HTML5, CSS3, and JavaScript

6th Edition

Tutorial 11

Working with Events and Styles

Carey

Objectives

- Create an event handler
- Reference an attribute from a page element
- Change the inline style of a page element
- Create code for mouse events
- Create code for keyboard events

Objectives (continued)

- Design and apply custom cursors
- Create and apply anonymous functions
- Work with alert, confirm, and prompt dialog boxes

Introducing JavaScript Events

- JavaScript programs run in response to events
- **Events:** Actions initiated by the user or by the browser
- Example:
  - Clicking an object on a form
  - Closing a web page
- JavaScript events can be used to build a Hanjie puzzle, i.e., a grid in which each grid cell is either filled or left empty

Introducing JavaScript Events (continued)

- The Hanjie puzzle app contains three puzzles named puzzle1, puzzle2, and puzzle3

Figure 11-4 HTML code for the puzzle1 table generated by the drawPuzzle() function

column and row totals are calculated by the drawPuzzle() function

filled cells are placed in the filled class

empty cells are placed in the empty class

```
<table id="hanjieGrid">
    <caption>Triangle (Easy)</caption>
    <tr>
      <th></th><th class="cols">5</th><th class="cols">4</th>
      <th class="cols">3</th><th class="cols">2</th><th class="cols">1</th>
    </tr>
    <tr>
      <th class="rows">5</th><td class="filled"></td><td class="filled"></td>
      <td class="filled"></td><td class="filled"></td><td class="filled"></td>
    </tr>
    <tr>
      <th class="rows">4</th><td class="filled"></td><td class="filled"></td>
      <td class="filled"></td><td class="filled"></td><td class="empty"></td>
    </tr>
    <tr>
      <th class="rows">3</th><td class="filled"></td><td class="filled"></td>
      <td class="filled"></td><td class="empty"></td><td class="empty"></td>
    </tr>
    <tr>
      <th class="rows">2</th><td class="filled"></td><td class="filled"></td>
      <td class="empty"></td><td class="empty"></td><td class="empty"></td>
    </tr>
    <tr>
      <th class="rows">1</th><td class="filled"></td><td class="empty"></td>
      <td class="empty"></td><td class="empty"></td><td class="empty"></td>
    </tr>
</table>
```

Creating an Event Handler

- **Event handler:** A property that controls how an object will respond to an event
- Event handler waits until the event occurs and then responds by running a function or command block to execute an action

Creating an Event Handler (continued 1)

- Event handlers can be added to a page element using the following attribute:

  `<element oneevent = "script">`

  where *element* is the element in which the event occurs, *event* is the name of the event, and *script* are the commands that the browser runs in response to the event

Creating an Event Handler (continued 2)

- Event handlers can also be defined as object properties using the command

  `object.on`*`event`* `=` *`function`*`;`

  where *`object`* is the object in which the event occurs, *`event`* is the name of the event, and *`function`* is the name of a function run in response to the event

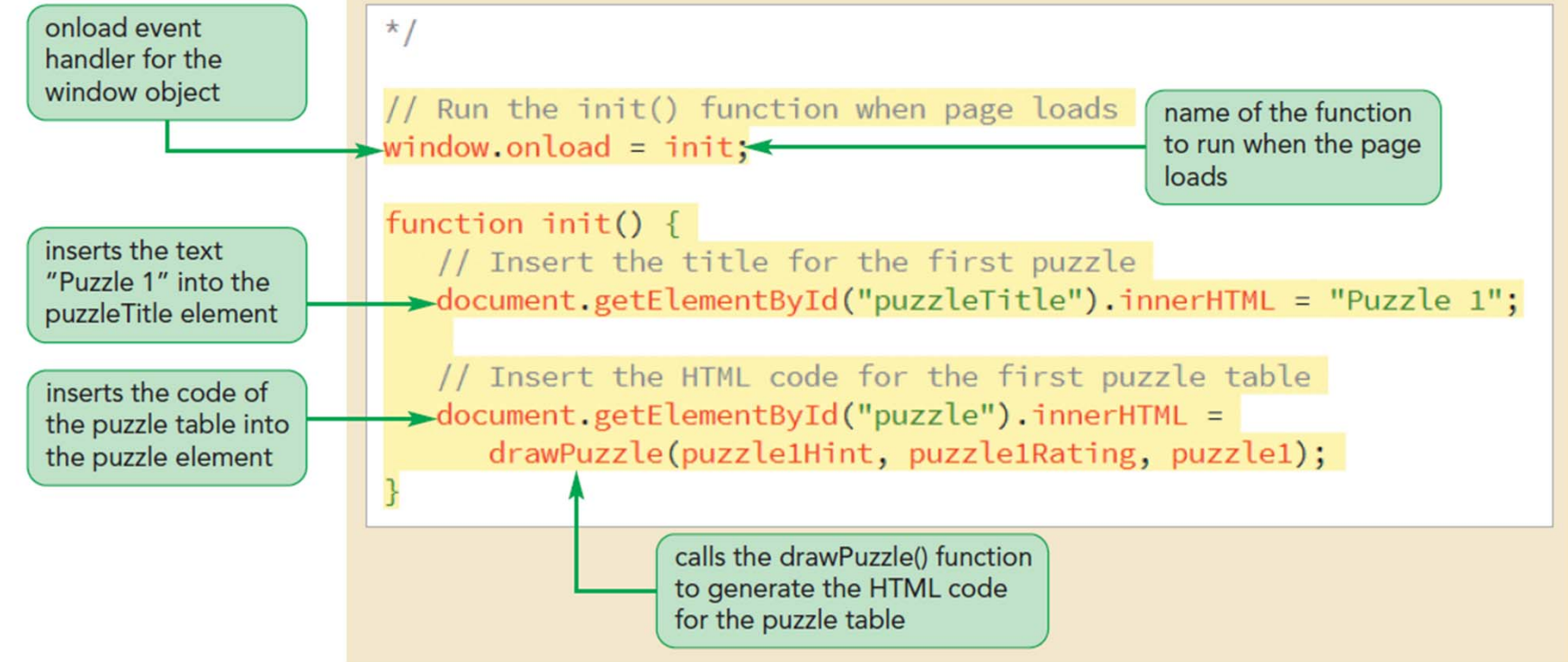
Creating an Event Handler (continued 3)

Figure 11-6 Event handlers for the browser window

Event Handler	Run Script
`onbeforeunload`	when page is about to be unloaded by the browser
`oncopy`	when the user copies the content of an element
`oncut`	when the user cuts the content of an element
`onerror`	when an error occurs while loading an external file, such as an image or a video clip
`onload`	after the page has finished loading
`onpaste`	when the user pastes some content into an element
`onresize`	when the browser window is resized
`onunload`	when the page is unloaded by the browser (or the browser window is closed)

Creating an Event Handler (continued 4)

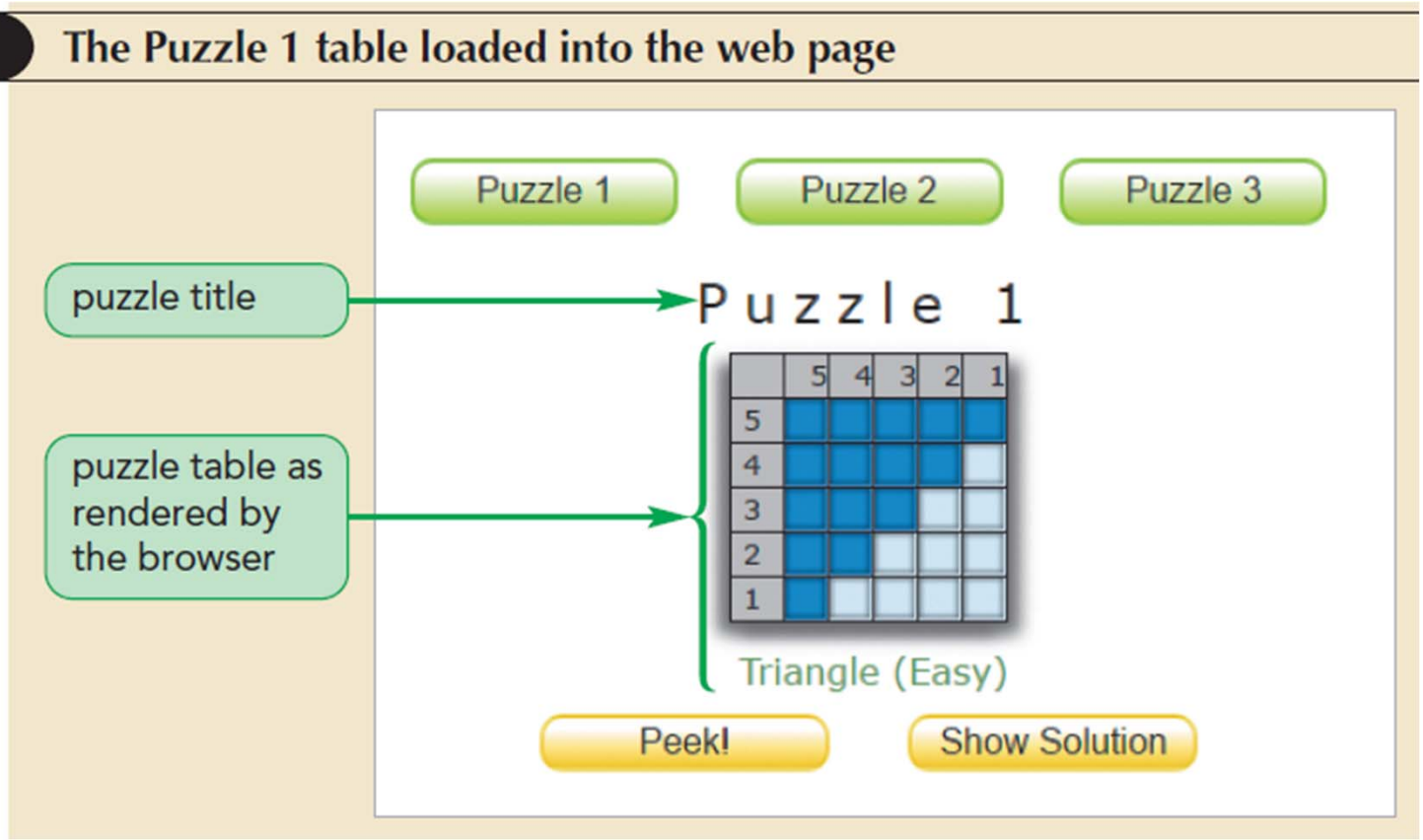
Figure 11-7 Applying the onload event handler




```
*/

// Run the init() function when page loads
window.onload = init;

function init() {
   // Insert the title for the first puzzle
   document.getElementById("puzzleTitle").innerHTML = "Puzzle 1";

   // Insert the HTML code for the first puzzle table
   document.getElementById("puzzle").innerHTML =
      drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);
}
```

Creating an Event Handler (continued 5)

Figure 11-8 The Puzzle 1 table loaded into the web page



Creating an Event Handler (continued 6)

- Add an event handler to the buttons in the Figure 11-8
- Apply the following `onclick` event handler to respond to a mouse click:

```
object.onclick = function;
```

where *`object`* is the page element that is being clicked and *`function`* is the function run in response to the `click` event

Creating an Event Handler (continued 7)

Figure 11-9 Assigning the onclick event handler

```
function init() {
   // Insert the title for the first puzzle
   document.getElementById("puzzleTitle").innerHTML = "Puzzle 1";

   // Insert the HTML code for the first puzzle table
   document.getElementById("puzzle").innerHTML =
      drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);

   //Add event handlers for the puzzle buttons
   var puzzleButtons = document.getElementsByClassName("puzzles");
   for (var i = 0; i < puzzleButtons.length; i++) {
      puzzleButtons[i].onclick = swapPuzzle;
   }
}
```

creates an object collection of all elements in the puzzles class

loops through every object in the puzzleButtons collection

attaches an onclick event handler to each puzzle button

runs the swapPuzzle() function when the button is clicked

Creating an Event Handler (continued 8)

- The challenge lies in determining which button was clicked
- There is no way of knowing which puzzle to load into the page without knowing which button activated the `onclick` event handler
- This information can be determined using the event object

Using the Event Object

- Event object is an object that contains properties and methods associated with an event
- Example: The action of clicking a mouse button generates an event object containing information such as which mouse button was clicked, where in the page it was clicked, the time at which it was clicked, and so forth

Using the Event Object (continued 1)

- Event object is passed as an argument to the function handling the event to retrieve the information contained in the event object
- Example:

```
function myFunction(evt) {
function code
}
```

where *evt* is the name assigned to the parameter receiving the event object from the event handler

Using the Event Object (continued 2)

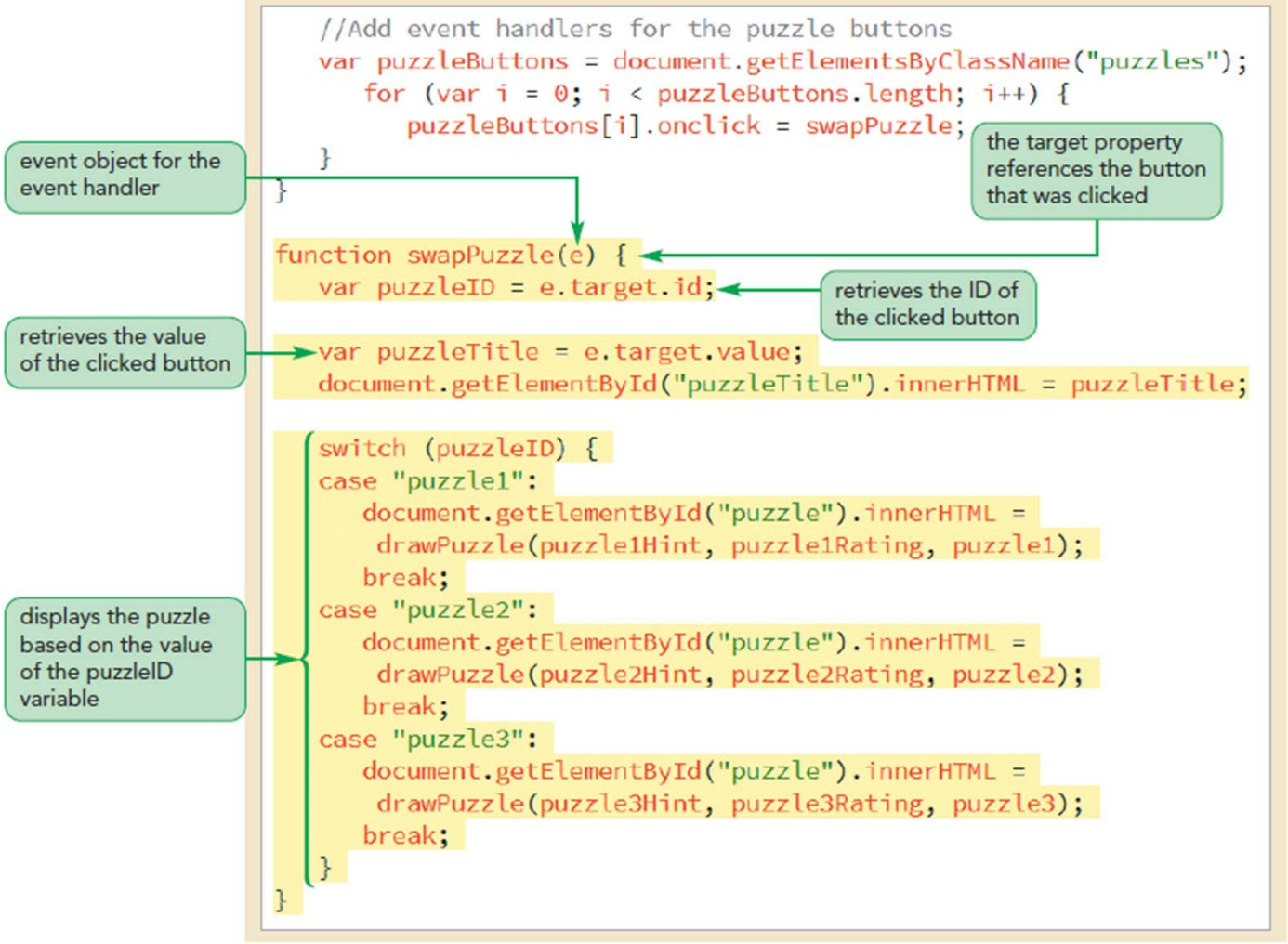
Figure 11-10 Event object properties and methods

Property	Description
`evt.bubbles`	Returns a Boolean value indicating whether the event is bubbling up through the object hierarchy, where `evt` is the event object for the event
`evt.cancelable`	Returns a Boolean value indicating whether the event can have its default action canceled
`evt.currentTarget`	Returns the object that is currently experiencing the event
`evt.defaultPrevented`	Returns a Boolean value indicating whether the `preventDefault()` method was called for the event
`evt.eventPhase`	Returns the phase of the event propagation the event object is currently at, where 0 = NONE, 1 = CAPTURING_PHASE, 2 = AT_TARGET, and 3 = BUBBLING_PHASE
`evt.isTrusted`	Returns a Boolean value indicating whether the event is trusted by the browser
`evt.target`	Returns the object in which the event was initiated
`evt.timeStamp`	Returns the time (in milliseconds) when the event occurred
`evt.type`	Returns the type of the event
`evt.view`	Reference the browser window in which the event occurred

Method	Description
`evt.preventDefault()`	Cancels the default action associated with the event
`evt.stopImmediatePropagation()`	Prevents other event listeners of the event from being called
`evt.stopPropagation()`	Prevents further propagation of the event in the capturing and bubbling phase

Using the Event Object (continued 3)

Figure 11-11 Creating the swapPuzzle() function




```
   //Add event handlers for the puzzle buttons
   var puzzleButtons = document.getElementsByClassName("puzzles");
      for (var i = 0; i < puzzleButtons.length; i++) {
         puzzleButtons[i].onclick = swapPuzzle;
   }
}

function swapPuzzle(e) {
   var puzzleID = e.target.id;

   var puzzleTitle = e.target.value;
   document.getElementById("puzzleTitle").innerHTML = puzzleTitle;

   switch (puzzleID) {
   case "puzzle1":
      document.getElementById("puzzle").innerHTML =
       drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);
      break;
   case "puzzle2":
      document.getElementById("puzzle").innerHTML =
       drawPuzzle(puzzle2Hint, puzzle2Rating, puzzle2);
      break;
   case "puzzle3":
      document.getElementById("puzzle").innerHTML =
       drawPuzzle(puzzle3Hint, puzzle3Rating, puzzle3);
      break;
   }
}
```

Exploring Object Properties

- JavaScript object properties that mirror HTML attributes follow certain conventions
- All JavaScript properties must begin with a lowercase letter
- If the HTML attribute consists of multiple words, JavaScript property follows a format known as **camel case**, i.e., the initial word is in lowercase and the first letter of each subsequent word is in uppercase

Exploring Object Properties (continued)

- If the name of the HTML attribute is a reserved JavaScript name or keyword, the corresponding JavaScript property is often prefaced with the text string `html`
- `class` attribute is an exception to this convention because the class name is reserved by JavaScript for other purposes
- References to the HTML class attribute use the `className` property

Object Properties and Inline Styles

- Inline styles for each page element can be applied using the following `style` attribute:

  `<`*`element`* `style = “`*`property`*`:`*`value`*`”>` …

  where

  – *`element`* is the page element

  – *`property`* is a CSS style property

  – *`value`* is the value assigned to that property

Object Properties and Inline Styles (continued)

- The equivalent command in JavaScript is

  ```
  object.style.property = "value";
  ```

  with the *property* style written in camel case
- Inline styles have precedence over style sheets

Creating Object Collections with CSS Selectors

- Change the background color of all table cells by applying the following CSS style rule for all td elements:

```
table#hanjieGrid td {
background-color: rgb(233, 207, 29);
}
```

Creating Object Collections with CSS Selectors (continued 1)

- In JavaScript, to change the background color of all table cells, you must first define an object collection based on a CSS selector using the following `querySelectorAll()` method:

  `document.querySelectorAll(`*`selector`*`)`

  where *`selector`* is the CSS selector that the object collection is based on

Creating Object Collections with CSS Selectors (continued 2)

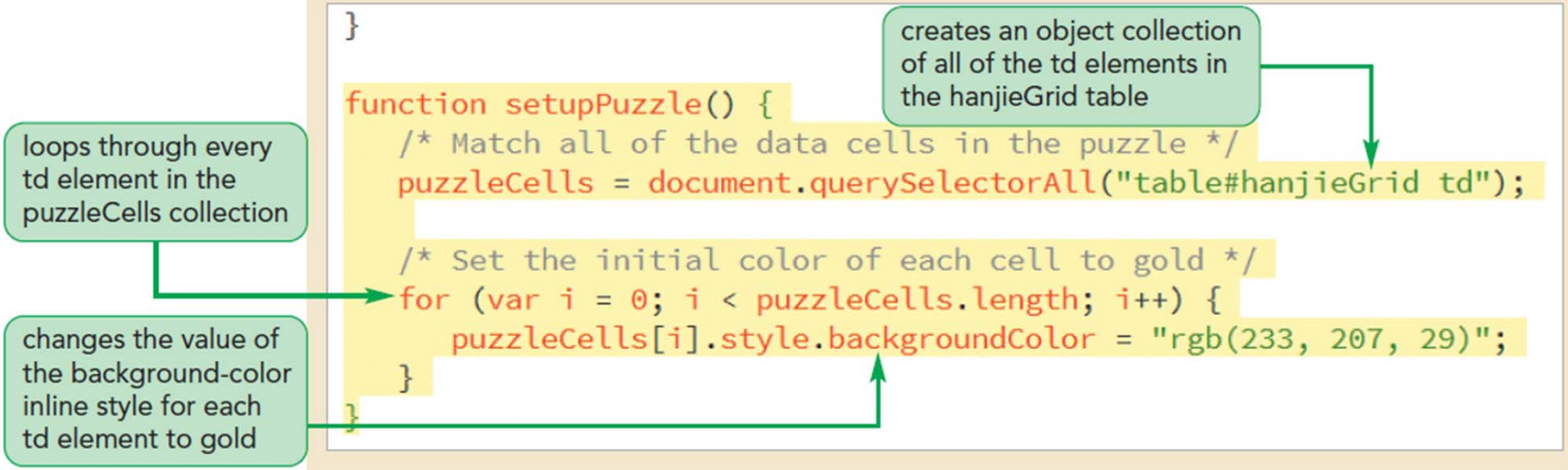
- Once the object collection has been defined, change the `background-color` style of each `td` element by applying the `backgroundColor` property to the objects in the object collection
- Reference only the first element that matches a selector pattern using the following JavaScript method:

  `document.querySelector(`*`selector`*`)`

  where *`selector`* is a CSS selector

Creating Object Collections with CSS Selectors (continued 3)

Figure 11-13 Creating the setupPuzzle() function




```
}

function setupPuzzle() {
   /* Match all of the data cells in the puzzle */
   puzzleCells = document.querySelectorAll("table#hanjieGrid td");

   /* Set the initial color of each cell to gold */
   for (var i = 0; i < puzzleCells.length; i++) {
      puzzleCells[i].style.backgroundColor = "rgb(233, 207, 29)";
   }
}
```

Creating Object Collections with CSS Selectors (continued 4)

Figure 11-14 Revising the init() function

defines puzzleCells as a global variable so it can be used in all functions

sets up the initial puzzle displayed in the web page

```
var puzzleCells;

function init() {
   // Insert the title for the first puzzle
   document.getElementById("puzzleTitle").innerHTML = "Puzzle 1";

   // Insert the HTML code for the first puzzle table
   document.getElementById("puzzle").innerHTML =
      drawPuzzle(puzzle1Hint, puzzle1Rating, puzzle1);

   //Add event handlers for the puzzle buttons
   var puzzleButtons = document.getElementsByClassName("puzzles");
   for (var i = 0; i < puzzleButtons.length; i++) {
      puzzleButtons[i].onclick = swapPuzzle;
   }

   setupPuzzle();
}
```

Working with Mouse Events

- JavaScript supports events associated with the mouse such as clicking, right-clicking, double-clicking, and moving the pointer over and out of page elements

Figure 11-17 Mouse and pointer events

Event	Description
`click`	The mouse button has been pressed and released
`contextmenu`	The right mouse button has been pressed and released
`dblclick`	The mouse button has been double-clicked
`mousedown`	The mouse button is pressed
`mouseenter`	The mouse pointer is moved onto the element
`mouseleave`	The pointer is moved off the element
`mousemove`	The pointer is moving over the element
`mouseout`	The pointer is moved off the element and any nested elements
`mouseover`	The pointer is moved onto the element and any nested elements
`mouseup`	The mouse button is released
`select`	Text is selected by the pointer
`wheel`	The mouse scroll wheel has been rotated

Working with Mouse Events (continued 1)

- A mouse action can be comprised of several events
- The action of clicking the mouse button is comprised of three events, fired in the following order:
  - `mousedown` (the button is pressed down)
  - `mouseup` (the button is released)
  - `click` (the button has been pressed and released)

Working with Mouse Events (continued 2)

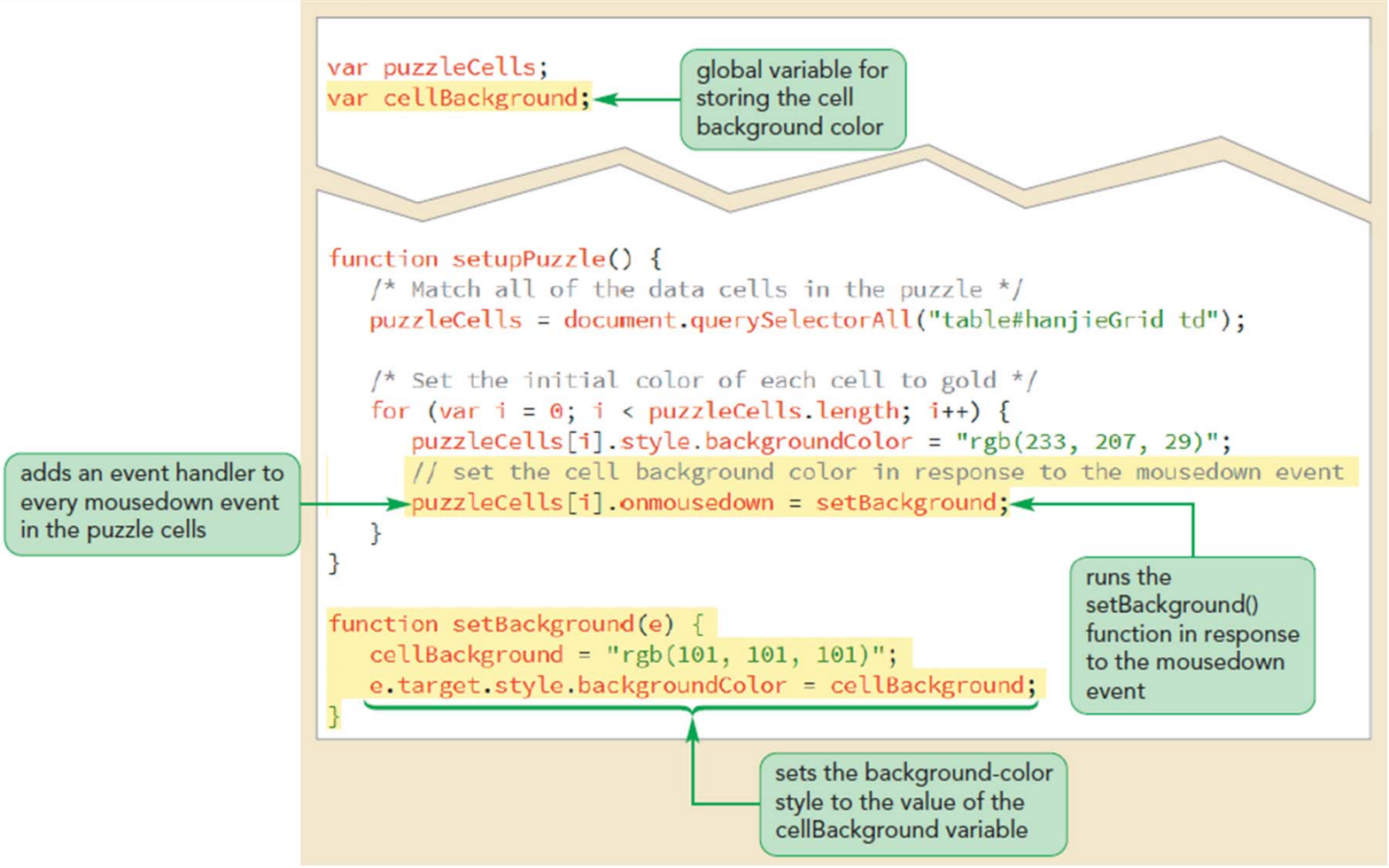
- The event object for mouse events has a set of properties that can be used to give specific information about the state of the mouse

Figure 11-18 Mouse event object properties

Event Property	Description
`evt.button`	Returns a number indicating the mouse button that was pressed, where 0 = left, 1 = wheel or middle, and 3 = right and `evt` is event object for the mouse event
`evt.buttons`	Returns a number indicating the mouse button or buttons that were pressed, where 1 = left, 2 = right, 4 = wheel or middle, 8 = back, 16 = forward, and other multiple buttons are indicated by the sum of their numbers
`evt.clientX`	Returns the horizontal coordinate (in pixels) of the mouse pointer relative to the browser window
`evt.clientY`	Returns the vertical coordinate (in pixels) of the mouse pointer relative to the browser window
`evt.detail`	Returns the number of times the mouse button was clicked
`evt.pageX`	Returns the horizontal coordinate (in pixels) of the mouse pointer relative to the document
`evt.pageY`	Returns the vertical coordinate (in pixels) of the mouse pointer relative to the document
`evt.relatedTarget`	References the secondary target of the event; for the `mouseover` event this is the element that the pointer is leaving and for the `mouseout` event this is the element that the pointer is entering
`evt.screenX`	Returns the horizontal coordinate (in pixels) of the mouse pointer relative to the physical screen
`evt.screenY`	Returns the vertical coordinate (in pixels) of the mouse pointer relative to the physical screen
`evt.which`	Returns a number indicating the mouse button that was pressed, where 0 = none, 1 = left, 2 = wheel or middle, and 3 = right

Working with Mouse Events (continued 3)

Figure 11-19 Applying the onmousedown event handler



Introducing the Event Model

- **Event model:** Describes how events and objects interact within the web page and web browser
- The process in which a single event is applied to a hierarchy of objects is part of the event model

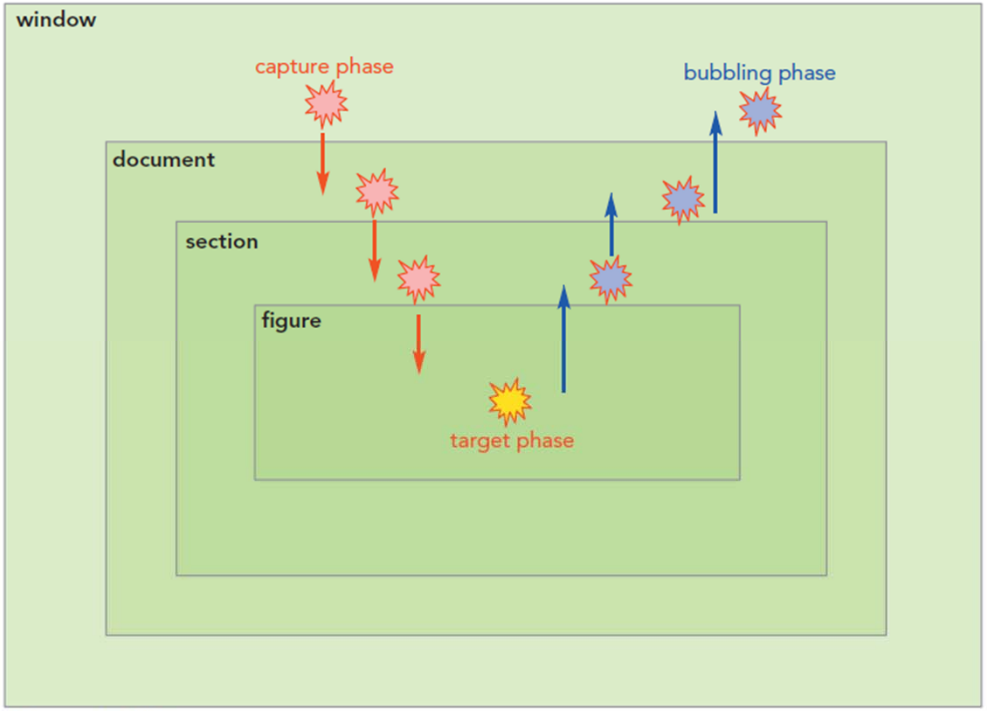
Introducing the Event Model (continued 1)

- Once an event has been initiated, it propagates through the object hierarchy in three phases
  - **Capture phase:** The event moves down the object hierarchy starting from the root element (the browser window) and moving inward until it reaches the object that initiated the event
  - **Target phase:** The event has reached the target of the event object and no longer moves down the object hierarchy

Introducing the Event Model (continued 2)

- **Bubbling phase:** The event propagates up the object hierarchy back to the root element (browser window) where the propagation stops

Figure 11-21 Event propagation in the event model



© Courtesy Patrick Carey

Introducing the Event Model (continued 3)

- Limitations of event handlers
  - Event handlers, such as `onclick` and `onmousedown`, respond to events during the target phase, but they do not recognize the propagation of events through the capture and bubbling phases
  - Only one function can be applied to an event handler at a time

Adding an Event Listener

- **Event listener**: Listens for events as they propagate through the capture, target, and bubble phases, allowing the script to respond to an event within any phase
- Unlike event handlers, more than one function can be applied to an event using event listeners

Adding an Event Listener (continued 1)

- Add an event listener to an object by applying the `addEventListener()` method

```
object.addEventListener(event,
function [, capture = false]);
```

where

- *`object`* is the object in which the event occurs
- *`event`* is the event
- *`function`* is the function that is run in response to the event

Adding an Event Listener (continued 2)

- *capture* is an optional Boolean value
  - `true` indicates that the function is executed during the capture phase
  - `false` (the default) indicates that the function is run during the bubbling phase

Figure 11-23 Adding an event listener for the mouseup event

```
//Add event handlers for the puzzle buttons
var puzzleButtons = document.getElementsByClassName("puzzles");
   for (var i = 0; i < puzzleButtons.length; i++) {
      puzzleButtons[i].onclick = swapPuzzle;
}

setupPuzzle();

// Add an event listener for the mouseup event
document.addEventListener("mouseup", endBackground);
}
```

runs the endBackground() function in response to the mouseup event

listens for the mouseup event occurring anywhere within document

Removing an Event Listener

- The event model allows to remove event listeners from the document by applying `removeEventListener()` method

```
object.removeEventListener(event,
function [, capture = false]);
```

where *object*, *event*, *function*, and *capture* have the same meanings as the `addEventListener()` method

Removing an Event Listener (continued)

Figure 11-24 Removing an event listener

```
function extendBackground(e) {
   e.target.style.backgroundColor = cellBackground;
}

function endBackground() {
   // Remove the event listener for every puzzle cell
   for (var i = 0; i < puzzleCells.length; i++) {
      puzzleCells[i].removeEventListener("mouseenter", extendBackground);
   }
}
```

removes the event listener that runs the extendBackground() function in response to the mouseenter event

Controlling Event Propagation

- The browser has its own default responses to events
- Apply the following `preventDefault()` method to the event object to prevent the occurrence of the browser's default actions:

```
evt.preventDefault()
```

Controlling Event Propagation (continued 1)

- Alternatively, you can prevent the browser's default action by returning the value `false` from the event handler function
- The `return false;` statement does not prevent default actions if event listeners are used in place of event handlers

Controlling Event Propagation (continued 2)

Figure 11-26 Preventing the default browser action

```
function setBackground(e) {
   cellBackground = "rgb(101, 101, 101)";
   e.target.style.backgroundColor = cellBackground;

   // Create an event listener for every puzzle cell
   for (var i = 0; i < puzzleCells.length; i++) {
      puzzleCells[i].addEventListener("mouseenter", extendBackground);
   }

   // Prevent the default action of selecting table text
   e.preventDefault();
}
```

prevents the default browser action of selecting text in the table

Exploring Keyboard Events

- JavaScript supports the keydown, keypress, and keyup events that allow users to interact with the web page and browser through the keyboard

Figure 11-27 Keyboard Events

Event	Description
keydown	A key is pressed down
keypress	A key is pressed down and released, resulting in a character being typed
keyup	A key is released

Exploring Keyboard Events (continued 1)

- The `keydown` and `keypress` events are similar in name; the difference between them is as follows:
  - The `keydown` and `keyup` events are fired in response to the physical act of pressing the key down and of a key moving up when it is no longer held down
  - The `keypress` event is fired in response to the computer generating a character

Exploring Keyboard Events (continued 2)

Figure 11-28 Keyboard Event Properties

Event Property	Description
evt.altKey	Returns a Boolean value indicating whether the Alt key was used in the event object, *evt*
evt.ctrlKey	Returns a Boolean value indicating whether the Ctrl key was used in the event
evt.charCode	Returns the Unicode character code of the key used in the keypress event
evt.key	Returns the text of the key used in the event (not supported by the Safari browser)
evt.keyCode	Returns the Unicode character code of the key used in the keypress event or the key code used in the keydown or keyup events
evt.location	Returns the location number of the key where 0 = key located in the standard position, 1 = a key on the keyboard's left edge, 2 = a key on the keyboard's right edge, and 3 = a key on the numeric keypad
evt.metaKey	Returns a Boolean value indicating whether the meta key (the Command key on Mac keyboards or the Windows key on PC keyboards) was used in the event
evt.shiftKey	Returns a Boolean value indicating whether the Shift key was used in the event
evt.which	Returns the Unicode character code of the key used in the keypress event or the key code used in the keydown or keyup events

Exploring Keyboard Events (continued 3)

- The value associated with a key event property is affected by the event itself
- For the `keypress` event, the `charCode`, `keyCode`, and `which` properties all return a Unicode character number
- Sample values of the `charCode`, `keyCode`, `which`, and `key` properties for different keyboard keys and events are shown in Figure 11-29

Exploring Keyboard Events (continued 4)

Figure 11-29 Keyboard Event Properties

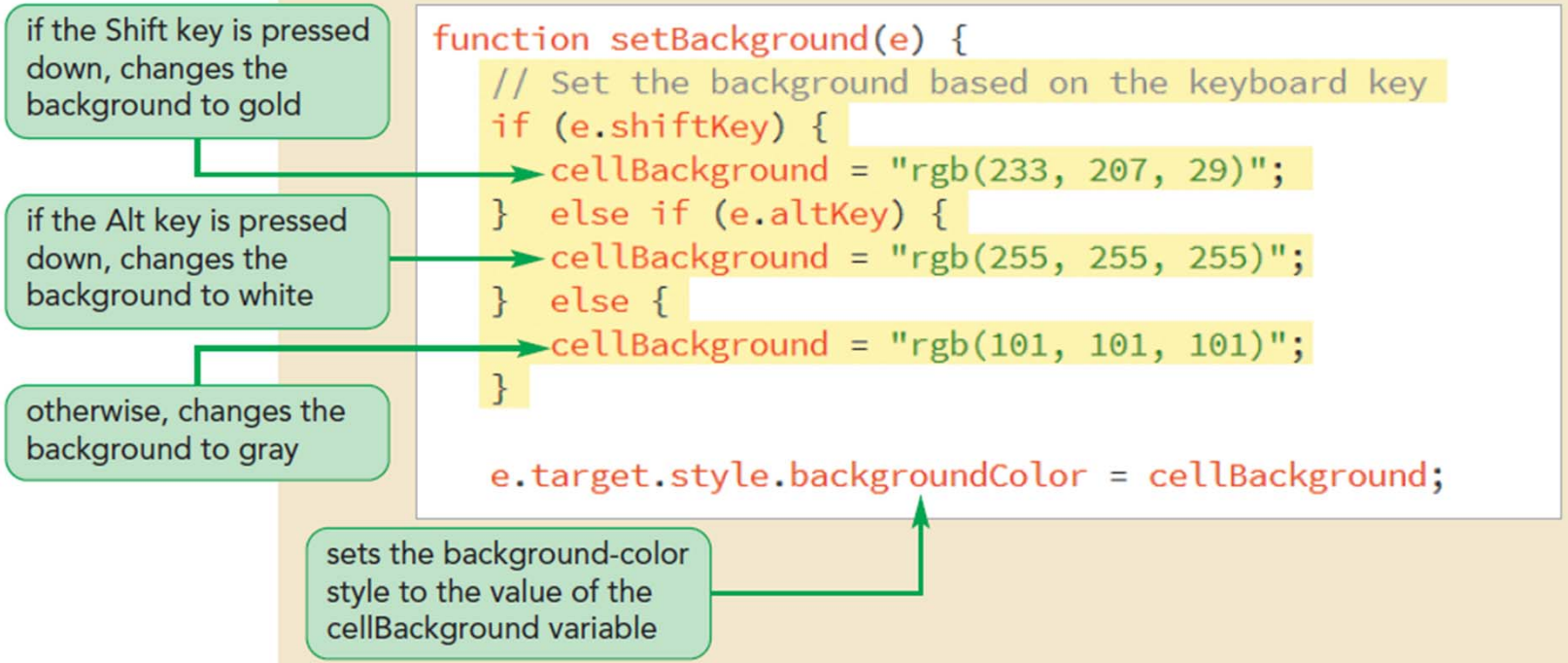
Key(s)	Values with keypress	Values with keydown and keyup
a, z	charCode = 97, 122 keyCode = 97, 122 which = 97, 122 key = "a", "z"	charCode = 0, 0 keyCode = 65, 90 which = 65, 90 key = "a", "z"
1, 9 (on numeric keypad)	charCode = 49, 57 keyCode = 49, 57 which = 49, 57 key = "1", "9"	charCode = 0, 0 keyCode = 97, 105 which = 97, 105 key = "1", "9"
1, 9 (on top keyboard row)	charCode = 49, 57 keyCode = 49, 57 which = 49, 57 key = "1", "9"	charCode = 0, 0 keyCode = 49, 57 which = 49, 57 key = "1", "9"
Shift, Ctrl, Alt, Meta (Mac Command key; PC Window key) Command	*no event detected*	charCode = 0, 0, 0, 0 keyCode = 16, 17, 18, 91 which = 16, 17, 18, 91 key = "Shift", "Control", "Alt", "Meta"
←, ↑, →, ↓	*no event detected*	charCode = 0, 0, 0, 0 keyCode = 37, 38, 39, 40 which = 37, 38, 39, 40 key = "ArrowLeft", "ArrowUp", "ArrowRight", "ArrowDown"

Exploring Keyboard Events (continued 5)

- **Modifier keys:** Alt, Ctrl, Shift, and Command keys
- In addition to character keys, JavaScript supports the modifier keys through the use of the `altKey`, `ctrlKey`, `shiftKey`, and `metaKey` properties

Exploring Keyboard Events (continued 6)

Figure 11-30 Changing the background color for different modifier keys




```
function setBackground(e) {
   // Set the background based on the keyboard key
   if (e.shiftKey) {
      cellBackground = "rgb(233, 207, 29)";
   } else if (e.altKey) {
      cellBackground = "rgb(255, 255, 255)";
   } else {
      cellBackground = "rgb(101, 101, 101)";
   }

   e.target.style.backgroundColor = cellBackground;
```

Changing the Cursor Style

- Cursors can be defined using the following CSS cursor style:

```
cursor: cursorTypes;
```

where *cursorTypes* is a comma-separated list of cursor types

Changing the Cursor Style (continued 1)

- JavaScript command to define cursors is as follows:

```
object.style.cursor = cursorTypes;
```

where *object* is the page object that will display the cursor style when hovered over by the mouse pointer

Changing the Cursor Style (continued 2)

- Create a customized cursor from an image file using `url(`*`image`*`)` where *`image`* is an image file
- Example: `cursor: url(jpf_pencil.png), pointer;`

Figure 11-33 Setting cursor style for the puzzle cells

```
function setupPuzzle() {
   /* Match all of the data cells in the puzzle */
   puzzleCells = document.querySelectorAll("table#hanjieGrid td");

   /* Set the initial color of each cell to gold */
   for (var i = 0; i < puzzleCells.length; i++) {
      puzzleCells[i].style.backgroundColor = "rgb(233, 207, 29)";
      // Set the cell background color in response to the mousedown event
      puzzleCells[i].onmousedown = setBackground;
      // Use a pencil image as the cursor
      puzzleCells[i].style.cursor = "url(jpf_pencil.png), pointer";
   }
}
```

uses the pencil image as the cursor for puzzle cells

if the image file is not supported, uses the generic pointer cursor

Changing the Cursor Style (continued 3)

- By default, the click point for a cursor is located in the top-left corner of the cursor image at the coordinates (0, 0)
- Specify a different location by adding the (x, y) coordinates of the click point to the cursor definition as follows:

  `url(`*`image`*`)` *`x y`*

  where *x* is the x-coordinate and *y* is the y-coordinate of the click point in pixels

Working with Functions as Objects

- Everything in JavaScript is an object, including functions
- Anything that can be done with an object can be done with a function, including
  - Storing a function as variable
  - Storing a function as an object property

Working with Functions as Objects (continued)

- Using one function as a parameter in another function
- Nesting one function within another function
- Returning a function as the result of another function
- Modifying the properties of a function

Function Declarations and Function Operators

- The following hello() function is created using the **function declaration** format:

```
function hello(){
   alert("Welcome to Hanjie!");}
```

- **Function operator:** The definition of the function becomes the variable's "value"

```
var hello = function (){
   alert("Welcome to Hanjie!");
}
```

Function Declarations and Function Operators (continued 1)

- The two ways of defining the hello() function differ in how they are stored
  - Functions defined with a function declaration are created and saved as the browser parses the code prior to the script being run
    - Since the function is already stored in memory, the statements that run the function can be placed prior to the statement that declares the function

Function Declarations and Function Operators (continued 2)

- Function operators are evaluated as they appear in the script after the code has been parsed by the browser
  - Function operators are more flexible than function declarations, allowing a function to be placed anywhere a variable can be placed

Anonymous Functions

- Anonymous function has function declaration without the function name
- Example: The following structure is an anonymous function:

```
function() {
commands
}
```

- An anonymous function can be inserted into any expression where a function reference is required

Anonymous Functions (continued 1)

- Functions that are named are called **named functions**
- Anonymous functions are more concise and easier to manage because the function is directly included with the expression that invokes it
- Anonymous functions limit the scope of the function to exactly where it is needed

Anonymous Functions (continued 2)

Figure 11-37 Using an anonymous function

```
   setupPuzzle();

   // Add an event handler for the mouseup event
   document.addEventListener("mouseup", endBackground);

   // Add an event listener to the Show Solution button
   document.getElementById("solve").addEventListener("click",
      function() {
         // Remove the inline backgroundColor style from each cell
         for (var i = 0; i < puzzleCells.length; i++) {
            puzzleCells[i].style.backgroundColor = "";
         }
      }
   );
}
```

listens for the Show Solution button click event

inserts the response to the click event as an anonymous function

erases the inline background-color style from each puzzle cell by setting the style value to an empty text string

Passing Variable Values into Anonymous Functions

- JavaScript supports two types of variables:
  - Global variables: Declared outside of any function and thus are accessible throughout the app
  - Local variables: Declared within a function and are only accessible to code within that function

Passing Variable Values into Anonymous Functions (continued 1)

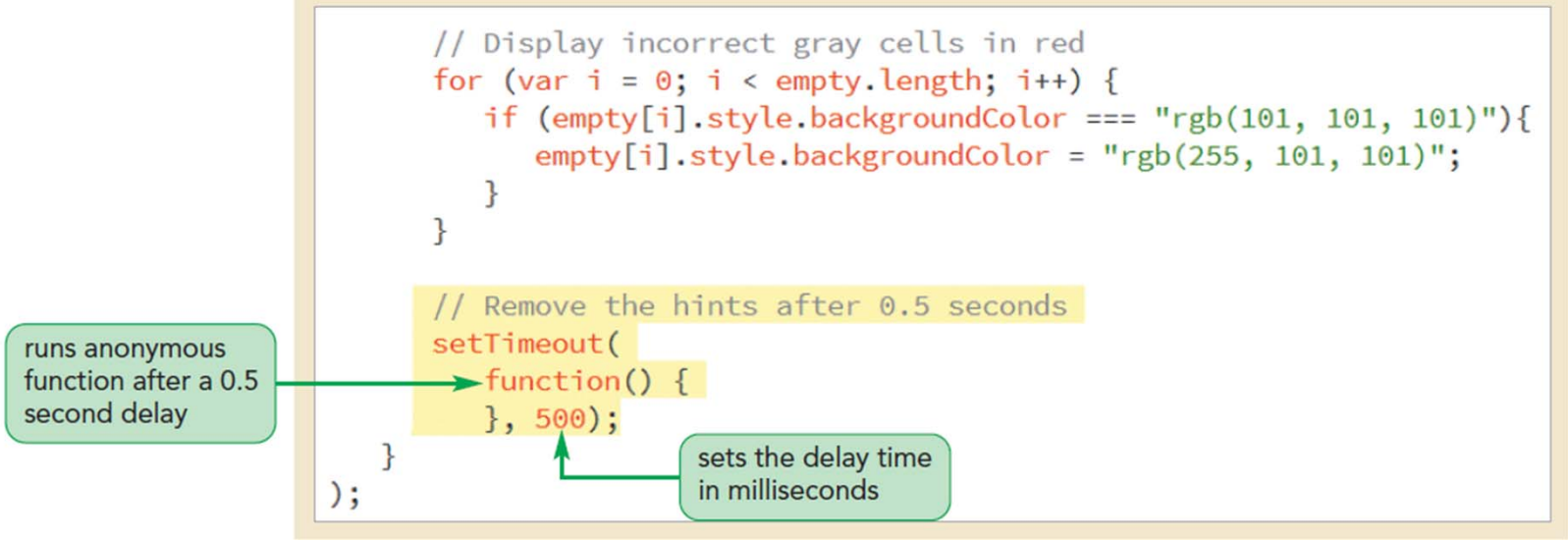
- Global variables should be avoided when possible because
  - global variables are accessible to every function in the application
  - the task of tracking which functions are using and modifying the global variables becomes increasingly difficult as the application grows in size and complexity

Passing Variable Values into Anonymous Functions (continued 2)

- An advantage of using anonymous functions is that they reduce the need for global variables because they perform their actions locally within a function
- One of the challenges of anonymous functions is keeping track of all of the nested levels of functions and procedures

Passing Variable Values into Anonymous Functions (continued 3)

Figure 11-43 Using an anonymous function with the setTimeout() method



Displaying Dialog Boxes

- Alert dialog box can be created using the following `alert()` method:

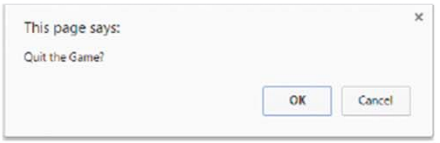
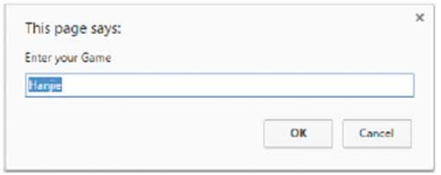
  `alert(`*`text`*`)`

  where *`text`* is the message displayed in the alert dialog box
- When an alert dialog box is displayed, the execution of the program code halts until the user clicks the OK button in the dialog box

Displaying Dialog Boxes (continued 1)

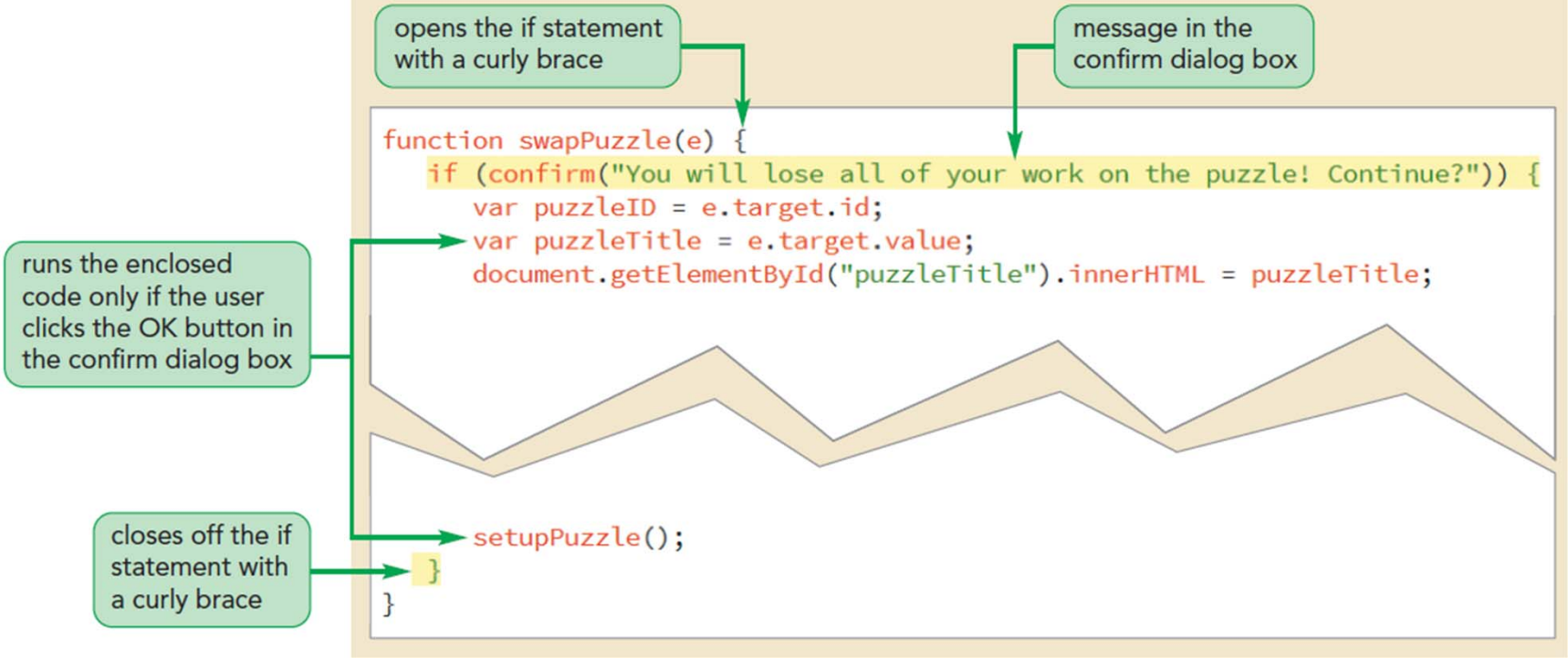
- JavaScript supports confirmation and prompt dialog boxes

Figure 11-45 JavaScript dialog boxes

Dialog Box	Description	Example
`alert(text)`	Displays an alert message	`alert("Welcome to Hanjie!");` 
`confirm(text)`	Displays a confirmation box, returning a value of `true` if the user clicks the OK button and `false` if the Cell button is clicked	`var quitGame = confirm("Quit the Game?");` 
`prompt(text [, defaultInput])`	Displays a prompt box, prompting the user to enter input text, where *`defaultInput`* is an optional attribute specifying the default input value; if the user clicks OK the input text is returned, if the Cancel button is clicked a value of `null` is returned	`var gameType = prompt("Enter your Game", "Hanjie");` 

Displaying Dialog Boxes (continued 2)

Figure 11-46 Applying a confirm dialog box



Displaying Dialog Boxes (continued 3)

Figure 11-50 A successfully completed puzzle