CHAPTER 6

Enhancing and Validating Forms

When you complete this chapter, you will be able to:

- Use JavaScript to reference form and form elements
- Retrieve values from selection lists
- Retrieve values from option buttons
- Format numeric values and currency values based on local standards
- Write scripts that respond to form events
- Store values in hidden fields
- Understand how web forms are submitted for validation
- Validate web form fields using customized tools
- Test a field value against a regular expression
- Create a customized validation check for credit card data
- Manage the form validation process

In this chapter you will learn how to use JavaScript to manage web forms, perform calculations based on data from those forms, report the results, and validate data entry to catch user error.

Exploring Forms and Form Elements

You have been given two web pages containing commonly used forms: an order form for calculating the cost of a purchase and a payment form for entering credit information to complete the purchase. The general code for the order form web page has already been created for you. Open the files for that page now.

To open the files for the order form:

1. Go to the js06 ▶ chapter folder of your data files.
2. Use your code editor to open the **js06a_txt.html** and **js06a_txt.js** files. Enter your name and the date in the comment section of each file and then save them as **js06a.html** and **js06a.js**, respectively.

3. Return to the **js06a.html** file in your code editor. Within the head section, add the following code to run the js06a.js script, deferring the loading the script file until after the entire page has loaded.

```
<script src="js06a.js" defer></script>
```

4. Take some time to scroll through the contents of the HTML file, noting that a web form containing several `input` elements has been enclosed within a web table.
5. Close the file, saving your changes.
6. Open the **js06a.html** file in your web browser. **Figure 6-1** shows the current layout and contents of the page.

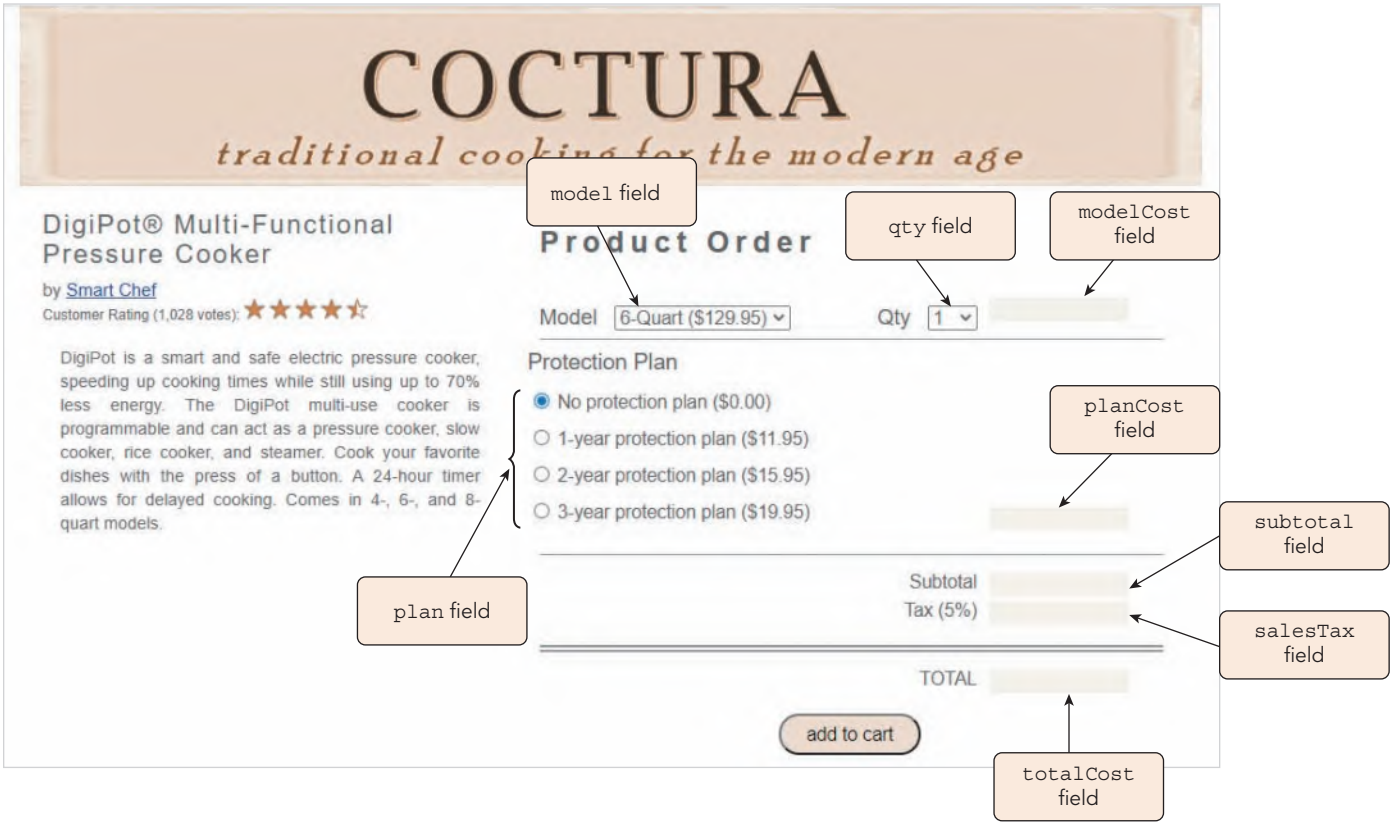

Figure 6-1 Product order form

The order form contains the following fields:

- The `model` field displayed as a selection list from which customers can choose a model to order
- The `qty` field displayed as another selection list from which customers specify the amount of the selected model to order
- The `plan` field displayed as a collection of option buttons from which customers choose the protection plan, if any, for the selected model
- The `modelCost` field calculating the cost of the model times the quantity ordered
- The `planCost` field calculating the cost of the protection plan times the quantity ordered
- The `subtotal` field calculating the sum of the `modelCost` and `planCost` fields
- The `salesTax` field calculating a 5% sales tax on the subtotal
- The `totalCost` field adding the values of the `subtotal` and `salesTax` fields

Note that some of these fields are entered by the customer and some are automatically calculated by the web form using the script you will write. Currently no calculations have been done on the web form data.

The Forms Collection

To program a web form, you work with the properties and methods of the `form` object and the elements it contains. Because a page can contain multiple web forms, JavaScript organizes forms into the following HTML collection:

```
document.forms
```

The first several forms listed in the page are referenced using the expressions `document.forms[0]`, `document.forms[1]`, and so forth. You can also reference a form using the value of the form's `name` attribute using either of the following expressions:

```
document.forms[fname]
document.forms.fname
```

where *fname* is the form's name. As always, you can reference a form using the `document.getElementById()` method if the form has been assigned an id. **Figure 6-2** describes some of the properties and methods associated with individual form objects.

PROPERTY OR METHOD	DESCRIPTION
form.action	Sets or returns the `action` attribute of the web *form*
form.autocomplete	Sets or returns the `autocomplete` attribute; allows the browser to automatically complete form fields
form.enctype	Sets or returns the `enctype` attribute
form.length	Returns the number of elements in the form
form.method	Sets or returns the `method` attribute
form.name	Sets or returns the `name` attribute
form.noValidate	Sets or returns whether the form should be validated upon submission. Use `true` for no validation, `false` to apply validation.
form.target	Sets or returns the `target` attribute
form.reset()	Resets the web form
form.submit()	Submits the web form
form.requestAutocomplete()	Triggers the browser to initiate autocompletion of those form fields that have autocomplete activated

Figure 6-2 Form properties and methods

For example, the following statement resets the form with the name `orderForm` in the current document:

```
document.forms.orderForm.reset()
```

Resetting a form will replace all current values and selections in the form with their default values and selections.

Working with Form Elements

A form and its elements are organized into a hierarchy like the one shown in **Figure 6-3**.

To reference specific elements within that hierarchy, use the following HTML collection:

```
form.elements
```

where *form* is the reference to the web form.

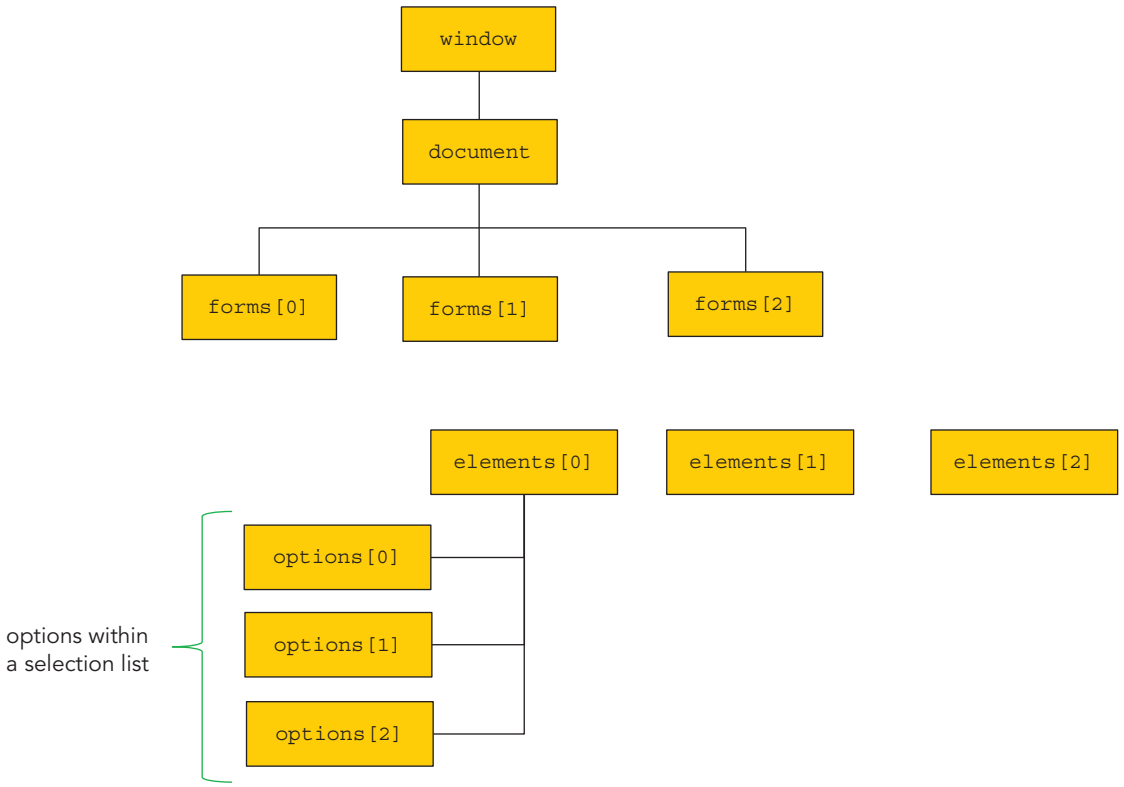

Figure 6-3 Web form hierarchy

You can reference an element using the value of the element's name attribute using the following expression:

```
form.elements[ename]
form.elements.ename
```

where *ename* is name of a field contained within the web form or the index number of the element within the `elements` collection. Fields are always associated with a web form control like an input box or selection list. For example, to reference the `model` field from the `orderForm` form, apply either of the following object references:

```
document["orderForm"].elements["model"]
document.orderForm.elements.model
```

As with the form itself, you can always reference the control associated with a field using the `document.getElementById()` method.

Note The `id` attribute for a web form element references the control that the user interacts with; the `name` attribute references the field in which the element's value is stored.

Properties and Methods of `input` Elements

A common form element is one marked with the HTML `<input>` tag. Every attribute that is associated with the `<input>` tag is mirrored by a JavaScript property. **Figure 6-4** lists some of the properties and methods associated with `input` boxes.

Thus, to set the value of the `username` field within the `orderForm` web form, apply the statement:

```
document.orderForm.username.value = "John Smith";
```

and the text string "John Smith" will appear within the input box control and stored as the value of the `username` field.

PROPERTY OR METHOD	DESCRIPTION
input.autocomplete	The value of the input box's autocomplete attribute
input.defaultValue	The default value for the input box
input.form	The form containing the input box
input.maxLength	The maximum number of characters allowed in the input box
input.name	The name of the field associated with the input box
input.pattern	The value of the input box's pattern attribute
input.placeholder	The value of the input box's placeholder attribute
input.readOnly	Returns whether the input box is read-only or not
input.required	Returns whether the input box is required or not
input.size	The value of the input box's size attribute
input.type	The data type associated with the input box
input.value	The current value displayed in the input box
input.blur()	Removes the focus from the input box
input.focus()	Gives focus to the input box
input.select()	Selects the contents of the input box

Figure 6-4 Properties and methods of input boxes

Navigating Between Input Controls

You might need your script to manage how users navigate between the controls on your form. A form control like an input box, text area box, or selection list receives the focus of the browser when it becomes active, either by moving the cursor into the control or by clicking it. A control that has received the focus is ready for data entry. To use JavaScript to give focus to a form element, apply the following property to the element:

```
element.focus()
```

where *element* is a reference to the web form control that will become active in the document. To remove focus from a form element, apply the following blur() method:

```
element.blur()
```

Note that using the blur() method to remove the focus from an element doesn't give any other element the focus. To speed up data entry, many web forms will open with an input control automatically given the focus. You will program the order form on the Coctura website to give the focus to the selection list from which a customer selects a product model to order.

To give the focus to the selection list:

1. Return to the **js06a.js** file in your code editor.
2. Directly below the initial comment section, add the following code to be run when the page is initially opened in the browser window (see **Figure 6-5**):

```
window.addEventListener("load", function() {
   let orderForm = document.forms.orderForm;
   let model = orderForm.elements.model;

   // Select Model selection list when form opens
   model.focus();
});
```

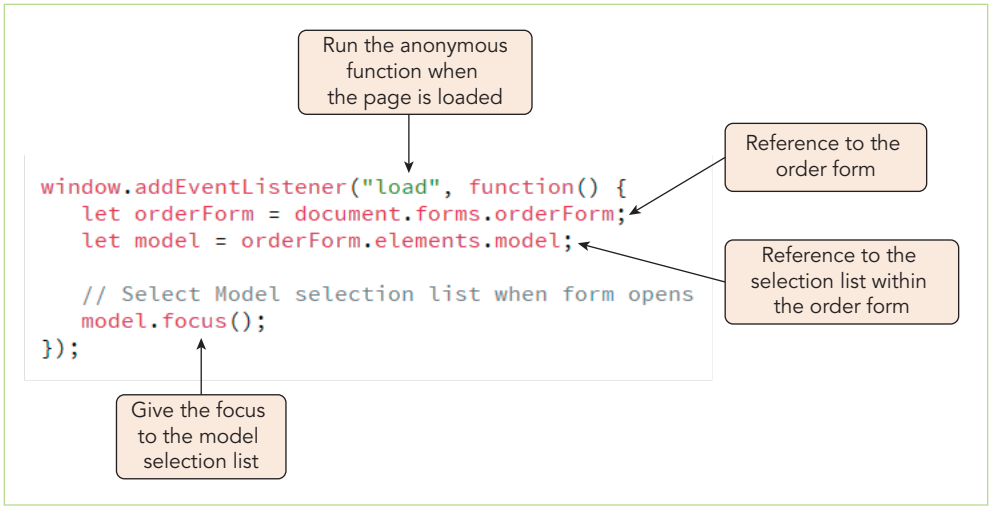

Figure 6-5 Giving the focus to the Model selection list

3. Save your changes to the file and then reload **js06a.html** in your browser.
4. Verify that the selection list box for the model field has the focus, by pressing the up and down arrows on your keyboard to change the selected mode option without having to select the selection list first.

In the rest of the form, you will calculate the cost of a customer's order. This involves (1) determining the price of the order (equal to the price of the selected model multiplied by the quantity ordered); (2) adding the cost of the protection plan, if any, for the quantity of models ordered; (3) calculating the sales tax; and (4) adding all these costs to determine the grand total. Start with a function to calculate the cost of ordering a model.

Note You can also ensure that a field gets the focus when the page loads by adding the `autofocus` attribute to element's markup tag in the HTML file.

Working with Selection Lists

Extracting a value from a form control like an input box is very straightforward: You only need to reference the `value` property of the input box. Selection lists, however, do not have a `value` property because they contain a multitude of possible options each with a different value. **Figure 6-6** describes some of the properties associated with selection lists that you will use in your program.

PROPERTY OR METHOD	DESCRIPTION
select.length	The number of options in the selection list, *select*
select.multiple	Returns `true` if more than one option can be selected from the list
select.name	The selection list field name
select.options	The object collection of the selection list *options*
select.selectedIndex	The index number of the currently selected option
select.size	The number of options displayed in the selection list
select.add(*option*)	Adds *option* to the selection list
select.remove(*index*)	Removes the option with the index number, *index*, from the selection list

Figure 6-6 Properties and methods of selection lists

The value of the field associated with a selection list is the value of the option selected by the user. Selection list options are organized into the following HTML collection:

select.options

where *select* is the reference to a selection list within the web form. As with other HTML collections, an individual option is referenced either by its index value within the collection or by the value of its `id` attribute. **Figure 6-7** describes the properties associated with individual selection list options within the `options` collection.

PROPERTY OR METHOD	DESCRIPTION
option.defaultSelected	Returns true if *option* is selected by default
option.index	The index number of *option* within the options collection
option.selected	Returns `true` if the option has been selected by the user
option.text	The text associated with *option*
option.value	The field value of *option*

Figure 6-7 Properties and methods of selection list options

To return the value from a selection list field, you must first determine which option has been selected using the `selectedIndex` property and then reference the `value` property of that selected option to determine the field's value. The following code demonstrates how to return the cost of the product chosen from the model selection list box:

```
let mIndex = model.selectedIndex;
let mValue = model.options[mIndex].value;
```

Note | If no option is selected, the `selectedIndex` property returns a value of –1.

The initial cost of the customer's order is equal to the price of the selected model multiplied by the quantity of items ordered. Because both the `model` and `qty` fields are entered as selection lists, you will retrieve the values of the two selected options. Add the code into the `calcOrder()` function, which will be nested and called within the anonymous function you created in the previous set of steps.

To create the `calcOrder()` function:

1. Return to the **js06a.js** file in your code editor.
2. Within the anonymous function add the following statement to call the `calcOrder()` function:

```
// Calculate the cost of the order
calcOrder();
```

3. Next, add the following initial code for the `calcOrder()` function described in **Figure 6-8**.

```
function calcOrder() {
   // Determine the selected model
   let mIndex = model.selectedIndex;
   let mValue = model.options[mIndex].value;

   // Determine the selected quantity
   let qIndex = orderForm.elements.qty.selectedIndex;
   let quantity = orderForm.elements.qty[qIndex].value;

   // Model cost = model cost times quantity
   let modelCost = mValue*quantity;
   orderForm.elements.modelCost.value = modelCost;
}
```

```
                    // Select Model selection list when form opens
                    model.focus();

Calculate the cost of the customer order
                    // Calculate the cost of the order
                    calcOrder();

                    function calcOrder() {
                       // Determine the selected model
Retrieve the value of the selected model
                       let mIndex = model.selectedIndex;
                       let mValue = model.options[mIndex].value;

                       // Determine the selected quantity
Retrieve the quantity ordered of that model
                       let qIndex = orderForm.elements.qty.selectedIndex;
                       let quantity = orderForm.elements.qty[qIndex].value;

                       // Model cost = model cost times quantity
Calculate the cost of ordering that model in the indicated quantity
                       let modelCost = mValue*quantity;
                       orderForm.elements.modelCost.value = modelCost;
                    }

                 });
```

Figure 6-8 Calculating the cost of models ordered

4. Save your changes to the file and then reload **js06a.html** in your browser. As shown in **Figure 6-9**, a cost of $129.95 appears in the input box as the result of multiplying the selected product (the 6-Quart pressure cooker for $129.95) by the select quantity (1).

Figure 6-9 Cost of one 6-Quart pressure cooker

To the cost of the order, add the cost of the protection plan, if any, selected by the customer. To retrieve the cost of the protection plan, you will work with the values stored in option buttons associated with the different protection options.

Programming Concepts | Selection List with Multiple Values

Some selection lists allow multiple selections. In those cases, the `selectedIndex` property returns the index of the first selected item. To determine the indices of all the selected items, create a `for` loop that runs through the options in the list, checking each to determine whether the `selected` property is `true` (indicating that the option was selected by the user). If the option is selected, it can then be added to an array of the selected options using the `push()` method. The general structure of the `for` loop is:

```
let selectedOpt = new Array();
for (let i = 0; i < select.options.length; i++) {
   if (select.options[i].selected) {
      selectedOpt.push(select.options[i]);
   }
}
```

where *select* is a selection list object. After this code runs, the `selectedOpt` array will contain all the selected options. To extract the values of the selected options, create another `for` loop that iterates through the items in the `selectedOpt` array to extract the `text` and `value` properties of each.

Working with Option Buttons

Option or radio buttons are grouped together when they share a common field name stored in their `name` attribute. For example, the four protection plan options shown in the order form all share the common name of "plan". Option buttons associated with a common field are placed within the following HTML collection:

```
form.elements.options
```

where *form* is the reference to the web form and *options* is the common field name. To reference a specific option from the collection use either the index number or the id value of the option button control. Thus, the first option button for the `plan` field from the customer order form would have the reference:

```
document.forms.orderForm.elements.plan[0]
```

Figure 6-10 describes some of the properties associated with individual option buttons.

PROPERTY	DESCRIPTION
option.checked	Boolean value indicating whether the option button, *option*, is currently checked by the user
option.defaultChecked	Boolean value indicating whether *option* is checked by default
option.disabled	Boolean value indicating whether *option* is disabled or not
option.name	The field name associated with *option*
option.value	The field value association with *option*

Figure 6-10 Properties of option buttons

Locating the Checked Option

The option selected by the user will be indicated by the presence of the `checked` property. The following code uses a `for` loop to go through each option button associated with the `plan` field, storing the value of the selected option in the `pCost` variable, and breaking off the `for` loop and storing the option button value once the checked button has been found:

```
let orderForm = document.forms.orderForm;
let plan = orderForm.elements.plan;
for (let i = 0; i < plan.length; i++) {
   if (plan[i].checked) {
      planValue = plan[i].value;
      break;
   }
}
```

In place of a `for` loop, you can use the following CSS selector, which references the checked option button from the `plan` field:

```
input[name="plan"]:checked
```

By placing this selector within a `querySelector()` method, you can retrieve the value of the checked option without the need for a `for` loop as the following code demonstrates:

```
let planValue =
document.querySelector('input[name="plan"]:checked').value;
```

Note that these techniques will return a value only if an option has been selected from the option button collection. Use this method now to retrieve the cost of the selected protection plan.

To retrieve the cost of the selected plan:

1. Return to the **js06a.js** file in your code editor.
2. Within the `calcOrder()` function add the following code described in **Figure 6-11**.

```
// Retrieve the cost of the protection plan
let planValue =
document.querySelector('input[name="plan"]:checked').value;

// Charge the plan to each item ordered
let planCost = planValue * quantity;
orderForm.elements.planCost.value = planCost;
```

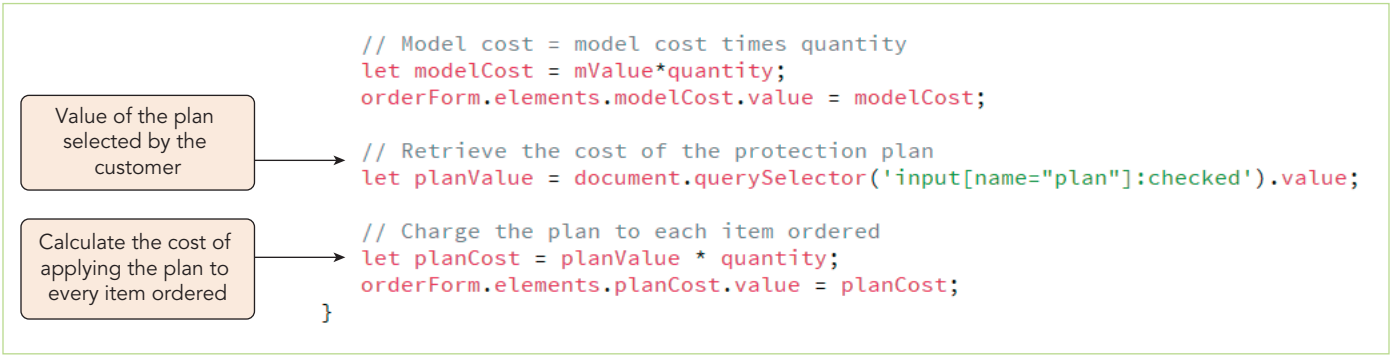

Figure 6-11 Calculating the cost of the protection plan

3. Save your changes to the file and then reload **js06a.html** in your browser. A 0 should now appear in the `planCost` field because no protection plan ($0.00) is selected by default.

Note

Checkbox controls work the same way as option buttons because the `checked` property indicates whether the box is checked and the field value associated with a checked box is stored in the `value` property of the checkbox object. However, this value is applied only when the checkbox is checked; otherwise there is no field value associated with the element.

To complete the order form calculations, add commands to the `calcOrder()` function to calculate the subtotal, taxes due, and the total cost of the order.

To complete the cost calculations:

1. Return to the **js06a.js** file in your code editor.
2. Add the following statements to the `calcOrder()` function to calculate and display the order subtotal:

```
// Calculate the order subtotal
let subtotal = modelCost + planCost;
orderForm.elements.subtotal.value = subtotal;
```

3. Add the following statements to `calcOrder()` to calculate the 5% sales tax:

```
// Calculate the 5% sales tax
let salesTax = subtotal * 0.05;
orderForm.elements.salesTax.value = salesTax;
```

4. Finally, add the following statements to calculate the total cost of the order by adding the subtotal and sales tax:

```
// Calculate the total cost of the order
let totalCost = subtotal + salesTax;
orderForm.elements.totalCost.value = totalCost;
```

5. **Figure 6-12** describes the newly added code in the function.

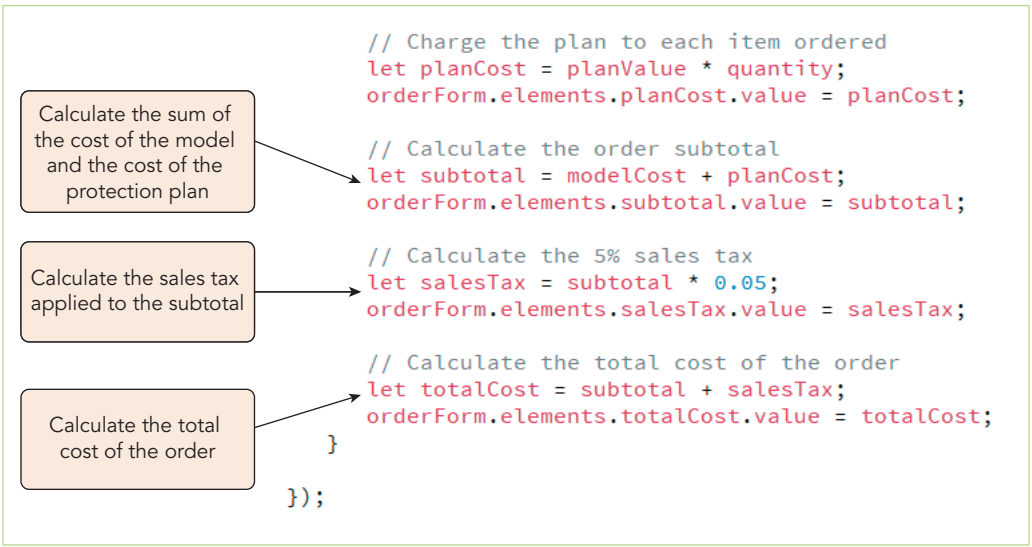

Figure 6-12 Calculating the subtotal, sales tax, and total order cost

6. Save your changes to the file and then reload **js06a.html** in your browser. **Figure 6-13** shows the calculated values from the initial state of the order form.

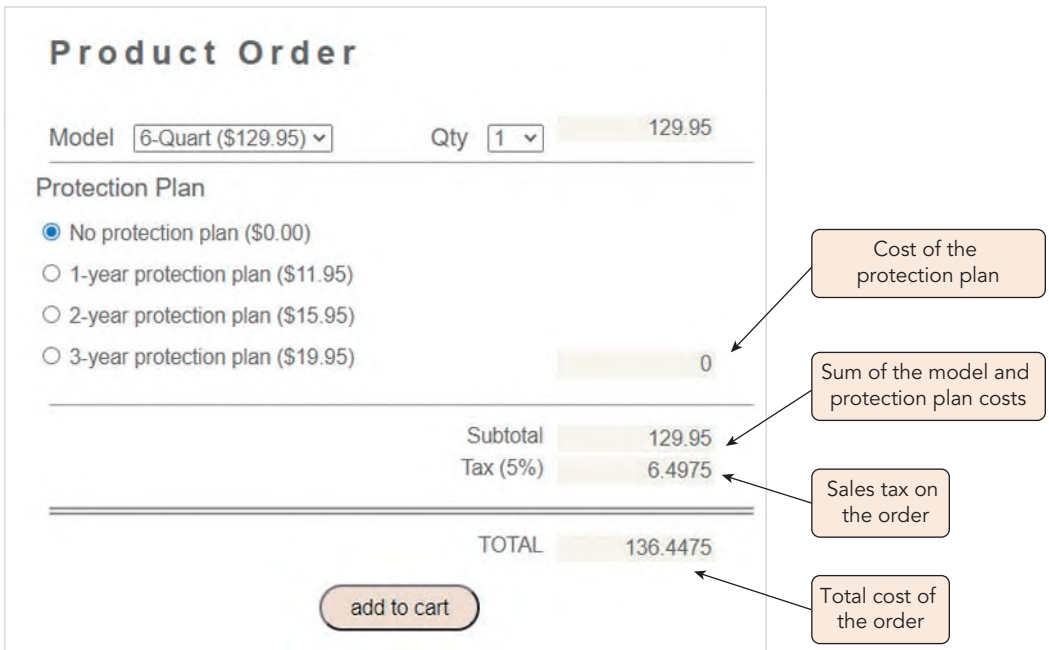

Figure 6-13 Initial order calculations

Accessing the Option Label

A challenge with option buttons is that the text associated with the button is not part of the `<input>` tag but instead is displayed alongside the button. HTML deals with this challenge with the `<label>` tag, which marks text associated with a specified form control. For example, the order form contains the following code for the No Protection Plan option button and its label:

```
<input type="radio" id="plan_0" name="plan" value="0" checked />
<label for="plan_0">No protection plan ($0.00)</label>
```

The value of the input control's `id` attribute (plan_0) is the same as the value of label's `for` attribute, associating the input control with the label. Because an input control can be associated with more than one label, JavaScript supports the following `labels` node list for any `input` element:

```
input.labels
```

where *input* is a reference to an `input` element. The text of the No Protection Plan option button's one (and only) label would be retrieved using the following commands:

```
let noProtection = document.getElementById("plan_0");
let planLabel = noProtection.labels[0].textContent;
```

You can also use the `querySelector()` method to retrieve the text of the label associated with the checked option button using the code:

```
let plan = document.querySelector('input[name="plan"]:checked');
let planLabel = plan.labels[0].textContent;
```

If the label contains HTML code in addition to plain text, it can be retrieved using the `innerHTML` property.

Formatting Data Values in a Form

The content of a form needs to be simple and clear. The numeric values shown in the order form, while calculated correctly, are difficult to read. Currency values should be displayed to two decimal places with commas used as thousands separators. In some cases, you may want to preface the value with a currency symbol, such as $ or €. You can use JavaScript's formatting methods to format calculated values as currency.

The `toFixed()` Method

JavaScript stores numeric values to 16 decimal places. This level of precision can result in calculated values displayed with a long string of digits. For example, a value such as 1/3 would be stored and displayed as 0.3333333333333333. It is rare that you would need more than a few decimal places in any calculation, so to make your forms more readable you will often want to reduce the number of digits.

To set the number of digits displayed by the browser, apply the following `toFixed()` method:

```
value.toFixed(n)
```

where *value* is the value to be displayed and *n* is the number of decimal places. The following examples demonstrate how the `toFixed()` method would display numeric values to different levels of precision:

```
let total = 2.835;
total.toFixed(0);   // returns "3"
total.toFixed(1);   // returns "2.8"
total.toFixed(2);   // returns "2.84"
```

Notice that the `toFixed()` method converts a numeric value to a text string and rounds the last digit in the expression rather than truncating it. Do not apply this method until your script has finished all calculations. Prior to that, you should keep the complete 16-digit level of accuracy in your calculations.

Formatting Values Using a Locale String

The `toFixed()` method is limited to setting the decimal place accuracy; it does not format numbers as currency or separate groups of thousand with the comma symbol. To do those tasks, apply the following `toLocaleString()` method:

```
value.toLocaleString(locale, {options})
```

where *locale* is a comma-separated list of location and language codes that indicate the locale for displaying numeric values, and *options* is a comma-separated list of formatting options for numeric values. With no arguments, the `toLocaleString()` method displays a numeric value using the computer's own local standards. The following code demonstrates the format applied to a sample number in which the user's computer employs standard English United States formatting:

```
let total = 14281.478;
total.toLocaleString();  // returns "14,281.478"
```

Different locales have different formatting standards. In France, the convention is to use spaces to separate groups of a thousand and commas to mark the decimal place. A French locale applied to the same test value would appear as:

```
let total = 14281.478;
total.toLocaleString("fr");  // returns "14 281,47"
```

To create your own standards, customize the appearance of the formatted text using the *options* argument of the `toLocaleString()` method. **Figure 6-14** describes the options that provide for complete control over number formatting.

OPTION	DESCRIPTION
`style: type`	Formatting style to use where *type* is "decimal", "currency", or "percent"
`currency: code`	Currency symbol to use for currency formatting where *code* designates the country or language
`currencyDisplay: type`	Currency text to display where *type* is "symbol" for a currency symbol, "code" for the ISO currency code, or "name" for the currency name
`useGroup: Boolean`	Indicates whether to use a thousands grouping symbol (`true`) or not (`false`)
`minimumIntegerDigits: num`	The minimum number of digits to display where *num* ranges from 1 (the default) to 21
`minimumFractionDigits: num`	The minimum number of fraction digits where *num* varies from 0 to 20; 2 digits are used for currency and 0 digits are used for plain number and percentages
`maximumFractionDigits: num`	The maximum number of fraction digits where *num* varies from 0 to 20; 2 digits are used for currency and 0 digits are used for plain number and percentages
`minimumSignficantDigits: num`	The minimum number of significant digits where *num* varies from 1 (the default) to 21
`maximumSignificantDigits: num`	The maximum number of significant digits where *num* varies from 1 (the default) to 21

Figure 6-14 Options of the `toLocaleString()` method

The following code demonstrates how to display a numeric value as currency using U.S. dollars by setting the locale to U.S. English, the number style to currency, and the currency symbol to "USD" (the $ symbol):

```
let total = 14281.5;
total.toLocaleString("en-US", {style: "currency", currency: "USD"})
// returns "$14,281.50"
```

To display the same currency amount in Euros under a French locale, apply the following commands:

```
let total = 14281.5;
total.toLocaleString("fr", {style: "currency", currency: "EUR"})
// returns "14 281,50€"
```

You can set the *locale* value to `undefined` to apply a format based on whatever is set on the user's computer.

Note

The `toLocaleString()` method can be used with date values as well. By applying the method to a date value, you can display a date or time in a wide variety of standard formats.

Use the `toLocaleString()` method to display every calculated value in the order form as U.S. currency prefaced with the $ symbol and with commas separating each group of one thousand.

To apply the `setLocaleString()` method:

1. Return to the **js06a.js** file in your code editor.
2. Go to the `calcOrder()` function and locate the statement that stores the value of the `modelCost` variable in the `modelCost` field of the order form. Append the `toLocaleString()` method to the statement changing it from `orderForm.elements.modelCost.value = modelCost` to:

```
orderForm.elements.modelCost.value = modelCost.toLocaleString("en-US", {style: "currency",
currency: "USD"})
```

3. Repeat Step 2 to the four statements displaying the values of the `planCost`, `subtotal`, `salesTax`, and `totalCost` fields. You can use the copy and paste feature of your code editor to duplicate the code for each field. **Figure 6-15** highlights the changed code in the file.

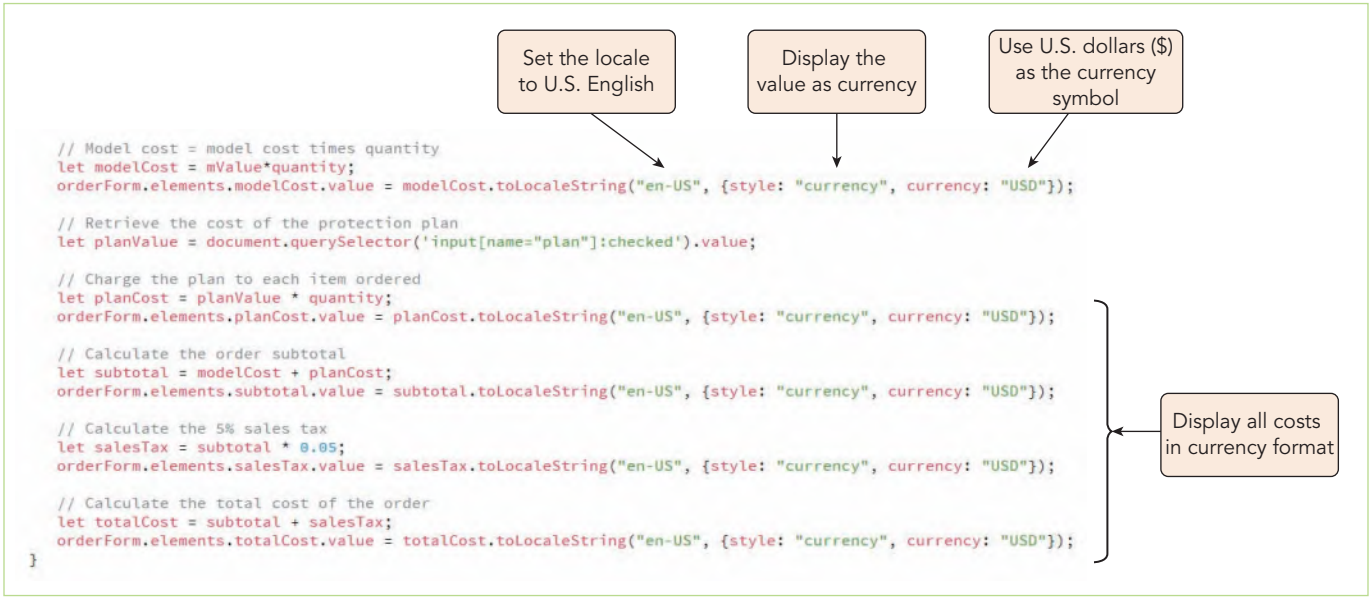

Figure 6-15 Applying the `toLocaleString()` method

4. Save your changes to the file and then reload **js06a.html** in your browser. **Figure 6-16** shows the values in the order form formatted as U.S. currency.

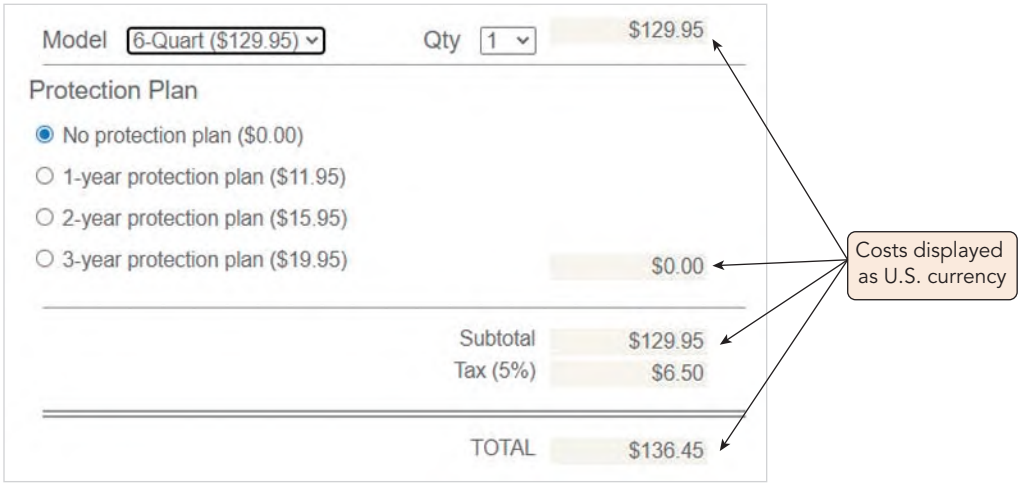

Figure 6-16 Displaying numeric values as currency

Next, add code to recalculate the costs whenever the customer changes a selection in the order form.

Skills at Work | Making a Website International Friendly

On the World Wide Web, your customers and associates can come from anywhere. You need to plan your website for international visitors as well as domestic clients. Consider the following tips as you plan to go "international":

- Support international conventions for dates, times, numbers, and currency using country and language codes in your HTML content and JavaScript programs.
- Avoid images that contain text strings. A picture is a worth a thousand words but not if that picture includes language foreign to your audience.
- Make your layout flexible. The translated version of your content might contain more words or fewer words than your web page. Ensure that the page layout can adapt to different text content. Remember that in some countries, text is read from right to left.
- Optimize your site for international searches, including using country-specific domain names and keywords tailored to international customers.
- Provide customers with a way to convert their payments into their own currency or provide information on exchange rates.
- Be aware of cultural differences: Color, working hours, holidays, and so forth have different meanings in different countries.

Once you have established an international website, monitor its usage with various analytical tools such as Google Webmaster and Analytics. A poor traffic report might indicate a problem with the international content of your website.

Responding to Form Events

Web forms need to be able to respond instantly to changes made to data values and form selections. JavaScript supports provides this interactivity using the event handlers described in **Figure 6-17**.

To run the `calcOrder()` function when the user selects a different model to order, apply the following `onchange` event handler to the `model` field:

```
orderForm.elements.model.onchange = calcOrder;
```

EVENT HANDLER	DESCRIPTION
element.onblur	The form *element* has lost the focus
element.onchange	The value of *element* has changed, and *element* has lost the focus
element.onfocus	The *element* has received the focus
element.oninput	The *element* has received user input
element.oninvalid	The *element* value is invalid
form.onreset	The *form* has been reset
element.onsearch	The user has entered something into a search field
element.onselect	Text has been selected within the *element*
form.onsubmit	The *form* has been submitted

Figure 6-17 Form and element event handlers

Edits made to an input field do not invoke a `change` event until field loses the focus, signifying that the changes to the field are completed. This is to avoid the event firing while the user is typing values into the input control.

In contrast, the `input` event is fired whenever the user changes a value within a control even if the control has not lost the focus. If a script needs to respond immediately to changes made while typing a field's value, use the `oninput` event handler or listen for the `input` event. Note that the `input` event does not apply to selection lists or option buttons because no content is changed.

Add an event listener for the `change` event to every element within the order form, running the `calcOrder()` function in response.

To add an event listener to every element in the order form:

1. Return to the **js06a.js** file in your code editor.
2. Directly below the statement that gives the focus to the model field, add the following `for` loop that adds event listeners to every item in the elements collection of the order form (see **Figure 6-18**):

```
// Add an event listener for every form element
for (let i = 0; i < orderForm.elements.length; i++) {
   orderForm.elements[i].addEventListener("change", calcOrder);
}
```

```
window.addEventListener("load", function() {
   let orderForm = document.forms.orderForm;
   let model = orderForm.elements.model;

   // Select Model selection list when form opens
   model.focus();

   // Add an event listener for every form element
   for (let i = 0; i < orderForm.elements.length; i++) {
      orderForm.elements[i].addEventListener("change", calcOrder);
   }
```

Run the `calcOrder()` function when any order form element changes its value

Figure 6-18 Adding event listeners to a form

3. Save your changes to the file and then reload **js06a.html** in your browser.
4. Test the event listeners by changing the model to **8-Quart ($159.95)**, the quantity to **8** and the protection plan to **3-year protection plan ($19.95)**. **Figure 6-19** shows the updated cost estimate for the order.

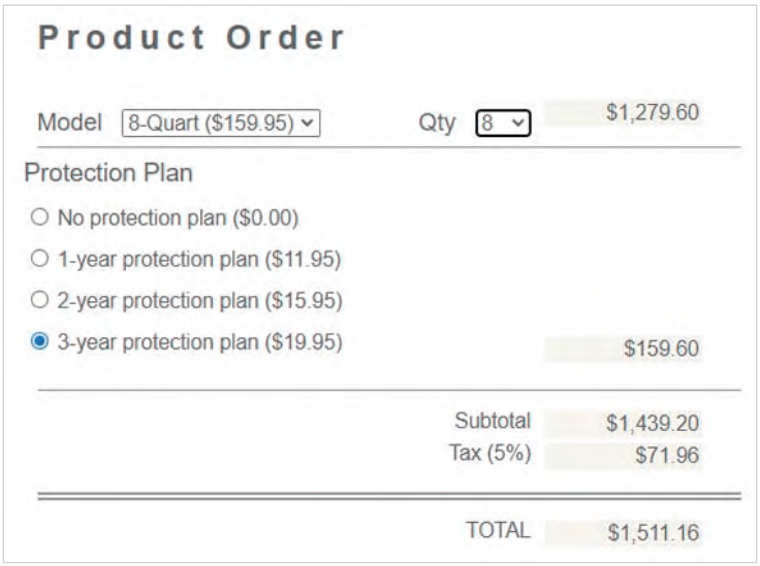

Figure 6-19 Order costs recalculated for different customer choices

Working with Hidden Fields

In many web forms, important data is stored within hidden fields making that data available to the server processing the form but hiding that data from the user. The product order page contains the following hidden fields to store the model and protection plan chosen by the consumer:

```
<input type="hidden" id="modelName" name="modelName" />
<input type="hidden" id="planName" name="planName" />
```

To store values in these two hidden fields extract the name of the model chosen in the Model selection list and the name of the plan chosen from the list of protection plan option.

The model name is contained in the text of the selected option and can be stored in the `modelName` field using the `text` attribute in the following command:

```
orderForm.elements.modelName.value =
orderForm.elements.model.options[mIndex].text;
```

where the `mIndex` variable provides the index of the option chosen from the model selection list.

The text of the selected plan must be retrieved from the text of the `label` element associated with that option button. Retrieve that text using the following statements:

```
let planOpt = document.querySelector('input[name="plan"]:checked');
orderForm.elements.planName.value =  planOpt.labels[0].textContent;
```

Add these two sets of commands to the `calcOrder()` function.

To store a value in a hidden field:

1. Return to the **js06a.js** file in your code editor.

2. At the bottom of the `calcOrder()` function add the following code as shown in **Figure 6-20**:

```
orderForm.elements.modelName.value = model.options[mIndex].text;
let selectedPlan =
document.querySelector('input[name="plan"]:checked');
orderForm.elements.planName.value =
selectedPlan.labels[0].textContent;
```

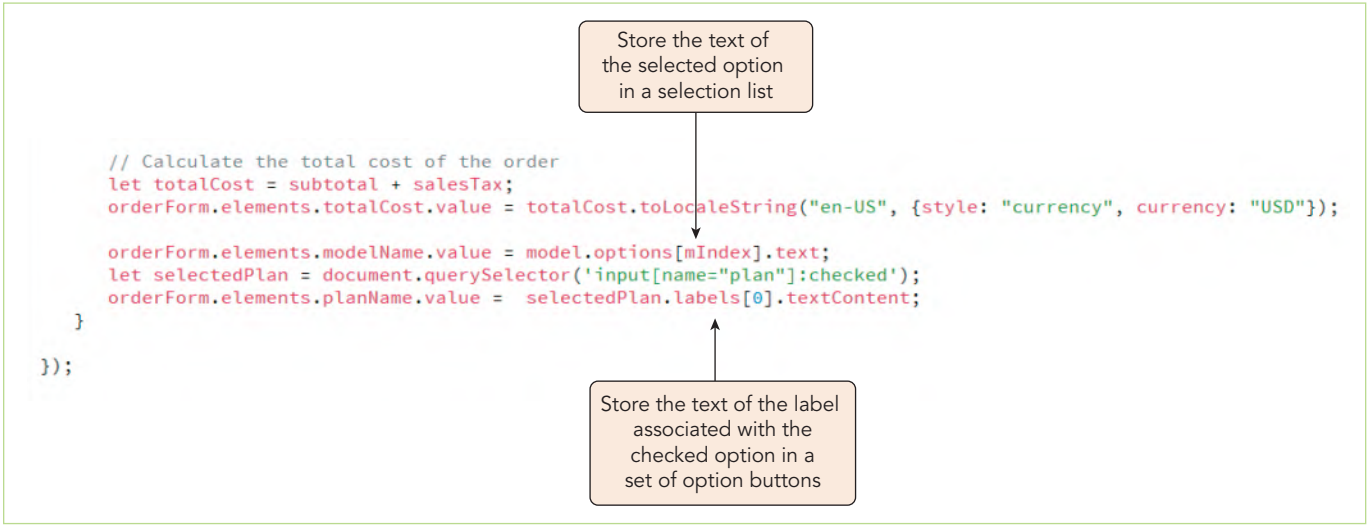

Figure 6-20 Setting the value of hidden fields

3. Save your changes to the file and then reload **js06a.html** in your browser.
4. Open the browser debugger and verify that no errors in the code are noted by the debugger.
5. Close the debugger window when you are satisfied that the code is working without error; otherwise return to your code editor to fix any reported mistakes.

You have completed the coding for the order form report. At this point a customer could click the Add to Cart button and submit the order to the web server for processing. However, because the focus of this book is client-side JavaScript, this form will submit the data to another HTML document named ordersubmit.html located in the same folder as the js06a.html file. The only purpose of the ordersubmit.html document is to display the form data that would be submitted to a server. However, you can view the submitted information to verify that the form is working correctly.

To submit the completed order:

1. Enter the following data in the order form as displayed in your browser: Model = **8-Quart ($159.95)**, Quantity = **6** and Protection Plan = **1-year protection plan ($11.95)**.
2. Click the **add to cart** button to submit the completed form. The browser opens the submitorder.html file with your choices and the calculated values already filled in. See **Figure 6-21**.

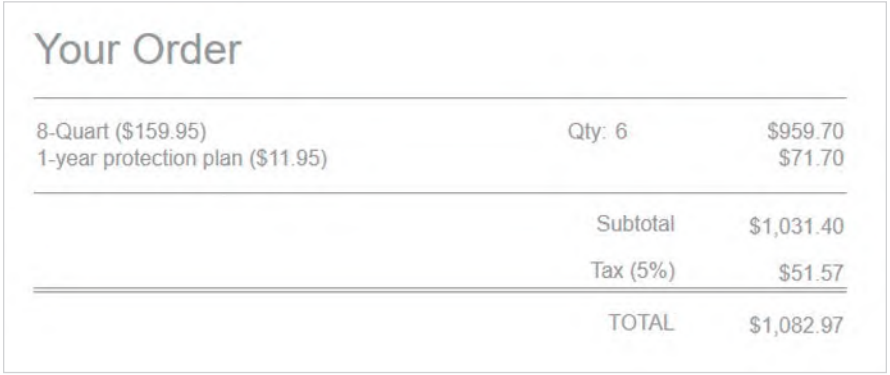

Figure 6-21 Contents of the shopping cart

Notice in the browser's address box that the address for the ordersubmit.html file is appended with a long text string called a query string containing field names and values from the product order form. The script embedded in the ordersubmit.html file extracts these field names and values and formats for them for the order summary shown in Figure 6-21. Techniques for extracting data from a query string are beyond the scope of this chapter.

Next you will learn how to use JavaScript and web forms to validate the payment information for this order.

Quick Check 1

1. Provide the object reference to the second element within the first web form on the web page.
2. Provide code to retrieve the value of the selected option in the selection list with the id "state", storing the value in the `stateName` variable.
3. Provide code to retrieve the value of the checked option in the option group for the `shipping` field.
4. Provide code to display the value of the `payment` variable as United States currency.
5. Provide code to run the `calcShipping()` function when the value of the `state` field in the `shoppingCart` form is changed.

Exploring Form Submission

When the user has completed a web form, the data can be submitted for processing as you did with the product order form. Forms are submitted when the user clicks (or otherwise interacts) with a submit button. Submit buttons are marked using `<input>` or `<button>` tags with the attribute `type="submit"`. When a submit button is activated, the following actions occur within the browser:

1. The field values are checked for invalid data.
2. If no invalid data is found, a `submit` event is fired indicating that the form is being submitted.
3. If no errors occur in the form submission, a request is sent to the server or other resource handling the form data.

Once the request has been sent to the server or other resource, the action of the form is completed until another submit button is activated.

Using the `submit` Event

The submission of a form by clicking a submit button creates a `submit` event after a successful validation. A form can also be submitted via JavaScript using the following `submit()` method:

```
form.submit()
```

where *form* is a reference to the web form. Note that submitting a form in this fashion bypasses a validation of the form's contents and does not fire the `submit` event of the `form` object.

Resetting a Form

Forms are reset whenever the user clicks a reset button, marked within an `<input>` or `<button>` tag containing the `type="reset"` attribute. When a form is reset, all fields are set back to their default values. Clicking a reset button fires the `reset` event which also be initiated using the `reset()` method:

```
form.reset()
```

In general, avoid resetting a form because restoring all fields to their default values can be confusing to the user.

Validating Form Data with JavaScript

A big part of any form submission is checking the form for invalid data in a process known as validation. When this validation is done using the user's own computer, it is known as client-side validation as opposed to server-side validation, which is handled by the web server.

The product order form you worked on earlier used selection lists and option buttons to limit data to a predetermined list of options. In so doing, the form reduced the possibility of data entry error, making it more likely that any information sent to the server was complete and accurate.

Not every form can be so constructed. Many fields cannot be so easily confined to a list of options, such as fields that request a customer name, password, credit card number, or address. In those situations, client-side validation should be applied as much as possible to catch errors and notify the user of the mistake. While some validation still must be done on the web server, it is good practice to do as much validation as possible on the user's own computer to reduce server load.

HTML provides attributes to restrict what data the user can and cannot enter within a form, and CSS provides style rules that highlight data entry errors. You will want to take advantage of these browser-based validation or native validation tools whenever possible but in other situations you may need to augment these features with a validation script of your own.

A payment form for the customer's order has been created for you. Your task is to write a script providing validation checks and feedback not covered by the built-in browser validation tools. Open the web page form and the script file now.

To open the files for the payment form:

1. Go to the js06 ▶ chapter folder of your data files.
2. Use your code editor to open the **js06b_txt.html** and **js06b_txt.js** files. Enter your name and the date in the comment section of each file and then save them as **js06b.html** and **js06b.js**, respectively.
3. Return to the **js06b.html** file in your code editor. Within the head section, add a `script` element to run the js06b.js script file, deferring the loading the script file until after the entire page has loaded.
4. Take some time to scroll through the contents of the HTML file, studying the elements used for the input controls on the payment form and the field names associated with each control.
5. Open the **js06b.html** file in your web browser. **Figure 6-22** describes the fields contained within the payment form.

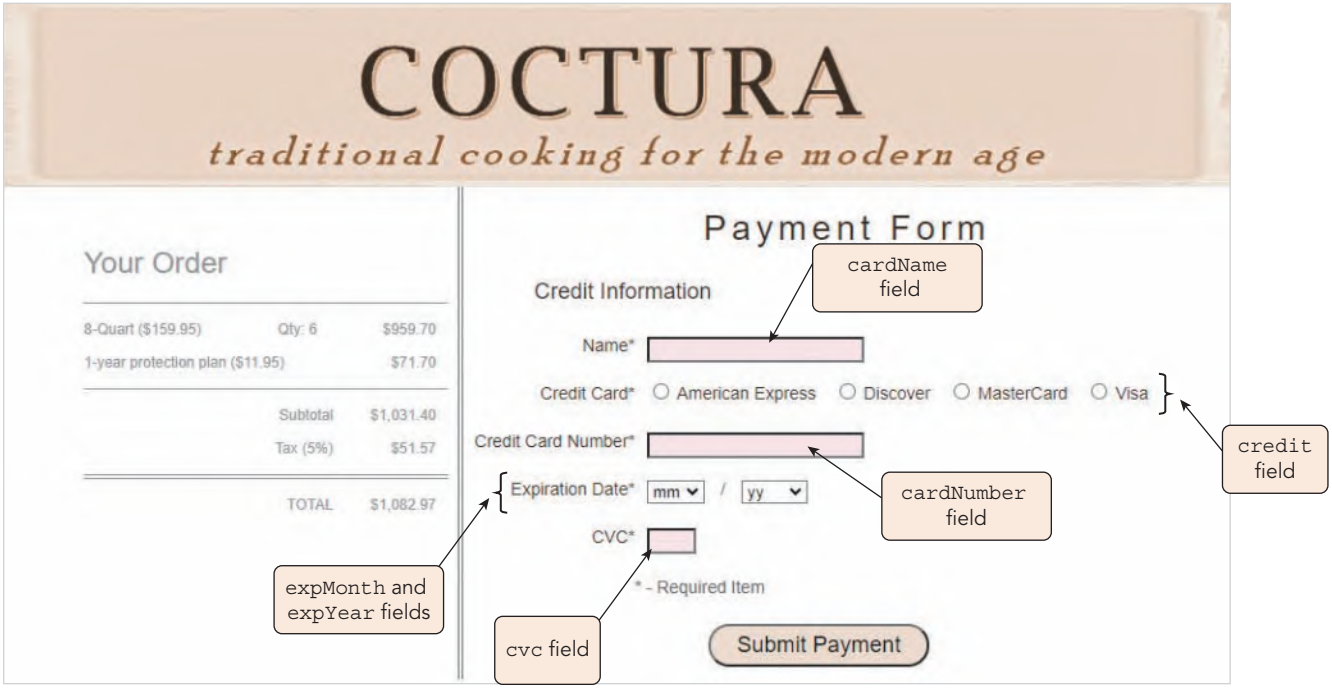

Figure 6-22 Payment form

Before creating your own validation tests, explore the validation features built into the HTML and CSS code.

To explore a validation check:

1. With no data in the payment form, click the **Submit Payment** button on the payment form.
2. The browser returns an error bubble requesting that the Name field be filled out. See **Figure 6-23**.
3. Reload the page and the web form.

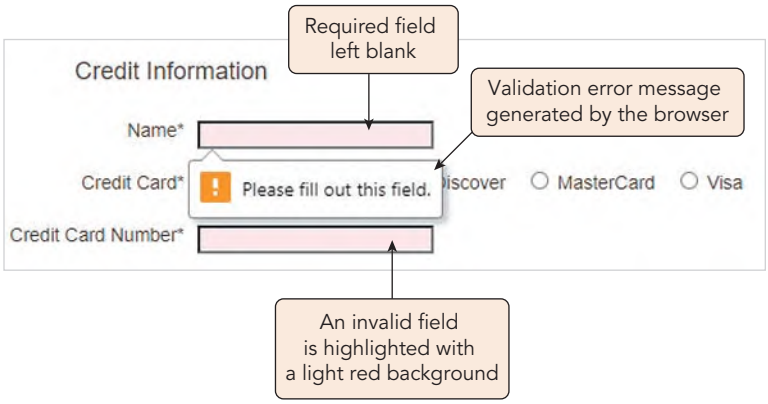

Figure 6-23 Browser validation message and highlighting

The error bubble displayed by the browser occurs because `cardName` is a required field as indicated by the following `<input>` tag in the js06b.html file:

```
<input name="cardName" id="cardName" required type="text" />
```

The other data fields in the form also have the `required` attribute so that a payment cannot be submitted unless data is entered in each field. The appearance and content of the error bubble is determined by the browser.

CSS styles highlight those form fields that are invalid for one reason or another. In this example, input boxes containing invalid data are displayed with a semi-transparent red background based on the following CSS style rule:

```
input:invalid {
   background-color: rgba(221,147,148,0.2);
 }
```

where the `invalid` pseudo-class selects those input elements containing invalid data. **Figure 6-24** describes some of the other HTML attributes that can be added to `input` elements to help mark invalid data.

For example, to ensure that a data for an age field must fall between 18 and 35, include the following `min` and `max` attributes with the `<input>` tag:

```
<input name="age" min="18" max="35" type="number" />
```

As much as possible, use these HTML attributes as the first line of defense against invalid data. However, browser-generated validation checks and CSS styles have some important limitations in protecting against invalid data:

- The validation error message is generic and might not contain specific information about the source of the error.
- The validation tests are based on a single field value and do not allow for tests involving multiple fields.
- The validation tests are limited to what was entered (or not entered) into the data field and, thus, cannot be generalized to work with calculated items or functions.

ATTRIBUTE	DESCRIPTION
maxlength="*value*"	Sets the maximum number of characters allowed in an input field
min="*value*"	Sets the minimum allowed value in an input field; can be used with numbers, ranges, dates, and times
max="*value*"	Sets the maximum allowed value in an input field; can be used with numbers, ranges, dates, and times
pattern="*regex*"	Specifies a regular expression pattern that text within an input field must satisfy to be valid
required	Makes it a requirement that data be entered into an input field for the field to be valid
step="*value*"	Sets the step interval between values entered into a numeric field
type="*date*"	The input field must contain a date
type="email"	The input field must contain an email address
type="month"	The input field must contain a month and year
type="number"	The input field must contain a numeric value
type="tel"	The input field must contain a phone number
type="time"	The input field must contain a time value
type="url"	The input field must contain a URL
type="week"	The input field must contain a week and year

Figure 6-24 Attributes of the `input` element

To supplement the native browser validation tools, use the form validation properties and methods built into JavaScript, which are known collectively as the Constraint Validation API. For this payment form, you will want your script to do the following:

- Provide customized error messages to explain the source of the error.
- Verify that the customer has entered a name for the credit card owner.
- Verify that one of four credit card brands has been selected.
- Verify that a valid credit card number for the user's credit card has been entered.
- Verify that the card's expiration date has been selected from the drop-down lists.
- Verify that a valid CVC number for the user's credit card has been entered.

You will start by exploring how to work with JavaScript's validation properties and methods.

Working with the Constraint Validation API

The Constraint Validation API includes the properties and methods listed in **Figure 6-25**.

For example, the following expression returns the Boolean value `true` if the `cardName` field from the payment form contains valid data:

```
document.forms.payment.elements.cardName.valid
```

You can also test for valid data using the following `checkValidity()` method:

```
document.forms.payment.elements.cardName.checkValidity()
```

The `checkValidity()` method returns a value of `true` for valid data; but, if the field is invalid, the method returns a value of `false` while firing an invalid event. An invalid event is an event that occurs whenever the browser encounters a field whose value does not match the rules specified for its content.

Note

The `checkValidity()` method can be applied to a `form` object as well as individual form elements. When applied to a `form` object, if at least one element within that form fails validation, the form fails.

PROPERTY OR METHOD	DESCRIPTION
form.noValidate	Set to true to prevent the native browser tools from validating the web form *form*
form.reportValidity()	Reports on the validation status of *form* using the native browser validation tools
element.willValidate	Returns true if web form element *element* is capable of being validated by the browser (regardless of where the data itself is actually valid)
element.valid	Sets or return the validity of the element where true indicates the element contain valid data and false indicates an invalidate field value
element.validationMessage	Sets or returns the text of the validation message returned by the browser when *element* fails validation
element.validity	Returns a ValidityState object containing specific information about the validation of *element*
element.setCustomValidity(*msg*)	Sets the validity message displayed by the browser where *msg* is the text displayed when *element* fails validation (set *msg* to an empty text string to indicate that the element does not have a validation error)
element.checkValidity()	Returns true if *element* is valid and false if it is not invalid; a false value also fires the invalid event

Figure 6-25 Constraint Validation API properties and methods

Exploring the ValidityState Object

There are several reasons why data might be flagged as invalid. Information about the cause of invalid data is stored in the ValidityState object referenced with the expression:

```
element.validity
```

where *element* is a field from a web form. To determine the current validation state of an element, apply the expression

```
element.validity.ValidityState
```

where *ValidityState* is one the validation states described in **Figure 6-26**.

VALIDITY STATE	DESCRIPTION
element.validity.badInput	The field element, *element*, contains data that the browser is unable to convert, such as when an e-mail address lacks the @ character
element.validity.customError	A custom validation message has been set to a non-empty text string using the setCustomValidity() method
element.validity.patternMismatch	The *element* value does not match the character pattern specified in the pattern attribute
element.validity.rangeOverflow	The *element* value is greater than the max attribute
element.validity.rangeUnderflow	The *element* value is less than the min attribute
element.validity.stepMismatch	The *element* value does not match the step attribute
element.validity.tooLong	The *element* character length exceeds the value of the maxLength attribute
element.validity.tooShort	The *element* character length is less than the minLength attribute
element.validity.typeMismatch	The *element* value does not match the data type specified by the type attribute
element.validity.valid	The *element* contains valid data, satisfying all constraints
element.validity.valueMissing	The *element* specified in the pattern does not contain data though it is marked with the required attribute

Figure 6-26 Properties of a ValidityState object

In summary, the checkValidity() method will tell you whether a field is invalid; the validity property will tell you why.

In the following code the validity state of the `userMail` field is evaluated to determine whether it contains the correct data type. If the field's data type was set to "mail" and the field did not contain an email address, the `notValid` variable would return a value of `true`, indicating that the `userMail` field does not conform to its specified data type.

```
let email = document.getElementById("userMail");
let notValid = email.validity.typeMismatch;
```

Note that the `typeMismatch` property can only test whether a field's value matches the data type, it does not test whether that field's value is legitimate. Testing an email address only confirms that the text string "looks" like an email address it does not test whether that email address exists.

Creating a Custom Validation Message

You have already seen that when a data field fails validation, the browser displays its own native error bubble notifying the user of the problem. Different browsers may display different messages. For example, Google Chrome displays the message "Please fill out this field" for missing data while Microsoft Edge displays the message "This is a required field." To display the same error message across all browsers, apply the following `setCustomValidity()` method to the element:

```
element.setCustomValidity(msg)
```

where *msg* is the custom message displayed by all browsers, overriding the native browser error message. If an element is valid, store an empty text string for the *msg* parameter, which also marks the element as being valid.

Note

You can access the text of the validation error message associated with an invalid data field by using the *element*`.validationMessage` property where *element* is the form element containing the error.

Use the `setCustomValidty()` method to display the message "Enter your name as it appears on the card" if the user leaves the required field `cardName` blank. Otherwise, store an empty string using the `setCustomValidity()` method so that the `cardName` field is marked as valid.

To create custom validation message:

1. Return to the **js06b.js** file in your code editor.
2. Directly below the initial comment section, add the following `validateName()` function:

```
// Check if the owner's name is entered on the cardfunction
validateName() {
   let cardName = document.getElementById("cardName");
   if (cardName.validity.valueMissing) {
      cardName.setCustomValidity("Enter your name as it appears on the card");
   } else {
      cardName.setCustomValidity("");
   }
}
```

Figure 6-27 describes the code in the function.

By creating your own customized error message, you have provided more specific information to the customer about the nature of the error.

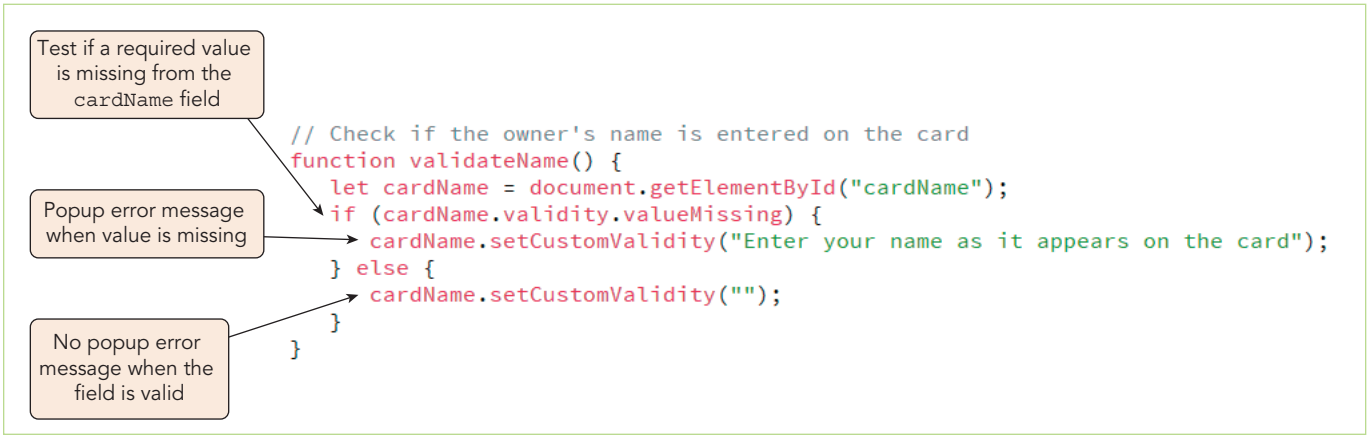

Figure 6-27 Creating the `validateName()` function

Responding to Invalid Data

Form data can be tested after the user enters the data into an input control and when the submit button is clicked to initiate form submission. To check the validity of form data after it is entered, use an event handler or event listener for the `change` event. To catch invalid data before the form is submitted, add an event handler or event listener for the `click` event of the form's submit button.

There are reasons for both approaches. If you want users to be notified immediately of invalid data so that mistakes can be corrected before continuing with the form, validate the data after it is entered. If such constant prompting would annoy and frustrate the user, save all the validation checks until the entire form has been completed and submitted. For the payment form you save all validation checks until the submit button is clicked.

Add an event listener now to run the `validateName()` function when the submit button is clicked.

To add the event listener for the `click` event:

1. Directly above the `validateName()` function add the following statement as described in **Figure 6-28**:

```
let subButton = document.getElementById("subButton");

// Validate the payment when the submit button is clicked
subButton.addEventListener("click", validateName);
```

Figure 6-28 Calling the `validateName()` function

2. Save your changes to the file and then reload **js06b.html** in your web browser.
3. Without entering any data, click the **Submit Payment** button. Verify that the revised error message appears next to the empty `cardName` field. See **Figure 6-29**.
4. Enter a sample name into the `cardName` field and click the **Submit Payment** button again. Verify that no error bubble appears next to the `cardName` field.

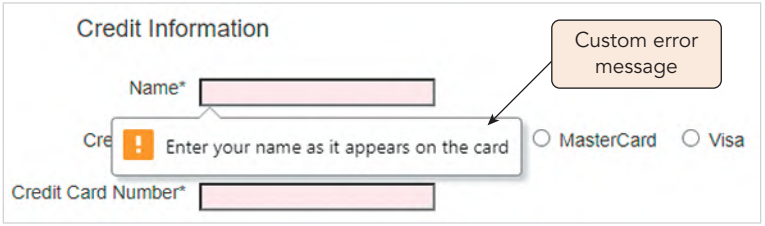

Figure 6-29 Customized popup error message

The next field on the payment form is the `credit` field, which is laid out as a set of option buttons. This is also a required field; however, with option buttons clicking one option button from the list automatically sets the values of the remaining option buttons; therefore, the `required` attribute is needed only with the first option button and validation tests need to be performed only on that first option button.

Create the `validateCredit()` function to test whether the user has selected an option button from the group and if no option button is selected, display the custom validation message "Select your credit card".

To create the `validateCredit()` function:

1. Return to the **js06b.js** file in your code editor.
2. Directly below the event listener for the `click` event, add the following event listener to add another function to the `click` event:

```
subButton.addEventListener("click", validateCard);
```

3. Directly below the `validateName()` function add the following function as described in **Figure 6-30**:

```
// Check if a credit card has been selected
function validateCard() {
   let card = document.forms.payment.elements.credit[0];
   if (card.validity.valueMissing) {
      card.setCustomValidity("Select your credit card");
   } else {
      card.setCustomValidity("");
   }
}
```

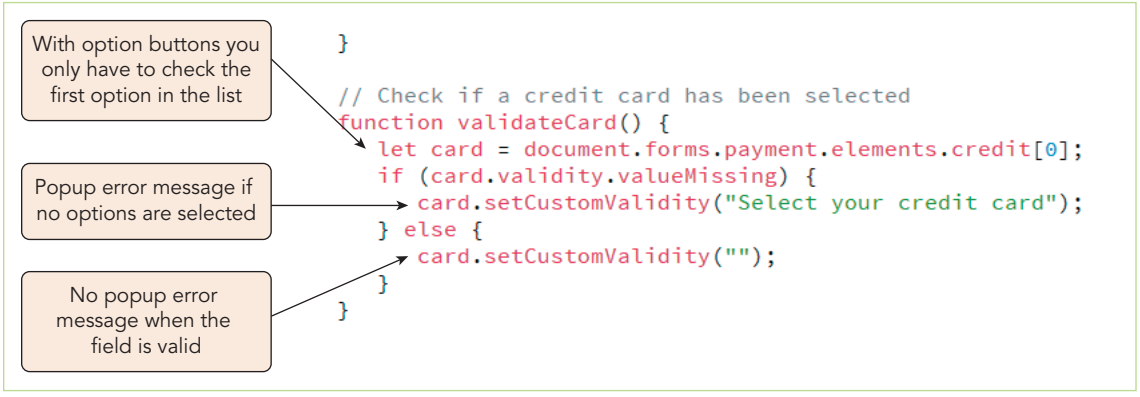

Figure 6-30 Creating the `validateCard()` function

4. Save your changes to the file and then reload **js06b.html** in your web browser.
5. Enter a sample name in the Name box, but do not select a credit card from the list of options.

6. Click the **Submit Payment** button. Verify that the error message "Select your credit card" appears next to the list of unselected credit card options.
7. Click one of the credit card option buttons and click the **Submit Payment** button again. Verify that the next error bubble appears alongside the empty Credit Card Number input box.

The next field in the payment form is the `cardNumber` field in which not only must a value be entered but the value must also be a string of numbers fitting a recognized credit card number pattern.

Validating Data with Pattern Matching

The content of a text string can be validated against a regular expression, which is concise code describing the general pattern and content of the characters within a text string. For example, the regular expression:

```
^\d{5}5$
```

matches any text string containing exactly 5 digits, such as "12345", "80517", or "00314", but not "abcde" or "123456".

It is beyond the scope of this chapter to discuss the syntax of regular expressions, but they are an important tool in the validation of any form, especially forms used in e-commerce. Regular expressions can be long and complicated to accommodate a wide range of possible character patterns, such as would be used to validate a credit card number. The following rather imposing regular expression pattern has already been entered in the `<input>` tag for the `cardNumber` field:

```
pattern = "^(?:4[0-9]{12}(?:[0-9]{3})?|5[1-5][0-9]{14}|6(?:011|5[0-9][0-9])[0-9]{12}|3[47]
[0-9]{13}|3(?:0[0-5]|[68][0-9])[0-9]{11}|(?:2131|1800|35\d{3})\d{11})$"
```

This long and complicated expression matches the valid credit card number patterns associated with the four types of credit cards listed in the payment form. Though you don't have to duplicate this expression in your script file, you want to display a customized error message if the number entered by the customer does not match this regular expression pattern or if no credit card number is entered at all.

Create a function named `validateNumber()` that displays the error message "Enter your card number" if the customer leaves the `cardNumber` field blank or the error message "Enter a valid card number" if the customer enters a number that does not match an approved credit card number pattern. You will use the `valueMissing` property to test for a missing value and the `patternMismatch` property to test for a card number that does not follow the prescribed character pattern.

To create the `validateNumber()` function:

1. Return to the **js06b.js** file in your code editor.
2. Directly below the event listener for the `click` event, add the following event listener to run the `validateNumber()` function in response to the `click` event:

```
subButton.addEventListener("click", validateNumber);
```

3. Directly below the `validateCard()` function add the following function as described in **Figure 6-31**:

```
// Check if the card number is valid
function validateNumber() {
   let cNum = document.getElementById("cardNumber");
   if (cNum.validity.valueMissing) {
      cNum.setCustomValidity("Enter your card number");
   } else if (cNum.validity.patternMismatch) {
      cNum.setCustomValidity("Enter a valid card number");
   } else {
      cNum.setCustomValidity("");
   }
}
```

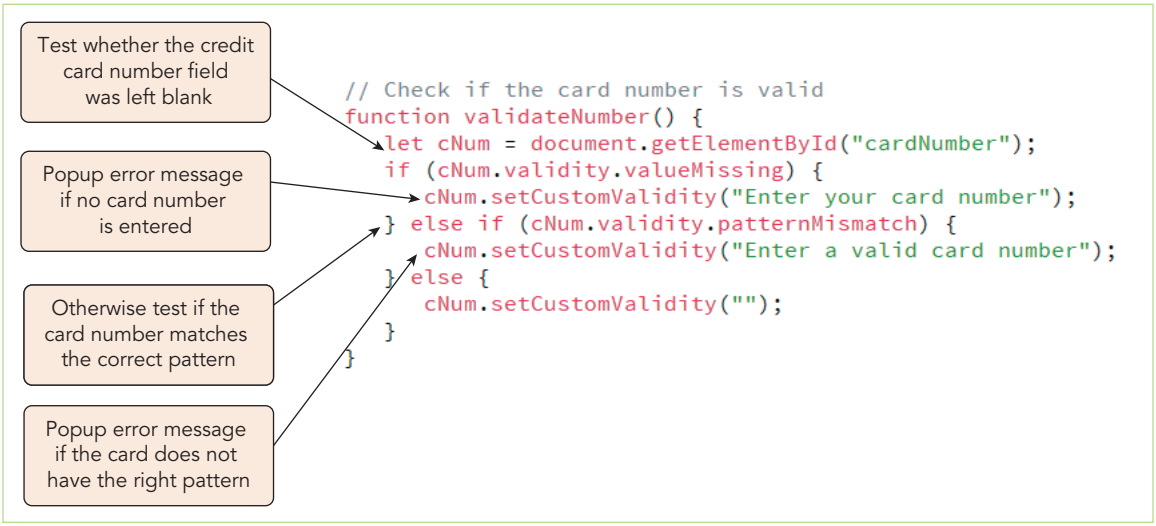

Figure 6-31 Creating the `validateNumber()` function

4. Save your changes to the file and then reload **js06b.html** in your web browser.
5. Enter a sample name in the Name box and select the **MasterCard** credit card.
6. Click the **Submit Payment** button. Verify that the error message "Enter your card number" appears next to the Credit Card Number box.
7. Enter the invalid credit card number **1234567890** into the `cardNumber` field and click the **Submit Payment** button. Verify that the browser displays the message "Enter a valid card number", as shown in **Figure 6-32**.

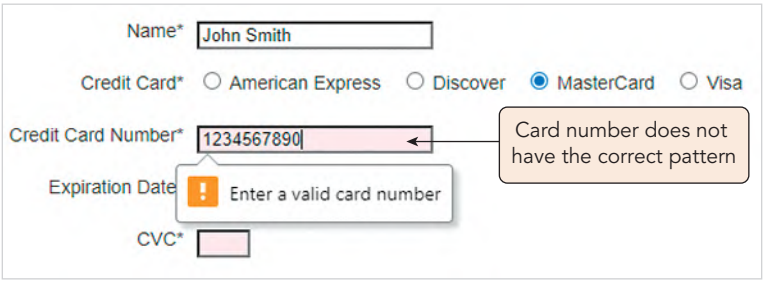

Figure 6-32 Popup error message for an invalid card number

8. Enter the valid credit card number **6011485077126974** into the `cardNumber` field and click the **Submit Payment** button. Verify that the browser accepts this number and does not display an error bubble.

The next part of the payment form contains two drop-down list boxes for the month and year of the credit card expiration date.

Validating a Selection List

The payment form has placed the possible expiration date values of the credit card in two selection lists named `expMonth` and `expYear`. The first entry in each of the two selection lists is "mm" and "yy", respectively. You must validate these two fields so that if either "mm" or "yy" is left selected, their respective fields will be flagged as invalid. Use the `selectedIndex` property to determine if the selected index is 0 (the first entry). If the index is 0, then the browser will declare the field value as invalid, otherwise it will accept the selected month or year as valid.

To validate the expiration date:

1. Return to the **js06b.js** file in your code editor.
2. Directly below the event listeners for the `click` events, add the following event listeners:

```
subButton.addEventListener("click", validateMonth);
subButton.addEventListener("click", validateYear);
```

3. Directly below the `validateNumber()` function add the following function to validate the expiration month:

```
// Check that a month is selected for the expiration date
function validateMonth() {
   let month = document.getElementById("expMonth");
   if (month.selectedIndex === 0) {
      month.setCustomValidity("Select the expiration month");
   } else {
      month.setCustomValidity("");
   }
}
```

4. Add the following `validateYear()` function to validate the expiration year:

```
// Check that a year is selected for the expiration date
function validateYear() {
   let year = document.getElementById("expYear");
   if (year.selectedIndex === 0) {
      year.setCustomValidity("Select the expiration year");
   } else {
      year.setCustomValidity("");
   }
}
```

Figure 6-33 describes the newly added code.

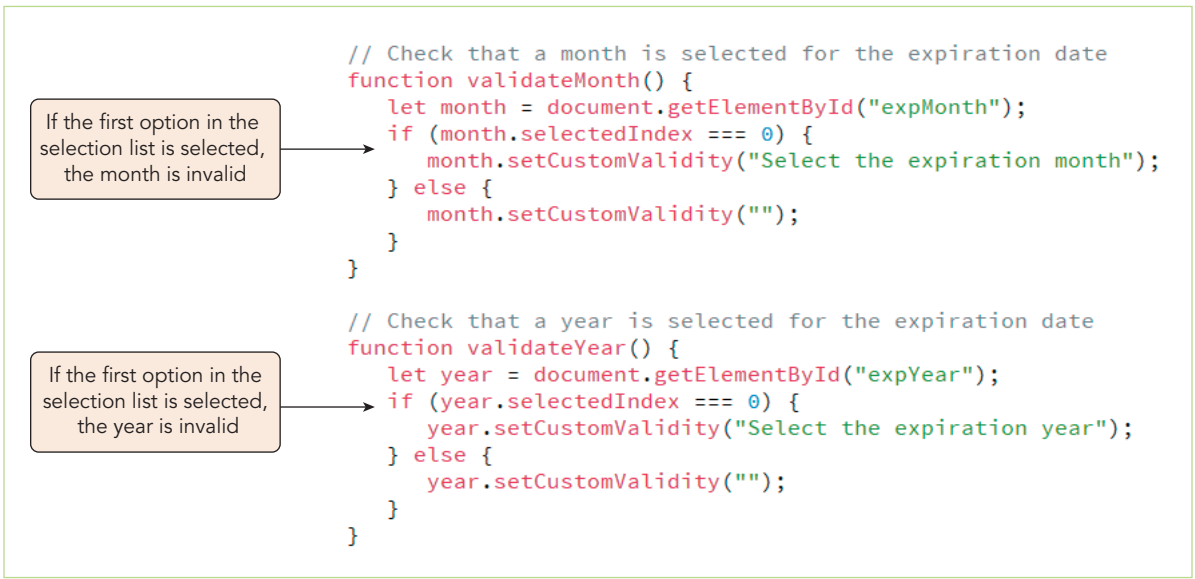

Figure 6-33 Creating the `validateMonth()` and `validateYear()` functions

5. Save your changes to the file and then reload **js06b.html** in your web browser.
6. Verify that unless you select a month and a year from the selection lists, validation error messages appear when you submit the payment form.

The last field remaining in the payment form is the CVC field, which is the card verification code printed on credit cards to provide additional security in financial transactions.

Testing a Form Field Against a Regular Expression

Credit card CVC numbers are either 3- or 4-digit numbers depending on the card being used. American Express cards use 4-digit CVC numbers while Discover, MasterCard, and Visa use 3-digit numbers. The regular expression that matches the 4-digit CVC numbers used by American Express is:

```
/^\d{4}$/
```

while for the other cards, the regular expression is

```
/^\d{3}$/
```

You can determine whether a text string conforms to a regular expression pattern using the following `test()` method:

```
regExp.test(text)
```

where *regExp* is the regular expression pattern and *text* is the text string containing the characters to be tested. If the text matches the regular expression pattern, the `test()` method returns the value `true`, otherwise it returns `false`. For example, the following code returns a Boolean value indicating whether the value in the `cardCVC` field matches the 4-digit pattern:

```
let cvc = document.getElementById("cvc");
let isValid = /^\d{4}$/.test(cvc.value)
```

To test whether the CVC is valid for American Express cards, use the following expression:

```
if ((card === "amex") && !(/^\d{4}$/.test(cvc.value)))
```

which returns `false` if the card is American Express and not a 4-digit number. For cards that are not American Express and require a 3-digit CVC code, use the `if` condition:

```
if ((card !== "amex") && !(/^\d{3}$/.test(cvc.value)))
```

which returns `false` if the card is not American Express and not a 3-digit number. You will use both of these `if` conditions in the `validateCVC()` function testing whether the customer has entered a valid CVC based on their selected credit card.

To create the `validateCVC()` function:

1. Return to the **js06b.js** file in your code editor.
2. Directly below the event listener statements, add the following statement to run the `validateCVC()` function when the submit button is clicked.

```
subButton.addEventListener("click", validateCVC);
```

3. Directly below the `validateYear()` function add the following code the for `validateCVC()` function:

```
function validateCVC() {
   // Determine which card was selected
   let card =
document.querySelector('input[name="credit"]:checked').value;
```

```
    let cvc = document.getElementById("cvc");

    // Validate the CVC value
    if (cvc.validity.valueMissing) {
       cvc.setCustomValidity("Enter your CVC number");
    } else if ((card === "amex") && !(/^\d{4}$/.test(cvc.value))) {
       cvc.setCustomValidity("Enter a 4-digit number");
    } else if ((card !== "amex") && !(/^\d{3}$/.test(cvc.value))) {
       cvc.setCustomValidity("Enter a 3-digit number");
    } else {
       cvc.setCustomValidity("");
    }
}
```

See **Figure 6-34**.

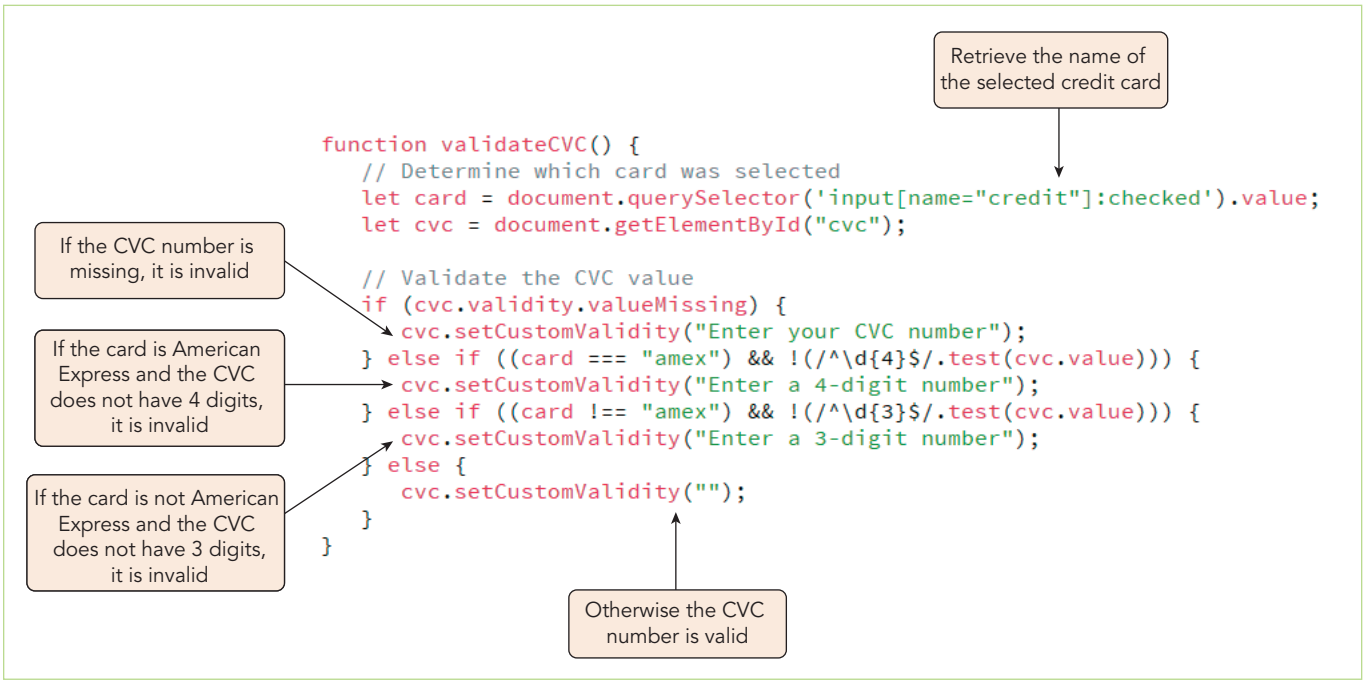

Figure 6-34 Creating the `validateCVC()` function

4. Save your changes to the file and then reload **js06b.html** in your web browser.
5. Complete the payment form by entering a sample name in the Name box, click the **American Express** option button, select **04/2026** as the expiration date, and enter **6011485077126974** as the credit card number.
6. Enter **123** as the CVC number for the card and click the **Submit Payment** button. Verify that the form rejects the CVC number as shown in **Figure 6-35**.
7. Change the CVC number to **1234** and click the **Submit Payment** button again. Verify that the form is successfully submitted and a web page confirming this fact is displayed.

If your form does not work, use the debugger tools on your browser to examine the code and compare your code to that shown in Figure 6-35. The statements in this function are very complicated, and it is easy to misplace a parenthesis or quotation mark.

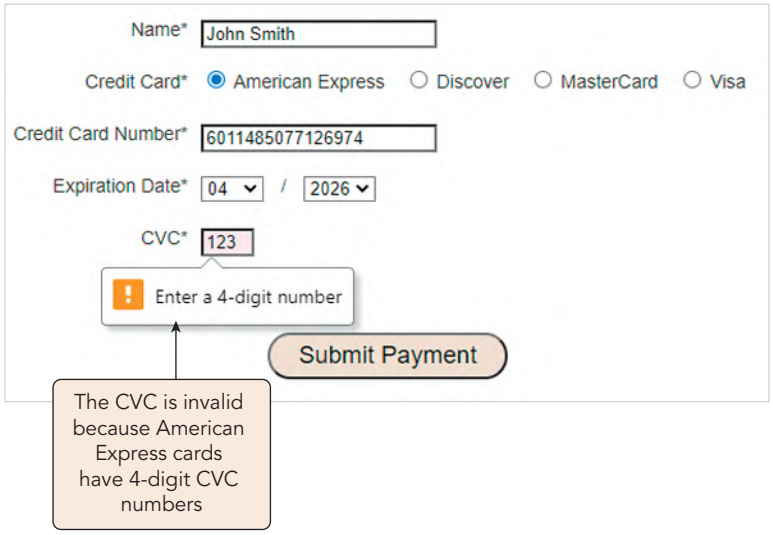

Figure 6-35 Validating the CVC number

Common Mistakes

Submitting a Form

You can submit a form using the method *form*.`submit()` where *form* is a reference to the web form. A common mistake is to assume that this method is equivalent to clicking a submit button. It is not. The key differences are as follows:

- The `submit` event is not triggered, so any event listeners or event handlers associated with the `submit` event will not be accessed.
- The native browser validation tools that are part of the Constraint Validation API will be bypassed.

If your script needs to use the native validation tools supplied by your browser, provide the user with a submit button and run your code in response to its use.

Creating a Custom Validity Check

A credit card number might fit the numeric pattern indicated by a regular expression but still be invalid. In addition to a specified pattern of characters, numerical ids like credit card numbers employ a checksum algorithm in which the sum of the digits must satisfy specific mathematical conditions. Most credit card numbers use the checksum algorithm known as the Luhn algorithm or the mod 10 algorithm, which calculates the sum of the digits in the credit card number after doubling every other digit going backwards from the end of the number. If the sum of the digits is a multiple of 10, the credit card number is legitimate, otherwise it is not.

A function named `luhn()` has been created for you and saved in the js06b.js file. The function performs the necessary calculations on the digits in a numerical id to determine if those digits satisfy the conditions of the Luhn algorithm. If they do, the function returns a value of `true`, otherwise it returns `false`. You will use this function as one last test of the customer's credit card number, verifying that not only is the pattern of the digits in the credit card number correct but also that the digits satisfy the Luhn algorithm.

To validate the expiration date:

1. Return to the **js06b.js** file in your code editor and go to the `validateNumber()` function.
2. Add the following condition directly before the final `else` condition in the function:

```
} else if (luhn(cNum.value) === false) {
   cNum.setCustomValidity("Enter a legitimate card number");
```

Figure 6-36 shows the complete code of the `validateNumber()` function.

If the card number fails the Luhn algorithm, display a validation error

```
// Check if the card number is valid
function validateNumber() {
   let cNum = document.getElementById("cardNumber");
   if (cNum.validity.valueMissing) {
      cNum.setCustomValidity("Enter your card number");
   } else if (cNum.validity.patternMismatch) {
      cNum.setCustomValidity("Enter a valid card number");
   } else if (luhn(cNum.value) === false) {
      cNum.setCustomValidity("Enter a legitimate card number");
   } else {
      cNum.setCustomValidity("");
   }
}
```

Figure 6-36 Validating the credit card number

3. Save your changes to the file and then reload **js06b.html** in your browser.
4. Enter a sample name in the Name box, click the **Discover** option button, select **04/2026** as the expiration date, enter **6011280768434850** as the credit card number and enter **123** as the CVC number.
5. Click the **Submit Payment** button.

   As shown in **Figure 6-37**, the credit card number is rejected because it fails the Luhn algorithm even though it has the correct general pattern.

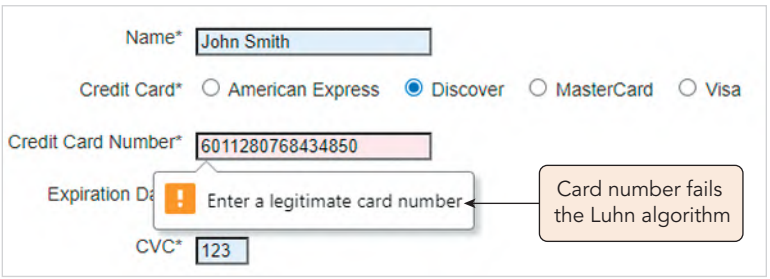

Figure 6-37 Reporting an illegitimate credit card number

6. Edit the credit card number by changing the last digit from 0 to 6 so that it reads **6011280768434856** and click the **Submit Payment** button. Verify that the form now passes validation as the credit card number fits the correct pattern and satisfies the Luhn algorithm.

You have completed your work on the validation of the payment form. At this point further validation would be done by the web server to verify that the credit card information matches a real account; but in doing some of the validation on the customer's own computer, you would have weeded out faulty data and reduced the workload on the server.

Managing Form Validation

In completing the payment form, you took advantage of the native browser tools for managing invalid data. With some applications, you might want to disable the native browser validation tools altogether and supply your own validation framework. Such a situation might occur if you need to support older browsers that do not supply native validation. It might also be the case that you do not want to use error bubbles to notify users of errors but would prefer to highlight validation errors in a different way such as with side notes or overlays.

To disable the built-in validation tools supplied by your browser, apply the following statement:

```
form.noValidate = true;
```

where *form* is the reference to the web form. You can achieve the same effect by adding the `novalidate` attribute to the `<form>` tag in the HTML file or by adding the attribute `formnovalidate` to the tag for the form's submit button.

Another way to control the native browser validation is by preventing the default browser actions associated with an `invalid` event (such as displaying an error bubble), which are fired whenever the browser notes an invalid data value. The following code uses the `addEventListener()` method to listen for an occurrence of an invalid event within a form element, running an anonymous function in response:

```
element.addEventListener("invalid", function(evt) {
   evt.preventDefault();
   commands;
});
```

The *evt* parameter in this code is an example of the event object, which is the object associated with an event captured by the script. Every event creates an event object. The anonymous function in this example applies the `preventDefault()` method to this event object to prevent the browser's default action of reporting the error. Having prevented the default actions associated with the event, the script is free to run a set of custom commands to respond to the invalid event.

Note

You can determine which form element reported the invalid data by using the *evt*`.target` property where *evt* is the name of the event object variable.

You can also turn off the built-in validation tools and write your own set of validation procedures and run them in response to the `submit` event occurring within the web form. The general code using an event handler is as follows:

```
form.onsubmit = myValidation;
function myValidation(e) {
   e.preventDefault();
   commands to determine if form passes validation
   if (form is valid) {
      commands run when form passes validation
      return true;
   } else {
      commands run when form doesn't pass validation
      return false;
}
```

The `myValidation()` function runs when the form is submitted, prevents the default actions associated with the submit event and then runs a different set of commands whether the form is valid or not. Notice that the function returns a value of `true` for valid forms and `false` for invalid forms, thus, indicating whether the submission was successful or not.

Using any of these approaches, you can create your own framework of validation tools customized to meet the specific needs of your application and your customers.

Best Practices | Designing an E-Commerce Website

When customers shop online, they are looking for what every customer looks for: good products at a fair price in a shopping experience that is pleasant and easy. While all these things are important, you cannot forget that your competition is only a click away and, if your customers don't have confidence in your website design, they might also not have confidence in the products you sell.

Here are some tips to keep in mind as you build your e-commerce website:

- Do not burden your customers with a long and complicated registration process. Provide guest users with easy access to your catalog because they will be more likely to register after viewing all you have to offer.
- Provide robust search tools. Make it easy to match your customers with the products they are most likely to purchase.
- Make it easy to navigate the purchasing process. Customers should be able to easily move forward and backward in the purchase process so that mistakes can be easily fixed. Provide information to the customer at each step in the process about what is being purchased and how much it will cost. Do not hide fees or taxes until later in the shopping process or you run the risk of irritating your customer.
- Put discount options and membership deals up front so that your customers can take advantage of deals that make a final purchase more likely.
- Use validation tests and security measures to reassure your customers that their credit information is safe and secure.
- Incorporate social media in your e-commerce website, providing your customers the opportunity to discuss with you and other customers your products and services.

Technology and customer tastes are constantly in flux and websites need to respond to a quickly changing market. Evaluate and revaluate your e-commerce design to ensure that it meets the needs of your customers today while you prepare for your customers of tomorrow.

Quick Check 2

1. Provide code to turn off the native browser validation for the web form with the name reviewForm.
2. Provide code to indicate whether there is a type mismatch for data entered in the input box with the id "reviewDate".
3. Provide code to indicate whether the field with the id "reviewRating" has a value greater than allowed by the `max` attribute.
4. Provide code to change the validation message for the reviewRating input box to "Value larger than allowed".
5. Provide code to test whether the value entered in the customerID input box matches the regular expression `/^[A-Z]{3}-\d{2}$/`

Summary

- A web form is a hierarchical structure consisting of `form` object containing form elements.
- Each attribute of a form control is matched by a JavaScript property for an element object.