

WICKET IN ACTION

Martijn Dashorst
Eelco Hillenius

MEAP Unedited Draft

 MANNING



**MEAP Edition
Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

Table of Contents

Part 1: Getting started with Wicket

Chapter 1: What is Wicket?

Chapter 2: The architecture of Wicket

Chapter 3: Setting up a Wicket project

Chapter 4: Building a cheesy Wicket application

Part 2: Getting a basic grip on Wicket

Chapter 5: Understanding models

Chapter 6: Using basic components

Chapter 7: Using forms for data entry

Chapter 8: Composing your pages

Part 3: Advanced Wicket

Chapter 9: Creating custom components

Chapter 10: Creating rich components

Chapter 11: Authorization and authentication

Chapter 12: Working with resources

Chapter 13: Localization and internationalization

Chapter 14: Database driven applications

Chapter 15: Putting your Wicket application in production

Chapter 16: Component index

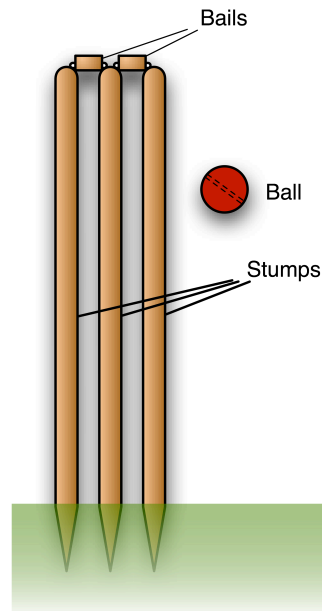
Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=328>

1

What is Wicket?

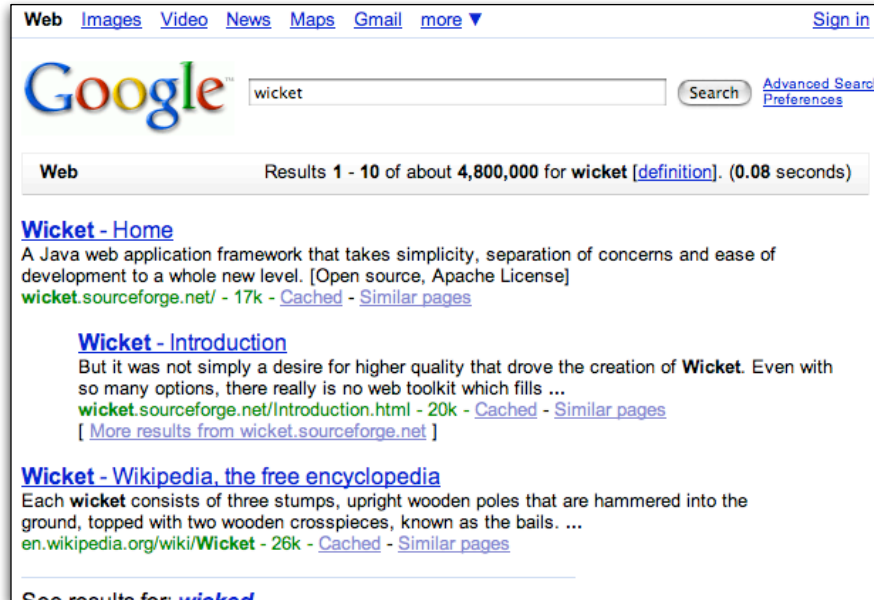
This is a question about 1 billion people will happily answer and *get it wrong!* What most of these people would answer is: cricket equipment (see figure 1.1 for a wicket in the cricket sense). Cricket is a bat-and-ball sport pretty much like baseball, but far more complicated to the untrained eye. Cricket is mostly popular in The United Kingdom and South Asia where it is by far the most popular sport in several countries including India and Pakistan.

Figure 1.1 A wicket: a set of three stumps topped by bails. This is not the topic of this book.



A keen Star Wars fan would say that Wicket is a furry creature called an Ewok from the planet moon Endor. A true Star Wars fan would also say that they were invented for merchandise purposes and that the movie could do very well without them, thank you very much. However, the Star Wars fan would also be proved wrong by his favorite search engine (see figure 1.2, showing a search for wicket on Google).

Figure 1.2 Google search results for 'wicket'. The number one result has nothing to do with cricket or with the furry and highly dangerous inhabitants of the planet moon Endor.



What google and other search engines list as their top result for the term wicket is not anything related to 22 men running after a red ball in white suits on a green field, nor a furry alien creature capable of slaying elite commandos of the Empire with sticks and stones. Instead they will find a website dedicated to a popular open source web application framework for the Java™ platform.

1.1 How we got here

Don't get us wrong. Cricket is probably a great sport once you understand the rules. However, in this book we will stick to something easier to grasp, and talk about the software framework that showed up as the first result.

But before going into the details of that, we would like to share the story of how we got involved in the first place.

1.1.1 A developer's tale

It was one of these projects.

The analysts in our development team entered the meeting area in a cheerful mood. They were still excited about the meeting they had with clients the evening before. Our team had been working on this web application for a few months, and we had been able to implement the first test version perfectly according to schedule. The clients were pleasantly surprised with that they could already play around with the software themselves. They were able to formulate their needs much better now that they could see it in action. The demo sparked a flurry of discussions, and the analysts left the meeting with a book full of notes on what was good and what should be

improved. So, the meeting with the development team next morning started with everyone in a great mood, with compliments and cheers flying across the room.

And then the book of notes was opened...

The clients, who had ratified the user interface designs before we started building, generally liked what we had produced so far. However, seeing the application in action, they also saw plenty of room for improvement. They wanted pagination and sorting in a list here. They wanted a wizard instead of a single form there. An extra row of tabs in that place. And a filter function there, there and there. All that time, the analysts were presenting the change requests with an air of pride. They were doing their part of making this an application everyone would be proud of!

The developers however were less amused. When I looked around the room I saw an expression of horror on the faces of every single one of them. Soon the first protests started.: "Why didn't you think about a wizard earlier!", "Do you have any idea how much work it will be to implement that extra row of tabs?", "We just implemented that feature and now you want us to throw that away?!". The developers were angry with the analysts for asking for these changes, and the analysts were angry with the developers for making such a big deal out of these what they thought were simple change requests.

Of course, the analysts were right for aiming for the best product, the best user experience. They didn't really know the technical implications of what they were proposing, and the changes they had in mind really didn't seem that outrageous on the surface. On the other hand, I had been developing web applications long enough to know that what looked simple at first sight would actually mean quite a bit of pain to go through. We would have to rewire - and often rewrite - the 'actions' and 'navigations' for our Struts-like framework and come up with new hacks to get the hard parts done. Introducing an extra row of tabs would mean rewriting about a quarter of all our the templates, and our Java IDEs wouldn't be helping much with that. Taking care of these changes was going to take weeks! It would cause a lot of frustration and would probably introduce a truckload of new bugs.

So there we were. Like I had experienced with many web application projects before, we arrived in maintenance hell well before the application even reached it's 1.0 version!

At about that same time, Martijn started working for the same company as I did. Prior to that, Martijn had mainly been building desktop applications using Delphi and C++, and the difficult parts he had to deal with were usually directly related to the 'business' problem he was trying to solve. Rarely was the user interface the development bottleneck. But now Martijn found himself in an upside-down world. Solving relatively simple business problems, like editing a couple of tables via a graphical user interface, suddenly proved to be big, tedious tasks. Building web applications sure wasn't as much fun as he thought it would be.

When Martijn and I talked about the kind of problems we encountered in our projects, It didn't take us long to figure out that one of the most urgent problems for us to solve was the way we developed the web tier. On the other tiers, we were using frameworks like Hibernate, Apache Lucene, jBPM, Quartz, JasperReports and a home grown Spring-like framework (this was before Spring was even released), and those frameworks served our purposes well. But maybe more importantly, in those other tiers we could just use (and improve!) our object oriented coding skills. We felt we were thrown back to procedural programming as soon as we had to do anything related to the web interface.

It was clear to us that if we would ever want to be able to develop web applications in a more productive yet maintainable fashion, we would need to do things differently. And not just a little bit different, like the 40+ Struts clones (model 2 or web mvc frameworks) do. No, that something would have to be a radical improvement.

We spent the next year looking into just about every framework we came across. Some frameworks, like Echo, JSF and Tapestry, came close to what we wanted, but they never made the click with us. I think we had already given up our search when one afternoon Johan Compagner, a freelancer we regularly worked with and often our savior when it came to fixing really hard issues, stormed into our office saying that he found the framework of our dreams.

It took the three of us less than an hour to decide that this framework, called Wicket and that didn't even had an alpha version out yet, would be the framework we were going to employ for future web application projects.

1.2 Wicket in a nutshell

It's not an official mission statement, but the next sentence is probably the most compact description of what Wicket is:

A Java software framework to enable component oriented, programmatic manipulation of markup.

This sentence captures the framework's fundamental goals and scope. It is not very precise, but some of the most important ideas driving Wicket are in there:

Manipulation of markup

Markup can be anything from XML to HTML to WML and SVG. When building web applications, you typically use (X)HTML to instruct the browser how to display what. For instance, surrounding a piece of text with <h3> tags instructs the browser to render that text using the 'Header 3' style. You use Wicket to manipulate those tags and their contents.

Programmatic manipulation

With Wicket, you maintain markup templates in separate (HTML) files and use the Java™ programming language to drive the dynamic parts of those templates. As we will see later, Wicket forces a strict separation of presentation and logic: clean HTML (or other markup) for presentation and plain Java™ (opposed to lots of XML or annotations) for user interface logic.

Component Oriented

When you design applications with Wicket, you'll break up bits of functionality in self contained user interface components. Components are encapsulated (they have their own state and behavior), typically have a well defined external API and are reusable by default. For example, a label prints text, a text field receives input and displays the current value of something and a link triggers actions. Programming with Wicket is much closer to desktop programming with, say, Swing or SWT than it is to most of the other Java™ web application frameworks, like Apache Struts, WebWork and Stripes.

(callout)

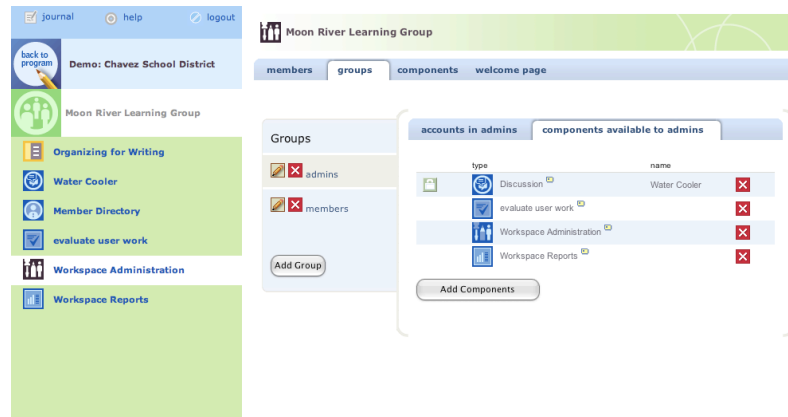
Apache Wicket or just Wicket?

Wicket started out as an open source project at SourceForge.net, a huge repository for open source projects. In June 2007 the Wicket project joined the Apache Software Foundation as a top level project (<http://wicket.apache.org>) and is since known as 'Apache Wicket'. However, that is a mouthful and quite straining to the eyes to read every time we mention the project's name. So we use Wicket instead of the more accurate and official Apache Wicket.

(end callout)

While Wicket is well suited for the development of any kind of web site, the framework is particularly well suited for developing 'web applications'. Web applications are sites, like the example shown in figure 1.3, with a complex, layered user interface employing things like tabs and wizards, up to 'rich' controls like sliders, date pickers, drag and drop functionality and more. They are typically built to enable users to perform certain tasks whereas document oriented web sites just expose information.

Figure 1.3 An example of a 'web application'. It shows a screen of Teachscape, a learning management application that is used by universities throughout the United States.



Browser based applications have been very popular in the enterprise for a while now. Using web applications, businesses have been able to reduce maintenance cost, they found more ways to cooperate both internally and externally, and they brought their customers closer to their businesses than ever.

Unfortunately, the technical transition from a model where applications run locally (desktop applications) has been quite bumpy. Just coming out of the era of character based terminal applications, people were just getting used to the fancy user interfaces that are possible on Personal Computers (PCs). Compared to terminal applications, desktop applications can display much more information at once and in different graphical formats, giving users a much better sense of context and enabling them to work more efficiently. But when the first web applications were introduced, users often had to go back to a much simpler interface, losing many of the efficiency boosters they were starting to like so much. Why was this? Was it the browser? Or amateurism from the first web application builders?

1.2.1 Problems with the stateless protocol

The main reason why web applications were often a step back compared to their desktop application counterparts is that it is very hard to create web applications that behave like desktop applications. And the primary reason for that being so difficult is the fact that web pages are served over a stateless protocol: the HyperText Transfer Protocol (HTTP).

The HTTP protocol is called a 'stateless' protocol because it doesn't facilitate remembering what you were doing before in a 'conversation'. To illustrate what this means, let's go back to the search we performed on Google for the term 'wicket'. If you scroll down the results you'll see that not all of the results are displayed. At the bottom, there is a bar of links that will let you scroll through the results. These links look like this:

<http://www.google.com/search?q=wicket&start=10>

Even though you just searched for Wicket, it is still there in the URL. By providing this information in the URL, Google knows everything it needs to know to display the second page of our search query whatever server the request is sent to. And that is great for scalability as the first request might be handled by a server in San Francisco, and the second request by a server in Seattle. Without providing the search argument again, all requests in the same conversation would have to be handled by the same server.

We can define a conversation or session as the time-span that a user is active on a system.

The user opens the application, works on it for a while, and at the end logs off (or closes the window), after which the session will 'expire'. The next time the user logs on, a new session is started.

There are some problems with this REST approach to developing web applications (REST stands for Representational State Transfer and implicates encoding all the necessary information in requests).

First and foremost it scales very bad for complexity of the user interface. Say you have a function for searching for a product in your application. The results are displayed in pages of ten, and you can sort on name, item number or brand. Clicking on one of the results takes you to a detail page of that product with a description and a form to add the product to your shopping cart. It also has a tab for displaying customer reviews, which includes a wizard for adding your own review. And a tab that shows when, how many and for what price you bought this product before. You probably feel it coming: you'll have to pass the search argument, page number and sorting information with every request as you are working on the details. You'll have to read the parameters in with every request you handle, and make sure you write the relevant bits out again with every link or form in your application. Compare this to what you know about Object Oriented programming (encapsulation et-cetera)... what an incredible leak of scope! Imagine adding another filter option to the search; you would have to go through all the possible details paths and add the new parameter. Or embedding this search in a tabbed panel. You would have to add the parameter for the selected tab with every request as well. Maintaining an application with a user interface of even a medium complexity gets very, very painful this way.

Secondly, you'll have to be aware of the security implications of passing state via the URL. Your URLs will be visible in your web browser location bar and if you are not working over a secure connection, they can be intercepted by others. This may or may not be a problem for you,

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=328>

but when it is, it means you'll have to do some extra work making your application secure. Say for instance that you are authorized to view only part of the product database. With the product ids being passed around in the URLs what stops you from just trying a code or writing down one you saw your colleague from department Z use (you noticed the id in the URL when getting some coffee)? You'll have to explicitly check whether the user is allowed to see a certain result. This can mean quite a lot of work and chances are you forget something.

A third problem is that HTTP works with strings. REST works fine when passing search arguments, but what if you just loaded a product object? You obviously can't encode the Java object in the URLs, so you'll have to pass a representation of it (the R in REST, which in this case would mean a product id). Great, are we coding in Java, now we have to have string representations for every object we consider 'state' in our application.

The last problem is minor but annoying nevertheless. What you send in URLs is basically a flat map of key/ value pairs. So you have to watch out the keys don't conflict. If you decide to have two rows of tabs in your application, you would have to name the argument for the first row differently than the second row (for instance, selectedTab1 and selectedTab2).

You can get rid of all of these problems by using Wicket. It is a stateful framework, so you don't have to follow the REST (though you can, but we will talk about that later in this book) approach. The main idea behind REST is scalability. Fine. But let me make a bold statement here:

Very often, REST is premature optimization.

Many programmers, maybe the majority, are writing applications that are meant for a couple of tens of thousand users concurrently at most. For such projects, it is often more efficient to spend money on more server memory and maybe an extra node in the server cluster than spending money on an extra developer because the application needs to be RESTful. If you are not developing an application for millions of concurrent users (like Google), why would you be worried about spending 50kb per live user session (so 500MB enables you to serve 10,000 concurrent sessions, which you probably won't be serving from one machine anyway).

The real edge of Wicket and the ultimate reason why it was started, is the programming model. Wicket provides a programming model that is as simple as possible without being needlessly limiting. The programming model can be summarized as:

With Wicket, you program in just Java™ and just HTML using meaningful abstractions.

Let's investigate this statement a little bit further.

1.2.2 Just Java™

The 'Just Java' bit of the programming model consists of two pillars:

- You decide how components are created, assembled and combined and - to some extent - what their life cycle looks like.
- The state of components is managed for you.

Wicket puts you in charge of how components are created. And because you can instantiate components however you want, you have the freedom to design your component classes in any way - with any contract - you want.

Almost all the other Java web frameworks around today provide their users with a declarative programming model. The idea of declarative programming is that building an application should just be a matter of configuring the already existing building blocks and patterns, and the actual coding that is required should be kept minimal. The framework is responsible for managing object instances (like components or 'actions'), and combining all the different bits (like validation) that make the framework. Most frameworks also make the assumption that functionality is organized in pages, and that between page instances, there is a 'page flow'.

Not too long ago (before Java 5 got traction), most frameworks had their users use XML to configure page flow, declare authorization, and for instance configure validation checks for forms. Nowadays, many frameworks use annotations for such tasks, up to the point that annotations starts to function as the 'new XML'. Some frameworks support convention based configuration, which means that you can instruct the framework to behave in a certain way by adhering certain coding conventions, like how things are named and where they are placed, and it also typically means the framework will default to the most sensible settings if no 'configuration' is available for a certain task.

Configuration by convention is used in Wicket quite a bit. For instance, HTML templates are automatically loaded if they sit next to the component class with the same name. But what really sets Wicket apart is that it does not provide a declarative programming model, but rather puts it's users in charge of how components are created and configured, which is done using plain old Java.

To illustrate this, with Wicket you would never do:

```
<command name="echo">
  <controller class="com.foo.SomeAction" />
  <view name="success" path="/someView.jsp" />
</command>
```

which is what a piece of Maverick XML configuration would look like (and Maverick is very similar to for instance Apache Struts or Spring MVC). Rather, your code would look like:

```
add(new Link("foo") {
    public void onClick() {
        setResponsePage(new SomePage(someArgument));
    }
});
```

The fact that you can decide how your components are created gives you an unmatched level of flexibility. You can code your pages and components in such a fashion that they require certain arguments to be passed in, components can have their own initialization (in 'web MVC' frameworks like Apache Struts such things would have to be accomplished by 'chaining' commands) and dependencies can be static (compile safe) rather than string based.

Take this component for instance:

```
public class EditBarLink extends Link {
    private final Person person;

    public EditBarLink(String id, Person person) {
```

```

        super(id);
        this.person = person;
    }

    public void onClick() {
        setResponsePage(new EditPersonPage(person));
    }
}

```

This above code fragment shows the code for a custom component that forces its users to create it with a Person instance. As a writer of that component, you don't have to care in what context it is used, as you know you will have a person available when you need it.

Another thing to notice is that the link is created before the `onClick` method is called. Of course, this is because for a user to be able to click on a link, he or she must see one first. Look at the example again, and notice that `onClick` uses a member variable (`person`). This may look like a very straightforward thing: you create an object, set a member variable, and expect to use it later in exactly the same state you left it behind. Crazy as it may sound, this is not something that is widely supported by frameworks, even not by the component oriented ones. Java Server Faces, Apache Tapestry and even Microsoft's ASP.NET have some facilities to manage state, but in an explicit and disjointed fashion (on top of the fact that those frameworks do the component instantiation).

The advantages of Wicket managing your state are numerous. You can quite easily implement component interdependencies, like a page to go back to when an action is done, things like pageable search results and wizards and in general makes that any component assembly works without Wicket having to understand it. And there are the arguments of the previous section against the stateless protocol.

1.2.3 Just HTML

Not everything is done in Java though. What is considered pure (non-dynamic) presentation is defined in HTML templates. And here we arrive at another thing that sets Wicket apart from most frameworks: it forces its users to use what we call 'clean templates'. This means so much as that the HTML templates you use with Wicket only contain static presentation code (markup) and placeholders where Java components are hooked in. There is never any logic in templates.

For instance, you will never see a fragment like:

```

<tr>
  <c:forEach var="item" items="{sessionScope.list}">
    <td>
      <c:out value="item.name" />
    </td>
  </c:forEach>
</tr>

```

which is how a Java Server Page would look, or like:

```

<tr>
  #foreach ($item in $sessionScope.list)
  <td>

```

```

        ${item.name}
    </td>
#end
</tr>

```

which is how an Apache Velocity template would look, but rather:

```

<tr>
  <td wicket:id="list">
    <span wicket:id="name" />
  </td>
</tr>

```

If you are used to the first fragment (which is from a Java Server Pages page or JSP using the JSTL tag library), the way you code with Wicket may be annoying at first. With JSPs you just have to make sure the context (page-, request- and session attributes) is properly 'filled' with the objects you need in your page, and you can add loops, conditional parts, et-cetera to your JSP without ever having to go back to the Java code.

In contrast, with Wicket you pretty much have to know the structure of your page upfront. In the above example a list view component would have to be added to the page with id 'list' and for every row of the list view, there has to be a child component with id 'name'.

The way you code JSPs sounds a lot easier doesn't it? Why did Wicket choose that rigid separation between presentation and logic? After using the JSP style of development (including WebMacro, Apache Velocity and Freemarker) for many commercial projects, we - Wicket's developers - believe mixing logic and presentation like you do with scripting in templates has the following problems:

- Scripting in templates often results in 'spaghetti code'. The above example looks fine, but often you want to do much more complex things. It often gets incredibly verbose and hard to distinguish between pieces of logic and pieces of normal HTML, so that the whole template gets really hard to read (and thus to maintain).
- If you code logic with scripts, you won't have the compiler to help you out with things like refactoring, navigating the logic, and avoiding stupid little things like syntax errors.
- It will be harder to work with 'designers'. If you work with separate web designers (like I do), they'll have a hard time figuring out JSP-like templates. Rather than just staying focussed on their job - the presentation of the application - they now have to understand at least the basics of the scripting that the templating engine supports, including things like tag libraries, magical objects (like Velocity tools when you use that) et-cetera. They often cannot use their tools of choice and there is a difference between the mockups they would normally deliver and the templates with the logic in it.

With Just Java and Just HTML we are half way to having a good framework. The last part that makes a framework good is to have and facilitate the proper abstractions for a domain.

1.2.4 The right abstractions

Wicket's typical domain is the user interface running in a web browser. As such, it has abstractions for all the widgets you can see in a typical web page like links, drop down lists and

text fields. The framework also provides abstractions that are not directly visible, but that make sense in this context: applications, sessions, pages, validators, converters, et-cetera. Having these abstractions will ensure you will be able to discover your way around Wicket fairly easy, and will help you put together your application in a sensible way.

Since Wicket enables you to write truly self contained components, you can implement the proper abstractions you need for your application yourself. You can have a 'SearchForProductPanel', or a 'UserSelector' or 'CustomerNameLabel'. Whatever works for you. Remember that one of the important innovations behind Object Orientation was that you could create abstractions from real world things and model those abstractions with data and behavior.

Now that you have a fair idea of what Wicket stands for in theory, it's time to take a look at what it looks like when you would use it.

1.3 Have a quick bite of Wicket

There is nothing like seeing some code when you want to get an idea about a framework. We won't get into too much detail in this chapter, as we will have ample opportunity to do that later on in this book. If things are unclear in this part, don't worry. We will discuss everything in more depth in the rest of the book and more.

In the examples in this section, and throughout this book you will encounter some Java features you may not be familiar with: anonymous subclassing is something you will see a lot. It is a way to quickly extend a class and provide it with your specific behavior. We will use this idiom a lot in this book, as it makes the examples more concise. In chapter XXX we will give some best practices on how to use this idiom to your benefit and provide some nice alternatives.

We also use one Java 5 annotation quite a lot: `@override`. This annotation exists to help you and the Java compiler: it gives the compiler a signal that you intent to override that specific method. If there is no such overridable method in the super hierarchy, the compiler will generate an error. This is much more preferable than having to figure out why your program doesn't call your method (depending on the amount of coffee, this could take hours).

As we need a starting point for showing off Wicket, the obligatory Hello World example seems like a good way to start: how can there be a book on programming without that!

1.3.1 Hello, uhm... World

The first example will introduce you to the very foundations of each Wicket application: HTML markup and Java classes. In this example we want to display the famous text: 'Hello, World!' in a browser, and have the text delivered to us by Wicket. Figure 1.4 shows a screenshot of a browser window displaying the message.

Figure 1.4 The 'Hello, World!' example as rendered in a browser window.



In a Wicket application, each page consists of an HTML markup file and an associated Java class. Both files should reside in the same package folder (how you can customize this is explained in chapter 10):

```
src/wicket/in/action/chapter01/HelloWorld.java
src/wicket/in/action/chapter01/HelloWorld.html
```

Creating a 'Hello, World!' page in static html would look like the following markup file:

```
<html>
<body>
<h1>[text goes here]</h1>    #1
</body>
</html>
```

(Annotation) <#1 Dynamic part>

If you look closely at the markup, the part that we want to make dynamic is enclosed between the open and closing h1 tags. But first things first. We need to create a class for the page: the HelloWorld class (in HelloWorld.java).

```
package wicket.in.action.chapter01;

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

public class HelloWorld extends WebPage {
    public HelloWorld() {
    }
}
```

This is the most basic web page you can build using Wicket: only markup, with no components on the page. When building web applications, this is usually a good starting point.

(callout)

In this example we showed imports and package names. These are typically not a very interesting read in programming books, so we will omit them in the other examples. Use your IDE's auto-import features to get the desired import for your Wicket class.

If you use the PDF version of this book and want to copy-paste the example code, you can use the organize import facilities of your IDE to fix the imports in one go.

Be sure to pick the Wicket components, as there is an overlap between components available from Swing and AWT. `java.awt.Label` does not work in a Wicket page.

(callout)

How should we proceed with making the text between the `<h1>` tags change from within the Java program? To achieve this goal, we will add a label component (`org.apache.wicket.markup.html.basic.Label`) to the page to display the dynamic text. This is done in the constructor of the HelloWorld page:

```
public class HelloWorld extends WebPage {
    public HelloWorld() {
        add(new Label("message", "Hello, World!"));
    }
}
```

↑ identifier ↑ model

In the constructor we create a new Label instance, and give it two parameters: the component identifier and the text to display (the model). The component identifier needs to be identical to the identifier in the markup file, in this case 'message'. The text we provide as the model for the component will replace any text inside the component's tags. To learn more about the label component, please take a look at chapter XXX.

When we add a component to the Java class, we supply the component with an identifier. In the markup file we need to identify the markup tag where we want to bind the component. In this case, we need to tell Wicket to use the <h1> tag, and have the contents replaced:

```
<html>
<body>
  <h1 wicket:id="message">[text goes here]</h1>
</body>
</html>
```

↑ identifier ↑ gets replaced

The component identifier in the HTML and the Java file needs to be identical (case sensitive). The rules regarding component identifiers are explained in the next chapter. In figure 1.x we show how the two parts line up.

Figure 1.5 Lining up the component in the markup file and Java class.

```
<html>
<body>
  <h1 wicket:id="message">Hello, World!</h1>
</body>
</html>
```

↑ component identified by wicket:id

```
public class HelloWorld extends WebPage {
    public HelloWorld() {
      add(new Label("message", "Hello, Wicket!"));
    }
}
```

If we created an actual Wicket application, and direct our browser to the server running the application, Wicket would render the following markup and send it to the web client:

```
<html>
<body>
  <h1 wicket:id="message">Hello, Wicket!</h1>
</body>
</html>
```

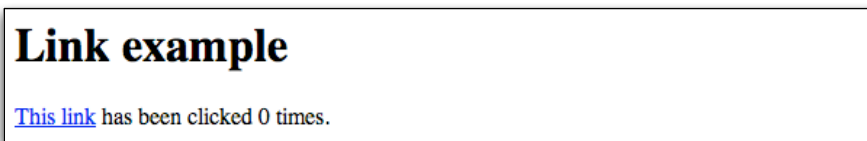
We provided the label in this example with a static string, but we could have retrieved the text from a database or a resource bundle, allowing for an internationalized greeting: 'Hallo, Wicket!', 'Bonjour, Wicket!' or 'Gutentag, Wicket!'. More information on internationalizing your applications is available in chapter 12.

Let's say goodbye to the Hello World example and look at something more dynamic: links.

1.3.2 Having fun with links

One of the most basic forms of user input in web applications is clicking a link. Most links take you to another page or even another website. Some show you a detail page of an item in a list, others delete the record they belong to. In this example we are going to use a link to increment a counter and use a label to display the number of clicks. Figure 1.5 shows what we want in a browser window.

Figure 1.5 The Link example shows a link that increases a counter with each click.



If we would handcraft the markup for this page, it would look something like this:

```
<html>
<body>
<a href="#">This link</a> has been clicked 123 times.
</body>
</html>
```

↙
↖

Link component
Label component

As you can see, there are two places where we need to add behavior to this page: the link and the number. This markup can serve us well. Let's make this file a Wicket markup file by adding the component identifiers.

```
<html>
<body>
<a href="#" wicket:id="link">This link</a> has been clicked #1
<span wicket:id="label">123</span> times. #2
</body>
</html>
```

(Annotation) <#1 Link component>
(Annotation) <#2 Label component>

In this markup file (LinkCounter.html) we added a Wicket identifier ('link') to the link and we surrounded the number with a span, identified by the Wicket identifier 'label'. This enables us to replace the contents of the span with the value of the counter. Now that we have the markup prepared, we can focus on the Java class for this page.

Creating the LinkCounter page

We need to have a place to store our counter value which is incremented every time the link is clicked, and we need a label to display the value of the counter. Let's see how this looks like in the next example.

```
public class LinkCounter extends WebPage {
    private int counter = 0; #1

    public LinkCounter() {
        add(new Link("link") { #3 #2
            @Override
            public void onClick() {
                counter++;
            }
        });
        add(new Label("label", #4
            new PropertyModel(this, "counter")); #5
    }
}
```

(Annotation) <#1 counts clicks>

(Annotation) <#2 adds link>

(Annotation) <#3 handles onclick>

(Annotation) <#4 shows counter value>

(Annotation) <#5 binds counter value>

First we added a property to the page so we can count the number of clicks (#1). Next we added the link component to the page (#2). We can't just instantiate this particular Link component, because it is abstract and requires us to implement the behavior for clicking on the link in the method 'onClick'. Using an anonymous subclass of the Link class (#3) we provide the link with the desired behavior: we increase the counter in the onClick method.

Finally we added the label showing the value of our counter (#4). Instead of querying the value of the counter ourselves, converting it to a String and setting the value on the label, we provided the label with a PropertyModel (#5). We will explain how property models work in more detail in chapter 5 where we discuss models. For now it is sufficient to say that this enables the Label component to read the counter value (using the expression "counter") from the page (the 'this' parameter) every time the page is refreshed.

If you run the LinkCounter and click on the link you should see the counter increase with each click. While this example may have been sufficient for a book written in 2004, no book on web applications today is complete without Ajax.

Putting Ajax in the mix

If you haven't heard of Ajax yet - and we don't mean the Dutch soccer club nor the house cleaning agent -, then it is best think of it as the technology that enables websites such as Google Maps, Google Mail and Microsoft Live to have a rich user experience. This user experience is typically achieved by only updating part of a page, instead of reloading the whole document in the browser. In chapter 8 (Rich Components) we will discuss Ajax in much more detail. For now, let's make our link update only the label and not the whole page.

With this new Ajax technology we are able to update only part of a page as opposed to having to reload the whole page on each request. To implement this Ajax behavior we have to add an

extra identifier to our markup and we will need to change our link so that it knows how to answer these special Ajax requests.

First our markup: when Wicket updates a component in the page using Ajax, it will try to find the tags of the component in the browser's document object model (DOM). The component's tags make up a DOM element. All DOM elements have a markup identifier (the id attribute of HTML tags), and it is used to query the document object model to find the specific element.

Note that the markup identifier is not the same as the Wicket identifier. Though they can have the same value (and often will), the Wicket identifier serves a different purpose and its allowed values have different constraints. You can read more on these subjects in chapters 2, 4 and 8. For now, just follow our lead. Remember the LinkCounter.html markup file? Here it is again, unmodified:

```
<html>
<body>
<a href="#" wicket:id="link">This link</a> has been clicked
<span wicket:id="label">123</span> times.
</body>
</html>
```

Let's now take a look at the Java side of the matter. In our non-Ajax example we use a normal link, answering to normal, non-Ajax requests. When the link is clicked it updates the whole page. In our Ajax example we will ensure that this link behaves in both Web 1.0 and Web 2.0 surroundings by utilizing the AjaxFallbackLink.

The AjaxFallbackLink is a Wicket component that works in browsers with and without JavaScript support. When JavaScript is available it will use Ajax to update the specified components on the page. If JavaScript is unavailable it will use a normal web request just as the normal link, updating the whole page.

This fallback behavior is pretty handy if you have to comply with government regulations regarding accessibility (for example section 508 of the Rehabilitation Act, US federal law). Let's see how this looks like in our next example.

```
public class LinkCounter extends WebPage {
    private int counter;
    private Label label;          #1

    public LinkCounter() {
        add(new AjaxFallbackLink("link") {          #2
            @Override
            public void onClick(AjaxRequestTarget target) { #3
                counter++;
                if(target != null) {                #4
                    target.addComponent(label);      #5
                }
            }
        });
        label = new Label("label", new PropertyModel(this, "counter"));
        label.setOutputMarkupId(true);             #6
        add(label);
    }
}
```

```

}

(Annotation) <#1 Added reference>
(Annotation) <#2 Changed class>
(Annotation) <#3 New parameter>
(Annotation) <#4 Null in fallback>
(Annotation) <#5 Updates label>
(Annotation) <#6 Generate id attribute>

```

In this class we added a reference to the label in our page (#1), so we can reference it when we need to update the label in our Ajax request (#5). We changed our link to an `AjaxFallbackLink` (#2) and had to add a new parameter to our `onClick` implementation: an `AjaxRequestTarget` (#3). This target requires some explanation: it is used to identify the components that need to be updated in the Ajax request. It is specifically used for Ajax requests. You can add components and JavaScript to it which will be executed on the client. In this case, we add the label component to the target, updating it with every Ajax request.

Because our link is an `AjaxFallbackLink`, it also responds to non-Ajax requests. When a normal request comes in (i.e. when JavaScript is not available) the `AjaxRequestTarget` is null. Therefore we have to check for that condition (#4) when we try to update the label component (#5).

Finally we have to tell Wicket to generate a markup identifier for the label (#6). To be able to update the markup in the browser, the label needs to have a markup identifier. This is the 'id' attribute of a HTML tag:

```
<span id="foo"></span>
```

In the Ajax processing, Wicket will generate the new markup for the label and replace that part of the document, using the markup identifier to look up the specific markup in the page.

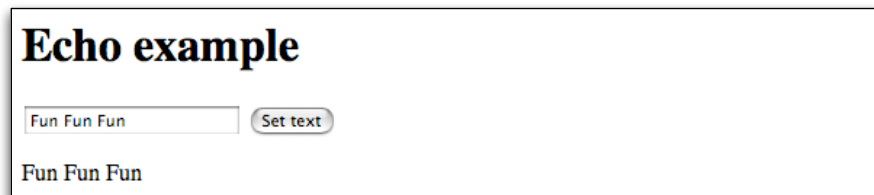
As you can see, you didn't have to create a single line of JavaScript yourself. All it took was adding a markup identifier to the label component, and to make our link Ajax aware. To learn more about creating rich internet applications, please refer to chapter 8. If you want to learn more about links and linking between pages, please read chapter 4.

In this example we performed some action in response to an user clicking a link. This is not the only way to interact with your users. Using a form and input fields is another way.

1.3.3 The Wicket Echo application

Another fun example is a page with form for collecting a line from a user, and a label which displays the last input. A screenshot of a possible implementation is shown in figure 1.6.

Figure 1.6 The echo example echoes the text in the input field on the page.



If we just focus on the markup, it would look something like the following:

```
<html>
<head><title>Echo Application</title></head>
<body>
  <h1>Echo example</h1>
  <form>
    <input type="field" />
    <input type="submit" value="Set text" />
  </form>
  <p>Fun Fun Fun</p>
</body>
</html>
```

(Annotation) <#1 input form>
 (Annotation) <#2 text field>
 (Annotation) <#3 submit button>
 (Annotation) <#4 message>

The input for the echo application is submitted using a form (#1). The form contains a text field for typing in the message (#2) and a submit button (#3). The echoed message is show below the form (#4). The following markup file shows the result of assigning Wicket identifiers to the components in the markup:

```
<html>
<head><title>Echo Application</title></head>
<body>
  <h1>Echo example</h1>
  <form wicket:id="form">
    <input wicket:id="field" type="text" />
    <input wicket:id="button" type="submit" value="Set text" />
  </form>
  <p wicket:id="message">Fun Fun Fun</p>
</body>
</html>
```

We added Wicket component identifiers to all markup tags identified in the previous example: the form, the text field, the button and the message. Now we have to create a corresponding Java class that echoes the message send using the form in the message container. Have a look at the next class:

```
public class EchoPage extends WebPage {
  private Label label;
  private TextField field;

  public EchoPage() {
    Form form = new Form("form");
    field = new TextField("field", new Model(""));
    form.add(field);
    form.add(new Button("button") {
      @Override
      public void onSubmit() {
        String value = (String)field.getModelObject();
        label.setModelObject(value);
      }
    });
  }
}
```

```

        field.setModelObject("");
    }
};
    label = new Label("message", new Model(""));
}
}

```

(Annotation) <#1 For later reference>
 (Annotation) <#2 Creates input form>
 (Annotation) <#3 Adds field to form>
 (Annotation) <#4 Submits form>
 (Annotation) <#5 Gets message>
 (Annotation) <#6 Sets label>
 (Annotation) <#7 Clears field>

The EchoPage keeps references (#1) to two components: the label and the field. We will use these references to modify the components' model values when the form is submitted.

We introduced three new components for this page: the Form, the TextField and the Button. The Form component (#2) is necessary for listening to submit events: it will parse the incoming request and populate the fields that are part of the form. We discuss forms and how submitting them works in much greater detail in chapter 5.

The TextField (#3) is used to receive the input of the user. In this case we add a new model with an empty string to store the input. This sets the contents of the text field to be empty, so it is rendered as an empty field.

The Button component is used to submit the form. The Button requires us to create a subclass and implement the onSubmit event (#4). In the onSubmit handler we retrieve the value of the field (#5) and set it on the label (#6). Finally we clear the contents of the text field (#7) so it is available for new input.

This example shows how a component framework works. Using Wicket gives you just HTML and Java. In fact the way we developed this page is quite the way many of our core contributors work in their day jobs: create markup, identify components, assign Wicket identifiers, create Java.

1.4 Summary

You have read in this chapter that Apache Wicket is a Java software framework to enable component oriented, programmatic manipulation of markup. It provides a stateful programming model based on Just Java and Just HTML. After telling our tale of how we found Wicket and explaining a bit about the motivations behind the programming model, we looked at some examples of what coding with Apache Wicket looks like.

We hope you liked our story so far. The next chapter will provide a high level view of the most important concepts of Wicket. Feel free to skip this chapter for now if you are more interested in looking at code, though it might help you put the chapters after that better in context.