Computer Science Principles
Web Programming

JavaScript Programming Essentials

CHAPTER 17: MAKING A SNAKE GAME PART 2

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- In this chapter, we’ll finish building our Snake game. In Chapter 16, we set up the playing area and covered how the game would work in general. Now we’ll create the objects that represent the snake and apple in the game, and we’ll program a keyboard event handler so that the player can control the snake with the arrow keys. Finally, we’ll look at the complete code listing for the program.

Objectives

- As we create the snake and apple objects for this game, we’ll use the object-oriented programming techniques we learned in Chapter 12 to create constructors and methods for each object. Both our snake and apple objects will rely on a more basic block object, which we’ll use to represent one block on the game board grid. Let’s start by building a constructor for that simple block object.

Building the Block Constructor

LECTURE 1

Building the Block Constructor

- In this section, we'll define a Block constructor that will create objects that represent individual blocks on our invisible game grid. Each block will have the properties col (short for column) and row, which will store the location of that particular block on the grid.
- Figure 17-1 shows this grid with some of the columns and rows numbered. Although this grid won't actually appear on the screen, our game is designed so that the apple and the snake segments will always line up with it.

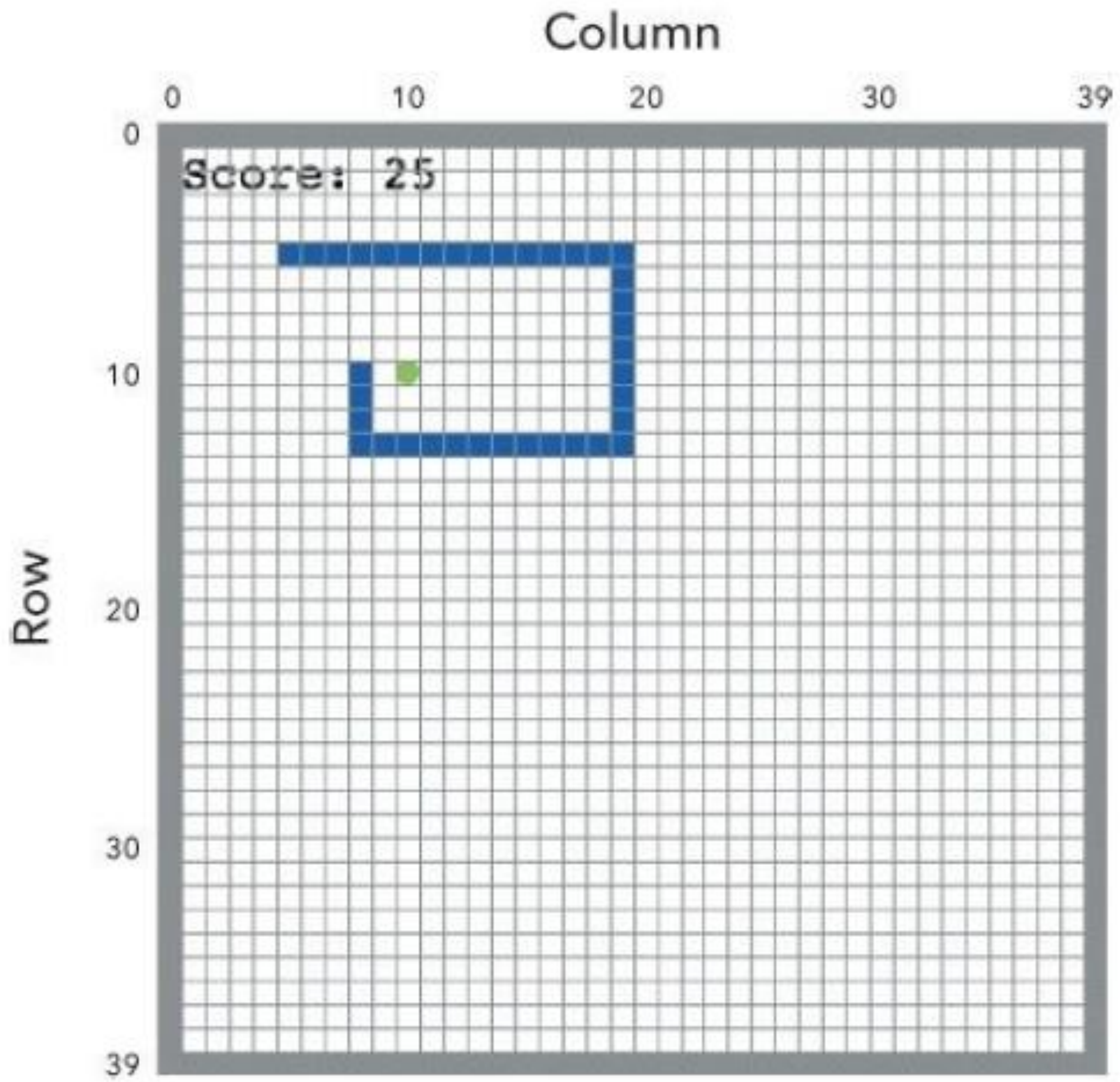

Figure 17-1. The column and row numbers used by the `Block` constructor

Building the Block Constructor

- In Figure 17-1, the block containing the green apple is at column 10, row 10. The head of the snake (to the left of the apple) is at column 8, row 10. Here's the code for the Block constructor:

```
var Block = function (col, row) {
  this.col = col;
  this.row = row;
};
```

- Column and row values are passed into the Block constructor as arguments and saved in the col and row properties of the new object. Now we can use this constructor to create an object representing a particular block on the game grid.

- For example, here's how we'd create an object that represents the block in column 5, row 5:

```
var sampleBlock = new Block(5, 5);
```

Adding the drawSquare Method

- So far this block object lets us represent a location on the grid, but to actually make something appear at that location, we'll need to draw it on the canvas. Next, we'll add two methods, drawSquare and drawCircle, that will let us draw a square or a circle, respectively, in a particular block on the grid.
- First, here's the drawSquare method:

```
    Block.prototype.drawSquare = function (color) {
❶     var x = this.col * blockSize;
❷     var y = this.row * blockSize;
      ctx.fillStyle = color;
      ctx.fillRect(x, y, blockSize, blockSize);
    };
```

Adding the drawSquare Method

- In Chapter 12 we learned that if you attach methods to the prototype property of a constructor, those methods will be available to any objects created with that constructor. So by adding the drawSquare method to Block.protoype, we make it available to any block objects.
- This method draws a square at the location given by the block's col and row properties. It takes a single argument, color, which determines the color of the square. To draw a square with canvas, we need to provide the *x*- and *y*-positions of the top-left corner of the square. At ❶ and ❷ we calculate these *x*- and *y*-values for the current block by multiplying the col and row properties by blockSize. We then set the fillStyle property of the drawing context to the method's color argument.

Adding the drawSquare Method

- Finally, we call ctx.fillRect, passing our computed *x*- and *y*-values and blockSize for both the width and height of the square.
- Here's how we would create a block in column 3, row 4, and draw it:

```
var sampleBlock = new Block(3, 4);
sampleBlock.drawSquare("LightBlue");
```

- Figure 17-2 shows this square drawn on the canvas and how the measurements for the square are calculated.

Adding the drawCircle Method

Now for the drawCircle method. It is very similar to the drawSquare method, but it draws a filled circle instead of a square.

```
Block.prototype.drawCircle = function (color) {
    var centerX = this.col * blockSize + blockSize / 2;
    var centerY = this.row * blockSize + blockSize / 2;
    ctx.fillStyle = color;
    circle(centerX, centerY, blockSize / 2, true);
};
```

Adding the drawCircle Method

- First we calculate the location of the circle's center by creating two new variables, centerX and centerY. As before, we multiply the col and row properties by blockSize, but this time we also have to add blockSize / 2, because we need the pixel coordinates for the circle's center, which is in the middle of a block (as shown in Figure 17-3).
- We set the context fillStyle to the color argument as in drawSquare and then call our trusty circle function, passing centerX and centerY for the *x*- and *y*-coordinates, blockSize / 2 for the radius, and true to tell the function to fill the circle. This is the same circle function we defined in Chapter 14, so we'll have to include the definition for that function once again in this program (as you can see in the final code listing).

Adding the drawCircle Method

Here’s how we could draw a circle in column 4, row 3:

```
var sampleCircle = new Block(4, 3);
sampleCircle.drawCircle("LightGreen");
```

Figure 17-3 shows the circle, with the calculations for the center point and radius.

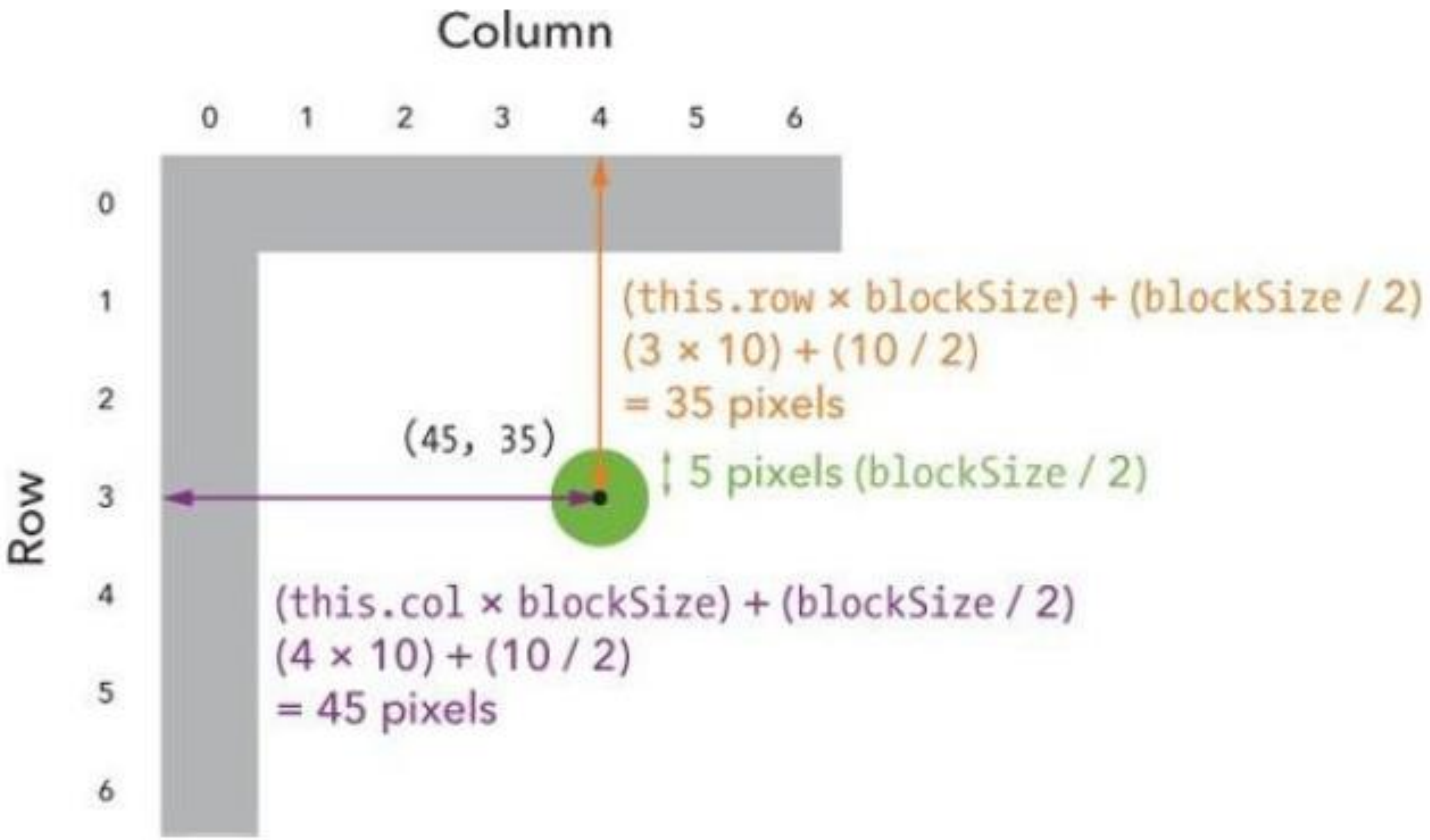

Figure 17-3. Calculating the values for drawing a circle

Adding the equal Method

- In our game, we'll need to know whether two blocks are in the same location. For example, if the apple and the snake's head are in the same location, that means the snake has eaten the apple. On the other hand, if the snake's head and tail are in the same location, then the snake has collided with itself.
- To make it easier to compare block locations, we'll add a method, equal, to the Block constructor prototype. When we call equal on one block object and pass another object as an argument, it will return true if they are in the same location (and false if not). Here's the code:

```
Block.prototype.equal = function (otherBlock) {
  return this.col === otherBlock.col && this.row === otherBlock.row;
};
```

Adding the equal Method

This method is pretty straightforward: if the two blocks (this and otherBlock) have the same col and row properties (that is, if this.col is equal to otherBlock.col and this.row is equal to otherBlock.row), then they are in the same place, and the method returns true.

For example, let's create two new blocks called apple and head and see if they're in the same location:

```
var apple = new Block(2, 5);

var head = new Block(3, 5);

head.equal(apple);

false
```

Adding the equal Method

- Although apple and head have the same row property (5), their col properties are different. If we set the head to a new block object one column to the left, now the method will tell us that the two objects are in the same location:

```
head = new Block(2, 5);
head.equal(apple);
true
```

- Note that it doesn't make any difference whether we write head.equal(apple) or apple.equal(head); in both cases we're making the same comparison. We'll use the equal method later to check whether the snake has eaten the apple or collided with itself.

Creating the Snake

LECTURE 2

Creating the Snake

- Now we'll create the snake. We'll store the snake's position as an array called **segments**, which will contain a series of block objects. To move the snake, we'll add a new block to the beginning of the segments array and remove the block at the end of the array. The first element of the **segments** array will represent the head of the snake.

Writing the Snake Constructor

First we need a constructor to create our snake object:

```
   var Snake = function () {
❶     this.segments = [
         new Block(7, 5),
         new Block(6, 5),
         new Block(5, 5)
      ];
❷     this.direction = "right";
❸     this.nextDirection = "right";
   };
```

Defining the Snake Segments

- The **segments** property at ❶ is an array of block objects that each represent a segment of the snake's body. When we start the game, this array will contain three blocks at (7, 5), (6, 5), and (5, 5). Figure 17-4 shows these initial three segments of the snake.

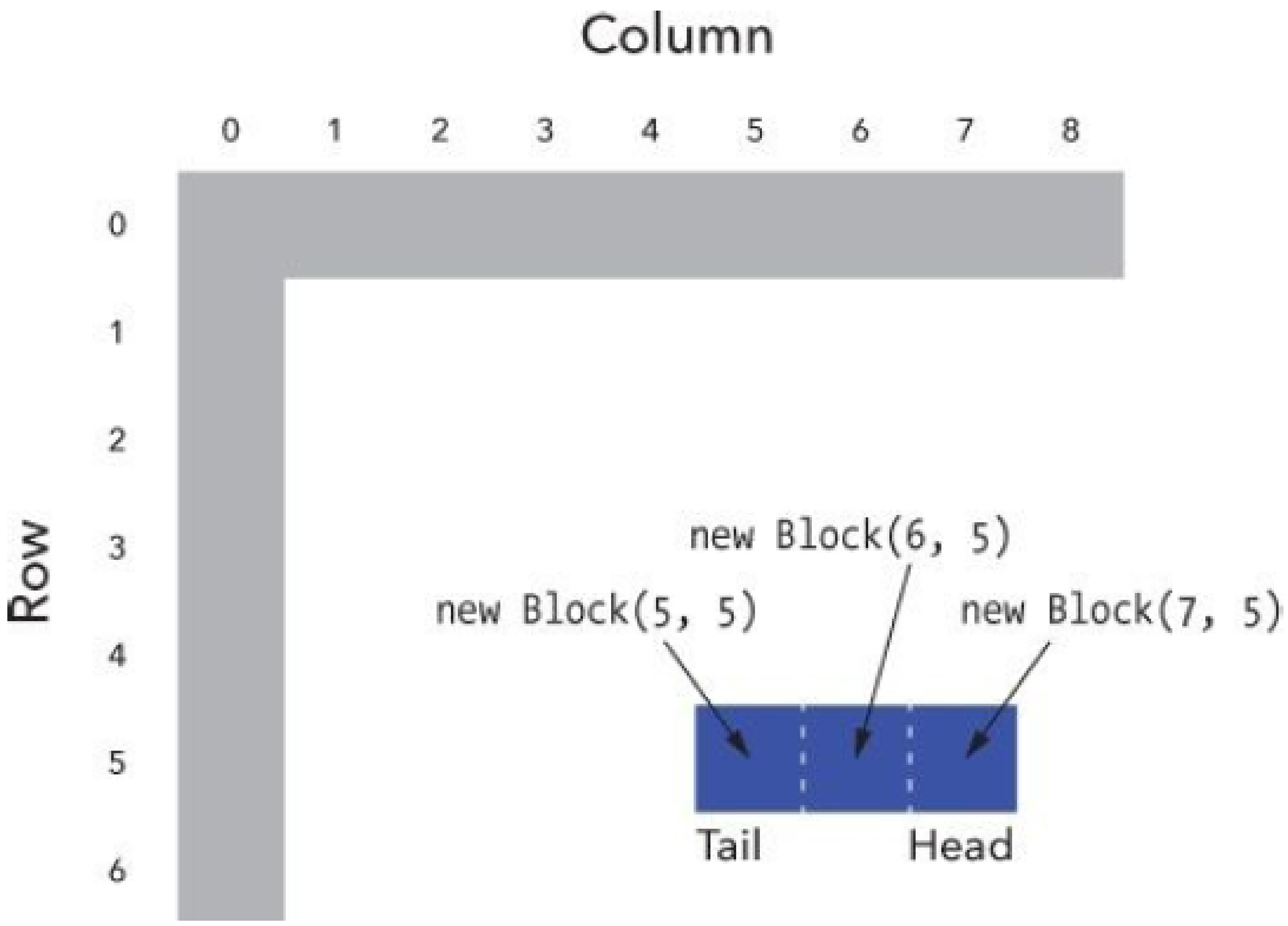

Figure 17-4. The initial blocks that make up the snake

Setting the Direction of Movement

- The direction property at ❷ stores the current direction of the snake. Our constructor also adds the nextDirection property at ❸, which stores the direction in which the snake will move for the next animation step. This property will be updated by our keydown event handler when the player presses an arrow key (see Adding the keydown Event Handler). For now, the constructor sets both of these properties to "right", so at the beginning of the game our snake will move to the right.

Drawing the Snake

- To draw the snake, we simply have to loop through each of the blocks in its segments array, calling the drawSquare method we created earlier on each block. This will draw a square for each segment of the snake.

```
Snake.prototype.draw = function () {
  for (var i = 0; i < this.segments.length; i++) {
  this.segments[i].drawSquare("Blue");
  }
};
```

Drawing the Snake

- The draw method uses a for loop to operate on each block object in the segments array. Each time around the loop, this code takes the current segment (this.segments[i]) and calls drawSquare("Blue") on it, which draws a blue square in the corresponding block.
- If you want to test out the draw method, you can run the following code, which creates a new object using the Snake constructor and calls its draw method:

```
var snake = new Snake();
snake.draw();
```

Moving the Snake

LECTURE 3

Moving the Snake

- We'll create a move method to move the snake one block in its current direction. To move the snake, we add a new head segment (by adding a new block object to the beginning of the segments array) and then remove the tail segment from the end of the segments array.
- The move method will also call a method, checkCollision, to see whether the new head has collided with the rest of the snake or with the wall, and whether the new head has eaten the apple. If the new head has collided with the body or the wall, we end the game by calling the gameOver function we created in Chapter 16. If the snake has eaten the apple, we increase the score and move the apple to a new location.

Adding the move Method

The move method looks like this:

```
    Snake.prototype.move = function () {
❶       var head = this.segments[0];
❷       var newHead;
❸       this.direction = this.nextDirection;
❹       if (this.direction === "right") {
            newHead = new Block(head.col + 1, head.row);
        } else if (this.direction === "down") {
            newHead = new Block(head.col, head.row + 1);
        } else if (this.direction === "left") {
            newHead = new Block(head.col - 1, head.row);
        } else if (this.direction === "up") {
            newHead = new Block(head.col, head.row - 1);
        }
❺       if (this.checkCollision(newHead)) {
            gameOver();
            return;
        }
❻       this.segments.unshift(newHead);
❼       if (newHead.equal(apple.position)) {
            score++;
            apple.move();
        } else {
            this.segments.pop();
        }
    };
```

Let’s walk through this method piece by piece.

Creating a New Head

- At ❶ we save the first element of the this.segments array in the variable head. We'll refer to this first segment of the snake many times in this method, so using this variable will save us some typing and make the code a bit easier to read. Now, instead of repeating this.segments[0] over and over again, we can just type head.
- At ❷ we create the variable newHead, which we'll use to store the block representing the new head of the snake (which we're about to add).
- At ❸ we set this.direction equal to this.nextDirection, which updates the direction of the snake's movement to match the most recently pressed arrow key. (We'll see how this works in more detail when we look at the keydown event handler.)

Creating a New Head

- Beginning at ❹, we use a chain of if...else statements to determine the snake’s direction. In each case, we create a new head for the snake and save it in the variable newHead.
- Depending on the direction of movement, we add or subtract one from the row or column of the existing head to place this new head directly next to the old one (either right, left, up, or down depending on the snake’s direction of movement). For example, Figure 17-5 shows how the new head is added to the snake when this.nextDirection is set to "down".

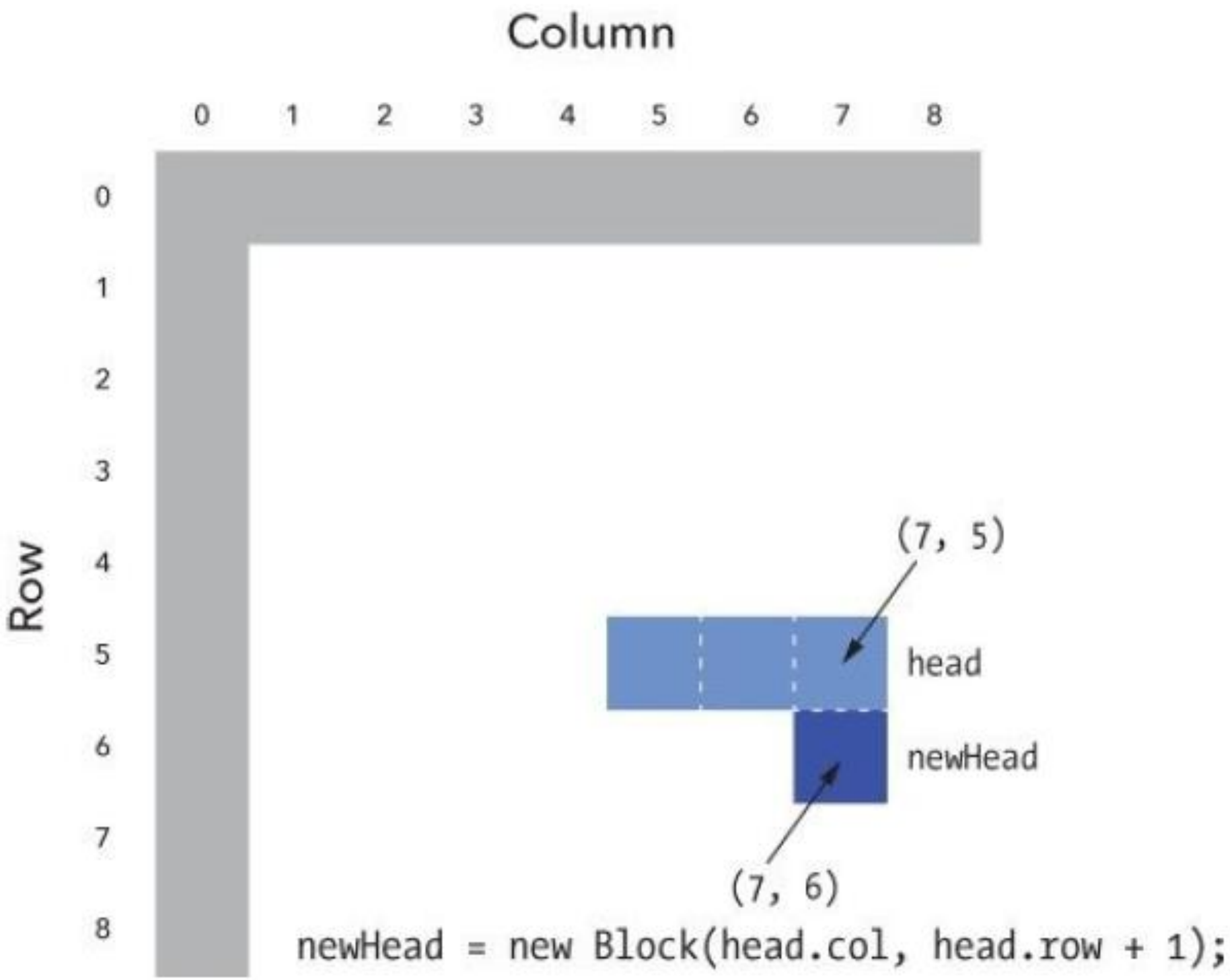

Figure 17-5. Creating `newHead` when `this.nextDirection` is "down"

Checking for Collisions and Adding the Head

- At ❺ we call the checkCollision method to find out whether the snake has collided with a wall or with itself. We'll see the code for this method in a moment, but as you might guess, this method will return true if the snake has collided with something. If that happens, the body of the if statement calls the gameOver function to end the game and print "Game Over" on the canvas.
- The return keyword that follows the call to gameOver exits the move method early, skipping any code that comes after it. We reach the return keyword only if checkCollision returns true, so if the snake hasn't collided with anything, we execute the rest of the method.

Checking for Collisions and Adding the Head

- As long as the snake hasn't collided with something, we add the new head to the front of the snake at ❻ by using unshift to add **newHead** to the beginning of the segments array. For more about how the unshift method works on arrays, see Adding Elements to an Array.

Eating the Apple

- At ❼, we use the equal method to compare newHead and apple.position. If the two blocks are in the same location, the equal method will return true, which means that the snake has eaten the apple.
- If the snake has eaten the apple, we increase the score and then call move on the apple to move it to a new location. If the snake has not eaten the apple, we call pop on this.segments. This removes the snake’s tail while keeping the snake the same size (since move already added a segment to the snake’s head). When the snake eats an apple, it grows by one segment because we add a segment to its head without removing the tail.

Eating the Apple

- We haven't defined apple yet, so this method won't fully work in its current form. If you want to test it out, you can delete the whole if...else statement at ❼ and replace it with this line:

```
this.segments.pop();
```

- Then all you need to do is define the checkCollision method, which we'll do next.

Adding the checkCollision Method

- Each time we set a new location for the snake's head, we have to check for collisions. Collision detection, a very common step in game mechanics, is often one of the more complex aspects of game programming. Fortunately, it's relatively straightforward in our Snake game.

- We care about two types of collisions in our Snake game: collisions with the wall and collisions with the snake itself. A wall collision happens if the snake hits a wall. The snake can collide with itself if you turn the head so that it runs into the body. At the start of the game, the snake is too short to collide with itself, but after eating a few apples, it can.

Adding the checkCollision Method

Here is the checkCollision method:

```
   Snake.prototype.checkCollision = function (head) {
❶     var leftCollision = (head.col === 0);
      var topCollision = (head.row === 0);
      var rightCollision = (head.col === widthInBlocks - 1);
      var bottomCollision = (head.row === heightInBlocks - 1);
❷     var wallCollision = leftCollision || topCollision ||
                          rightCollision || bottomCollision;
❸     var selfCollision = false;
❹     for (var i = 0; i < this.segments.length; i++) {
         if (head.equal(this.segments[i])) {
❺             selfCollision = true;
         }
      }
❻     return wallCollision || selfCollision;
   };
```

Checking for Wall Collisions

- At ❶ we create the variable leftCollision and set it to the value of head.col === 0. This variable will be true if the snake collides with the left wall; that is, when it is in column 0. Similarly, the variable topCollision in the next line checks the row of the snake's head to see if it has run into the top wall.
- After that, we check for a collision with the right wall by checking whether the column value of the head is equal to widthInBlocks - 1. Since widthInBlocks is set to 40, this checks whether the head is in column 39, which corresponds to the right wall, as you can see back in Figure 17-1. Then we do the same thing for bottomCollision, checking whether the head's row property is equal to heightInBlocks - 1.

Checking for Wall Collisions

- At ❷, we determine whether the snake has collided with a wall by checking to see if **leftCollision** *or* **topCollision** *or* **rightCollision** *or* **bottomCollision** is true, using the || (or) operator. We save the Boolean result in the variable wallCollision.

Checking for Self-Collisions

- To determine whether the snake has collided with itself, we create a variable at ❸ called selfCollision and initially set it to false. Then at ❹ we use a for loop to loop through all the segments of the snake to determine whether the new head is in the same place as any segment, using **head.equal(this.segments[i]).** The head and all of the other segments are blocks, so we can use the equal method that we defined for block objects to see whether they are in the same place. If we find that any of the snake's segments are in the same place as the new head, we know that the snake has collided with itself, and we set **selfCollision** to true (at ❺).
- Finally, at ❻, we return **wallCollision** || **selfCollision**, which will be true if the snake has collided with either the wall or itself.

Setting the Snake's Direction with the Keyboard

- Next we'll write the code that lets the player set the snake's direction using the keyboard. We'll add a **keydown** event handler to detect when an arrow key has been pressed, and we'll set the snake's direction to match that key.

Setting the Snake’s Direction with the Keyboard

LECTURE 4

Setting the Snake's Direction with the Keyboard

- Next we'll write the code that lets the player set the snake's direction using the keyboard. We'll add a **keydown** event handler to detect when an arrow key has been pressed, and we'll set the snake's direction to match that key.

Adding the keydown Event Handler

This code handles keyboard events:

```
❶ var directions = {
    37: "left",
    38: "up",
    39: "right",
    40: "down"
    };
❷ $("body").keydown(function (event) {
    var newDirection = directions[event.keyCode];
❸           if (newDirection !== undefined) {
      snake.setDirection(newDirection);
    }
  });
```

Adding the keydown Event Handler

- At ❶ we create an object to convert the arrow keycodes into strings indicating the direction they represent (this object is quite similar to the **keyActions** object we used in Reacting to the Keyboard).
- At ❷ we attach an event handler to the **keydown** event on the body element. This handler will be called when the user presses a key (as long as they've clicked inside the web page first).
- This handler first converts the event's keycode into a direction string, and then it saves the string in the variable **newDirection**. If the keycode is not 37, 38, 39, or 40 (the keycodes for the arrow keys we care about), **directions[event.keyCode]** will be **undefined**.

Adding the keydown Event Handler

- At ❸ we check to see if **newDirection** is not equal to undefined. If it's not **undefined**, we call the **setDirection** method on the snake, passing the **newDirection** string. (Because there is no **else** case in this if statement, if **newDirection** is undefined, then we just ignore the keypress.)
- This code won't work yet because we haven't defined the **setDirection** method on the snake. Let's do that now.

Adding the setDirection Method

The **setDirection** method takes the new direction from the keyboard handler we just looked at and uses it to update the snake's direction. This method also prevents the player from making turns that would have the snake immediately run into itself. For example, if the snake is moving right, and then it suddenly turns left without moving up or down to get out of its own way, it will collide with itself. We'll call these *illegal* turns because we do not want to allow the player to make them. For example, Figure 17-6 shows the valid directions and the one illegal direction when the snake is moving right.

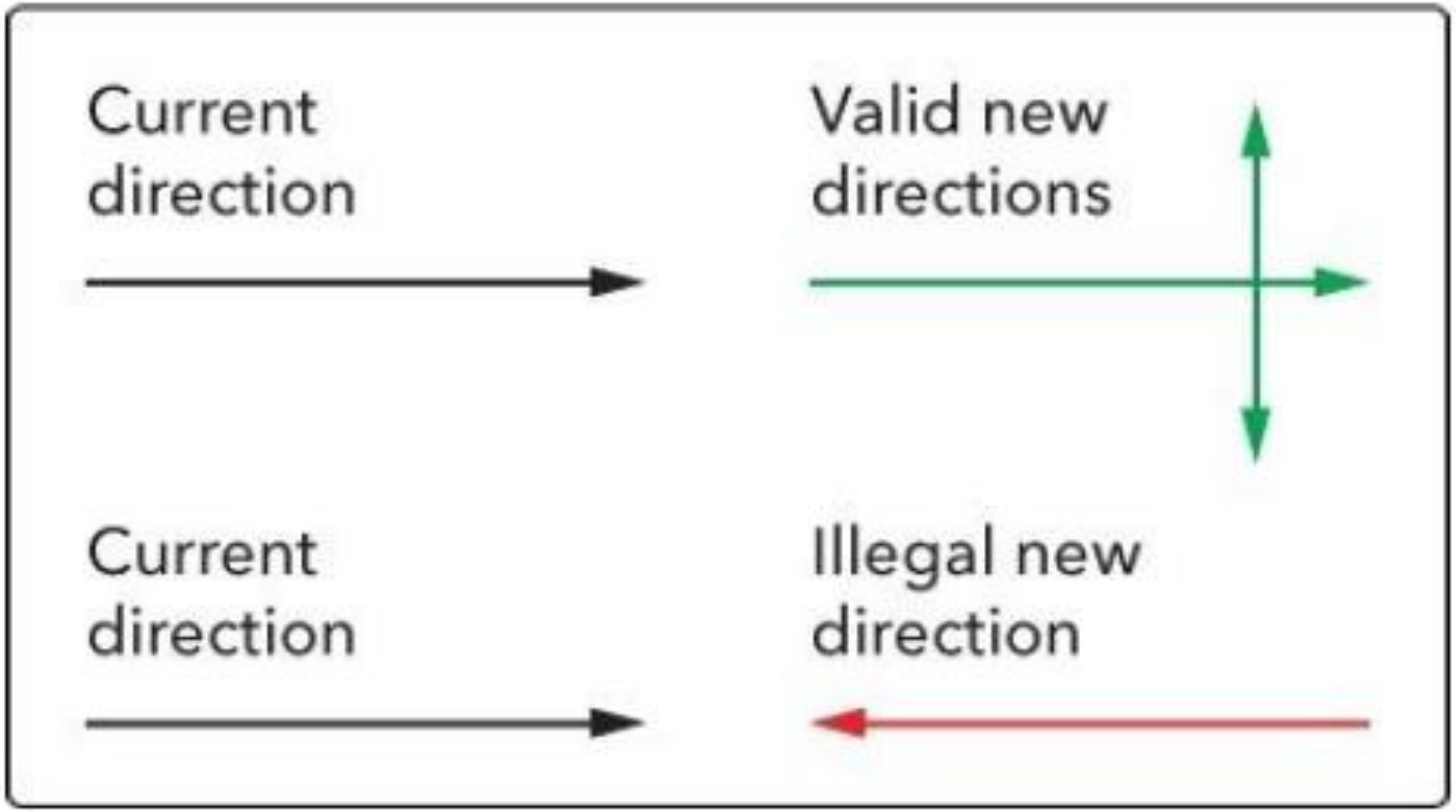

Figure 17-6. Valid new directions based on the current direction

Adding the setDirection Method

The setDirection method checks whether the player is trying to make an illegal turn. If they are, the method uses return to end early; otherwise, it updates the nextDirection property on the snake object.

Here's the code for the setDirection method.

```
Snake.prototype.setDirection = function (newDirection) {
❶  if (this.direction === "up" && newDirection === "down") {
       return;
   } else if (this.direction === "right" && newDirection === "left") {
       return;
   } else if (this.direction === "down" && newDirection === "up") {
       return;
   } else if (this.direction === "left" && newDirection === "right") {
       return;
   }
❷  this.nextDirection = newDirection;
};
```

Adding the setDirection Method

- The if...else statement at ❶ has four parts to deal with the four illegal turns we want to prevent. The first part says that if the snake is moving up (**this.direction** is "up") and the player presses the down arrow (**newDirection** is "down"), we should exit the method early with return. The other parts of the statement deal with the other illegal turns in the same way.
- The **setDirection** method will reach the final line only if **newDirection** is a valid new direction; otherwise, one of the return statements will stop the method.
- If **newDirection** *is* allowed, we set it as the snake's **nextDirection** property, at ❷.

Creating the Apple

LECTURE 5

Creating the Apple

- In this game, we'll represent the apple as an object with three components: a position property, which holds the apple's position as a block object; a draw method, which we'll use to draw the apple; and a move method, which we'll use to give the apple a new position once it's been eaten by the snake.

Writing the Apple Constructor

- The constructor simply sets the apple's position property to a new block object.

```
var Apple = function () {
  this.position = new Block(10, 10);
};
```

- This creates a new block object in column 10, row 10, and assigns it to the apple's position property.
- We'll use this constructor to create an apple object at the beginning of the game.

Drawing the Apple

- We'll use this draw method to draw the apple:

```
Apple.prototype.draw = function () {
    this.position.drawCircle("LimeGreen");
};
```

- The apple's draw method is very simple, as all the hard work is done by the **drawCircle** method (created in Adding the drawCircle Method). To draw the apple, we simply call the **drawCircle** method on the apple's **position** property, passing the color **"LimeGreen"** to tell it to draw a green circle in the given block.

- To test out drawing the apple, run the following code:

```
var apple = new Apple();
apple.draw();
```

Moving the Apple

The move method moves the apple to a random new position within the game area (that is, any block on the canvas other than the border). We'll call this method whenever the snake eats the apple so that the apple reappears in a new location.

```
Apple.prototype.move = function () {
❶  var randomCol = Math.floor(Math.random() *
                    (widthInBlocks - 2)) + 1;
   var randomRow = Math.floor(Math.random() *
                    (heightInBlocks - 2)) + 1;
❷  this.position = new Block(randomCol,randomRow);
};
```

Moving the Apple

- At ❶ we create the variables randomCol and randomRow. These variables will be set to a random column and row value within the playable area. As you saw in Figure 17-1, the columns and rows for the playable area range from 1 to 38, so we need to pick two random numbers in that range.
- To generate these random numbers, we can call Math.floor (Math.random() * 38), which gives us a random number from 0 to 37, and then add 1 to the result to get a number between 1 and 38 (for more about how Math.floor and Math.random work, see Decision Maker).

Moving the Apple

- This is exactly what we do at ❶ to create our random column value, but instead of writing 38, we write **(widthInBlocks - 2).** This means that if we later change the size of the game, we won't also have to change this code. We do the same thing to get a random row value, using **Math.floor(Math.random() * (heightInBlocks - 2)) + 1**.
- Finally, at ❷ we create a new block object with our random column and row values and save this block in **this.position**. This means that the position of the apple will be updated to a new random location somewhere within the playing area.

Moving the Apple

You can test out the move method like this:

```
var apple = new Apple();
    apple.move();
    apple.draw();
```

Putting It All Together

LECTURE 6

Putting It All Together

- Our full code for the game contains almost 200 lines of JavaScript! After we assemble the whole thing, it looks like this.

```html
<!DOCTYPE html>
<html>
<head>
<title>Canvas</title>
    <style>
        canvas{
            border: 3px solid lightblue;
        }
    </style>
    <script src="http://code.jquery.com/jquery-3.5.1.min.js"></script>
</head>
<body>
<canvas id="canvas" width="200" height="200"></canvas>
```

```
<script>
    // Set up canvas
    var canvas = document.getElementById("canvas");
    var ctx = canvas.getContext("2d");
    // Get the width and height from the canvas element
    var width = canvas.width;
    var height = canvas.height;
    // Work out the width and height in blocks
    var blockSize = 10;
    var widthInBlocks = width / blockSize;
    var heightInBlocks = height / blockSize;
    // Set score to 0
    var score = 0;

```

```
// Draw the border
var drawBorder = function () {
    ctx.fillStyle = "Gray";
    ctx.fillRect(0, 0, width, blockSize);
    ctx.fillRect(0, height - blockSize, width, blockSize);
    ctx.fillRect(0, 0, blockSize, height);
    ctx.fillRect(width - blockSize, 0, blockSize, height);
};
// Draw the score in the top-left corner
var drawScore = function () {
    ctx.font = "20px Courier";
    ctx.fillStyle = "Black";
    ctx.textAlign = "left";
    ctx.textBaseline = "top";
    ctx.fillText("Score: " + score, blockSize, blockSize);
};
```

```
    // Clear the interval and display Game Over text
    var gameOver = function () {
        clearInterval(intervalId);
        ctx.font = "28px Courier";
        ctx.fillStyle = "red";
        ctx.textAlign = "center";
        ctx.textBaseline = "middle";
        ctx.fillText("Game Over", width / 2, height / 2);
    };
        // Draw a circle (using the function from Chapter 14)
    var circle = function (x, y, radius, fillCircle) {
        ctx.beginPath();
        ctx.arc(x, y, radius, 0, Math.PI * 2, false);
        if (fillCircle) {
           ctx.fill();
        } else {
            ctx.stroke();
        }
    };

```

```
// The Block constructor
var Block = function (col, row) {
    this.col = col;
    this.row = row;
};

// Draw a square at the block's location
Block.prototype.drawSquare = function (color) {
    var x = this.col * blockSize;
    var y = this.row * blockSize;
    ctx.fillStyle = color;
    ctx.fillRect(x, y, blockSize, blockSize);
};

// Draw a circle at the block's location
Block.prototype.drawCircle = function (color) {
    var centerX = this.col * blockSize + blockSize / 2;
    var centerY = this.row * blockSize + blockSize / 2;
    ctx.fillStyle = color;
    circle(centerX, centerY, blockSize / 2, true);
};

```

```
    // Check if this block is in the same location as another block
    Block.prototype.equal = function (otherBlock) {
        return this.col === otherBlock.col && this.row === otherBlock.row;
    };

    // The Snake constructor
    var Snake = function () {
        this.segments = [
            new Block(7, 5),
            new Block(6, 5),
            new Block(5, 5)
        ];
        this.direction = "right";
        this.nextDirection = "right";
    };

    // Draw a square for each segment of the snake's body
        Snake.prototype.draw = function () {
        for (var i = 0; i < this.segments.length; i++) {
            this.segments[i].drawSquare("Blue");
        }
    };

```

```
// Create a new head and add it to the beginning of
// the snake to move the snake in its current direction
Snake.prototype.move = function () {
    var head = this.segments[0];
    var newHead;
    this.direction = this.nextDirection;
    if (this.direction === "right") {
        newHead = new Block(head.col + 1, head.row);
    } else if (this.direction === "down") {
        newHead = new Block(head.col, head.row + 1);
    } else if (this.direction === "left") {
        newHead = new Block(head.col - 1, head.row);
    } else if (this.direction === "up") {
        newHead = new Block(head.col, head.row - 1);
    }
    if (this.checkCollision(newHead)) {
        gameOver();
        return;
    }
    this.segments.unshift(newHead);
    if (newHead.equal(apple.position)) {
        score++;
        apple.move();
    } else {
        this.segments.pop();
    }
};
```

```
// Check if the snake's new head has collided with the wall or itself
Snake.prototype.checkCollision = function (head) {
var leftCollision = (head.col === 0);
var topCollision = (head.row === 0);
var rightCollision = (head.col === widthInBlocks - 1);
var bottomCollision = (head.row === heightInBlocks - 1);
var wallCollision = leftCollision || topCollision ||
rightCollision || bottomCollision;
var selfCollision = false;
for (var i = 0; i < this.segments.length; i++) {
if (head.equal(this.segments[i])) {
selfCollision = true;
}
    }
return wallCollision || selfCollision;
};
```

```
// Set the snake's next direction based on the keyboard
Snake.prototype.setDirection = function (newDirection) {
    if (this.direction === "up" && newDirection === "down") {
        return;
    } else if (this.direction === "right" && newDirection === "left") {
        return;
    } else if (this.direction === "down" && newDirection === "up") {
        return;
    } else if (this.direction === "left" && newDirection === "right") {
        return;
    }
    this.nextDirection = newDirection;
};

// The Apple constructor
var Apple = function () {
    this.position = new Block(10, 10);
};

// Draw a circle at the apple's location
Apple.prototype.draw = function () {
    this.position.drawCircle("LimeGreen");
};
```

```
// Move the apple to a new random location
Apple.prototype.move = function () {
    var randomCol = Math.floor(Math.random()*(widthInBlocks - 2)) + 1;
    var randomRow = Math.floor(Math.random()*(heightInBlocks - 2)) + 1;
    this.position = new Block(randomCol, randomRow);
};

// Create the snake and apple objects
var snake = new Snake();
var apple = new Apple();
// Pass an animation function to setInterval
var intervalId = setInterval(function () {
    ctx.clearRect(0, 0, width, height);
    drawScore();
    snake.move();
    snake.draw();
    apple.draw();
    drawBorder();
}, 100);
```

```
    // Convert keycodes to directions
    var directions = {
        37: "left",
        38: "up",
        39: "right",
        40: "down"
    };
    // The keydown handler for handling direction key presses
    $("body").keydown(function (event) {
        var newDirection = directions[event.keyCode];
        if (newDirection !== undefined) {
        snake.setDirection(newDirection);
        }
    });
</script>
</body>
</html>
```

Putting It All Together

- This code is made up of a number of sections. The first section, at ❶, is where all the variables for the game are set up, including the canvas, context, width, and height (we looked at these in Chapter 16).
- Next, at ❷, come all the individual functions: **drawBorder**, **drawScore**, **gameOver**, and **circle**.
- At ❸ comes the code for the Block constructor, followed by its **drawSquare**, **drawCircle**, and **equal** methods. Then, at ❹, we have the Snake constructor and all of its methods. After that, at ❺, is the Apple constructor and its **draw** and **move** methods.

Putting It All Together

- Finally, at ❻, you can see the code that starts the game and keeps it running. First we create the snake and apple objects. Then we use **setInterval** to get the game animation going. Notice that when we call **setInterval**, we save the interval ID in the variable **intervalId** so we can cancel it later in the **gameOver** function.
- The function passed to **setInterval** is called for every step of the game. It is responsible for drawing everything on the canvas and for updating the state of the game. It clears the canvas and then draws the score, the snake, the apple, and the border. It also calls the move method on the snake, which, as you saw earlier, moves the snake one step in its current direction. After the call to **setInterval**, at ❼, we end with the code for listening to keyboard events and setting the snake's direction.

Putting It All Together

- As always, you'll need to type all this code inside the script element in your HTML document. To play the game, just load snake.html in your browser and use the arrows to control the snake's direction. If the arrow keys don't work, you might need to click inside the browser window to make sure it can pick up the key events.
- If the game doesn't work, there might be an error in your JavaScript. Any error will be output in the console, so look there for any helpful messages. If you can't determine why things aren't working, check each line carefully against the preceding listing.
- Now that you have the game running, what do you think? How high a score can you get?

Summary

LECTURE 7

Summary

- In this chapter, we made a full game using the canvas element. This game combines many of the data types, concepts, and techniques you learned throughout this book: numbers, strings, Booleans, arrays, objects, control structures, functions, object-oriented programming, event handlers, setInterval, and drawing with canvas.
- Now that you've programmed this Snake game, there are lots of other simple two-dimensional games that you could write using JavaScript. You could make your own version of classic games like Breakout, Asteroids, Space Invaders, or Tetris. Or you could make up your own game!

Summary

- Of course, you can use JavaScript for programs besides games. Now that you've used JavaScript to do some complicated math, you could use it to help with your math homework. Or maybe you want to create a website to show off your programming skills to the world. The possibilities are endless!