garbage. But what if the user enters an empty

string? By default the `TextField` assumes that you're allowing the input to be optional and will convert empty input to null (in this case as the `Date` object). Then your code will crash at the code below:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

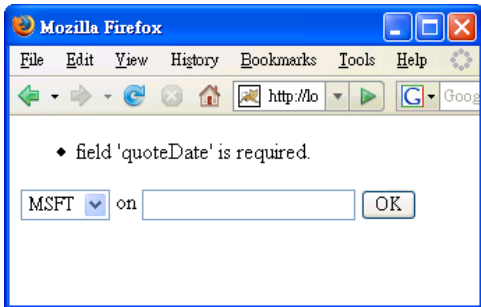
    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        form.add(quoteDate);
        add(form);
    }
}
```

To mark it as required, do it this way:

```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

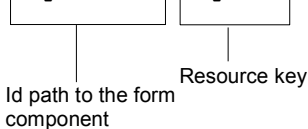
    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        quoteDate.setRequired(true);
        form.add(quoteDate);
        add(form);
    }
}
```

Now, run it while setting the date to empty, you'll see:

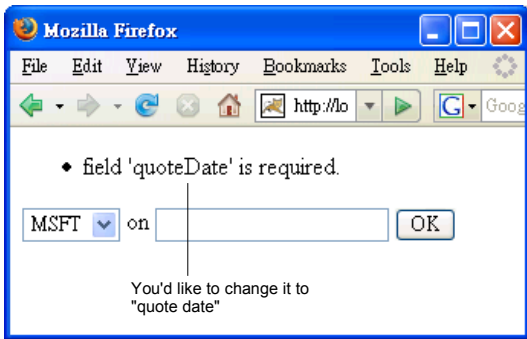


Again, if you don't like the error message, you can change it:

```
f.sym.null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The quote date is missing
```



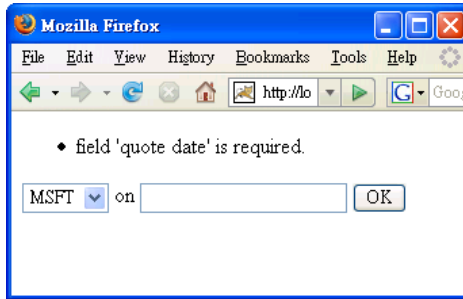
If you'd like to keep the default message but would like to call the field "quote date" instead of its component id ("quoteDate"):



You can do it this way:

```
f.sym.null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The quote date is missing
f.quoteDate=quote date
```

Now run it and it will work:



If you decide to use your own error message, you can also refer to the field name:

```
f.sym.null=Pick a symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The ${label} is missing
f.quoteDate=quote date
```

You can also mark the "sym" DropDownChoice as required:

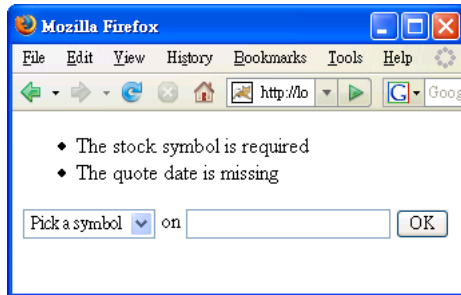
```
public class GetQuoteSymbol extends WebPage {
    private Model model;
    private Model dateModel;

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                String sym = (String) model.getObject();
                Date date = (Date) dateModel.getObject();
                if (sym != null) {
                    int stockValue = (sym + date.toString()).hashCode() % 100;
                    QuoteResult quoteResult = new QuoteResult(stockValue);
                    setResponsePage(quoteResult);
                }
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        symbol.setRequired(true);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        quoteDate.setRequired(true);
        form.add(quoteDate);
        add(form);
    }
}
```

You may set the error message and field name:

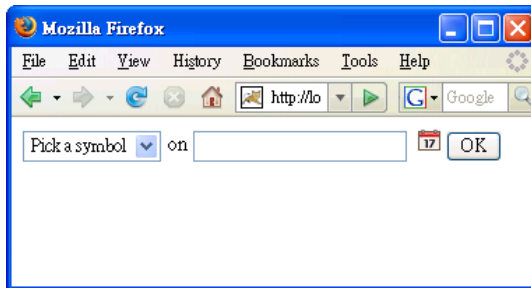
```
f.sym.null=Pick a symbol
f.sym.Required=The ${label} is required
f.sym=stock symbol
f.quoteDate.IConverter.Date=You must input a date
f.quoteDate.Required=The ${label} is missing
f.quoteDate=quote date
```

Now, run the application again and it should work:

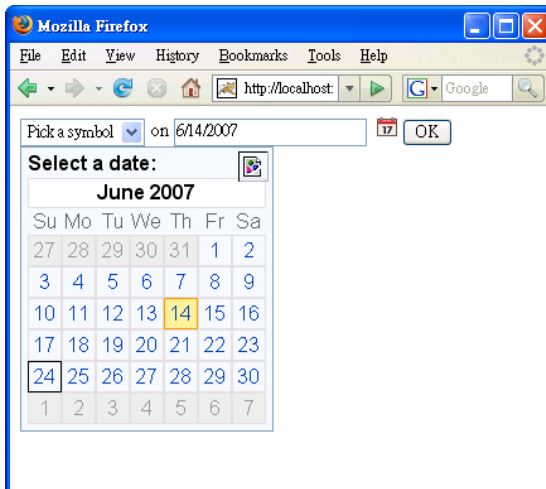


Using the DatePicker

In fact, you can also allow the user to choose a date:



Clicking on the calendar icon will display a calendar:



To do that, modify GetQuoteSymbol.html:

The calendar needs to use Javascript to work. The script will be put into the `<head>` element. So you must have one in the template.

```

<html>
<head></head>
<body>
<span wicket:id="msgs"/>
<form wicket:id="f">
  <select wicket:id="sym">
    <option>MSFT</option>
    <option>IBM</option>
  </select>
  on <input type="text" wicket:id="quoteDate">
  <input type="submit" value="OK">
</form>
</body>
</html>

```

If you have `<head>`, you should have `<body>`, otherwise the HTML will be very much invalid.

Modify GetQuoteSymbol.java:

```

import org.apache.wicket.extensions.yui.calendar.DatePicker;
import org.apache.wicket.util.convert.IConverter;
import org.apache.wicket.util.convert.converters.DateConverter;

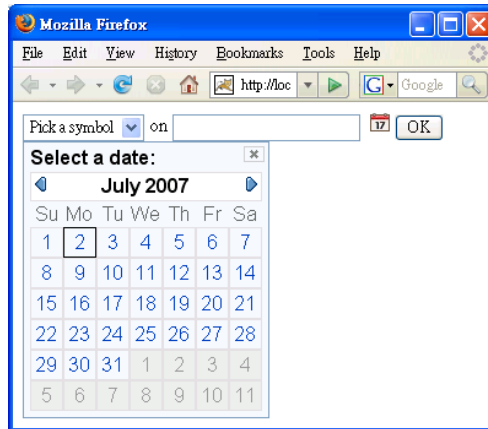
public class GetQuoteSymbol extends WebPage {
    ...

    public GetQuoteSymbol() {
        FeedbackPanel feedback = new FeedbackPanel("msgs");
        add(feedback);
        Form form = new Form("f") {
            protected void onSubmit() {
                ...
            }
        };
        model = new Model();
        List symbols = new ArrayList();
        symbols.add("MSFT");
        symbols.add("IBM");
        symbols.add("RHAT");
        DropDownChoice symbol = new DropDownChoice("sym", model, symbols);
        symbol.setRequired(true);
        form.add(symbol);
        dateModel = new Model();
        TextField quoteDate = new TextField("quoteDate", dateModel, Date.class);
        quoteDate.setRequired(true);
        quoteDate.add(new DatePicker());
        form.add(quoteDate);
        add(form);
    }
}

```

A `DatePicker` object is a "behavior". You're attaching it to the "quoteDate" component. When the "quoteDate" component is rendered or after it is rendered, a behavior will be given a chance to output extra HTML code. Here, it will output the code to show the calendar icon.

Now, run it and it should work:



Summary

The component hierarchy in the template must match that in Java code. Otherwise you'll get a component not found exception.

To get input from the user, use a Form component and put some form components in it. When the form is submitted, the `onSubmit()` method of the Form component will be called. In that method(), to tell Wicket which page to display next, call `setResponsePage()`. If you don't do that, Wicket will redisplay the page containing the Form component.

Form components such as `TextField` and `DropDownChoice` will get the current value from the model. When the form is submitted, each of them will get its value and store it into the model. If any of them fails to convert the value, an error message will be logged and the `onSubmit()` method won't be called.

You can customize the error message or the field name (label). The resource key for the error message starts with the id of the form component. To display error messages, you can use a `FeedbackPanel`. Once a message is rendered, it will be removed.

A session is a memory area allocated on the server for each currently connected client. The list of error messages is stored there.

A Form component can be marked as required. This way it won't accept empty input and will treat it as an error.

A `TextField` by default deals with strings. However, you can tell it that the value in the model is of a particular type such as `java.util.Date`. It knows a few common types such as `java.lang.Integer` and `java.lang.Double`.

To allow the user to choose a date from a calendar, use the `DatePicker` behavior. One or more behaviors can be added to a component to modify or add to the HTML output of the component.

Some components or behaviors such as DatePicker use Javascript. To work with them, you need to have a `<head>` element in the template.

Chapter 3

Validating Input

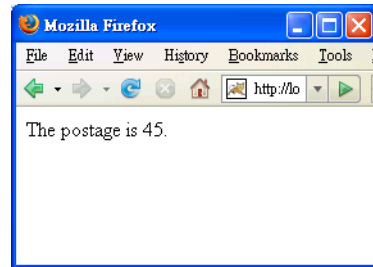
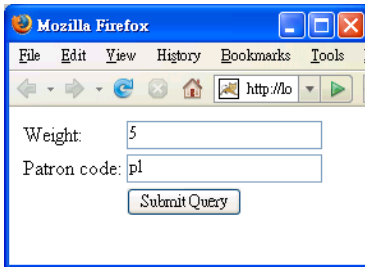


What's in this chapter?

In the previous chapter you've learned some basic ways of input validation: marking a field as required and specifying its data type. In this chapter you'll learn more advanced ways to validate input.

Postage calculator

Suppose that you'd like to develop an application to calculate the postage for sending a package from some place to another. The user will enter the weight of the package in kg (check the screenshots below). Optionally, he can enter a "patron code" identifying himself as a patron to get a certain discount. After clicking OK, it will display the postage:



To do that, in your existing MyApp project, create a `GetRequest` class and `GetRequest.html` in the `myapp.postage` package. `GetRequest.html` is like:

```
<html>
<form wicket:id="form">
<table>
<tr>
<td>Weight:</td>
<td><input type="text" wicket:id="weight"/></td>
</tr>
<tr>
<td>Patron code:</td>
<td><input type="text" wicket:id="patronCode"/></td>
</tr>
<tr>
<td></td>
<td><input type="submit"/></td>
</tr>
</table>
</form>
</html>
```

`GetRequest.java` is like:

```

public class GetRequest extends WebPage {
    private Model weightModel = new Model();
    private Model patronCodeModel = new Model();
    private Map patronCodeToDiscount;

    public GetRequest() {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
        Form form = new Form("form") {
            protected void onSubmit() {
                int weight = ((Integer) weightModel.getObject()).intValue();
                Integer discount = (Integer) patronCodeToDiscount
                    .get(patronCodeModel.getObject());
                int postagePerKg = 10;
                int postage = weight * postagePerKg;
                if (discount != null) {
                    postage = postage * discount.intValue() / 100;
                }
                ShowPostage showPostage = new ShowPostage(postage);
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", weightModel, Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode", patronCodeModel);
        form.add(patronCode);
        add(form);
    }
}

```

Models for the two input fields

Hard code some patrons and their discounts. For example, for the patron whose code is "p1", the discount is 90% (i.e., 10% off).

It has to be an Integer object because you have specified the type:

Use the patron code to look up the map to find out his discount

Create a result page instance (you'll create the page class next), pass the postage value to it and set it as the response page.

For simplicity, assume the postage per kg is \$10 to calculate the postage

Next, create the ShowPostage page. ShowPostage.html is like:

```

<html>
The postage is <span wicket:id="postage">10</span>.
</html>

```

ShowPostage.java is like:

```

public class ShowPostage extends WebPage {
    public ShowPostage(int postage) {
        add(new Label("postage", Integer.toString(postage)));
    }
}

```

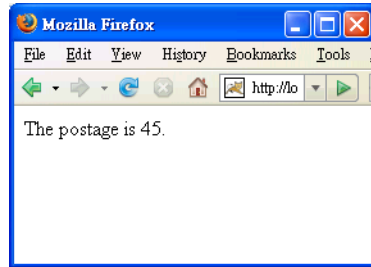
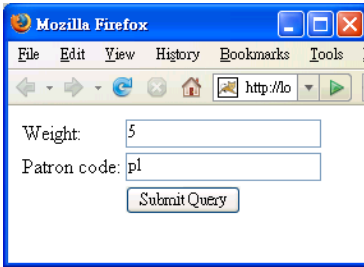
Now, you're about to run the application. However, before that, you need to modify MyApp.java to use GetRequest as the home page:

```

public class MyApp extends WebApplication {
    public Class getHomePage() {
        return GetRequest.class;
    }
}

```

Now, run the application by going to <http://localhost:8080/MyApp/app>, it should work:



Using an object to represent the request

At the moment you're calculating the postage in the `onSubmit()` method:

```
Form form = new Form("form") {
    protected void onSubmit() {
        int weight = ((Integer) weightModel.getObject()).intValue();
        Integer discount = (Integer) patronCodeToDiscount
            .get(patronCodeModel.getObject());
        int postagePerKg = 10;
        int postage = weight * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        ShowPostage showPostage = new ShowPostage(postage);
        setResponsePage(showPostage);
    }
};
```

This is no good. This kind of domain logic should go into a domain class. For example, let's create a class to represent the request and let the request calculate the postage itself:

```

package myapp.postage;

public class Request {
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

    static {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
    }

    public Request(int weight, String patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }

    public int getPostage() {
        Integer discount = (Integer) patronCodeToDiscount.get(patronCode);
        int postagePerKg = 10;
        int postage = weight * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        return postage;
    }
}

```

They are now attributes of the request object

This code will be called when the Request class is loaded by the class loader. It is done once for the class, not for each instance.

Calculate the postage here, using its own attributes.

Now, GetRequest.java can be simplified:

```

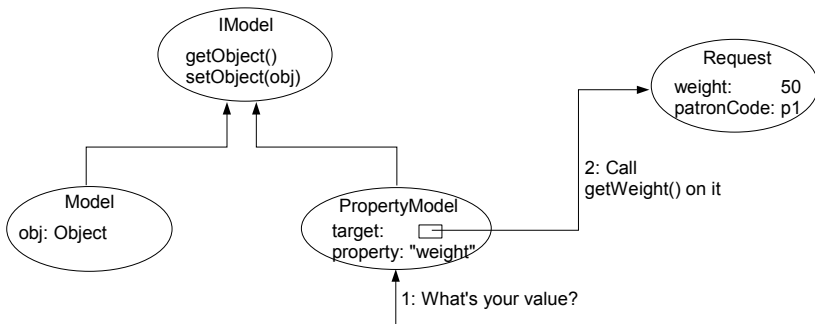
public class GetRequest extends WebPage {
    private Model weightModel = new Model();
    private Model patronCodeModel = new Model();
    private Map patronCodeToDiscount;

    public GetRequest() {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
        Form form = new Form("form") {
            protected void onSubmit() {
                Request request = new Request(
                    ((Integer) weightModel.getObject()).intValue(),
                    (String) patronCodeModel.getObject());
                int weight = ((Integer) weightModel.getObject()).intValue();
                Integer discount = (Integer) patronCodeToDiscount
                    .get(patronCodeModel.getObject());
                int postagePerKg = 10;
                int postage = weight * postagePerKg;
                if (discount != null) {
                    postage = postage * discount.intValue() / 100;
                }
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", weightModel, Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode", patronCodeModel);
        form.add(patronCode);
        add(form);
    }
}

```

Run the application and it should continue to work. Note that now you're

practically asking the user to edit the properties of a request object. In cases like this, you can use another kind of model called `PropertyModel` (see the diagram below). You have seen the `Model` class for the components. Actually, all the components in Wicket work with an `IModel` interface. It declares the `getObject()` and `setObject(obj)` methods but obviously as an interface, it has no implementation. The `Model` class implements the `IModel` interface and stores the object in itself. The `PropertyModel` class is another implementation of `IModel`. It has a `target` field pointing to another object (a `Request` object in your case) and a property name ("weight" in this case). When you call `getObject()` on it, it will call `getWeight()` on the `Request` object. Similarly, when you call `setObject(obj)`, it will call `setWeight()` on the `Request` object:



To implement this idea, modify `GetRequest.java`:

```

public class GetRequest extends WebPage {
    private Model weightModel = new Model();
    private Model patronCodeModel = new Model();
    private Request request = new Request(0, "");
}

public GetRequest() {
    Form form = new Form("form") {
        protected void onSubmit() {
            Request request = new Request(
                ((Integer)weightModel.getObject()).intValue(),
                (String)patronCodeModel.getObject());
            ShowPostage showPostage = new ShowPostage(request.getPostage());
            setResponsePage(showPostage);
        }
    };
    TextField weight = new TextField("weight",
        weightModel new PropertyModel(request, "weight"),
        Integer.class);
    form.add(weight);
    TextField patronCode = new TextField("patronCode",
        patronCodeModel new PropertyModel(request, "patronCode"));
    form.add(patronCode);
    add(form);
}
}

```

Annotations in the code:

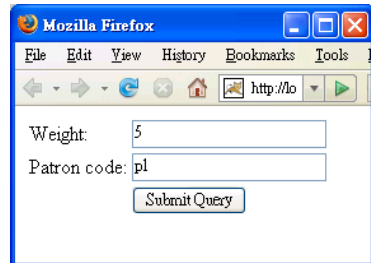
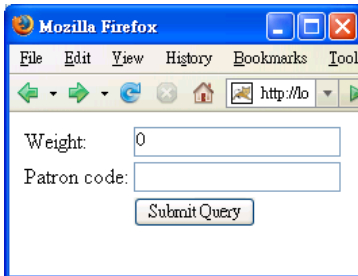
- Create a request object: points to `Request request = new Request(0, "");`
- Use a PropertyModel: points to `new PropertyModel(request, "weight")`
- The property name: points to `"weight"`
- The target object: points to `request`

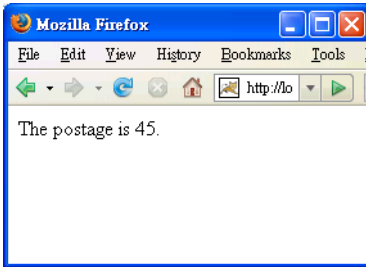
For this to work, you need to provide getters and setters for the `weight` and `patronCode` in the `Request` class (actually it is not strictly necessary but you're advised to do it):


```
public class Request {
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

    static {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
    }
    public Request(int weight, String patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }
    public String getPatronCode() {
        return patronCode;
    }
    public void setPatronCode(String patronCode) {
        this.patronCode = patronCode;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public int getPostage() {
        Integer discount = (Integer) patronCodeToDiscount.get(patronCode);
        int postagePerKg = 10;
        int postage = weight * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        return postage;
    }
}
```

Now, run it and it should continue to work (except that you'll see that the weight field will have 0 as the default):



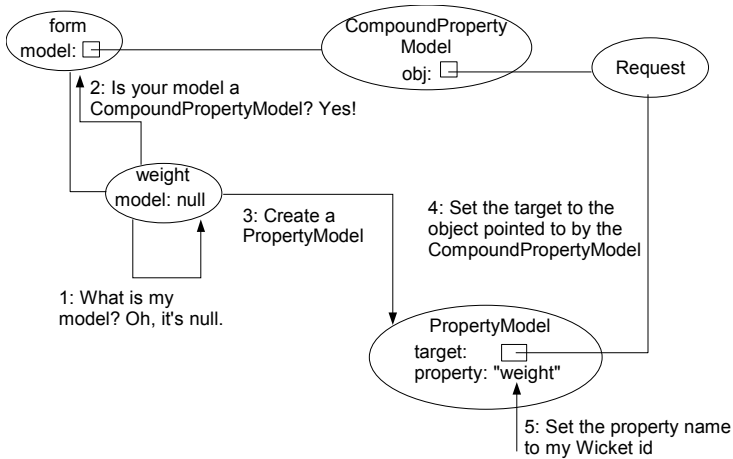


However, having to create a `PropertyModel` for each form component is still quite a lot of work. If for all the components, their Wicket ids are the same as the property names (which is the case here):

```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        Form form = new Form("form") {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight",
            new PropertyModel(request, "weight"),
            Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode",
            new PropertyModel(request, "patronCode"));
        form.add(patronCode);
        add(form);
    }
}
```

Then you can further simplify the code (see the diagram below): Instead of assigning a `PropertyModel` to each form component, you don't specify the model (so it is null). As a replacement, you assign a `CompoundPropertyModel` to the form itself. That `CompoundPropertyModel` in turn points to the `Request` object. When a component such as the weight text field needs to access its model but find that it's null, it will look for a `CompoundPropertyModel` in its parent (or further up). Here it will find the `CompoundPropertyModel` in the form. Then conceptually it will create a `PropertyModel` as its model, set the target to the object pointed to by the `CompoundPropertyModel` and set the property name to its Wicket id ("weight"):



To implement this idea, modify `GetRequest.java`:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", new PropertyModel(request,
"weight"),
        Integer.class);
        form.add(weight);
        TextField patronCode = new TextField("patronCode", new
PropertyModel(request, "patronCode"));
        form.add(patronCode);
        add(form);
    }
}
  
```

Assign a CompoundProperty Model to the form

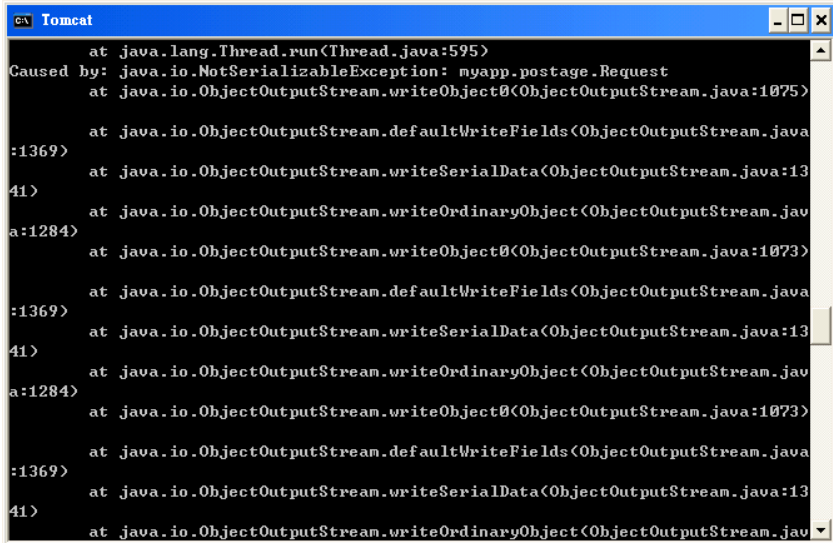
Let it point to the Request object

Do not specify a model

Now run it and it should continue to work.

Making sure the page is serializable

If you look at the Tomcat console, you should notice an exception when Tomcat is trying to serialize your `GetRequest` page:



```

Tomcat
Caused by: java.io.NotSerializableException: myapp.postage.Request
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1075)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java
:1369)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:13
41)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.jav
a:1284)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java
:1369)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:13
41)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.jav
a:1284)
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1073)
    at java.io.ObjectOutputStream.defaultWriteFields(ObjectOutputStream.java
:1369)
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:13
41)
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.jav

```

This is because the `GetRequest` page contains a `Request` object but the `Request` object is not implementing `Serializable`. This prevents the `GetRequest` page from being serialized. To solve the problem:

```

public class Request implements Serializable {
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

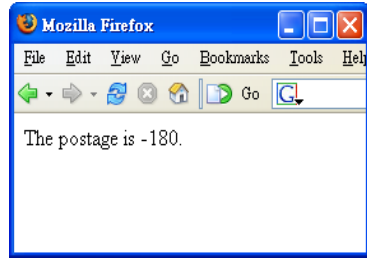
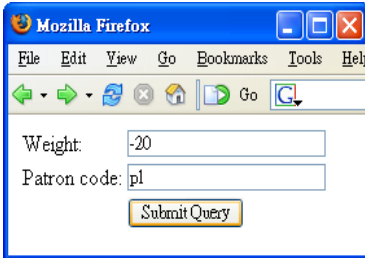
    static {
        patronCodeToDiscount = new HashMap();
        patronCodeToDiscount.put("p1", new Integer(90));
        patronCodeToDiscount.put("p2", new Integer(95));
    }
    public Request(int weight, String patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }
    public String getPatronCode() {
        return patronCode;
    }
    public void setPatronCode(String patronCode) {
        this.patronCode = patronCode;
    }
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
    public int getPostage() {
        Integer discount = (Integer) patronCodeToDiscount.get(patronCode);
        int postagePerKg = 10;
        int postage = weight * postagePerKg;
        if (discount != null) {
            postage = postage * discount.intValue() / 100;
        }
        return postage;
    }
}

```

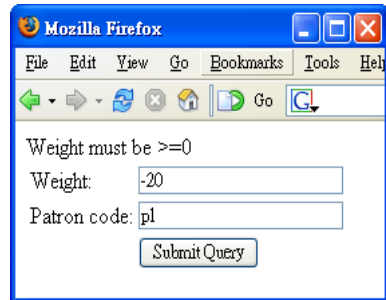
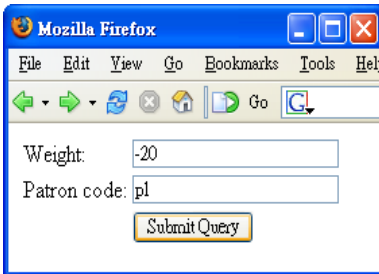
Then you shouldn't see that exception again.

What if the input is invalid?

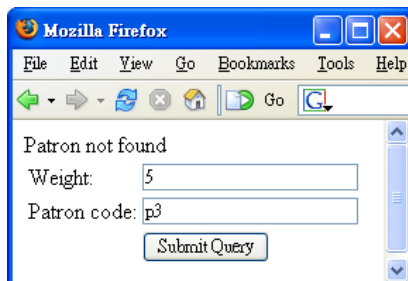
At the moment if the user enters a negative number as the weight (e.g., -20), it will go ahead and return a negative postage:



This is no good. Instead, you'd like the application to tell the user that the weight is invalid:

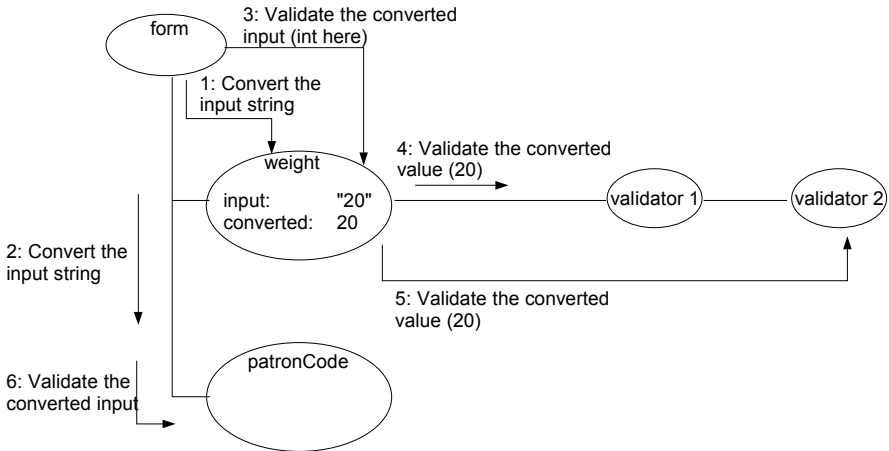


Similarly, it should also check if the patron code is valid or not. For example, if the user enters "p3", it should tell him that this code is not found:



Note that as the patron code is optional, if he doesn't enter anything, it should NOT be treated as an error. In order to validate the user input, you can add one or more validator objects to each form component (see the diagram below). When the form is submitted, the form will ask each form component to convert

the input string into the appropriate type (e.g., for the weight text field, the converted input is an int). Then it will ask each form component to validate itself. Suppose the weight text field has two validator objects. It will ask each one in turn to validate the converted int value. If the type conversion fails (e.g., user entered "abc" for the weight) or a validator fails, an error message will be logged and no further processing will occur on that form component:



Now let's do it. Modify `GetRequest.java`:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback")); // Need a FeedbackPanel to display the error message
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.add(new NumberValidator.MinimumValidator(0));
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}
  
```

Create a `MinimumValidator` object and then add it to the text field. This class is defined inside the `NumberValidator` class so the syntax is a bit weird.

0 here is the minimum value. The `MinimumValidator` object will check if the type-converted input value (int) is at least 0. Otherwise it will log an error message.

Modify `GetRequest.html` to add the `FeedbackPanel`:

```

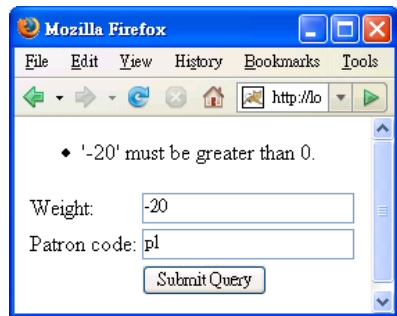
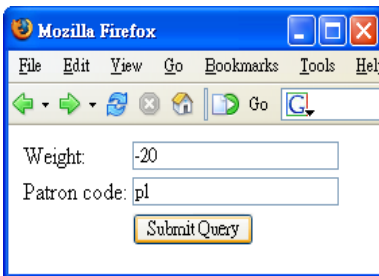
<html>
<span wicket:id="feedback"/>
<form wicket:id="form">
<table>
<tr>
<td>Weight:</td>
<td><input type="text" wicket:id="weight"/></td>
  
```

```

</tr>
<tr>
  <td>Patron code:</td>
  <td><input type="text" wicket:id="patronCode"/></td>
</tr>
<tr>
  <td></td>
  <td><input type="submit"/></td>
</tr>
</table>
</form>
</html>

```

Now run the application again and it should work:



At the moment you're explicitly creating a `MinimumValidator` object yourself. In fact, there is a shortcut:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.add(new NumberValidator.MinimumValidator(0));
        weight.add(NumberValidator.minimum(0));
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}

```

This `minimum()` method does exactly the same thing. It simply hides the `MinimumValidator` class from you, i.e., you don't know what is the class of the validator object. All you know is that it will check to make sure the input integer is \geq the minimum value specified. There is an overloaded `minimum()` method that accepts a double value, which will create an appropriate validator object to check double values.

Again, you can customize the error message by creating `GetRequest.properties`:

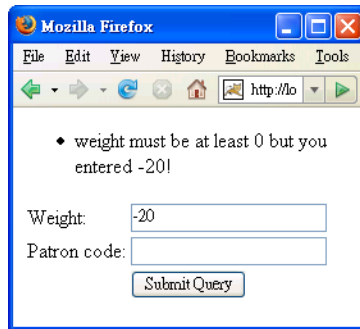
```

form.weight.NumberValidator.minimum=${label} must be at least ${minimum} \
but you entered ${input}!

```

The label of the field ("weight" in this case)
 The minimum value stored in the validator (0 in this case). Actually, toString() will be called on it to convert it to a string.
 Id path to the form component
 Resource key
 The value (string) entered by the user
 If you need to enter multiple lines as the value for a key, you can enter a backslash to tell it to continue with the next line. Do NOT enter anything after the backslash!

Make sure the application is reloaded. Then run it and it should work:



In addition to this validator, there are other similar ones making sure that the input is not larger than a maximum value or is in a certain range. Here is a summary:

<i>Purpose</i>	<i>Sample code</i>	<i>Resource key</i>
Make sure that the input number is ≥ 10	<code>NumberValidator.minimum(10)</code>	<code>NumberValidator.minimum</code>
Make sure that the input number is ≤ 10	<code>NumberValidator.maximum(10)</code>	<code>NumberValidator.maximum</code>
Make sure that the input number is in the range of 10-20 (inclusive)	<code>NumberValidator.range(10, 20)</code>	<code>NumberValidator.range</code>

These are for numbers. There are similar ones for strings and dates:

```

StringValidator.minLength(10); //Resource key is StringValidator.minimum
StringValidator.maxLength(10); //Resource key is StringValidator.maximum
StringValidator.lengthBetween(10, 20); //Resource key is StringValidator.range
DateValidator.minimum(...); //Resource key is DateValidator.minimum
DateValidator.maximum(...); //Resource key is DateValidator.maximum
DateValidator.range(...); //Resource key is DateValidator.range

```

Because it is very common to call `minimum()` with a 0 value, `NumberValidator`

has a static validator object for use:

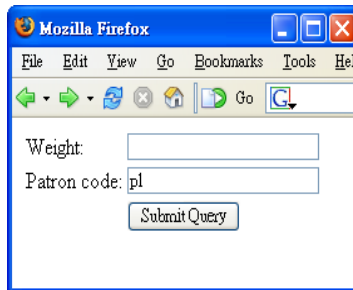
```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
weight.add(NumberValidator.minimum(0));
weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}
```

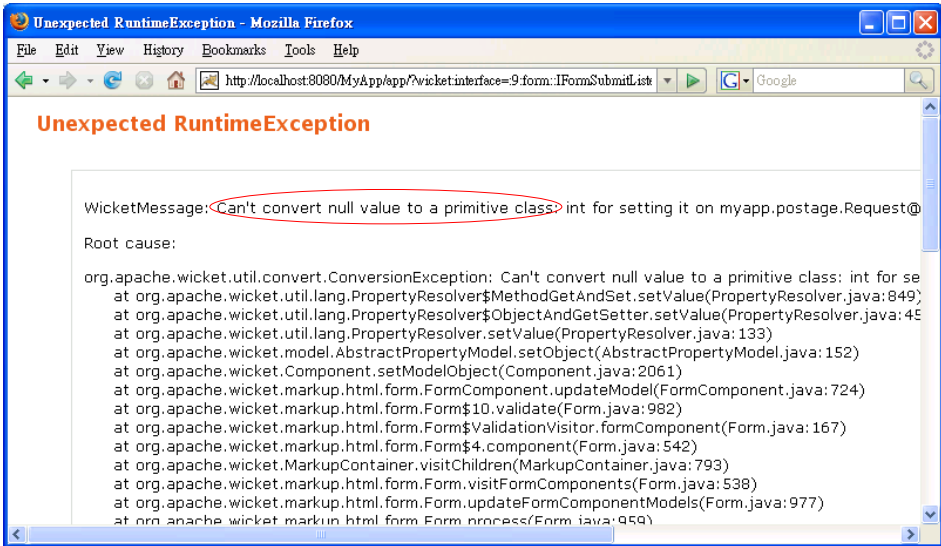
Now run it and it should continue to work.

Null input and validators

What if the user doesn't input anything as the weight? As mentioned in the previous chapter, the text field will treat it as null. How will the minimum validator handle this null value? It will let it pass and treat it as valid. Why? This design is to allow the case when some input is optional, but if the user does provide some input, then it must be validated. Here in this case, if you enter an empty string as the weight:



The application will throw an exception because the property model can't store a null into an int property (It could if it was an Integer property):

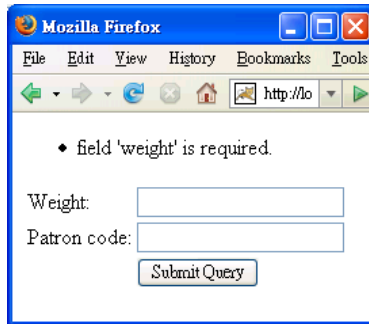


In this case, you can simply solve the problem by marking the weight as required:

```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}
```

Then the user will see:



Validating the patron code

Now the weight field is working fine. How to validate the patron code? There is no built-in validator suitable, so you can validate it in `onSubmit()`:

```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");
    private TextField patronCode;

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                if (!request.isPatronCodeValid()) {
                    patronCode.error("Patron code is invalid");
                } else {
                    ShowPostage showPostage = new ShowPostage(request
                        .getPostage());
                    setResponsePage(showPostage);
                }
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        form.add(patronCode);
        add(form);
    }
}
```

Log an error message for this component

Session for client 1

Message list

Message	Reporter
Patron code is invalid	

"patronCode"
TextField

Define the `isPatronCodeValid()` method in the Request class:

```
public class Request {
    private int weight;
    private String patronCode;
    private static Map patronCodeToDiscount;

    static {
```

```

patronCodeToDiscount = new HashMap();
patronCodeToDiscount.put("p1", new Integer(90));
patronCodeToDiscount.put("p2", new Integer(95));
}
...
public boolean isPatronCodeValid() {
    return patronCode == null
        || patronCodeToDiscount.containsKey(patronCode);
}
}

```

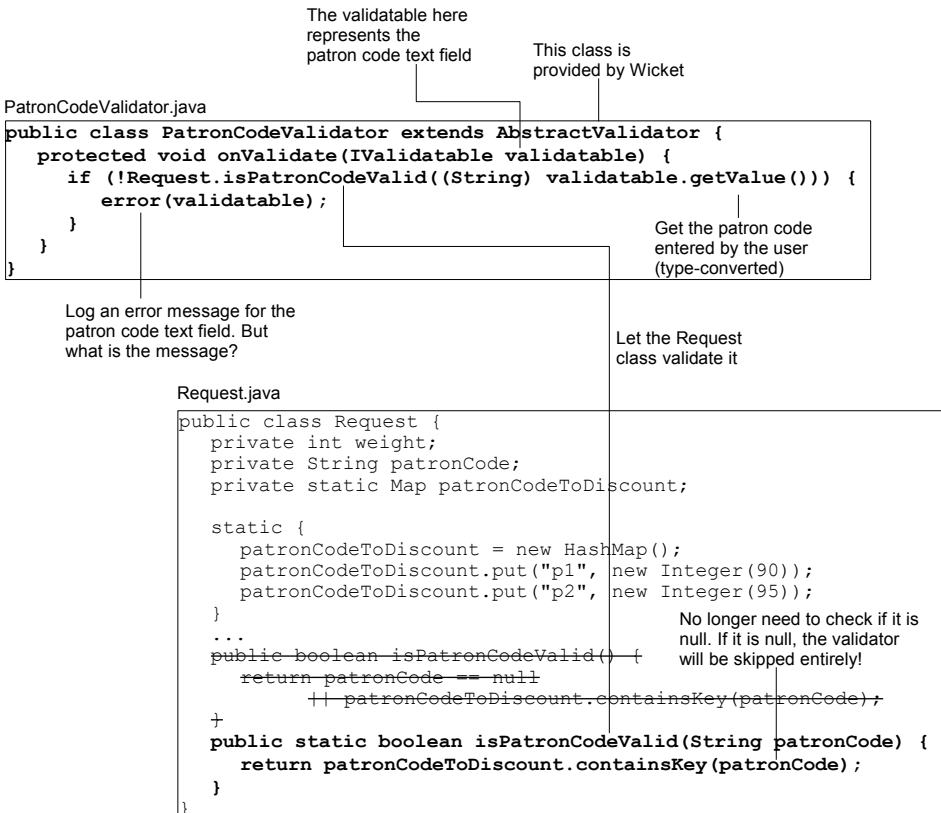
What if you needed to input the patron code on different pages? Then you would duplicate the validation code below in each page:

```

protected void onSubmit() {
    if (!request.isPatronCodeValid()) {
        patronCode.error("Patron code is invalid");
    } else {
        ShowPostage showPostage = new ShowPostage(request
            .getPostage());
        setResponsePage(showPostage);
    }
}
}

```

In that case you should extract the code into a custom validator. For example, create a PatronValidator class:



What is the error message? Just like all built-in validators, it will load the error

message using a resource key. By default, it will use the class name of the validator as the resource key. So, modify `GetRequest.properties`:

```
form.weight.NumberValidator.minimum=${label} must be at least ${minimum} \
but you entered ${input}!
form.patronCode.PatronCodeValidator=Could not find patron: ${input}!
```

Use the validator in `GetRequest.java`:

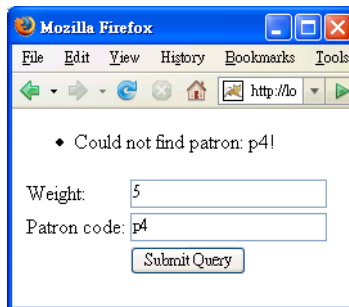
```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");
    private TextField patronCode;

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                if (!request.isPatronCodeValid()) {
                    weight.error("Patron code is invalid");
                } else {
                    ShowPostage showPostage = new ShowPostage(request.getPostage());
                    setResponsePage(showPostage);
                }
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
    }
}
```

_____ No longer need this

Use the validator here

Run it and it should work:



Displaying the error messages in red

Suppose that you'd like the error messages to be in red. To do that, view the HTML code generated by the Feedback Panel:

```
<ul wicket:id="feedbackul">
  <li wicket:id="messages" class="feedbackPanelERROR">
    ...
  </li>
  <li wicket:id="messages" class="feedbackPanelERROR">
    ...
  </li>
</ul>
```

```
</li>
</ul>
```

To make the `` elements appear in red, all you need is to modify `GetRequest.html`:

```
<html>
<head>
  <style type="text/css">
    li.feedbackPanelERROR { color: red }
  </style>
</head>
<body>
<span wicket:id="feedback"/>
<form wicket:id="form">
<table>
  <tr>
    <td>Weight:</td>
    <td><input type="text" wicket:id="weight"/></td>
  </tr>
  <tr>
    <td>Patron code:</td>
    <td><input type="text" wicket:id="patronCode"/></td>
  </tr>
  <tr>
    <td></td>
    <td><input type="submit"/></td>
  </tr>
</table>
</form>
</body>
</html>
```

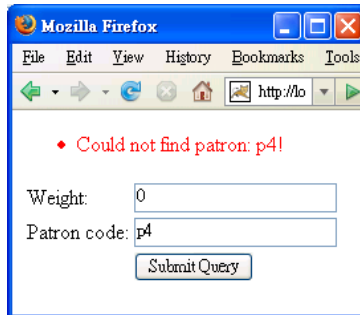
Define some styles

These styles are called "CSS styles". CSS stands for cascading style sheet.

Set the color of the list items (``) to red

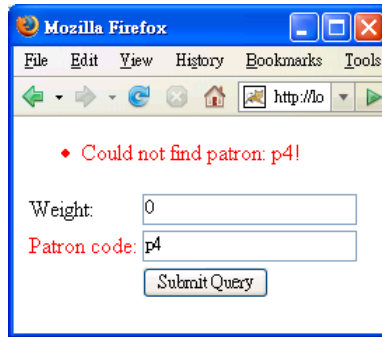
The following style will be applied to only those `` elements whose "class" attributes have the value of "feedbackPanelERROR"

Now run it and it will work:



Displaying invalid fields in red

Suppose that you'd like to display the invalid fields in red:



To do that, modify GetRequest.html:

```

<html>
<head>
  <style type="text/css">
    li.feedbackPanelERROR { color: red }
    td.invalidField { color: red }
  </style>
</head>
<body>
<span wicket:id="feedback"/>
<form wicket:id="form">
<table>
  <tr>
    <td wicket:id="weightLabel">Weight:</td>
    <td><input type="text" wicket:id="weight"/></td>
  </tr>
  <tr>
    <td wicket:id="patronCodeLabel">Patron code:</td>
    <td><input type="text" wicket:id="patronCode"/></td>
  </tr>
  <tr>
    <td></td>
    <td><input type="submit"/></td>
  </tr>
</table>
</form>
</body>
</html>

```

This style class is defined there

<td class="invalidField">...

Make it a Wicket component. You will use that component to add a "class" attribute to the <td> tag if the weight text field is invalid:

Define the components in GetRequest.java:

```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");
    private TextField weight;
    private TextField patronCode;

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        WebMarkupContainer weightLabel = new WebMarkupContainer("weightLabel") {
            protected void onComponentTag(ComponentTag tag) {
                if (!weight.isValid()) {
                    tag.put("class", "invalidField");
                }
                super.onComponentTag(tag);
            }
        };
        form.add(weightLabel);
        WebMarkupContainer patronCodeLabel =
            new WebMarkupContainer("patronCodeLabel") {
                protected void onComponentTag(ComponentTag tag) {
                    if (!patronCode.isValid()) {
                        tag.put("class", "invalidField");
                    }
                    super.onComponentTag(tag);
                }
            };
        form.add(patronCodeLabel);
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
    }
}

```

A WebMarkupContainer is a component that basically does nothing special: It outputs the start tag, the body (which could include components) and the end tag. Usually you'll use it to add attributes to the start tag.

Process the start tag

isValid() will check if there is an error message reported by the weight component:

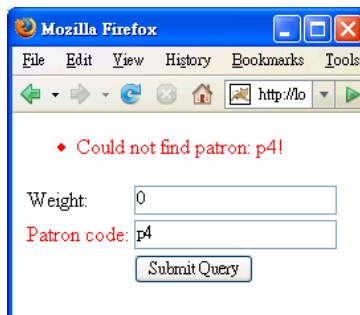
Add the "class" attribute to the tag

This will output the tag

"weight" TextField

Message	Reporter
Some error message..	

Now run it and it should work:



Creating a feedback label component

Note that the code for the weightLabel and that for the patronCodeLabel is very

much similar. When you see such duplicate code, you should consider putting the code into a component. Here, create a `FeedbackLabel` component:

```
public class FeedbackLabel extends WebMarkupContainer {
    private FormComponent subject;

    public FeedbackLabel(String id, FormComponent subject) {
        super(id);
        this.subject = subject;
    }
    protected void onComponentTag(ComponentTag tag) {
        if (!subject.isValid()) {
            tag.put("class", "invalidField");
        }
        super.onComponentTag(tag);
    }
}
```

Use it in `GetRequest.java`:

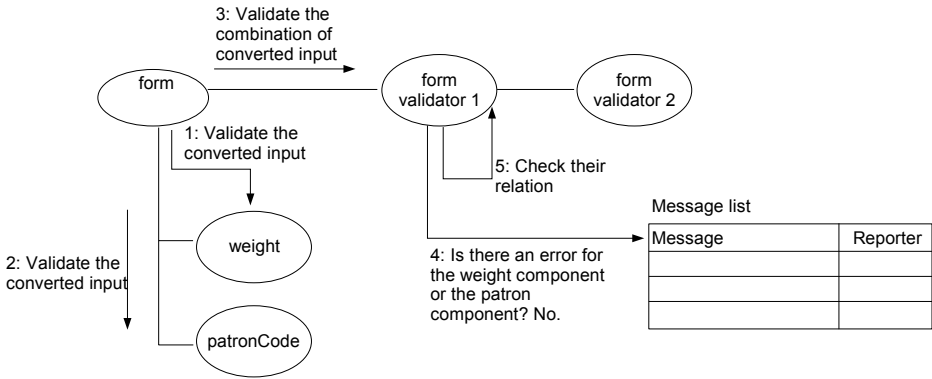
```
public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        FeedbackLabel weightLabel = new FeedbackLabel("weightLabel", weight);
        form.add(weightLabel);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
        FeedbackLabel patronCodeLabel = new FeedbackLabel("patronCodeLabel",
            patronCode);
        form.add(patronCodeLabel);
    }
}
```

Now run it and it should continue to work.

Validating a combination of multiple input values

Suppose that for a particular patron `p1`, you will never ship a package that is weighted more than 50kg. As this involves both the weight and the patron code (two form components), you can't make a validator and assign it to a single form component. In this case, you can make a "form validator" (see the diagram below). After asking each form component to validate itself using its own validators, the form will invoke its form validators one by one. Suppose the first form validator is involved with the weight component and the `patronCode` component, it will first check if they are valid so far, by checking if there are any error messages for any of them. If yes, it will not do anything. If no, it will go ahead to validate their combination:



To implement this idea, create a `LightValidator` class as a form validator:

```
public class LightValidator extends AbstractFormValidator {
    private TextField weight;
    private TextField patronCode;

    public LightValidator(TextField weight, TextField patronCode) {
        this.weight = weight;
        this.patronCode = patronCode;
    }

    public FormComponent[] getDependentFormComponents() {
        return new FormComponent[] { weight, patronCode };
    }

    public void validate(Form form) {
        String patronCodeEntered = (String) patronCode.getConvertedInput();
        if (patronCodeEntered != null) {
            if (patronCodeEntered.equals("p1")
                && ((Integer) weight.getConvertedInput()).intValue() > 50) {
                error(weight);
            }
        }
    }
}
```

This is a form validator

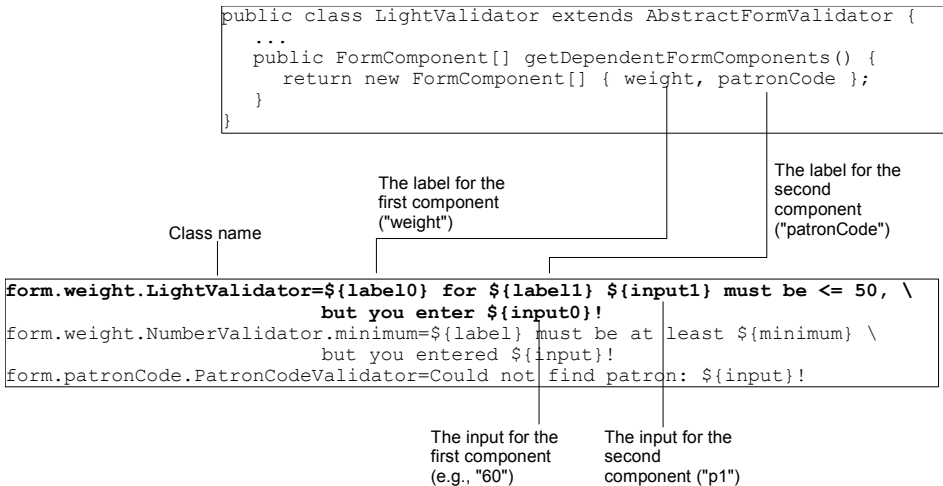
Tell `AbstractFormValidator` that this validator involves these two form components, so that it can check their individual validity.

It could be null. Validate only if it is p1. So ignore it if it is null.

Log an error message for the weight component. You could do it for the patronCode component instead. It's up to you but the weight seems to be more reasonable.

This method is called only after the two related form components are valid individually

Define the error message in `GetRequest.properties`:



Use this form validator in GetRequest.java:

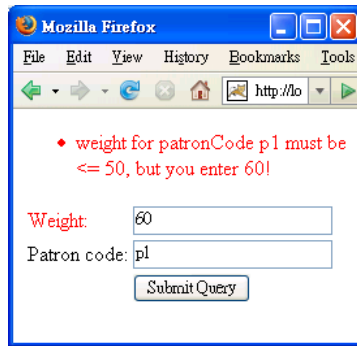
```

public class GetRequest extends WebPage {
    private Request request = new Request(0, "");

    public GetRequest() {
        add(new FeedbackPanel("feedback"));
        Form form = new Form("form", new CompoundPropertyModel(request)) {
            protected void onSubmit() {
                ShowPostage showPostage = new ShowPostage(request.getPostage());
                setResponsePage(showPostage);
            }
        };
        TextField weight = new TextField("weight", Integer.class);
        weight.setRequired(true);
        weight.add(NumberValidator.POSITIVE);
        form.add(weight);
        TextField patronCode = new TextField("patronCode");
        patronCode.add(new PatronCodeValidator());
        form.add(patronCode);
        add(form);
        FeedbackLabel weightLabel = new FeedbackLabel("weightLabel", weight);
        form.add(weightLabel);
        FeedbackLabel patronCodeLabel = new FeedbackLabel("patronCodeLabel",
            patronCode);
        form.add(patronCodeLabel);
        form.add(new LightValidator(weight, patronCode));
    }
}

```

Run the application and it should work:



Pattern validator

You have seen some validators checking the minimum, maximum or range of a certain value (number, string or date). Another useful one is the pattern validator. It checks if a string matches a "regular expression". For example, to check if the input string is a name, i.e., consisting of one or more letters (a-z) or digits:

```
new PatternValidator("\\w+");
```

in which `\\` means a single backslash. Then `\\w` means a word character (letter or digit) and `XXX+` means to expect `XXX` one or more times. To accept an empty name, change it to:

```
new PatternValidator("\\w*");
```

in which `XXX*` means to expect `XXX` zero or more times. If you'd like to check if the input string is a phone number like 123-4567:

```
new PatternValidator("\\d{3}-\\d{4}");
```

in which `\\d` means a digit and `XXX{3}` means to expect `XXX` three times.

Summary

If a form component is used to edit a property of an object, you can use a `PropertyModel`. If the Wicket id is the same as the property name, you can simply use a `CompoundPropertyModel` with the form and not set the model of the form component.

On form submission, the form will ask each form component to type convert the input. Then it will ask each one to validate the converted input. You can add one or more validators to a form component. They will be activated one by one. If the input is found to be invalid, the validator will log an error message for that form component. You can customize the error message using a properties file. If the input is null, the validators will be skipped to allow optional input. Finally, the form will invoke its form validators. They are useful for checking the combination of two or more form input values. A form validator will be activated only if the related form components have been found to be valid individually (no error message for them).

There are some built-in validators coming with Wicket to check if the value (number, string or date) is not less than a minimum, not greater than a maximum or in a range. There is also one checking if a string value matches a regular expression.

A `WebMarkupContainer` is a boring component. It will basically output what's in the template (it will render components in its body). It is commonly used when all you need to do is to modify the attributes of the start tag.

When you see duplicate code for different components, consider creating a new component class and put the code there.

