**Creating Labels for Controls and the <label> Element**

Forms can be confusing enough at the best of times. I ’ve received many insurance and tax forms that have left me scratching my head, and I’ m sure I’ m not the only one. If you are creating a form for your site, it is worth spending time to provide good labeling so that the user knows what data he or she should be entering where. If visitors have difficulty understanding your form, they will be less likely to complete the form (in particular if they are purchasing something), or they are more likely to make a mistake when filling it in. Some form controls, such as buttons, already have labels. For the majority of form controls, however, you will have to provide the label yourself.

For controls that do not have a label, you should use the <label> element. This element does not affect the form in any way other than telling users what information they should be entering.

Example/Source Code (Using Label):



Output:



**Note:**

***You can see that this form has been placed inside a table; this ensures that even if the labels are of different lengths, the text inputs are aligned in their own column. If a list of text inputs is not aligned, it can be harder to use.***

As you can see here, the <label> element carries an attribute called for, which indicates the form control associated with the label. The value of the **for attribute** should be the same as the value of the **id attribute** on the corresponding form control. For example, the textbox form control, where a user enters his or her username, has an id attribute whose value is Uname, and the label for this textbox has a for attribute whose value is also Uname.

Another way to use the <label> element is as a containing element. When you use the <label> element this way, you do not need to use the **for attribute** because it applies to the form element that is inside it. This kind of label is sometimes known as an ***implicit label***.

Example:

<form name=”frmLogin” >

<label> Username <input type=”text” id=”Uname” name=”txtUserName” > < /label >

<label> Password <input type=”password” id=”Pwd” name=”pwdPassword” >

</label>

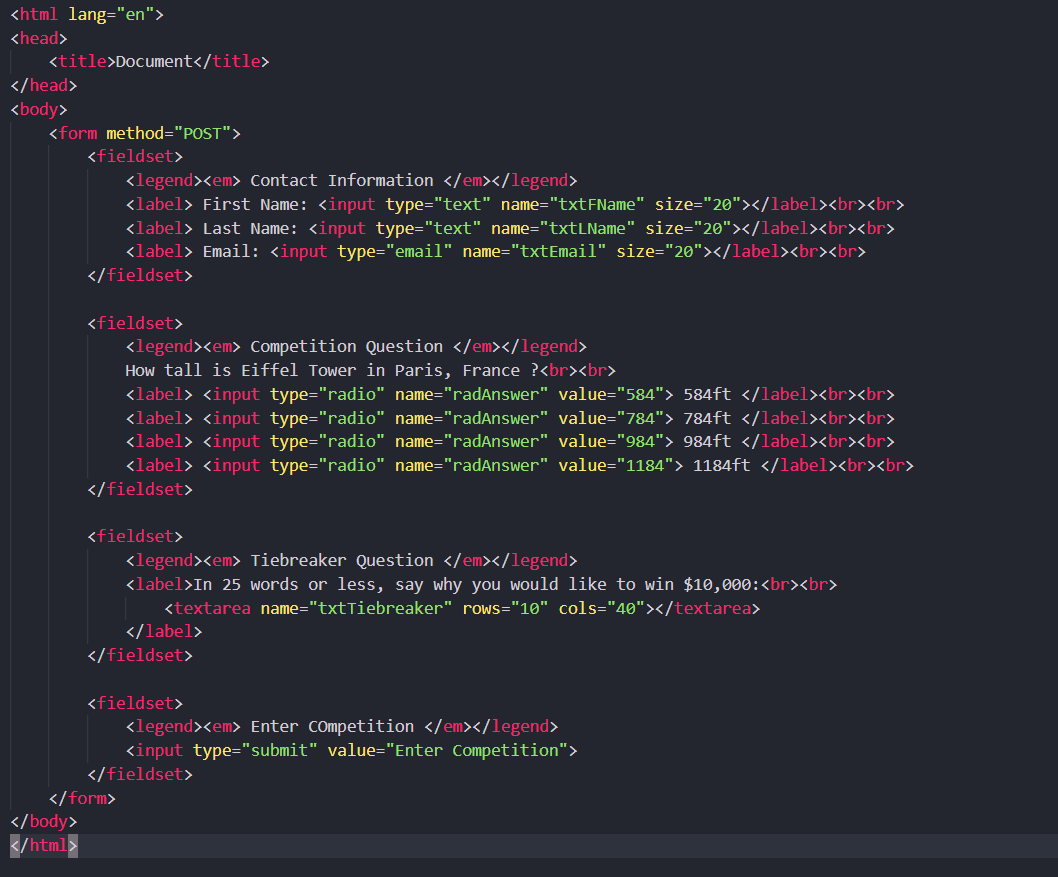
</form>

**Structuring Your Forms with <fieldset> and <legend> Elements:**

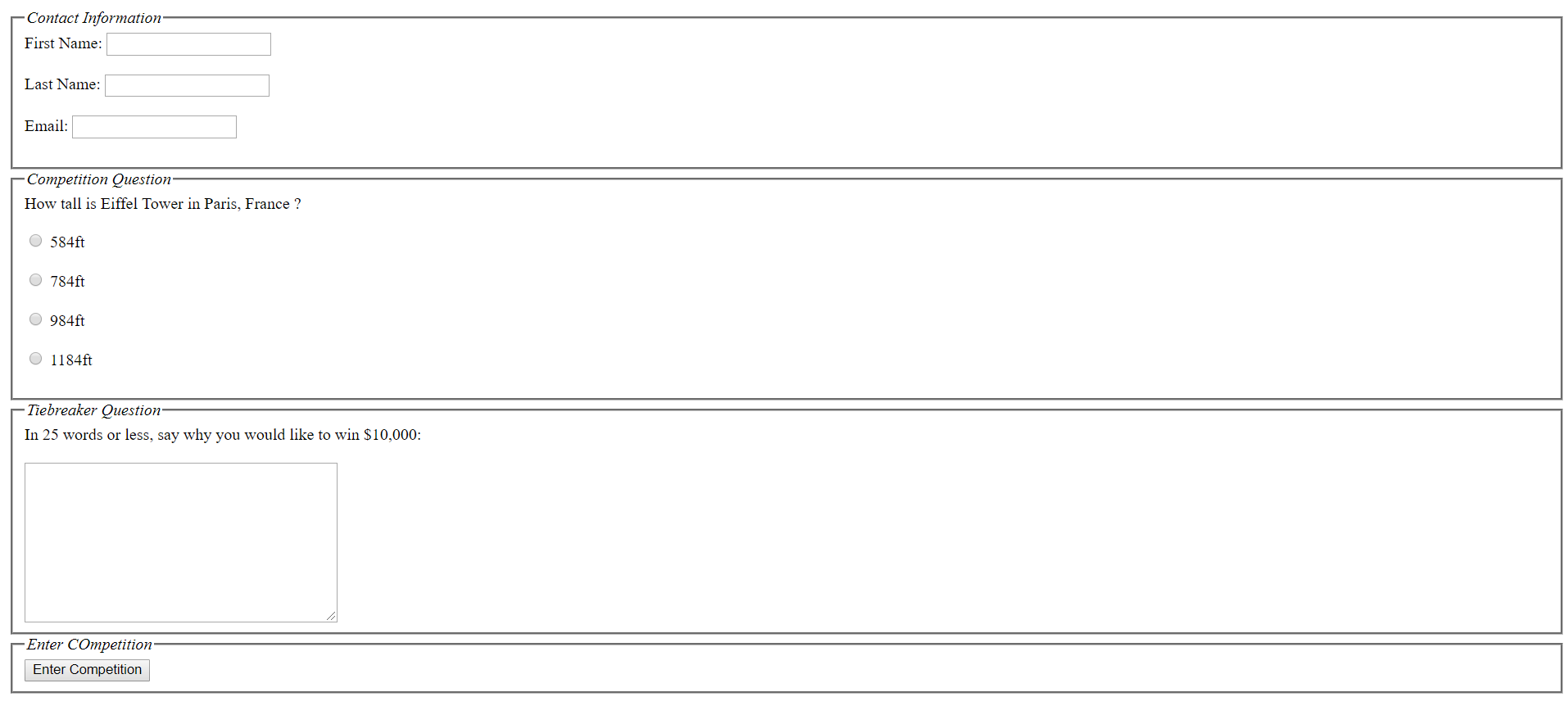
Large forms can be confusing for users, so it’ s good practice to group together related form controls. The <fieldset> and < legend > elements do exactly this — help you group controls.

* The <fieldset> element creates a border around the group of form controls to show that they are related.
* The <legend> element allows you to specify a caption for the <fieldset> element, which acts as a title for the group of form controls. When used, the <legend> element should always be the first child of the <fieldset> element.

Example/Source Code:



Output:



The <fieldset> element can take the following attributes:

* All the universal attributes
* The basic event attributes

If you use a table to format in your form, the entire <table> element must appear inside the <fieldset> element. If a <fieldset> resides within a table that is used to format the page, then the entire fieldset must reside within the same cell.

The <legend> element can take the following attributes:

* accesskey (The accesskey attribute specifies a shortcut key to activate/focus an element)
* align, which you have seen already, and is deprecated — you should use CSS positioning

instead.

* All the universal attributes.
* UI event attributes.

**Focus:**

When a web page featuring several links or several form controls loads, you may have noticed that you are able to use your Tab key to move between those elements (or Shift+Tab to move backward through elements). As you move between them, the web browser tends to add some type of border or highlighting to that element (be it a link or a form control). This is known as *focus*.

Only elements that a user can interact with, such as links and form controls, can receive focus. Indeed, if a user is expected to interact with an element, that element *must* be able to receive focus.

An element can gain focus in three ways:

* An element can be selected using a pointing device such as a mouse or trackball.
* Elements that can gain focus can be navigated between using the keyboard — often

using the Tab key (or Shift Tab to move backward through elements). As you are about to see, the elements in some documents can be given a fixed *tabbing order,* indicating the order in which elements gain focus when the user pressed the tab key.

* A web - page author can indicate that an element should receive focus when a user presses a keyboard shortcut known as an *access key.* For example, if the page author set the access key on a search box to be the key for the letter *s,* on a PC you would likely press the Alt key plus the access key (Alt+S), whereas on a Mac you would press the Control key with an access key (Control+S), and the corresponding form control would gain focus.

**Tabbing Order:**

If you want to control the order in which elements can gain focus, you can use the tabindex attribute to give that element a number between 0 and 32767, which becomes part of the tabbing order. Every time the user presses the Tab key, the focus moves to the element with the next highest tabbing order (and again, Shift+Tab moves focus in reverse order).

The following elements can carry a tabindex attribute:

* <a>
* <area>
* <button>
* <input>
* <object>
* <select>
* <textarea>

After a user has tabbed through all elements in a document that can gain focus, then focus may be given to other browser features (most commonly the address bar).

Example/Source Code:

To demonstrate how tabbing order works, the following example gives focus to the checkboxes in a different order than you might expect :

<form action=”/a.html” method=”get” name=”frmTabExample”>

<input type=”checkbox” name=”chkNumber” value=”1” tabindex=”3” > One <br>

<input type=”checkbox” name=”chkNumber” value=”2” tabindex=”5” > Two <br>

<input type=”checkbox” name=”chkNumber” value=”3” tabindex=”4” > Three <br>

<input type=”checkbox” name=”chkNumber” value=”4” tabindex=”1” > Four <br>

<input type=”checkbox” name=”chkNumber” value=”5” tabindex=”2” > Five <br>

<input type=”submit” value=”Submit” / >

< /form >

**Access Keys:**

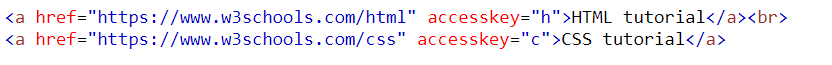
*Access keys* act just like keyboard shortcuts. The access key is a single character from the document’ s character set that is expected to appear on the user’ s keyboard. When this key is used in conjunction with another key or keys (such as Alt with IE on Windows, Alt and Shift with Firefox on Windows, and Control on an Apple), the browser automatically goes to that section. (Exactly which key must be used in conjunction with the access key depends upon the operating system and browser.)

The access key is defined using the accesskey attribute. The value of this attribute is the character (and key on the keyboard) you want the user to be able to press (in conjunction with the other key/keys that are dependent upon the operating system and browser).

The following elements can carry an access key attribute:

* <a>
* <area>
* <button>
* <input>
* <label>
* <legend>
* <textarea>

Example:



Output:



**Disabled and Read - Only Controls:**

Throughout the chapter, you have seen that several of the elements can carry attributes called disabled and readonly:

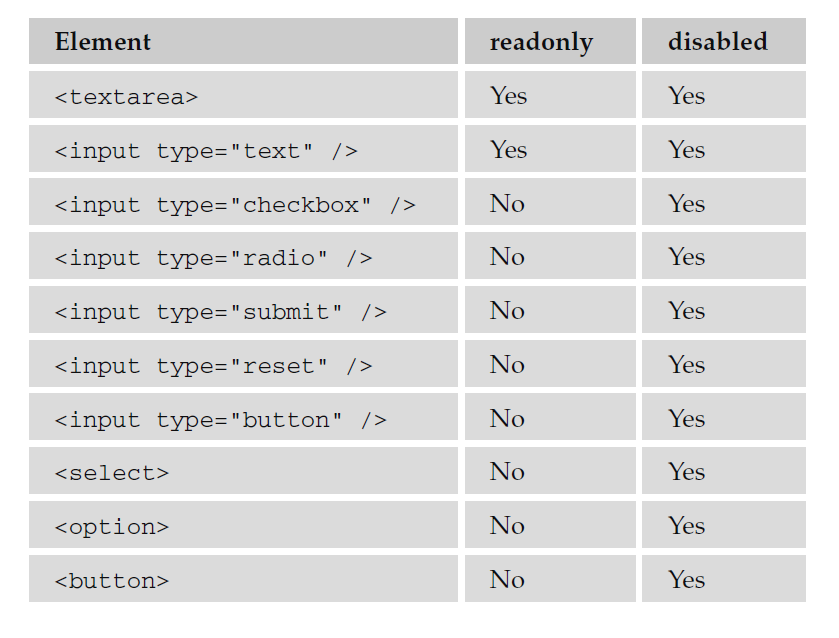
* The readonly attribute prevents users from changing the value of the form controls themselves, although it may be modified by a script. The name and value of any readonly control **will be sent to the server**. The value of this attribute should be readonly.
* The disabled attribute disables the form control so that users cannot alter it. A script can be used to re - enable the control, but unless a control is re - enabled, the name and **value will not be sent to the server**. The value of this attribute should be disabled.

A readonly control is particularly helpful when you want to stop visitors from changing a part of the form, perhaps because it must not change (for example, if you put terms and conditions inside a text area).

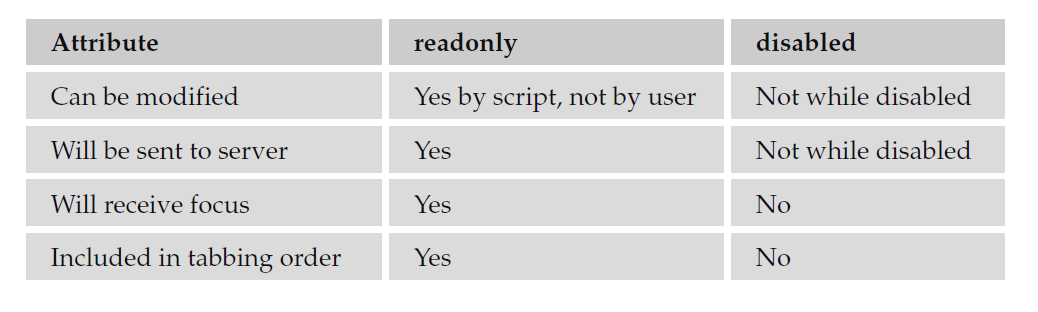
The disabled attribute is particularly helpful when preventing users from interacting with a

control until they have done something else. For example, you might use a script to disable a submit button until all of the form fields contain a value.

The following table indicates which form controls work with the readonly and disabled attributes:



The following table indicates the main differences between the readonly and disabled attributes:



**Sending Form Data to the server:**

When your browser requests a web page and when the server sends a page back to the browser, you use the Hypertext Transfer Protocol (HTTP).

There are two methods that a browser can use to send form data to the server — HTTP get and HTTP post — and you specify which should be used by adding the method attribute on the <form> element.

If the < form > element does not carry a method attribute, then by default the get method will be used. If you are using a file upload form control, you must choose the post method (and you must set the enctype attribute to have a value of multipart/form - data). Let’s take a closer look at each of these methods.

**HTTP get:**

When you send form data to the server using the HTTP get method, the form data is appended to the URL that is specified in the action attribute of the <form> element.

The form data is separated from the URL using a question mark. Following the question mark, you get the name/value pairs for each form control. Each name/value pair is separated by an ampersand (&).

For example, take the following login form, which you saw when the password form control was introduced:

<form action = ”http://www.example.com/login.aspx” method=”get”>

Username:

<input type=”text” name=”txtUsername” value=”” size=”20” maxlength=”20”> < br >

Password:

<input type=”password” name=”pwdPassword” value=”” size=”20” maxlength=”20”>

<input type=”submit”>

</form>

When you click the submit button, your username and password are appended to the URL http://www.example.com/login.aspx in what is known as the *query string.* It should look like this:

http://www.example.com/login.aspx?txtUsername=Bob & pwdPassword=LetMeIn

Note that when a browser requests a URL with any spaces or unsafe characters such as /, \, =, &, and + (which have special meanings in URLs), they are replaced with a hex code to represent that character.

This is done automatically by the browser, and is known as *URL encoding.* When the data reaches the server, the server will usually un - encode the special characters automatically.

One of the advantages of passing form data in a URL is that it can be bookmarked. If you look at

searches performed on major search engines such as Google, they tend to use the get method so that the page can be bookmarked.

The get method, however, has some disadvantages. Indeed, when sending sensitive data such as the password shown here, or credit card details, you should not use the get method because the sensitive data becomes part of the URL and is in full view to everyone (and could be bookmarked).

You should not use the HTTP get method when:

* You are dealing with sensitive information, such as passwords or credit card details (because the sensitive form data would be visible as part of a URL).
* You are updating a data source such as a database or spreadsheet (because someone could make up URLs that would alter your data source).
* Your form contains a file upload control (because uploaded files cannot be passed in the URL).
* Your users might enter non - ASCII characters such as Hebrew or Cyrillic characters.

In these circumstances, you should use the HTTP post method.

**HTTP post:**

When you send data from a form to the server using the HTTP post method, the form data is sent transparently in what is known as the *HTTP headers.* While you do not see these headers, they are not, strictly speaking, secure on their own. If you are sending sensitive information such as credit card details, the data should be sent under a *Secure Sockets Layer,* or *SSL,* and they should be in encrypted.

If the login form you just saw was sent using the post method, it could be represented like this in the HTTP headers:

User-agent: MSIE 7

Content-Type: application/x-www-form-urlencoded

Content-length: 35

...other headers go here...

txtUserName=Bob & pwdPassword=LetMeIn

Note that the last line is the form data, and that it is in exactly the same format as the data after the question mark in the get method — it would also be URL - encoded so any spaces or unsafe characters such as /, \ , =, & , and + (which have special meanings in URLs) are replaced with a hex code to represent that character as they were in HTTP get requests.

There is nothing to stop you from using the post method to send form data to a page that also contains a query string. For example, you might have one page to handle users who want to subscribe to or unsubscribe from a newsletter, and you might choose to indicate whether a user wanted to subscribe or unsubscribe in the query string. Meanwhile, you might want to send their actual contact details in a form that uses the post method because you are updating a data source. In this case, you could use the following <form> element:

<form action = “http://www.example.com/newsletter.asp?action=subscribe” method= ”post” >

The only issue with using the HTTP post method is that the information the user entered on the form cannot be bookmarked in the same way it can when it is contained in the URL. So you cannot use it to retrieve a page that was generated using specific form data as you can when you bookmark a page generated by most search engines, but it is good for security reasons.