HTML5, CSS3, and JavaScript 6th Edition

Tutorial 14

Exploring Object-Based Programming

Carey

Objectives

- Use nested functions
- Create an object literal
- Define object properties and methods
- Define an object class
- Use object constructor functions
- Instantiating an object

Objectives (continued)

- Define an object prototype
- Explore prototype chains
- Use the `apply()` and `call()` methods
- Work with objects and arrays
- Create a `for…in` loop

Working with Nested Functions

- Any function, including named functions, can be nested within another function as follows:

```
function outsideFn() {
   commands
   function insideFn() {
         commands       }
   commands
}
```

where

– *outsideFn()* is the outer or containing function

Working with Nested Functions (continued 1)

  – *`insideFn()`* is the inner or nested function

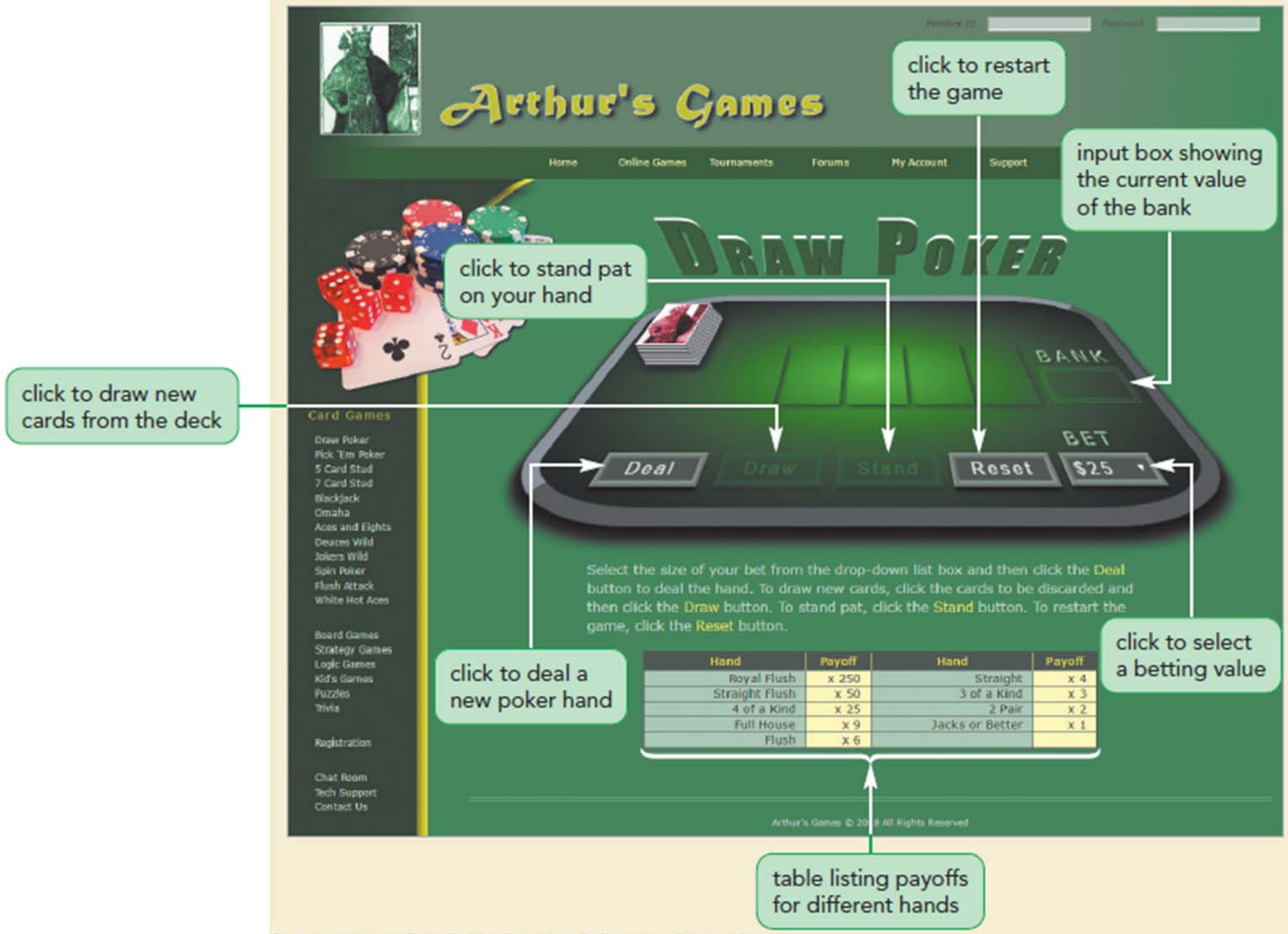
- Scope of a nested function is limited to the commands within the containing function
- Nested function is hidden from other code in the script, making the code contained and easier to manage

Working with Nested Functions (continued 2)

- Example: Nested functions can be used in developing the poker game app for Arthur’s Games website
- The app displays the poker table on which users will play a game of draw poker

Working with Nested Functions (continued 3)

Figure 14-1 Draw Poker web page



Select the size of your bet from the drop-down list box and then click the Deal button to deal the hand. To draw new cards, click the cards to be discarded and then click the Draw button. To stand pat, click the Stand button. To restart the game, click the Reset button.

Hand	Payoff	Hand	Payoff
Royal Flush	x 250	Straight	x 4
Straight Flush	x 50	3 of a Kind	x 3
4 of a Kind	x 25	2 Pair	x 2
Full House	x 9	Jacks or Better	x 1
Flush	x 6		

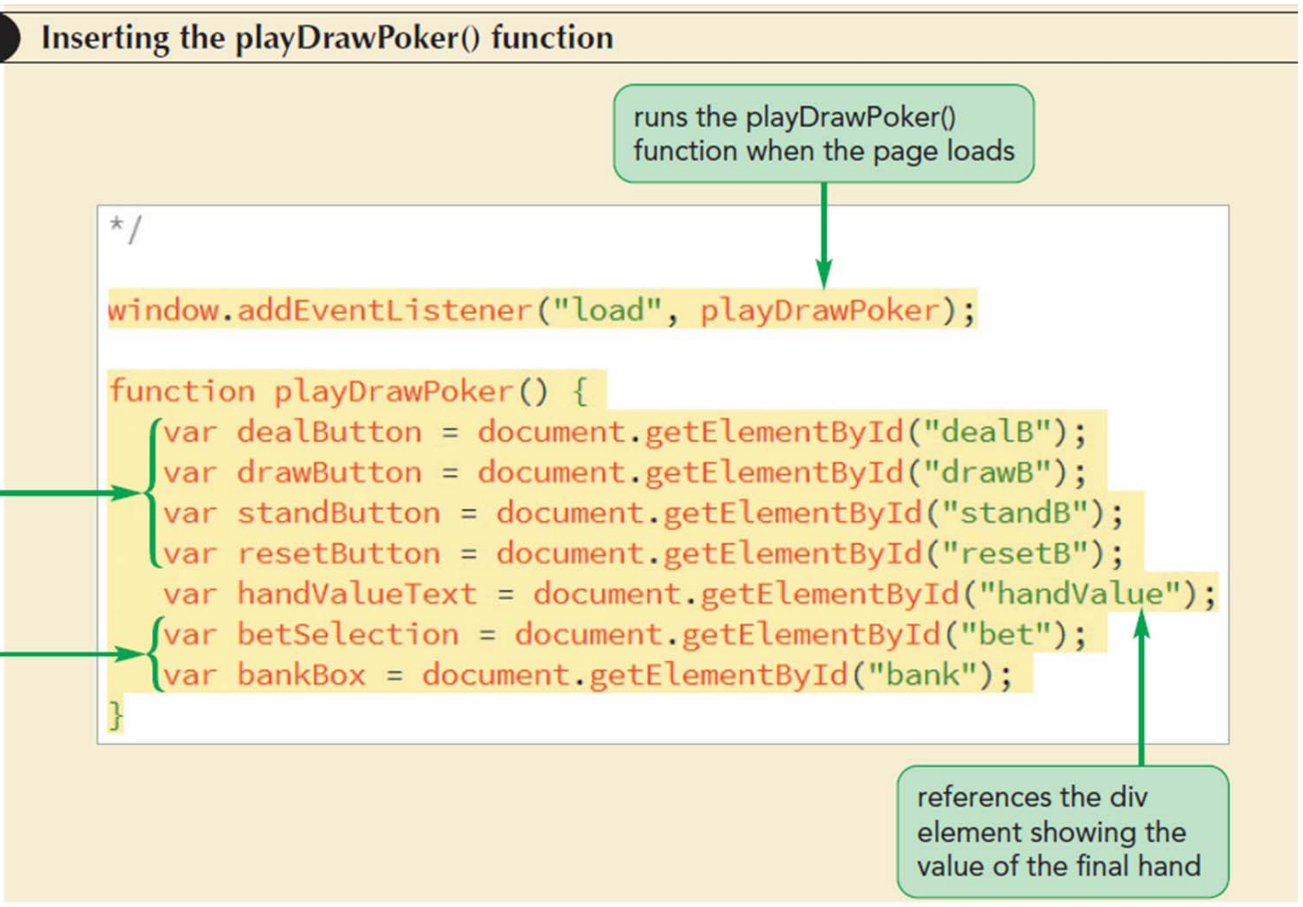
Source: Howard Pyle/Public Domain; © Courtesy Patrick Carey

Working with Nested Functions (continued 4)

- All operations for the game will be stored within the playDrawPoker() function
- playDrawPoker() function runs automatically when the page loads
- playDrawPoker() function is created by adding references to all the buttons on the poker table

Working with Nested Functions (continued 5)

Figure 14-2 Inserting the playDrawPoker() function


runs the playDrawPoker() function when the page loads
references the buttons on the poker table
references the bet and bank boxes on the poker table
references the div element showing the value of the final hand

```
*/

window.addEventListener("load", playDrawPoker);

function playDrawPoker() {
   var dealButton = document.getElementById("dealB");
   var drawButton = document.getElementById("drawB");
   var standButton = document.getElementById("standB");
   var resetButton = document.getElementById("resetB");
   var handValueText = document.getElementById("handValue");
   var betSelection = document.getElementById("bet");
   var bankBox = document.getElementById("bank");
}
```

Working with Nested Functions (continued 6)

- Elements on the Draw Poker page perform the following tasks:
  - Deal button deals five cards from the poker deck into the player’s hand
  - Draw button replaces all selected cards in the player’s hand with new cards from the deck

Working with Nested Functions (continued 7)

- Stand button signals to the dealer that the player wants to keep all the cards in the dealt hand
- Reset button restarts the game with a fresh pot, resetting the bank value to $500
- Bet selection list places the bet before the next hand is dealt

Working with Nested Functions (continued 8)

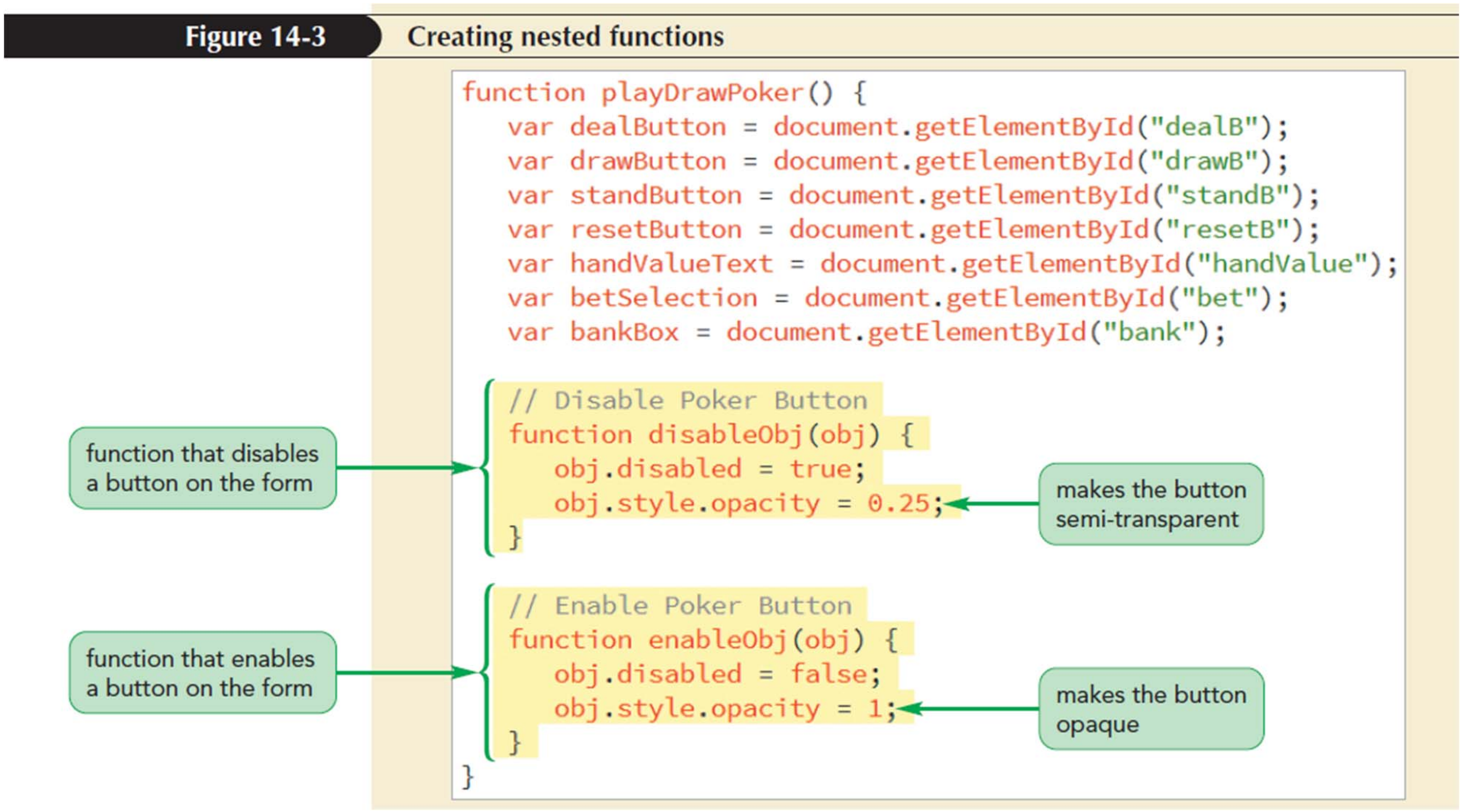
- The Deal, Draw, and Stand buttons and the Bet selection list will be turned on and off depending on the state of the game
- The Draw and Stand buttons are disabled before the deal
- The Deal button and Bet selection list are disabled while the current hand is in play

Working with Nested Functions (continued 9)

- To disable and enable the buttons, nest the required functions within the `playDrawPoker()` function
- The functions for disabling or enabling the selected object also set the opacity style of the selected object
- Disabled objects are semi-transparent on the page

Working with Nested Functions (continued 10)

Figure 14-3 Creating nested functions



```
function playDrawPoker() {
   var dealButton = document.getElementById("dealB");
   var drawButton = document.getElementById("drawB");
   var standButton = document.getElementById("standB");
   var resetButton = document.getElementById("resetB");
   var handValueText = document.getElementById("handValue");
   var betSelection = document.getElementById("bet");
   var bankBox = document.getElementById("bank");

   // Disable Poker Button
   function disableObj(obj) {
      obj.disabled = true;
      obj.style.opacity = 0.25;
   }

   // Enable Poker Button
   function enableObj(obj) {
      obj.disabled = false;
      obj.style.opacity = 1;
   }
}
```

function that disables a button on the form
makes the button semi-transparent
function that enables a button on the form
makes the button opaque

Working with Nested Functions (continued 11)

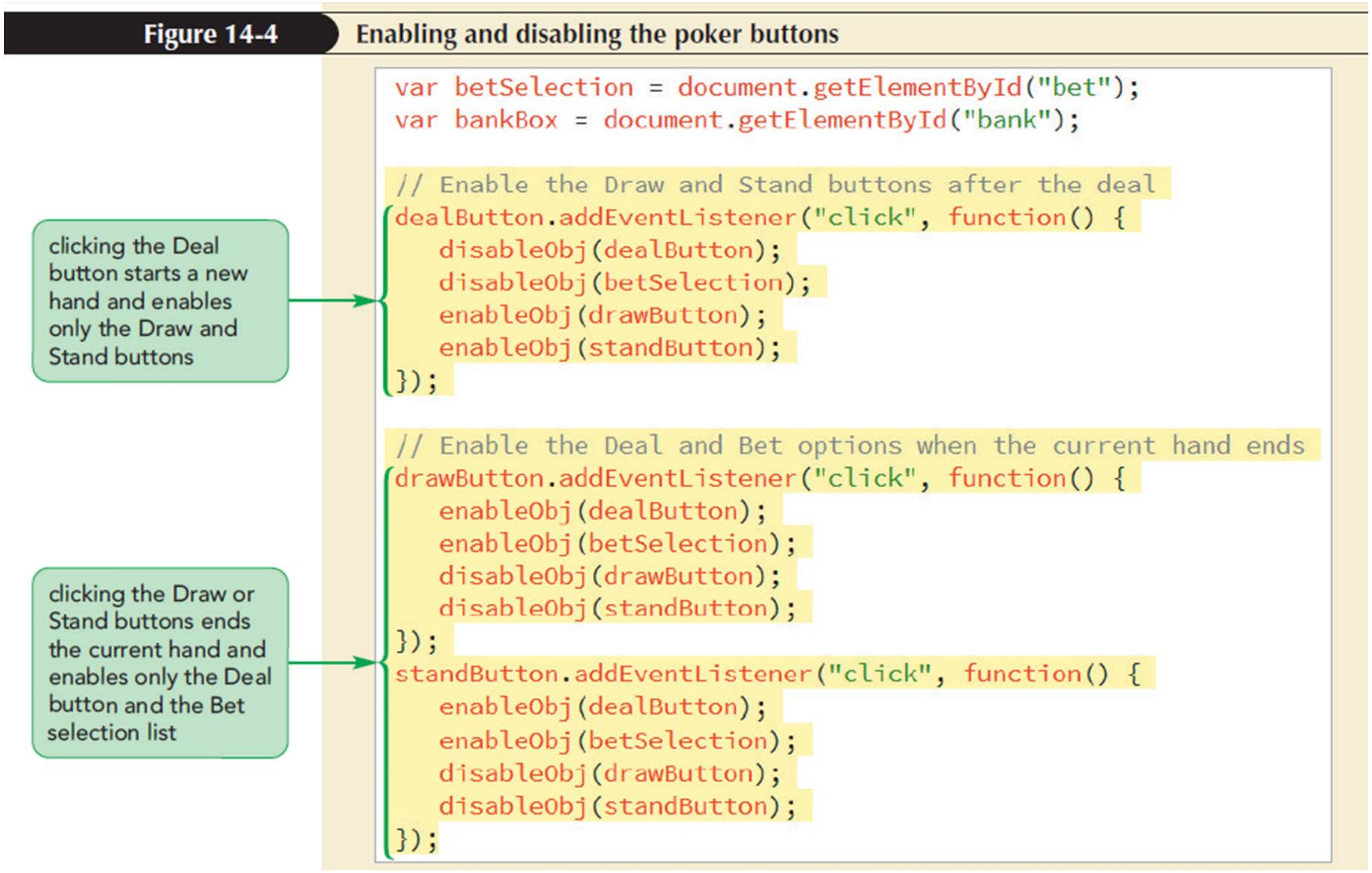


Figure 14-4 Enabling and disabling the poker buttons

```
var betSelection = document.getElementById("bet");
var bankBox = document.getElementById("bank");

// Enable the Draw and Stand buttons after the deal
dealButton.addEventListener("click", function() {
   disableObj(dealButton);
   disableObj(betSelection);
   enableObj(drawButton);
   enableObj(standButton);
});

// Enable the Deal and Bet options when the current hand ends
drawButton.addEventListener("click", function() {
   enableObj(dealButton);
   enableObj(betSelection);
   disableObj(drawButton);
   disableObj(standButton);
});
standButton.addEventListener("click", function() {
   enableObj(dealButton);
   enableObj(betSelection);
   disableObj(drawButton);
   disableObj(standButton);
});
```

clicking the Deal button starts a new hand and enables only the Draw and Stand buttons
clicking the Draw or Stand buttons ends the current hand and enables only the Deal button and the Bet selection list

Introducing Custom Objects

- There are three kinds of JavaScript objects
  - **Native objects,** such as `Date` or `Array` objects, are part of the JavaScript language
  - **Host objects** are objects provided by the browser for use in interacting with the web document and browser, such as the `window`, `document`, or `form` objects
  - **Custom objects**, also known as **user-defined objects**, are objects created by the user for specific programming tasks

Introducing Custom Objects (continued 1)

- The following custom objects are created for the poker game application:
  - A poker game object that contains information about the card game being played
  - A poker deck object that contains information about the cards used in the game
  - A poker hand object that contains information about the hand played by the user in the game
  - Poker card objects that contain information about the individual cards in the poker hand

Introducing Custom Objects (continued 2)

- A custom object can be defined in one of the following three ways:
  - Creating it as an object literal
  - Using an object constructor
  - Applying the object `create()` method

Object Literals

- The general syntax to create a custom object as an object literal

```
var objName = {
name1: value1,
name2: value2,
…
};
```

where

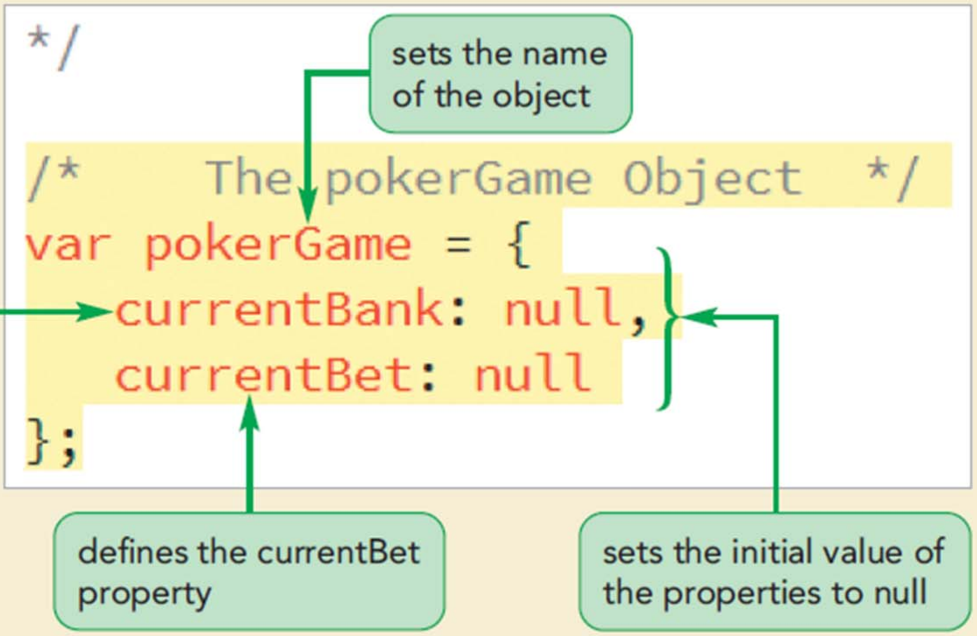
  - *objName* is the name of the object

Object Literals (continued 1)

  - *`name1`*, *`name2`*, and so on are the names associated with that object
  - *`value1`*, *`value2`*, and so on are the values assigned to those names
- Each *`name:value`* pair contains a property and property value associated with the object

Object Literals (continued 2)

Figure 14-5 Creating the pokerGame object literal



```
*/

/*    The pokerGame Object  */
var pokerGame = {
   currentBank: null,
   currentBet: null
};
```

sets the name of the object
defines the currentBank property
defines the currentBet property
sets the initial value of the properties to null

Dot Operators and Bracket Notation

- **Dot operator**: Connects the object name with an object property

    `object.property`

- Object properties can also be written using the **bracket notation**

    `object[“property”]` (with *object* and *property* in italics)

  where *`object`* is the object name and *`property`* is the object property

Dot Operators and Bracket Notation (continued 1)

- Example: The value of the currentBank property of the pokerGame object could be set with either

  ```
  pokerGame.currentBank = 500;
  ```

  or

  ```
  pokerGame[“currentBank”] = 500;
  ```

Dot Operators and Bracket Notation (continued 2)

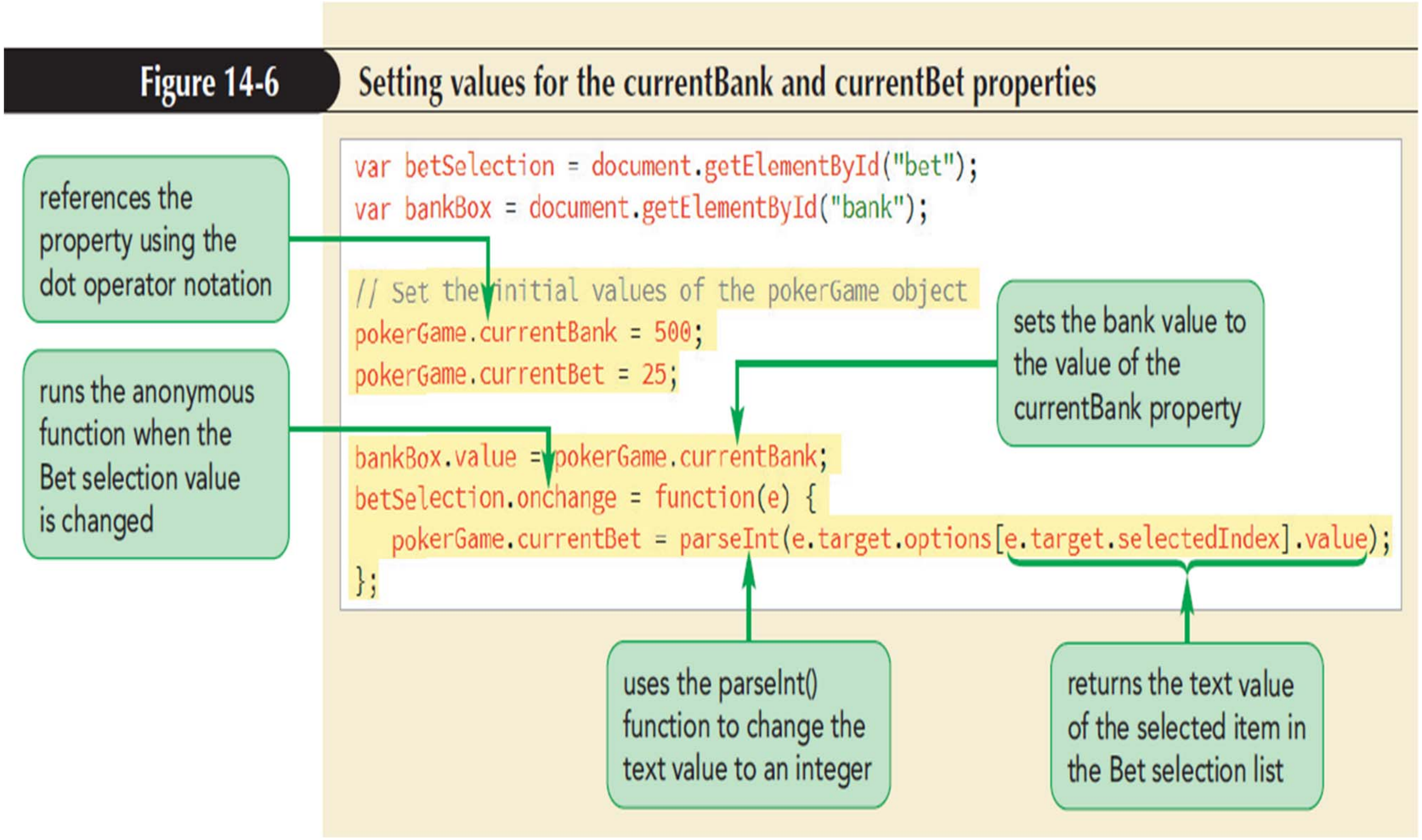


Figure 14-6 Setting values for the currentBank and currentBet properties

```
var betSelection = document.getElementById("bet");
var bankBox = document.getElementById("bank");

// Set the initial values of the pokerGame object
pokerGame.currentBank = 500;
pokerGame.currentBet = 25;

bankBox.value = pokerGame.currentBank;
betSelection.onchange = function(e) {
   pokerGame.currentBet = parseInt(e.target.options[e.target.selectedIndex].value);
};
```

references the property using the dot operator notation
runs the anonymous function when the Bet selection value is changed
sets the bank value to the value of the currentBank property
uses the parseInt() function to change the text value to an integer
returns the text value of the selected item in the Bet selection list

Creating a Custom Method

- Methods are added to a custom object by including a function name and its commands as the following *name*:*value* pair:

```
var objName = {
   method: function() {
         commands
   }
}
```

where *method* is the name of the method and *commands* are commands run by the method

Creating a Custom Method (continued 1)

- Example: The following code adds the placeBet() method to the pokerDeck object:

```
var pokerDeck = {
   currentBank: null,
   currentBet: null,
   placeBet: function() {
   this.currentBank -= this.currentBet;
   return currentBank;
   }
}
```

Creating a Custom Method (continued 2)

- The placeBet() method uses the `this` keyword to reference the current object
- The `-=` assignment operator is used to subtract the value of the current bet from the current bank value
- The method concludes by returning the value of the currentBank property

Creating a Custom Method (continued 3)

- To apply the placeBet() method to the pokerDeck object, run the following expression:

```
    pokerDeck.placeBet()
```

- Add placeBet() method to the pokerGame object to reduce the bank value by the size of the bet
- The placeBet() method should be run whenever the user clicks the Deal button

Creating a Custom Method (continued 4)

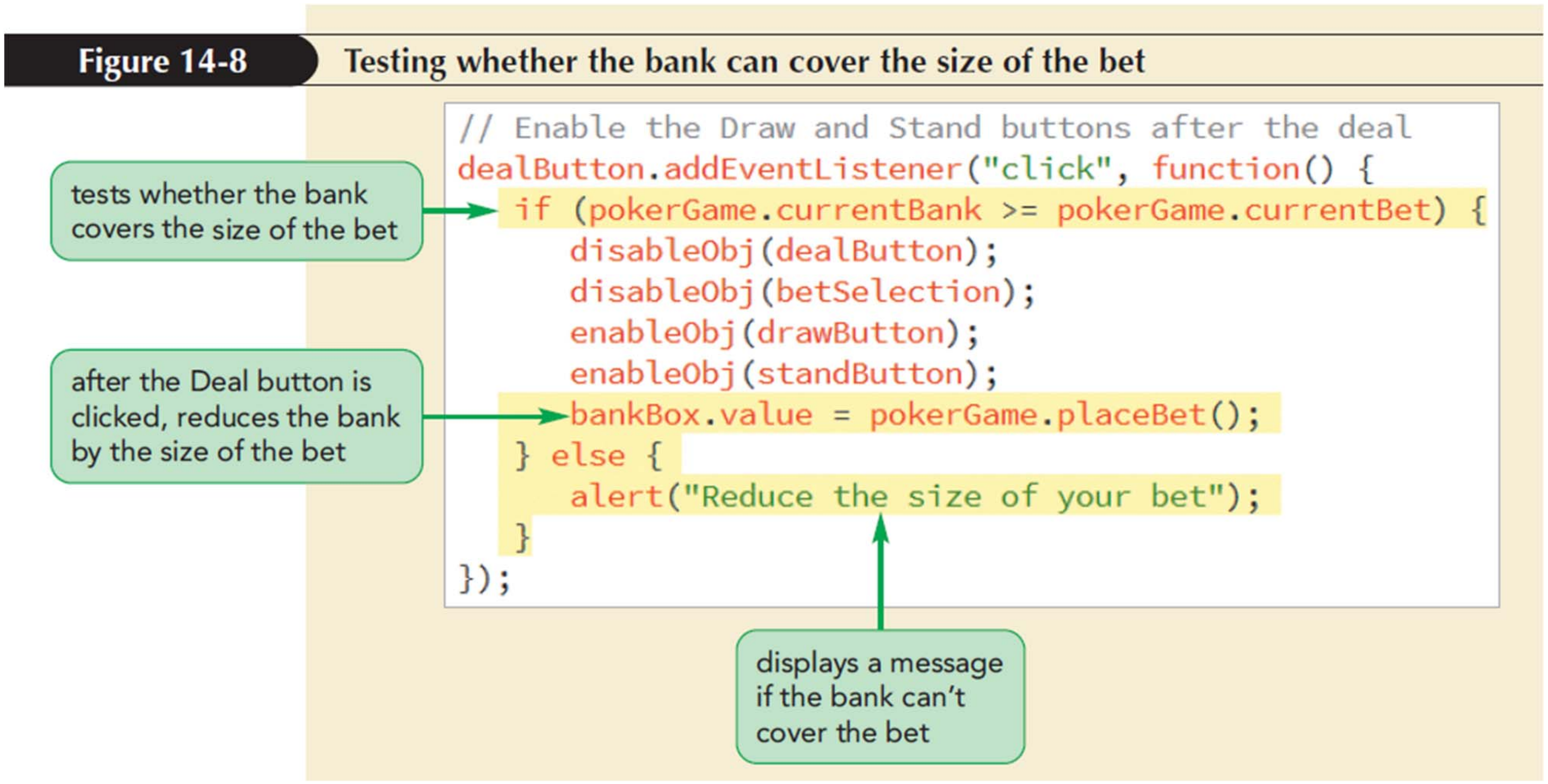


Figure 14-8 Testing whether the bank can cover the size of the bet

```
// Enable the Draw and Stand buttons after the deal
dealButton.addEventListener("click", function() {
   if (pokerGame.currentBank >= pokerGame.currentBet) {
      disableObj(dealButton);
      disableObj(betSelection);
      enableObj(drawButton);
      enableObj(standButton);
      bankBox.value = pokerGame.placeBet();
   } else {
      alert("Reduce the size of your bet");
   }
});
```

tests whether the bank covers the size of the bet
after the Deal button is clicked, reduces the bank by the size of the bet
displays a message if the bank can't cover the bet

Creating an Object with the *new* Operator

- Syntax to create a custom object with the *`new`* operator

```
var objName = new Object();
object.property = value;
object.method = function() {
    commands
};
```

where *`objName`* is the object name, *`property`* is a property defined for that object, and *`method`* is a method assigned to that object

Creating an Object with the *new* Operator (continued)

- The `new Object()` statement is equivalent to an empty object literal `{}` that creates an object devoid of properties and methods
- The biggest limitation of an object created either as an object literal or with the `new Object()` command is that the object is not reusable

Constructor Functions

- Object class can be created using an object constructor or a constructor that defines the properties and methods associated with the object type
- **Object instance** or **instance**: Specific object that is based on an object class
- Creating the object based on an object class is known as **instantiating** an object

Constructor Functions (continued 1)

- Object constructors are defined with the following **constructor function**:

```
function objClass(parameters) {
   this.prop1 = value1;
   this.prop2 = value2;…
   this.method1 = function1;
   this.method2 = function2;…
}
```

where

  – *`objClass`* is the name of the object class

Constructor Functions (continued 2)

- `parameters` are the parameters used by the constructor function
- `prop1`, `prop2`, and so on are the properties associated with that object class
- `method1`, `method2`, and so on are the methods associated with that object class
- `this` keyword refers to any object instance of the particular object class

Constructor Functions (continued 3)

- Instances of an object are created with the following command:

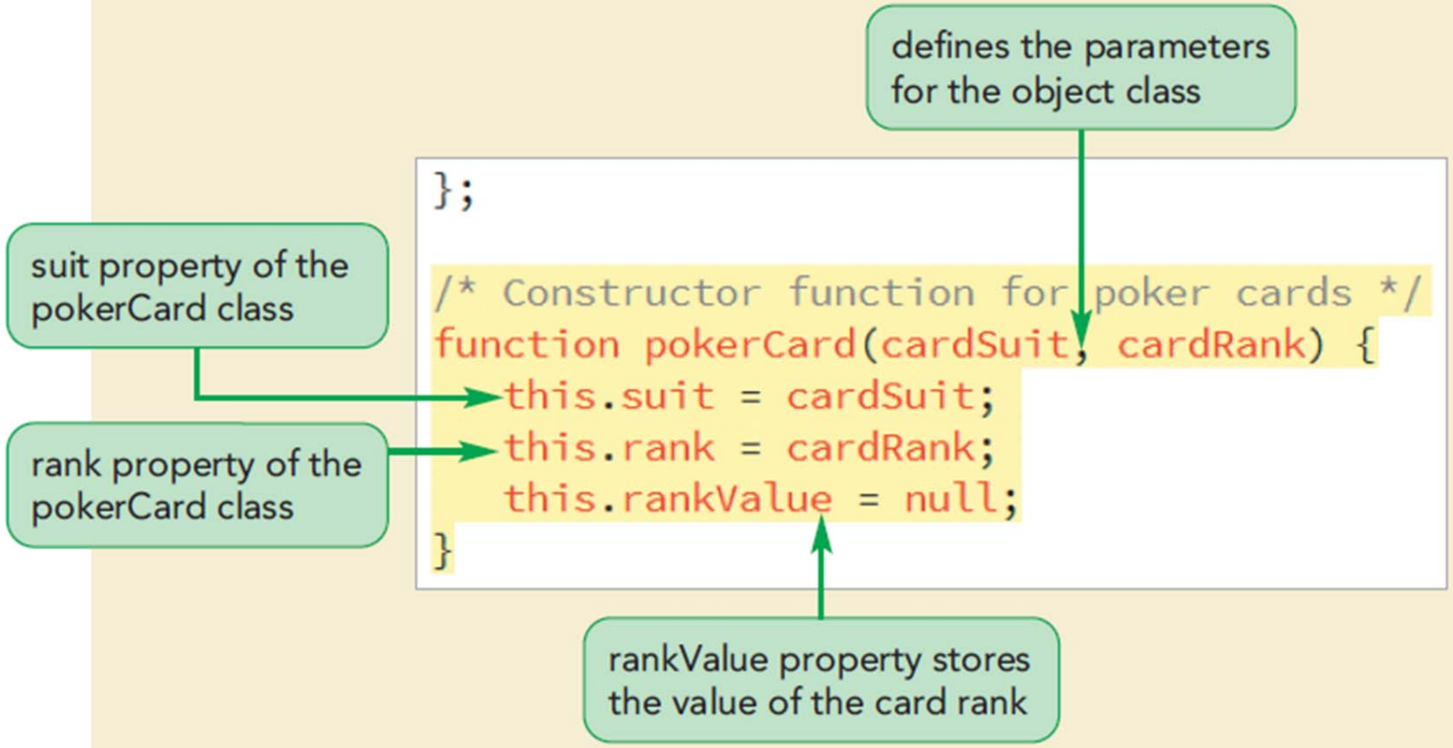
```
var objInstance = new
objClass(parameters);
```

where

  - *`objInstance`* is a specific instance of the object
  - *`objClass`* is the object class as defined by the constructor function
  - *`parameters`* are the values of any parameters included in the constructor function

Constructor Functions (continued 4)

Figure 14-11 Constructor function for pokerCard object class



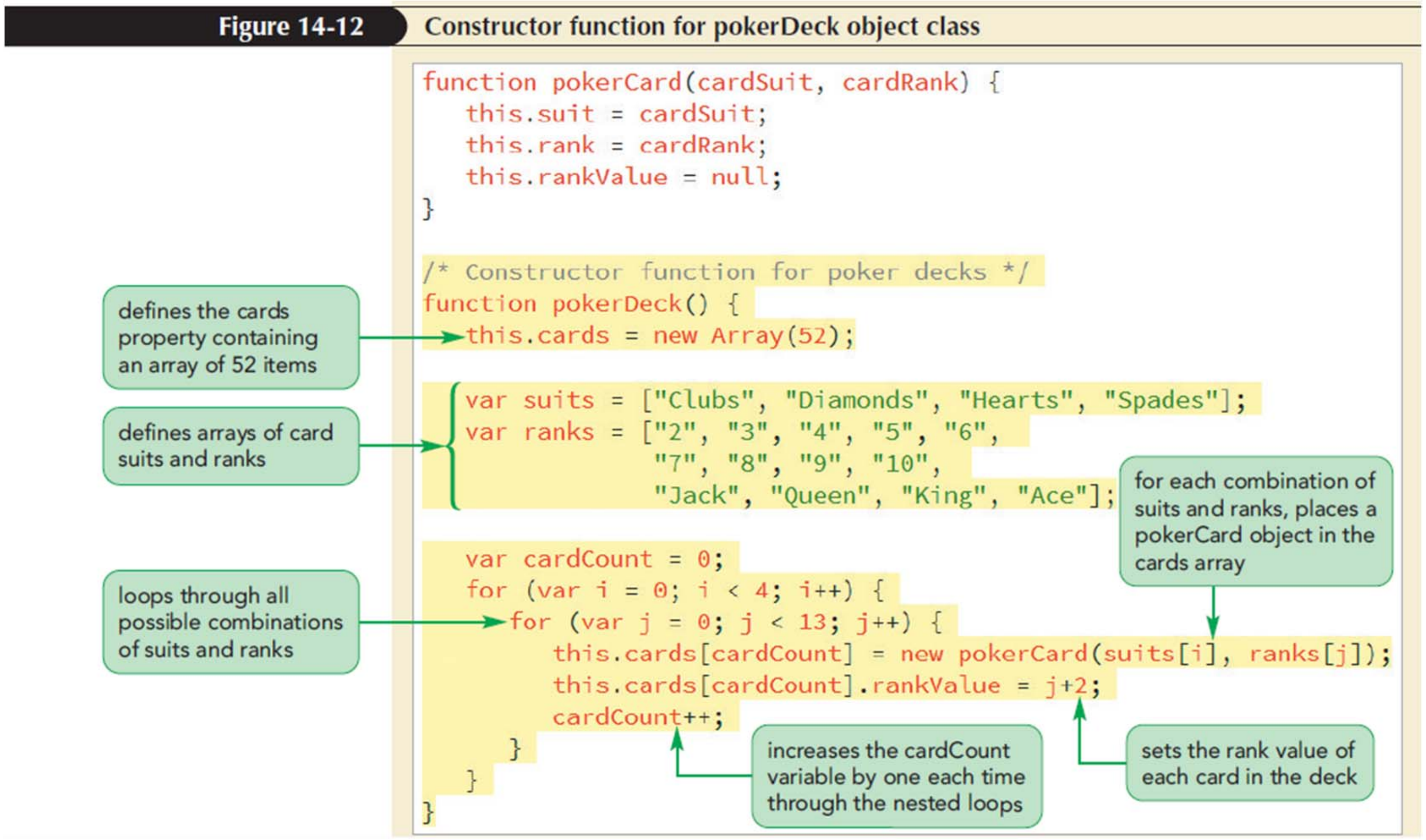
```
};

/* Constructor function for poker cards */
function pokerCard(cardSuit, cardRank) {
   this.suit = cardSuit;
   this.rank = cardRank;
   this.rankValue = null;
}
```

defines the parameters for the object class
suit property of the pokerCard class
rank property of the pokerCard class
rankValue property stores the value of the card rank

Combining Object Classes

- One object class can include objects defined in other classes

Figure 14-12 Constructor function for pokerDeck object class



```
function pokerCard(cardSuit, cardRank) {
   this.suit = cardSuit;
   this.rank = cardRank;
   this.rankValue = null;
}

/* Constructor function for poker decks */
function pokerDeck() {
   this.cards = new Array(52);

   var suits = ["Clubs", "Diamonds", "Hearts", "Spades"];
   var ranks = ["2", "3", "4", "5", "6",
                "7", "8", "9", "10",
                "Jack", "Queen", "King", "Ace"];

   var cardCount = 0;
   for (var i = 0; i < 4; i++) {
      for (var j = 0; j < 13; j++) {
         this.cards[cardCount] = new pokerCard(suits[i], ranks[j]);
         this.cards[cardCount].rankValue = j+2;
         cardCount++;
      }
   }
}
```

defines the cards property containing an array of 52 items
defines arrays of card suits and ranks
loops through all possible combinations of suits and ranks
for each combination of suits and ranks, places a pokerCard object in the cards array
increases the cardCount variable by one each time through the nested loops
sets the rank value of each card in the deck

Combining Object Classes (continued 1)

- To instantiate an object from the pokerDeck class, create a variable named “myDeck” using the following command:

```
var myDeck = new pokerDeck();
```

- The array of pokerCard objects for the myDeck variable is referenced with the following expression:

```
myDeck.cards
```

Combining Object Classes (continued 2)

- Each card in the deck can be retrieved by referencing an index in the cards array
- Example: The following expression retrieves the fourth card from the deck:

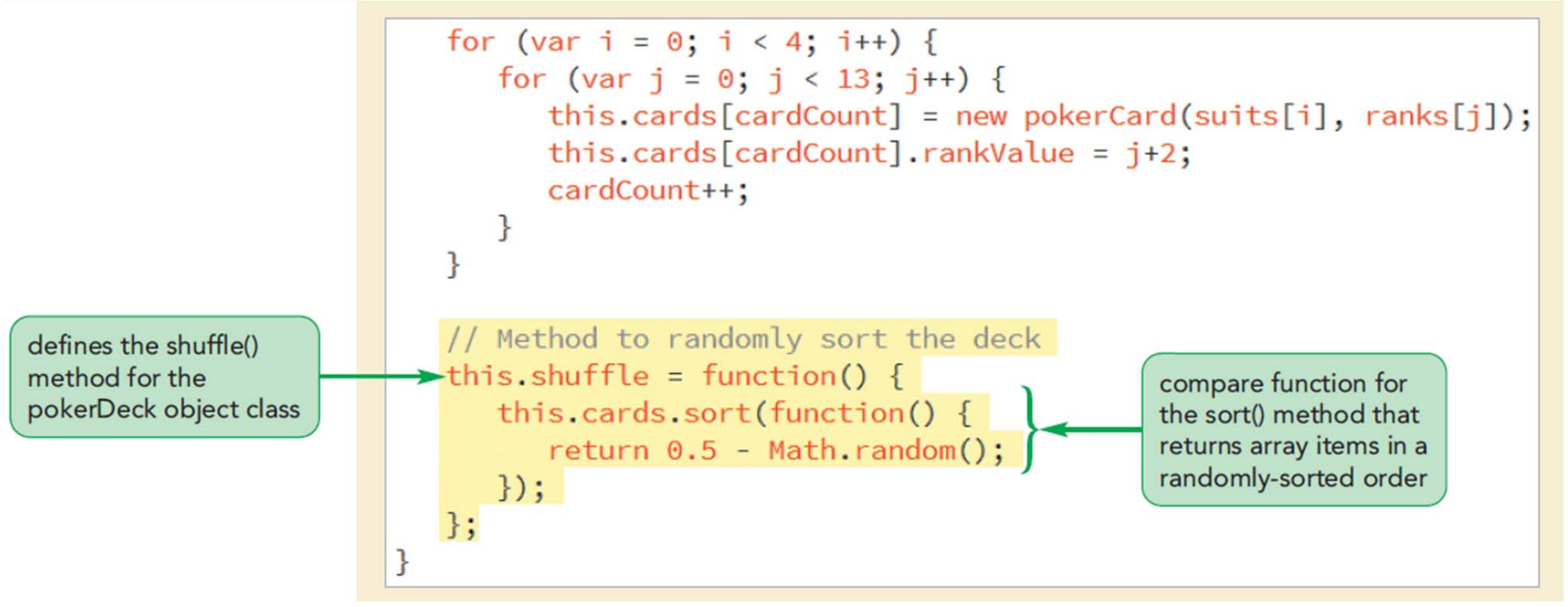
```
    myDeck.cards[4]
```

Combining Object Classes (continued 3)

- All card games require the deck of cards to be randomly sorted
- The `sort()` method of Array objects can be used for random arrangement of the array items
- Add the shuffle() method to the pokerDeck object class to randomize the order of items in the cards array

Combining Object Classes (continued 4)

Figure 14-13 Defining the shuffle() method for pokerDeck objects



```
      for (var i = 0; i < 4; i++) {
         for (var j = 0; j < 13; j++) {
            this.cards[cardCount] = new pokerCard(suits[i], ranks[j]);
            this.cards[cardCount].rankValue = j+2;
            cardCount++;
         }
      }

      // Method to randomly sort the deck
      this.shuffle = function() {
         this.cards.sort(function() {
            return 0.5 - Math.random();
         });
      };
   }
```

defines the shuffle() method for the pokerDeck object class
compare function for the sort() method that returns array items in a randomly-sorted order

Working with Object Prototypes

- Every object has a prototype, which is a template for all the properties and methods associated with the object’s class
- When an object is instantiated from a constructor function, it copies the properties and methods from the prototype into the new object

Defining a Prototype Method

- The prototype is itself an object and is referenced using the following expression:

  ```
  objName.prototype
  ```

  where *`objName`* is the name of the object class
- Example: The prototype for the pokerCard class of objects is referenced as follows:

  ```
  pokerCard.prototype
  ```

Defining a Prototype Method (continued 1)

- Apply the following command to add a method to a prototype:

  ```
  objName.prototype.method = function;
  ```

  where *`method`* is the name of the method and *`function`* is the function applied by the method

Defining a Prototype Method (continued 2)

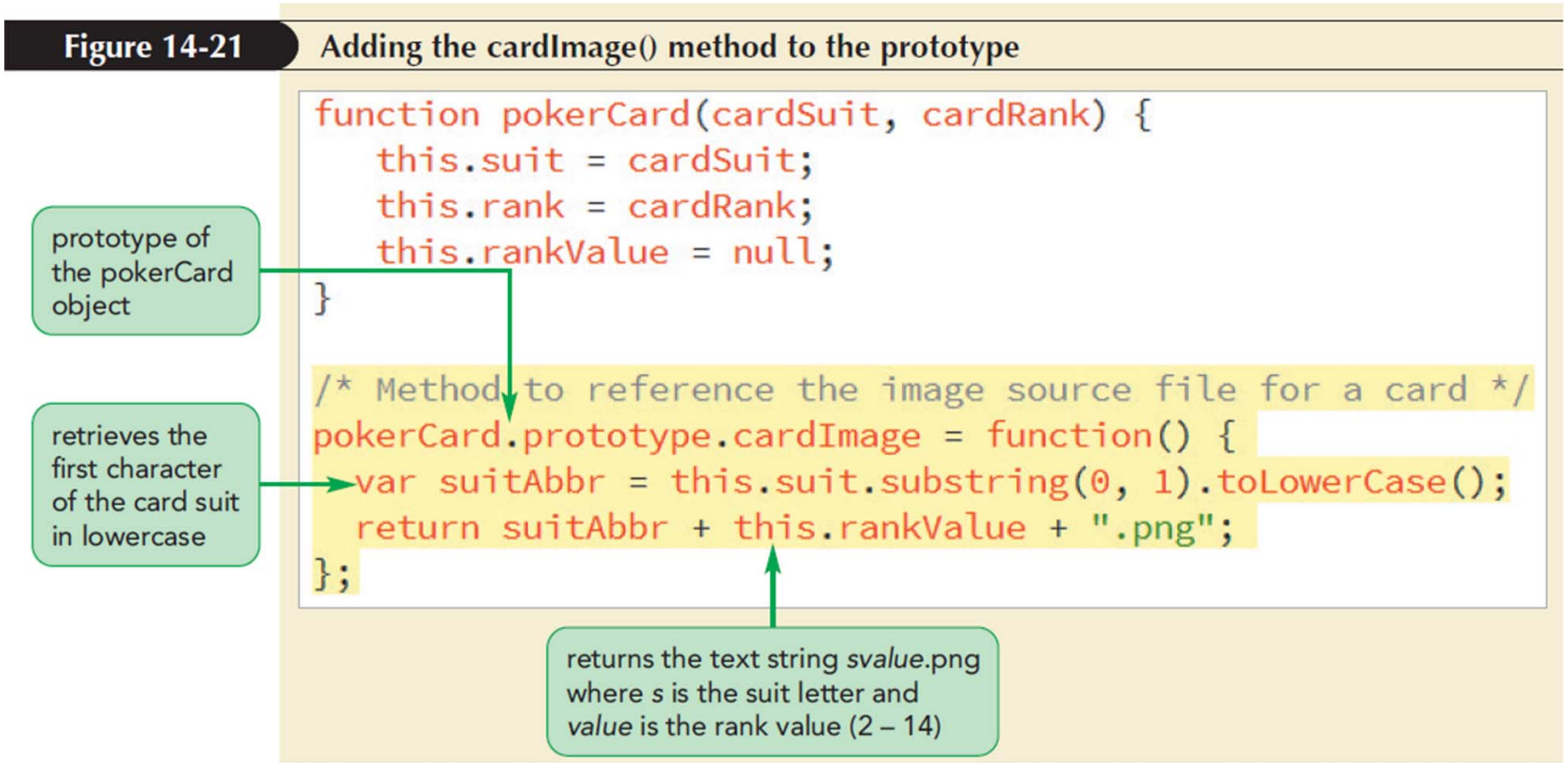


Figure 14-21 Adding the cardImage() method to the prototype

```
function pokerCard(cardSuit, cardRank) {
   this.suit = cardSuit;
   this.rank = cardRank;
   this.rankValue = null;
}

/* Method to reference the image source file for a card */
pokerCard.prototype.cardImage = function() {
  var suitAbbr = this.suit.substring(0, 1).toLowerCase();
  return suitAbbr + this.rankValue + ".png";
};
```

prototype of the pokerCard object
retrieves the first character of the card suit in lowercase
returns the text string *svalue*.png where *s* is the suit letter and *value* is the rank value (2 – 14)

Defining a Prototype Method (continued 3)

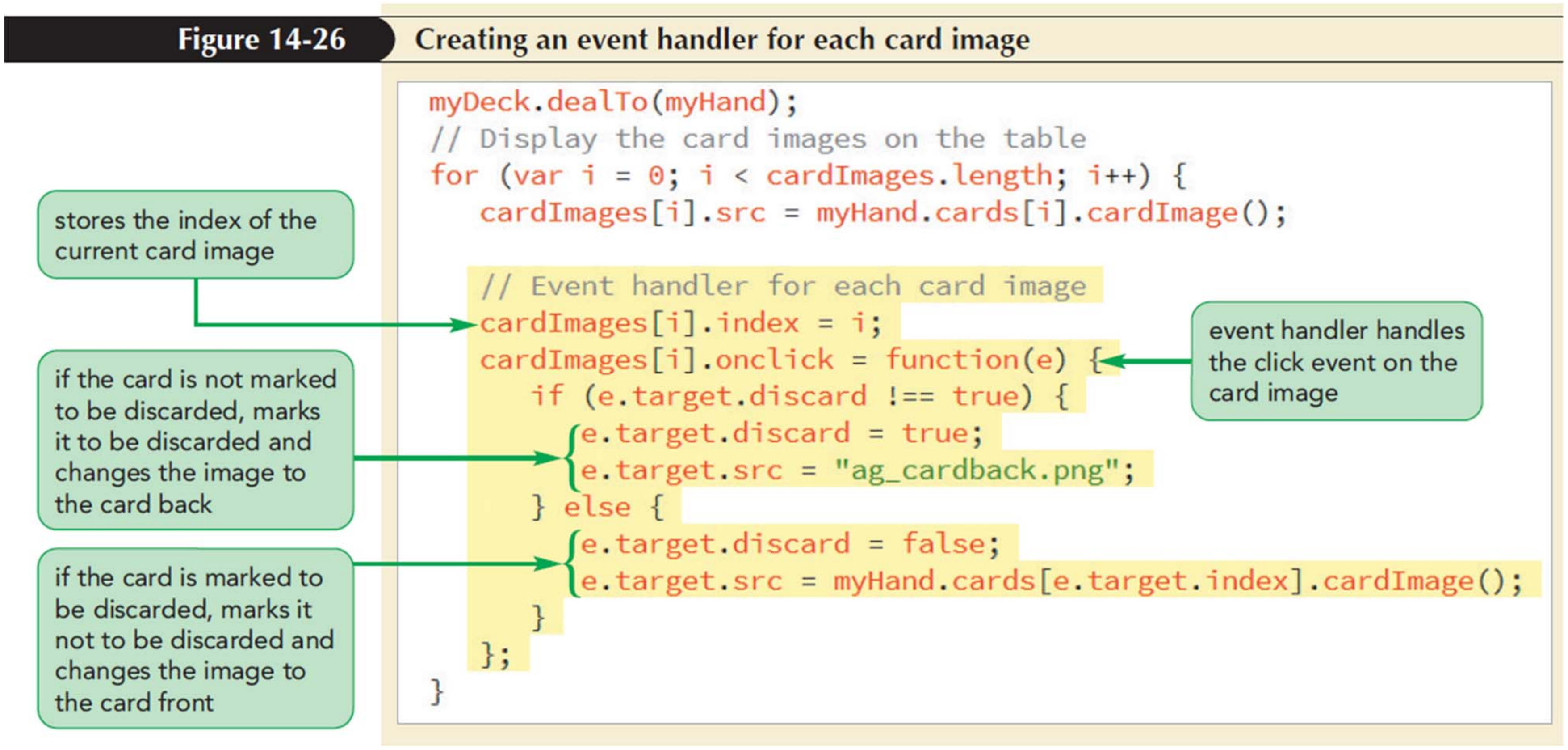


Figure 14-26 Creating an event handler for each card image

```
myDeck.dealTo(myHand);
// Display the card images on the table
for (var i = 0; i < cardImages.length; i++) {
   cardImages[i].src = myHand.cards[i].cardImage();

   // Event handler for each card image
   cardImages[i].index = i;
   cardImages[i].onclick = function(e) {
      if (e.target.discard !== true) {
         e.target.discard = true;
         e.target.src = "ag_cardback.png";
      } else {
         e.target.discard = false;
         e.target.src = myHand.cards[e.target.index].cardImage();
      }
   };
}
```

stores the index of the current card image
if the card is not marked to be discarded, marks it to be discarded and changes the image to the card back
if the card is marked to be discarded, marks it not to be discarded and changes the image to the card front
event handler handles the click event on the card image

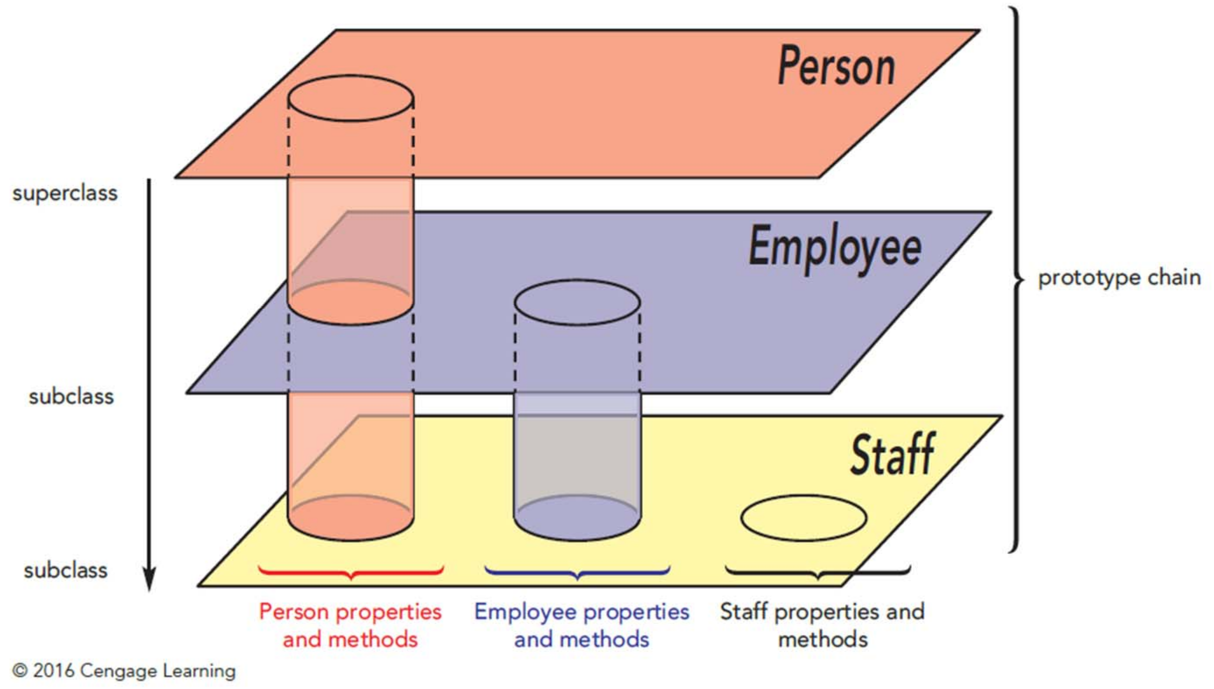
Combining Objects

- Any object class can inherit the properties and methods from another class using prototypes
- The hierarchy of object classes creates a **prototype chain** ranging from **superclass** to **subclasses**
- Superclass: Base object class in a prototype chain
- Subclasses: Lower classes in a prototype chain

Combining Objects (continued)

- **Prototypal inheritance:** Process by which the properties and methods of superclasses are shared with the subclasses

Figure 14-30 Prototypal Inheritance


Person
Employee
Staff
superclass
subclass
subclass
prototype chain
Person properties and methods
Employee properties and methods
Staff properties and methods
© 2016 Cengage Learning

Creating a Prototype Chain

- A prototype chain is created by defining an object prototype as an instance of an object class
- Order of classes is important while defining the prototype chain
- Start at the top of the hierarchy, and move down to the lower subclasses

Creating a Prototype Chain (continued 1)

- JavaScript resolves the code that references an object property or method in the following order:
  - Check for the property or method within the current object instance
  - Check for the property or method with the object’s prototype

Creating a Prototype Chain (continued 2)

  - If the prototype is an instance of another object, check for the property or method in that object
  - Continue moving down the chain until the property or method is located or the end of the chain is reached
- All prototype chains ultimately find their source in the base object

The Base Object

- **Base object** or `Object`: Fundamental JavaScript object whose methods are available to all objects
- A subclass of the base object is created when a custom object is created using an object literal or by applying the `new Object()` command

The Base Object (continued 1)

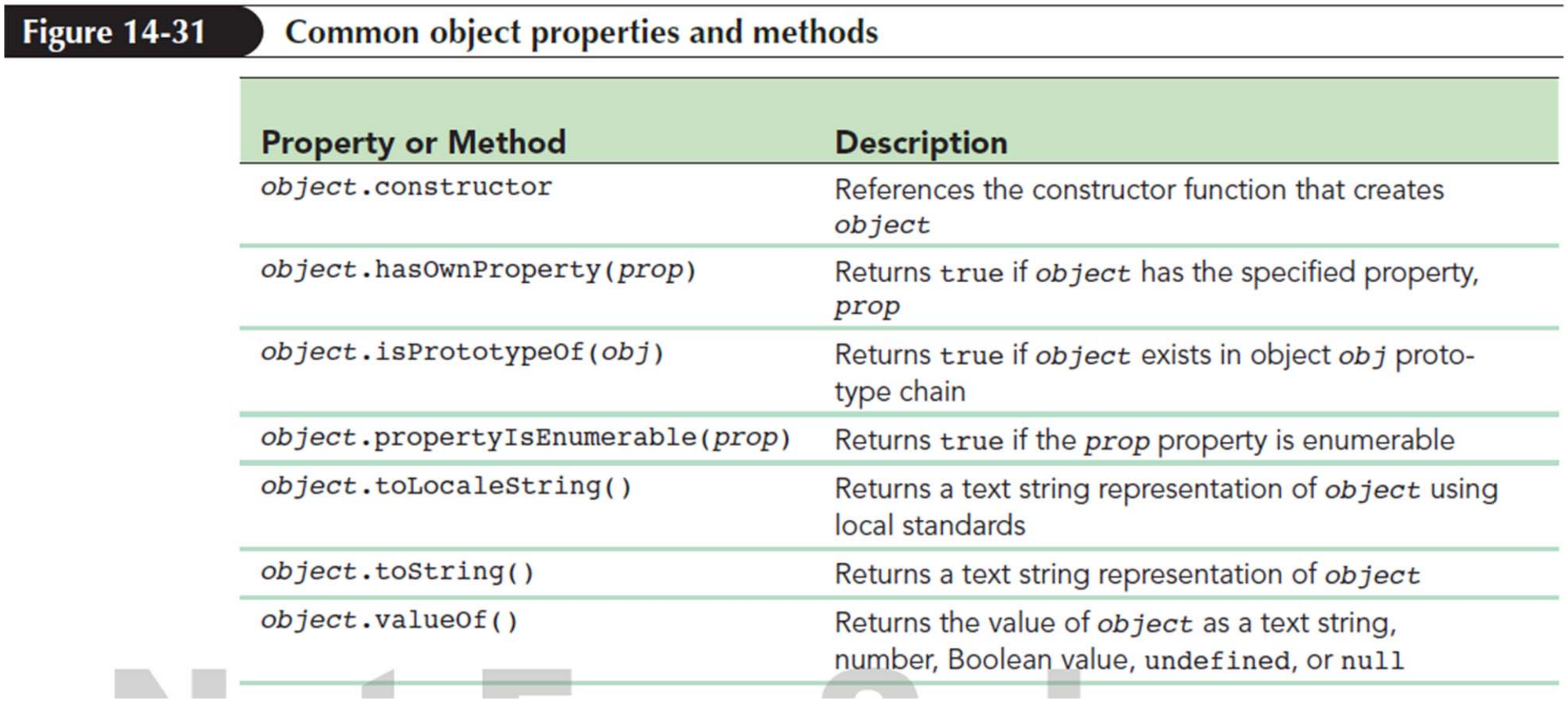


Figure 14-31 Common object properties and methods

Property or Method	Description
object.constructor	References the constructor function that creates *object*
object.hasOwnProperty(*prop*)	Returns true if *object* has the specified property, *prop*
object.isPrototypeOf(*obj*)	Returns true if *object* exists in object *obj* prototype chain
object.propertyIsEnumerable(*prop*)	Returns true if the *prop* property is enumerable
object.toLocaleString()	Returns a text string representation of *object* using local standards
object.toString()	Returns a text string representation of *object*
object.valueOf()	Returns the value of *object* as a text string, number, Boolean value, undefined, or null

The Base Object (continued 2)

- Example: To determine whether an object supports a particular property use the `hasOwnProperty()` method as follows:

```
pokerCard.hasOwnProperty(“rank”);
```

- The code returns `true` if the pokerCard object supports the rank property

The Base Object (continued 3)

- The constructor for `Object` also supports methods that can be used to retrieve and define properties for any object

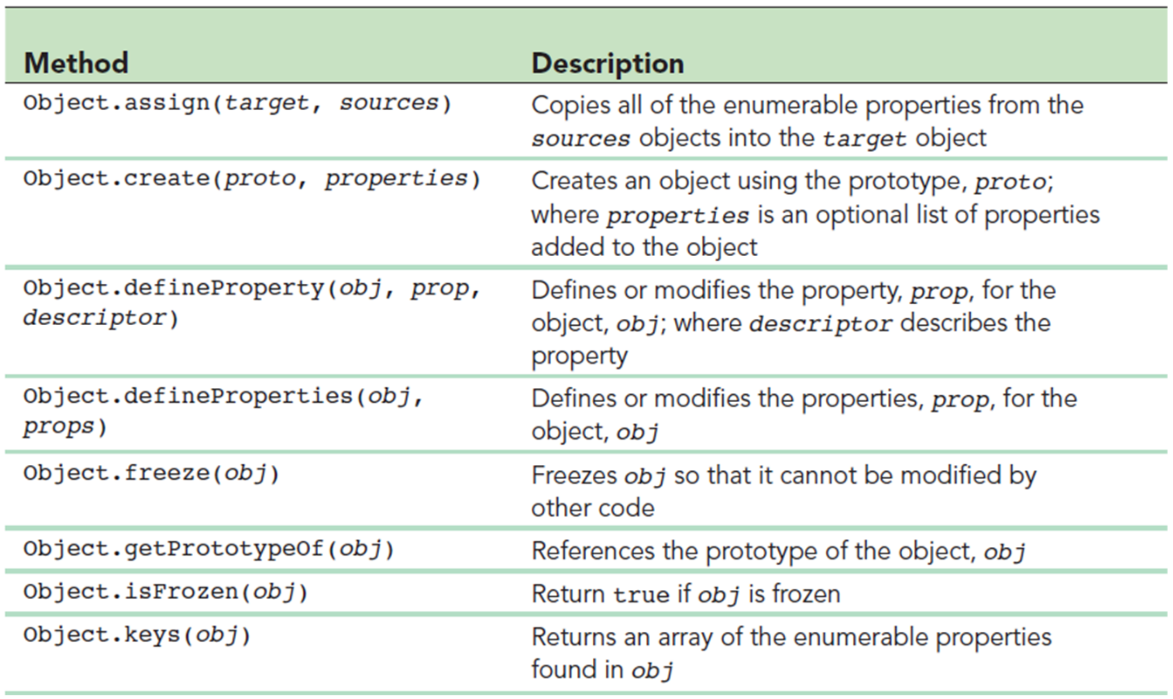


Figure 14-32 Methods of the Object constructor

Method	Description
`Object.assign(`*`target`*`, `*`sources`*`)`	Copies all of the enumerable properties from the *`sources`* objects into the *`target`* object
`Object.create(`*`proto`*`, `*`properties`*`)`	Creates an object using the prototype, *`proto`*; where *`properties`* is an optional list of properties added to the object
`Object.defineProperty(`*`obj`*`, `*`prop`*`, `*`descriptor`*`)`	Defines or modifies the property, *`prop`*, for the object, *`obj`*; where *`descriptor`* describes the property
`Object.defineProperties(`*`obj`*`, `*`props`*`)`	Defines or modifies the properties, *`prop`*, for the object, *`obj`*
`Object.freeze(`*`obj`*`)`	Freezes *`obj`* so that it cannot be modified by other code
`Object.getPrototypeOf(`*`obj`*`)`	References the prototype of the object, *`obj`*
`Object.isFrozen(`*`obj`*`)`	Return `true` if *`obj`* is frozen
`Object.keys(`*`obj`*`)`	Returns an array of the enumerable properties found in *`obj`*

Using the `apply()` and `call()` Methods

- Apply or call a method from one object for use in another object in order to share methods between objects
- Borrow a method from one object class using the following `apply()` method:

  ```
  function.apply(thisObj [,argArray])
  ```

Using the `apply()` and `call()` Methods (continued 1)

where

- *`function`* is a reference to a function
- *`thisObj`* is the object that receives the actions of the function
- *`argArray`* is an optional array of argument values sent to the function

Using the `apply()` and `call()` Methods (continued 2)

- The `call()` method is similar to the `apply()` method except that the argument values are placed in a comma-separated list of values instead of an array
- The syntax of the `call()` method

```
function.call(thisObj, arg1, arg2,
arg3, ...)
```

where *`arg1`*, *`arg2`*, *`arg3`*, and so on is the comma-separated list of argument values for *`function`*

Combining Objects and Arrays

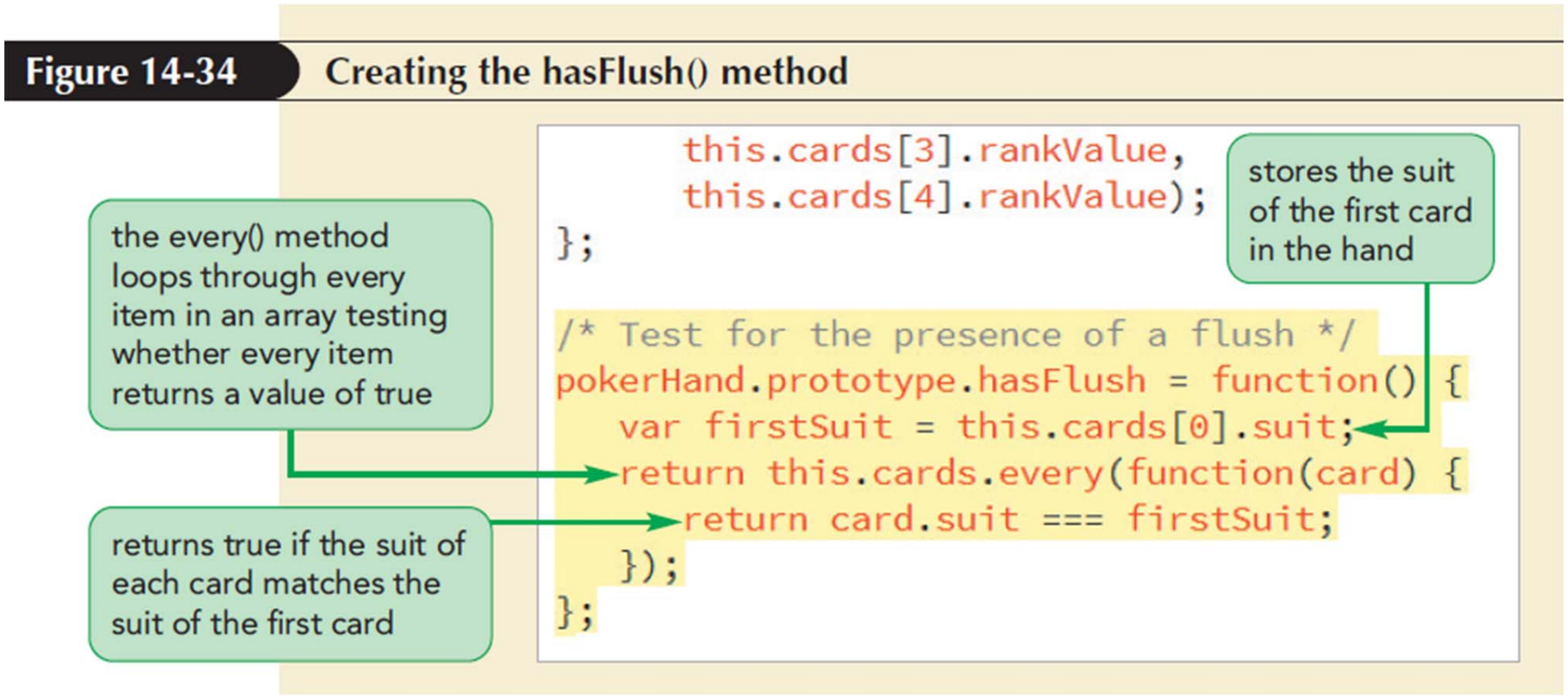
- A custom object contains data stored in arrays
- JavaScript’s built-in `Array` object methods speed up the efficiency of a code by looping through the contents of an array

Applying the `every()` Array method

- Use the `every()` method to test whether every item in an array matches a specified condition
- Example: Use `every()` method to test whether every card in the cards array of the pokerHand object has the same value for the suit property

Applying the `every()` Array method (continued)

Figure 14-34 Creating the hasFlush() method



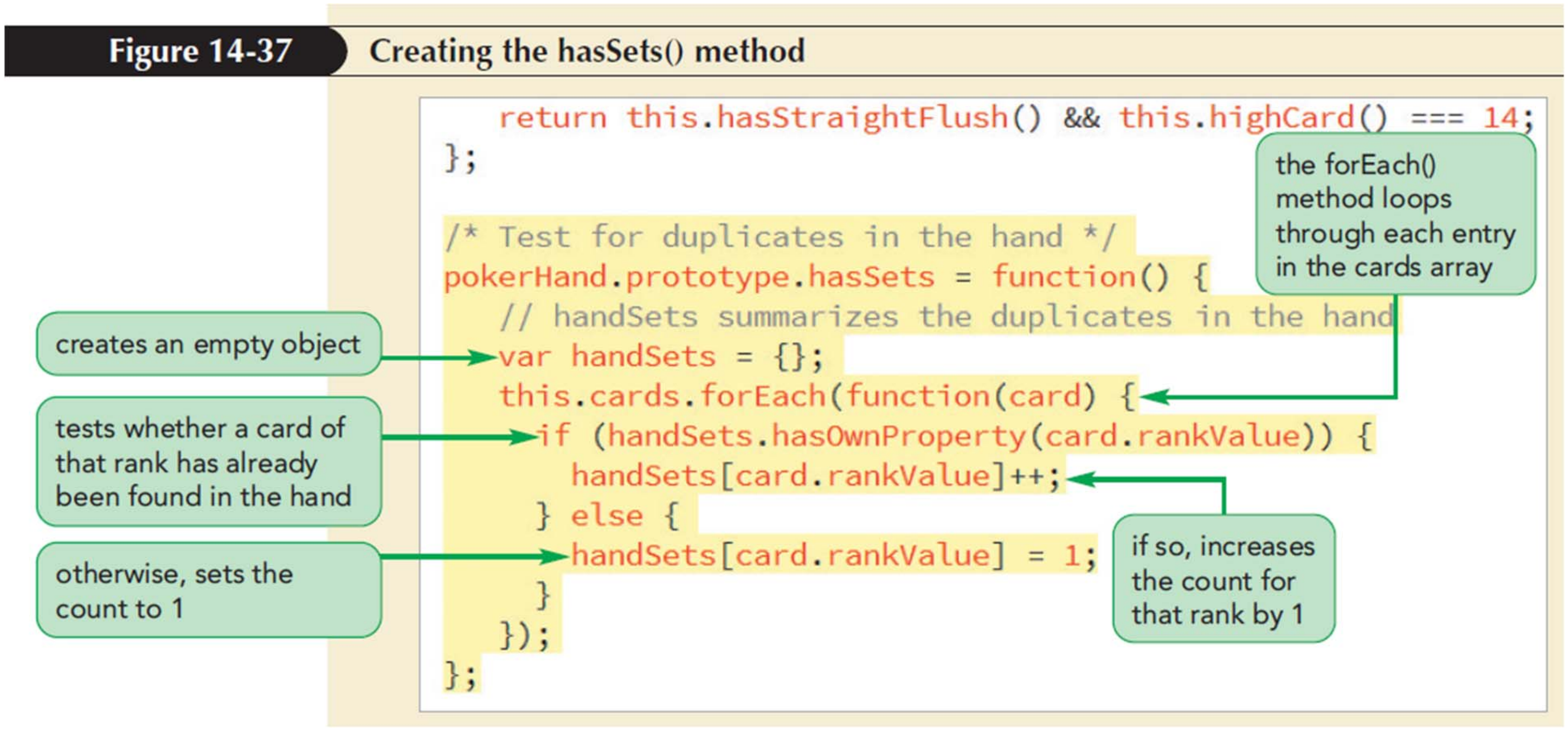
```
        this.cards[3].rankValue,
        this.cards[4].rankValue);
};

/* Test for the presence of a flush */
pokerHand.prototype.hasFlush = function() {
   var firstSuit = this.cards[0].suit;
   return this.cards.every(function(card) {
      return card.suit === firstSuit;
   });
};
```

stores the suit of the first card in the hand
the every() method loops through every item in an array testing whether every item returns a value of true
returns true if the suit of each card matches the suit of the first card

Creating an Object Literal with the `forEach()` Method

- `forEach()` method is used to run a command block for each item in an array

Figure 14-37 Creating the hasSets() method



```
   return this.hasStraightFlush() && this.highCard() === 14;
};

/* Test for duplicates in the hand */
pokerHand.prototype.hasSets = function() {
   // handSets summarizes the duplicates in the hand
   var handSets = {};
   this.cards.forEach(function(card) {
      if (handSets.hasOwnProperty(card.rankValue)) {
         handSets[card.rankValue]++;
      } else {
         handSets[card.rankValue] = 1;
      }
   });
};
```

the forEach() method loops through each entry in the cards array
creates an empty object
tests whether a card of that rank has already been found in the hand
if so, increases the count for that rank by 1
otherwise, sets the count to 1

Applying a `for…in` loop

- `for…in` loop is used to examine the properties and keys of an object as follows:

```
for (prop in obj) {
   commands
}
```

where *`prop`* references the properties contained within the *`obj`* object

Applying a `for…in` loop (continued 1)

- Example: Loop through the contents of the following employee object:

```
    var employee = {
    name: “Robert Voiklund”,
    position: “manager”,
    email: “rvoiklund@example.com”  };
```

- Solution: Apply the following `for … in` loop:

```
    for (prop in employee) {
    console.info(prop + “ is “ +
    employee[prop]);  }
```

Applying a `for…in` loop (continued 2)

- `for…in` loops do not follow a specific order because properties can be listed and read out in any order
- Only the properties that are countable or **enumerable** are accessible to `for…in` loops

Applying a `for…in` loop (continued 3)

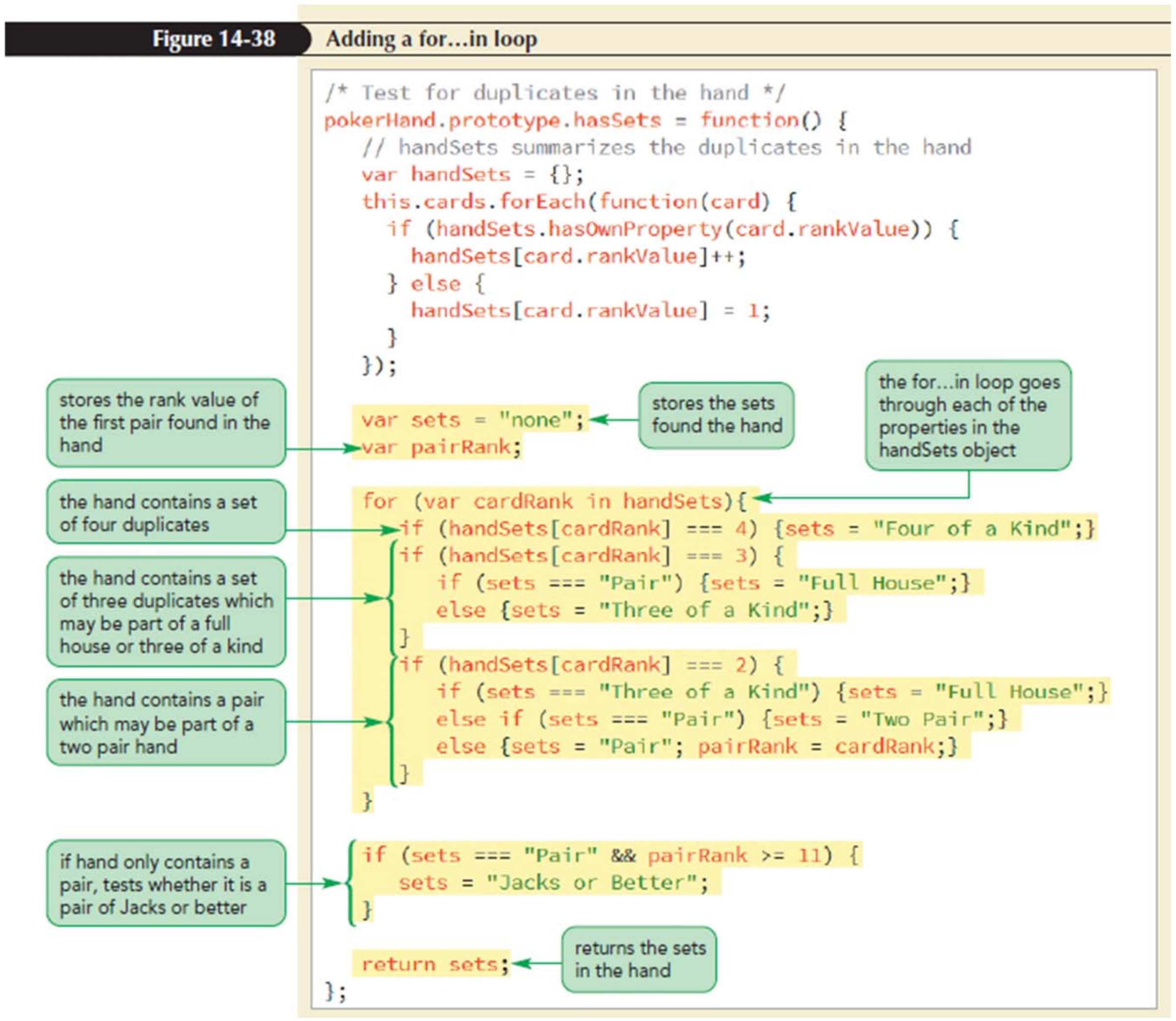
- Determine whether a property is enumerable using the `propertyIsEnumerable()` method

  ```
  obj.propertyIsEnumerable(prop)
  ```

  where *`obj`* is the object and *`prop`* is the property
- Use a `for…in` loop to loop through every property in the object

Applying a `for…in` loop (continued 4)

Figure 14-38 Adding a for...in loop



```
/* Test for duplicates in the hand */
pokerHand.prototype.hasSets = function() {
   // handSets summarizes the duplicates in the hand
   var handSets = {};
   this.cards.forEach(function(card) {
     if (handSets.hasOwnProperty(card.rankValue)) {
       handSets[card.rankValue]++;
     } else {
       handSets[card.rankValue] = 1;
     }
   });

   var sets = "none";
   var pairRank;

   for (var cardRank in handSets){
      if (handSets[cardRank] === 4) {sets = "Four of a Kind";}
      if (handSets[cardRank] === 3) {
         if (sets === "Pair") {sets = "Full House";}
         else {sets = "Three of a Kind";}
      }
      if (handSets[cardRank] === 2) {
         if (sets === "Three of a Kind") {sets = "Full House";}
         else if (sets === "Pair") {sets = "Two Pair";}
         else {sets = "Pair"; pairRank = cardRank;}
      }
   }

   if (sets === "Pair" && pairRank >= 11) {
      sets = "Jacks or Better";
   }

   return sets;
};
```

stores the rank value of the first pair found in the hand
stores the sets found the hand
the for...in loop goes through each of the properties in the handSets object
the hand contains a set of four duplicates
the hand contains a set of three duplicates which may be part of a full house or three of a kind
the hand contains a pair which may be part of a two pair hand
if hand only contains a pair, tests whether it is a pair of Jacks or better
returns the sets in the hand