HTML5, CSS3, and JavaScript

6th Edition

Tutorial 10

Exploring Arrays, Loops, and Conditional Statements

Carey

Objectives

- Create an array
- Work with array properties and methods
- Create a program loop
- Work with the `for` loop
- Write comparison and logical operators
- Create a conditional statement
- Use the `if` statement

Introducing the Monthly Calendar

- Calendar should appear in the form of a web table with links to specific events placed within the table cells
- Appearance and placement of the calendar will be set using a CSS style sheet
- The program created should be easily adaptable so that it can be used to create other monthly calendars

Introducing the Monthly Calendar (continued)

Figure 10-2 Linking to the style sheet and JavaScript file

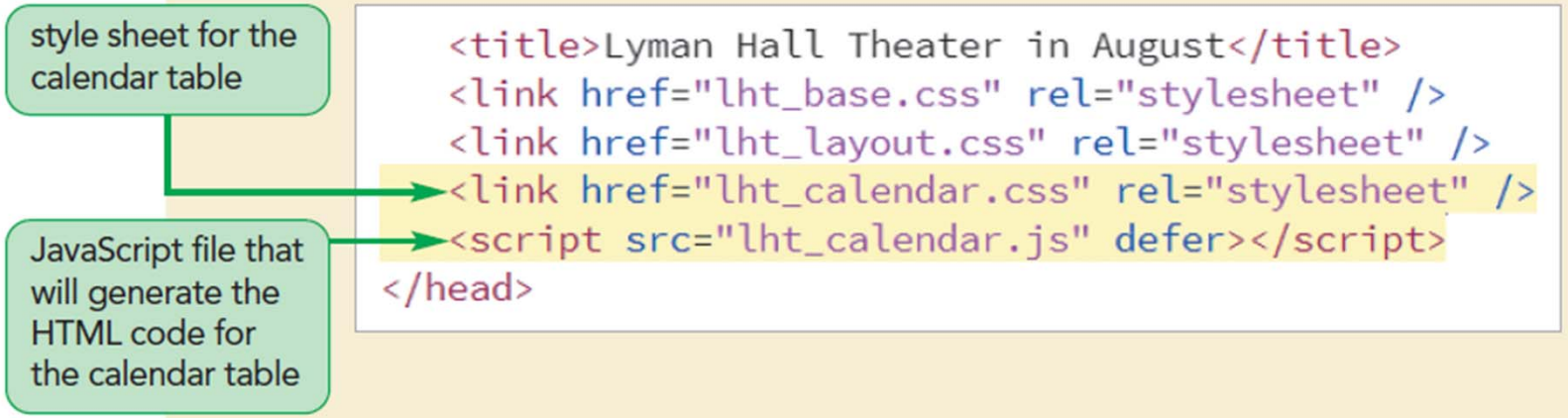
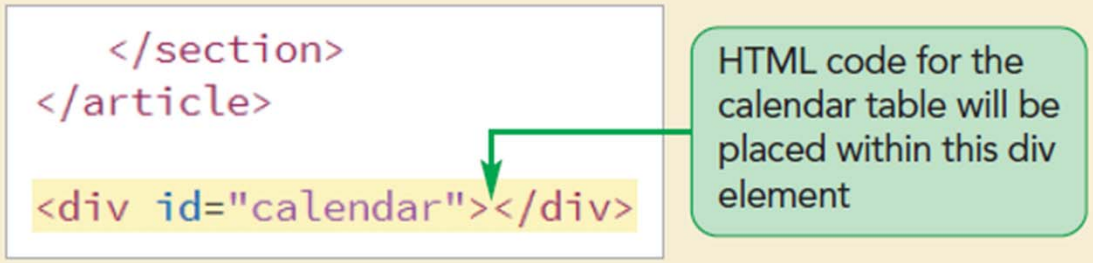


Figure 10-3 Location of the calendar table



Reviewing the Calendar Structure

- Names and IDs assigned to the different parts of the table
  - Entire calendar is set in a web table with the ID *calendar_table*
  - Cell containing the calendar title has the ID *calendar_head*
  - Seven cells containing the days of the week abbreviations all belong to the class *calendar_weekdays*

Reviewing the Calendar Structure (continued)

- All cells containing the dates of the month belong to the class *calendar_dates*
- The cell containing the current date has the ID *calendar_today*

- Class and ID designations make it easier for page developers to assign different styles to the different parts of the calendar

Adding the calendar() Function

- Place the commands that generate the calendar within a single function named createCalendar()
- Initial code to generate the calendar

```
var thisDay = new Date("August 24, 2018");
document.getElementById("calendar").innerHTML = createCalendar(thisDay);
```

Adding the calendar() Function (continued)

```
function createCalendar(calDate) {
var calendarHTML = "<table
id='calendar_table'>";
calendarHTML += "</table>";
return calendarHTML;
}
```

- `thisDay` variable: Stores the current date

Introducing Arrays

- `getMonth()` method: Extracts a month number
- `getFullYear()` method: Extracts the four-digit year value
- The `Date` method does not return the name of the month
- One way to associate each month number with a month name is by using an array

Introducing Arrays (continued 1)

- Array: Collection of values organized under a single name
- **Index**: The number that each individual value is associated with and that distinguishes it from other values in the array
- Array values are referenced using the expression *array*[*i*]

  where *array* is the name of the array and *i* is the index of a specific value in the array

Introducing Arrays (continued 2)

- Index values start with 0 so that the initial item in an array has an index value of 0
- Second item in the array will have an index value of 1, and so on
- Example

  The expression `monthName[4]` references the fifth (not the fourth) item in the `monthName` array

Creating and Populating an Array

- To create an array, apply the object constructor

  `var array = new Array(length);`

  where *array* is the name of the array and *length* is the number of items in the array
- The *length* value is optional; if the *length* parameter is omitted then the array expands automatically as more items are added to it

Creating and Populating an Array (continued 1)

- An array can be populated with values by specifying both the array name and the index number of the array item
- Command to set the value of a specific item in an array

  ```
  array[i] = value;
  ```

  where *value* is the value assigned to the array item with the index value *i*

Creating and Populating an Array (continued 2)

- Populate the entire array in a single statement using the following command:

  ```
  var array = new Array(values);
  ```

  where *values* is a comma-separated list of the values in the array

- Example

  ```
  var monthName = new Array("January",
  "February", "March", "April", "May",
  "June", "July", "August", "September",
  "October", "November", "December");
  ```

Creating and Populating an Array (continued 3)

- **Array literal:** Creates an array in which the array values are a comma-separated list within a set of square brackets

  ```
  var array = [values];
  ```

  where *values* are the values of the array

- Example

  ```
  var monthName = [“January”, February”,
  “March”, “April”, “May”, “June”,
  “July”, “August”, “September”,
  “October”, “November”, “December”];
  ```

Creating and Populating an Array (continued 4)

- Array values need not be of the same data type
- Mix of numeric values, text strings, and other data types within a single array is allowed
- Example

```
var x = [“April”, 3.14, true, null];
```

Working with Array Length

- JavaScript array automatically expands in length as more items are added
- Apply the following `length` property to determine the array's current size:

  array`.length`

  where *array* is the name of the array
- Value returned by the `length` property is equal to one more than the highest index number in the array

Working with Array Length (continued)

- **Sparse arrays**: Created in JavaScript in which some of the array values are undefined
- The `length` value is not always the same as the number of array values
- Occurs frequently in database applications
- Value of the `length` property cannot be reduced without removing items from the end of the array

Reversing an Array

- Items are placed in an array either in the order in which they are defined or explicitly by index number, by default
- JavaScript supports two methods for changing the order of the array items
  - `reverse()`
  - `sort()`
- `reverse()`: Reverses the order of items in an array, making the last items first and the first items last

Sorting an Array

- `sort()`: Rearranges array items in alphabetical order
- The `sort()` method when applied to numeric values will sort the values in order by their leading digits, rather than by their numerical values

Sorting an Array (continued 1)

- **Compare function**: Compares values of two adjacent array items
- General form of a compare function is

```
function fname(a, b) {
return a negative, positive, or 0 value
}
```

where *fname* is the name of the compare function and *a* and *b* are parameters that represent a pair of array values

Sorting an Array (continued 2)

- Based on comparison of two adjacent array item values, the function returns a negative, positive, or zero value
  - If a negative value is returned, then *a* is placed before *b* in the array
  - If a positive value is returned, then *b* is placed before *a*
  - If a zero value is returned, *a* and *b* retain their original positions

Sorting an Array (continued 3)

- Function to sort numeric values in ascending order

```
function ascending(a, b) {
return a - b;
}
```

- Function to sort numbers in descending order

```
function descending(a, b) {
return b - a;
}
```

Sorting an Array (continued 4)

- The compare function is applied to the `sort()` method as follows

  `array``.sort(`*`fname`*`)`

  where *`fname`* is the name of the compare function
- Example

  `x.sort(ascending)`

Extracting and Inserting Array Items

- **Subarray**: Section of an array
- To create a subarray use `slice()` method

  array`.slice(`*start, stop*`)`

  where *`start`* is the index value of the array item at which the slicing starts and *`stop`* is the index value at which the slicing ends
- The *`stop`* value is optional; if it is omitted, the array is sliced to its end

Extracting and Inserting Array Items (continued)

- `splice()`: Removes and inserts array items
  `array``.splice(`*`start, size, values`*`)`

  `start` is the starting index in the array, *`size`* is the number of array items to remove after the *`start`* index, and *`values`* is an optional comma-separated list of values to insert into the array
- If no *`values`* are specified, the splice method simply removes items from the array
- Always alters the original array

Using Arrays as Data Stacks

- **Stack**: Arrays can be used to store information in a data structure
- New items are added to the top of the stack or to the end of the array
- A stack data structure employs the **last-in first-out (LIFO)** principle
- In the **LIFO** principle the last items added to the stack are the first ones removed

Using Arrays as Data Stacks (continued 1)

- `push()` method: Appends new items to the end of an array

  `array``.push(`*`values`*`)`

  where *`values`* is a comma-separated list of values to be appended to the end of the array

- `pop()` method: Removes or **unstacks** the last item

  `array``.pop()`

Using Arrays as Data Stacks (continued 2)

- **Queue:** Employs the **first-in-first-out (FIFO)** principle in which the first item added to the data list is the first removed
- Similar to a stack
- `shift()` method: Removes the first array item
- `unshift()` method: Inserts new items at the front of the array

Using Arrays as Data Stacks (continued 3)

Figure 10-7 Array methods

Method	Description
copyWithin(*target*, *start*[, *end*])	Copies items within the array to the *target* index, starting with the *start* index and ending with the optional *end* index
concat(*array1*, *array2*,...)	Joins the array to two or more arrays, creating a single array containing the items from all the arrays
fill(*value*[, *start*][, *end*])	Fills the array with items having the value *value*, starting from the *start* index and ending at the *end* index
indexOf(*value*[, *start*])	Searches the array, returning the index number of the first element equal to *value*, starting from the optional *start* index
join(*separator*)	Joins all items in the array into a single text string; the array items are separated using the text in the *separator* parameter; if no *separator* is specified, a comma is used
lastIndexOf(*value*[, *start*])	Searches backward through the array, returning the index number of the first element equal to *value*, starting from the optional *start* index
pop()	Removes the last item from the array
push(*values*)	Appends the array with new items, where *values* is a comma-separated list of item values
reverse()	Reverses the order of items in the array
shift()	Removes the first item from the array
slice(*start*, *stop*)	Extracts the array items starting with the *start* index up to the *stop* index, returning a new subarray
array.splice(*start*, *size*, *values*)	Extracts *size* items from the array starting with the item with the index *start*; to insert new items into the array, specify the array items in a comma-separated *values* list
array.sort(*fname*)	Sorts the array where *fname* is the name of a function that returns a positive, negative, or 0 value; if no function is specified, *array* is sorted in alphabetical order
array.toString()	Converts the contents of the array to a text string with the array values in a comma-separated list
array.unshift(*values*)	Inserts new items at the start of the array, where *values* is a comma-separated list of new values

Working with Program Loops

- **Program loop:** Set of commands executed repeatedly until a stopping condition is met
- Two commonly used program loops in JavaScript are
  - `for` loops
  - `while` loops

Exploring the `for` Loop

- In a `for` loop, a variable known as a counter variable is used to track the number of times a block of commands is run
- When the counter variable reaches or exceeds a specified value, the `for` loop stops
- General structure of a `for` loop

```
for (start; continue; update) {
commands
}
```

Exploring the `for` Loop (continued 1)

where *start* is an expression that sets the initial value of a counter variable

continue is a Boolean expression that must be true for the loop to continue

update is an expression that indicates how the value of the counter variable should change each time through the loop

commands are the JavaScript statements that are run for each loop

Exploring the for Loop (continued 2)

- **Command block:** Collection of commands that is run each time through a loop
- Indicated by its opening and closing curly braces { }
- One `for` loop can be nested within another

Figure 10-10 for loop counter values

for Loop	Counter Values
`for (var i = 1; i <= 5; i++)`	i = 1, 2, 3, 4, 5
`for (var i = 5; i > 0; i--)`	i = 5, 4, 3, 2, 1
`for (var i = 0; i <= 360; i+=60)`	i = 0, 60, 120, 180, 240, 360
`for (var i = 1; i <= 64; i*=2)`	i = 1, 2, 4, 8, 16, 32, 64

Exploring the `while` Loop

- **`while` loop:** Command block that is run as long as a specific condition is met
- Condition in a `while` loop does not depend on the value of a counter variable
- General syntax for the `while` loop

```
while (continue) {
commands
}
```

where *`continue`* is a Boolean expression

Exploring the do/while Loop

- **do/while** loop: Generally used when the program loop should run at least once before testing stopping condition
- Tests the condition to continue the loop right after the latest command block is run
- Structure of the do/while loop

```
do {
commands
}
while (continue);
```

Comparison and Logical Operators

- **Comparison operator:** Compares the value of one expression to another returning a Boolean value indicating whether the comparison is true or false

Figure 10-11 Comparison operators

Operator	Example	Description
==	x == y	Tests whether x is equal in value to y
===	x === y	Tests whether x is equal in value to y and has the same data type
!=	x != y	Tests whether x is not equal to y
>	x > y	Tests whether x is greater than y
>=	x >= y	Tests whether x is greater than or equal to y
<	x < y	Tests whether x is less than y
<=	x <= y	Tests whether x is less than or equal to y

Comparison and Logical Operators (continued)

- **Logical operators** allow several expressions to be connected
- Example: The logical operator && returns a value of true only if both of the expressions are true

Figure 10-12 Logical operators

Operator	Definition	Example	Description
&&	and	(x === 5) && (y === 8)	Tests whether x is equal to 5 and y is equal to 8
||	or	(x === 5) || (y === 8)	Tests whether x is equal to 5 or y is equal to 8
!	not	!(x < 5)	Tests whether x is not less than 5

Program Loops and Arrays

- Program loops: Cycle through different values contained within an array
- General structure to access each value from an array using a `for` loop

```
for (var i = 0; i < array.length; i++)
{commands involving array[i]
}
```

where *`array`* contains the values to be looped through and *`i`* is the counter variable used in the loop

Array Methods to Loop Through Arrays

- JavaScript supports several methods to loop through the contents of an array without having to create a program loop structure
- Each of these methods is based on calling a function that will be applied to each item in the array

Array Methods to Loop Through Arrays (continued 1)

- The general syntax

  array.*method*(*callback* [, *thisArg*])

  where *array* is the array, *method* is the array method, and *callback* is the name of the function that will be applied to each array item
- *thisArg*: A *callback* optional argument that can be included to pass a value to the function

Array Methods to Loop Through Arrays (continued 2)

- General syntax of the callback function

```
function callback(value [, index, array]) {
commands }
```

  where *value* is the value of the array item during each pass through the array, *index* is the numeric index of the current array item, and *array* is the name of the array

- Only the *value* parameter is required; the other are optional

Running a Function for Each Array Item

- `forEach()` method: Runs a function for each item in the array
- General syntax

  `array``.forEach(`*`callback`*` [, `*`thisArg`*`])`

  where *`callback`* is the function that is applied to each item in the array

Mapping an Array

- `map()` method: The function it calls returns a value that can be used to map the contents of an existing array into a new array
- Example

```
var x = [3, 8, 12];
var y = x.map(DoubleIt);
function DoubleIt(value) {
return 2*value;
}
```

Filtering an Array

- `filter()` method: Extracts array items that match some specified condition

  `array``.filter(`*`callback`*` [, `*`thisArg`*`])`

  where *`callback`* is a function that returns a Boolean value of `true` or `false` for each item in the array
- Array items that return a value of `true` are copied into the new array

Filtering an Array (continued 1)

- `every()` method: Returns the value `true` if every item in the array matches the condition specified by the callback function; if otherwise, returns `false`

  `array``.every(`*`callback`*` [, `*`thisArg`*`])`
- `some()` method: Returns a value of `true` if some array items match a condition specified in the function, but otherwise returns `false` if none of the array items match the condition specified in the function

Filtering an Array (continued 2)

Figure 10-16 Array methods to loop through arrays

Array Method	Description
every(*callback* [, *thisArg*])	Tests whether the condition returned by the *callback* function holds for all items in *array*; in all array methods, the optional *thisArg* parameter is used to pass values to the *callback* function
filter(*callback* [, *thisArg*])	Creates a new array populated with the elements of *array* that return a value of *true* from the *callback* function
forEach(*callback* [, *thisArg*])	Applies the *callback* function to each item in *array*
map(*callback* [, *thisArg*])	Creates a new array by passing the original array items to the *callback* function, which returns the mapped value of the array items
reduce(*callback* [, *thisArg*])	Reduces *array* by keeping only those items that return a value of *true* from the *callback* function
reduceRight(*callback* [, *thisArg*])	Reduces *array* from the last element by keeping only those items that return a value of *true* from the *callback* function
some(*callback* [, *thisArg*])	Tests whether the condition returned by the *callback* function holds for at least one item in *array*
find(*callback* [, *thisArg*])	Returns the value of the first element in the array that passes a test in the *callback* function
findIndex(*callback* [, *thisArg*])	Returns the index of the first element in the array that passes a test in the *callback* function

Introducing Conditional Statements

- **Parallel array:** Each entry in the array matches or is parallel to an entry in the other array
- **Conditional statement:** Runs a command or command block only when certain circumstances are met

Exploring the `if` Statement

- The most common conditional statement is the `if` statement
- General structure of the `if` statement

```
if (condition) {
commands
}
```

where *`condition`* is a Boolean expression that is either true or false and *`commands`* is the command block that is run if *`condition`* is true

Exploring the `if` Statement (continued)

- Conditional statement uses the same comparison and logical operators that are used with program loops
- **Modulus operator:** Returns the integer remainder after dividing one integer by another

Nesting `if` Statements

- One `if` statement is nested inside another

Figure 10-19 Process to calculate leap years

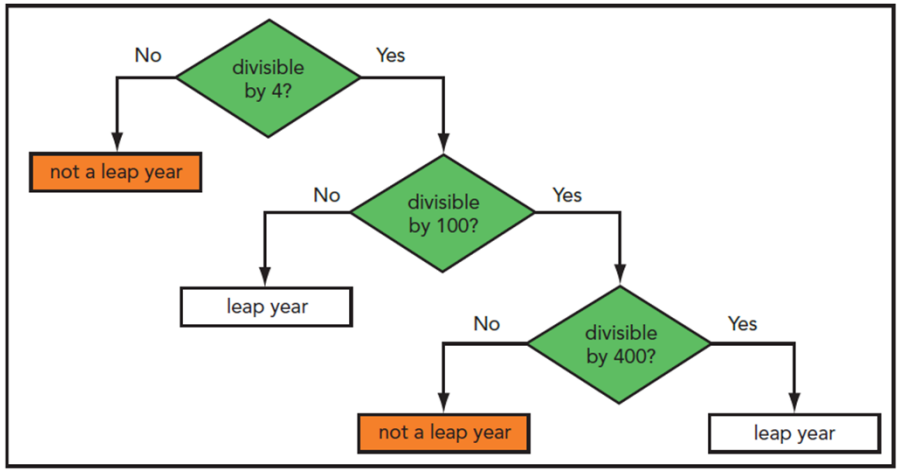


Figure 10-20 Inserting a nested if statement

if the year is divisible by 4 and either not divisible by 100 or divisible by 400, it's a leap year

```
// Revise the days in February for leap years
if (thisYear % 4 === 0) {
   if ((thisYear % 100 != 0) || (thisYear % 400 === 0)) {
      dayCount[1] = 29;
   }
}
```

Exploring the `if else` Statement

- `if else` statement: Chooses between alternate command blocks
- It runs one command block if the conditional expression is true and a different command block if the expression is false
- General structure

```
if (condition) {
     commands if condition is true }
  else {
     commands if condition is false }
```

Using Multiple `else if` Statements

- Structure of an `if else` statement is:

```
if (condition1) {
commands1
} else if (condition2) {
commands2
}...
else {
default commands}
```

where *condition 1, condition 2,…* are the different conditions to be tested

Completing the Calendar App

- The completed calendar app must do the following:
  - Calculate the day of the week in which the month starts
  - Write blank table cells for the days before the first day of the month
  - Loop through the days of the current month, writing each date in a different table cell and starting a new table row on each Sunday

Completing the Calendar App (continued)

Figure 10-21 Inserting the calDays() function and comments

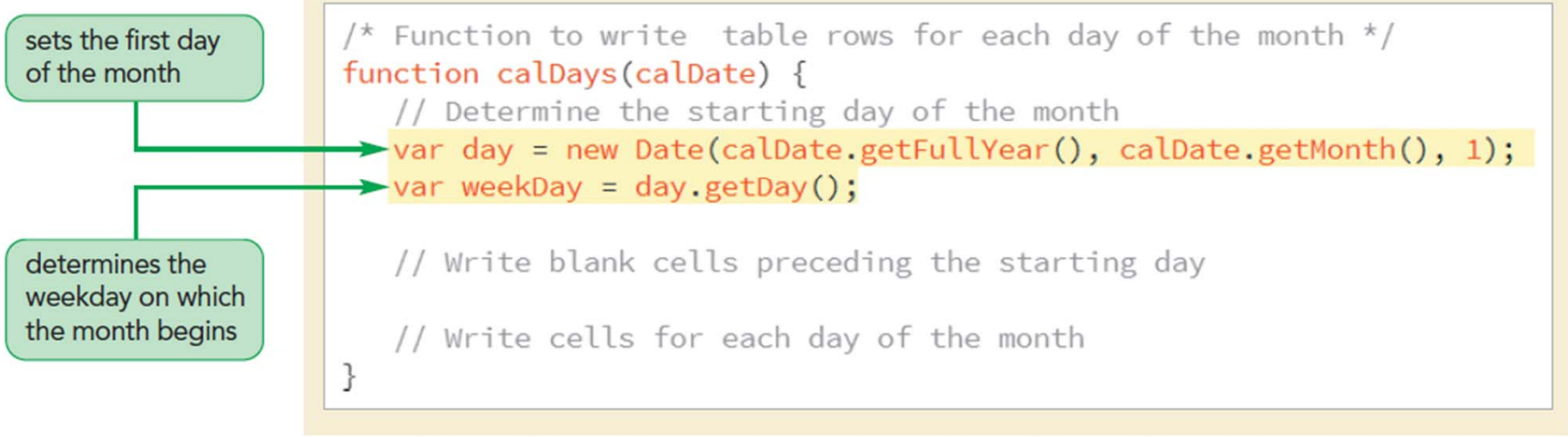
```
/* Function to write  table rows for each day of the month */
function calDays(calDate) {
   // Determine the starting day of the month

   // Write blank cells preceding the starting day

   // Write cells for each day of the month
}
```

Setting the First Day of the Month

Figure 10-22 Calculating the start day of the month



Placing the First Day of the Month

Figure 10-23 Inserting blank cells for the days that precede the start of the month

inserts opening <tr> tag for the initial table row

inserts a blank table cell for each weekday prior to the first of the month

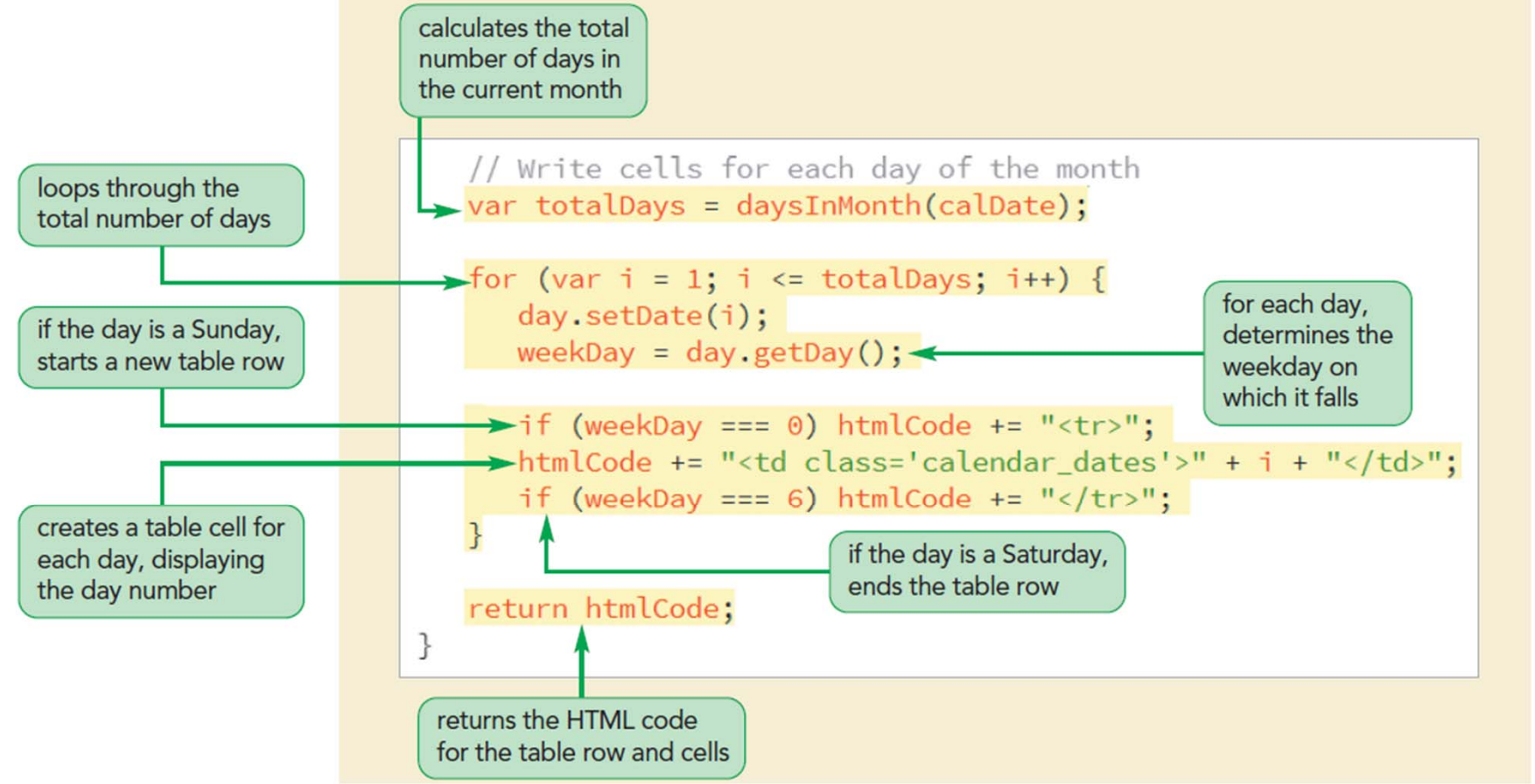
```
/* Function to write  table rows for each day of the month */
function calDays(calDate) {
   // Determine the starting day of the month
   var day = new Date(calDate.getFullYear(), calDate.getMonth(), 1);
   var weekDay = day.getDay();

   // Write blank cells preceding the starting day
   var htmlCode = "<tr>";
   for (var i = 0; i < weekDay; i++) {
      htmlCode += "<td></td>";
   }

   // Write cells for each day of the month
}
```

Writing the Calendar Days

Figure 10-24 Writing the HTML code for the table row and cells



Highlighting the Current Date

Figure 10-27 Highlighting the current date in the calendar

stores the current day in the highlightDay variable

if the day is the highlight day, write a table cell with the id 'calendar_today'

otherwise write a table cell with no id value

```
   // Write cells for each day of the month
   var totalDays = daysInMonth(calDate);

   var highlightDay = calDate.getDate();
   for (var i = 1; i <= totalDays; i++) {
      day.setDate(i);
      weekDay = day.getDay();

      if (weekDay === 0) htmlCode += "<tr>";
      if (i === highlightDay) {
         htmlCode += "<td class='calendar_dates' id='calendar_today'>" + i + "</td>";
      } else {
         htmlCode += "<td class='calendar_dates'>" + i + "</td>";
      }
      if (weekDay === 6) htmlCode += "</tr>";
   }

   return htmlCode;
}
```

Displaying Daily Events

Figure 10-31 Displaying events for each day of the month

```
if (weekDay === 0) htmlCode += "<tr>";
if (i === highlightDay) {
   htmlCode += "<td class='calendar_dates' id='calendar_today'>" + i + dayEvent[i] + "</td>";
} else {
   htmlCode += "<td class='calendar_dates'>" + i + dayEvent[i] + "</td>";
}
if (weekDay === 6) htmlCode += "</tr>";
```

displays the event for the day

Exploring the `break` Command

- `break` statement: Terminates any program loop or conditional statement
- Used anywhere within the program code
- When a `break` statement is encountered, control is passed to the statement immediately following it
- It is most often used to exit a program loop before the stopping condition is met

Exploring the `continue` Command

- **`continue` statement:** Stops processing the commands in the current iteration of the loop and continues on to the next iteration
- It is used to jump out of the current iteration if a missing or null value is encountered

Exploring `Statement` Labels

- **`Statement` labels:** Identifies statements in JavaScript code
- Can be referenced elsewhere in a program
- Syntax of the `statement` label

  `label``: `*`statements`*

  where *`label`* is the text of the label and *`statements`* are the statements identified by the label