As you can see, there are two places where we need to add dynamic 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
<span wicket:id="label">123</span> times.
</body>
</html>
```

Link component

Label component

In this markup file (LinkCounter.html), we add a Wicket identifier (`link`) to the link and surround the number with a `span`, using the Wicket identifier `label`. This enables us to replace the contents of the `span` with the actual value of the counter at runtime. Now that we have the markup prepared, we can focus on the Java class for this page.

CREATING THE LINKCOUNTER PAGE

We need 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 in the next example:

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

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

❶ **Count clicks**

❷ **Add link to page**

❸ **Show counter value**

First, we add a property to the page so we can count the number of clicks ❶. Next, we add the `Link` component to the page ❷. We can't simply instantiate this particular `Link` component, because the `Link` class is abstract and requires us to implement the behavior for clicking the link in the method `onClick`. Using an anonymous subclass of the `Link` class, we provide the link with the desired behavior: we increase the value of the counter in the `onClick` method.

Finally, we add the label showing the value of the counter ❸. Instead of querying the value of the counter ourselves, converting it to a `String`, and setting the value on the label, we provide the label with a `PropertyModel`. We'll explain how property models work in more detail in chapter 4, where we discuss models. For now, it's 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 the link, you should see the counter's value increase with each click.