

---

# CHAPTER 18

---

## JAVASCRIPT / HTML PROGRAMMING

On the following pages you'll find the source files of some simple Internet pages:

- **ButtonDemo.html** is a page that contains a button to modify a text element.
- **PictureViewing.html** shows how to modify an `<img>` element with JavaScript.
- **DrawingDemoCanvas.html** shows how to draw and fill some graphical shapes like lines, rectangles, and arcs on a `<canvas>` element.
- **Ball.js** is an 'importable' file that specifies a JavaScript class to show a ball on a `<canvas>` element.
- **MovingBallsWithPointerCanvas.html** is an object-oriented application that uses the `Ball` class that is defined in **Ball.js**.

2018-02-19 File created.

2018-02-25 Last modification

Copyright © Kari Laitinen

**ButtonDemo.html – A page with a <button> element**

```
<!DOCTYPE html>
<html>
<head>
<title>ButtonDemo.html by Kari Laitinen</title>
<!--
  This is an HTML comment.
-->

<style>

.centered
{
  text-align: center;
  margin:      64px auto;    /* top and bottom margins are set,
                             right and left margins are automatic */
}

</style>

<script>

// The following JavaScript function is able to modify the text inside
// the HTML tags <h1> and </h1>. The HTML element is identified with its
// id "text_on_screen".

function change_text()
{
  var text_element = document.getElementById( "text_on_screen" ) ;

  if ( text_element.innerHTML == "Well done!" )
  {
    text_element.innerHTML = "Hello!" ;
  }
  else
  {
    text_element.innerHTML = "Well done!" ;
  }
}
</script>

</head>

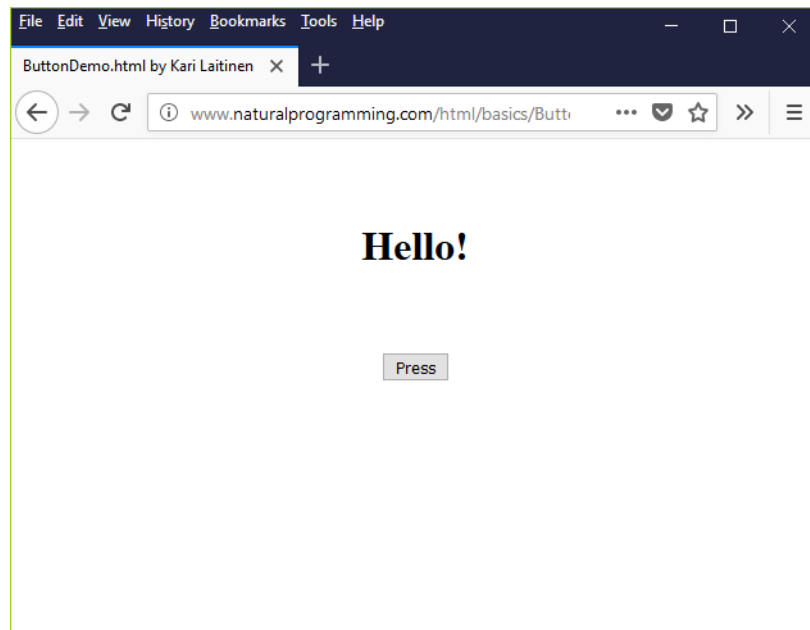
<body>

  <h1 id=text_on_screen class=centered>Hello!</h1>

  <div class=centered>
    <button onclick="change_text()">Press</button>
  </div>

</body>
</html>
```

**ButtonDemo.html - 1. Complete source code of the page.**



**ButtonDemo.html: The page shown in a rather small browser window..**

#### SOME BASIC TERMS RELATED TO HTML

- HTML tags are keywords (tag names) surrounded by angle brackets like `<html>`.
- Usually tags are used in pairs so that each start tag (opening tag) has a corresponding end tag (closing tag). For example, `<p>` is a start tag and `</p>` is an end tag.
- Tags are used to form HTML elements. The start tag is often used to name the element. For example, an `<html>` element is everything that is between the tags `<html>` and `</html>` including the tags.
- HTML elements can contain other elements that are called nested elements. Usually, the nested elements of an `<html>` element are a `<head>` element and a `<body>` element.
- Most elements can have attributes, like `id`, `class`, etc.

**PictureViewing.html – <img> and <button> elements in a page**

```

<!DOCTYPE html>
<html>
<head>
<title>PictureViewing.html by Kari Laitinen</title>

<style>
.centered
{
    text-align: center;
    margin:    16px auto;    /* top and bottom margins are set,
                             right and left margins are automatic */
}
</style>

<script>

var picture_file_names = [ "images/yellow_field_by_vincent_van_gogh.jpg",
                           "images/night_watch_by_rembbrandt.jpg",
                           "images/persistence_of_memory_by_dali.jpg",
                           "images/demoiselles_de_avignon_by_picasso.jpg",
                           "images/mona_lisa_by_leonardo.jpg" ] ;

var picture_index = 0 ;

function show_next_picture()
{
    picture_index ++ ;

    if ( picture_index >= picture_file_names.length )
    {
        picture_index = 0 ;
    }

    // First we get a reference to the img element in HTML code.

    var image_element = document.getElementById( "image_element_id" ) ;

    // Then we rewrite the src attribute of the img element.

    image_element.src = picture_file_names[ picture_index ] ;
}
</script>
</head>

<body>
<div class=centered>
    <button onclick="show_next_picture()">Next</button>
</div>

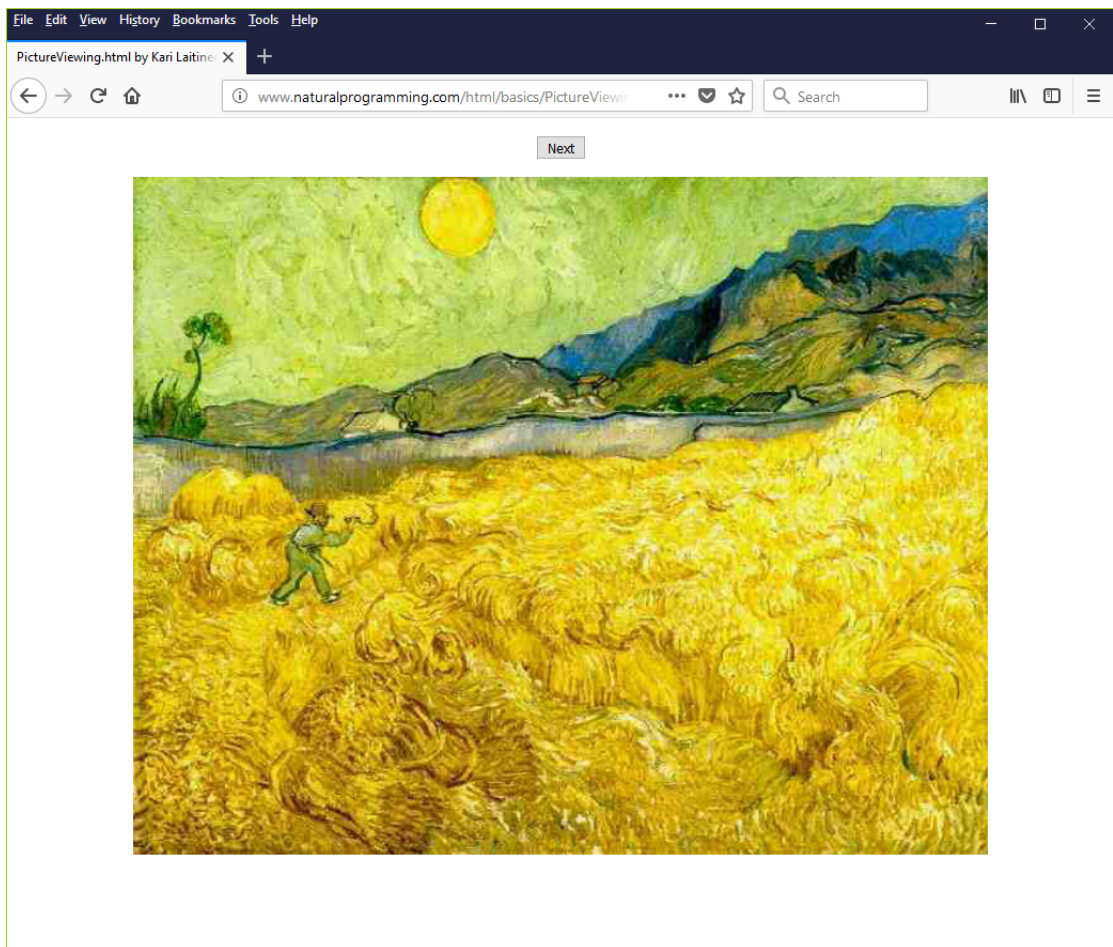
<div class=centered>
    
</div>
</body>
</html>

```

**PictureViewing.html - 1. Complete source code of the page.**

By studying the page **PictureViewing.html** you should note the following things about JavaScript.

- The array and the variable that are defined outside the function are global data that keep their values between calls to the function.
- An initialized array can be specified by writing the array elements inside brackets [ and ].



**PictureViewing.html - X. Here the button has not yet been pressed.**

**DrawingDemoCanvas – Drawing operations demonstrated**

Below you'll find out how to use some of the drawing methods that are available to draw or fill various graphical shapes in the 'drawing context'.

The first two parameters for the `fillRect()` method specify the upper left corner of the rectangle in the graphical coordinate system. The third parameter specifies the width and the fourth parameter specifies the height of the rectangle.

In this case the specified rectangle occupies the entire canvas.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>DrawingDemoCanvas.html &copy; Kari Laitinen</title>

<script type="text/javascript">

function draw_on_canvas()
{
    var canvas = document.getElementById( "canvas_for_drawings" ) ;
    var context = canvas.getContext( "2d" ) ;

    // We'll fill the entire canvas with light color, which overdraws
    // the previous drawings.

    context.fillStyle = "LightYellow" ;

    context.fillRect( 0, 0, canvas.width, canvas.height ) ;

    context.lineWidth    = 3 ;
    context.strokeStyle = "Blue" ; // Color for the first line.

    context.fillStyle = "Black" ;
    context.fillText( "Canvas size is " + canvas.width
        + " x " + canvas.height, 20, 20 ) ;

    context.beginPath() ;           // Start a new drawing path
    context.moveTo( 64, 128 ) ;     // Specifying a beginning for a line.
    context.lineTo( 512, 128 ) ;    // Specifying the end point of the line.
    context.stroke() ;              // This actually draws the line.

    context.fillStyle = "Cyan" ;
    context.fillRect( 64, 192, 128, 128 ) ; // Filled square with size 128x128

    context.strokeRect( 256, 192, 148, 128 ) ; // Hollow rectangle 148x128
    context.fillStyle = "Magenta" ;
    context.fillRect( 266, 202, 128, 108 ) ; // Filled rectangle 128x108
```

**DrawingDemoCanvas.html - 1: The first part of the page source code.**

```
context.beginPath() ; // New path for a 'ball' or circle.
context.arc( 512, 256, // The center point of the circle is (512, 256)
           64, 0, 2 * Math.PI, false ) ; // Radius is 64 points.
context.fillStyle = "Yellow" ;
context.fill() ; // Fill the defined path with filling color.
context.stroke() ; // Draw the outline of the path.

// The following statements draw a 'Pacman'

context.fillStyle = "LightGray" ;
context.strokeStyle = "Black" ;
context.beginPath() ;
context.moveTo( 704, 256 ) ;
context.arc( 704, 256,
           64, 0.25 * Math.PI, 1.75 * Math.PI, false ) ;
context.lineTo( 704, 256 ) ;
context.fill() ; // Fill the defined path with filling color.
context.stroke() ; // Draw the outline of the path.

// The following statements draw the 'missing' part of the Pacman.

context.beginPath() ;
context.moveTo( 768, 256 ) ;
context.arc( 768, 256,
           64, 0.25 * Math.PI, 1.75 * Math.PI, true ) ;
context.lineTo( 768, 256 ) ;
context.fill() ;
context.stroke() ;

context.beginPath() ; // New path to make a horizontal line.
context.moveTo( 512, 384 ) ;
context.lineTo( 768, 384 ) ;
context.stroke() ;
}
</script> <!-- End of JavaScript code. -->
```

**DrawingDemoCanvas.html - 2: The rest of function draw\_on\_canvas().**

```
<style type="text/css">

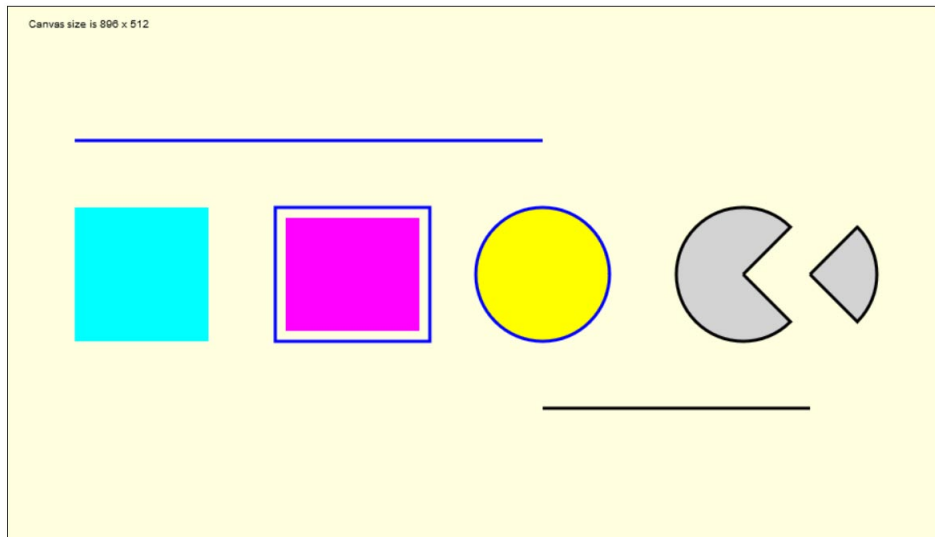
  #centered
  {
    width: 896px;      /* 1024 - 128 = 896 */
    height: 512px;
    margin: 30px auto; /* top and bottom margins are 30p;
                       right and left margins are automatic */
    border: 1px solid black;
  }

</style> <!-- End of CSS style definitions. -->
</head>

<body onload="draw_on_canvas()">

  <div id=centered>
    <canvas id=canvas_for_drawings width=896 height=512>
      </canvas>
    </div>
</body>
</html>
```

DrawingDemoCanvas.html - 3. The CSS definitions and <body> element of the page.



DrawingDemoCanvas.html - X. The canvas area of the page.

**Ball.js – An example of a JavaScript class**

```
class Ball
{
  constructor( given_center_point_x,
              given_center_point_y,
              given_color )
  {
    this.ball_center_point_x = given_center_point_x ;
    this.ball_center_point_y = given_center_point_y ;
    this.ball_color          = given_color ;

    this.ball_diameter      = 128 ;

    this.ball_is_activated  = false ;
  }

  activate_ball()
  {
    this.ball_is_activated = true ;
  }

  deactivate_ball()
  {
    this.ball_is_activated = false ;
  }

  get_ball_center_point_x()
  {
    return this.ball_center_point_x ;
  }

  get_ball_center_point_y()
  {
    return this.ball_center_point_y ;
  }

  get_ball_color()
  {
    return this.ball_color ;
  }

  set_ball_color( new_color )
  {
    this.ball_color = new_color ;
  }

  set_diameter( new_diameter )
  {
    if ( new_diameter > 3 )
    {
      this.ball_diameter = new_diameter ;
    }
  }
}
```

With the **this** keyword it is possible to create data members, or to refer to data members inside objects.

**Ball.js - 1:** The constructor and some 'getter' and 'setter' methods for Ball objects.

When methods are written inside JavaScript classes, no keywords are needed. You just write the method name, and its formal parameters inside parentheses.

```

> move_right()
  {
    this.ball_center_point_x += 3 ;
  }

move_left()
{
  this.ball_center_point_x -= 3 ;
}

move_up()
{
  this.ball_center_point_y -= 3 ;
}

move_down()
{
  this.ball_center_point_y += 3 ;
}

move_this_ball( movement_in_direction_x, movement_in_direction_y )
{
  this.ball_center_point_x += movement_in_direction_x ;
  this.ball_center_point_y += movement_in_direction_y ;
}

move_to_position( new_center_point_x, new_center_point_y )
{
  this.ball_center_point_x = new_center_point_x ;
  this.ball_center_point_y = new_center_point_y ;
}

```

With method `move_this_ball()` it is possible to move a **Ball** object in relation to its current position. Method `move_to_position()` moves a **Ball** to a completely new position.

## Ball.js - 2: Methods for moving Ball objects.

Objects that represent graphical shapes on the screen should be such that they 'know' their place on the screen, they know their size, color, and other features, and they can draw themselves. As `Ball` objects contain the necessary data members, they can be asked to draw themselves by calling the `draw()` method.

With the `contains_point()` method a `Ball` object can be 'asked' whether or not a certain point belongs to the ball area.

```
contains_point( given_point_x, given_point_y )
{
    var ball_radius = this.ball_diameter / 2 ;

    // Here we use the Pythagorean theorem to calculate the distance
    // from the given point to the center point of the ball.
    // See the note at the end of this file.

    var distance_from_given_point_to_ball_center =

        Math.sqrt(

            Math.pow( this.ball_center_point_x - given_point_x, 2 ) +
            Math.pow( this.ball_center_point_y - given_point_y, 2 ) ) ;

    return ( distance_from_given_point_to_ball_center <= ball_radius ) ;
}

draw( context )
{
    // We'll first save the current drawing context so that we'll
    // not disturb any drawing operations made by other methods.

    context.save() ;

    context.fillStyle = this.ball_color ;

    context.beginPath() ;
    context.arc( this.ball_center_point_x, this.ball_center_point_y,
                this.ball_diameter / 2, 0, 2 * Math.PI, true )
    context.closePath() ;
    context.fill() ;

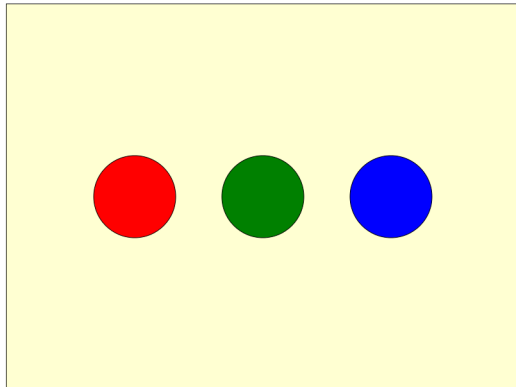
    if ( this.ball_is_activated == true )
    {
        context.lineWidth = 6 ; // Thick edge for an activated ball.
    }

    context.stroke() ; // Draw the outline of the ball.

    context.restore() ; // Restore the saved drawing context.
}

} // End of the definition of the Ball class.
```

### Ball.js - 3. Methods `contains_point()` and `draw()` of class `Ball`.

**MovingBallsWithPointerCanvas.html – An object-oriented application**

When the page is loaded, there are three Ball objects visible on the screen.

The user of the page can move the balls with the mouse.

**MovingBallsWithPointerCanvas.html - X. The <canvas> element of the page.**

The constructor of class **Ball** is executed when these statements are processed by the browser. The supplied parameters become data members inside the **Ball** objects, and each object will have its own set of data members, and the objects can be manipulated with the methods that are written inside **Ball.js**.

```

<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>MovingBallsWithPointerCanvas.html &copy; Kari Laitinen</title>

<script src="Ball.js"></script>

<script>

// Now the 'main' program begins.

var first_ball = new Ball( 200, 300, "red" ) ;
var second_ball = new Ball( 400, 300, "green" ) ;
var third_ball = new Ball( 600, 300, "blue" ) ;

var ball_being_moved = null ;

// In this program we speak about pointer positions instead
// of mouse or cursor positions because it is possible
// that the balls can be pointed with some other devices
// than a mouse.

var previous_pointer_position_x = 0 ;
var previous_pointer_position_y = 0 ;

```

**MovingBallsWithPointerCanvas.html - 1. The public data items.**

The properties `offsetX` and `offsetY` contain information about the clicked point. The values contained in these properties are relative to the upper left corner of the canvas area.

All major browsers seem to have these properties nowadays.

```
function on_mouse_down( event )
{
    // The mouse or some other pointing device was
    // pressed down in the canvas area.

    var pointer_position_x = event.offsetX ;
    var pointer_position_y = event.offsetY ;

    // Now we know which point was pointed. We will 'ask' all the
    // Ball objects whether or not the pressed point is inside
    // the area of the ball in question.

    if ( first_ball.contains_point( pointer_position_x,
                                    pointer_position_y ) )
    {
        ball_being_moved = first_ball ;
        ball_being_moved.activate_ball() ;
    }
    else if ( second_ball.contains_point( pointer_position_x,
                                          pointer_position_y ) )
    {
        ball_being_moved = second_ball ;
        ball_being_moved.activate_ball() ;
    }
    else if ( third_ball.contains_point( pointer_position_x,
                                         pointer_position_y ) )
    {
        ball_being_moved = third_ball ;
        ball_being_moved.activate_ball() ;
    }

    previous_pointer_position_x = pointer_position_x ;
    previous_pointer_position_y = pointer_position_y ;

    draw_on_canvas() ;
}
```

The value of `ball_being_moved` will remain `null` if none of the balls contained the pointed position.

## MovingBallsWithPointerCanvas.html - 2: Handling pressings of mouse buttons.

```
function on_mouse_move( event )
{
  if ( ball_being_moved != null )
  {
    var new_pointer_position_x = event.offsetX ;
    var new_pointer_position_y = event.offsetY ;

    var pointer_movement_x = new_pointer_position_x
                            - previous_pointer_position_x ;

    var pointer_movement_y = new_pointer_position_y
                            - previous_pointer_position_y ;

    previous_pointer_position_x = new_pointer_position_x ;
    previous_pointer_position_y = new_pointer_position_y ;

    ball_being_moved.move_this_ball( pointer_movement_x,
                                     pointer_movement_y ) ;

    draw_on_canvas() ;
  }
}

function on_mouse_up( event )
{
  if ( ball_being_moved != null )
  {
    ball_being_moved.deactivate_ball() ;
    ball_being_moved = null ;

    draw_on_canvas() ;
  }
}
```

`on_mouse_move()` will be called many times when mouse is moved. Here we calculate how much the **Ball** object was moved since the previous call to this method. The method `move_this_ball()` is called to actually change the position of the ball on the screen.

When the mouse button is released, the movement operation is finished. When a **Ball** object is deactivated, it means that it will no longer be drawn as an activated ball.

Drawing the balls on the canvas is simple as the `Ball` objects 'know' how to draw themselves.

```
function draw_on_canvas()
{
  var canvas = document.getElementById( "canvas_for_balls" ) ;
  var context = canvas.getContext("2d") ;

  context.fillStyle = "LightYellow" ;
  context.fillRect( 0, 0, canvas.width, canvas.height ) ;

  first_ball.draw( context ) ;
  second_ball.draw( context ) ;
  third_ball.draw( context ) ;
}

</script>

<style type="text/css">

  #centered
  {
    width: 800px;
    height: 600px;
    margin: 30px auto;    /* top and bottom margins are 30p;
                          right and left margins are automatic */
    border: 1px solid black;
  }

</style>
</head>

<body onload="draw_on_canvas()">

  <div id=centered>
    <canvas id=canvas_for_balls
      width=800 height=600
      onmousedown = "on_mouse_down( event )"
      onmousemove = "on_mouse_move( event )"
      onmouseup   = "on_mouse_up( event )" >
    </canvas>

  </div>

</body>
</html>
```

Here we specify which JavaScript methods will be called automatically when mouse-related events take place inside the `<canvas>` element.

**MovingBallsWithPointerCanvas.html - 4. The rest of the page source code.**

