CHAPTER 18

JAVASCRIPT / HTML CANVAS PROGRAMMING

On the following pages you'll find the source files of some Internet pages that contain `<canvas>` elements:

- **HelloCanvas.html** is the first example which shows how we can draw text.
- **DrawingDemoCanvas.html** shows how to draw and fill some graphical shapes like lines, rectangles, arcs, and polygons.
- **KeyboardInputCanvas.html** shows how to specify functions that will be called automatically when the keys of the keyboard are used.
- **AnimationDemoCanvas.html** is an example in which the `draw_on_canvas()` function is executed once during every second.
- **FlyingArrowCanvas.html** shows how we can translate, rotate, and scale the graphical coordinates.
- **Ball.js** is an 'importable' file that specifies a JavaScript class.
- **MovingBallsWithPointerCanvas.html** is an object-oriented application that uses the `Ball` class that is defined in **Ball.js**.

2014-04-10 File created.
2017-03-20 Last modification

Copyright © Kari Laitinen

HelloCanvas.html – A simple page with a <canvas> element

Function **`draw_on_canvas()`** will be called automatically when this page is loaded. At the beginning of the function a reference to the **`<canvas>`** element is obtained by using the id **`hello_canvas`**.

A reference to a context object is obtained by calling the **`getContext()`** method. We'll have these two lines practically in all sample pages which contain a **`<canvas>`** element.

```
<!DOCTYPE html>  <!-- This is an HTML 5 document. -->
<html>
<head>
   <meta charset="ISO-8859-1">
   <title>A CANVAS THAT SAYS "Hello ..."</title>

   <script type="text/javascript">

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

         // We'll fill the entire canvas with light color.

         context.fillStyle = "rgb( 255, 255, 210 )" ;  // light yellow

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

         context.fillStyle = "rgb( 0, 0, 0 )" ; // black filling color

         //  The following method call draws a text string to
         //  a position that is located 80 pixels to the right
         //  and 100 pixels down from the upper left corner
         //  of the canvas

         context.fillText( "Hello. I am an HTML 5 canvas.", 80, 100 ) ;

         context.fillText( "The coordinates of this line are (80,150).",
                           80, 150 ) ;
      }

   </script> <!-- End of JavaScript code. -->
```

HelloCanvas.html - 1: The first part of the HTML code of the page.

```
   <!-- Here begin the the style definitions made with CSS.
        With these definitions we are able to center the <div>
        element on the screen. -->
   <style type="text/css">

      #centered
      {
         width:  400px;
         height: 250px;
         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()">

   <!-- The HTML body of this document has a single <div> element
        that is 'connected' to the CSS style definition with the id
        "centered". Inside the <div> element we have the <canvas>
        element. -->

   <div id=centered>
      <canvas id=hello_canvas width=400 height=250>
      </canvas>
   </div>
</body>
</html>
```

HelloCanvas.html - 2. The last part of the HTML code of the page.

Hello. I am an HTML 5 canvas.

The coordinates of this line are (80,150).

HelloCanvas.html: The canvas of the page.

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
```

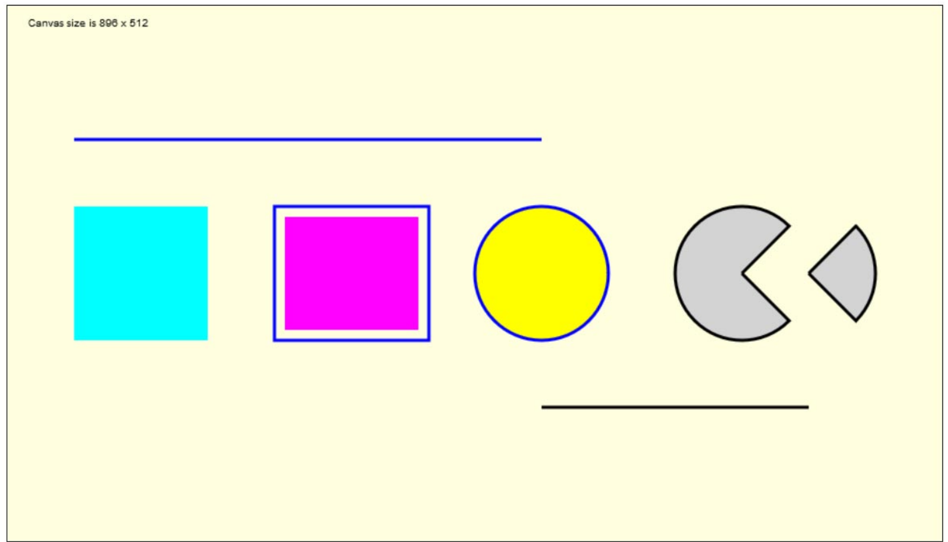
DrawingDemoCanvas.html - 1: The first part of the page source code.

```
   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.

KeyboardInputCanvas.html – Reacting to key pressings

This sample page demonstrates how we can react to keyboard events such as pressings and releases of a key. As this program produces output to the console window of the browser, you can use this to test keyboard behaviour.

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

<!-- Next we import JavaScript code that allows us to use a function
     named sprintf() which works in the same way as corresponding
     functions in the C language or the String.format() method of Java.
-->

<script src="../basics/importables/sprintf.js"></script>

<script type="text/javascript">

var large_font   =  "bold 24px monospace" ;

var code_of_last_pressed_key  =  63 ; // Code of question mark ?

// In the following functions toString( 16 ) is used to convert
// the received key codes into hexadecimal values.

function on_key_down( event )
{
   code_of_last_pressed_key = event.which ;

   console.log( "\n on_key_down()  " + event.which.toString( 16 ) ) ;
   draw_on_canvas() ;
}

function on_key_press( event )
{
   console.log( "\n on_key_press() " + event.which.toString( 16 ) ) ;
   draw_on_canvas() ;
}
```

Function `on_key_down()` will be called automatically when a key of the keyboard is pressed down. The method receives an event object as a parameter. This object contains data related to the key pressing. With property `which` it is possible to find out the code of the used key.

There are two functions that will be called when key pressings take place. The difference between these functions is that `on_key_press()` will be called mainly for visible character keys, and not for keys such as the Ctrl key.

KeyboardInputCanvas.java - 1: Reacting to key pressings.

```
function on_key_up( event )
{
   console.log( "\n on_key_up()    " + event.which.toString( 16 ) ) ;
   draw_on_canvas() ;
}


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

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

   context.fillStyle = "rgb(  200, 220, 255 )" ; // Light blueish
   context.fillRect( 0, 0, canvas.width, canvas.height ) ;

   context.fillStyle = "black" ;
   context.font = large_font ;

   // The sprintf() function returns a string in which
   //    - in place of %c the key code is shown as a character,
   //    - in place of %X the key code is shown in hexadecimal form
   //    - in place of %d the key code is shown in decimal form

   context.fillText( sprintf( "Last pressed key:  %c  %X  %d",
                              code_of_last_pressed_key,
                              code_of_last_pressed_key,
                              code_of_last_pressed_key ),
                     100, 200 ) ;

   if ( code_of_last_pressed_key  ==  112 )
   {
      context.fillText( "You pressed the F1 key", 100, 250 ) ;
   }
   else if ( code_of_last_pressed_key  ==  38 )
   {
      context.fillText( "You pressed the Arrow Up key", 100, 250 ) ;
   }
   else if ( code_of_last_pressed_key  ==  40 )
   {
      context.fillText( "You pressed the Arrow Down key", 100, 250 ) ;
   }
}

</script>
```

This program shows an extra line of text in the case of some special keys. Here we are using the numerical key codes of the keys. For example, 112 is the numerical code of the F1 key of the PC keyboad.

KeyboardInputCanvas.java - 2. The rest of the JavaScript code of the page.

Here is specified which JavaScript methods/ functions will be called automatically when keyboard events take place. Note that the methods that will react to key pressings are connected to the **<body>** element, not the **<canvas>** element.

```
<style type="text/css">
   #centered
   {
      width: 600px;
      height: 400px;
      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()"
      onkeydown  = "on_key_down( event )"
      onkeypress = "on_key_press( event )"
      onkeyup    = "on_key_up( event )">

   <div id=centered>
      <canvas id=keyboard_input_canvas  width=600 height=400>
      </canvas>

   </div>

</body>
</html>
```

KeyboardInputCanvas.java - X - 3. The CSS definitions and <body> element of the page.

```
Last pressed key:  A  41  65
```

This program shows the character and the numerical code associated with the used key. The numerical code is shown both in hexadecimal and decimal numbering systems. Function **sprintf()** can be used to insert numerical values into character strings. For example, the format specifier **%X** specifies that the value is shown in hexadecimal form.

KeyboardInputCanvas.java - X. Here the program has reacted to the pressing of key ’A’.

AnimationDemoCanvas.html – a blinking ball animated

This public variable is used to control the animation on the page. A ball is drawn only when the value of the variable **`ball_must_be_shown`** is **`true`**. Always after the ball has been drawn once, the value of this data field is set to **`false`**. This way every second call to the **`draw_on_canvas()`** method draws the ball.

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

<script type="text/javascript">

var ball_center_point_x = 300 ;
var ball_center_point_y = 240 ;

var current_ball_color = "cyan" ;
var ball_must_be_shown  =  true ;

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

   context.fillStyle = "rgb( 255, 255, 210 )" ;  // light yellow
   context.fillRect( 0, 0, canvas.width, canvas.height ) ;

   if ( ball_must_be_shown  ==  true )
   {
      context.fillStyle = current_ball_color ;

      context.beginPath() ;
      context.arc( ball_center_point_x, ball_center_point_y,
                   50, 0, 2 * Math.PI, true )
      context.closePath() ;
      context.stroke() ;
      context.fill() ;

      ball_must_be_shown  =  false ;
   }
   else
   {
      ball_must_be_shown  =  true ;
   }

   setTimeout( "draw_on_canvas()", 1000 ) ;
}

</script>
```

With function **`setTimeout()`** we specify that this function **`draw_on_canvas()`** will be automatically called again after 1000 milliseconds, i.e., after one second.

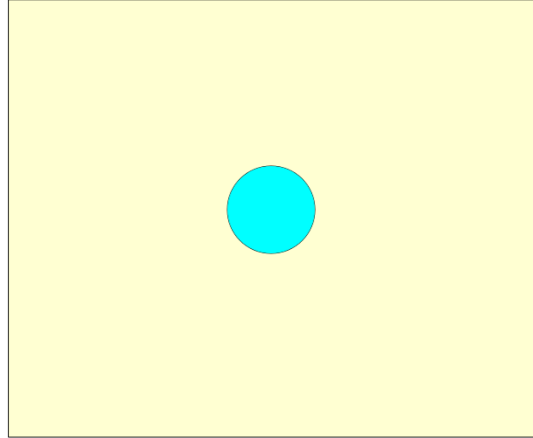
This has the effect that **`draw_on_canvas()`** is executed repeatedly once in every second. As the ball is not drawn during every execution, it seems to blink on the screen.

AnimationDemoCanvas.html - 1: Function draw_on_canvas() of the page.

As the animation is based entirely on the use of JavaScript, nothing specific to animation is written in the CSS part or into the body of the page.

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

   #centered
   {
      width: 600px;
      height: 500px;
      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=animation_demo_canvas
         width=600 height=500>
      </canvas>
   </div>

</body>
</html>
```

AnimationDemoCanvas.html - 2. The CSS definitions and the <body> element.



AnimationDemoCanvas.html - X. The page blinks a ball on the screen.

FlyingArrowCanvas.html – More advanced drawing operations

The page whose source code is presented on this and the following pages contains a `<canvas>` element on which arrow shapes are created with this function. (Note that it is possible to have functions inside functions in JavaScript.) When the coordinate system of the canvas is altered between drawing operations, the arrow shapes are created in different positions to the screen.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>FlyingArrowCanvas.html &copy; Kari Laitinen</title>
<script type="text/javascript">

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

   context.fillStyle = "rgb( 200, 200, 200 )" ;  // light gray background
   context.fillRect( 0, 0, canvas.width, canvas.height ) ;

   context.fillStyle = "purple" ;

   //  The following is a function inside function that will
   //  be used to re-create the arrow path after the coordinate
   //  system has been altered.

   function create_arrow_path()
   {
      //  The arrow coordinates are selected so that Point (0, 0)
      //  is at the tip of the arrow, and the arrow points upwards.

      var  arrow_shape_coordinates_x  =
             [ 0,  15,   5,   5,  15,   0, -15,  -5,  -5, -15 ] ;

      var  arrow_shape_coordinates_y  =
             [ 0,  40,  30, 120, 160, 130, 160, 120,  30,  40 ] ;

      context.beginPath() ;  // Begin a path for arrow shape.

      context.moveTo( arrow_shape_coordinates_x[ 0 ],
                      arrow_shape_coordinates_y[ 0 ] ) ;

      for ( var point_index  =  1 ;
                point_index  <  arrow_shape_coordinates_x.length ;
                point_index  ++ )
      {
         context.lineTo( arrow_shape_coordinates_x[ point_index ],
                         arrow_shape_coordinates_y[ point_index ] ) ;
      }

      context.closePath() ;
   }
```

FlyingArrowCanvas.html - 1: The first part of function draw_on_canvas().

Method **translate()** moves the zero point of the coordinate system to another position on the canvas. By default the zero point, i.e., the point in which both coordinate values are zeroes, is the upper left corner of the canvas.

Method **rotate()** is able to rotate the coodinate system. When a graphical shape is re-created in the rotated coordinate system, its appearance is changed according to the rotation.

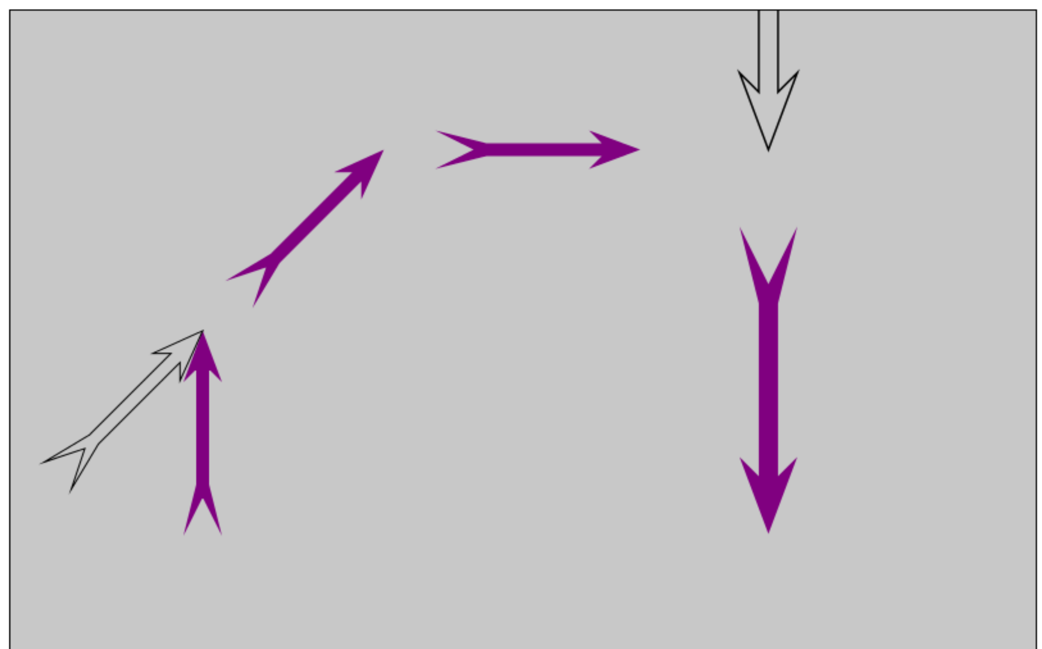
```
   context.translate( 150, 250 ) ;  // arrow tip to point (150, 250 )
   create_arrow_path() ;
   context.fill() ;                 // draw solid arrow

   context.rotate( Math.PI / 4 ) ;  // 45 degrees clockwise
   create_arrow_path() ;            // re-create arrow path
   context.stroke() ;               // draw a hollow arrow

   context.translate( 0, -200 ) ;   // flying "up" 200 points
   create_arrow_path() ;
   context.fill() ;

   context.rotate( Math.PI / 4 ) ;  // 45 degrees clockwise
   context.translate( 0, -200 ) ;   // flying "up" (i.e. to the right)
   create_arrow_path() ;
   context.fill() ;

   context.translate( 0, -100 ) ;   // flying "up" 100 points
   context.rotate( Math.PI / 2 ) ;  // 90 degrees clockwise
   context.scale( 1.5, 1.5 ) ;      // magnify everything by 1.5
   create_arrow_path() ;
   context.stroke() ;               // draw a hollow arrow

   context.translate( 0, -200 ) ;   // flying "up" (i.e. down) 300 points
   create_arrow_path() ;
   context.fill() ;
}

</script> <!-- End of JavaScript code. -->
```

With method **scale()** it is possible to scale the coordinate system. When arrow shapes will be re-created after this statement has been executed, the arrows shall be 1.5 times bigger than the earlier arrows.

FlyingArrowCanvas.html - 2: The rest of function draw_on_canvas().

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

   #centered
   {
      width:  800px;
      height: 500px;
      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=flying_arrow_canvas width=800 height=500>
      </canvas>
   </div>
</body>
</html>
```

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



FlyingArrowCanvas.html - X. The content of the <canvas> element in 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.

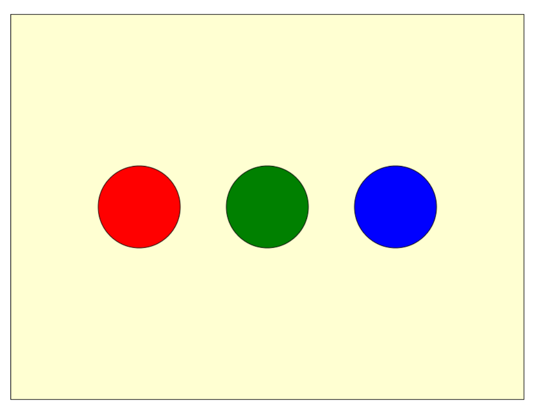
Objects that represent graphical shapes on the screen should be such that they 'know' their place on the screen, they know they size, color, and other features, and they can draw themselves. As `Ball` objects contain the necessary data members, they can be asked to draw themselves by calling the `draw()`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

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.

```
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() ;
   }
}
```

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.

MovingBallsWithPointerCanvas.html - 3: More mouse-related methods.

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.