# CS 50 Web Design

## APCSP Module 2: Internet

## Unit 4: Web Graphics Design

LECTURE 15: MULTIMEDIA COMPONENT USING CANVAS

DR. ERIC CHOU IEEE SENIOR MEMBER

## Objectives

- Integrate a project with audio, image, video, animation and other multimedia contents on Canvas
- Motion Bee Project
- Keyboard control on canvas

# Introduction

SECTION 1

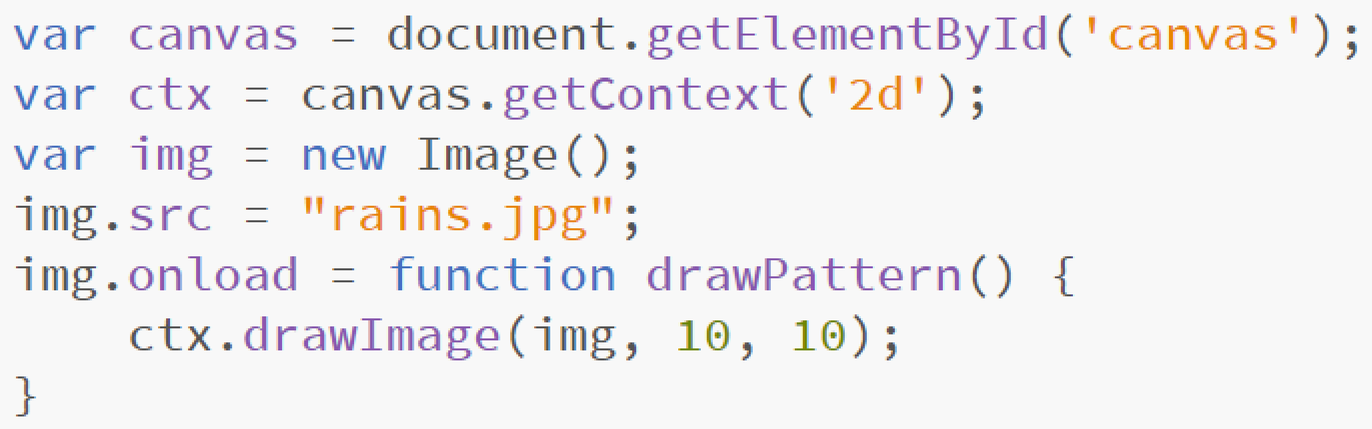
## Images

ACTIVITY

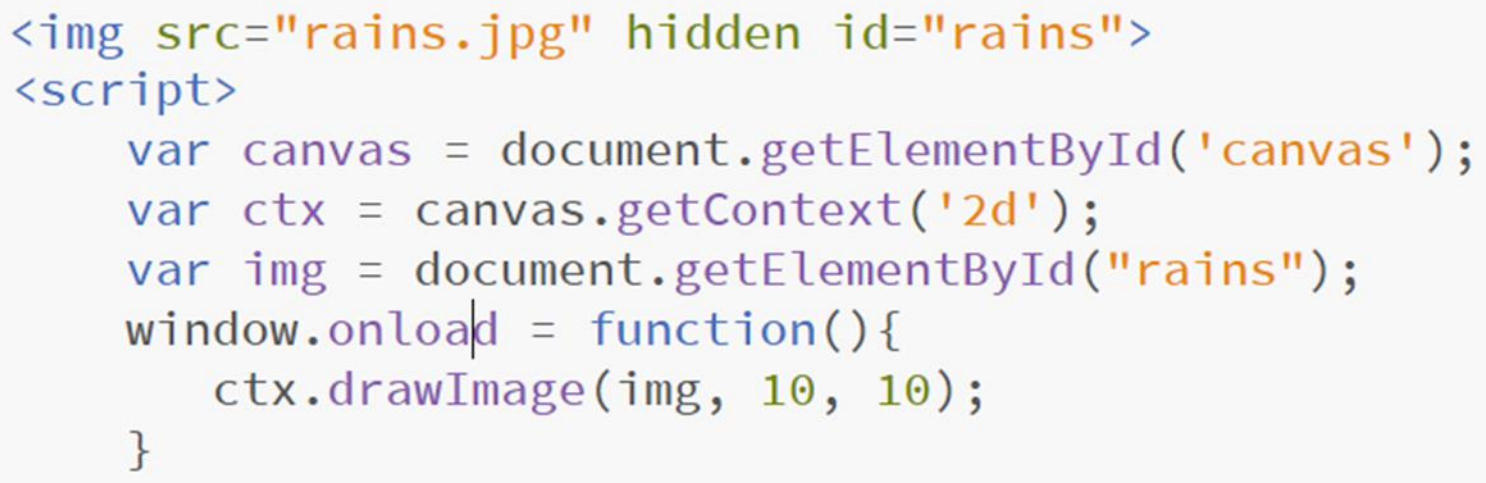
## Image

- Image on Canvas is of completely different nature than the image on HTML node.
- The img tag element define an image node on a HTML page.
- The image on canvas is a colorful dot-matrix representation of an image. So, it can be overlapped, cleared or reloaded. It is not an element on HTML page.

## Two ways of loading images



```
var canvas = document.getElementById('canvas');
var ctx = canvas.getContext('2d');
var img = new Image();
img.src = "rains.jpg";
img.onload = function drawPattern() {
    ctx.drawImage(img, 10, 10);
}
```



```
<img src="rains.jpg" hidden id="rains">
<script>
    var canvas = document.getElementById('canvas');
    var ctx = canvas.getContext('2d');
    var img = document.getElementById("rains");
    window.onload = function(){
       ctx.drawImage(img, 10, 10);
    }
```

## IMAGES

**CANVAS API CAN USE ANY OF THE FOLLOWING DATA TYPES**

HTMLImageElement
HTMLVideoElement
HTMLCanvasElement

**GET AN IMAGE**

**from the same page:**
**from other domain:**
**use another canvas element:**
**Create images from scratch**
**Embedding an image via** data: url.
**Using frames from a video <video></video>**

**DRAW AN IMAGE:**

drawImage(image, x, y)

**SCALE AN IMAGE:**

drawImage(image, x, y, width, height)

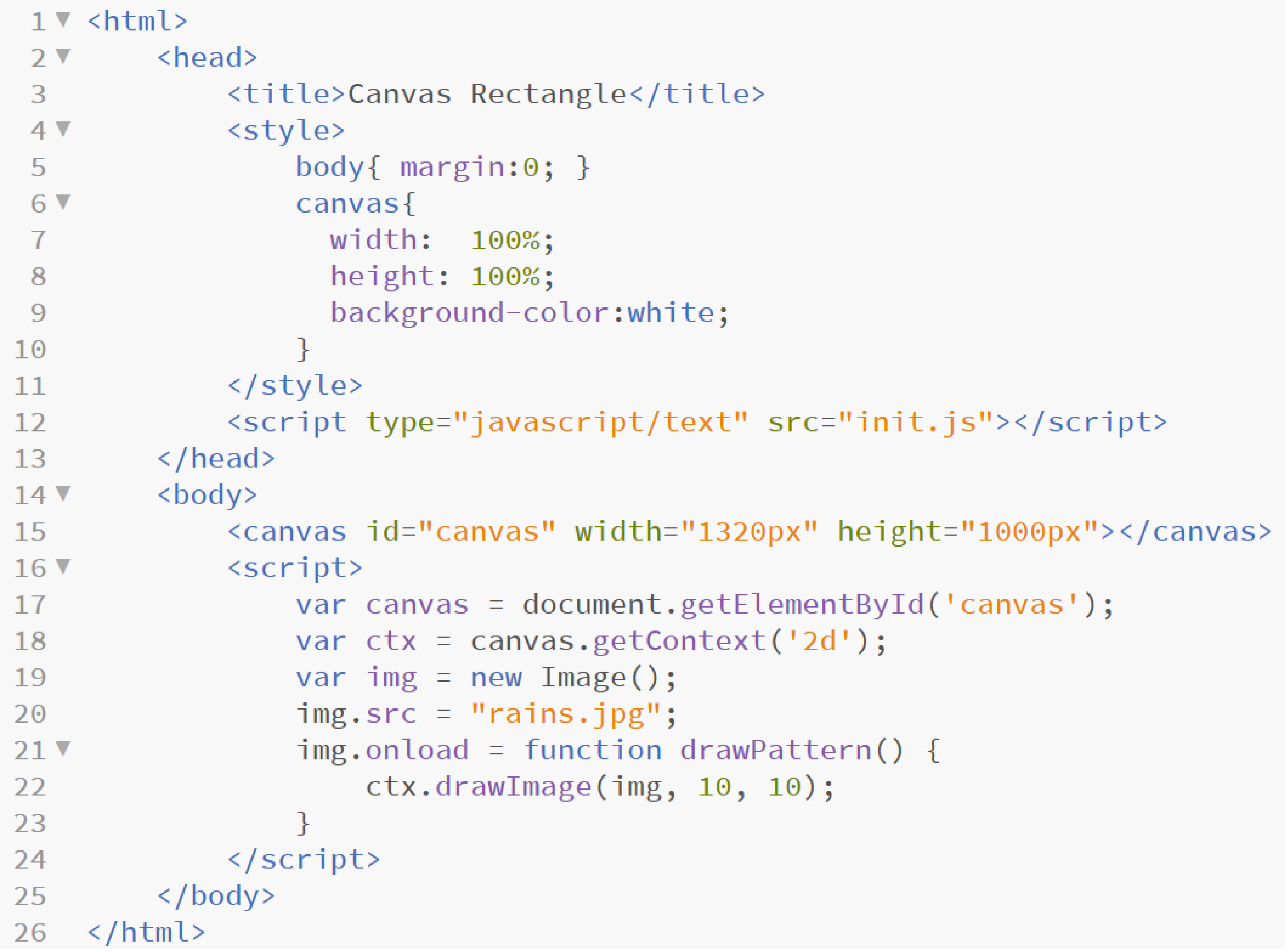
**SLICE AN IMAGE:**

drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)

**CONTROL IMAGE SCALING BEHAVIOR**

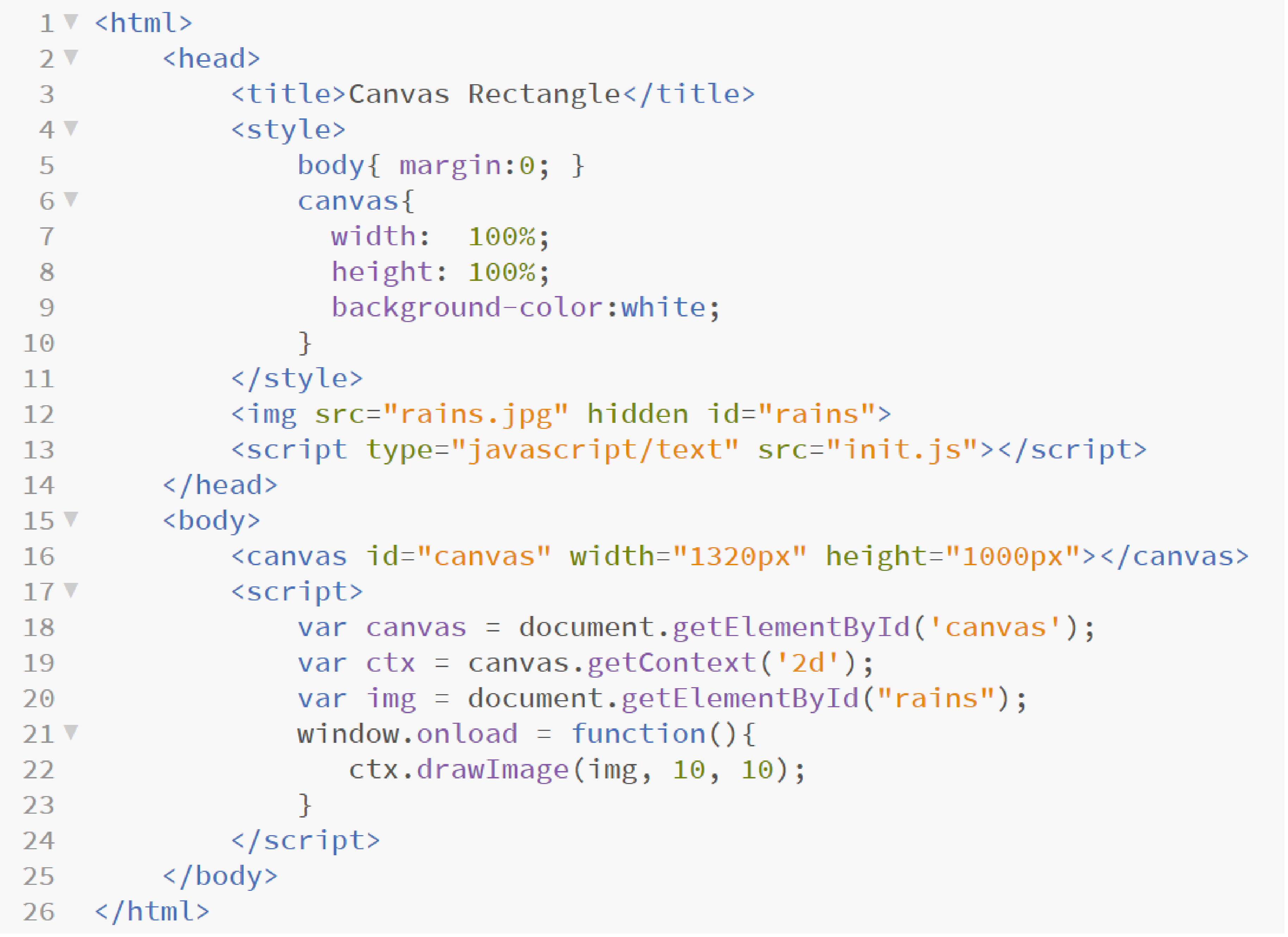
```
ctx.mozImageSmoothingEnabled = false;
ctx.webkitImageSmoothingEnabled = false;
ctx.msImageSmoothingEnabled = false;
ctx.imageSmoothingEnabled = false;
```

## image1.html

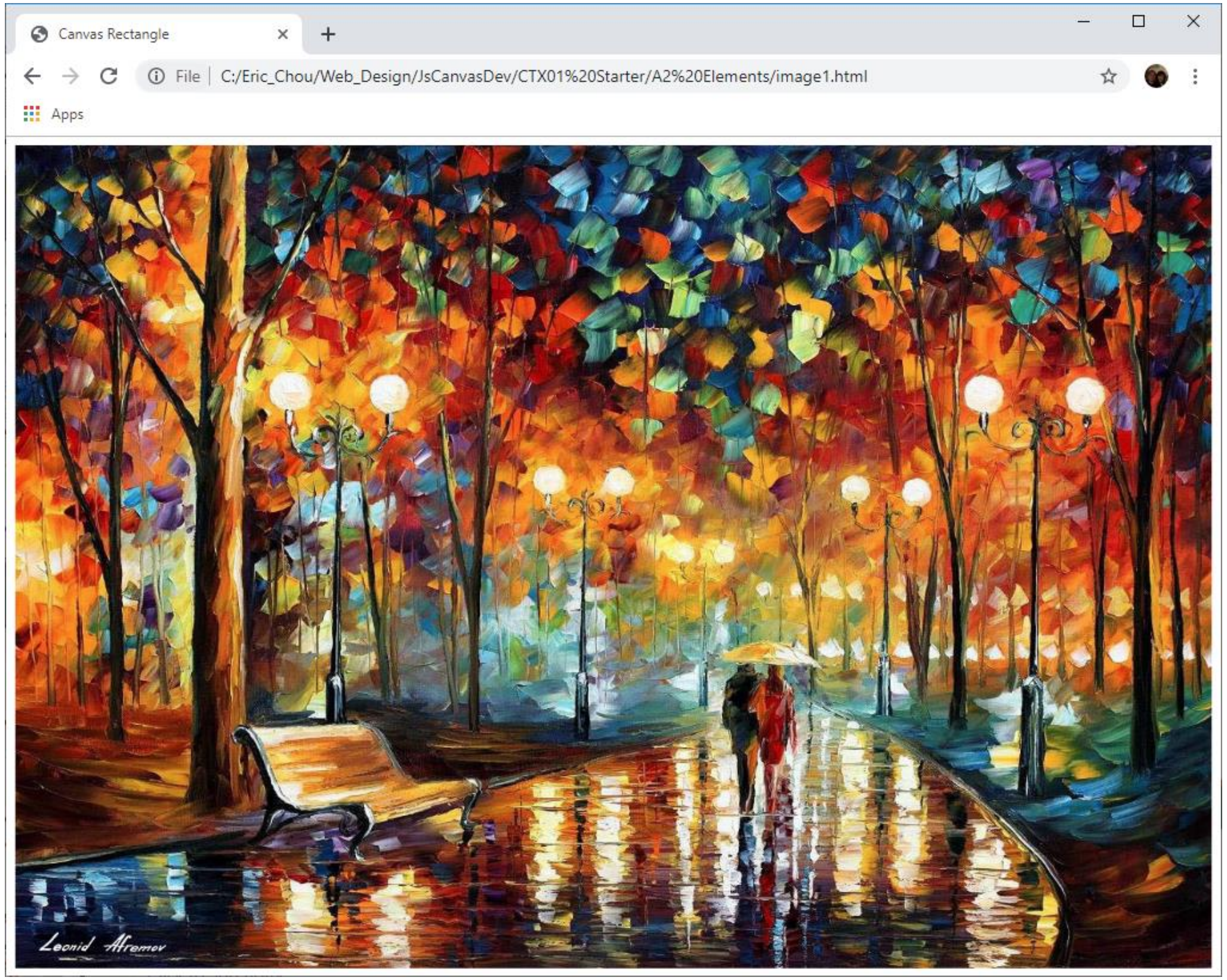


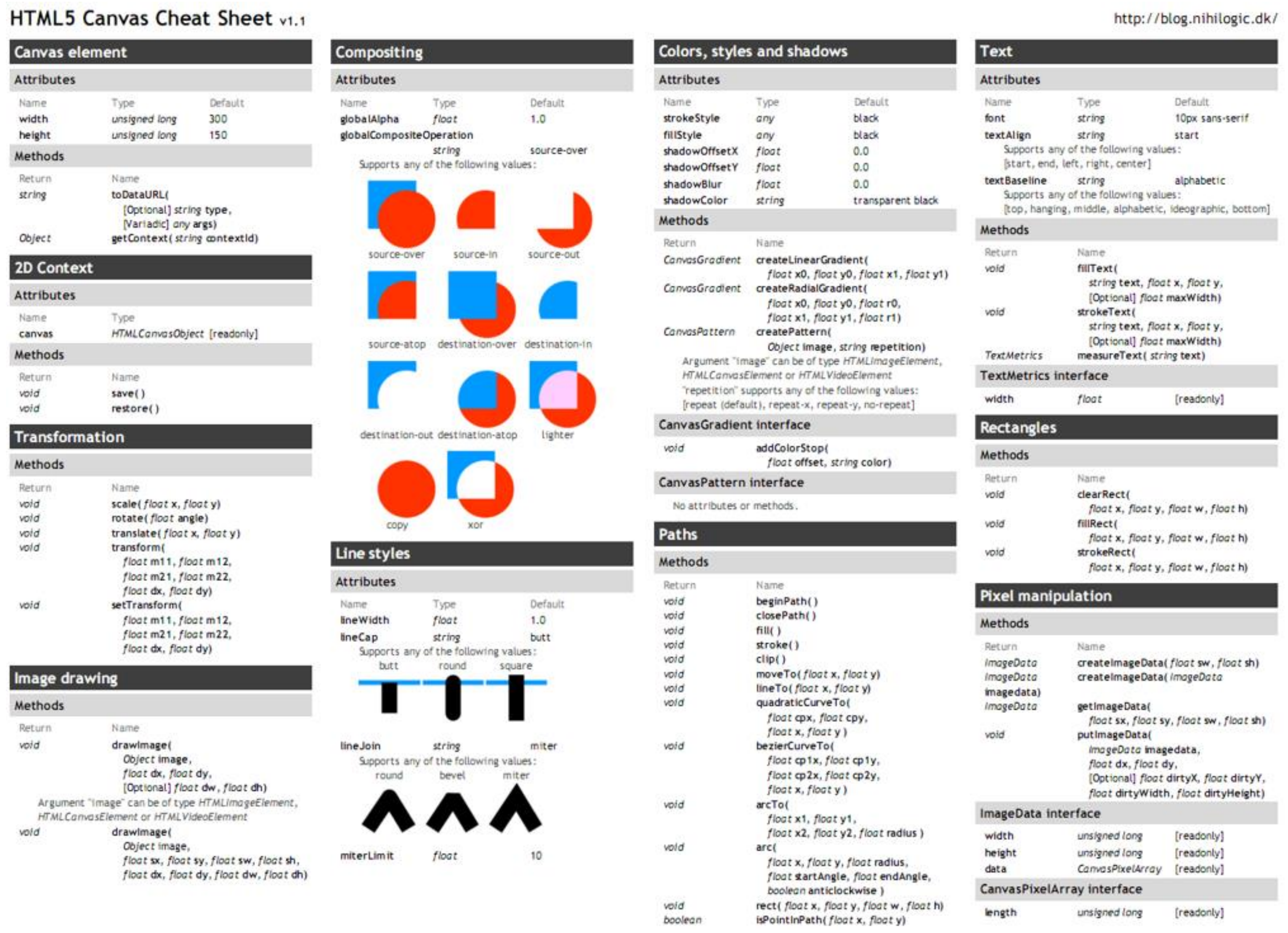
```html
<html>
    <head>
        <title>Canvas Rectangle</title>
        <style>
            body{ margin:0; }
            canvas{
              width:  100%;
              height: 100%;
              background-color:white;
            }
        </style>
        <script type="javascript/text" src="init.js"></script>
    </head>
    <body>
        <canvas id="canvas" width="1320px" height="1000px"></canvas>
        <script>
            var canvas = document.getElementById('canvas');
            var ctx = canvas.getContext('2d');
            var img = new Image();
            img.src = "rains.jpg";
            img.onload = function drawPattern() {
                ctx.drawImage(img, 10, 10);
            }
        </script>
    </body>
</html>
```

## image2.html



```html
<html>
    <head>
        <title>Canvas Rectangle</title>
        <style>
            body{ margin:0; }
            canvas{
              width:  100%;
              height: 100%;
              background-color:white;
            }
        </style>
        <img src="rains.jpg" hidden id="rains">
        <script type="javascript/text" src="init.js"></script>
    </head>
    <body>
        <canvas id="canvas" width="1320px" height="1000px"></canvas>
        <script>
            var canvas = document.getElementById('canvas');
            var ctx = canvas.getContext('2d');
            var img = document.getElementById("rains");
            window.onload = function(){
               ctx.drawImage(img, 10, 10);
            }
        </script>
    </body>
</html>
```


Canvas Rectangle
File
C:/Eric_Chou/Web_Design/JsCanvasDev/CTX01%20Starter/A2%20Elements/image1.html
Apps
Leonid Afremov



# HTML5 Canvas Cheat Sheet v1.1

http://blog.nihilogic.dk/

## Canvas element

### Attributes

| Name | Type | Default |
|---|---|---|
| width | *unsigned long* | 300 |
| height | *unsigned long* | 150 |

### Methods

| Return | Name |
|---|---|
| *string* | toDataURL( [Optional] *string* type, [Variadic] *any* args) |
| *Object* | getContext( *string* contextId) |

## 2D Context

### Attributes

| Name | Type |
|---|---|
| canvas | *HTMLCanvasObject* [readonly] |

### Methods

| Return | Name |
|---|---|
| *void* | save( ) |
| *void* | restore( ) |

## Transformation

### Methods

| Return | Name |
|---|---|
| *void* | scale( *float* x, *float* y) |
| *void* | rotate( *float* angle) |
| *void* | translate( *float* x, *float* y) |
| *void* | transform( *float* m11, *float* m12, *float* m21, *float* m22, *float* dx, *float* dy) |
| *void* | setTransform( *float* m11, *float* m12, *float* m21, *float* m22, *float* dx, *float* dy) |

## Image drawing

### Methods

| Return | Name |
|---|---|
| *void* | drawImage( *Object* image, *float* dx, *float* dy, [Optional] *float* dw, *float* dh) |
| *void* | drawImage( *Object* image, *float* sx, *float* sy, *float* sw, *float* sh, *float* dx, *float* dy, *float* dw, *float* dh) |

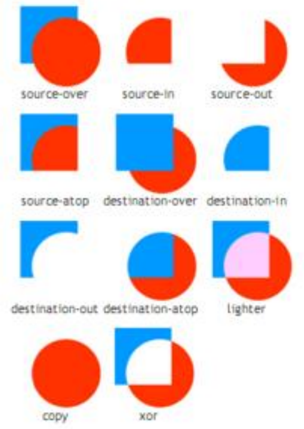
Argument "image" can be of type *HTMLImageElement*, *HTMLCanvasElement* or *HTMLVideoElement* (applies to the first drawImage)

## Compositing

### Attributes

| Name | Type | Default |
|---|---|---|
| globalAlpha | *float* | 1.0 |
| globalCompositeOperation | *string* | source-over |

Supports any of the following values:

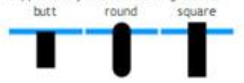


## Line styles

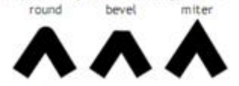
### Attributes

| Name | Type | Default |
|---|---|---|
| lineWidth | *float* | 1.0 |
| lineCap | *string* | butt |
| lineJoin | *string* | miter |
| miterLimit | *float* | 10 |

lineCap supports any of the following values:



lineJoin supports any of the following values:



## Colors, styles and shadows

### Attributes

| Name | Type | Default |
|---|---|---|
| strokeStyle | *any* | black |
| fillStyle | *any* | black |
| shadowOffsetX | *float* | 0.0 |
| shadowOffsetY | *float* | 0.0 |
| shadowBlur | *float* | 0.0 |
| shadowColor | *string* | transparent black |

### Methods

| Return | Name |
|---|---|
| *CanvasGradient* | createLinearGradient( *float* x0, *float* y0, *float* x1, *float* y1) |
| *CanvasGradient* | createRadialGradient( *float* x0, *float* y0, *float* r0, *float* x1, *float* y1, *float* r1) |
| *CanvasPattern* | createPattern( *Object* image, *string* repetition) |

Argument "image" can be of type *HTMLImageElement*, *HTMLCanvasElement* or *HTMLVideoElement*
"repetition" supports any of the following values:
[repeat (default), repeat-x, repeat-y, no-repeat]

### CanvasGradient interface

| Return | Name |
|---|---|
| *void* | addColorStop( *float* offset, *string* color) |

### CanvasPattern interface

No attributes or methods.

## Paths

### Methods

| Return | Name |
|---|---|
| *void* | beginPath( ) |
| *void* | closePath( ) |
| *void* | fill( ) |
| *void* | stroke( ) |
| *void* | clip( ) |
| *void* | moveTo( *float* x, *float* y) |
| *void* | lineTo( *float* x, *float* y) |
| *void* | quadraticCurveTo( *float* cpx, *float* cpy, *float* x, *float* y ) |
| *void* | bezierCurveTo( *float* cp1x, *float* cp1y, *float* cp2x, *float* cp2y, *float* x, *float* y ) |
| *void* | arcTo( *float* x1, *float* y1, *float* x2, *float* y2, *float* radius ) |
| *void* | arc( *float* x, *float* y, *float* radius, *float* startAngle, *float* endAngle, *boolean* anticlockwise ) |
| *void* | rect( *float* x, *float* y, *float* w , *float* h) |
| *boolean* | isPointInPath( *float* x, *float* y) |

## Text

### Attributes

| Name | Type | Default |
|---|---|---|
| font | *string* | 10px sans-serif |
| textAlign | *string* | start |
| textBaseline | *string* | alphabetic |

textAlign supports any of the following values:
[start, end, left, right, center]

textBaseline supports any of the following values:
[top, hanging, middle, alphabetic, ideographic, bottom]

### Methods

| Return | Name |
|---|---|
| *void* | fillText( *string* text, *float* x, *float* y, [Optional] *float* maxWidth) |
| *void* | strokeText( *string* text, *float* x, *float* y, [Optional] *float* maxWidth) |
| *TextMetrics* | measureText( *string* text) |

### TextMetrics interface

| | | |
|---|---|---|
| width | *float* | [readonly] |

## Rectangles

### Methods

| Return | Name |
|---|---|
| *void* | clearRect( *float* x, *float* y, *float* w , *float* h) |
| *void* | fillRect( *float* x, *float* y, *float* w , *float* h) |
| *void* | strokeRect( *float* x, *float* y, *float* w , *float* h) |

## Pixel manipulation

### Methods

| Return | Name |
|---|---|
| *ImageData* | createImageData( *float* sw , *float* sh) |
| *ImageData* | createImageData( *ImageData* imagedata) |
| *ImageData* | getImageData( *float* sx, *float* sy, *float* sw , *float* sh) |
| *void* | putImageData( *ImageData* imagedata, *float* dx, *float* dy, [Optional] *float* dirtyX, *float* dirtyY, *float* dirtyWidth, *float* dirtyHeight) |

### ImageData interface

| | | |
|---|---|---|
| width | *unsigned long* | [readonly] |
| height | *unsigned long* | [readonly] |
| data | *CanvasPixelArray* | [readonly] |

### CanvasPixelArray interface

| | | |
|---|---|---|
| length | *unsigned long* | [readonly] |

# Canvas, Audio and Video

SECTION 2

Audio/Video/Graphics/Animation

This course includes:

- HTML5 Canvas
- Timing control for script-based animations
- HTML5 video and audio

## Audio

## <audio> tag and Audio() object

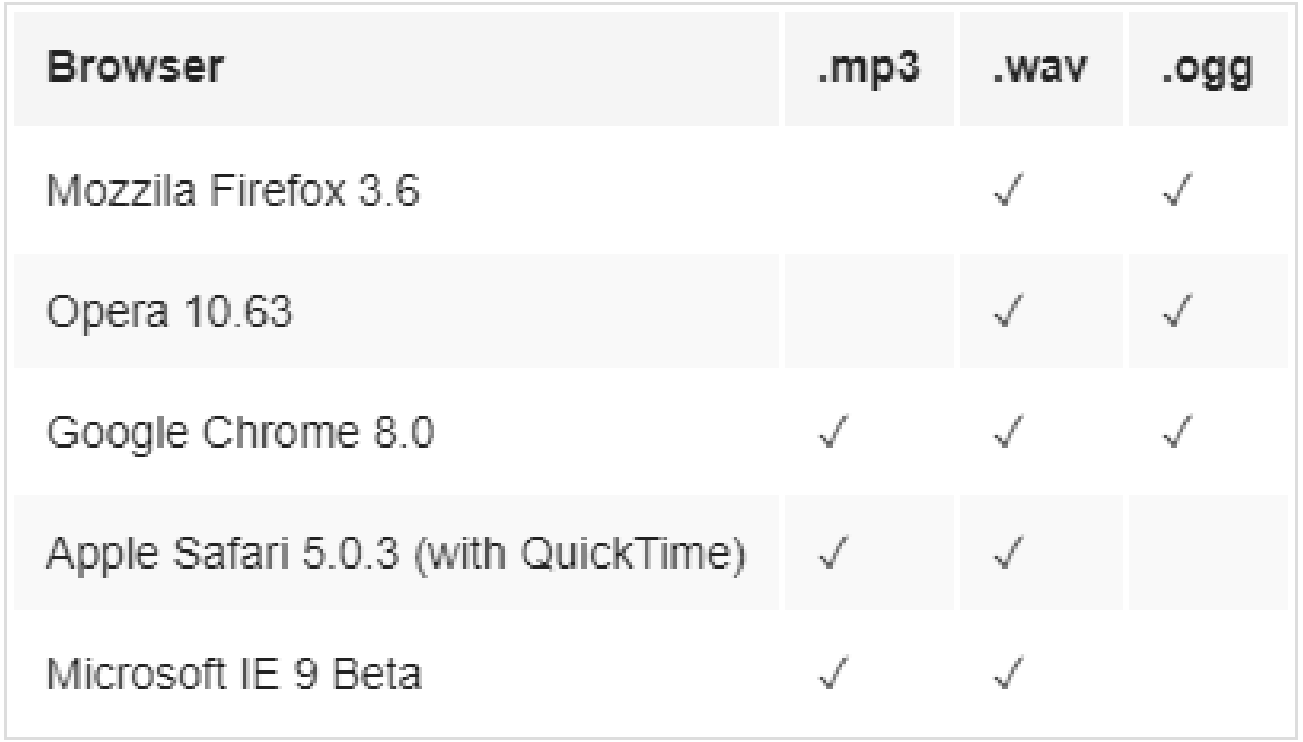
- Let's have something fun to start with. We are now talking about web browser supporting audio file in native, just like how <img> tag is supported since 1994. HTML5 is likely to put an end to audio plug-in such as Microsoft Windows Media player, Microsoft Silverlight, Apple QuickTime and the infamous Adobe Flash.

## How to?

- In order to make your web page plays music, the html code can be as simple as

```
<audio src="vincent.mp3"
controls> </audio>
```

- Unfortunately, the most popular audio format MPEG3(.mp3) is not an Open standard, it is patent encumbered. That means, web browser needs to pay a sum of money in order to decode it, and that might not be financially feasible for smaller company or organization. As you can see from table below, only those big boys are rich enough to decode MP3. Firefox and Opera supports only Vorbis (.ogg) format which is an Open standard.
- On the other hand, the open standard Vorbis (*.ogg) is not supported by Safari and IE9. Hence, it is always good to have both Mp3 and Ogg side to side available.



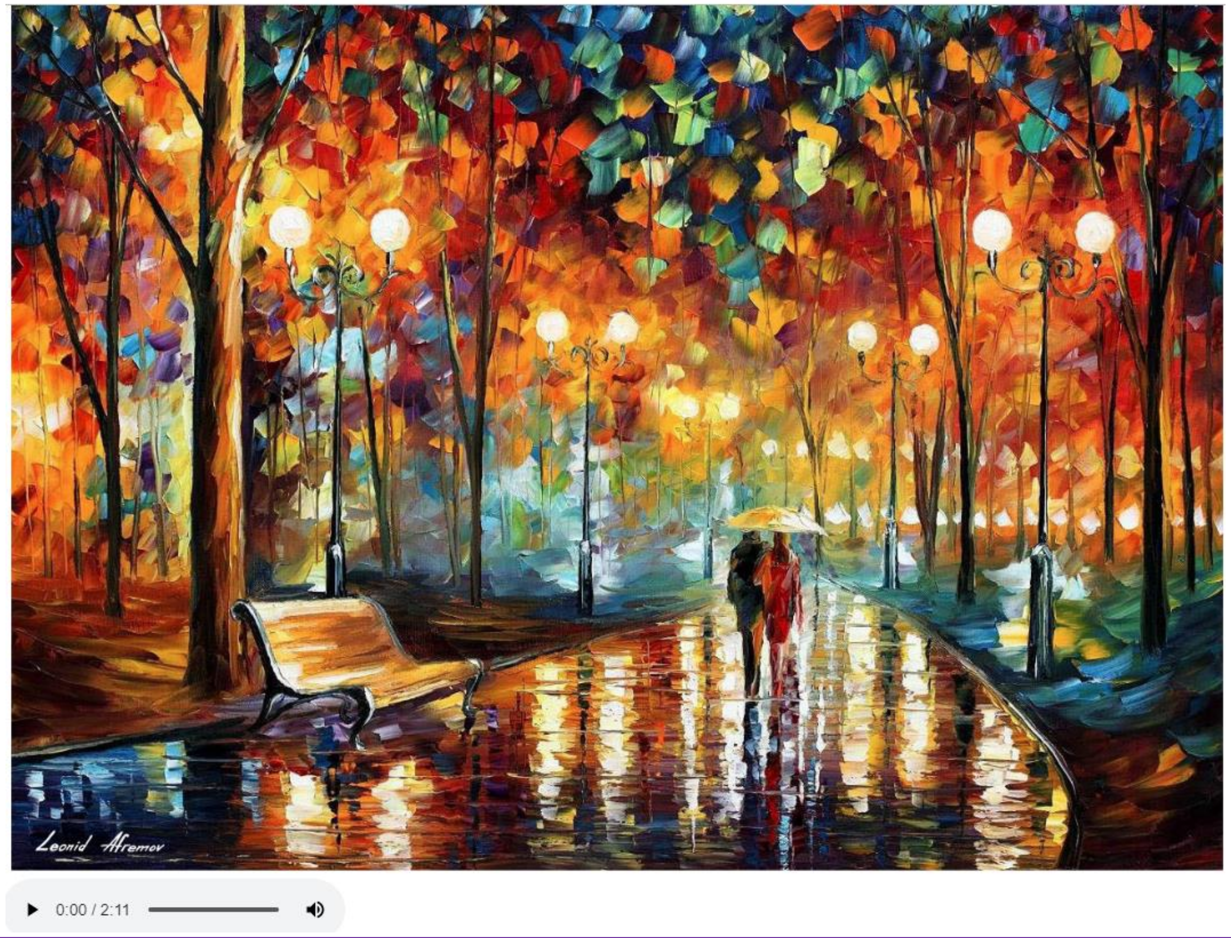
| Browser | .mp3 | .wav | .ogg |
|---|---|---|---|
| Mozzila Firefox 3.6 | | ✓ | ✓ |
| Opera 10.63 | | ✓ | ✓ |
| Google Chrome 8.0 | ✓ | ✓ | ✓ |
| Apple Safari 5.0.3 (with QuickTime) | ✓ | ✓ | |
| Microsoft IE 9 Beta | ✓ | ✓ | |

## Browser Support

## Demo Program: audio1.html

- Control Panel

```
<div>
<audio controls src="Raindrops.mp3"></audio>
</div>
```



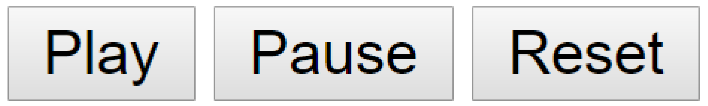
## Demo Program: audio2.html

- Control the music play, pause, and reload by events.
- Hide the control panel.
- Write event-handlers for each audio playback functions.

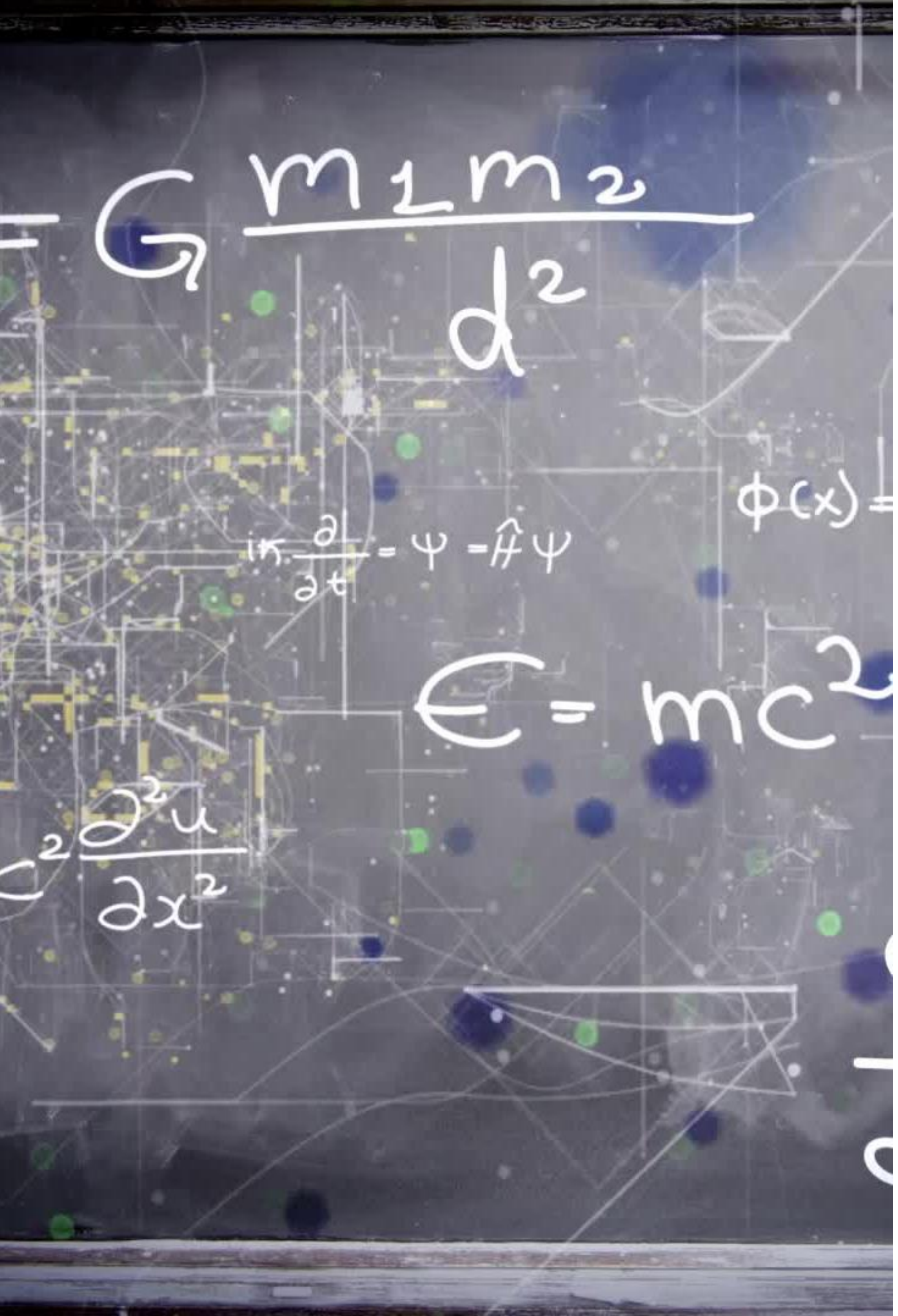
audio2.html

```
<html>
    <head>
        <audio controls hidden id="music">
        <source src="Raindrops.mp3" type="audio/mpeg">
        Your browser does not support the audio element.
        </audio>
        <script>
            var myMusic= document.getElementById("music");
            function play() {
            myMusic.play();
            }

            function pause() {
            myMusic.pause();
            }
            function load() {
            myMusic.load();
            }
            function ap(){
                myMusic.play();
            }
            window.onload = ap;
    </script>
    </head>
    <body onload="ap();">
    <button onclick="play()" type="button">Play </button>
    <button onclick="pause()" type="button">Pause</button>
    <button onclick="load()" type="button">Reset</button>
    </body>
</html>
```

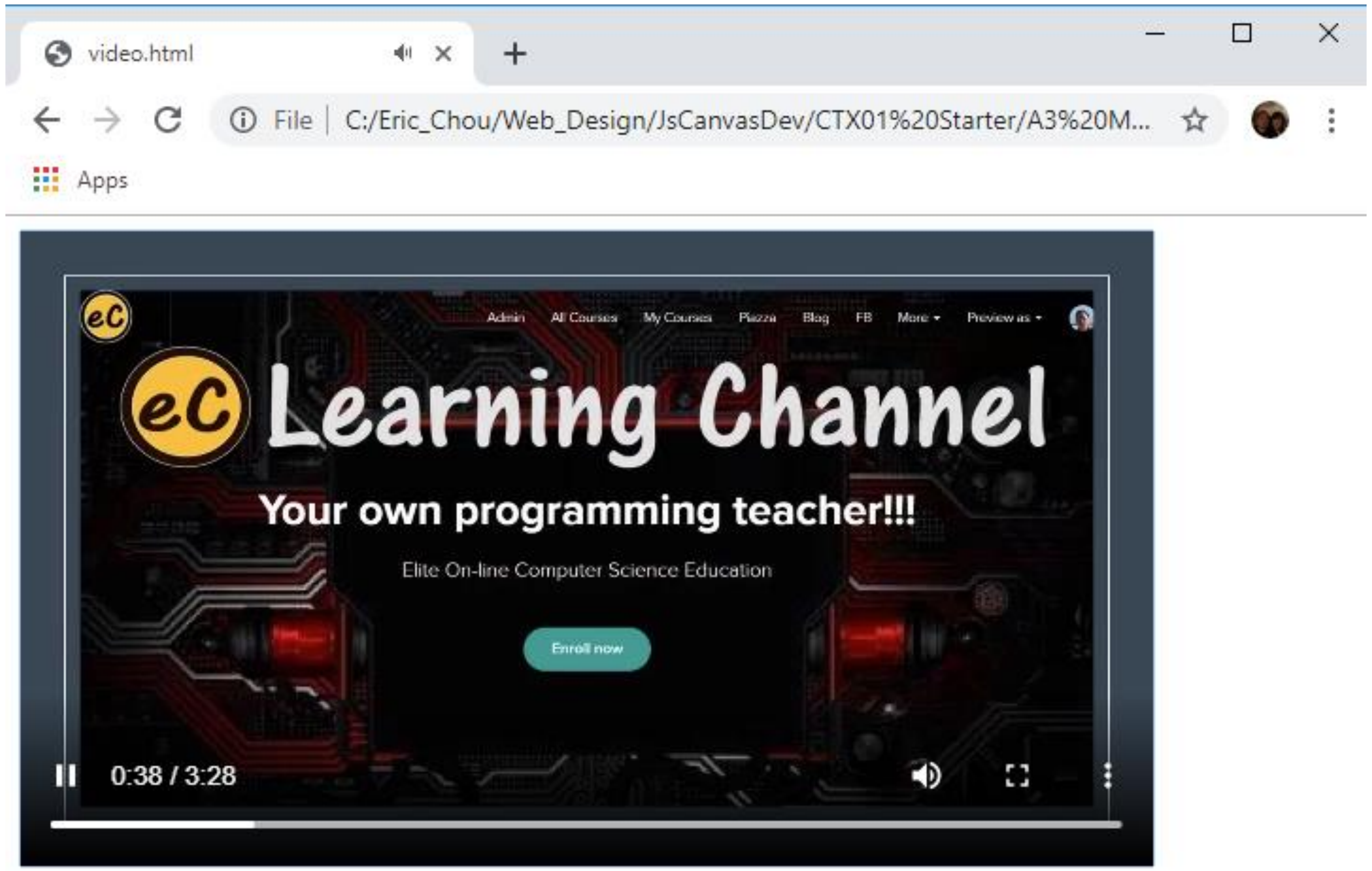


## Video



## Video

```
<html>
  <body>
    <video controls src="video.mp4">
    </video>
  </body>
</html>
```


video.html
File | C:/Eric_Chou/Web_Design/JsCanvasDev/CTX01%20Starter/A3%20M...
Apps
Admin
All Courses
My Courses
Piazza
Blog
FB
More
Preview as
Learning Channel
Your own programming teacher!!!
Elite On-line Computer Science Education
Enroll now
0:38 / 3:28

# Video

VIDEO.MP4

## Animation

## Simple Game Loop - setInterval

- setInterval(**Time_Up_Event_Handler**,

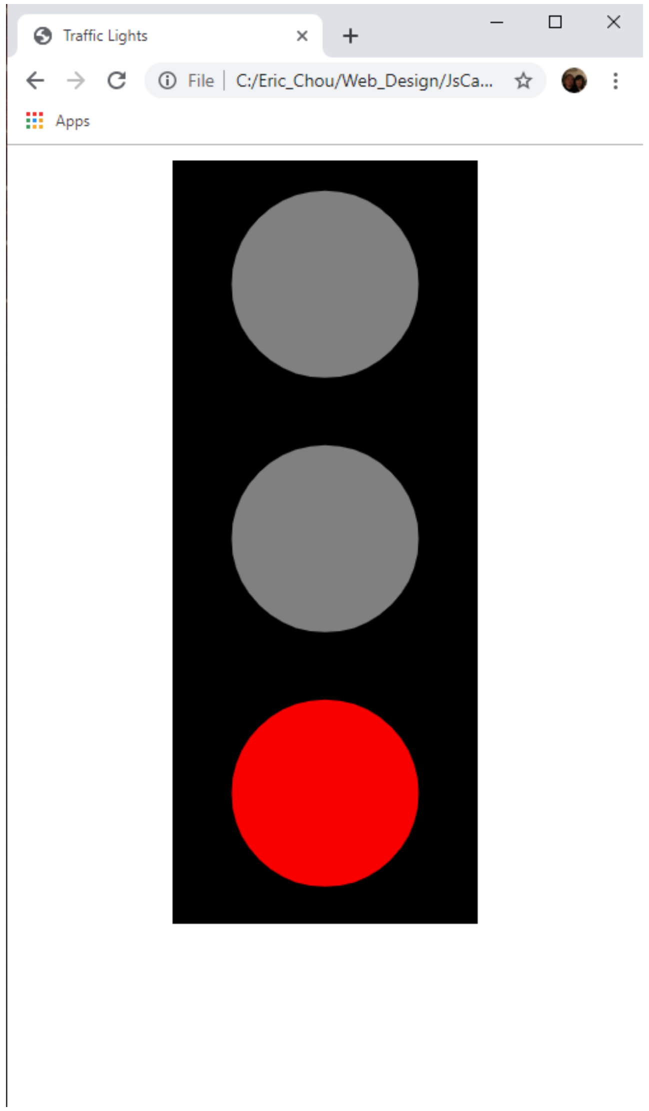
  **Time_in_milli_secs**)
- Time_Up_Event_Handler should be a function.
- Time_in_milli_secs should be an integer.

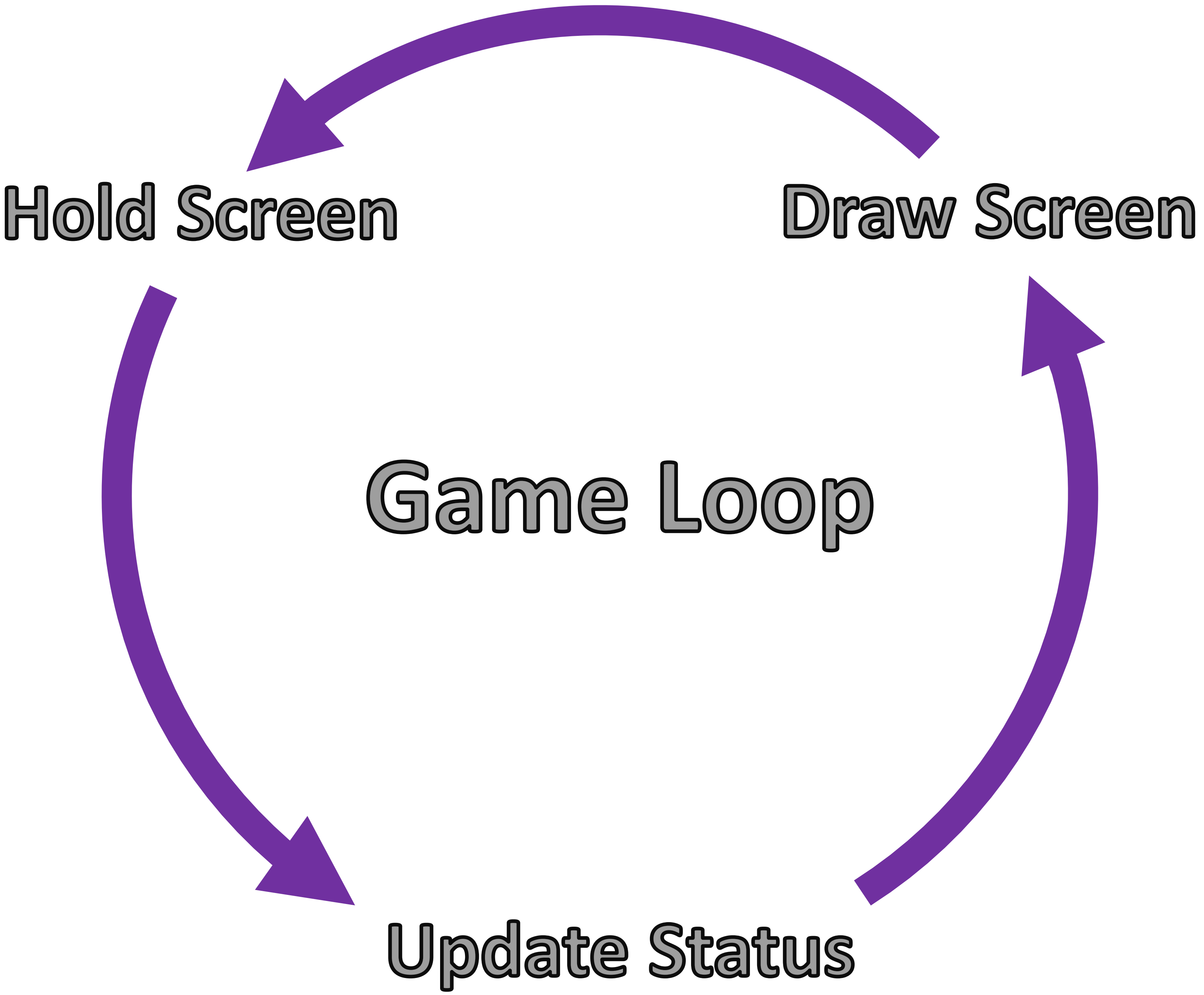
## Demonstration Program

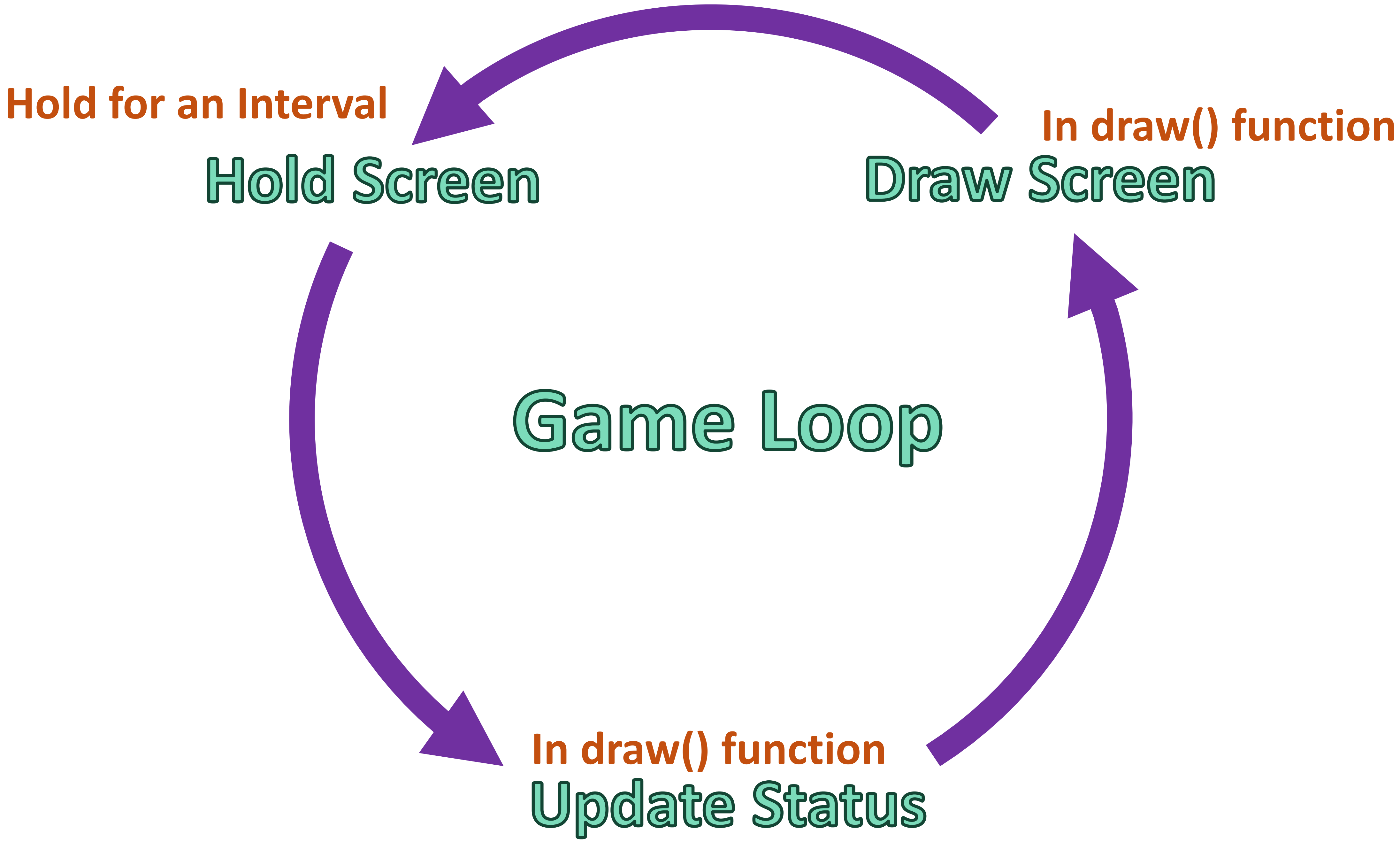
ANIME1.HTML + ANIME1.JS

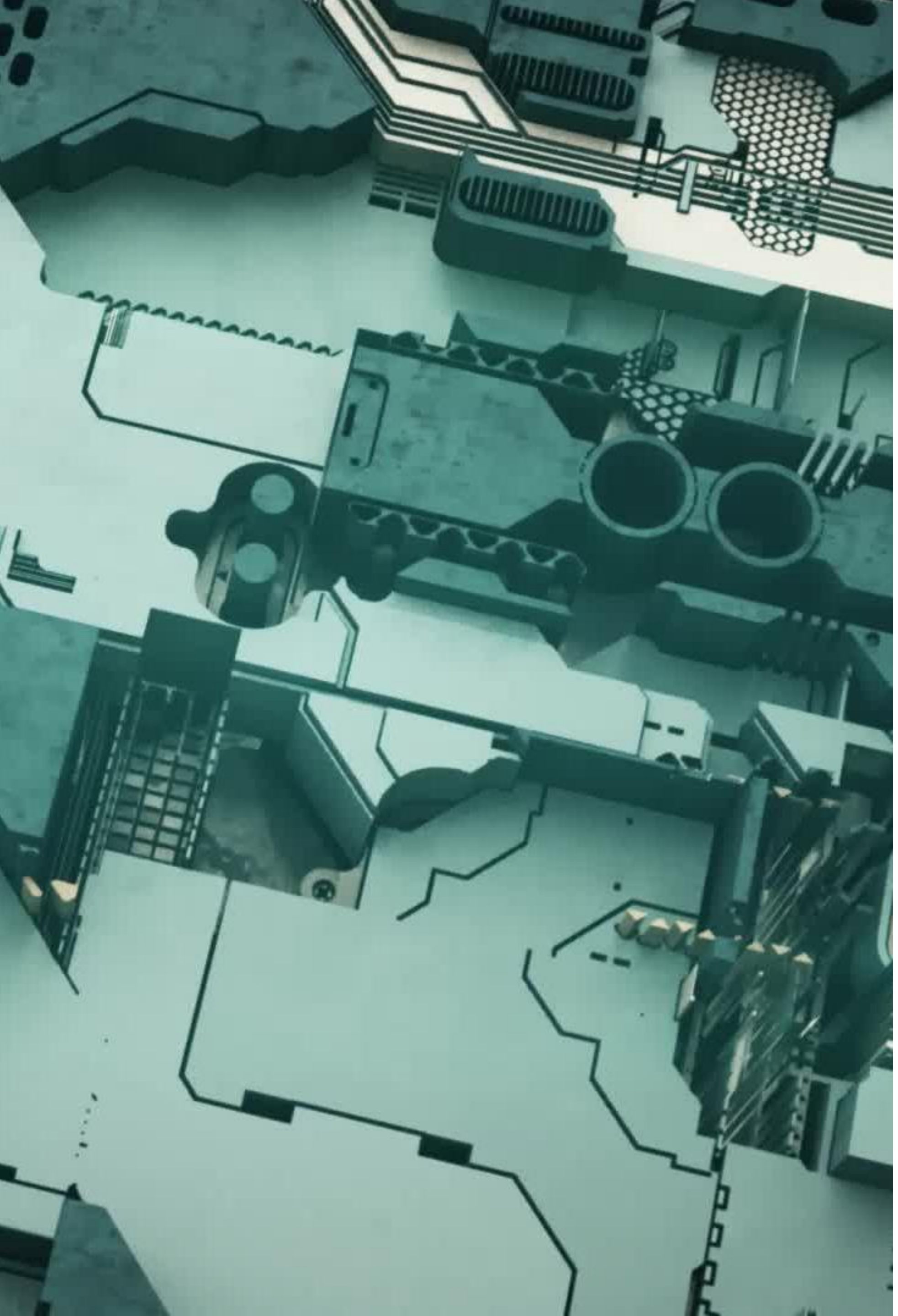
## Demo Program: anime1.html

- Simple 3-way no exit game loop.
- Draw Screen and Update Status in one handler function.
- Interval defined in setInterval();


Traffic Lights
File
C:/Eric_Chou/Web_Design/JsCa...
Apps


Hold Screen
Draw Screen
Game Loop
Update Status


Hold for an Interval
Hold Screen
In draw() function
Draw Screen
Game Loop
In draw() function
Update Status

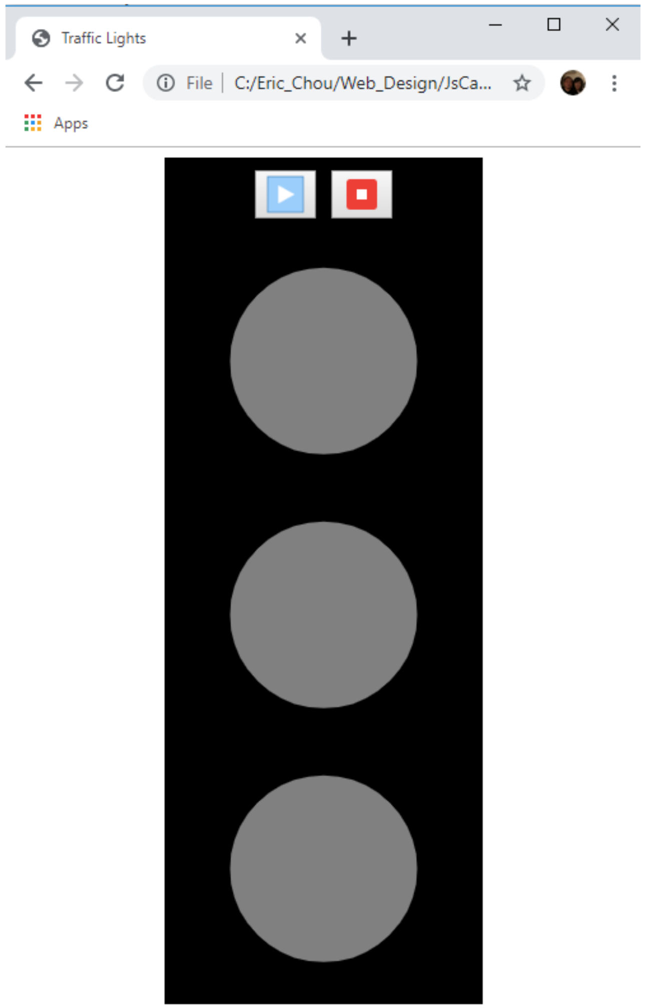


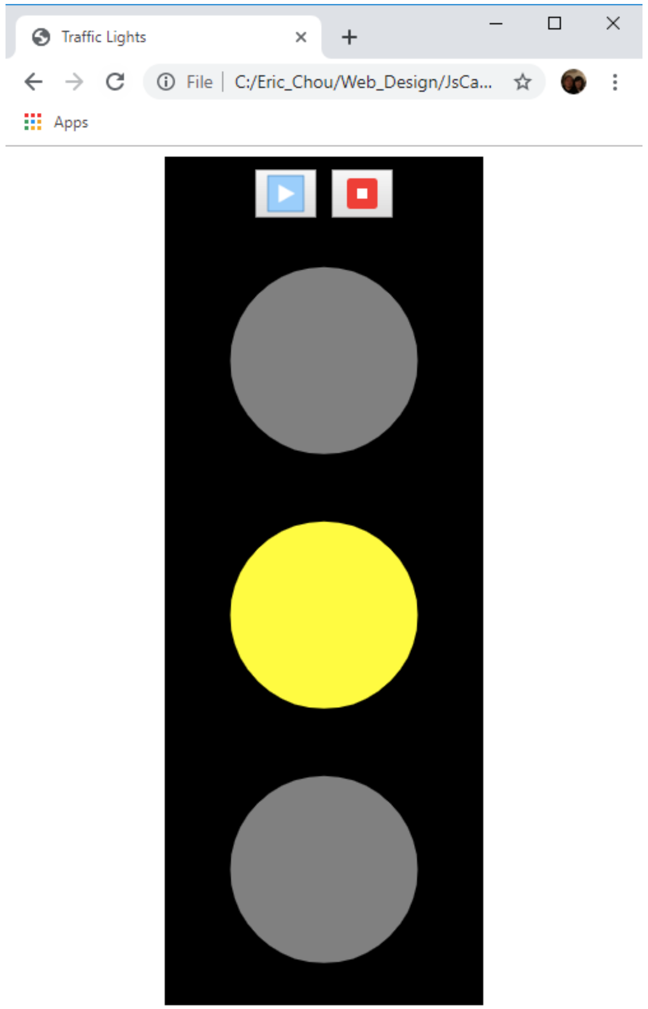
## Simple Game Loop without Exit

- Design a state machine to control the game.
- Initial state to set up the initial condition
- Final state to set up the postgame condition.
- Provide an event to break the game loop.
- Provide an event to start the game loop.

## Demo Program: anime2.html

- Start event: onclick of start button.
- Game over event: onclick of stop button.
- 4-state state machine.
- Both update status and draw screen in draw() function.


Traffic Lights
File
C:/Eric_Chou/Web_Design/JsCa...
Apps


Traffic Lights
File
C:/Eric_Chou/Web_Design/JsCa...
Apps

## SVG and Canvas

SECTION 3

## Clarifying the SVG – Canvas Relationship

It’s important to understand the differences between **SVG** and canvas elements. **SVG** is an **XML**-based vector graphics format. You can add styles to it with **CSS** and add dynamic behavior to it using the **SVG DOM**.

**Canvas is bitmap based.** It allows you to draw graphics and shapes through **JavaScript**. Like **SVG**, you may add style and dynamic behavior to it. Here are some reasons to use the canvas over **SVG**.

- When it comes to draw complex graphics, canvas is faster
- you can save images off the canvas whereas you can’t using **SVG**
- everything in the canvas is a pixel.

## Advantage of SVG

The SVG has some advantages too.

- Being resolutions independent, it can scale for different screen resolutions
- Since it is XML under the hood, targeting different elements is easier
- it’s good at complex animations

So which one to choose over another? to develop a resolution dependent, highly interactive and vector-based graphics, choose SVG. If you to render graphics really fast, like in a game, or don’t want to deal with XML, choose the canvas. Actually, they complement each other while delivering real-world applications.

## Lab

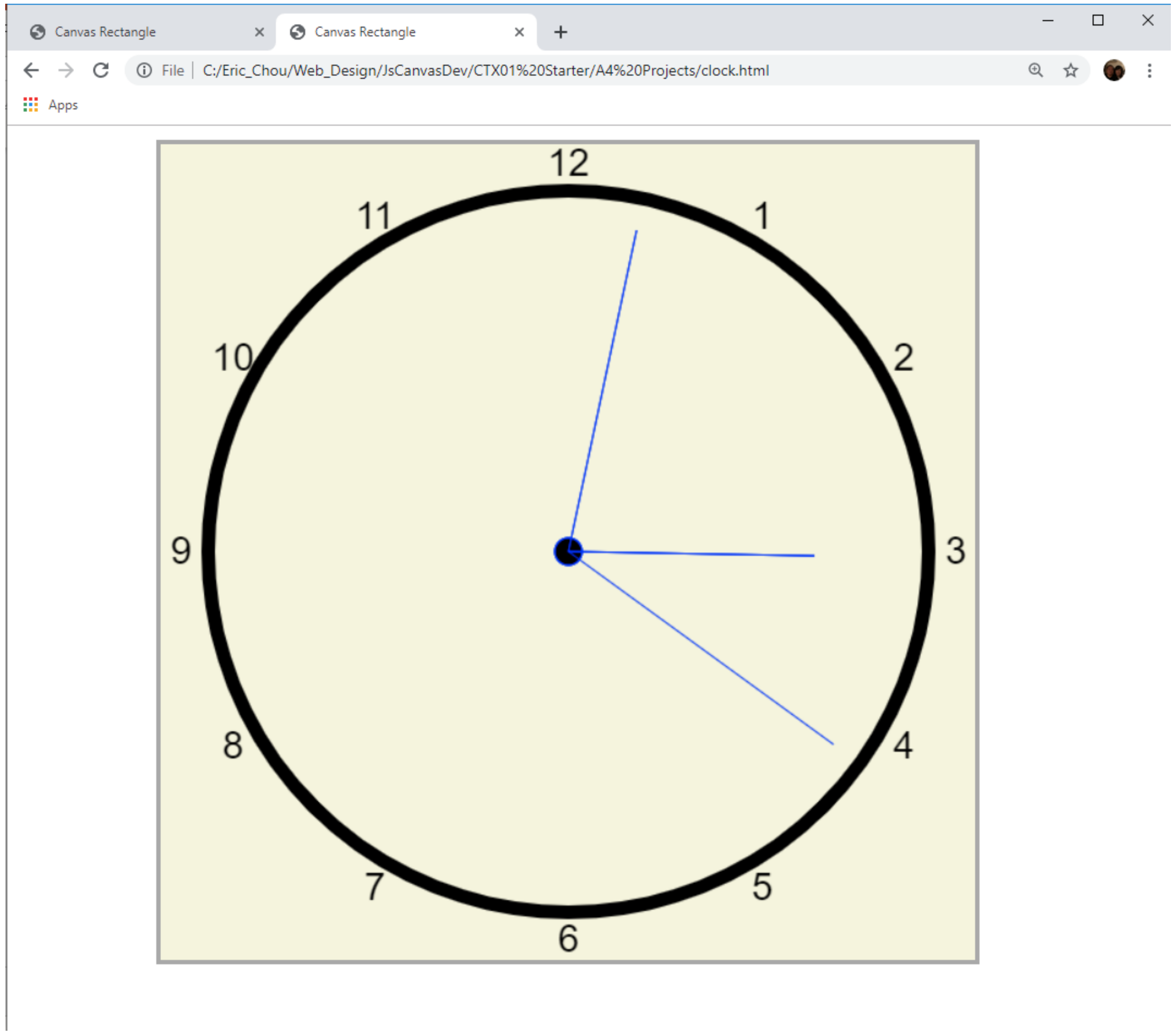
WEB CLOCK

## Demo Program: clock.html

- The clock application’s **drawCircle()** method draws the circle representing the clock face by invoking **beginPath()** to begin a path, and subsequently invokes **arc()** to create a circular path. That path is invisible until the application invokes **stroke()**. Likewise, the application’s **drawCenter()** method draws the small filled circle at the center of the clock with a combination of **beginPath()**, **arc()**, and **fill().**
- The application’s **drawNumerals()** method draws the numbers around the face of the clock with the **fillText()** method, which draws filled text in the canvas.
- Unlike the **arc()** method, **fillText()** does not create a path; instead, **fillText()** immediately renders text in the canvas.

## Demo Program: clock.html

- The clock hands are drawn by the application’s **drawHand()** method, which uses three methods to draw the lines that represent the clock hands: **moveTo()**, **lineTo()**, and **stroke()**. The **moveTo()** method moves the graphics pen to a specific location in the canvas, **lineTo()** draws an invisible path to the location that you specify, and **stroke()** makes the current path visible.
- The application animates the clock with setInterval(), which invokes the application’s **drawClock()** function once every second. The **drawClock()** function uses **clearRect()** to erase the canvas, and then it redraws the clock.


Canvas Rectangle
Canvas Rectangle
File
C:/Eric_Chou/Web_Design/JsCanvasDev/CTX01%20Starter/A4%20Projects/clock.html
Apps
12
1
2
3
4
5
6
7
8
9
10
11

HTML 5

## CANVAS

## Moving Across the Page

SECTION 4

# Moving Across the Page

- Let’s use **canvas** and **setInterval** to draw a square and move it slowly across a page. Create a new file called *canvasanimation.html* and add the following HTML:

```
<!DOCTYPE html>
<html>
 <head>
 <title>Canvas Animation</title>
 </head>
 <body>
 <canvas id="canvas" width="200" height="200"></canvas>
  <script>
     // We'll fill this in next
  </script>
 </body>
</html>
```

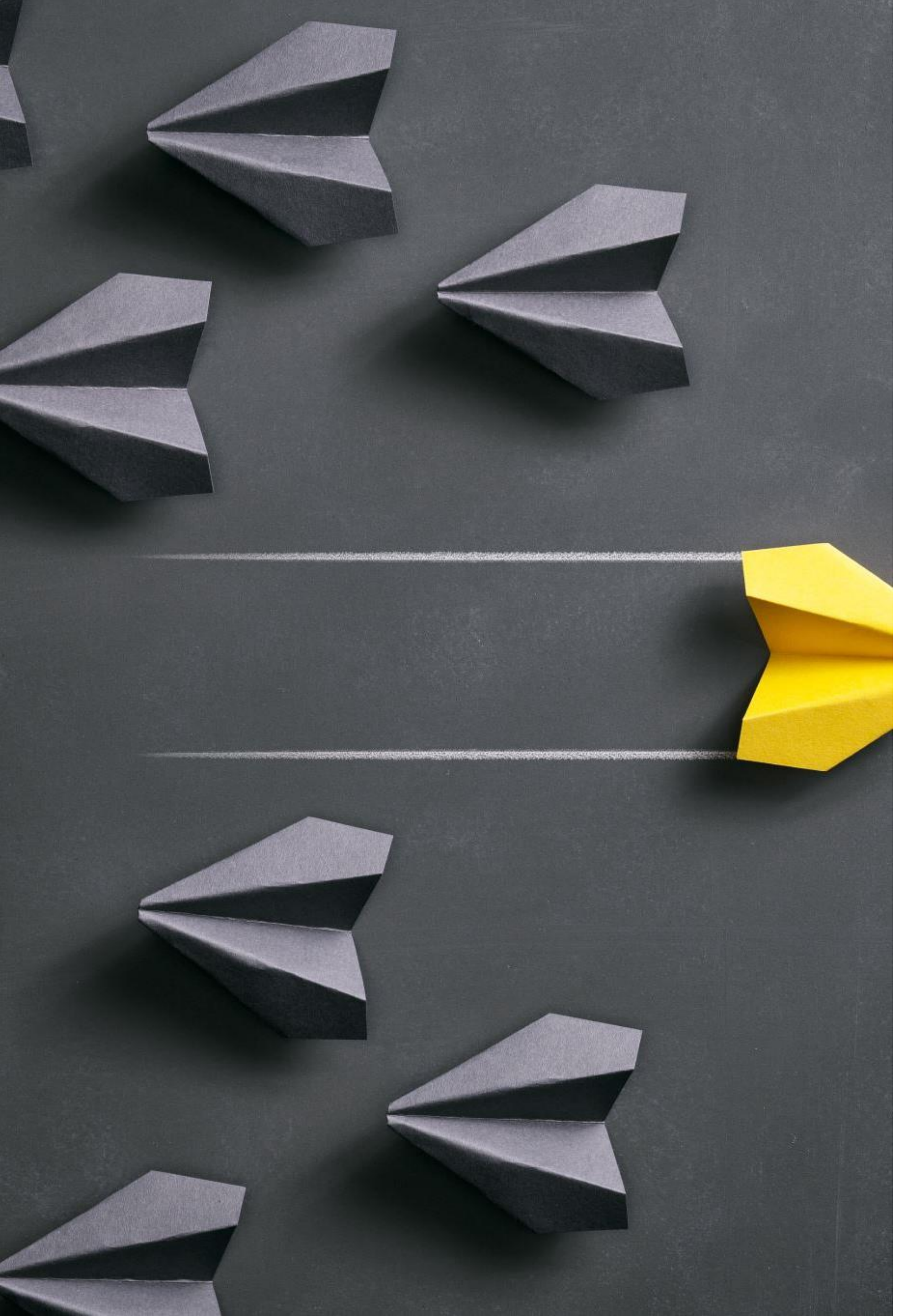
# Moving Across the Page

Now add the following JavaScript to the script element:

```
     var canvas = document.getElementById("canvas");
     var ctx = canvas.getContext("2d");
     var position = 0;
     setInterval(function () {
❶        ctx.clearRect(0, 0, 200, 200);
❷        ctx.fillRect(position, 0, 20, 20);
❸        position++;
❹        if (position > 200) {
              position = 0;
         }
❺    }, 30);
```

## Clearing the Canvas

- Inside the function we passed to setInterval, we call clearRect at ❶, which clears a rectangular area on the canvas. The clearRect method takes four arguments, which set the position and size of the rectangle to be cleared. As with fillRect, the first two arguments represent the x- and y-coordinates of the top-left corner of the rectangle, and the last two represent the width and height.
- Calling ctx.clearRect(0, 0, 200, 200) erases a 200-by-200-pixel rectangle, starting at the very top-left corner of the canvas. Because our canvas is exactly 200 by 200 pixels, this will clear the entire canvas.



## Drawing the Rectangle

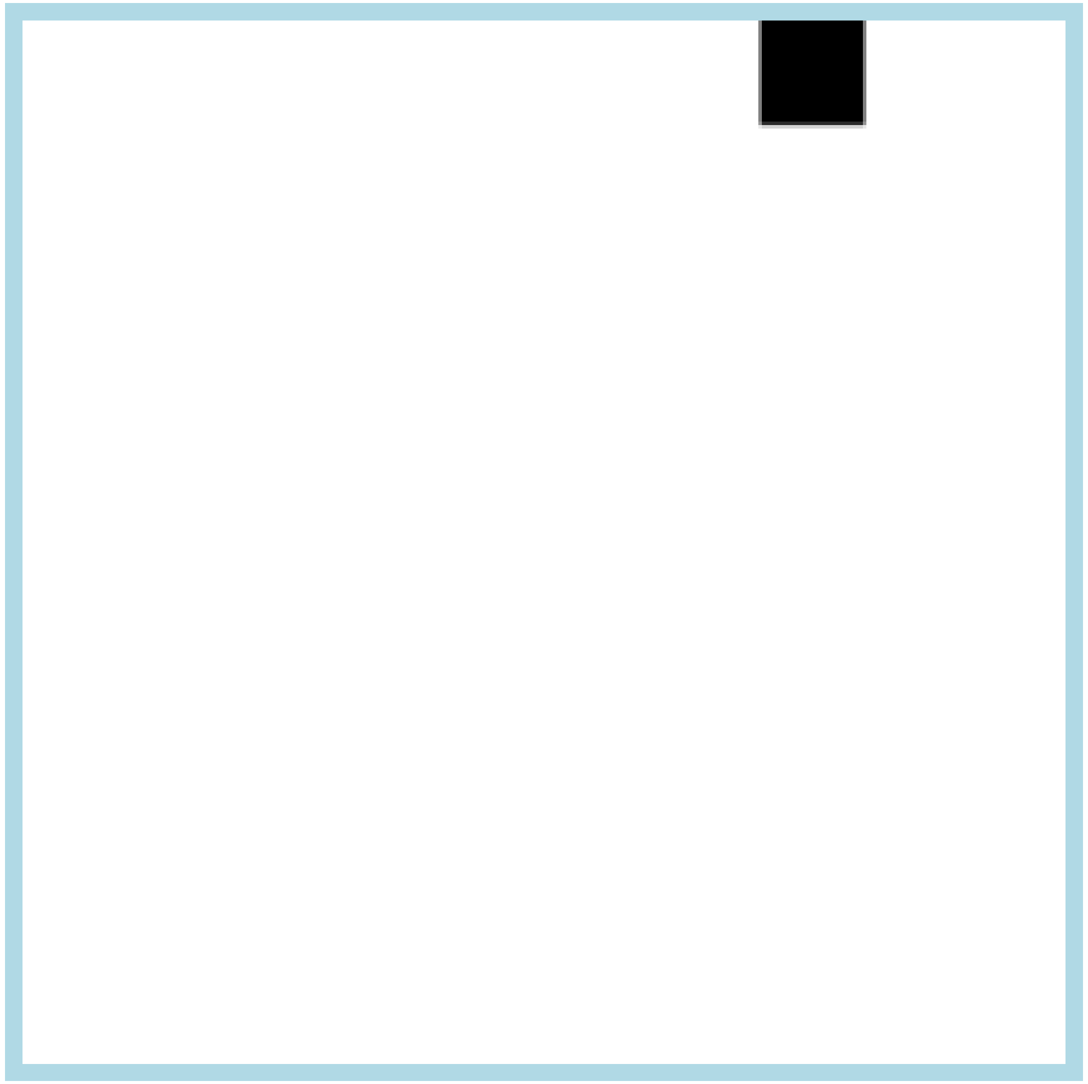
- Once we’ve cleared the canvas, at ❷ we use ctx.fillRect (position, 0, 20, 20) to draw a 20-pixel square at the point (position, 0). When our program starts, the square will be drawn at (0, 0) because position starts off set to 0.

## Changing the Position

- Next, we increase position by 1, using position++ at ❸. Then at ❹ we ensure that position doesn't get larger than 200 with the check if (position > 200). If it is, we reset it to 0.

## Viewing the Animation in the Browser

- When you load this page in your browser, setInterval will call the supplied function once every 30 milliseconds, or about 33 times a second (this time interval is set by the second argument to setInterval, at ❺). Each time the supplied function is called, it clears the canvas, draws a square at (position, 0), and increments the variable position. As a result, the square gradually moves across the canvas. When the square reaches the end of the canvas (200 pixels to the right), its position is reset to 0.
- Figure 14-1 shows the first four steps of the animation, zoomed in to the top-left corner of the canvas.



## Demo Program: anime1.html

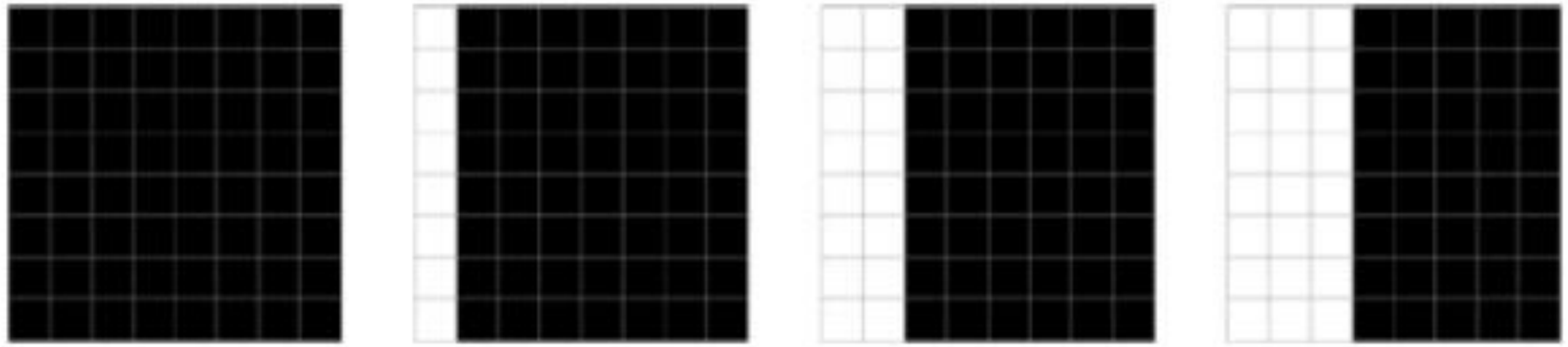


*Figure 14-1. A close-up of the top-left corner of the canvas for the first four steps of the animation. At each step,* `position` *is incremented by 1 and the square moves 1 pixel to the right.*

## Animating the Size of a Square

SECTION 5

# Animating the Size of a Square

By making only three changes to the code in the previous section, we can create a square that grows larger instead of moving. Here’s what that code would look like:

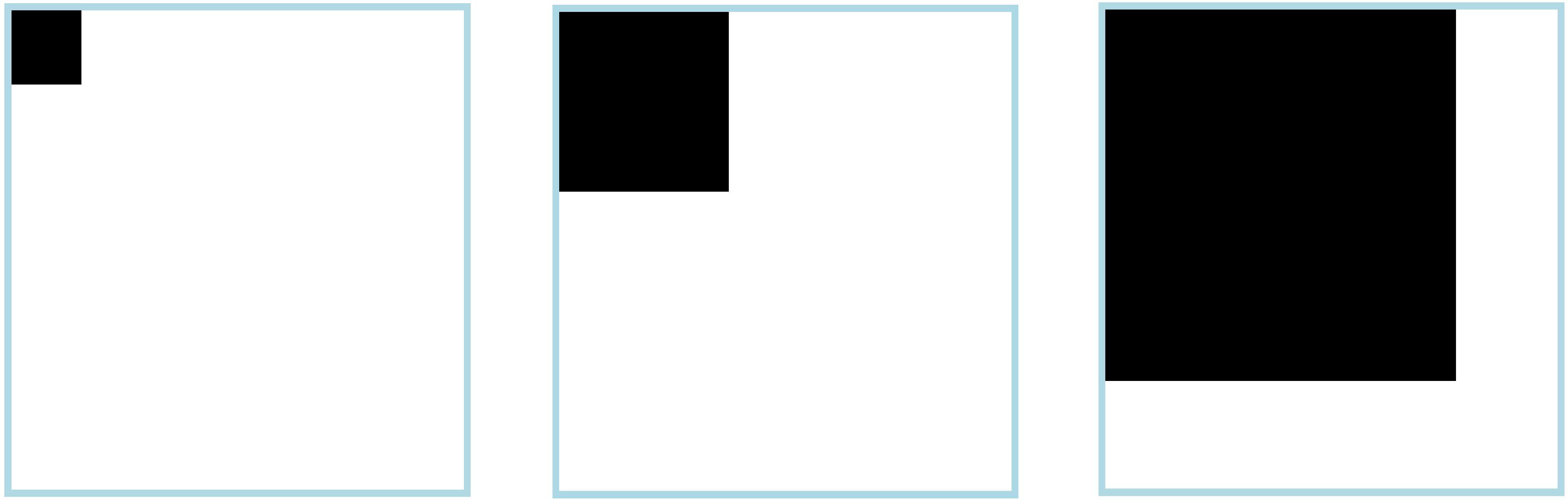
```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
var size = 0;
setInterval(function () {
  ctx.clearRect(0, 0, 200, 200);
  ctx.fillRect(0, 0, size, size);
  size++;
  if (size > 200) {
   size = 0;
  }
}, 30);
```

## Animating the Size of a Square

- As you can see, we’ve done two things. First, instead of a position variable, we now have a variable named size, which will control the dimensions of the square. Second, instead of using this variable to set the square’s horizontal position, we’re using it to set the square’s width and height with the code ctx.fillRect(0, 0, size, size). This will draw a square at the top-left corner of the canvas, with the width and height both set to match size. Because size starts at 0, the square will start out invisible.

## Animating the Size of a Square

- The next time the function is called, size will be 1, so the square will be 1 pixel wide and tall. Each time the square is drawn, it grows a pixel wider and a pixel taller. When you run this code, you should see a square appear at the top-left corner of the canvas and grow until it fills the entire canvas. Once it fills the entire canvas — that is, if (size > 200) — the square will disappear and start growing again from the top-left corner.
- Figure 14-2 shows a close-up of the top-left corner of the canvas for the first four steps of this animation.



## Demo Program: anime2.html

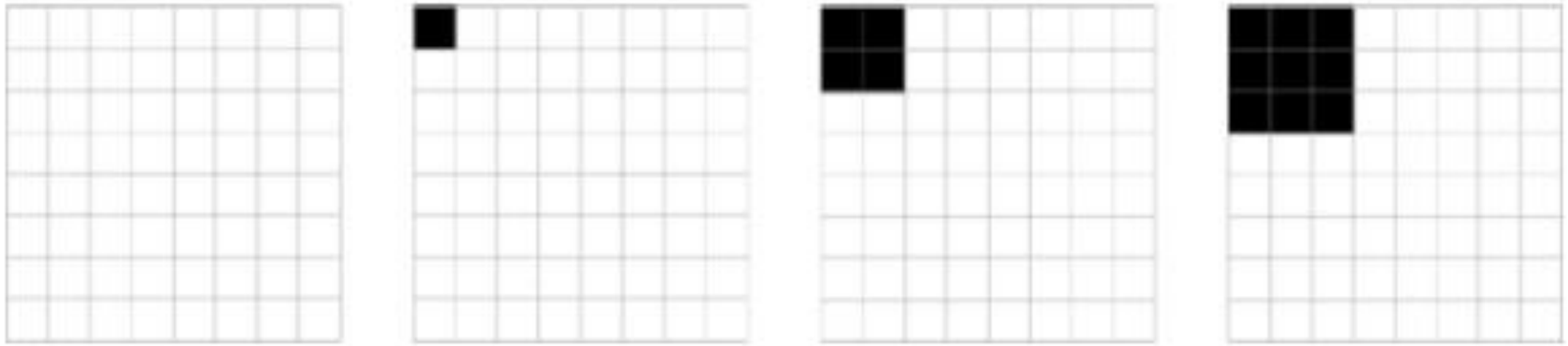


*Figure 14-2. In each step of this animation,* `size` *is incremented by 1 and the width and height of the square grow by 1 pixel.*

## A Random Bee

SECTION 6

## A Random Bee

- Now that we know how to move and grow objects on our screen, let’s try something a bit more fun. Let’s make a bee that flies randomly around the canvas! We’ll draw our bee using a number of circles, like this:

- The animation will work very similarly to the moving square animation: we’ll set a position, and then for every step of the animation, we’ll clear the canvas, draw the bee at that position, and modify the position. The difference is that to make the bee move randomly, we’ll need to use more complex logic for updating the bee’s position than we used for the square animation. We’ll build up the code for this animation in a few sections.

## A New circle Function

We'll draw our bee using a few circles, so first we'll make a circle function to fill or outline circles:

```
  var circle = function (x, y, radius, 
fillCircle) {
    ctx.beginPath();
❶    ctx.arc(x, y, radius, 0, Math.PI 
* 2, false);
❷    if (fillCircle) {
❸      ctx.fill();
    } else {
❹      ctx.stroke();
    }
  };
```

## A New circle Function

- The function takes four arguments: x, y, radius, and fillCircle. We used a similar circle function in Chapter 13, but here we’ve added fillCircle as an extra argument. When we call this function, this argument should be set to true or false, which determines whether the function draws a filled circle or just an outline.
- Inside the function, we use the arc method at ❶ to create the circle with its center at the position (x, y) and a radius of radius. After this, we check to see if the fillCircle argument is true at ❷. If it is true, we fill the circle using ctx.fill at ❸. Otherwise, we outline the circle using ctx.stroke at ❹.

## Drawing the Bee

Next, we create the drawBee function to draw the bee. The drawBee function uses the circle function to draw a bee at the coordinates specified by its x and y arguments. It looks like this:

```
var drawBee = function (x, y) {
❶  ctx.lineWidth = 2;
   ctx.strokeStyle = "Black";
   ctx.fillStyle = "Gold";
❷  circle(x, y, 8, true);
   circle(x, y, 8, false);
   circle(x - 5, y - 11, 5, false);
   circle(x + 5, y - 11, 5, false);
   circle(x - 2, y - 1, 2, false);
   circle(x + 2, y - 1, 2, false);
};
```

## Drawing the Bee

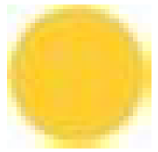
- In the first section of this code at ❶, we set the lineWidth, strokeStyle, and fillStyle properties for our drawing. We set the lineWidth to 2 pixels and the strokeStyle to Black. This means that our outlined circles, which we’ll use for the bee’s body, wings, and eyes, will have thick black borders. The fillStyle is set to Gold, which will fill the circle for our bee body with a nice yellow color.
- In the second section of the code at ❷, we draw a series of circles to create our bee. Let’s go through those one at a time.

## Drawing the Bee

- The first circle draws the bee’s body using a filled circle with a center at the point (x, y) and a radius of 8 pixels:

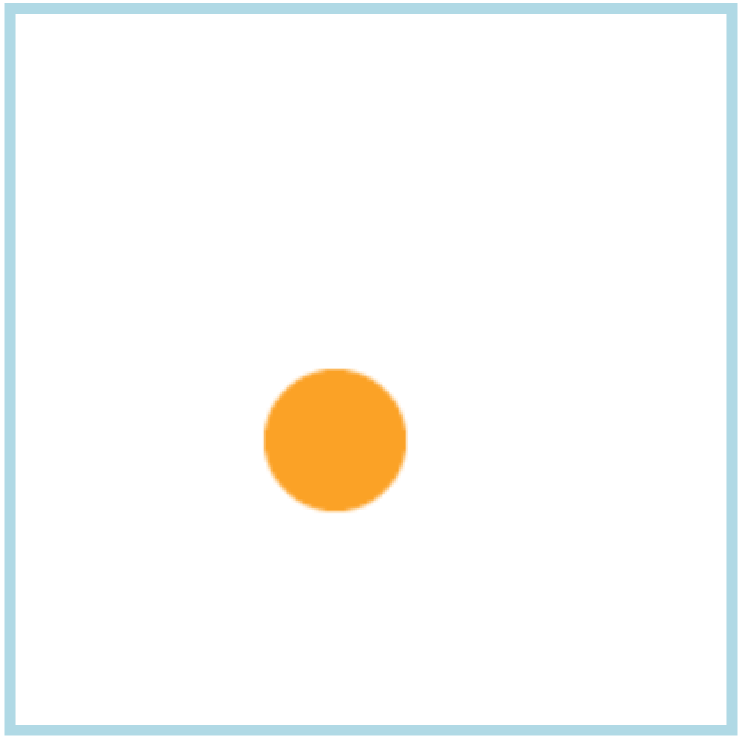
```
circle(x, y, 8, true);
```

- Because we set the fillStyle to Gold, this circle will be filled in with yellow like so:



# Demo Program: bees1.html

```
<body>
<canvas id="canvas" width="200" height="200"></canvas>
<script>
    var canvas = document.getElementById("canvas");
    var ctx = canvas.getContext("2d");
    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();
    }
  };
  var x = 90;
  var y = 120;
  ctx.fillStyle = "orange";
  circle(x, y, 20, true);
</script>
</body>
```



## Drawing the Bee

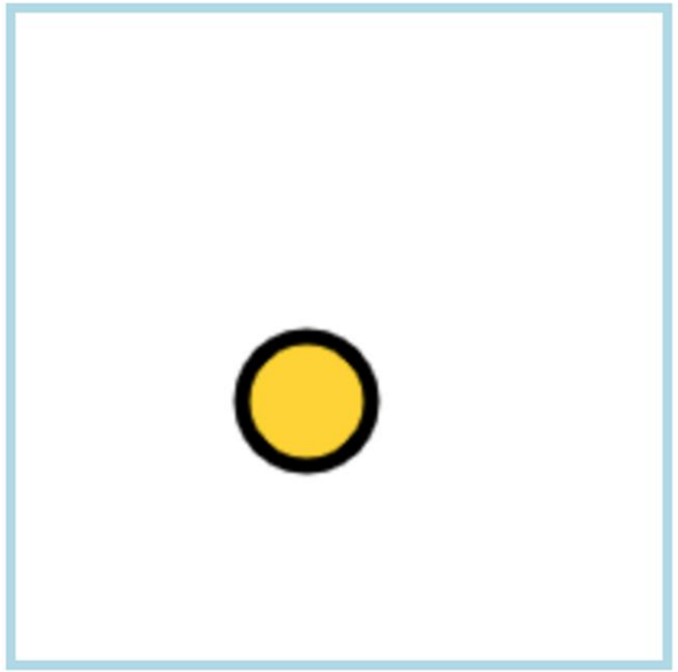
- This second circle draws a black outline around the bee’s body that’s the same size and in the same place as the first circle:

```
    circle(x, y, 8, false);
```

- Added to the first circle, it looks like this:

# Demo Program: bees2.html

```
  var canvas = document.getElementById("canvas");
  var ctx = canvas.getContext("2d");
  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();
  }
};
var x = 90;
var y = 120;
ctx.fillStyle = "gold";
ctx.strokeStyle = "black";
ctx.lineWidth = "5";
circle(x, y, 20, true);
circle(x, y, 20, false);
```

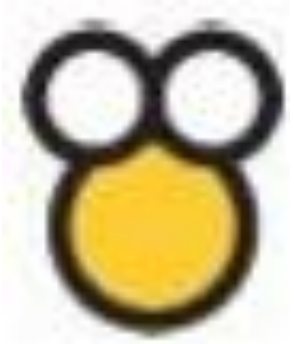


## Drawing the Bee

- Next, we use circles to draw the bee's wings. The first wing is an outlined circle with its center 5 pixels to the left and 11 pixels above the center of the body, with a radius of 5 pixels. The second wing is the same, except it's 5 pixels to the *right* of the body's center.

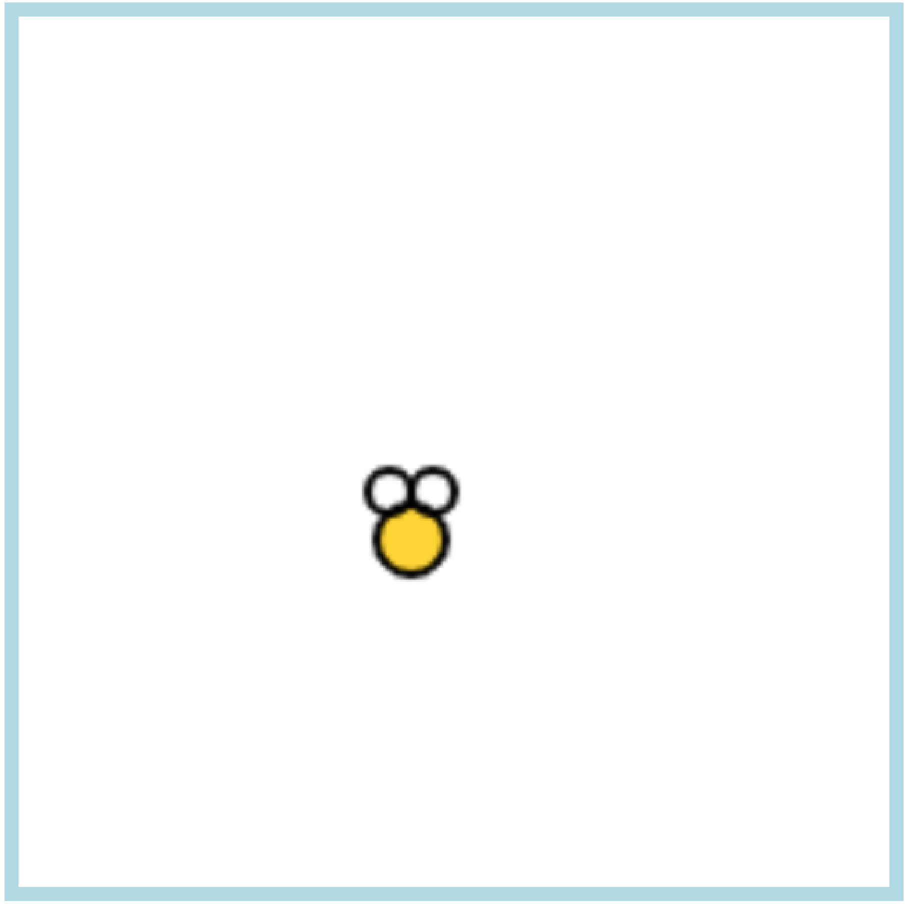
```
    circle(x - 5, y - 11, 5, false);
    circle(x + 5, y - 11, 5, false);
```

- With those circles added, our bee looks like this:



# Demo Program: bees3.html

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
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();
  }
};
var x = 90;
var y = 120;
ctx.fillStyle = "gold";
ctx.strokeStyle = "black";
ctx.lineWidth = "2";
circle(x, y, 8, true);
circle(x, y, 8, false);
circle(x - 5, y - 11, 5, false);
circle(x + 5, y - 11, 5, false);
```



## Drawing the Bee

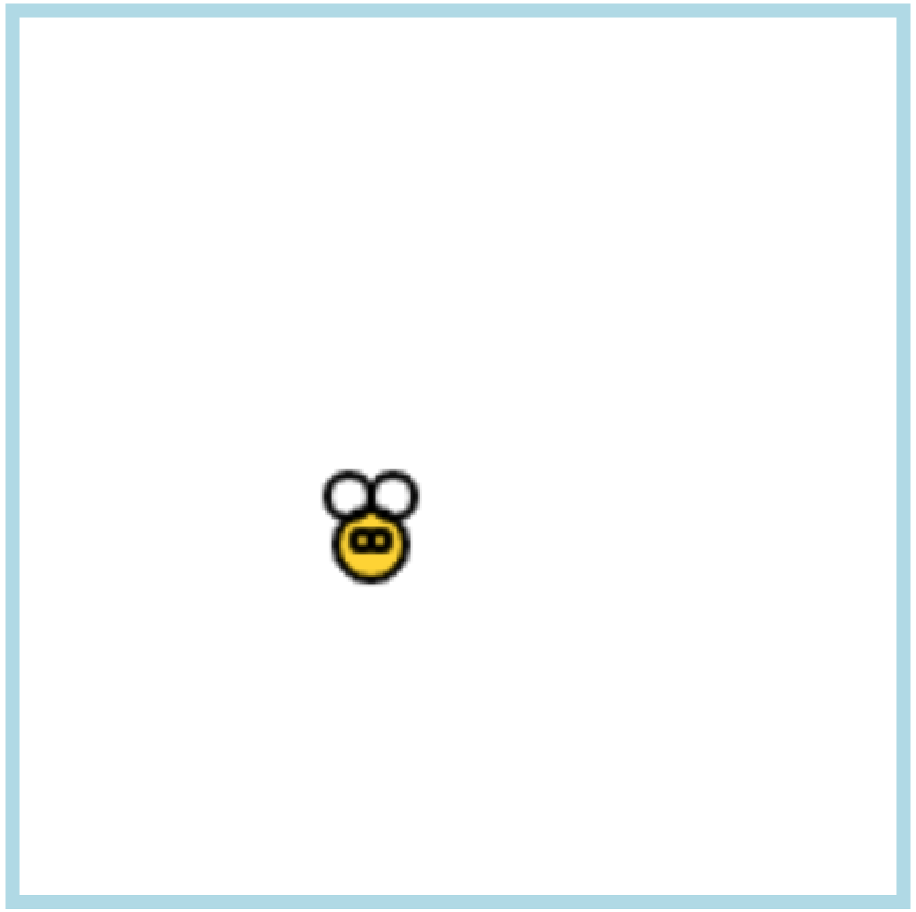
- Finally, we draw the eyes. The first one is 2 pixels to the left of the center of the body and 1 pixel above, with a radius of 2 pixels. The second one is the same, except it's 2 pixels right of center.

```
    circle(x - 2, y - 1, 2, false);
    circle(x + 2, y - 1, 2, false);
```

- Together, these circles create a bee, with its body centered around the (x, y) coordinate passed into the drawBee function.

# Demo Program: bees4.html

```
var canvas = document.getElementById("canvas");
var ctx = canvas.getContext("2d");
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();
  }
};
var drawBee = function (x, y) {
  ctx.lineWidth = 2;
  ctx.strokeStyle = "Black";
  ctx.fillStyle = "Gold";
  circle(x, y, 8, true);
  circle(x, y, 8, false);
  circle(x - 5, y - 11, 5, false);
  circle(x + 5, y - 11, 5, false);
  circle(x - 2, y - 1, 2, false);
  circle(x + 2, y - 1, 2, false);
};
drawBee(80, 120);
```



## Updating the Bee’s Location

- We’ll create an update function to randomly change the bee’s *x*-and *y*-coordinates in order to make it appear to buzz around the canvas. The update function takes a single coordinate; we update the *x* and *y*-coordinates one at a time so that the bee will move randomly left and right and up and down. The **update** function looks like this:

## Updating the Bee's Location

```
  var update = function (coordinate) {
❶   var offset = Math.random() * 4 - 2;
❷   coordinate += offset;
❸   if (coordinate > 200) {
      coordinate = 200;
    }
❹   if (coordinate < 0) {
      coordinate = 0;
    }
❺   return coordinate;
  };
```

## Updating the Bee’s Location

## Changing the Coordinate with an Offset Value

- At ❶, we create a variable called offset, which will determine how much to change the current coordinate. We generate the offset value by calculating Math.random() * 4 - 2. This will give us a random number between –2 and 2. Here’s how: calling Math.random() on its own gives us a random number between 0 and 1, so Math.random() * 4 produces a random number between 0 and 4. Then we subtract 2 to get a random number between –2 and 2.
- At ❷ we use coordinate += offset to modify our coordinate with this offset number. If offset is a positive number, coordinate will increase, and if it’s a negative number, coordinate will decrease. For example, if coordinate is set to 100 and offset is 1, then after we run the line at ❷, coordinate will be 101. However, if coordinate is 100 and offset is -1, this would change coordinate to 99.

## Updating the Bee’s Location Checking if the Bee Reaches the Edge

- At ❸ and ❹ we prevent the bee from leaving the canvas by making sure coordinate never increases above 200 or shrinks below 0. If coordinate gets bigger than 200, we set it back to 200, and if it goes below 0, we reset it to 0.

## Updating the Bee's Location
## Returning the Updated Coordinate

- Finally, at ❺ we return coordinate. Returning the new value of coordinate lets us use that value in the rest of our code. Later we'll use this return value from the update method to modify the x and y values like this:

```
x = update(x);
y = update(y);
```

## Animating Our Buzzing Bee

Now that we have the circle, drawBee, and update functions, we can write the animation code for our buzzing bee.

```
   var canvas = document.getElementById("canvas");
   var ctx = canvas.getContext("2d");
   var x = 100;
   var y = 100;
   setInterval(function () {
❶    ctx.clearRect(0, 0, 200, 200);
❷    drawBee(x, y);
❸    x = update(x);
     y = update(y);
❹    ctx.strokeRect(0, 0, 200, 200);
   }, 30);
```

## Animating Our Buzzing Bee

- As usual, we start with the var canvas and var ctx lines to get the drawing context. Next, we create the variables x and y and set both to 100. This sets the bee's starting position at the point (100, 100), which puts it in the middle of the canvas, as shown in Figure 14-3.
- Next we call setInterval, passing a function to call every 30 milliseconds. Inside this function, the first thing we do is call clearRect at ❶ to clear the canvas. Next, at ❷ we draw the bee at the point (x, y). The first time the function is called, the bee is drawn at the point (100, 100), as you can see in
- Figure 14-3, and each time the function is called after that, it will draw the bee at a new, updated (x, y) position.

## Animating Our Buzzing Bee

- Next we update the x and y values starting at ❸. The update function takes a number, adds a random number between –2 and 2 to it, and returns that updated number. So the code x = update(x) basically means “change x by a small, random amount.”
- Finally, we call strokeRect at ❹ to draw a line around the edge of the canvas. This makes it easier for us to see when the bee is getting close to it. Without the border, the edge of the canvas is invisible.
- When you run this code, you should see the yellow bee randomly buzz around the canvas. Figure 14-4 shows a few frames from our animation.

## Animating Our Buzzing Bee

*Figure 14-3. The bee drawn at the point (100, 100)*

## Demo Program: bees5.html

```
var update = function (coordinate) {
  var offset = Math.random() * 4 - 2;
  coordinate += offset;
  if (coordinate > 200) {
     coordinate = 200;
  }
  if (coordinate < 0) {
     coordinate = 0;
  }
  return coordinate;
};
var draw = function(x, y) {
  ctx.clearRect(0, 0, 200, 200);
  drawBee(x, y);
  x = update(x);
  y = update(y);
  //ctx.strokeRect(0, 0, 200, 200);
}
// global setting
var x = 100;
var y = 100;
setInterval(draw(x, y), 30);
```

## Animating Our Buzzing Bee

- When you run this code, you should see the yellow bee randomly buzz around the canvas. Figure 14-4 shows a few frames from our animation.

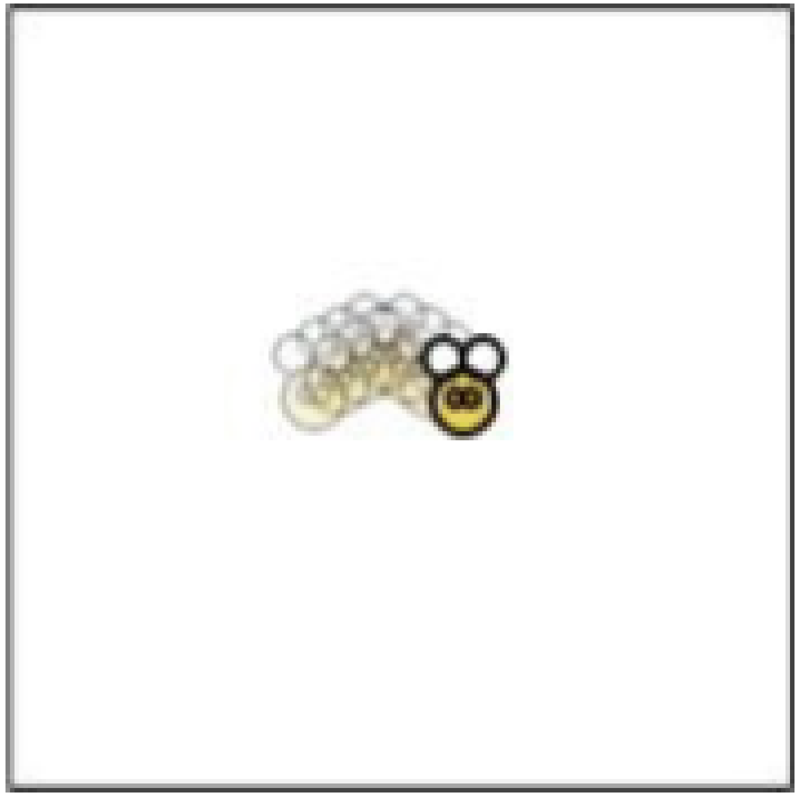
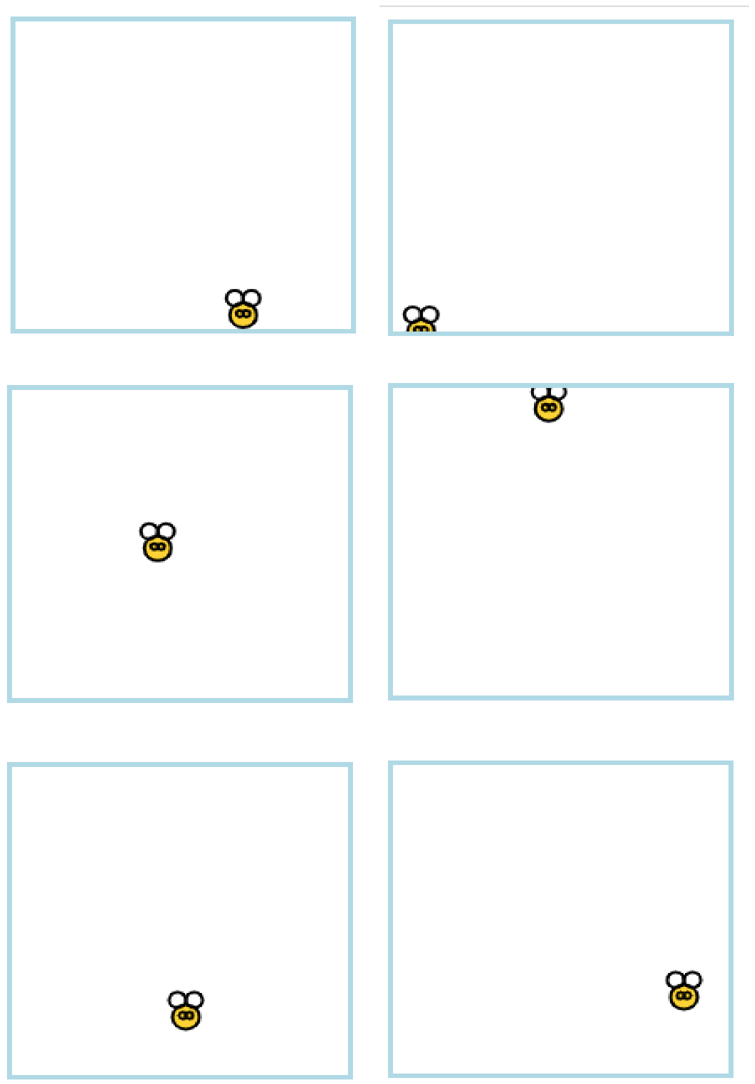


*Figure 14-4. The random bee animation*

## Demo Program: bees6.html



```
var update = function (coordinate, step) {
  var offset = Math.random() * step - step/2;
  coordinate += offset;
  if (coordinate > 200) {
     coordinate = 200;
  }
  if (coordinate < 0) {
     coordinate = 0;
  }
  return coordinate;
};

// global setting
var x = 100;
var y = 100;
setInterval(function() {
  ctx.clearRect(0, 0, 200, 200);
  drawBee(x, y);
  x = update(x, 10);
  y = update(y, 10);
  //ctx.strokeRect(0, 0, 200, 200);
}, 30);
```

# Bouncing a Ball!

SECTION 7

## Bouncing a Ball!

- Now let’s make a ball that bounces around the canvas. Whenever the ball hits one of the walls, it will bounce off at an angle, as a rubber ball would.
- First, we’ll create a JavaScript object to represent our ball with a Ball constructor. This object will store the ball’s speed and direction using two properties, xSpeed and ySpeed. The ball’s horizontal speed will be controlled by xSpeed, and the vertical speed will be controlled by ySpeed.
- We’ll make this animation in a new file. Create a new HTML file called *ball.html*, and add the following HTML:

## Bouncing a Ball!

```
<!DOCTYPE html>
<html>
 <head>
  <title>A Bouncing Ball</title>
 </head>
 <body>
  <canvas id="canvas" width="200"
  height="200"></canvas>
  <script>
  // We'll fill this in next
  </script>
 </body>
</html>
```

## The Ball Constructor

First we’ll create the Ball constructor, which we’ll use to create our bouncing ball. Type the following code into the <script> tags in *ball.html*:

```
var Ball = function () {
   this.x = 100;
   this.y = 100;
   this.xSpeed = -2;
   this.ySpeed = 3;
};
```

## The Ball Constructor

- Our constructor is very straightforward: it simply sets the starting position of the ball (this.x and this.y), the ball’s horizontal speed (this.xSpeed), and its vertical speed (this.ySpeed). We set the starting position to the point (100, 100), which is the center of our 200-by-200-pixel canvas. this.xSpeed is set to -2. This will make the ball move 2 pixels to the left for every step of the animation. this.ySpeed is set to 3. This will make the ball move 3 pixels down for every step of the animation. Therefore, the ball will move diagonally down (3 pixels) and to the left (2 pixels) between every frame.
- Figure 14-5 shows the starting position of the ball and its direction of movement.

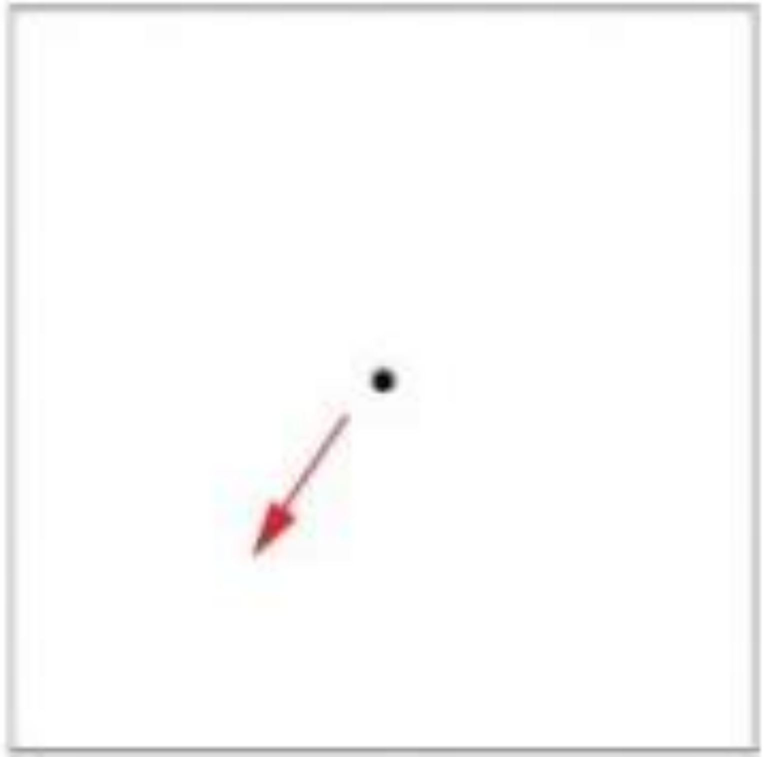


*Figure 14-5. The starting position of the ball, with an arrow indicating its direction*

## Drawing the Ball

- Next we'll add a draw method to draw the ball. We'll add this method to the Ball prototype so that any objects created by the Ball constructor can use it:

```
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();
 }
};
Ball.prototype.draw = function () {
 circle(this.x, this.y, 3, true);
};
```

## Drawing the Ball

- First we include our circle function, the same one we used earlier in A New circle Function. We then add the draw method to Ball.prototype. This method simply calls circle(this.x, this.y, 3, true) to draw a circle.
- The circle’s center will be at (this.x, this.y): the location of the ball. It will have a radius of 3 pixels. We pass true as the final argument to tell the circle function to fill the circle.

## Moving the Ball

To move the ball, we just have to update the x and y properties based on the current speed. We'll do that using the following move method:

```
 Ball.prototype.move = function ()
{
  this.x += this.xSpeed;
  this.y += this.ySpeed;
 };
```

## Moving the Ball

- We use this.x += this.xSpeed to add the horizontal speed of the ball to this.x. Then this.y += this.ySpeed adds the vertical speed to this.y. For example, at the beginning of the animation, the ball will be at the point (100, 100), with this.xSpeed set to -2 and this.ySpeed set to 3. When we call the move method, it subtracts 2 from the x value and adds 3 to the y value, which places the ball at the point (98, 103). This moves the ball's location to the left 2 pixels and down 3 pixels, as illustrated in Figure 14-6.

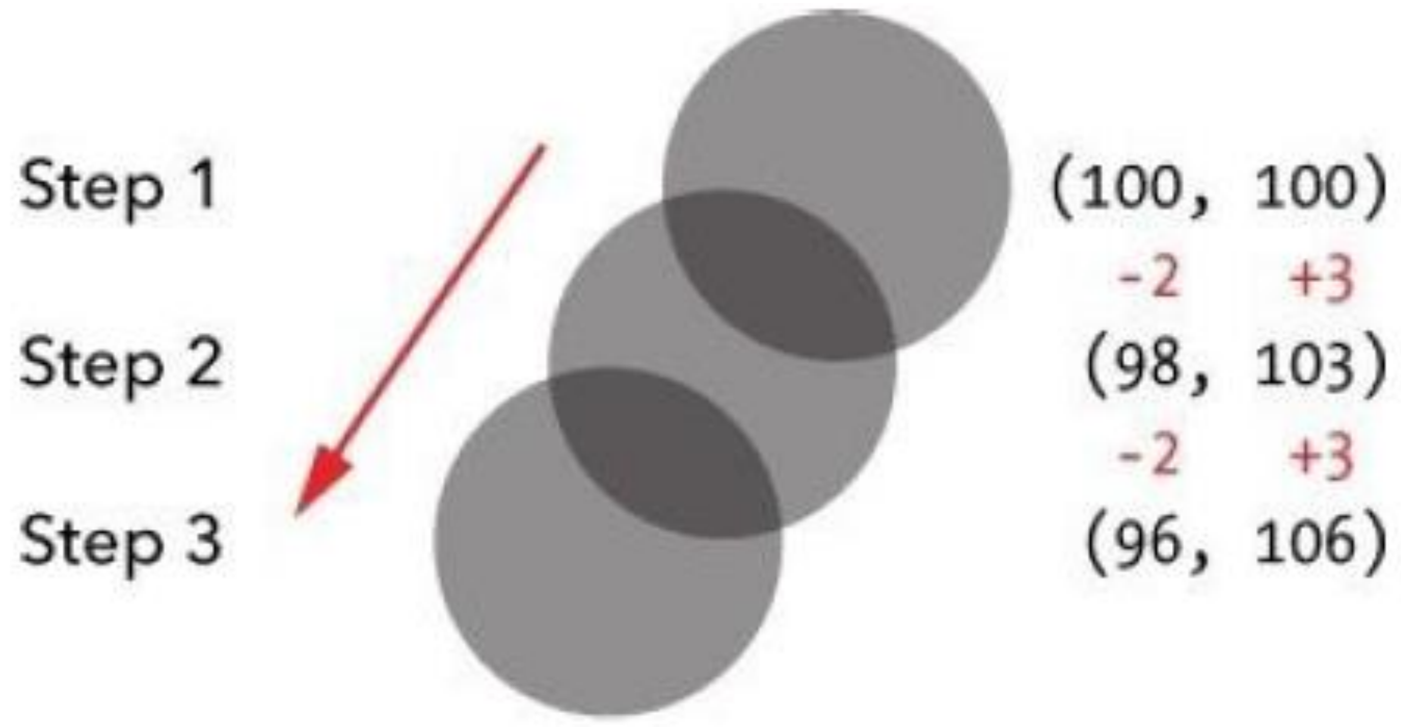


*Figure 14-6. The first three steps of the animation, showing how the x and y properties change*

## Bouncing the Ball

- At every step of the animation, we check to see if the ball has hit one of the walls. If it has, we update the xSpeed or ySpeed property by *negating* it (multiplying it by –1). For example, if the ball hits the bottom wall, we negate this.ySpeed. So if this.ySpeed is 3, negating it will make it -3. If this.ySpeed is -3, negating it will set it back to 3.
- We'll call this method checkCollision, because it checks to see if the ball has collided with (hit) the wall.

## Bouncing the Ball

```
 Ball.prototype.checkCollision=function(){
❶   if (this.x < 0 || this.x > 200) {
      this.xSpeed = -this.xSpeed;
     }
❷   if (this.y < 0 || this.y > 200) {
       this.ySpeed = -this.ySpeed;
     }
   };
```

## Bouncing the Ball

- At ❶, we determine whether the ball has hit the left wall or the right wall by checking to see if its x property is either less than 0 (meaning it hit the left edge) or greater than 200 (meaning it hit the right edge). If either of these is true, the ball has started to move off the edge of the canvas, so we have to reverse its horizontal direction. We do this by setting this.xSpeed equal to -this.xSpeed. For example, if this.xSpeed was -2 and the ball hit the left wall, this.xSpeed would become 2.
- At ❷, we do the same thing for the top and bottom walls. If this.y is less than 0 or greater than 200, we know the ball has hit the top wall or the bottom wall, respectively. In that case, we set this.ySpeed to be equal to -this.ySpeed.
- Figure 14-7 shows what happens when the ball hits the left wall. this.xSpeed starts as -2, but after the collision it is changed to 2. However, this.ySpeed remains unchanged at 3.

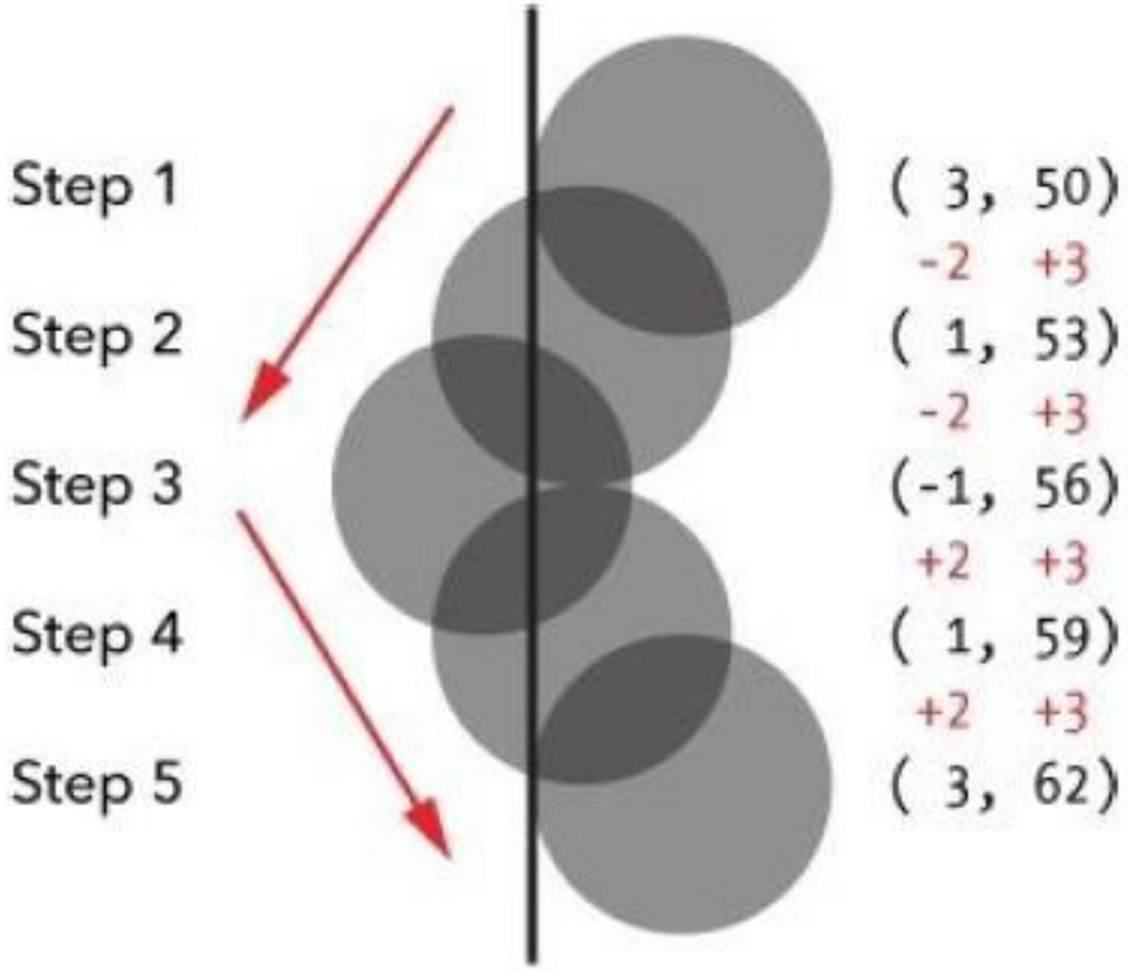


*Figure 14-7. How `this.xSpeed` changes after a collision with the left wall*

## Bouncing the Ball

- As you can see in Figure 14-7, in this case the center of the ball goes off the edge of the canvas at step 3 when it collides with a wall. During that step, part of the ball will disappear, but this happens so quickly that it’s barely noticeable when the animation is running.

## Animating the Ball

Now we can write the code that gets the animation running. This code sets up the object that represents the ball, and it uses setInterval to call the methods that draw and update the ball for each animation step.

```
  var canvas =
document.getElementById("canvas");
  var ctx = canvas.getContext("2d");
❶ var ball = new Ball();
❷ setInterval(function () {
❸   ctx.clearRect(0, 0, 200, 200);
❹   ball.draw();
    ball.move();
    ball.checkCollision();
❺   ctx.strokeRect(0, 0, 200, 200);
❻ }, 30);
```

## Animating the Ball

- We get the canvas and drawing context as usual on the first two lines. Then we create a ball object using new Ball() and save it in the variable ball at ❶. Next, we call setInterval at ❷, passing a function and the number 30 at ❻. As you've seen before, this means "call this function every 30 milliseconds."
- The function we pass to setInterval does several things. First, it clears the canvas, using ctx.clearRect(0, 0, 200, 200) at ❸. After this, it calls the draw, move, and checkCollision methods at ❹ on the ball object. The draw method draws the ball at its current *x*- and *y*-coordinates.
- The move method updates the position of the ball based on its xSpeed and ySpeed properties. Finally, the checkCollision method updates the direction of the ball, if it hits a wall.

## Animating the Ball

- The last thing we do in the function passed to setInterval is call ctx.strokeRect(0, 0, 200, 200) at ❺ to draw a line around the edge of the canvas, so we can see the walls the ball is hitting.
- When you run this code, the ball should immediately start moving down and to the left. It should hit the bottom wall first, and bounce up and to the left. It will continue to bounce around the canvas as long as you leave the browser window open.

## Demo Program: ball1.html

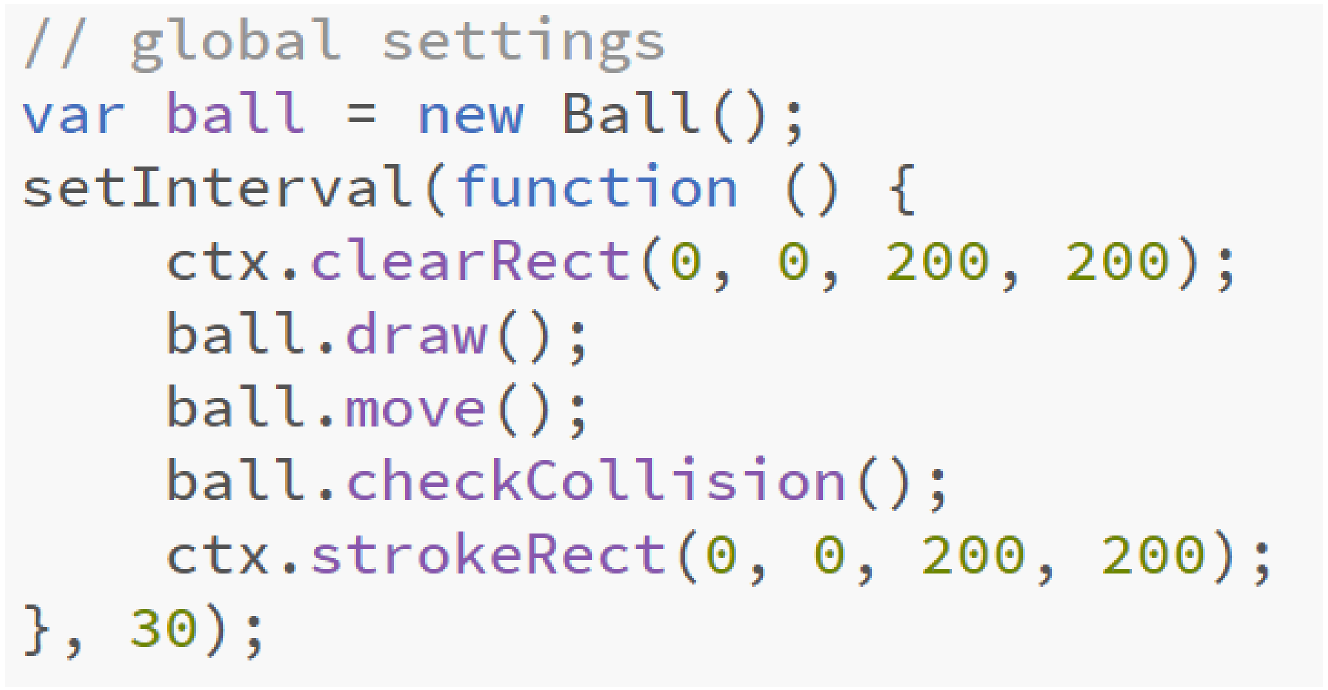
```
<body>
<canvas id="canvas" width="200" height="200"></canvas>
<script>
      var canvas = document.getElementById("canvas");
      var ctx = canvas.getContext("2d");

      var Ball = function () {
        this.x = 100;
        this.y = 100;
        this.xSpeed = -2;
        this.ySpeed = 3;
      };

    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();
        }
    };
```

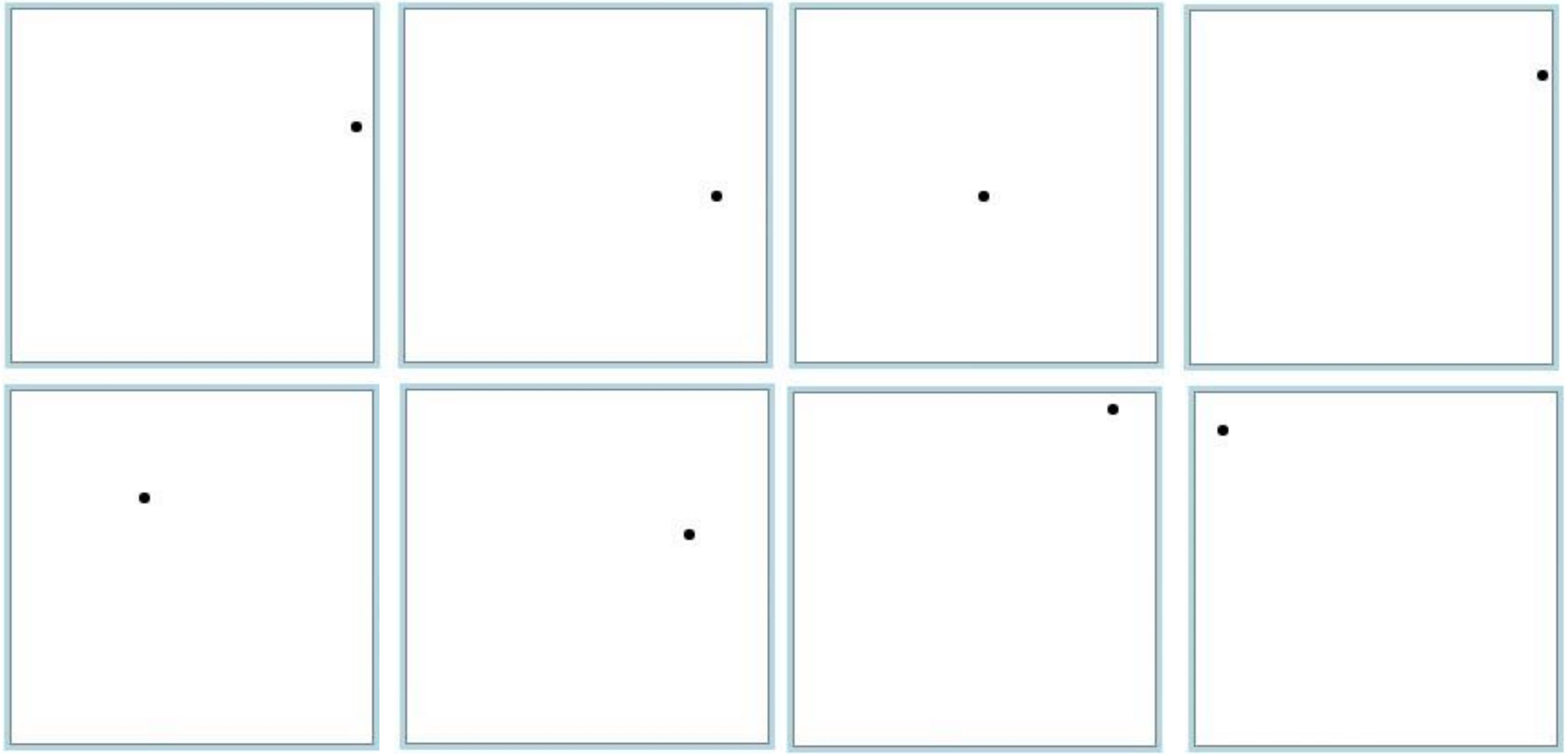
```
Ball.prototype.draw = function () {
    circle(this.x, this.y, 3, true);
};
Ball.prototype.move = function () {
    this.x += this.xSpeed;
    this.y += this.ySpeed;
};
Ball.prototype.checkCollision = function(){
   if (this.x < 0 || this.x > 200) {
     this.xSpeed = -this.xSpeed;
    }
   if (this.y < 0 || this.y > 200) {
      this.ySpeed = -this.ySpeed;
    }
};
```

## Demo Program: ball1.html



```
// global settings
var ball = new Ball();
setInterval(function () {
    ctx.clearRect(0, 0, 200, 200);
    ball.draw();
    ball.move();
    ball.checkCollision();
    ctx.strokeRect(0, 0, 200, 200);
}, 30);
```

## Demo Program: ball1.html



## Summary

SECTION 8

## Summary

- In this chapter, we combined our knowledge of animation from Chapter 11 with our knowledge of the canvas element to create various canvas-based animations. We began simply by moving and growing squares on the canvas.
- Next, we made a bee buzz randomly around the screen, and we ended with an animation of a bouncing ball.

## Summary

- All of these animations work in basically the same way: we draw a shape of a particular size in a particular position, then we update that size or position, and then we clear the canvas and draw the shape again. For elements moving around a 2D canvas, we generally have to keep track of the x- and ycoordinates of the element. For the bee animation, we added or subtracted a random number from the xand y-coordinates. For the bouncing ball, we added the current xSpeed and ySpeed to the x- and ycoordinates.
- In the next chapter, we'll add interactivity to our canvas, which will let us control what's drawn to the canvas using the keyboard.

## Objectives

- Now that you know how to work with the canvas; draw and color objects; and make objects move, bounce, and grow in size, let’s liven things up by adding some interactivity!
- In this chapter, you’ll learn how to make your canvas animations respond when a user presses a key on the keyboard. This way, a player can control an animation by pressing an arrow key or one of a few assigned letters on their keyboard (like the classic W, A, S, D game controls). For example, instead of just having a ball bounce across a screen, we can have a player control the movement of the ball using the arrow keys.

# jQuery

SECTION 9

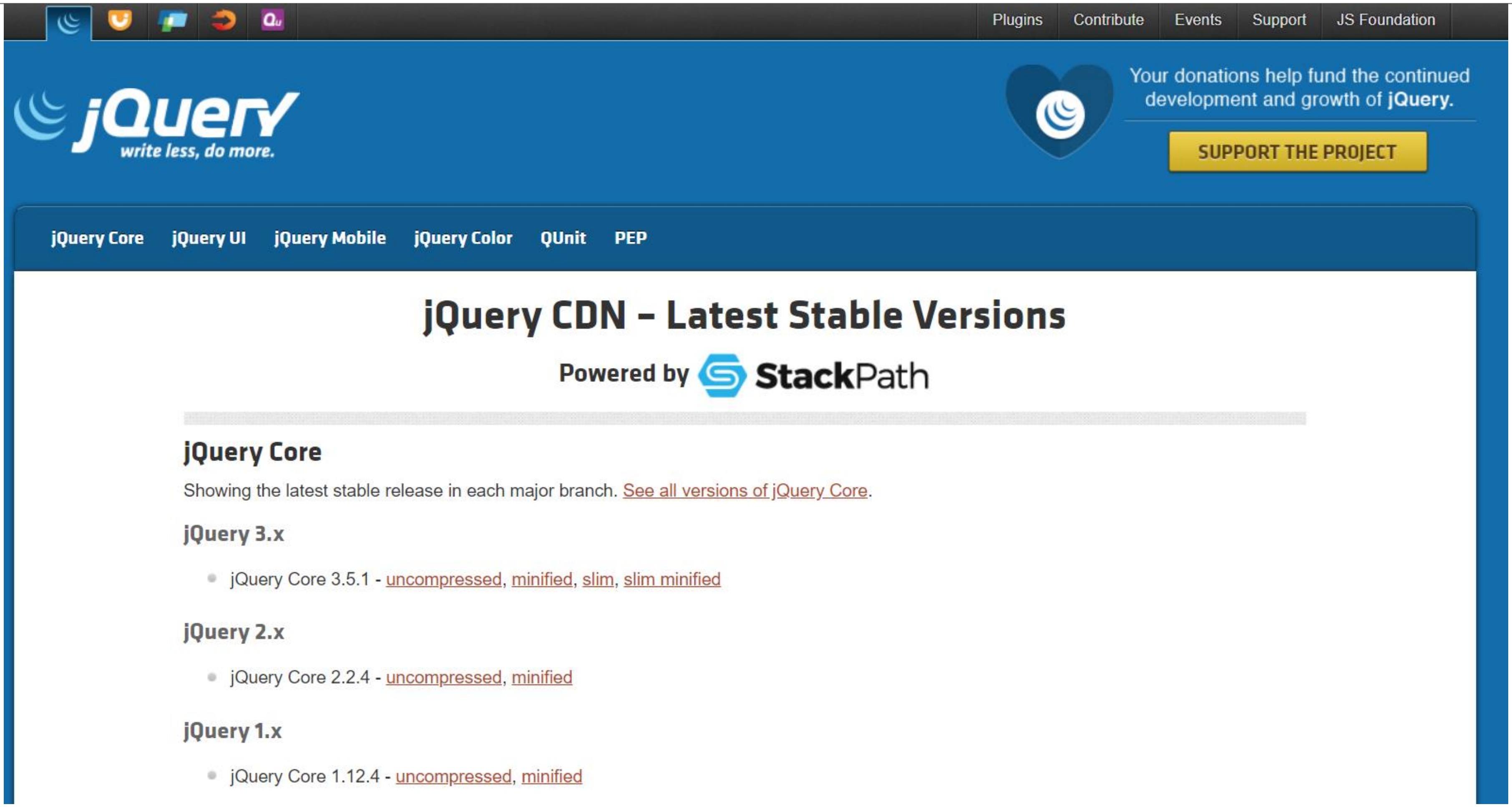
## jQuery Links

**Up until jQuery 1.11.1**, you could use the following URLs to get the latest version of jQuery:

- https://code.jquery.com/jquery-latest.min.js - jQuery hosted (minified)
- https://code.jquery.com/jquery-latest.js - jQuery hosted (uncompressed)
- https://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js - Google hosted (minified)
- https://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js - Google hosted (uncompressed)

# jQuery CDN – Latest Stable Versions

code.jQuery.com



## jQuery CDN – Latest Stable Versions

Powered by StackPath

### jQuery Core

Showing the latest stable release in each major branch. See all versions of jQuery Core.

#### jQuery 3.x

- jQuery Core 3.5.1 - uncompressed, minified, slim, slim minified

#### jQuery 2.x

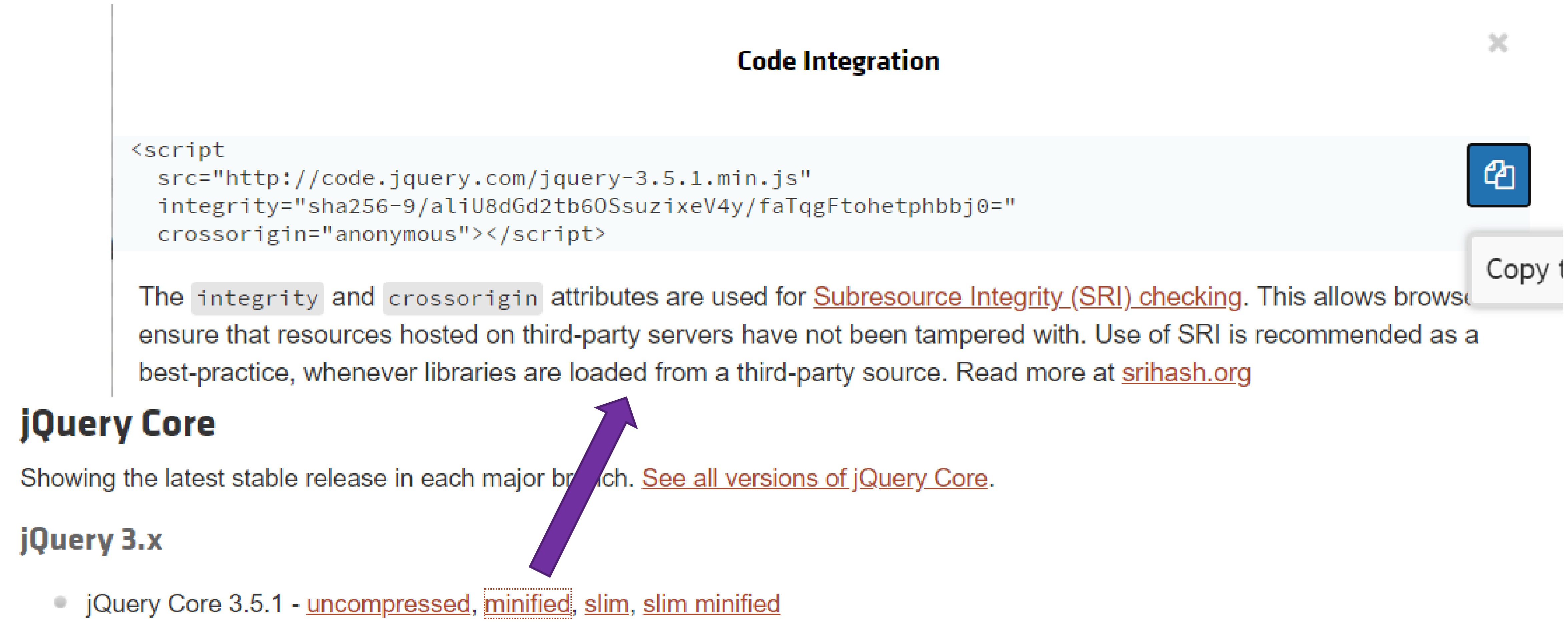
- jQuery Core 2.2.4 - uncompressed, minified

#### jQuery 1.x

- jQuery Core 1.12.4 - uncompressed, minified

## jQuery CDN – Latest Stable Versions

code.jQuery.com



Code Integration

```
<script
  src="http://code.jquery.com/jquery-3.5.1.min.js"
  integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
  crossorigin="anonymous"></script>
```

The `integrity` and `crossorigin` attributes are used for Subresource Integrity (SRI) checking. This allows brows ensure that resources hosted on third-party servers have not been tampered with. Use of SRI is recommended as a best-practice, whenever libraries are loaded from a third-party source. Read more at srihash.org

### jQuery Core

Showing the latest stable release in each major branch. See all versions of jQuery Core.

#### jQuery 3.x

- jQuery Core 3.5.1 - uncompressed, minified, slim, slim minified

## Keyboard Events

SECTION 10

## Keyboard Events

- JavaScript can monitor the keyboard through keyboard events. Each time a user presses a key on the keyboard, they generate a keyboard event, which is a lot like the mouse events we saw in Chapter 10.
- With mouse events, we used jQuery to determine where the cursor was when the event took place. With keyboard events, you can use jQuery to determine which key was pressed and then use that information in your code. For example, in this chapter we'll make a ball move left, right, up, or down when the user presses the left, right, up, or down arrow key.
- We'll use the keydown event, which is triggered whenever a user presses a key, and we'll use jQuery to add an event handler to the keydown event. That way, every time a keydown event occurs, our event handler function can find out which key was pressed and respond accordingly.

## Setting Up the HTML File

To begin, create a clean HTML file containing the following code and save it as *keyboard.html*.

```
<!DOCTYPE html>
<html>
   <head>
    <title>Keyboard input</title>
   </head>
   <body>
    <canvas id="canvas" width="400" height="400"></canvas>
    <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
    <script>
    // We'll fill this in next
    </script>
   </body>
</html>
```

## Adding the keydown Event Handler

- Now let’s add some JavaScript to respond to keydown events. Enter this code inside the empty <script> tags in your *keyboard.html* file.

```
$("body").keydown(function (event) {
console.log(event.keyCode);
});
```

- In the first line, we use the jQuery $ function to select the body element in our HTML and then call the keydown method. The argument to the keydown method is a function that will be called whenever a key is pressed. Information about the keydown event is passed in to the function through the event object.
- For this program, we want to know which key was pressed, and that information is stored in the event object as event.keyCode.

## Adding the keydown Event Handler

- Inside the function, we use console.log to output the event object’s keyCode property: a number representing the pressed key. Each key on your keyboard has a unique keycode. For example, the keycode for the spacebar is 32, and the left arrow is 37.
- Once you’ve edited your *keyboard.html* file, save it and then open it in a browser. Now open the console so you can see the output, and click in the main browser window to have JavaScript register your keypresses. Now, if you start pressing keys, the corresponding keycodes should be printed to the console.

## Adding the keydown Event Handler

- For example, if you type hi there, you should see the following output in the console:

```
72
73
32
84
72
69
82
69
```

- Every key you press has a different keycode. The H key is 72, the I key is 73, and so on.

## Using an Object to Convert Keycodes into Names

- To make it easier to work with keys, we’ll use an object to convert the keycodes into names so that the keypresses will be easier to recognize. In this next example, we create an object called keyNames, where the object keys are keycodes and the values are the names of those keys. Delete the JavaScript in *keyboard.html* and replace it with this:

## Using an Object to Convert Keycodes into Names

```
var keyNames = {
    32: "space",
    37: "left",
    38: "up",
    39: "right",
    40: "down"
};
$("body").keydown(function (event) {
❶    console.log(keyNames[event.keyCode]);
});
```

## Using an Object to Convert Keycodes into Names

- First, we create the keyNames object and fill it with the keycodes 32, 37, 38, 39, and 40. The keyNames object uses key-value pairs to match keycodes (such as 32, 37, and so on) with corresponding labels (such as "space" for the spacebar and "left" for the left arrow).
- We can then use this object to find out the name of a key based on its keycode. For example, to look up the keycode 32, enter keyNames[32]. That returns the string "space".

## Using an Object to Convert Keycodes into Names

- At ❶, we use the keyNames object in the keydown event handler to get the name of the key that was just pressed. If the event keycode referenced by event.keyCode matches one of the keys in the keyNames object, this function will log the name of that key. If no key matches, this code will log undefined.
- Load *keyboard.html* in your browser. Open the console, click in the main browser window, and try pressing a few keys. If you press one of the five keys in the keyName object (the arrow keys or spacebar), the program should print the name of the key. Otherwise, it will print undefined.

# Moving a Ball with the Keyboard

SECTION 11

## Moving a Ball with the Keyboard

- Now that we can determine which key is being pressed, we can write a program to use the keyboard to control the movement of a ball. Our program will draw a ball and move it to the right. Pressing the arrow keys will change the ball’s direction, and pressing the spacebar will stop it. If the ball goes off the edge of the canvas, it will wrap around to the opposite side. For example, if the ball goes off the right edge of the canvas, it will show up again on the left edge while continuing to move in the same direction, as shown in Figure 15-1.

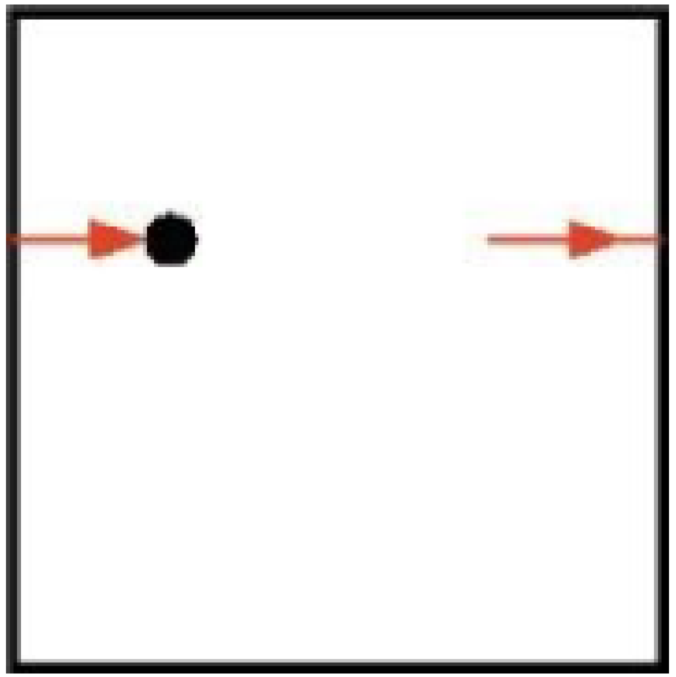


*Figure 15-1. If the ball moves off the right side of the canvas, it will reappear on the left.*

## Moving a Ball with the Keyboard

- We’ll use an object called keyActions to find out which key was pressed and then use that information to set the direction of the ball’s movement. We’ll use setInterval to continually update the ball’s position and redraw it at its new position.

## Setting Up the Canvas

- First we need to set up the canvas and the context object. Open *keyboard.html* and replace the JavaScript between the second set of <script> tags with this code:

```
var canvas = document.getElementById("canvas");
var ctx    = canvas.getContext("2d");
var width  = canvas.width;
var height = canvas.height;
```

## Setting Up the Canvas

- On the first line, we use document.getElementById to select the canvas element. On the second line, we call getContext on the canvas to get the context object. Then, in the var width and var height lines, we store the width and height of the canvas element in the variables width and height. This way, when we need the canvas dimensions, we can use these variables instead of having to enter the numbers manually. Now, if we choose to change the size of the canvas, we can simply edit the HTML, and the JavaScript code should still work.

## Defining the circle Function

- Next, we define the same circle function for the ball that we used in Chapter 14. Add this function after the code from the previous section:

```
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();
  }
};
```

## Creating the Ball Constructor

Now we'll create a Ball constructor. We'll use this constructor to create the moving ball object. We'll be using the same technique for moving this ball as we did in Chapter 14 — using the xSpeed and ySpeed properties to control the horizontal and vertical speed of the ball. Add this code after the circle function:

```
var Ball = function () {
   this.x = width / 2;
   this.y = height / 2;
   this.xSpeed = 5;
   this.ySpeed = 0;
};
```

## Creating the Ball Constructor

- We set the **x** and **y** values (the ball’s position) to **width** / 2 and **height** / 2 so that the ball will start at the center of the canvas. We also set **this.xSpeed** to 5 and **this.ySpeed** to 0. This means that the ball will start the animation by moving to the right (that is, with each animation step, its x position will increase by 5 pixels and its y position will stay the same).

## Defining the move Method

- In this section, we’ll define the move method. We’ll add this method to **Ball.prototype** to move the ball to a new location based on its current location, **xSpeed** and **ySpeed**. Add this method after the Ball constructor:

## Defining the move Method

```
    Ball.prototype.move = function () {
        this.x += this.xSpeed;
        this.y += this.ySpeed;
❶ if (this.x < 0) {
        this.x = width;
    } else if (this.x > width) {
        this.x = 0;
    } else if (this.y < 0) {
        this.y = height;
    } else if (this.y > height) {
        this.y = 0;
    }
```

## Defining the move Method

- First we update this.x and this.y using this.xSpeed and this.ySpeed, just as we did in Chapter 14 (see Moving the Ball). After that is the code for when the ball reaches the edge of the canvas.
- The if...else statement at ❶ checks the ball’s position to see if it has moved off the edge of the canvas. If it has, this code makes the ball wrap around to the other side of the canvas. For example, if the ball goes off the left edge of the canvas, it should reappear from the right side of the canvas. In other words, if this.x is less than 0, we set this.x to width, which places it at the very right edge of the canvas. The rest of the if...else statement deals with the other three edges of the canvas in a similar way.

## Defining the draw Method

- We’ll use the draw method to draw the ball. Add this after the definition of the move method:

```
Ball.prototype.draw = function () {
circle(this.x, this.y, 10, true);
};
```

- This method calls the circle function.
- It uses the ball’s x and y values to set the center of the ball, sets the radius to 10, and sets fillCircle to true. Figure 15-2 shows the resulting ball.

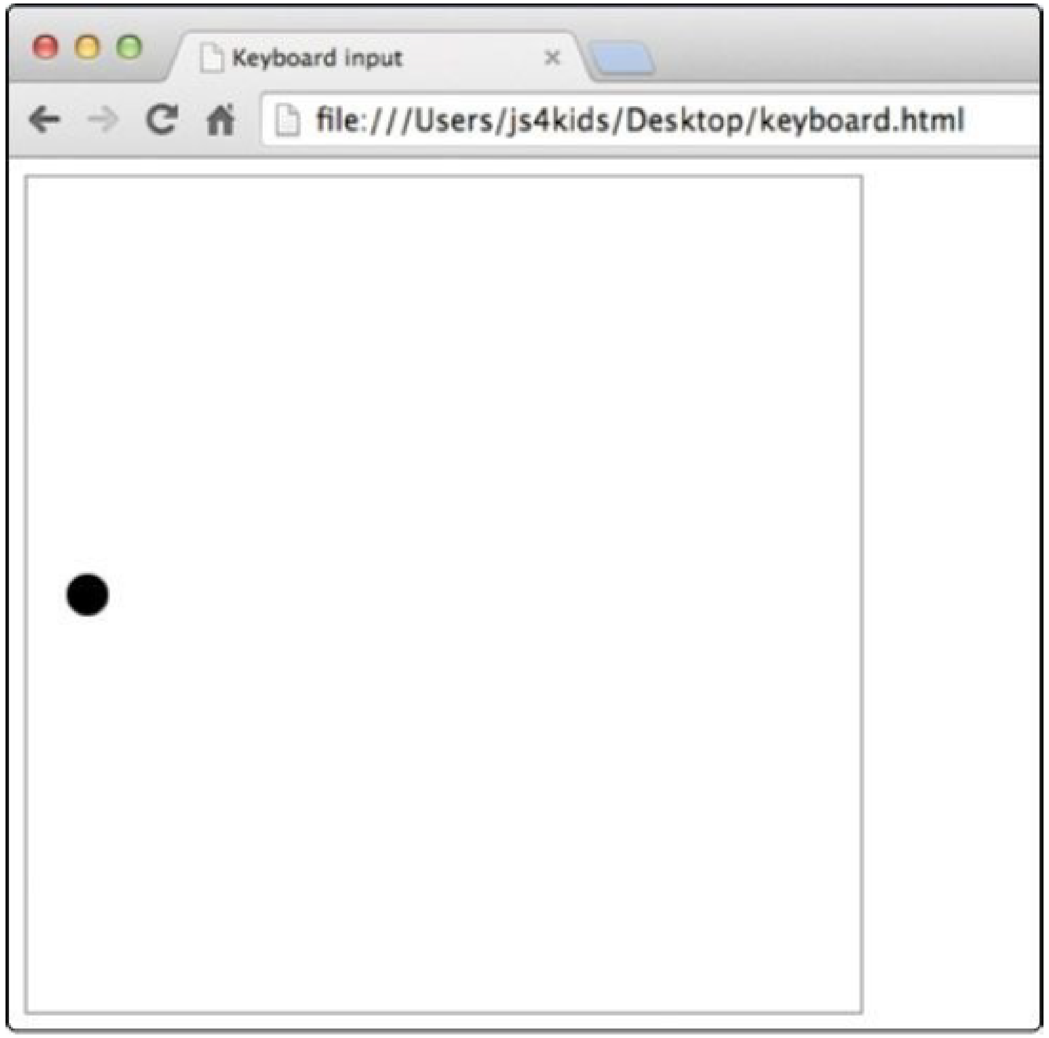


*Figure 15-2. The ball is a filled circle with a radius of 10.*

## Creating a setDirection Method

- Now we have to create a way to set the direction of the ball. We’ll do that with a method called **setDirection**. This method will be called by our **keydown** event handler, which you’ll see in the next section. The **keydown** handler will tell **setDirection** which key was pressed by passing it a string ("left", "up", "right", "down", or "stop"). Based on that string, **setDirection** will change the **xSpeed** and **ySpeed** properties of the ball to make it move in the direction that matches the keypress.
- For example, if the string "down" is passed, we set **this.xSpeed** to **0** and **this.ySpeed** to **5**. Add this code after the draw method:

# Creating a setDirection Method

```
Ball.prototype.setDirection = function (direction) {
    if (direction === "up") {
       this.xSpeed = 0;
       this.ySpeed = -5;
    } else if (direction === "down") {
       this.xSpeed = 0;
       this.ySpeed = 5;
    } else if (direction === "left") {
       this.xSpeed = -5;
       this.ySpeed = 0;
    } else if (direction === "right") {
       this.xSpeed = 5;
       this.ySpeed = 0;
    } else if (direction === "stop") {
       this.xSpeed = 0;
       this.ySpeed = 0;
    }
};
```

## Creating a setDirection Method

- The entire body of this method is one long if...else statement. The new direction is passed into the method as the direction argument. If direction is equal to "up", we set the ball’s xSpeed property to 0 and its ySpeed property to -5. The other directions are handled in the same way. Finally, if the direction is set to the string "stop", we set both this.xSpeed and this.ySpeed to 0, which means that the ball will stop moving.

# Reacting to the Keyboard

This next snippet of code creates a ball object using the Ball constructor, and it listens for keydown events in order to set the ball's direction. Add this code after the setDirection method:

```
❶ var ball = new Ball();
❷ var keyActions = {
     32: "stop",
     37: "left",
     38: "up",
     39: "right",
     40: "down"
  };
❸ $("body").keydown(function (event) {
❹ var direction = keyActions[event.keyCode];
❺     ball.setDirection(direction);
  });
```

## Reacting to the Keyboard

- At ➊, we create a ball object by calling new Ball(). At ➋ we create a keyActions object, which we'll use to convert keycodes to their corresponding direction. This object is the same as the keyNames object we created in Using an Object to Convert Keycodes into Names, except that for 32 (the keycode for the spacebar) we replace the label "space" with "stop" since we want the spacebar to stop the ball from moving.
- At ➌ we use the jQuery $ function to select the body element and then call the keydown method to listen for keydown events. The function passed to the keydown method is called every time a key is pressed.

## Reacting to the Keyboard

- Inside this function, we use keyAactions[event.keyCode] at ❹ to look up the label for the key that was pressed and assign that label to the direction variable. This sets the direction variable to a direction: "left" if the left arrow is pressed, "right" if the right arrow is pressed, "up" for the up arrow, "down" for the down arrow, and "stop" for the spacebar. If any other key is pressed, direction is set to undefined, and the animation won’t be affected.
- Finally, at ❺ we call the setDirection method on the ball object, passing the direction string. As you saw before, setDirection updates the ball’s xSpeed and ySpeed properties based on the new direction.

## Animating the Ball

- All we have left to do now is animate the ball. The following code should look familiar, since it’s quite similar to what we used in Chapter 14. It uses the setInterval function that we’ve seen in the animation code in previous chapters to update the ball’s position at regular intervals. Add this code after the code from the previous section:

## Animating the Ball

```
setInterval(function () {
 ctx.clearRect(0, 0, width, height);
 ball.draw();
 ball.move();
 ctx.strokeRect(0, 0, width, height);
}, 30);
```

## Animating the Ball

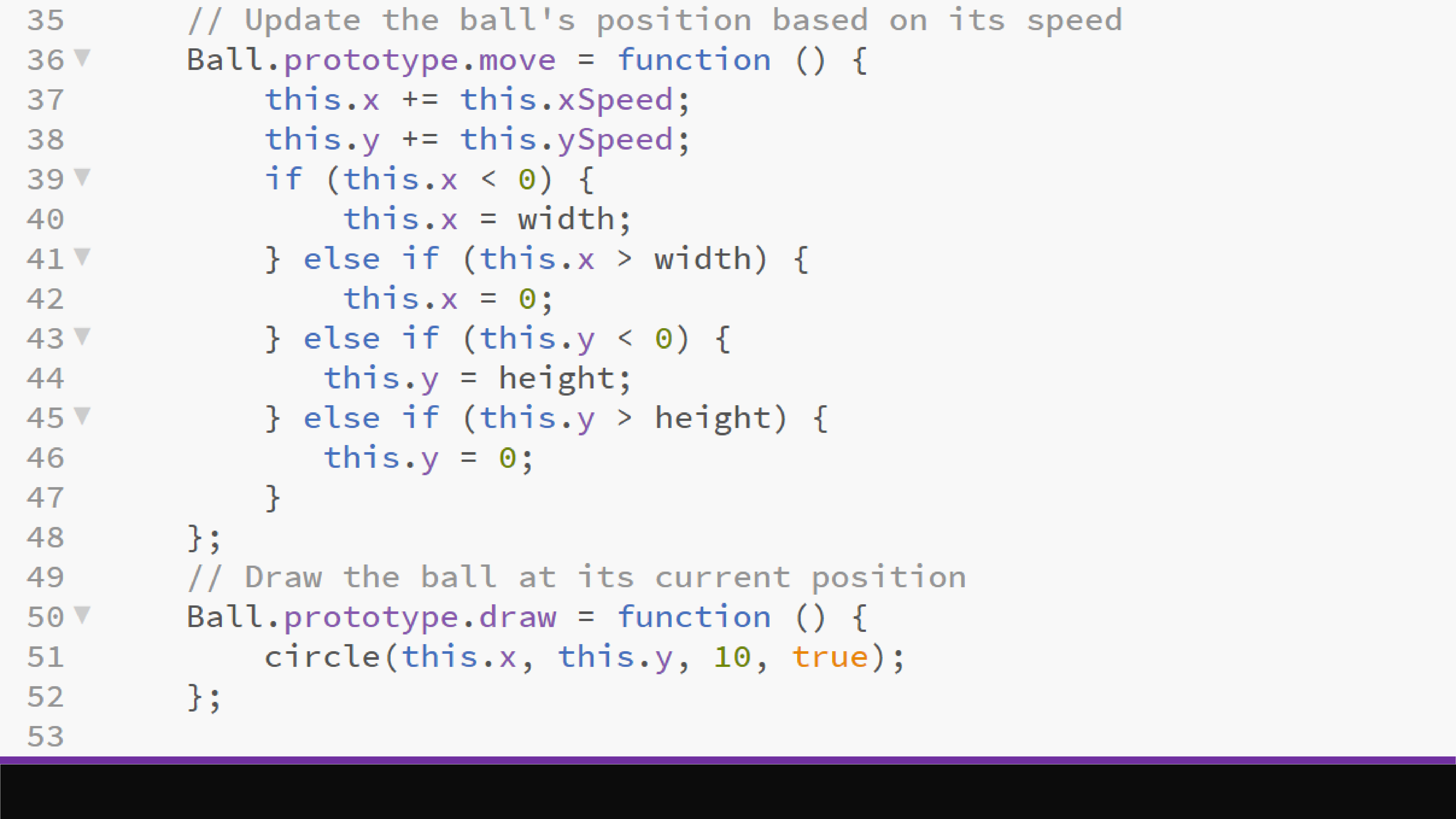
- We use **setInterval** to call our animation function every 30 milliseconds. The function first clears the entire canvas with **clearRect** and then calls the draw and move methods on the ball. As we've seen, the draw method simply draws a circle at the ball's current location, and the move method updates the ball's position based on its **xSpeed** and **ySpeed** properties. Finally, it draws a border with **strokeRect** so we can see the edge of the canvas.

# Putting It All Together

SECTION 12

```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>
```

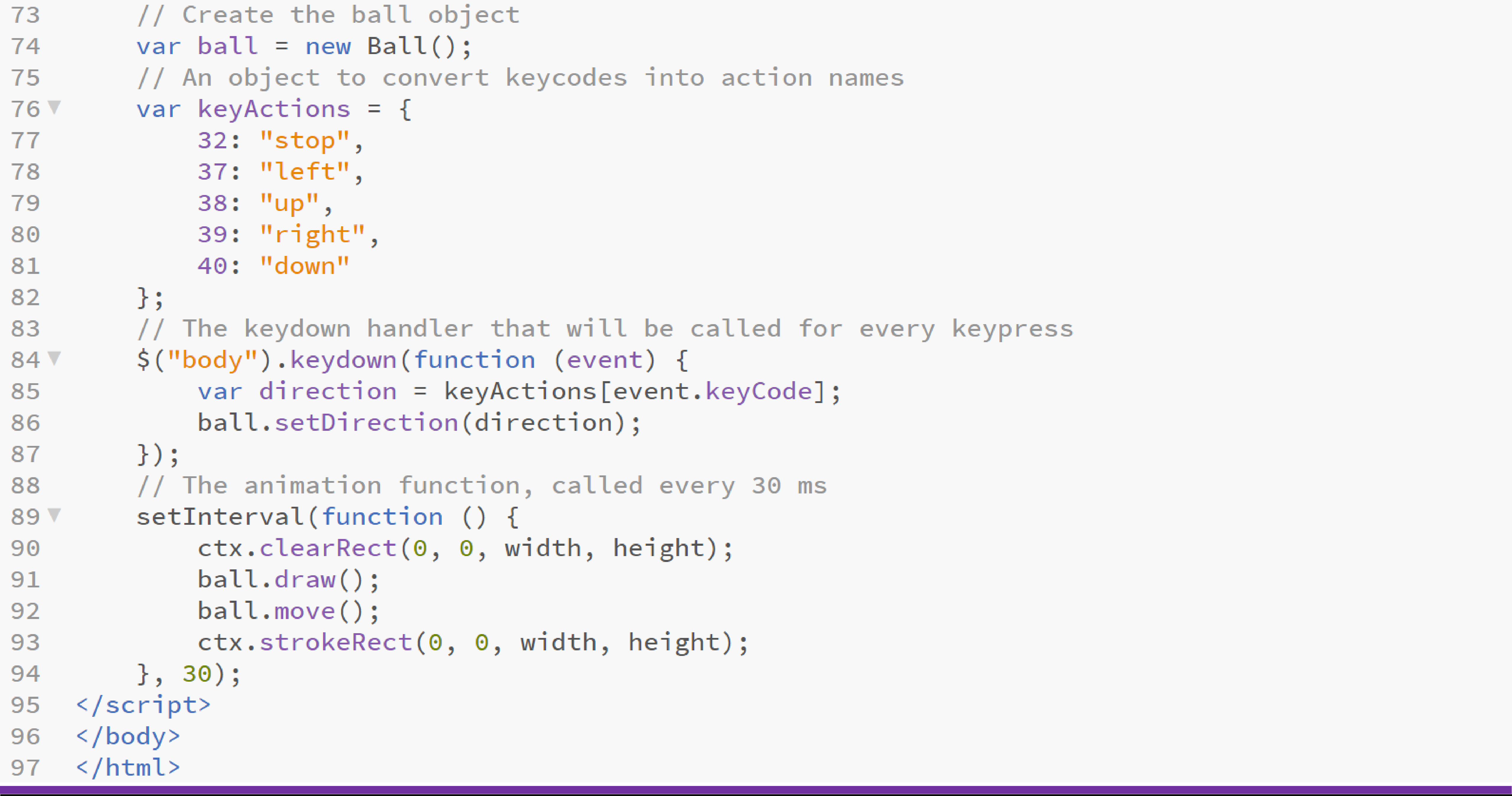
```
        var canvas = document.getElementById("canvas");
        var ctx = canvas.getContext("2d");
        var width = canvas.width;
        var height = canvas.height;
        var circle = function (x, y, radius, fillCircle) {
            ctx.beginPath();
            ctx.arc(x, y, radius, 0, Math.PI * 2, false);
            if (fillCircle) {
                ctx.fill();
            } else {
                ctx.stroke();
            }
        };
        // The Ball constructor
        var Ball = function () {
            this.x = width / 2;
            this.y = height / 2;
            this.xSpeed = 5;
            this.ySpeed = 0;
        };
```



```
    // Update the ball's position based on its speed
    Ball.prototype.move = function () {
        this.x += this.xSpeed;
        this.y += this.ySpeed;
        if (this.x < 0) {
            this.x = width;
        } else if (this.x > width) {
            this.x = 0;
        } else if (this.y < 0) {
           this.y = height;
        } else if (this.y > height) {
           this.y = 0;
        }
    };
    // Draw the ball at its current position
    Ball.prototype.draw = function () {
        circle(this.x, this.y, 10, true);
    };

```

```javascript
// Set the ball's direction based on a string
Ball.prototype.setDirection = function (direction) {
    if (direction === "up") {
        this.xSpeed = 0;
        this.ySpeed = -5;
    } else if (direction === "down") {
        this.xSpeed = 0;
        this.ySpeed = 5;
    } else if (direction === "left") {
        this.xSpeed = -5;
        this.ySpeed = 0;
    } else if (direction === "right") {
        this.xSpeed = 5;
        this.ySpeed = 0;
    } else if (direction === "stop") {
        this.xSpeed = 0;
        this.ySpeed = 0;
    }
};
```



```
    // Create the ball object
    var ball = new Ball();
    // An object to convert keycodes into action names
    var keyActions = {
        32: "stop",
        37: "left",
        38: "up",
        39: "right",
        40: "down"
    };
    // The keydown handler that will be called for every keypress
    $("body").keydown(function (event) {
        var direction = keyActions[event.keyCode];
        ball.setDirection(direction);
    });
    // The animation function, called every 30 ms
    setInterval(function () {
        ctx.clearRect(0, 0, width, height);
        ball.draw();
        ball.move();
        ctx.strokeRect(0, 0, width, height);
    }, 30);
</script>
</body>
</html>
```

## Running the Code

SECTION 13

## Running the Code

- Now our program is complete. When you run the program, you should see a black ball moving across the canvas to the right, as shown in Figure 15-3. When it reaches the right side of the canvas, it should wrap around to the left side and keep moving to the right.
- When you press the arrow keys, the ball should change direction, and pressing the spacebar should make the ball stop.

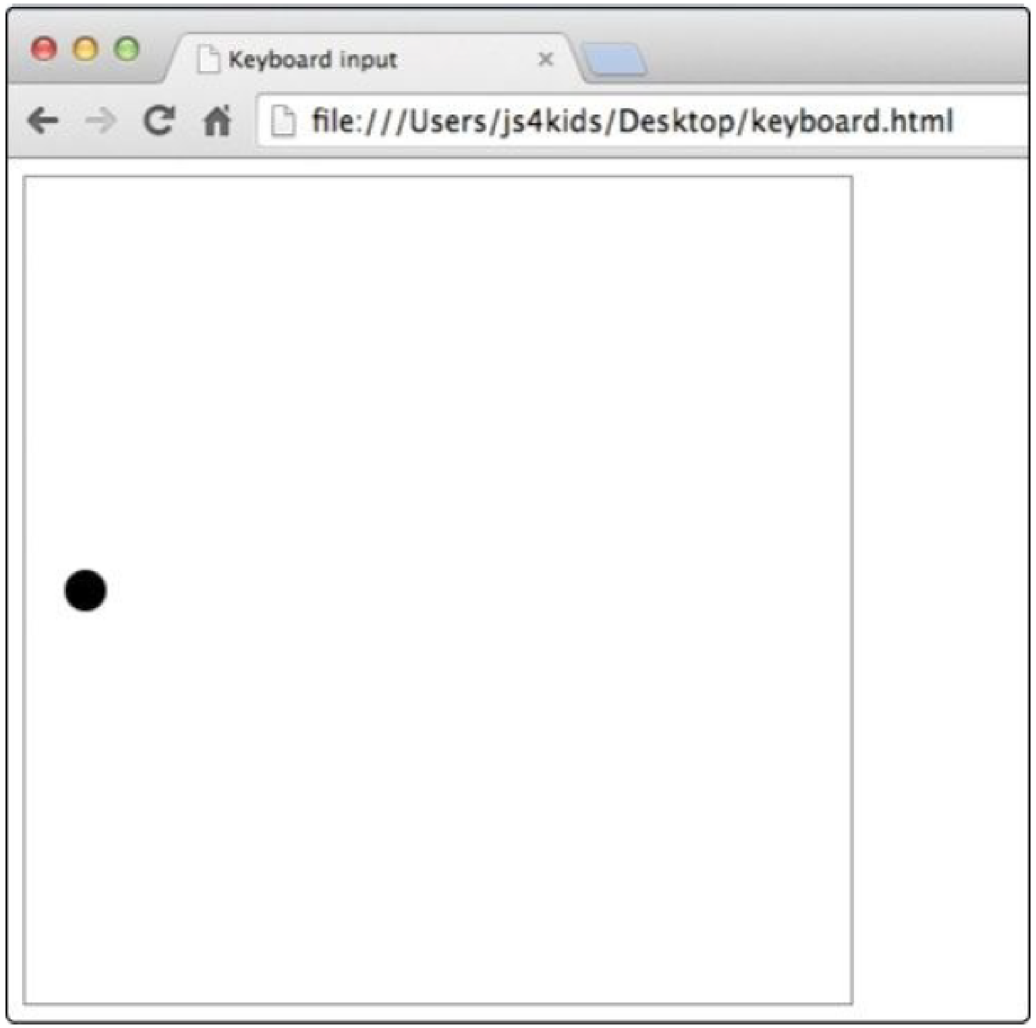


*Figure 15-3. A screenshot from the moving ball animation*

## Summary

SECTION 14

## Summary

- In this chapter, you learned how to make programs that react to keyboard events. We used this knowledge to create a moving ball, where the ball’s direction is set by the keyboard.
- Now that we can draw to the canvas, create animations, and update those animations based on user input, we can create a simple canvas-based game! In the next chapter, we’ll re-create the classic Snake game, combining everything we’ve learned up until now.