HTML5, CSS3, and JavaScript

6th Edition

Tutorial 13

Programming for Web Forms

Carey

Objectives

- Reference forms and form fields
- Create calculated fields
- Retrieve field values from selection lists and radio buttons
- Format numeric and currency data
- Append field names and data to URLs

Objectives (continued)

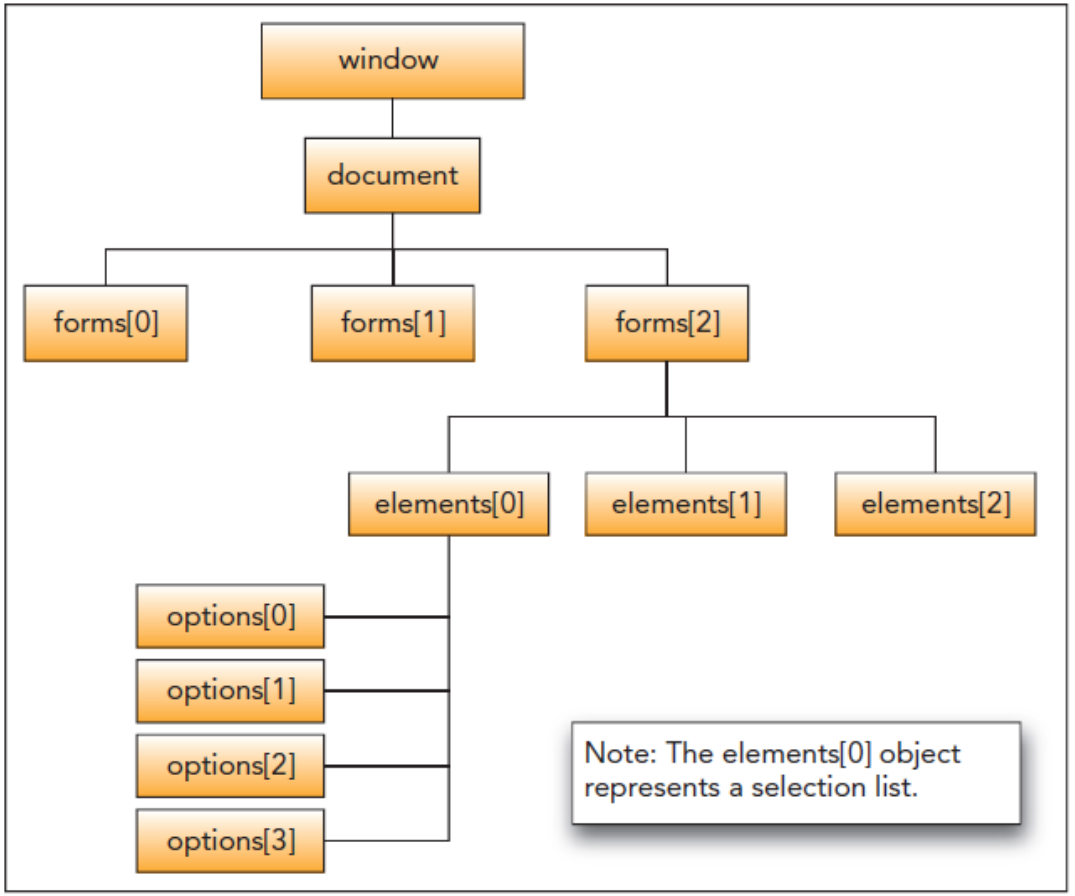
- Explore the syntax of regular expressions
- Use a regular expression to extract data from the URL query string
- Work with the validityState object
- Create custom validation messages
- Validate credit card numbers

Exploring the Forms Object

- The knowledge of properties and methods of the `form` object and the elements it contains is essential to program the behavior and contents of a web form
- Web forms and their elements are part of the hierarchy of objects within the web document

Exploring the Forms Object (continued 1)

Figure 13-1 Web form hierarchy



Exploring the Forms Object (continued 2)

- JavaScript organizes the forms into the following object collection:

  `document.forms[`*`idref`*`]`

  where *`idref`* is the index number or ID of the form

- The form can be referenced using either the `id` or `name` attribute as follows:

  `document.forms.`*`fname`*

  where *`fname`* is either the form's ID or name

Exploring the Forms Object (continued 3)

Figure 13-2 Form properties and methods

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.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

Working with Form Elements

- A form's input controls, selection lists, text area boxes, and field sets are organized into the following `elements` collection:

  form`.elements[`*idref*`]`

  where

  - *form* is the reference to the web form
  - *idref* is the index number or ID of the element

Working with Form Elements (continued 1)

- Element can also be referenced using its name or id attribute using the following expression:

  form.elements.*ename*;

  where *ename* is either the element's ID or name

- Example: Refer the orderDate field from the orderForm using the following object reference:

  document.orderForm.elements.orderDate

Working with Form Elements (continued 2)

- Reference a field using the value of the field name attribute using the following expression:

  form.field

  where *field* is the value of the field name

Setting the Field Value

- To set the value of an input control, use the following expression:

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

  where

  - *element* is a reference to a form element
  - *value* is the value to be stored in the element

- Example: Store the text string “2018-03-10” in the orderDate field as follows:

  ```
  document.orderForm.orderData.value =
  “2018-03-01”;
  ```

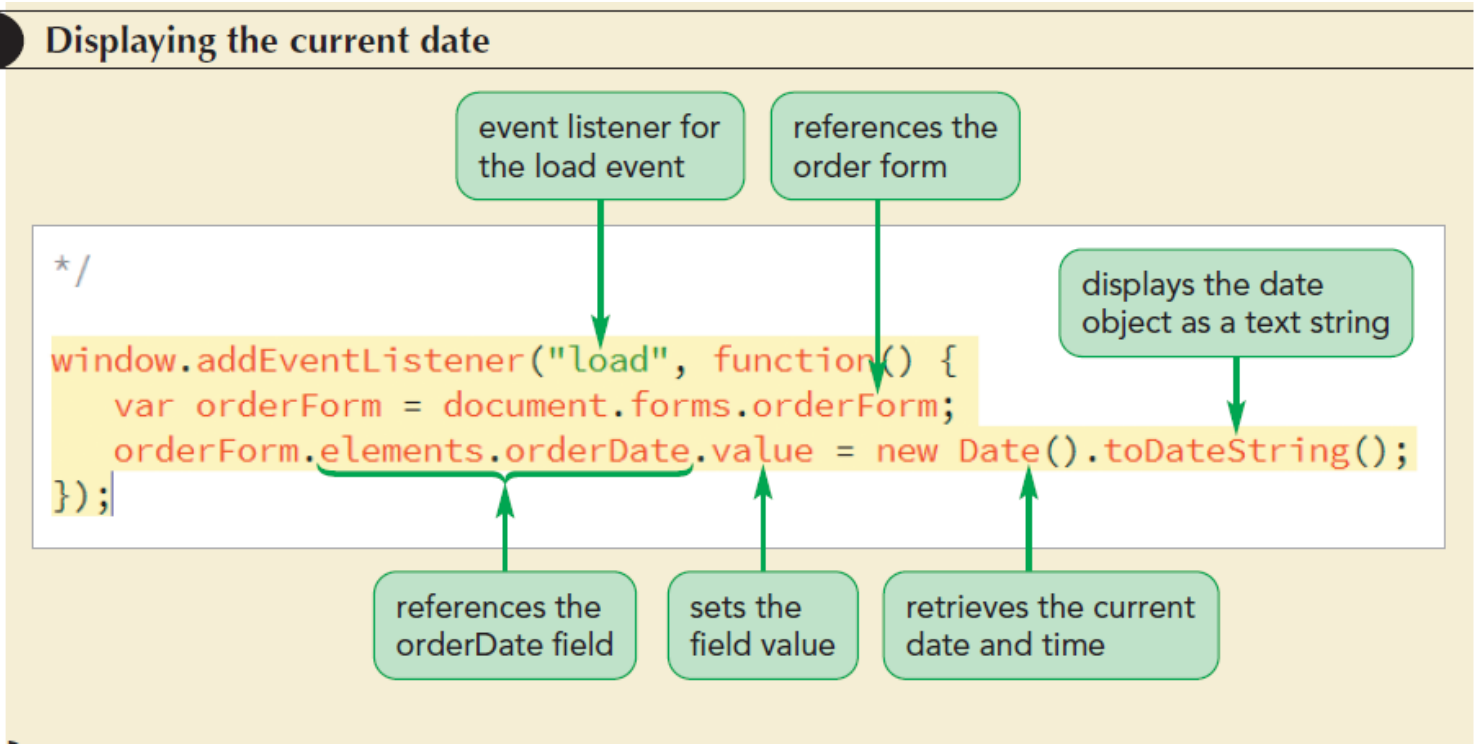
Setting the Field Value (continued 1)

Figure 13-4 Properties and methods of input boxes

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

Setting the Field Value (continued 2)

Figure 13-5 Displaying the current date



Navigating between Fields

- A field can be selected automatically when the form opens so that it is ready for data entry
- When a form field is selected either by clicking it or by moving it using keyboard buttons or arrows, it receives the focus of the browser
- To program this action, you use the following focus() method:

```
element.focus();
```

Navigating between Fields (continued 1)

Figure 13-7 Setting the form focus

```
window.addEventListener("load", function() {
   var orderForm = document.forms.orderForm;
   orderForm.elements.orderDate.value = new Date().toDateString();
   orderForm.elements.model.focus();
});
```

applies the focus to the model field

Working with Selection Lists

- Various properties and methods are associated with the selection list object

Figure 13-8 Selection list properties and methods

Property	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

Working with Selection Lists (continued 1)

- The value of a selection list is determined by the option that is selected by the user
- The selection list options are organized into the following options object collection:

  select.options[*idref*]

  where *select* is the reference to the selection list object and *idref* is the index number or ID of the option

Working with Selection Lists (continued 2)

Figure 13-9 Properties of selection list options

Property	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*

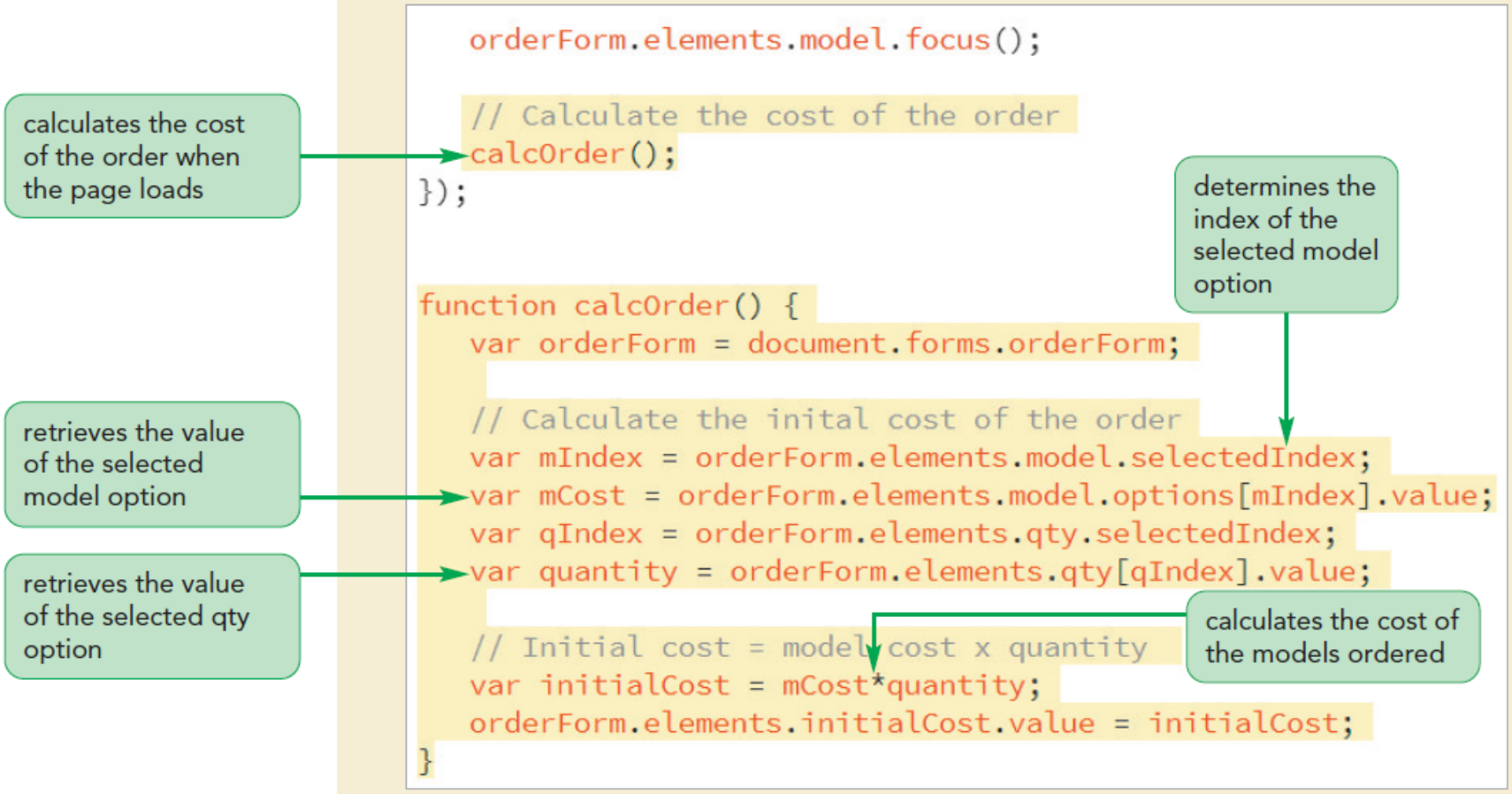
Working with Selection Lists (continued 3)

- `selectedIndex` property returns the value of the selected option from a selection list
- Example:

```
var orderForm =
document.forms.orderForm;
var model =
orderForm.elements.model;
var modelIndex =
model.selectedIndex;
var modelValue =
model.options[modelIndex].value;
```

Working with Selection Lists (continued 4)

Figure 13-10 Calculating the cost of models ordered



Working with Options Buttons and Check Boxes

- Option or radio buttons are grouped by a common field name placed within the following array:

  options[*idref*]

  where

  - *options* is the common field name used by all the option buttons
  - *idref* is either the index number or ID of the option button

Working with Options Buttons and Check Boxes (continued 1)

Figure 13-12 Properties of option or radio 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*

Working with Options Buttons and Check Boxes (continued 2)

- To retrieve the value of the checked option button without using a `for` loop, use the following CSS selector:

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

- Example: Determine the option button that has been checked by the user as follows:

```
var orderForm =
document.forms.orderForm;
var protection =
orderForm.elements.protection;
```

Working with Options Buttons and Check Boxes (continued 3)

```
for (var i = 0; i <
protection.length; i++) {
if (protection[i].checked === true){
var pCost = protection[i].value;
break;}}
```

Figure 13-13 Retrieving the cost of the selected protection plan

```
    // Initial cost = model cost x quantity
    var initialCost = mCost*quantity;
    orderForm.elements.initialCost.value = initialCost;

    // Retrieve the cost of the user's protection plan
    var pCost = document.querySelector('input[name="protection"]:checked').value;
    orderForm.elements.protectionCost.value = pCost;
}
```

shows the protection plan cost in the order form

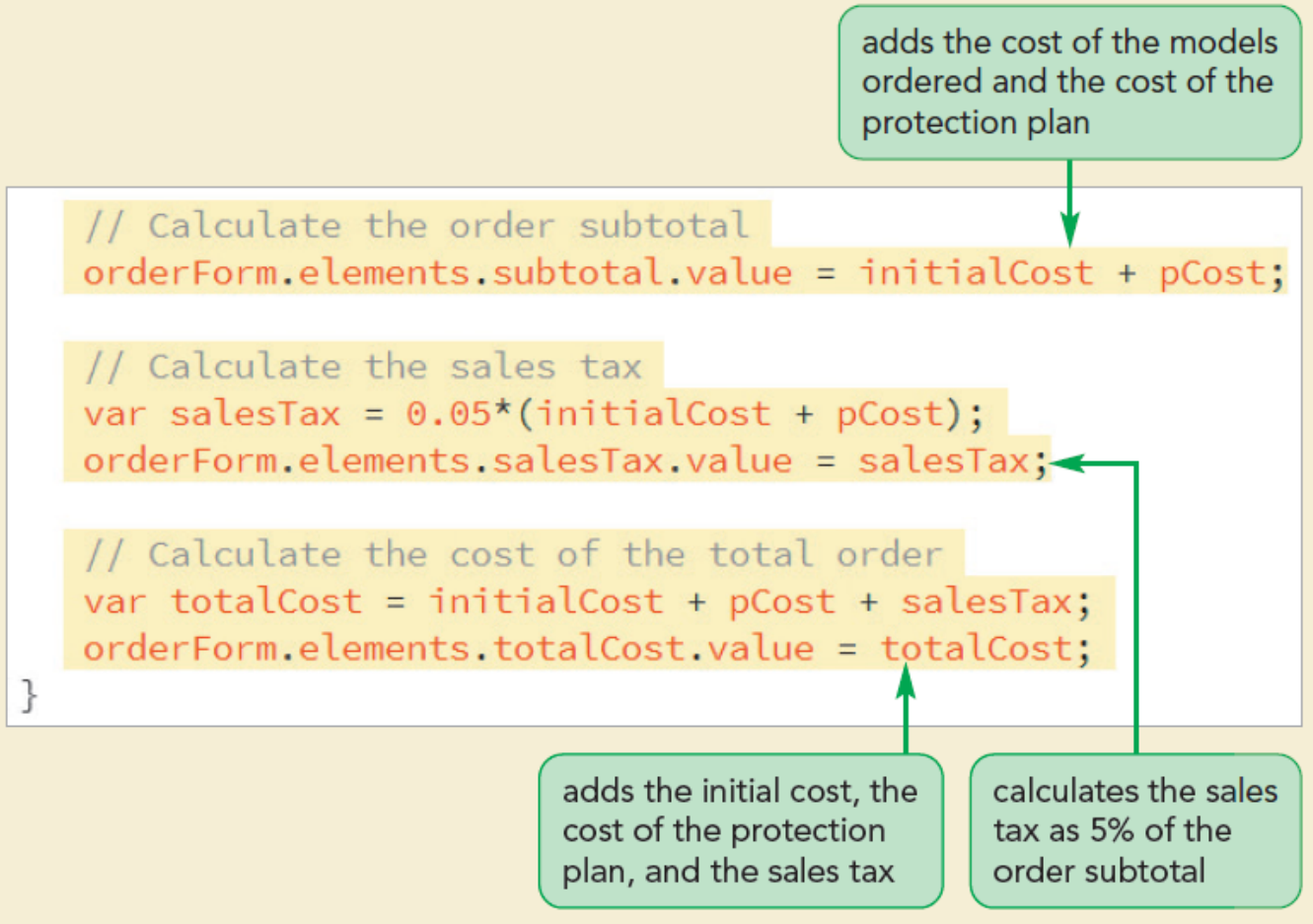
query to select the option button that is checked for the protection field

Working with Check Boxes

- Check box controls work the same way as option buttons in which the `checked` property indicates whether the box is checked
- The value associated with a checked box is stored in the value property of the check box object

Working with Check Boxes (continued 1)

Figure 13-14 Completing the calculations in the order form



Formatting Numeric Values

- Numeric values are displayed by JavaScript to 16 decimal place accuracy
- To control the number of digits displayed by the browser, use `toFixed()` method
- The `toFixed()` method is limited to only defining the decimal place accuracy and it does not format numbers as currency or separate thousands with the comma symbol

Formatting Numeric Values (continued 1)

- To have more control over the numeric format, use the following method:

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

  where *locale* is a comma-separated list of location and language codes and *options* is a comma-separated list of formatting options for numeric values

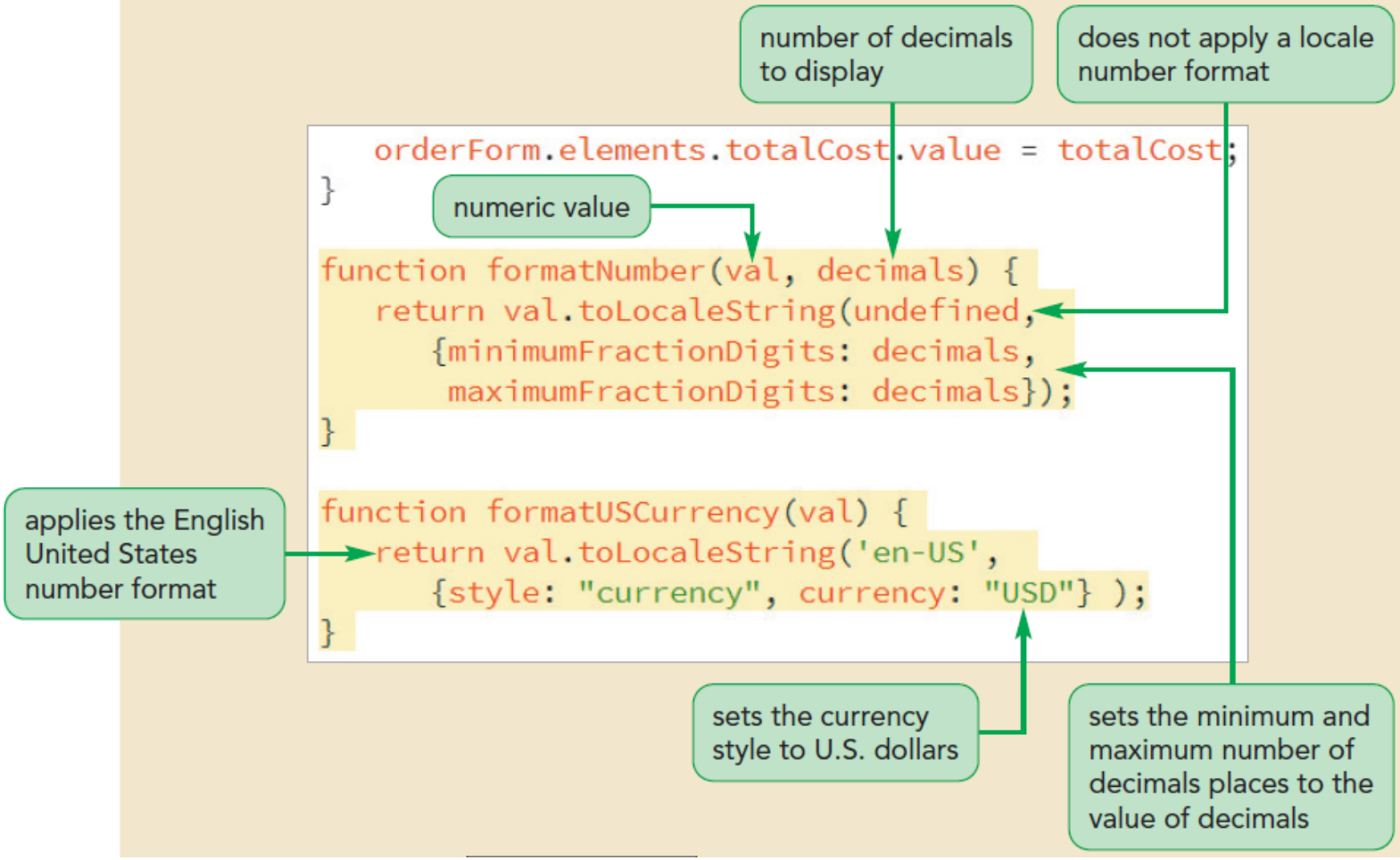
Formatting Numeric Values (continued 2)

Figure 13-16 Options from the toLocaleString() method

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

Formatting Numeric Values (continued 3)

Figure 13-17 Defining functions to format numeric values



Applying Form Events

- JavaScript supports event handlers to respond to user actions within a form

Figure 13-20 Form event handlers

Event Handler	Description
element.onblur	The form *element* has lost the focus
element.onchange	The value of *element* has changed
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

Applying Form Events (continued 1)

- Example: To rerun the calcOrder() function when the user changes the quantity of items to order, use the following event handler:

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

- Use the `onclick` event handler for options buttons and check boxes

Applying Form Events (continued 2)

Figure 13-21 Adding event handlers to form elements

runs the calcOrder() function when the value of the model or qty field changes

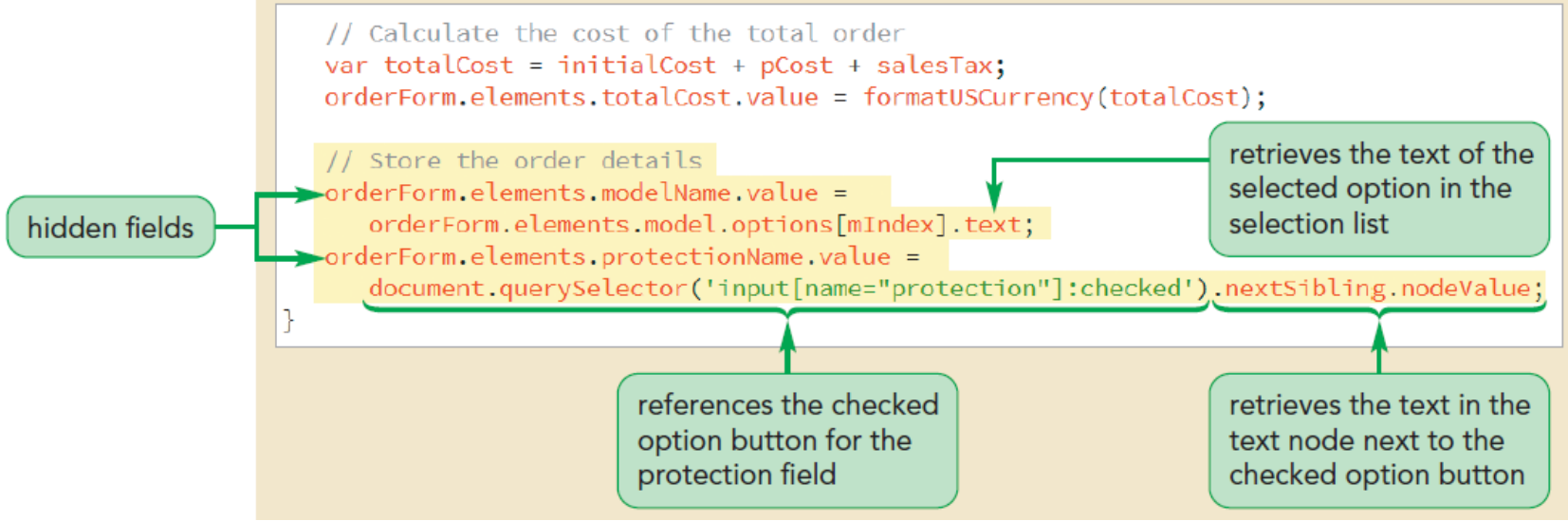
loops through the option buttons for the protection field adding onclick event handlers to each button

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

   // Event handlers for the web form
   orderForm.elements.model.onchange = calcOrder;
   orderForm.elements.qty.onchange = calcOrder;

   var planOptions = document.querySelectorAll('input[name="protection"]');
   for (var i = 0; i < planOptions.length; i++) {
      planOptions[i].onclick = calcOrder;
   }

});
```

Working with Hidden Fields

- Hidden fields are used to store important data
- The data stored in hidden fields is available only to programmers and removed from the user's control

Figure 13-22 Storing data in hidden fields



Appending Form Data

- To append form data to the URL, add `method` and `action` attributes to the `form` element as follows:

```
<form method="get" action="url">
```

where *url* is the URL of the page to which the form data has to be appended

Appending Form Data (continued 1)

- General format of the URL

  `http://server/path/file?field1=value1&field2=value2&field3=value3...`

  where

  - *server* and *path* are the server and path names for the web page
  - *file* is the filename of the web page
  - ? character followed by **query string** contains field names and data values appended to the URL

Examining the location Object

- **Location object** stores the current location of the web document
- Location object is referenced using the following expressions:
  - `window.location`
  - `document.location`
  - `location`
- To load a new page in the current window, use

  `window.location = url`

  where *`url`* is the URL of the new page

Examining the location Object (continued 1)

Figure 13-25 Location properties and methods

Property or Method	Description
`location.hash`	The anchor part of the location's URL including the # symbol
`location.host`	The hostname and port number of the URL
`location.hostname`	The hostname of the URL
`location.href`	The location's entire URL
`location.origin`	The protocol, hostname, and port number of the URL
`location.pathname`	The pathname of the URL
`location.port`	The port number of the URL
`location.protocol`	The protocol of the URL
`location.search`	The query string portion of the URL (all the text after the ? symbol)
`location.assign(`*`newurl`*`)`	Loads a new document with the URL, *`newurl`*
`location.reload(`*`fromServer`*`)`	Reloads the current location from the server where *`fromServer`* is `false` to load the location from the browser's cache (the default) or `true` to reload from the server
`location.replace(`*`newurl`*`)`	Loads a new document with the URL, *`newurl`* and replaces the current location in the browser history with *`newURL`*
`location.toString()`	Returns the href portion of the location's URL

Examining the location Object (continued 2)

Figure 13-26 Extracting the field/value pairs from the location URL

```
*/

window.addEventListener("load", function(){
   // Retrieve the field/value pairs from the URL
   var formData = location.search;
});
```

references the location of the current page

returns the query string portion of the location URL

Working with Text Strings

- Text string objects are created implicitly
  - when storing any text string value within a variable or
  - by extracting a text string from a web form field or a location URL
- Text string objects can be explicitly created using the following object constructor:

```
var stringVar = new String(text);
```

where *stringVar* is a variable that stores the text string and *text* is the text of the string

Extracting Substrings from a Text String

- To extract a character in a text string, use

  string`.charAt(`*i*`)`

  where *string* is the string object and *i* is the index value of the character
- The `charAt()` method extracts only a single character

Extracting Substrings from a Text String (continued)

- To extract **substrings**, use

  string.slice(*start* [,*end*])

  where *start* is the starting index value and *end* is the index value at which the slicing stops

- To create an array of substrings, use

  strArray = string.split(*str*)

  where *strArray* is the array to store the substrings and *str* is a **delimiter** that determines the break between substrings

Searching within a Text String

- To search for the occurrence of substrings within larger text strings, use

  string.indexOf(*str* [,*start*])

  - where *start* indicates the index value where the search starts
- The `indexOf()` method returns the index value of the first occurrence of the substring *str*

Introducing Regular Expressions

- **Regular expression:** Text string that defines a character pattern
- General form:

  ```
  /pattern/
  ```

  where *pattern* is the regular expression code that defines a character pattern

Matching a Substring

- The most basic regular expression is */chars/* where *chars* is the substring text
- To specify the beginning or end of the regular expressions, mark the beginning and end of a text string with the ^ and $ characters, respectively

Setting Regular Expression Flags

- By default, pattern matching stops after the first match is discovered and **flag** is added to the end of a regular expression to override the default
- To perform a global search and match all occurrences of a character pattern within the text string, use `/pattern/g` method
- To make a regular expression insensitive to case, use `/pattern/I` method

Defining Character Types and Character Classes

- Regular expression can match substrings based on the general type of character
- The four general types of characters
  - Alphabetical characters
  - Digits (numbers 0 to 9)
  - **Word characters:** Alphabetical characters, digits, or the underscore character (_)
  - White space characters (blank spaces, tabs, and new lines)

Defining Character Types and Character Classes (continued 1)

Figure 13-30 Character types

Character	Description
\b	a word boundary
\B	not a word boundary
\d	a digit from 0 to 9
\D	any non-digit character
\w	an alphabetical character (in uppercase or lowercase letters), a digit, or an underscore
\W	any non-word character
\s	a white-space character (a blank space, tab, new line, carriage return, or form feed)
\S	any non-white-space character
.	any character

Defining Character Types and Character Classes (continued 2)

- In regular expression language, **word** refers to any string of symbols consisting solely of word characters
- Example: The string “R2D2” is considered a single word but “R2D2&C3PO” is considered two words with the & symbol marking the boundary between the words

Defining Character Types and Character Classes (continued 3)

- **Character class** is used to limit the regular expression to a select group of characters

Figure 13-32 Character classes

Pattern	Description
[*chars*]	Match any character in the *chars* list
[^*chars*]	Do not match any character in the *chars* list
[*char1*-*charN*]	Match characters in the range *char1* through *charN*
[^*char1*-*charN*]	Do not match any characters in the range *char1* through *charN*
[a-z]	Match any lowercase letter
[A-Z]	Match any uppercase letter
[a-zA-Z]	Match any lowercase or uppercase letter
[0-9]	Match any digit
[0-9a-zA-Z]	Match any digit or letter

Specifying Repeating Characters

- To specify the exact number of times to repeat a character, append the character with {*n*} symbol

  where *n* is the number of times to repeat the character

Figure 13-34 Repetition characters

Repetition Characters	Description
*	Repeat 0 or more times
?	Repeat 0 or 1 time
+	Repeat 1 or more times
{*n*}	Repeat exactly *n* times
{*n*,}	Repeat at least *n* times
{*n*,*m*}	Repeat at least *n* times but no more than *m* times

Using Escape Sequences

- **Escape sequence** is used by prefacing the character with the backslash character (\)
- The backslash character indicates that the character that follows should be interpreted as a character and not a command

Using Escape Sequences (continued 1)

Figure 13-36 Escape sequences

Escape Sequence	Represents
\/	/
\\	\
\.	.
*	*
\+	+
\?	?
|	|
\(\)	( )
\{ \}	{ }
\^	^
\$	$
\n	a new line
\r	a carriage return
\t	a tab

Specifying Alternate Patterns and Grouping

- To define two patterns for the same text string, use the "|" character as follows:

  pattern1|*pattern2*

  where *pattern1* and *pattern2* are two distinct patterns

- To group character symbols as a single unit, use the following syntax:

  (*pattern*)

  where *pattern* is a regular expression pattern

Programming with Regular Expressions

- **Regular expression literal** is used to directly enter the code of a regular expression in JavaScript
- For example, the following command stores a regular expression in the regx variable:

```
var regx = /\d{5}-\d{4}/g;
```

Programming with Regular Expressions (continued)

- A regular expression can be defined using `new RegExp()` object constructor as follows:

```
new RegExp(pattern, flags);
```

  where *`pattern`* is the regular expression pattern and *`flags`* are any modifiers added to the pattern

- One of the advantages of using an object constructor is the ability of storing regular expressions as variables

Regular Expression Methods

- Regular expressions have their own methods as they are another type of JavaScript object

Figure 13-39 Regular expression methods

Method	Description
`re.exec(str)`	Searches the text string, *str*, for the character pattern expressed in the regular expression *re*, returning data about the search results in an array
`re.test(str)`	Searches *str* for the character pattern *re*; if a match is found returns the Boolean value `true`
`re.toString()`	Converts the regular expression *re* to a text string
`str.match(re)`	Searches *str* for the character pattern expressed in the regular expression *re*, returning the search results in an array
`str.search(re)`	Searches *str* for a substring matching the regular expression *re*; returns the index of the match, or -1 if no match is found
`str.replace(re, newsubstr)`	Replaces the characters in *str* defined by the regular expression *re* with the text string *newsubstr*
`str.split(re)`	Splits *str* at each point indicated by the regular expression *re*, storing each substring as an item in an array

Replacing URI Encoded Characters

- When the browser finds a character in a field name or value that is reserved for other purposes, it replaces the character with a character code known as **URI encoded character**
- To decode the field value back into its original values, use

  `decodeURIComponent(`*`string`*`)`

  where *`string`* is a text string containing URI–encoded characters

Writing URL Data to a Web Form

- To split the query string at each occurrence of the & or = character in the form data, use the `split()` method as follows:

```
var formFields formData.split(/&=/g);
```

where `/&=/g` is a regular expression that matches every "&" and "=" character in the formData text string

Validating Data with JavaScript

- **Constraint Validation API:** Form validation properties and methods built into JavaScript

Figure 13-47 Constraint Validation API properties and methods

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 *element* is capable of being validated by the browser (regardless of whether the data itself is actually valid)
element.valid	Returns true if *element* contains valid data
element.validationMessage	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 valid; a false value also fires the invalid event

Exploring the ValidityState Object

- **ValidityState** object: Stores the reason for validation fail of a data field

Figure 13-48 Validity properties

Validation 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* contains data that does not match the character pattern specified in the pattern attribute
element.validity.rangeOverflow	The *element* contains data greater than the value specified by the max attribute
element.validity.rangeUnderflow	The *element* contains data less than the value specified by the min attribute
element.validity.stepMismatch	The *element* contains a data value that does not fit the rules determined by the step attribute
element.validity.tooLong	The *element* contains data whose character length exceeds the value of the length attribute
element.validity.typeMismatch	The *element* contains data that 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* does not contain data though it is marked with the required attribute

Creating a Custom Validation Message

- To display the same error message across all browsers, use the following method:

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

  where *msg* is the custom message displayed by the browser when an element is invalid

Figure 13-49 Creating the validateName() function

```
   document.forms.order.elements.salesTax.value = formFields[19];
   document.forms.order.elements.totalCost.value = formFields[21];
} );

function validateName() {
   var cardName = document.getElementById("cardName");
   if (cardName.validity.valueMissing) {
      cardName.setCustomValidity("Enter your name as it appears on the card");
   } else {
      cardName.setCustomValidity("");
   }
}
```

tests if the required value is missing from the cardName field

no pop-up error message when the field is valid

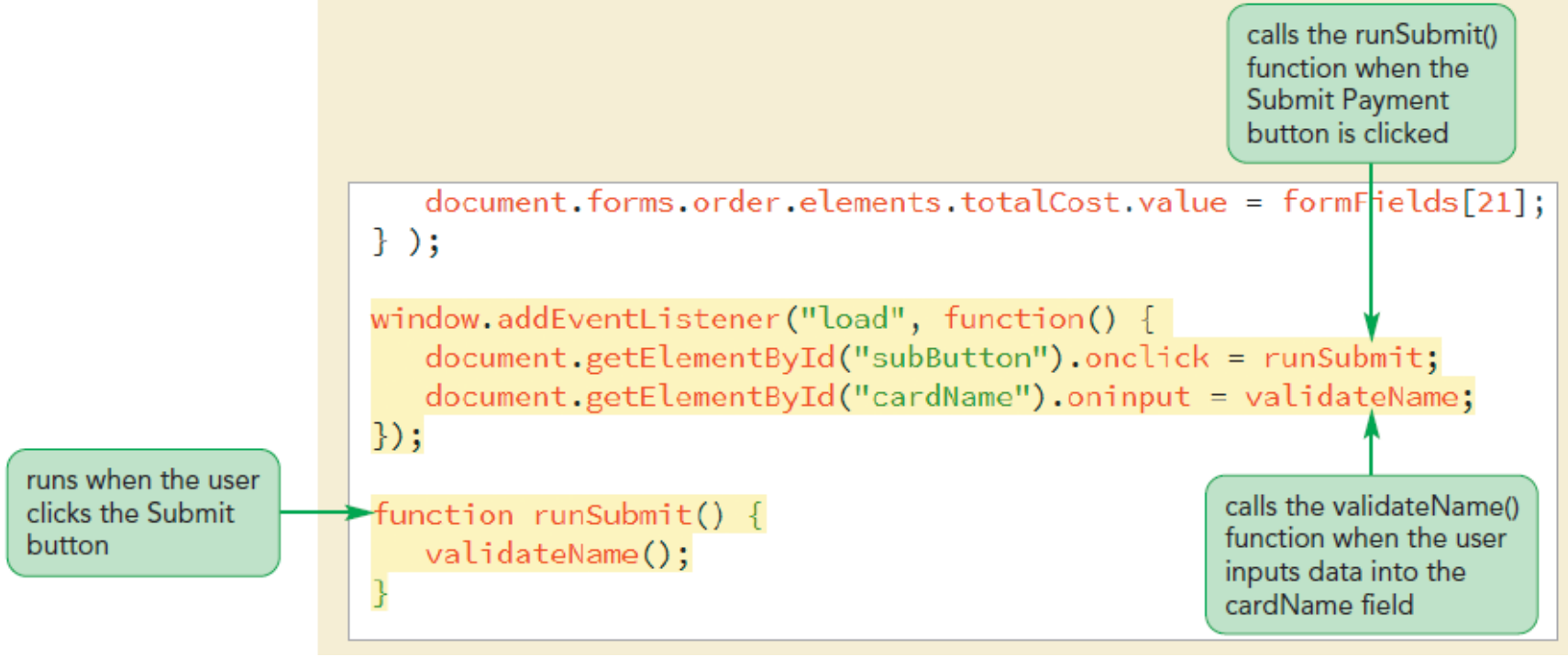
pop-up error message when the field is invalid

Responding to Invalid Data

- To check the validity of form data as it's being inserted, use an event handler or event listener for the `input` 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

Responding to Invalid Data (continued)

Figure 13-50 Calling the validateName() function



Validating Data with Pattern Matching

- Text strings can be matched against a regular expression pattern by adding the following `pattern` attribute to the `input` element:

  `pattern = "`*`regex`*`"`

  where *`regex`* is the regular expression
- Use the `valueMissing` property to test if the field has been left blank and `patternMismatch` property to test for the correct pattern

Validating a Selection List

- Use the `selectedIndex` property to determine the value of the index that is selected
- If the index is 0, the browser will declare the field value as invalid

Testing a Form Field Against a Regular Expression

- Credit card CVC numbers are either 3-digit numbers or 4-digit numbers
- The regular expressions for 4-digit CVC numbers and 3-digit CVC numbers are `/^\d{4}$/` and `/^\d{3}$/` respectively
- Regular expression patterns can be tested using the `test()` method

Testing for Legitimate Card Numbers

- **Luhn Algorithm/Mod10 Algorithm:** Provides a quick validation check on unique identification numbers
- Luhn Algorithm ensures that the sum of the digits in the number meet certain mathematical criteria

Testing for Legitimate Card Numbers (continued 1)

- The steps involved in Luhn Algorithm are as follows:
  - Start from the last digit and move to the left. Divide the alternating digits of the ID number into two groups
  - Add the digits in the first group
  - Double the digits in the second group and then add the sum of the doubled digits

Testing for Legitimate Card Numbers (continued 2)

- Calculate the total sum from the two groups
- If the total sum is evenly divisible by 10, the ID number is considered legitimate, otherwise the ID number is illegitimate

Testing for Legitimate Card Numbers (continued 3)

- To sum the digits found within a text string, use

```
function sumDigits(numStr) {
var digitTotal = 0;
for (var i = 0; i < numStr.length;
i++){digitTotal +=
parseInt(numStr.charAt(i));
}
return digitTotal;
}
```

Testing for Legitimate Card Numbers (continued 4)

Figure 13-60 Creating the luhn() function

```
    return digitTotal;
}

function luhn(idNum) {
    var string1 = "";
    var string2 = "";

    // Retrieve the odd-numbered digits
    for (var i = idNum.length - 1; i >= 0; i-= 2) {
        string1 += idNum.charAt(i);
    }
    // Retrieve the even-numbered digits and double them
    for (var i = idNum.length - 2; i >= 0; i-= 2) {
        string2 += 2*idNum.charAt(i);
    }

    // Return whether the sum of the digits is divisible by 10
    return sumDigits(string1 + string2) % 10 === 0;
}
```

loops through the alternate digit characters starting with the last digit

adds each digit in the string1 variable

adds the double of each digit to the string2 variable

returns a Boolean value indicating whether the total sum is divisible by 10

loops through the alternate digit characters starting with the second-to-last digit

Testing for Legitimate Card Numbers (continued 5)

Figure 13-61 Validating with the Luhn algorithm

tests whether the card number passes the Luhn test

if the number fails validation, displays the validation error message

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