Part 2

Browser-based apps

For a very long time developers were processing everything—form validation, file management, storage, messaging, and other vital application functionality—on the server. Server-side processing was a great idea for security reasons, lack of user processing power, and many other issues. There were workarounds through technologies such as Flash and Java, but the mobile market explosion revealed unanticipated limitations that HTML5 is aiming to fix.

Thanks to major advances in JavaScript processing power and new W3C standards, you can now perform server-side tasks through a user's browser (aka client-side). Performing complex tasks through browsers saves tons of money on server costs, allows startups to easily create complex apps, and creates seemingly instant application responses during heavy load times. It also opens up a completely different thought process on application development and deployment to mobile and desktop. And they can both be done at the same time if you play your cards right.

Many popular web applications use HTML5's application features. Google Drive, for example, uses a new storage technology known as the Indexed Database API. You've probably also used HTML5's WebSockets, forms, and many other features that we'll be covering throughout this section. By the time you've completed part 2 (chapters 2–5), you'll know enough to put together a small application with minimal server usage.

Chapter 2 at a glance

Topic	Description, methods, and so on	Page
New input types[1]	HTML5 `<input>` element types	
	▪ `email`	42
	▪ `tel`	42
	▪ `number`	46
	▪ `month`	49
New input attributes[1]	HTML5 attributes on `<input>` elements	
	▪ `required`	42
	▪ `pattern`	49
	▪ `autofocus`	43
	▪ `placeholder`	42
	▪ `min` and `max`	46
`data-*` attributes	Storing key/value data in attributes on elements	46
`valueAsNumber` property	Reading input values in numeric format	54
`<output>` element	Displaying the output of calculations	47
Preventing validation	Providing means of bypassing client-side validation	
	▪ `formnovalidate` attribute	51
	▪ `formaction` attribute	51
Constraint Validation API	Client-side API for validation	
	▪ `setCustomValidity` method	59
	▪ `validationMessage` property	59
	▪ `invalid` event	60
CSS3 pseudo-classes	Styling invalid elements with CSS3	61
Backward compatibility	Feature detection and unsupported browsers	
	▪ Modernizr.js	63
	▪ Polyfills	64
	▪ Validation	65

[1] Only the input types and attributes used or discussed in this chapter are listed here. For comprehensive coverage, visit mng.bz/wj56.

Look for this icon 

throughout the chapter to quickly locate the topics outlined in this table.

2 Form creation: input widgets, data binding, and data validation

This chapter covers

- New HTML5 input types and attributes
- `data-*` attributes, `valueAsNumber` property, and the `<output>` element
- Constraint Validation API
- Ways to bypass validation
- CSS3 pseudo-classes
- HTML5 feature detection with Modernizr and backward compatibility with polyfill

As the web has matured, the need for a much richer set of form field types and widgets has emerged. Today's users expect a consistent standard between web applications, particularly when it comes to data validation. HTML5 meets this requirement with 13 new form input types, ranging from number spinners and sliders to date- and color-pickers.

The standard also defines new attributes you can apply to `<input>` elements to enhance the functionality of your forms, including presentational attributes like `placeholder` and `autofocus`, as well as validation attributes such as `required` and `pattern`. You can even use a set of new CSS3 pseudo-classes to style valid or invalid form elements with zero JavaScript. And if you have advanced validation requirements you can't provide natively, the new Constraint Validation API offers a standardized

JavaScript API that can test for the validity of form fields, along with a new event you can use to detect an invalid data entry.

In this chapter, you'll implement all of these new features by building an order form for computer products. The form will use HTML5 validation to "sanitize" the input on the client side before it's submitted.

Why build this chapter's order form?

While working through this chapter's sample application, you'll learn to use:

- *New input types* to provide more widgets with less coding
- *New input attributes* to provide validation with less coding
- *data-* attributes* to bind data to HTML elements
- *Constraint Validation API* features to create custom validation tests

We'll get started by showing you a preview of the form and helping you get your prerequisites in order.

2.1 Previewing the form and gathering prerequisites

The order form you'll build in this chapter, shown in figure 2.1, allows users to enter personal data, login details, and order and payment information.

The form makes use of several new HTML5 features:

- *Form* `<input>` *element types* (`email`, `tel`, `number`, and `month`) and *attributes* (`required`, `pattern`, `autofocus`, `placeholder`, and `max` and `min`) to provide users with better widgets and data validation when appropriate.
- The `data-*` *attributes* to hold the price of each product, the `valueAsNumber` *property* to read input values in numeric format, and the `<output>` *element* to present subtotals and grand totals.
- The `formnovalidate` and `formaction` *attributes* to bypass data validation and save an incomplete form.
- The *Constraint Validation API* to perform custom validation and detect when the user attempts to submit the form with elements that are invalid, and *CSS3 pseudo-class selectors* to style invalid elements.
- The *Modernizr.js JavaScript library* and *polyfills* to serve users whose browsers don't support various HTML5 features. (Although we admit that Modernizr and polyfills aren't strictly HTML5 features, we recommend that you use them if you're serious about developing HTML5 applications.)

When you've finished, the order form will be functional in the latest versions of all the major browsers, although you may find varying levels of support for some features such as widgets for new input types and inline error messages for the Constraint Validation API. But browser hiccups will become less and less an issue as support for the new features increases.

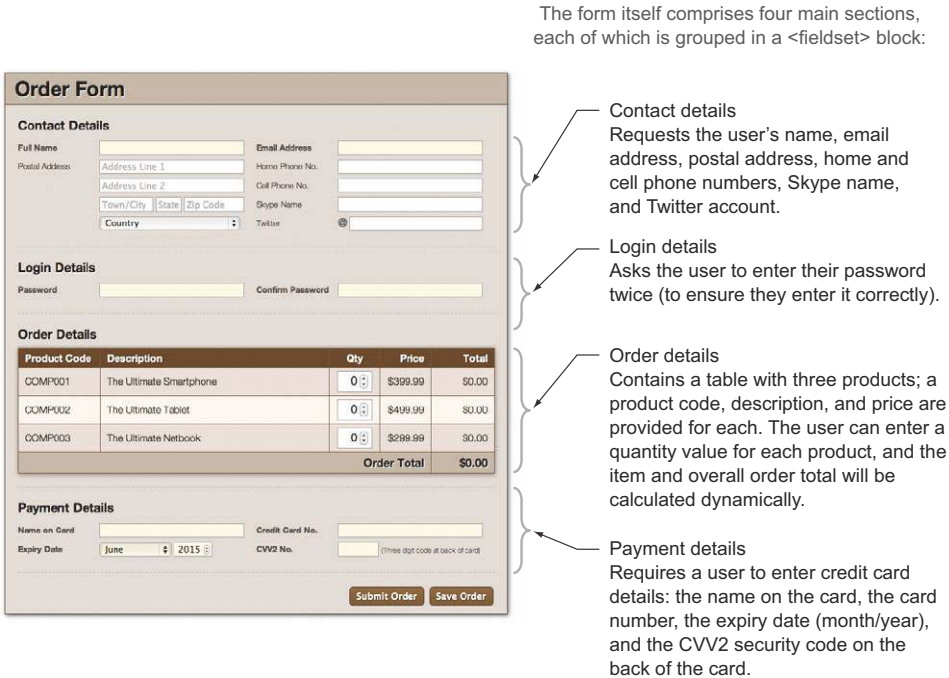

Figure 2.1 The order form running in Google Chrome. The user is given two options when submitting the form: Submit Order or Save Order. The Submit Order button performs validation and processes a user's order, whereas the Save Order button bypasses the validation and saves the details, so users can come back later and finish filling out their order.

NOTE This chapter covers only the client-side portion of the order form. When the form is submitted, it makes a request to a URL. To perform further processing, you'll need to implement the form on the server side using your choice of server-side language or framework (such as PHP or Ruby on Rails). The server-side aspect is outside the scope of this book.

2.1.1 Gathering the application prerequisites

You'll work with five files in this chapter:

- An HTML document
- A JavaScript source file
- A CSS stylesheet
- The Modernizr library
- The month-picker polyfill script

The stylesheet and polyfill are part of the chapter's source code archive, but you'll need to download the Modernizr library from its website at http://modernizr.com/. Rename the .js file to modernizr.js and place it, along with both the CSS file and monthpicker.js, in the application's directory.

TIP Modernizr offers two choices when you download the library—development or production builds. The development build contains the entire Modernizr test suite and isn't compressed or minified. If you're in a hurry and don't mind the large file size, use the development build. On the other hand, the production build allows you to configure which tests you want to include and will be compressed and minified to ensure a minimal file size. If you choose to use the production build, be sure to include the Input Attributes, Input Types, and Modernizr.load tests, because these are required in this chapter. You'll learn more about Modernizr later in the chapter.

With the preview and prerequisites out of the way, it's time to start working on the form's UI.

2.2 Building a form's user interface

The work in this section—building the UI—involves defining the HTML document structure, building the individual form sections, and allowing users to determine whether to save or submit form details.

In this section, you'll learn

- How to provide users with widgets and data validation using HTML5 form `<input>` element types and attributes
- How to store the price of each product with `data-*` attributes
- How to present subtotals and grand totals using the `<output>` element
- How to bypass form validation and save an incomplete form using the form attribute properties, `formnovalidate` and `formaction`

We'll walk you through the UI work in seven steps:

- Step 1: Create index.html and load external files.
- Step 2: Create the Contact Details form section.
- Step 3: Build the Login Details form section.
- Step 4: Build the Order Details form section.
- Step 5: Build the Payment Details form section.
- Step 6: Bypass form validation and save form data.
- Step 7: Change the form action in older browsers.

First up, the HTML document.

2.2.1 Defining a form's basic HTML document structure

Before you begin, we recommend that you create a new directory on your system. Ideally, it would be a location on a web server, but that's not a requirement for the example.

STEP 1: CREATE INDEX.HTML AND LOAD EXTERNAL FILES

Create a new file named index.html and place it in the new directory. Then, add the contents of the following listing to that file. The code loads external dependencies (CSS and JavaScript files) and defines the `<form>` element with the heading at the top and the buttons at the bottom.

Listing 2.1 index.html—HTML document structure

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Order Form</title>
    <link rel="stylesheet" href="style.css">
    <script src="modernizr.js"></script>
    <script src="app.js"></script>
</head>
<body>
    <form name="order" method="post" action="/submit">
        <h1>Order Form</h1>

        <div class="buttons">
            <input type="submit" value="Submit Order">
            <input type="submit" id="saveOrder" value="Save Order">
        </div>
    </form>
</body>
</html>
```

Load the Modernizr library. You may wonder why we don't include the monthpicker.js file—later we'll use the Modernizr.load method to dynamically load that file, but only if it's needed by the user's web browser.

The order form is split into four sections, which we'll work on sequentially: Contact Details, Login Details, and Payment Details in this section, and Order Details in the section that follows.

2.2.2 Using the form input types email and tel and the input attributes autofocus, required, and placeholder

Before you actually start building the order form, we'd like to give you more details about the new input types and attributes and show you how to use these types and attributes to build your forms in less time. As we proceed, we'll improve the example form with the `email` and `tel` (for telephone) input types and also make use of the `autofocus`, `required`, and `placeholder` attributes.

`email` input type	10.0	4.0	10.0	10.6	5.0*
`tel` input type	10.0	4.0	10.0	10.6	5.0

* Indicates partial support

Figure 2.2 Different virtual keyboards are displayed on an iPhone for different input types—from left to right: `text`, `email`, `url`, and `tel`. Notice how symbols are added and removed for the `email` and `url` input types. An entirely different keyboard is displayed for the `number` input type.

Core API

Both the `email` and `tel` input types look identical to the standard text input element. But if the user is browsing on a mobile device that supports these input types, it can display different virtual keyboard layouts depending on what type of data the user is entering. See figure 2.2 for an example of this in action.

In the case of the `email` input type, the browser will also check that the user inputs a valid email address. If not, it will raise an error when the user submits the form. The error style is defined by the browser, which means it will look somewhat different depending on the user's browser. Figure 2.3 illustrates this.

Core API

Now, let's look at three additional attributes: `autofocus`, `required`, and `placeholder`.

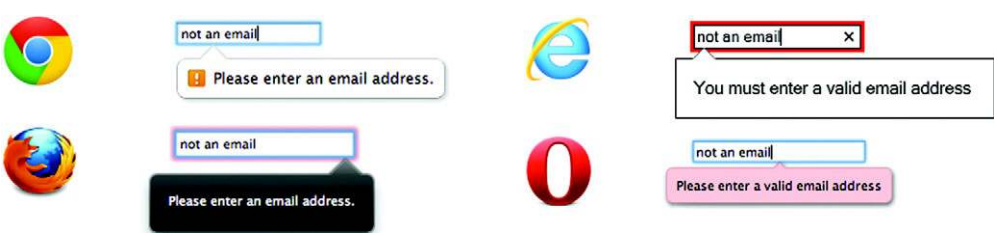

Figure 2.3 Each web browser vendor implements a different style when presenting input validation errors to the user. As more websites begin to use HTML5 form validation, users will become more familiar with the standard style of error message displayed by their browser of choice.

THE AUTOFOCUS, REQUIRED, AND PLACEHOLDER ATTRIBUTES

`autofocus` attribute	6.0	4.0	10.0	11.0	5.0
`required` attribute	10.0	4.0	10.0	10.0	5.0*
`placeholder` attribute	4.0	4.0	10.0	11.6	5.0

* Indicates partial support

The `autofocus` attribute is self-explanatory; it allows you to define which input element should receive focus when the page loads. The `required` attribute is also straightforward—it allows you to define that a field must contain input from the user in order to be valid. You'll learn much more about HTML5 form validation later in the chapter. The `placeholder` attribute allows you to define a piece of text that will appear in a field when it's empty and inactive. As soon as the user types in the field, the placeholder text will be cleared and replaced with the user's input. This is illustrated in figure 2.4.

Figure 2.4 Demonstration of the `placeholder` attribute. This example displays a search input field, with the placeholder text "What are you looking for?" When the user enters a value, the placeholder text is replaced with that value.

STEP 2: CREATE THE CONTACT DETAILS FORM SECTION

Let's integrate those new features into the Contact Details section of the form, the markup for which is shown in the next listing. You should add this code to the index.html file, immediately after the line `<h1>Order Form</h1>` from the previous listing.

Listing 2.2 index.html—The Contact Details form section

```
<fieldset>
    <legend>Contact Details</legend>
    <ul>
        <li>
            <label class="required">
                <div>Full Name</div>
                <input name="name" required autofocus>
            </label>
        </li>
        <li>
            <label class="required">
                <div>Email Address</div>
                <input type="email" name="email" required>
            </label>
        </li>
        <li>
```

The name field is the first in the form, so it makes sense to autofocus it. It's also a required field.

The email field uses the new email input type. It's also a required field.

Each of the lines in the address field uses the placeholder attribute to indicate what type of information is relevant for each of the fields.

```
                <label>
                    <div>Postal Address</div>
                    <input name="address1" placeholder="Address Line 1">
                </label>
                <div> </div>
                <input name="address2" placeholder="Address Line 2">
                <div> </div>
                <input name="city" class="city" placeholder="Town/City">
                <input name="state" class="state" placeholder="State">
                <input name="zip" class="zip" placeholder="Zip Code">
                <div> </div>
                <select name="country">
                    <option value="0">Country</option>
                    <option value="US">United States</option>
                    <option value="CA">Canada</option>
                </select>
            </li>
            <li>
                <label>
                    <div>Home Phone No.</div>
                    <input type="tel" name="homephone">
                </label>
            </li>
            <li>
                <label>
                    <div>Cell Phone No.</div>
                    <input type="tel" name="cellphone">
                </label>
            </li>
            <li>
                <label>
                    <div>Skype Name</div>
                    <input name="skype">
                </label>
            </li>
            <li>
                <label>
                    <div>Twitter</div>
                    <span class="twitter_prefix">@</span>
                    <input name="twitter" class="twitter">
                </label>
            </li>
        </ul>
    </fieldset>
```

The homephone and cellphone fields both use the tel input type. Although this will make no apparent difference on a regular browser, visitors using mobile browsers will benefit from a virtual keyboard that's designed specifically for entering telephone numbers.

2.2.3 Using the form input attribute required

The Login Details form section is the most unremarkable part of the form. It asks the user to enter an account password and to enter it a second time to confirm. The markup doesn't introduce any new HTML5 features, but later in this chapter you'll learn how to use the Constraints Validation API to provide password confirmation.

STEP 3: BUILD THE LOGIN DETAILS FORM SECTION

At this point, you need only to add the code from the following listing to index.html, after the end of the previous listing; then we'll move on to a more interesting section.

Listing 2.3 index.html—The Login Details form section

```
<fieldset>
    <legend>Login Details</legend>
    <ul>
        <li>
            <label class="required">
                <div>Password</div>
                <input type="password" name="password" required>
            </label>
        </li>
        <li>
            <label class="required">
                <div>Confirm Password</div>
                <input type="password" name="confirm_password" required>
            </label>
        </li>
    </ul>
</fieldset>
```

Both the password and confirm_password fields are required fields.

2.2.4 Building a calculator-style form using the input type number, the input attributes min/max and data-*, and the element <output>

If you look at figure 2.5, you'd be forgiven for thinking there's not much HTML5 form functionality in the Order Details section.

However, several HTML5 features are at work in the Order Details section of the form:

- The `number` input type for the quantity input fields
- The `min` and `max` attributes for validating the `quantity` inputs
- The `data-*` attributes for storing price data
- The `<output>` element for displaying totals

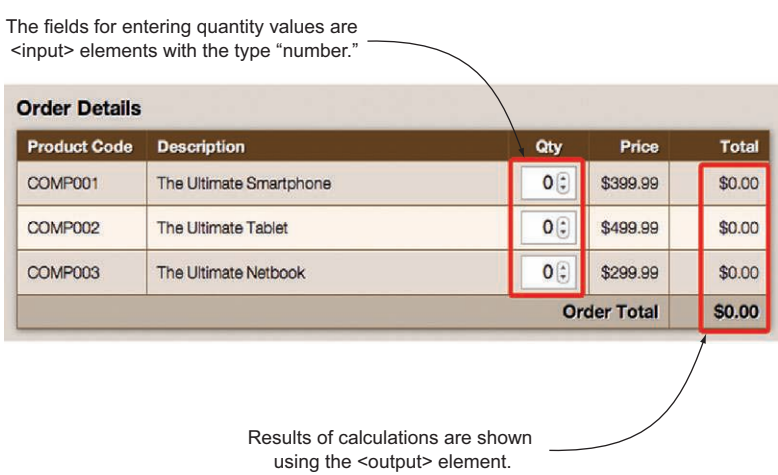

Figure 2.5 There's more going on here than meets the eye. This simple-looking form uses several HTML5 features: the `number` input type, `min` and `max` attributes, `data-*` attributes, and the `<output>` element.

THE NUMBER INPUT TYPE

`number` input type	10.0	N/A	10.0*	11.0**	5.2

* Indicates partial support; although IE10 does support validation of the `number` input type, it doesn't display a spinbox widget for the field.
** Opera 11 correctly displays a spinbox widget for picking a number but doesn't enforce numeric validation on the field.

Core API

The `number` input type should display a new UI widget on supported browsers—a spinbox component that allows the user to change the value by pressing the up button to increase the value and the down button to decrease the value. An example of this is shown in figure 2.6.

Figure 2.6 The `number` input type allows the user to increment and decrement the field value using the up and down buttons in the spinbox on the right-hand side of the field. The user can also change the value by typing a numeric value into the text field itself.

Core API

Two other new attributes that go hand in hand with the `number` input type are the `min` and `max` attributes.

THE MIN AND MAX ATTRIBUTES

`min` and `max` attributes	6.0	N/A	10.0	10.6	5.0

These attributes define the minimum and maximum numbers that a user can enter in a `number` input field (or the bounds of a slider input using the `range` input type). Yet `data-*` attributes, another new form of attribute, allow an elegant solution for automatically calculating updated totals when users enter numbers.

DATA-* ATTRIBUTES

Core API

The order form you're building will allow a user to enter a quantity for each of the products in the form. The form should then automatically calculate the total price for this item and the total order price. In order to do this, you'll need to know the price of a given item. In the past, you may have inserted a hidden field in each row to hold the price for that item, or perhaps you would have stored the price data in a JavaScript array and performed a lookup to get the price for a given product. Neither solution is elegant—that's where HTML5 `data-*` attributes come into play.

`data-*` attributes	7.0	6.0	N/A	11.1	5.1

HTML5 `data-*` attributes allow you to bind arbitrary key/value pair data to any element. JavaScript can then read this data to perform calculations and further client-side manipulation.

Using `data-*` attributes is simple: prefix the key with `data-` to form the attribute name and assign it a value. In this example, you're binding a price to a quantity input field:

```
<input type="number" data-price="399.99" name="quantity">
```

You can then listen to this field for changes and multiply the value of the user's input (the quantity) by the value of the `data-price` attribute to calculate the total price of the item. You'll see how to retrieve `data-*` attribute values a little later. First, we want to talk about the final feature we're introducing in this section: the new `<output>` element.

THE <OUTPUT> ELEMENT

Core API

The name of this element explains its purpose—it's used to display output to the user. A typical use case for the `<output>` element is displaying the result of a calculation based on some data, such as that entered by a user in an `<input>` element. You'll learn how to update the value of the `<output>` element later on, as the work progresses. For now, you'll add these new features to your application code.

STEP 4: CREATE THE ORDER DETAILS FORM SECTION

The following listing contains the code for the Order Details section. Let's put the `number` input type, `min/max` attributes, `data-*` attribute, and `<output>` element to work. Notice how these new features can simplify programming tasks for HTML5-compatible browsers. Add this code directly after the code from the previous listing.

Listing 2.4 index.html—The Order Details form section

```
<fieldset>
    <legend>Order Details</legend>
    <table>
        <thead>
        <tr>
            <th>Product Code</th><th>Description</th><th>Qty</th>
            <th>Price</th><th>Total</th>
        </tr>
        </thead>
        <tbody>
        <tr>
            <td>
                COMP001
                <input type="hidden" name="product_code" value="COMP001">
```

```
            </td>
            <td>The Ultimate Smartphone</td>
            <td>
                <input type="number" data-price="399.99" name="quantity"
                  value="0" min="0" max="99" maxlength="2">
            </td>
            <td>$399.99</td>
            <td>
                <output name="item_total" class="item_total">$0.00</output>
            </td>
        </tr>
        <tr>
            <td>
                COMP002
                <input type="hidden" name="product_code" value="COMP002">
            </td>
            <td>The Ultimate Tablet</td>
            <td>
                <input type="number" data-price="499.99" name="quantity"
                  value="0" min="0" max="99" maxlength="2">
            </td>
            <td>$499.99</td>
            <td>
                <output name="item_total" class="item_total">$0.00</output>
            </td>
        </tr>
        <tr>
            <td>
                COMP003
                <input type="hidden" name="product_code" value="COMP003">
            </td>
            <td>The Ultimate Netbook</td>
            <td>
                <input type="number" data-price="299.99" name="quantity"
                  value="0" min="0" max="99" maxlength="2">
            </td>
            <td>$299.99</td>
            <td>
                <output name="item_total" class="item_total">$0.00</output>
            </td>
        </tr>
        </tbody>
        <tfoot>
        <tr>
            <td colspan="4">Order Total</td>
            <td>
                <output name="order_total" id="order_total">$0.00</output>
            </td>
        </tr>
        </tfoot>
    </table>
</fieldset>
```

Use a number input type to allow the user to enter a quantity for each product in the order form. The minimum value is set to zero using the min attribute, whereas the max is set to 99. Each field has a data-* attribute, data-price, which holds the price of the product.

The <output> element will store the line total for each product and has a default value of $0.00.

2.2.5 Using the form input type month and input attribute pattern

The Payment Details section of the form asks users to enter their credit card details—the name on the card, the card number, the expiry date, and the CVV2 security code, found on the back of most cards. These fields use some of the HTML5 form features introduced in the Contact Details section: `required` and `placeholder` input attributes. The Payment Details section also uses some new features: the `pattern` input attribute and the `month` input type.

THE MONTH INPUT TYPE

`month` input type	N/A	N/A	N/A	9.0	N/A

Core API

The `month` type allows the user to select a month and year combination from a date-picker widget. HTML5 defines a number of date-related types: `date`, `datetime`, `datetime-local`, `month`, `week`, and `time`. Browser support for these widgets and validation has been slow moving—with the exception of Opera, which has had good support for these types for quite some time, albeit with an ugly date-picker widget, as shown in figure 2.7.

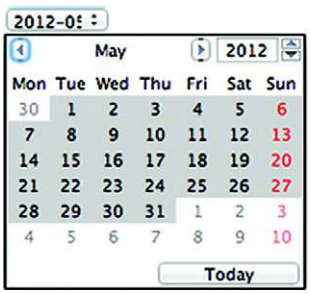

Figure 2.7 The Opera date-picker widget is used for all date/time input types, including `month`, as shown in this screenshot. When the date picker is used to select a month, clicking any day in the month will select that month.

Later in the chapter you'll learn how to provide a fallback for the `month` input type, which gives users something a little more intuitive than a plain text box to enter a month value.

THE PATTERN ATTRIBUTE

`pattern` attribute	10.0	4.0	10.0	11.0	N/A

Core API

The `pattern` attribute allows you to specify a regular expression pattern to test against data input in a field. In the order form, we'll use the `pattern` attribute on both the card_number and card_cvv2 fields to ensure they're numeric and of appropriate length.

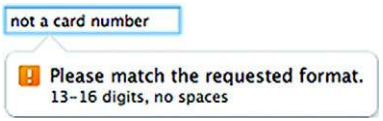

Figure 2.8 When the pattern matching fails, the browser will pick up extra information about the format required from the `title` attribute. If it's provided, tag it onto the end of the error message displayed to the user.

When using the `pattern` attribute, you can give a hint to your users about what format your data field requires by using the `title` attribute. This hint will be shown to users in a tooltip when they move their mouse over the field, but it will also be appended to the error message if users enter an invalid value in the field, as shown in figure 2.8.

STEP 5: BUILD THE PAYMENT DETAILS FORM SECTION

Let's add those two new features: the month-picker to give the user a quick and easy way to enter dates and the `pattern` attribute to define valid data patterns. Add the code from the following listing to index.html, directly after the code from the previous listing.

Listing 2.5 index.html—The Payment Details form section

```
<fieldset>
    <legend>Payment Details</legend>
    <ul>
        <li>
            <label class="required">
                <div>Name on Card</div>
                <input name="card_name" required>
            </label>
        </li>
        <li>
            <label class="required">
                <div>Credit Card No.</div>
                <input name="card_number" pattern="[0-9]{13,16}"
                  maxlength="16" required title="13-16 digits, no spaces">
            </label>
        </li>
        <li>
            <label class="required">
                <div>Expiry Date</div>
                <input type="month" name="card_expiry" maxlength="7"
                  placeholder="YYYY-MM" required value="2015-06">
            </label>
        </li>
        <li>
            <label class="required">
                <div>CVV2 No.</div>
                <input name="card_cvv2" class="cvv" maxlength="3"
                  pattern="[0-9]{3}" required title="exactly 3 digits">
                <span>(Three digit code at back of card)</span>
            </label>
        </li>
    </ul>
</fieldset>
```

The regular expression in the card number field specifies that the value should be numeric and between 13 and 16 characters in length. The title attribute is used to give users more detail about the field's requirements, should they attempt to submit an invalid value.

The expiry date for the card uses the month input type, which displays a date-picker widget on supported browsers and should validate based on the format mask YYYY-MM.

The CVV2 security code uses a pattern attribute and title hint to specify that the field value should contain exactly three numeric characters.

TRY IT OUT!

You should be able to use the form now. In browsers with good support for HTML5's form features, you'll be able to see the new input types, attributes, and validation functionality in action. In the next section, we'll allow users to choose whether they want to save the data in the form for later completion or to submit it right away.

2.2.6 Allowing users to choose whether to save or submit a form: using the input attributes formnovalidate and formaction

When users are filling out a form, they may not be able to complete the form in one session; you need to provide them with a means of saving their progress and returning to the form later. Because a user may need to leave the form quickly, forcing them to correct any validation errors before saving doesn't make sense; this is required only when the form is finally submitted. Therefore, you need to give the user a way to bypass validation.

You can force an entire form to bypass validation using the new `novalidate` attribute on the form itself. This is useful only if you want to use the new HTML5 form widgets but don't want to use any of the new validation features. An alternative approach is to have a separate button for saving progress, which uses the `formnovalidate` attribute to prevent the form from being validated when it's used. In addition, you may want to change the `formaction` property of the form to call a different URL when saving the data rather than submitting it. You can do this in HTML5 with the `formaction` attribute.

STEP 6: BYPASS FORM VALIDATION AND SAVE FORM DATA

Let's change the order form's Save Order button to make use of these new attributes:

- Find the line in index.html that reads

  ```
  <input type="submit" id="saveOrder" value="Save Order">
  ```

  Replace that line with the following:

  ```
  <input type="submit" id="saveOrder" value="Save Order" formnovalidate
    formaction="/save">
  ```

- Open the Order Form page in IE10 (and higher) and leave all the fields blank.
- Click the Submit Order button, and an error message will pop up on the Name field telling you that this field must be filled out.
- Click the Save Order button, and you'll notice that the validation will no longer be performed, and the URL the form has been submitted to will be /save rather than /submit.

That was easy, huh? Unfortunately, it's not all that simple, because this won't work on browsers that don't support these new attributes. Thankfully, with a little bit of JavaScript you can fix this problem.

STEP 7: CHANGE THE FORM ACTION IN OLDER BROWSERS

On older browsers, the application should also be able to change the form action. When the user submits the form, it should call a different URL than when saving the data.

Create a new file named app.js in the same directory as the index.html file. Add the contents of the next listing to this file.

Listing 2.6 app.js—Changing the form action in older browsers

```
(function() {
    var init = function() {
        var orderForm = document.forms.order,
            saveBtn = document.getElementById('saveOrder'),
            saveBtnClicked = false;

        var saveForm = function() {
            if(!('formAction' in document.createElement('input'))) {
                var formAction = saveBtn.getAttribute('formaction');
                orderForm.setAttribute('action', formAction);
            }
            saveBtnClicked = true;
        };

        saveBtn.addEventListener('click', saveForm, false);
    };

    window.addEventListener('load', init, false);
})();
```

When users click the Save button, check if their browser supports the formaction attribute.

If the browser doesn't support formaction, manually set the action attribute on the form using the setAttribute method.

This flag will be used later in the chapter when you provide fallback validation for browsers that don't support HTML5 validation.

If you open the page in a browser that doesn't support the `formaction` attribute, such as IE9, clicking the Submit Order button submits the form to the /submit URL. Under the same initial conditions, clicking the Save Order button submits the form to the /save URL. You'll also notice that the validation doesn't work; don't worry, you'll add a fallback for that later in the chapter.

PROGRESS CHECK

To this point, you've created the major pieces of the form: the Contact Details, Login Details, Order Details, and Payment Details sections. Using new HTML5 input types, such as `email` or `tel`, the new input attributes such as `required`, and the general `data-*` attribute and `<output>` element, can simplify coding for some browsers. Another tedious task, the implementation of bypassing data validation when saving an incomplete form, can be simplified for browsers that support the new input attributes `formnovalidate` and `formaction`.

In the next section, you'll implement the computational logic behind the Order Details section, taking the quantity values entered by the user, then calculating and displaying the total values for each item and the entire order.

2.3 Calculating totals and displaying form output

In the previous section, you used `data-*` attributes to associate key/value pair data with the quantity field, and you added `<output>` elements to the totals for each product and for the order total. Yet, in its present state, the order form doesn't seem to care what values you enter for the quantity fields—the total amounts are always $0.00.

In this section, you'll learn

- How to read input values in numeric format using the `valueAsNumber` property
- How to access data from HTML5 `data-*` attributes
- How to update the `<output>` element

In this section, you'll use the `data-*` attributes and `<output>` element to calculate the totals and output the results to the user's browser. Four steps will get you there:

- Step 1: Add functions to calculate total values.
- Step 2: Retrieve the value of quantity input fields.
- Step 3: Retrieve price values and calculate line and form totals.
- Step 4: Display updated totals on the order form.

2.3.1 Building calculation functions

We'll start by building the functions that will perform the calculations in the order form example.

STEP 1: ADD FUNCTIONS TO CALCULATE TOTAL VALUES

The code in listing 2.7 gets the relevant fields (quantity, item total, and order total) from the DOM and sets up an event listener on each of the quantity fields to calculate the totals whenever the user modifies the quantity value. The calculation code isn't shown in the listing; you'll add that later in the chapter.

Open app.js and add the following code to the end of the `init` function, below the line `saveBtn.addEventListener('click', saveForm, false);`.

Listing 2.7 app.js—Functions to calculate total values

```
var qtyFields = orderForm.quantity,
    totalFields = document.getElementsByClassName('item_total'),
    orderTotalField = document.getElementById('order_total');

var formatMoney = function(value) {
    return value.toString().replace(/\B(?=(\d{3})+(?!\d))/g, ",");
}

var calculateTotals = function() {
    var i = 0,
        ln = qtyFields.length,
```

Returns a number formatted for currency, using a comma as a 1,000 separator character.

Calculates the totals for each item and the overall order total.

```
            itemQty = 0,
            itemPrice = 0.00,
            itemTotal = 0.00,
            itemTotalMoney = '$0.00',
            orderTotal = 0.00,
            orderTotalMoney = '$0.00';

        for(; i<ln; i++) {
        }
};

calculateTotals();

var qtyListeners = function() {
    var i = 0,
        ln = qtyFields.length;

    for(; i<ln; i++) {
        qtyFields[i].addEventListener('input', calculateTotals, false);
        qtyFields[i].addEventListener('keyup', calculateTotals, false);
    }
};

qtyListeners();
```

You'll add calculation code in this for loop later in the section.

Perform an initial calculation, just in case any fields are prepopulated. Because the init function is called on page load, any prepopulated data will be ready for access.

Calls the qtyListeners function to add event listeners to fields.

The input event doesn't detect backspace or delete keystrokes or cut actions in IE9, so bind to the keyup event as well.

We'll now look at `valueAsNumber`, a new HTML5 property that allows you to get a numeric representation of the value of an input field element.

THE VALUEASNUMBER PROPERTY

Core API

The `value` property of an input element like `qtyFields[i]` allows you to read the current value of that element in JavaScript. But this value is always returned as a string. If you needed to convert the value to a floating-point number, you likely used `parseFloat`, but HTML5 has provided a new solution, the `valueAsNumber` property.

When you read the `valueAsNumber` property of a `number` input type, the property returns the number as a floating-point number. If you assign a floating-point number to the `valueAsNumber` property of a `number` input type, the property will convert the floating-point number to a string-based value.

Using valueAsDate for date and time fields

In the case of date/time fields, there is a property, `valueAsDate`, that works much like the `valueAsNumber` property. When you use it to retrieve the value of a date-oriented field, it will return a `Date` object. Similarly, you can use the property to set the value of the field to a `Date` object.

The `valueAsNumber` property should be available on browsers that support the new `number` input type—but what if the browser doesn't support this and has fallen back to a regular text input? In this case, you can fall back on the JavaScript `parseFloat`

function. The following statements are equivalent for reading the floating-point value of a field:

```
value = field.valueAsNumber;          //HTML5 version
value = parseFloat(field.value);      //Fallback version
```

Similarly, the following statements provide the same result when modifying the floating-point value of a field:

```
field.valueAsNumber = value;          //HTML5 version
field.value = value.toString();       //Fallback version
```

Why use valueAsNumber instead of parseFloat?

At this point, you may be wondering why you'd use `valueAsNumber` at all, when you can use `parseFloat` instead, and it'll work consistently across all browsers. `valueAsNumber` offers a more concise way to convert values between string and floating-point. Also, using `valueAsNumber` over `parseFloat` could lead to a tiny increase in performance, but this is unlikely to be noticeable in most web applications. When the usefulness of `valueAsNumber` was questioned on a W3C mailing list, HTML5 editor Ian Hickson provided a use case where the `valueAsNumber` property was much more concise than `parseFloat`—incrementing the value of a field programmatically. Here's an example:

```
field.valueAsNumber += 1;                                  //HTML5 version
field.value = (parseFloat(field.value) + 1).toString()     //Fallback
                                                           version
```

STEP 2: RETRIEVE THE VALUE OF QUANTITY INPUT FIELDS

In the order form example, you'll use the `valueAsNumber` property to get the value of the quantity fields for each product row in the Order Details section. Inside the empty `for` loop from listing 2.7, add the following code.

Listing 2.8 app.js—Getting the value of the quantity input fields

valueAsNumber isn't available in older browsers, so fall back to use parseFloat.

```
if(!!qtyFields[i].valueAsNumber) {
    itemQty = qtyFields[i].valueAsNumber || 0;
} else {
    itemQty = parseFloat(qtyFields[i].value) || 0;
}
```

Testing for existence of valueAsNumber property. The !! is used to cast the property valueAsNumber to a Boolean type. The first ! negates the truthness of the property and converts it to a Boolean. The second ! converts the Boolean to its original truth state.

Next you'll learn how to read HTML5 `data-*` attribute values to get the prices for each of the items, and then you'll implement it in the sample application.

2.3.2 Accessing values from HTML5 data-* attributes

Earlier, you learned how to bind key/value pair data to elements using the new `data-*` attributes in HTML5. This information is useful when you want to add extra

data to an element that can be easily picked up and used in your JavaScript code. It's straightforward to read `data-*` attributes—each element has a `dataset` property that contains all of the `data-*` attributes for that element. Each of the items in the `dataset` property has a key name that matches the key name in the element markup, with the `data-` prefix dropped. In listing 2.4 you defined the item's price using the `data-price` attribute. To retrieve that value, you can use the following code:

```
var price = element.dataset.price;
```

> **WARNING** If you hyphenate your `data-*` attribute names, they'll be camelcased in the `dataset` property. For example, if you use the attribute name `data-person-name`, you'd read this using `element.dataset.personName` rather than `element.dataset.person-name`.

The `dataset` *property* is new in HTML5, but it's not yet supported in all browsers. Thankfully, we can show you an easy fallback that'll work on all modern browsers (yes, even IE6)—the `getAttribute` *method*. To get the value of the `data-price` attribute using this fallback, you'd use the following code:

```
var price = element.getAttribute('data-price');
```

STEP 3: RETRIEVE PRICE VALUES AND CALCULATE LINE AND FORM TOTALS

In the order form example, let's add some code to get the price of each item and use it to calculate the total cost for each line by multiplying the quantity by the price, as well as the total cost for the entire order. Add the code from the following listing right below the code from the previous listing and before the terminating bracket of the `for` loop.

Listing 2.9 app.js—Getting the price values using `data-*` attributes

```
if(!!qtyFields[i].dataset) {
    itemPrice = parseFloat(qtyFields[i].dataset.price);
} else {
    itemPrice = parseFloat(qtyFields[i].getAttribute('data-price'));
}

itemTotal = itemQty * itemPrice;
itemTotalMoney = '$'+formatMoney(itemTotal.toFixed(2));
orderTotal += itemTotal;
orderTotalMoney = '$'+formatMoney(orderTotal.toFixed(2));
```

Fall back to getAttribute if the dataset property isn't available.

Now that you've calculated the totals for each item and the overall order total, all that's left is to display these values on the form using `<output>` elements. By writing values to the `<output>` element in browsers that support the `<output>` element, you can access it through the form, for example:

```
var element = document.forms.formname.outputname;
```

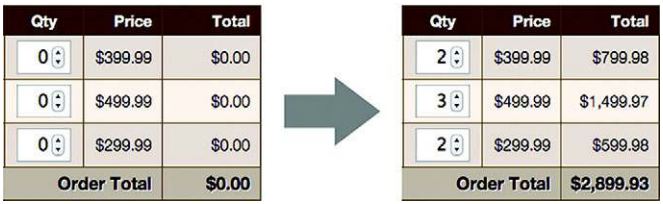

Figure 2.9 When the user enters a quantity for an item, the application multiplies it by the price for that item to get the total, then adds up all totals to get the overall order total.

To update the value of an `<output>` element, you can set the `value` property:

```
element.value = newValue;
```

What to do for browsers that don't support <output>

To access the element in browsers that don't support `<output>`, you'll need to give the element an ID and use `document.getElementById` instead:

```
var element = document.getElementById('outputid');
```

To update the value of the element, set the `innerHTML` property:

```
element.innerHTML = newValue;
```

Let's add the code you need to update the totals in your order form example.

STEP 4: DISPLAY UPDATED TOTALS ON THE ORDER FORM

Add the code from the next listing to app.js, right after the code from the previous listing and before the terminating bracket of the `for` loop.

Listing 2.10 app.js—Displaying updated totals using the `<output>` element

```
if(!!totalFields[i].value) {
    totalFields[i].value = itemTotalMoney;
    orderTotalField.value = orderTotalMoney;
} else {
    totalFields[i].innerHTML = itemTotalMoney;
    orderTotalField.innerHTML = orderTotalMoney;
}
```

Test if the <output> element is supported by the user's browser.

TRY IT OUT!

At this point, the calculation of item line and overall order total values should be working. Load the page in a modern browser and try changing the value of the quantity fields—you should notice the totals change accordingly. This is demonstrated in the screenshot in figure 2.9.

Your form now has the ability to compute totals and validate data, but what if you want to provide additional validation functions with custom error messages? In the next section, you'll extend the validation of the form to perform custom validation using the Constraint Validation API and to style invalid fields using CSS3.

2.4 Checking form input data with the Constraint Validation API

Earlier in the chapter, you learned about some of HTML5's new validation features—the `required`, `pattern`, and `min` and `max` attributes—that enable the browser itself to perform native validation on form input fields without requiring any additional JavaScript. These attributes are only the beginning when it comes to HTML5 validation—the Constraint Validation API offers many more possibilities.

In this section, you'll learn

- How to use validation properties and methods to design custom validation tests
- How to use the `invalid` event to detect invalid fields on a submitted form
- How to use the new pseudo-class selectors in CSS3 to apply styling to invalid fields without adding redundant class names to your input elements

The Constraint Validation API defines new properties and methods you can use to detect and modify the validity of a given element. Using this API, you can provide additional validation functionality and use custom error messages. The API allows you to detect whether a field has an error and, if so, what type of error and what error message you'll display. It also provides a method that allows you to set your own custom validation message that will be displayed natively by the browser.

Constraint Validation API	10.0	4.0	10.0	10.0	5.0*

* Indicates partial support; although Safari 5.0 supports the Constraints Validation API, it doesn't currently enforce it automatically and display inline error messages like other browsers do.

In this section, as you continue working on this chapter's sample application, you'll walk through two steps:

- Step 1: Add custom validation and error messages to input fields.
- Step 2: Detect form validation failure.

Also, although you won't have to do any coding because we've already provided the full CSS file for the sample application, at the end of the section we'll show you how to style invalid fields using CSS so you'll be prepared to do so in your own apps. First up, let's explore and use some of the Constraint Validation API's properties and methods.

2.4.1 Creating custom validation tests and error messages with the setCustomValidity method and the validationMessage property

Core API

When a validation function isn't supported by a browser or by HTML5, the application will have to implement a custom validation test. In these cases, you'll have to write some JavaScript to test the validity of the entered data and provide a custom error message when the validation fails. The Constraint Validation API simplifies the implementation of custom error messages by providing a `setCustomValidity` method and a `validationMessage` property. Both constructs allow the application to assign an error message to the `<input>` element's `validationMessage` attribute. Determining which construct to use will depend on the browser's support for `setCustomValidity`.

STEP 1: ADD CUSTOM VALIDATION AND ERROR MESSAGES TO INPUT FIELDS

The order form example will perform custom validation for a number of tests using the `setCustomValidity` method:

- Full Name must be at least four characters long.
- Password must be at least eight characters long.
- Password and Confirm Password must match.
- Name on Card must be at least four characters long.

Let's add this custom validation to the app.js file. Add the code from this listing to the end of the `init` function, directly after the call to `qtyListeners`.

Listing 2.11 app.js—Performing custom validation

```
var doCustomValidity = function(field, msg) {
    if('setCustomValidity' in field) {
        field.setCustomValidity(msg);
    } else {
        field.validationMessage = msg;
    }
};

var validateForm = function() {
    doCustomValidity(orderForm.name, '');
    doCustomValidity(orderForm.password, '');
    doCustomValidity(orderForm.confirm_password, '');
    doCustomValidity(orderForm.card_name, '');

    if(orderForm.name.value.length < 4) {
        doCustomValidity(
            orderForm.name, 'Full Name must be at least 4 characters long'
        );
    }

    if(orderForm.password.value.length < 8) {
        doCustomValidity(
            orderForm.password,
            'Password must be at least 8 characters long'
        );
    }
```

Check if the browser supports the setCustomValidity method; if not, manually set the value of validationMessage.

Perform custom validation check; if that fails, call the doCustomValidity function.

Perform custom validation check; if that fails, call the doCustomValidity function.

```
    if(orderForm.password.value != orderForm.confirm_password.value) {
        doCustomValidity(
            orderForm.confirm_password,
            'Confirm Password must match Password'
        );
    }

    if(orderForm.card_name.value.length < 4) {
        doCustomValidity(
            orderForm.card_name,
            'Name on Card must be at least 4 characters long'
        );
    }
};

orderForm.addEventListener('input', validateForm, false);
orderForm.addEventListener('keyup', validateForm, false);
```

The keyup event binding is required to detect backspace, delete, and cut actions in IE9.

TRY IT OUT!

If you load the form in a compatible browser and try to break the custom validation rules described previously, you'll notice that the custom error message will be displayed to the user, as illustrated in figure 2.10.

Next you'll use the `invalid` event, which fires any time the user tries to submit a form with one or more fields that are marked as invalid.

2.4.2 Detecting a failed form validation with the invalid event

Core API

When the user attempts to submit a form that uses HTML5 validation features, the `submit` event will only fire if the entire form has passed the validation tests. If you need to detect when form validation has failed, you can listen for the new `invalid` event. This event is fired when one of the following occurs:

- The user attempts to submit the form and validation fails.
- The `checkValidity` method has been called by the application and has returned `false`.

STEP 2: DETECT ORDER FORM VALIDATION FAILURE

Let's add a listener to the `invalid` event in the order form. Add the following code directly after the code from the previous listing.

Figure 2.10 A demonstration of custom validation in action. In this case, the user has entered a valid password (at least eight characters in length), but they've entered a value in the Confirm Password field that doesn't match the value in the Password field. This causes the error "Confirm Password must match Password" to be displayed.

Listing 2.12 app.js—Listening to the `invalid` event

```
var styleInvalidForm = function() {
    orderForm.className = 'invalid';
}

orderForm.addEventListener('invalid', styleInvalidForm, true);
```

Add a class invalid to the <form> element. You'll use this in the next section to style invalid fields on a submitted form.

Listens to the invalid event on the form and all other elements in the form.

The `invalid` event is useful if you want to apply styling to erroneous form fields on a submitted form. You'll learn how to do that next.

2.4.3 Styling invalid elements using CSS3 pseudo-classes

One way to style invalid elements would be to iterate over the fields, checking if each one is invalid and applying CSS classes to those that have errors. But this is a bit cumbersome, and you can do this much more elegantly using a bit of CSS3 magic.

Core API

CSS3 introduces a range of new pseudo-classes for styling form fields based on their validity. These styles will be applied only if the condition defined by the pseudo-class is true. The following self-explanatory pseudo-classes are available:

- `:valid`
- `:invalid`
- `:in-range`
- `:out-of-range`
- `:required`
- `:optional`

As you can probably guess, pseudo-classes make styling invalid fields easy. For example, the following code would style any element declared invalid by the Constraint Validation API with a light red background and a maroon border:

```
:invalid {
    background-color: #FFD4D4;
    border: 1px solid maroon;
}
```

But this declaration has a problem: Any field that uses validation attributes like `required` or `pattern` will be initially invalid because these order form fields are blank. As a result, those fields that apply validation attributes will display a red background and maroon border, which isn't nice.

Fortunately, you can easily get around this by applying a class to the parent form when the `invalid` event has fired and adding the pseudo-class selector, `:invalid`, to the CSS rules for the input and selector elements in the form.

> **NOTE** Please don't change the CSS file that you included in your application's directory when you started the chapter. In this section, we're walking through the theoretical changes you might make rather than directing you to make changes.

In the previous section, you applied a class to the parent form. So, now add the pseudo-class selector, `:invalid`, to the CSS:

```
form.invalid input:invalid, form.invalid select:invalid,
form.invalid input.invalid, form.invalid select.invalid {
    background-color: #FFD4D4;
    border: 1px solid maroon;
}
```

The order form also uses the `:required` pseudo-class to style required fields with a light yellow background:

```
input:required, select:required {
    background-color: lightyellow;
}
```

A screenshot of the required and invalid field styling is shown in figure 2.11.

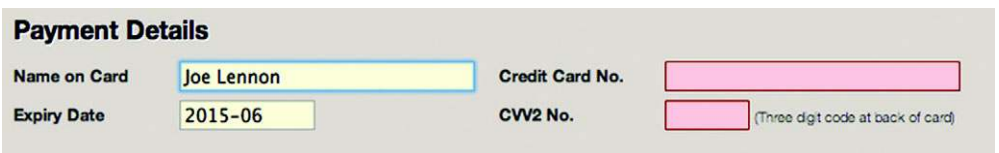

Figure 2.11 The required fields are styled with a light yellow background (left), as you can see in the Name on Card and Expiry Date fields. The invalid fields are styled with a light red background and a maroon border (right), as shown in the Credit Card No. and CVV2 No. fields.

At this point, the form is more or less fully functional for most recent versions of all browsers (with the exception of Safari). In the next section, you'll learn how to perform rock-solid feature detection using the Modernizr library and how to plug feature gaps using polyfills.

2.5 Providing fallbacks for unsupported browsers

One of the main drawbacks to using HTML5's new features is that browser support isn't uniform. Thus, you need to find ways to allow those with the latest and greatest browsers to make use of HTML5 features while ensuring that those using slightly older versions aren't left behind.

In this section, you'll learn

- How Modernizr simplifies detection of browser support for various features of HTML5 and conditionally loads fallbacks
- How to plug gaps in browser support with polyfills, a JavaScript fallback, that will only deploy if the browser lacks native support
- How to use JavaScript to implement basic fallback validation for those browsers that don't yet fully support the Constraint Validation API

You'll learn about these topics as you build out your form using these three steps:

- Step 1: Build feature detection and conditionally deploy a fallback for month-picker.
- Step 2: Build fallback constraint validation for Safari 5.1.
- Step 3: Build fallback constraint validation for IE9.

First up, though, we'd like to give you an overview of feature detection with Modernizr.

2.5.1 Detecting features and loading resources with Modernizr: an overview

An important concept when you're working with HTML5's new APIs is that of feature detection—testing to see if the browser supports a given feature. Unfortunately, the approaches for detecting feature support vary widely, making it difficult to remember how to test for each individual feature. Another issue with feature detection is that you may wish to load certain external resources only if the user's browser supports (or doesn't support) a given feature. We don't see a point, for example, to loading a large WebGL support framework if the user's browser doesn't support WebGL. In a similar way, why should we load a color-picker widget library if the user's browser includes a native widget that will be used instead? Dynamic loading of external resources is possible, but the JavaScript for doing so is hardly straightforward.

Enter Modernizr, a purpose-built JavaScript library for performing bulletproof feature detection and dynamic loading. When you include Modernizr in a web page, you can detect support for a feature using a much easier syntax. For example, to check to see if the user's browser supports the Canvas element, you'd use the following:

```
if(Modernizr.canvas) {
    //Canvas is supported, fire one up!
} else {
    //Canvas is not supported, use a fallback
}
```

To detect Canvas support without Modernizr, you'd need to use the following:

```
if(!!document.createElement('canvas').getContext) {
    //Canvas is supported, fire one up!
} else {
    //Canvas is not supported, use a fallback
}
```

It's also simple to use Modernizr to dynamically load resources (either .js or .css files) based on a feature test. Consider this example, in which Modernizr will determine if the browser supports the localStorage API. If supported, it will load the localstorage.js file, which would likely contain code that interacts with this API. Otherwise, it will load the localstorage-polyfill.js file, which contains a fallback.

```
Modernizr.load({
    test: Modernizr.localstorage,
    yep: 'localstorage.js',
    nope: 'localstorage-polyfill.js'
});
```

Moving on, let's explore the concept of a polyfill and how you can use it to plug features that aren't supported by a given browser.

2.5.2 Using polyfills and Modernizr to plug the gaps

Core API

The term *polyfill* was coined by Remy Sharp and refers to a piece of code (or shim) that aims to implement missing parts of an API specification. The origin of the term is from a product named Polyfilla, which builders use to fill gaps or cracks in walls. Likewise, we developers can use polyfills to fill the gaps or cracks in various web browsers' support for HTML5.

> **TIP** Paul Irish, one of the key contributors to the Modernizr library, edits and maintains a comprehensive list of polyfills, shims, and fallbacks for a wide variety of HTML5 features. This list is available on Modernizr's GitHub wiki at: http://mng.bz/cJhc.

STEP 1: BUILD FEATURE DETECTION AND CONDITIONALLY DEPLOY A FALLBACK FOR MONTH-PICKER

Let's look at how to use Modernizr to load a month-picker polyfill into those browsers without a built-in month-picker. We expect that you've already placed the monthpicker.js file from this chapter's source code (available at http://manning.com/crowther2) in the same directory as the files you've been building in this chapter. Now add the code from the next listing to the end of the `init` function, directly after the code you added from the previous listing.

Listing 2.13 app.js—Using the month-picker polyfill

```
Modernizr.load({
    test: Modernizr.inputtypes.month,
    nope: 'monthpicker.js'
});
```

If the user's browser doesn't support the month input type, load the monthpicker.js file.

Figure 2.12 Before the polyfill has been loaded, the Expiry Date field is represented merely by a text input. After the polyfill has been loaded, the field has been replaced with a month drop-down and a year number input field. The polyfill listens for changes to these fields and populates a hidden field, which stores the month in YYYY-MM format. This hidden field will be sent to the server when the form is submitted.

If you load the order form in any browser that doesn't natively support the `month` input type, you should see the standard text input replaced with a month drop-down and a year number input field. This is illustrated in the side-by-side screenshots in figure 2.12.

You can apply the same technique to most of the HTML5 form's functionality. In fact, several projects are in the works that aim to polyfill the entire set of forms features in HTML5. These projects include

- Webshims Lib by Alexander Farkas (http://afarkas.github.com/webshim/demos/)
- H5F by Ryan Seddon (https://github.com/ryanseddon/H5F)
- Webforms2 by Weston Ruter (https://github.com/westonruter/webforms2)
- html5Widgets by Zoltan "Du Lac" Hawryluk (https://github.com/zoltan-dulac/html5Forms.js)

Let's wrap up this chapter by performing some basic validation, even on browsers that don't support the Constraint Validation API.

2.5.3 Performing validation without the Constraint Validation API

Core API

If you run the order form example in Safari 5.1 or older versions of other browsers (such as IE9), you'll notice that the validation functionality doesn't work—the form will submit without performing any validation. In this section, you'll learn how to use JavaScript to perform this validation and, if any errors are found, prevent submission of the form.

STEP 2: BUILD FALLBACK CONSTRAINT VALIDATION FOR SAFARI 5.1

In the case of Safari 5.1, the Constraint Validation API is partially supported. This means if you have an `<input>` element in your form with the `required` attribute set, the element wouldn't pass validation in Safari 5.1. But Safari doesn't implement any of the UI features, such as displaying error messages next to invalid fields, nor does it prevent the form from submitting if errors exist in the form. Let's start off by reversing this and displaying an error message to the user if there are errors. Add the code from the following listing to your app.js file, right after the code from the previous listing.

Listing 2.14 app.js—Preventing an invalid form from submitting in Safari 5.1

```
var getFieldLabel = function(field) {
    if('labels' in field && field.labels.length > 0) {
        return field.labels[0].innerText;
    }
    if(field.parentNode && field.parentNode.tagName.toLowerCase()=== 'label')
     {
        return field.parentNode.innerText;
    }
    return '';
}

var submitForm = function(e) {
    if(!saveBtnClicked) {
        validateForm();
        var i = 0,
            ln = orderForm.length,
            field,
            errors = [],
            errorFields = [],
            errorMsg = '';

        for(; i<ln; i++) {
            field = orderForm[i];
            if((!!field.validationMessage &&
              field.validationMessage.length > 0) || (!!field.checkValidity
              && !field.checkValidity())
            ) {
                errors.push(
                    getFieldLabel(field)+': '+field.validationMessage
                );
                errorFields.push(field);
            }
        }

        if(errors.length > 0) {
            e.preventDefault();

            errorMsg = errors.join('\n');

            alert('Please fix the following errors:\n'+errorMsg, 'Error');
            orderForm.className = 'invalid';
            errorFields[0].focus();
        }
    }
};

orderForm.addEventListener('submit', submitForm, false);
```

This function retrieves the label for a field using either the labels property or by checking if the field's parent element is a label.

You previously added an event to the Save Order button. When it's clicked, a saveBtnClicked flag is marked as true. This flag is used to determine whether or not the form should be validated.

Loop through the fields in the order form and check if each field is valid.

If the checkValidity method is available and returns false, or if the validationMessage property is populated, then the field contains an error and should be pushed into the errors and errorFields arrays.

If there are errors, this stops the form from submitting and alerts the user with the errors that have been found. Also, this adds the class invalid to the order form to ensure invalid fields are styled correctly and sets the focus on the first invalid field.

If you load the form in Safari and try to submit with invalid fields, you'll get an error message like the one shown in figure 2.13, and the invalid fields will highlight in red. This isn't the prettiest way to inform your users of errors—in practice you'd probably try to mimic the behavior of one of the other browsers by showing an error bubble next to the first error that's encountered.

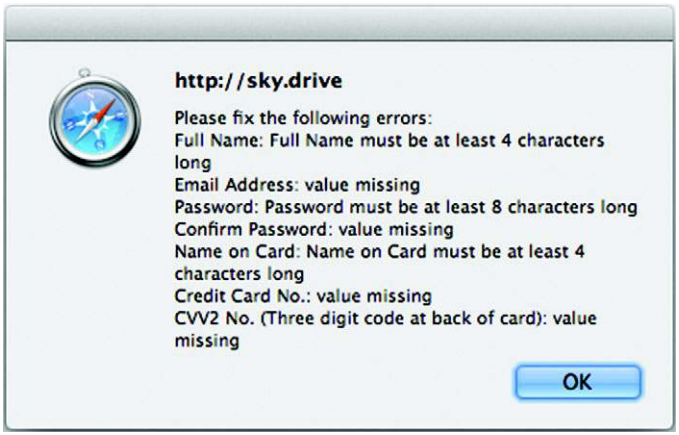

Figure 2.13 Safari now validates the form, displaying a generic alert dialog box with a list of errors that the user needs to correct. You'll notice that for each invalid field, the field's label has been picked up along with the relevant error message that's to be displayed to the user.

STEP 3: BUILD CONSTRAINT FALLBACK VALIDATION FOR IE9

You need to solve one last issue. If you try to submit the form in IE9, you'll see error messages if any input fields don't pass the custom validation tests you wrote earlier. This is great, but IE9 doesn't support the standard attribute-based validation parameters or the `email` input type. To fix this, you need to create a function to scan the form for the input field attributes `required` and `pattern` and input type `email`. When the app has collected those fields, you'll test their validity. Add the code from the next listing to app.js, directly after the code from the previous listing.

Listing 2.15 app.js—Fallback validation in IE9

```
var fallbackValidation = function() {
    var i = 0,
        ln = orderForm.length,
        field;

    for(;i<ln;i++) {
        field = orderForm[i];
        doCustomValidity(field, '');

        if(field.hasAttribute('pattern')) {
            var pattern = new
                RegExp(field.getAttribute('pattern').toString());
            if(!pattern.test(field.value)) {
                var msg = 'Please match the requested format.';
                if(field.hasAttribute('title') &&
                   field.getAttribute('title').length > 0) {
                   msg += ' '+field.getAttribute('title');
                }
                doCustomValidity(field, msg);
            }
        }
        if(field.hasAttribute('type') &&
           field.getAttribute('type').toLowerCase()=== 'email') {
```

If the pattern attribute is set, this matches its regular expression against the field's value.

If the input type is email, validate it with the defined pattern.

```
            var pattern = new RegExp(/\S+@\S+\.\S+/);
            if(!pattern.test(field.value)) {
                doCustomValidity(field, 'Please enter an email address.');
            }
        }
        if(field.hasAttribute('required') && field.value.length < 1) {
            doCustomValidity(field, 'Please fill out this field.');
        }
    }
};
```

If the required attribute is set, verify that the user has entered a value.

var pattern was chosen for brevity, not reliability. Designing a good pattern depends on many issues and exceeds this chapter's scope. To use this code, you need to call the `fallbackValidation` function when validating the form. Locate the `validateForm` function in your app.js file, and add the following snippet before the line `if(orderForm.name.value.length < 4) {`.

```
if(!Modernizr.input.required || !Modernizr.input.pattern) {
    fallbackValidation();
}
```

The snippet uses Modernizr to test whether the `required` and `pattern` attributes are supported, and if not, it calls the `fallbackValidation` function. If you run the example in IE9, you should see that the validation includes checking `required`, `pattern`, and `email`, as well as custom validation.

This fallback, Modernizr, and the month-picker polyfill are only a sample of the tools you can use to quickly provide backward compatibility in your HTML5 applications. You could easily expand on these to provide support for even older browsers such as IE6 (hint: use a library like jQuery to help with things like event handlers and DOM traversal). You shouldn't let a lack of browser support stop you from using HTML5 form features—it's easy to fill any gaps.

2.6 Summary

HTML5 gives you a lot of functionality for improving web forms. New input types like `email` and `tel` provide more widgets with less coding. Using the new input attribute, `pattern`, enables many validation tasks to be done with no JavaScript. Creating custom validation tests and error message is now much easier with the Constraint Validation API. Also, binding data to HTML elements can be done more efficiently with the `data-*` attribute.

Unfortunately, browser support is spotty, and browser vendors have been relatively slow to implement these features. Slow and partial implementation of form features appears unlikely to change anytime soon. But this shouldn't stop you from adding HTML5 form functionality to your web apps. When you have a powerful tool like Modernizr for detecting feature support and a growing list of polyfills, you have an efficient way to add HTML5 form support to your applications.

During the development of this form, you had to provide the form with a save feature. The application had no way to save the form on the client, so the application had to save the form on the server. Saving the form on the client's local system would have been a better solution; it would have delivered a faster response and required little or no server resources. And that's what you'll learn in the next chapter: how to create and save files on the client side with the File System API.

You'll also learn how to augment a form's editing functions with the Editing and Geolocation APIs. Sometimes, forms require users to add more than just plain numbers and names. For instance, text entered into a blog posting form will need special formatting (for example, **bolding** or *italics*). The Editing API has powerful constructs to quickly build in this kind of rich media support. If you need to insert a map, the next chapter will show you how to use the Geolocation API to add a localized mapping service to a web-based editor.

Chapter 3 at a glance

Topic	Description, methods, and so on	Page
Editing API	Allowing users to compose and edit HTML content ▪ `execCommand()` ▪ File Editor view markup	 81 77
Geolocation API	Providing geographic data about the user's location ▪ `getCurrentPosition()`	 82
Quota Management API	Querying local storage about availability and usage; requesting a local storage quota ▪ File System API	 85
File API	Reading file objects ▪ `readAsText()`	 89
File Writer API	Writing data to files stored with the File System API ▪ Editing files ▪ `CreateFormSubmit`	 89 91
Drag and Drop API	Using the mouse to select files for import and export ▪ Importing files using the `drop` and `dragover` events ▪ Saving files using the `draggable` attribute and `dragstart` event	 97 98

Look for this icon 

throughout the chapter to quickly locate the topics outlined in this table.