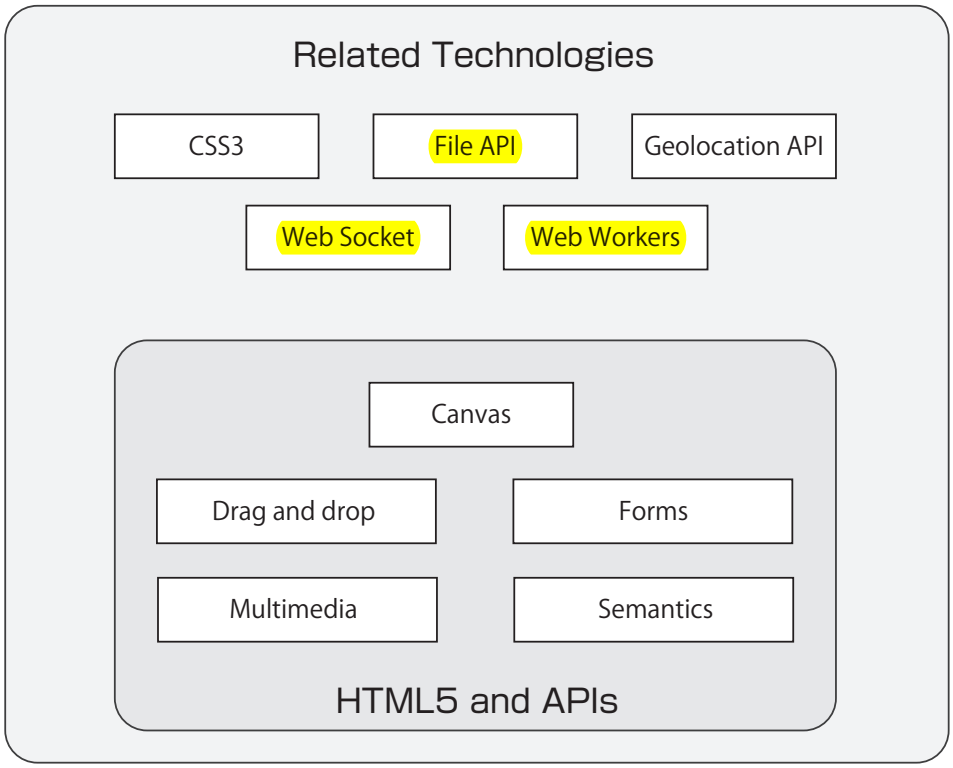
Lesson 10 HTML5 Web APIs

I What is HTML5?

HTML5 is the fifth, and the latest, version of the HTML standard being developed by the W3C (which is the main international standards organization for the Web). HTML5 proposes to develop richer Web applications than ever before only with HTML, CSS, JavaScript, and standard features of the browsers.

I - 1 Narrow and Broad Definitions of HTML5

HTML5, in the strict sense, refers to the core technologies centering the markup language developed by the W3C. Recently, however, HTML5 often means the collection of technologies as a whole including even CSS3, JavaScript API, etc. related to HTML5.



Hereinafter, we refer to the broad sense of "HTML5" with related technologies.

I - 2 HTML5 Web APIs

The notable point of HTML5 is various kind of APIs introduced for application developments rather than the extentions of markup elements. This is the main HTML5 APIs:

API	Descriptions
Drag and drop	Offers event-based drag & drop mechanism
Canvas	Offers ability to draw raster graphics in a page
File API	Offers how to read or write a local file
Geolocation API	Offers accessibility to geographical information of a user
Offline Web Application	Offers how to make offline appilications
Web Socket	Offers bidirectional communication bet. a server and client
Web Storage	Offers browser-base storage
Web Workers	Offers multi-threading model to scripts

Now, let's look at the detail of each API.

Ⅱ

Canvas

The Canvas API provides the scriptable way to draw raster graphics in your browser's screen using a combination of HTML and a script langauge. Before the birth of this API, it's ever required such an ad-on program as Flash, Silverlight, etc. to draw graphics dynamically in your browser. Using this Canvas API, you can make an equivalent with only standard features of browsers.

Ⅱ - 1
The canvas Element

When you use the Canvas API, the area for drawing graphics needs be defined by the canvas element. The default background color of the canvas element is transparent. Therefore, you can't see anything in the canvas area unless drawn something using the API.

```
<canvas width="WIDTH" height="HEIGHT">FALLBACK CONTENTS</canvas>
```

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's look at the source code.

```
  <canvas id="cv" width="300" height="200">
```

The canvas element requires the width and height attributes to specify its size and to set up its internal coordinate system. The unit is either pixel (px) or percentage (%). Note that the coordinate of the canvas is calculated in relation to the width and height attributes of HTML, not the width and height properties of CSS.

```
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
```

The text in the canvas element is a fall back content to be displayed when your browser doesn't support the canvas element. In this case, the fall back content has kind of warning text only, however it is better to provide the alternate graphics or other that is similar to the original canvas content. It should be the original data of a graph when the graph is going to be drawn in the canvas.

Ⅱ - 2
Canvas 2D Context API

The role of the canvas element is just putting the canvas for drawing graphics in the page. The Canvas 2D Context API is required to draw some graphics in the canvas area. This API includes various properties and methods for drawing graphics, setting styles, pasting pictures, etc.

```
Canvas 2D Context Object.getContext('2d');
```

When using the Canvas 2D Context API, the getContext('2d') method must be used first. Let's have a look at the following code:

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
    }
  }
);
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's go through the code.
You can access the Canvas API through the Canvas 2D Context object.

```
    if (HTMLCanvasElement) {
   ... snip ...
    }
```

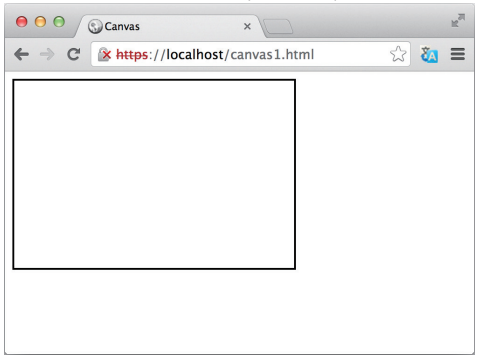
First it tries to access the HTMLCanvasElement. If succeeds, the following processes are executed.

```
var cv = document.querySelector('#cv');
var c = cv.getContext('2d');
```

After checking the availability of the Canvas API, create the Canvas 2D Context object. Context means the management of the canvas coordinates, style information and drawing itself, so this object is nothing but the fundamental object among the Canvas API technologies. In order to create the 2D Context object, just call the getContext() method of the Canvas object. "2d" as in the argument for the method means that it retrieves the context to draw some 2D graphics on the canvas. In near future, the argument "webgl" for drawing the 3D graphics is going to be implemented in each Web browser.

Now we can call any Canvas API methods or use any properties via this context object like "context object.xxxxx".

▼ As of now, the canvas is just transparent.



Ⅱ-3 Drawing a rectangle

Now we're ready to use the Canvas API. Let's draw some rectangle with the Canvas API. In order to draw a rectangle, the Canvas API has the following methods:

Methods	Descriptions
fillRect(x, y, width, height)	Paints the given rectangle onto the bitmap, using the current fill style.
strokeRect(x, y, width, height)	Paints the box that outlines the given rectangle onto the bitmap, using the current stroke style.
clearRect(x, y, width, height)	Clears all pixels on the bitmap in the given rectangle to transparent black.

In the above table, x or y specifies the X or Y coordinate of the Left Top corner of the rectangle, and width or height specifies the width or height of the rectangle, respectively. Let's have a look at the following sample.

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
```

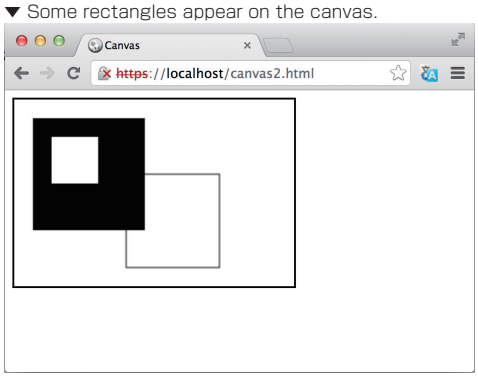
```
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
      c.fillRect(20, 20, 120, 120);
      c.strokeRect(120, 80, 100, 100);
      c.clearRect(40, 40, 50, 50);
    }
  }
);
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's look inside the code.

```
      c.fillRect(20, 20, 120, 120);
      c.strokeRect(120, 80, 100, 100);
      c.clearRect(40, 40, 50, 50);
```

In this sample, a filled rectangle is drawn by the fillRect method and a rectangular outline is drawn by the strokeRect method.
In addition, the clearRect method actually clears a part of the filled rectangle.

▼ Some rectangles appear on the canvas.



Ⅱ - 4
Creating a path

A path is a collection of multiple coordinates which are managed by the context. Basically, the Canvas API uses these coordinate information to draw lines or curves, or to fill some shape or some area. To create a path, we can use the following Canvas API methods:

Methods	Descriptions
beginPath()	Resets the current default path.
moveTo(x, y)	Creates a new subpath with the given point.
lineTo(x, y)	Adds the given point to the current subpath, connected by a straight line.
closePath()	Marks the current subpath as closed, and starts a new subpath with a point the same as the start and end of the newly closed subpath.
stroke()	Strokes the subpaths of the current default path or the given path with the current stroke style.
fill()	Fills the subpaths of the current default path or the given path with the current fill style.

The following code draws a polygon by using these methods.

▼ Example Code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
      c.beginPath();
      c.moveTo(100, 25);
      c.lineTo(50, 100);
      c.lineTo(100, 175);
      c.lineTo(175, 175);
      c.lineTo(225, 100);
      c.lineTo(175, 25);
      c.closePath();
      c.stroke();
    }
  }
);
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's check the source code.

```
c.beginPath();
```

We use the beginPath method to create a new path. We need to call the beginPath method each time we want to start drawing a new shape on a canvas. If you miss calling the beginPath method, the next point and the previous point would be connected by a straight line and you won't get a desired result.

```
c.moveTo(100, 25);
```

To position the first point of a path, we use the moveTo method of the context object. The X and Y coordinates of the starting point must be presented as the parameters of the moveTo method.

```
c.lineTo(50, 100);
c.lineTo(100, 175);
c.lineTo(175, 175);
c.lineTo(225, 100);
c.lineTo(175, 25);
```

We can add multiple points to a path by calling multiple lineTo methods consecutively. The above code defines the path consisting of the coordinates from (50, 100) to (175, 25) via (100, 175), (175, 175) and (225, 100).

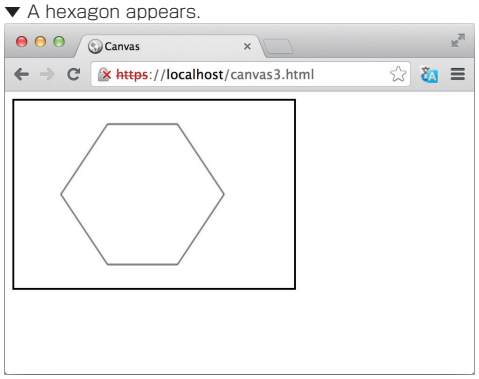
```
c.closePath();
```

Calling the closePath method connects the start point and the end point of a path by a straight line. In the above example, the start point (100, 25) and the end point (175, 25) are connected by a line. That's it for the path definition.
Notice that the path would result in just a broken line without the closePath method. That unconnected path is called "Open Path", while the path whose beginning and end points are connected is called "Closed Path."

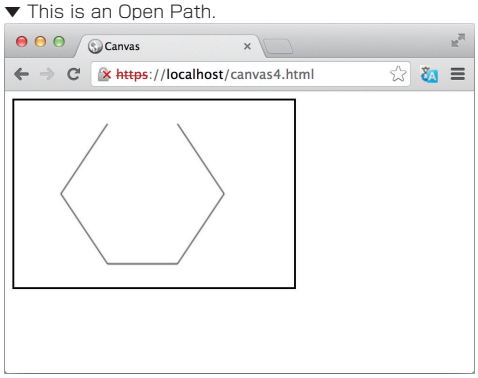
```
c.stroke();
```

However, we can't see anything on the canvas as of now because the path is just a kind of outline information of a shape. Now we need to call the stroke method after the path definition to draw a line along with the path information. Also, if you want to paint out the path area, call the fill method.

▼ A hexagon appears.



▼ This is an Open Path.



Ⅱ - 5
Style

In the 2D Context API, the following properties specify the color or thickness of a line, or the fill color of a path area, respectively.

Properties	Descriptions	Default Values
strokeStyle	Line color	#000000 (Black)
lineWidth	Line thickness	1 (px)
fillStyle	Fill color	#000000 (Black)

The following code example draws a filled polygon with these properties.

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
      c.beginPath();
      c.moveTo(100, 25);
      c.lineTo(50, 100);
      c.lineTo(100, 175);
      c.lineTo(175, 175);
      c.lineTo(225, 100);
      c.lineTo(175, 25);
      c.closePath();
      c.lineWidth = 5;
      c.strokeStyle = 'red';
      c.fillStyle = 'rgba(255, 0, 255, 0.5)'
      c.stroke();
      c.fill();
    }
  }
);
```

```
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's check the source code.

```
      c.lineWidth = 5;
```

We use the lineWidth property to specify the thickness of a line. The unit for the lineWidth property is Pixel. The above code says the thickness of a line is 5 pixels.

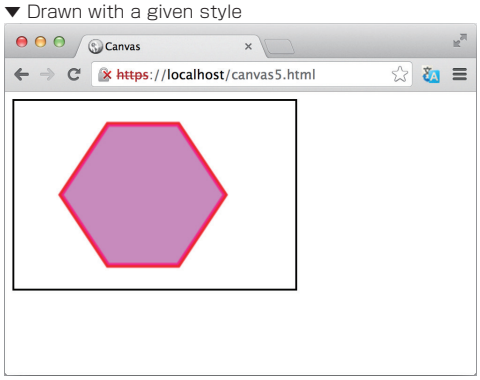
```
      c.strokeStyle = 'red';
      c.fillStyle = 'rgba(255, 0, 255, 0.5)'
```

The strokeStyle property specifies the color of an outline, while the fillStyle property defines the color for filling. To specify a color, we can use the color format which is defined in the specification "CSS Color Module Level 3." That means we can use such Color Names as "white" and "black", or such the RGB format as "#303030", or the rgb or rgba function to define a color.
In this sample, the strokeStyle property has the color name value "red" so after that we will draw an outline with a red color. The fillStyle property uses the rgba function in this time. The first three numbers 255, 0 and 255 as in rgba(255, 0, 255, 0.5) defines the strength of Red, Green and Blue within the range from 0 to 255, respectively. In this case, it turns purple. And the last number “0.5” specifies the opacity of the color (Alpha Channel). The value 0.0 means fully transparent, 1.0 means fully opaque and 0.5 means 50% opaque.
The color value is always enclosed with single quotes (' ') or double quotes (" ").

```
      c.stroke();
      c.fill();
```

After setting style information, now we can draw the shape with the stroke/fill method. If calling both methods together, a filled shape with a colored outline appears on the canvas.

▼ Drawn with a given style



Ⅱ - 6
Import an image

We can import an existing image to a canvas by using the drawImage method.

```
Canvas 2D Context Object.drawImage(img, x1, y1, w1, h1, x2, y2, w2, h2);
```

It takes an image, clips it to the rectangle (x1, y1, w1, h1), scales it to dimensions (w2, h2), and draws it on the canvas at coordinates (x2, y2).
If you don't want to clip the source image, just omit the parameters x1, y1, w1, h1.

The following code clips the source image "boy.jpg", scales it up, and draws it on the canvas.

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
      var img = new Image();
      img.src = 'images/boy.jpg';
      img.addEventListener(
        'load',
        function(e) {
          c.drawImage(img, 80, 30, 210, 200, 50, 10, 180, 180);
        }
      );
    }
  }
);
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's check the source code.

```
      var img = new Image();
      img.src = './images/boy.jpg';
```

To manipulate an image with JavaScript, we need the Image object. "New" is used to create a new object.

```
var InstanceName = new ObjectName();
```

The file path of a source image is set by the src property of the Image object.

```
img.addEventListener(
  'load',
  function(e) {
    ... snip ...
  }
);
```

Notice that loading a given source image which is specified by the src property is asynchronous to the script execution. Therefore, we need the load event listener to confirm the finish of loading, and then start processing after that.

```
  function(e) {
    c.drawImage(img, 80, 30, 210, 200, 50, 10, 180, 180);
  }
```

This is the content of the load event listener. To draw an image on a canvas, we need the drawImage method.

▼ Result

Ⅱ-7 Output of Canvas Data

By transforming the canvas bitmap data to the Data URL format, we can save it as a file or set it to the src attribute of the img element, for example. The Data URL is a format of encoding a multimedia data into a text string that can be easily understandable for a Web browser.

```
CanvasElement.toDataURL(Format);
```

To convert the canvas content to the Data URL format, just call the toDataURL method of the "cv" object which represents a canvas element. A desired image format must be set into the parameter of the toDataURL method.

The below sample code shows how to convert the canvas content into the Data URL format and how to set it to the src attribute of an img element. If a user clicks the button "Save", then she/he gets the image data which is transformed from a canvas element content.

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
      c.beginPath();
      c.moveTo(100, 25);
      c.lineTo(50, 100);
      c.lineTo(100, 175);
      c.lineTo(175, 175);
      c.lineTo(225, 100);
      c.lineTo(175, 25);
      c.closePath();
      c.lineWidth = 5;
      c.fillStyle = '#fcf'
      c.strokeStyle = '#f00';
      c.stroke();
      c.fill();
    }

    document.getElementById('save').
           addEventListener('click', function(e) {
      var img = new Image();
      img.src = cv.toDataURL('image/png');
      img.addEventListener('load', function(e) {
        document.body.appendChild(img);
      });
    }, false);
  }
);
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
  <input id="save" type="button" value="Save">
</body>
</html>
```

Let's check the source code.

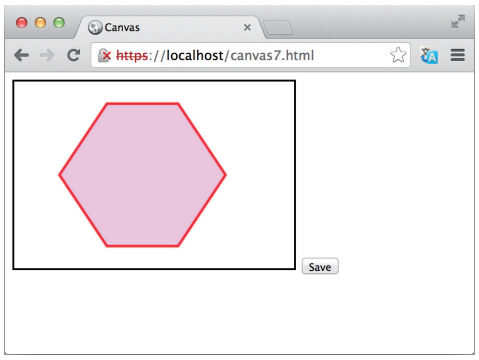
```
document.getElementById('save').
       addEventListener('click', function(e) {
 ...snip...
}, false);
```

This code attaches the event listener which is triggered when a user clicks the button "Save." The getElementById method refers to the "Save" button, and the addEventListener method defines the listener function which to call when clicking the "Save" button.

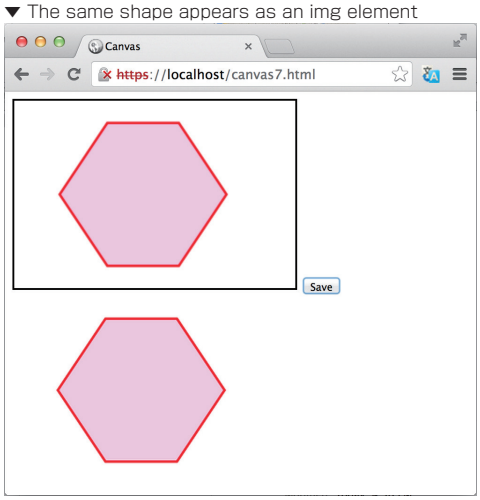
```
  var img = new Image();
  img.src = cv.toDataURL('image/png');
  img.addEventListener('load', function(e) {
    document.body.appendChild(img);
  });
```

This is the detail process of the listener function.
Using the toDataURL method, the canvas content is transformed to the Data URL format (the PNG image). We can set this Data URL string to the href attribute of the a element or the src attribute of the img element like a normal URL. In this time, the return value of the toDataURL method is set to the src property of a newly generated Image object (img). After that, the img element is added to the end of the page by the appendChild method.

▼ Click the "Save" button…



▼ The same shape appears as an img element



Ⅱ-8
Animation

We can make a simple animation on a canvas by "Drawing -> Reset -> Drawing in a little bit different position -> Reset" repeatedly. For example, the following code shows how to animate a square from the top-left corner to the bottom-right corner on a canvas.

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
}
</style>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');
      var x = 0, y = 0;
      setInterval(function() {
        c.clearRect(0, 0, 300, 200);
        c.fillRect(x, y, 20,20);
        x++;
        y++;
      }, 10);
    }
  }
);
</script>
<title>Canvas</title>
</head>
<body>
  <canvas id="cv" width="300" height="200">
      Your browser doesn't support the canvas element.</canvas>
</body>
</html>
```

Let's check the source code.

```
      var x = 0, y = 0;
```

First, prepare the variables "x" and "y" which are representing where are the coordinates of a shape. By changing these values little by little, we can make a simple animation on a canvas.

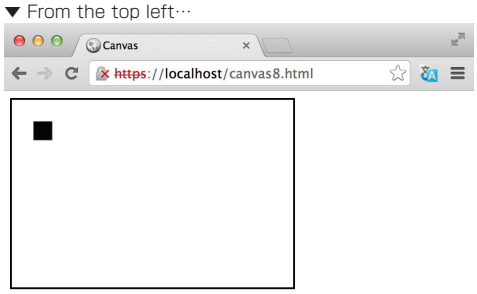
```
      setInterval(function() {
           ...snip...
      }, 10);
```

Specifically, to make an animation, drawing and clearing are executed repeatedly by the setInterval method. Although re-drawing of a square is repeated every 10 milliseconds in this code, this time frame is just an example. Actually we should adjust this interval so that it can work smoothly.

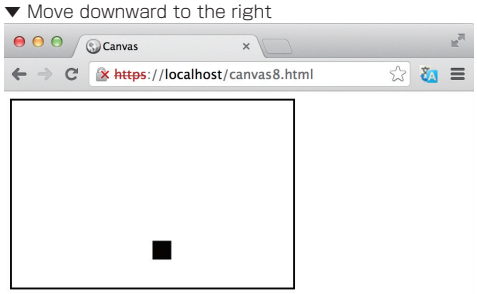
```
        c.clearRect(0, 0, 300, 200);
        c.fillRect(x, y, 20,20);
        x++;
        y++;
```

Here you can see the process in the callback function of the setInterval method.
First, the whole canvas is cleared by the clearRect method. Then, draw a 20x20 rectangle at the coordinates (x, y) by the fillRect method. The variables x and y should be incremented, so re-drawn rectangle must move downward to the right little by little.

▼ From the top left…



▼ Move downward to the right



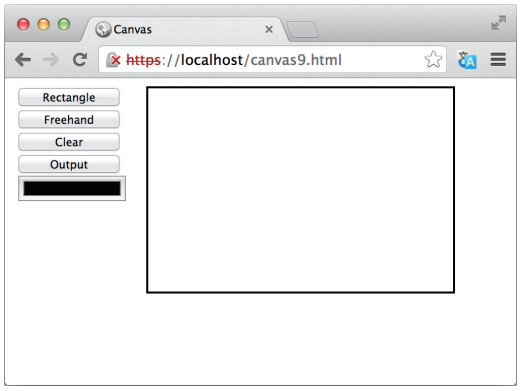
PRACTICE 3

Build a Painting App

Let's develop a simple paint Web app using the Canvas API. In this practice, we will make five main features of a paint app such as "Draw a rectangle", "Draw a freehand line", "Clear all", "Output as an image" and "Change a color."

First, have a look at the finished work.

▼ You can see a menu and a canvas area.



▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<style>
#cv {
  background-color: white;
  border: 2px solid black;
  float: left;
  margin-left: 20px;
}
ul{
  margin: 0px;
  padding: 0px;
  list-style-type: none;
  float: left;
}
ul li {
  display: block;
  margin-left: 5px;
}
button, input {
  width: 100px;
}
</style>
<script type="text/javascript">
window.addEventListener('DOMContentLoaded',
  function() {
    if (HTMLCanvasElement) {
      var cv = document.querySelector('#cv');
      var c = cv.getContext('2d');

      var startX = 0;
      var startY = 0;
      var endX;
      var endY;
      var tools = 0;

      document.getElementById('rectBtn')
              .addEventListener('click', function(e) {
        tools = 0;
      });

      document.getElementById('freeBtn')
              .addEventListener('click', function(e) {
        tools = 1;
      });

```

```
47          document.getElementById('colorInput')
                    .addEventListener('change', function(e) {
48            var newColor = e.target.value;
49            c.strokeStyle = newColor;
50          });
51
52          cv.addEventListener('mousedown', function(e) {
53            var rect = e.target.getBoundingClientRect();
54            startX = e.clientX-rect.left;
55            startY = e.clientY-rect.top;
56            if(tools === 0) {
57              cv.addEventListener('mousemove', drawRect);
58            }
59            else if(tools === 1) {
60              c.beginPath();
61              cv.addEventListener('mousemove', drawFree);
62            }
63          });
64
65          function drawRect(e){
66            var rect = e.target.getBoundingClientRect();
67            endX = e.clientX-rect.left;
68            endY = e.clientY-rect.top;
69            cv.addEventListener('mouseup', endRect);
70            function endRect(e) {
71              c.strokeRect(startX,startY,endX-startX,endY-startY);
72              cv.removeEventListener('mousemove', drawRect);
73              cv.removeEventListener('mouseup', endRect);
74            }
75          }
76
77          function drawFree(e){
78            var rect = e.target.getBoundingClientRect();
79            endX = e.clientX-rect.left;
80            endY = e.clientY-rect.top;
81            c.lineTo(endX, endY);
82            c.stroke();
83            cv.addEventListener('mouseup', function(e){
84              cv.removeEventListener('mousemove', drawFree)
85            });
86          }
87
88          document.getElementById('delete')
                    .addEventListener('click', function(e) {
89            c.clearRect(0, 0, cv.width, cv.height);
90          });
91
92          document.getElementById('save')
                    .addEventListener('click', function(e) {
93              var dataURL = cv.toDataURL("image/png");
94              window.location = dataURL;
95          });
96        }
97      }
98    );
99    </script>
100   <title>Canvas</title>
101   </head>
102   <body>
103   <ul>
104     <li><button id="rectBtn">Rectangle</button></li>
105     <li><button id="freeBtn">Freehand</button></li>
106     <li><button id="delete">Clear</button></li>
107     <li><button id="save">Output</button></li>
108     <li><input id="colorInput" type="color" value="#000000"></li>
109   </ul>
110   <canvas id="cv" width="300" height="200">
111     Your browser doesn't support the canvas element.</canvas>
112   </body>
113   </html>
```

Let's check the source code.

```
103  <ul>
104    <li><button id="rectBtn">Rectangle></button></li>
105    <li><button id="freeBtn">Freehand</button></li>
106    <li><button id="delete">Clear</button></li>
107    <li><button id="save">Output</button></li>
108    <li><input id="colorInput" type="color" value="#000000"></li>
109  </ul>
110  <canvas id="cv" width="300" height="200">
111    Your browser doesn't support the canvas element.</canvas>
```

In an HTML part, we put some buttons as a menu and a drawing area by the canvas element. These items have some style information in line 6 to 24. Please confirm it if needed.

```
37          var tools = 0;
38
39          document.getElementById('rectBtn')
                    .addEventListener('click', function(e) {
40            tools = 0;
41          });
42
43          document.getElementById('freeBtn')
                    .addEventListener('click', function(e) {
44            tools = 1;
45          });
```

This code shows how to check the button a user really clicks, "Rectangle" or "Freehand"? In this case, we use the variable "tools." If tools is 0, "Rectangle" is clicked; and if tools is 1, "Freehand" is clicked. Each button has the "click" event listener, and inside the listener, the value of tools is changed respectively.

```
30          var cv = document.querySelector('#cv');
31          var c = cv.getContext('2d');
```

Now we get into the Canvas 2D Context API. First, we need to call the getContext('2d') method to access the Canvas 2D Context API.

```
52          cv.addEventListener('mousedown', function(e) {
53            var rect = e.target.getBoundingClientRect();
54            startX = e.clientX-rect.left;
55            startY = e.clientY-rect.top;
                ... snip ...
63          });
```

In this code, the "mousedown" event listener is registered for the canvas element. "e.target.getBoundingClientRect()" (line 53) takes the coordinates of the target object of the "mousedown" event. Therefore, it takes the coordinates of the canvas element itself in this time.
"e.clientX" (line 54) gets the X coordinate, at which the "mousedown" event happens, calculated from the top left corner of a browser (X=0, Y=0). In other words, line 54 means the value which calculated by "X coordinate at which an event happens" - "X coordinate of the left side of the canvas element." This is the process in which we can get the relative position of a mouse cursor on a canvas when the mouse is clicked. Similarly, Y coordinate is calculated in line 55.

```
56            if(tools === 0) {
57              cv.addEventListener('mousemove', drawRect);
58            }
59            else if(tools === 1) {
60              c.beginPath();
61              cv.addEventListener('mousemove', drawFree);
62            }
```

Next, using if & else if statements, the process for "Rectangle" and that for "Freehand" are separated. If "tools" is 0, that is the case of "Rectangle", the "drawRect" function is called by the addEventListener method every time a mouse is moved. Similarly, if "tools" is 1, the "drawFree" function is executed. In case of "Freehand", we need to call the "beginPath" method before everything to start drawing a new path.
The basic processes in the "drawRect" and "drawFree" functions are the same as the Canvas 2D Context API, so let us explain about just the "drawRect" function here, and please focus on not the details of each instruction but the structure of the process as a whole.

```
65          function drawRect(e){
66            var rect = e.target.getBoundingClientRect();
67            endX = e.clientX-rect.left;
68            endY = e.clientY-rect.top;
                ... snip ...
75          }
```

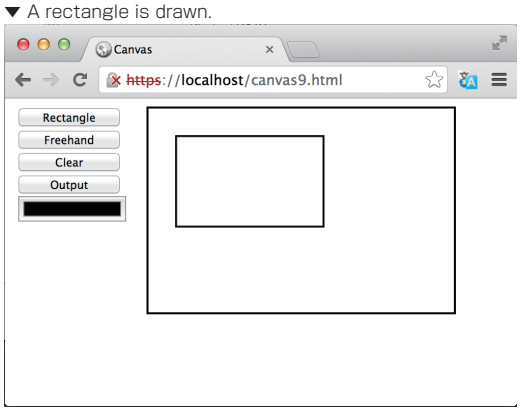
They are the same as we saw from line 53 to 55. The "drawRect" function needs the coordinates information every time a mouse is moved because this process is repeated until the mouse is stopped moving.

```
69            cv.addEventListener('mouseup', endRect);
70            function endRect(e) {
71              c.strokeRect(startX,startY,endX-startX,endY-startY);
72              cv.removeEventListener('mousemove', drawRect);
73              cv.removeEventListener('mouseup', endRect);
74            }
```

In addition, there is the "endRect" listener function to call when a mouse button is released. In line 71, the outline of the given rectangle is drawn. Also in line 72 and 73, they remove each event listener for "drawRect" and "endRect" at the same time (when the mouse button is released). To remove an event listener, we need to call the "removeEventListener" method. Otherwise, even if you release the mouse button, drawing a path cannot stop forever.

In conclusion, there is such a process flow as each event listener called when "Press a mouse button" -> when "Move the mouse" -> when "Release the mouse button" to draw a desired rectangle on the canvas.

▼ A rectangle is drawn.



Then, let's look into the "Clear all" process.

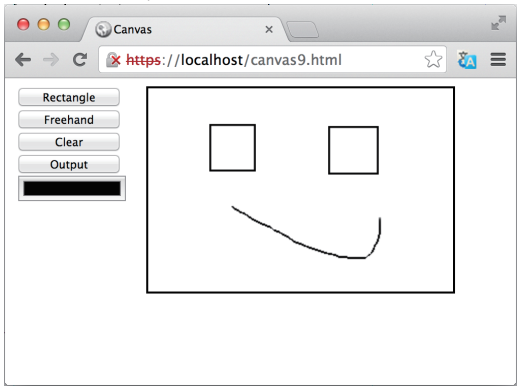
```
88          document.getElementById('delete')
                    .addEventListener('click', function(e) {
89            c.clearRect(0, 0, cv.width, cv.height);
90          });
```

This "Clear all" process is very simple. Just using the clearRect method to clear the whole canvas area. The third parameter specifies the width of cleared area. In this case, the width of the canvas area is specified. Similarly. the last parameter is the height of cleared area, and the height of the canvas area is given in the above code. These are the tricky points.

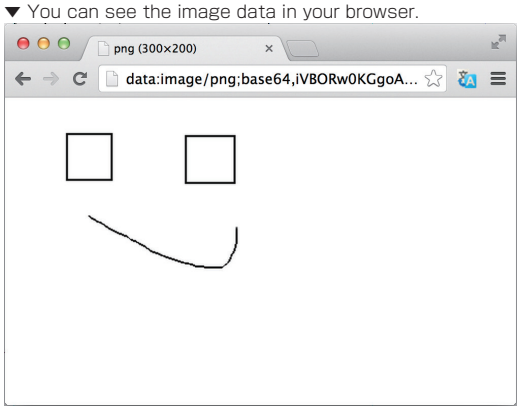
```
92          document.getElementById('save')
                    .addEventListener('click', function(e) {
93              var dataURL = cv.toDataURL("image/png");
94              window.location = dataURL;
95          });
```

Now let's check the "Output as an image" feature. In line 93, the canvas content is transformed into the Data URL format by using the toDataURL method. In line 94, that Data URL string is set into the browser object "location" so the result of the image data is output in the browser screen. Then you can save the image data as the PNG format by clicking "Save As" in "File" menu of your browser.

▼ Click the "Output" button, then…



▼ You can see the image data in your browser.



Finally, let's look at the "Color picker" function.

```
47          document.getElementById('colorInput')
                    .addEventListener('change', function(e) {
48            var newColor = e.target.value;
49            c.strokeStyle = newColor;
50          });
             ... snip ...
108     <li><input id="colorInput" type="color" value="#000000"></li>
```

This code uses the new HTML5 feature: <input type="color"> element. As of February 2014, this element can be working in the "Google Chrome" and "Opera" browsers only. If it works in a right way, you can see the color picker and change the color selection from the dialog. Then if the color selection is changed, the "change" event listener is triggered and the value of <input type="color"> is set into the strokeStyle property of the current context. That is to say, your selected color is set as the color of outlines.

▼ Color picker dialog in Google Chrome

Ⅲ Geolocation API

The Geolocation API offers the standardized way to get the geographical information of a user's device without depending on a manufacturer's proprietary standard.
Note that the accuracy of geographical information varies by situation because the Geolocation API calculates that information from such a source as GPS, Wi-Fi base station, IP address, etc.

Ⅲ - 1 Get the current location

The Geolocation API introduces the following methods:

Methods	Descriptions
getCurrentPosition	Gets the current position of the device
watchPosition	Registers location monitoring handlers
clearWatch	Unregisters location monitoring handlers installed by watchPosition()

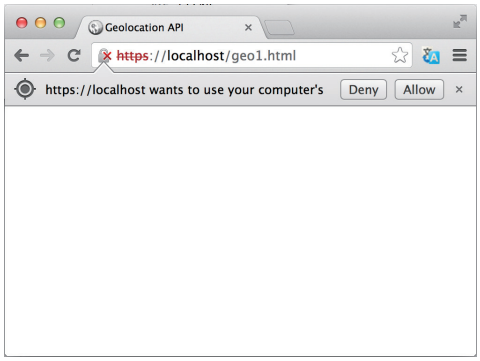
Let's get started with the most basic method, getCurrentPosition(), to write a code which will get the current location of a user's device.

▼ Example Code

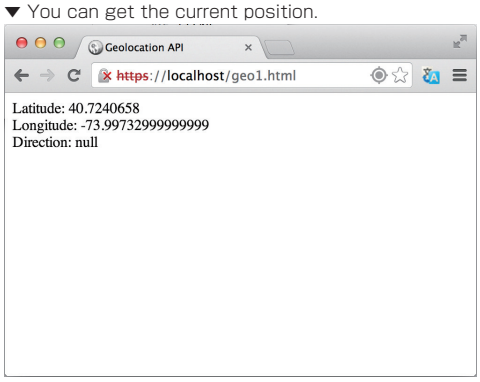
```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Geolocation API</title>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    var result = document.querySelector('#result');
    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(
        function(pos) {
          msg = 'Latitude: ' + pos.coords.latitude + '<br>' +
            'Longitude: ' + pos.coords.longitude + '<br>' +
            'Direction: ' + pos.coords.heading;
          result.innerHTML = msg;
        },
        function(err) {
          var msgs = [
            err.message,
            'Permission denied.',
            'Position unavailable.',
            'Timeout.'
          ];
          alert(msgs[err.code]);
        },
        {
          timeout : 10000,
          maximumAge : 0,
          enableHighAccuracy: true
        }
      );
    } else {
      alert("Your browser doesn't support the Geolocation API.");
    }
  }, false
);
```

```
</script>
</head>
<body>
<div id="result">
</div>
</body>
</html>
```

▼ Choose "Allow"



▼ You can get the current position.



First, your browser asks you like "(Domain name) wants to know your location." Click "Share Location" in the dialog. That means you need such a permission to get the geolocation information.
Okay, let's go over the source code of the example.

```
if (navigator.geolocation) {
... snip ...
} else {
  alert('Your browser doesn't support the Geolocation API.');
}
```

You can access the Geolocation API through the "navigator.geolocation" property. Here, we see if the browser supports this property, and if supports, the following process is executed.

```
navigator.geolocation.getCurrentPosition(
  function(pos) {
     ... snip ...
  },
     ... snip ...
);
```

To get the current position of the device, we need the "navigator.geolocation.getCurrentPosition" method. The first parameter "function(){…}" is the callback process executed when the browser gets the location information successfully. It is called a "Success Callback."

```
function(pos) {
  msg = 'Latitude: ' + pos.coords.latitude + '<br>' +
    'Longitude: ' + pos.coords.longitude + '<br>' +
    'Direction: ' + pos.coords.heading;
  result.innerHTML = msg;
},
```

This is the content of the success callback. The argument "pos" of the success callback refers to the position information (Position object) retrieved by the Geolocation API. Inside the success callback, we can access the position information through this Position object. The following properties are accessible through the Position object:

Properties	Descriptions	
coords	Coordinates object representing the position	
	latitude	Latitude
	longitude	Longitude
	accuracy	Accuracy of the latitude and longitude coordinates (in meters)
	altitude	Altitude
	altitudeAccuracy	Accuracy of the altitude coordinate (in meters)
	heading	Direction (in degrees counting clockwise relative to the North)
	speed	Velocity (in meters per second)
timestamp	Update date and time (in milliseconds from the year of 1970)	

In this time, we get the latitude, longitude and direction coordinates by "pos.coords.(Property)", organize these data into the list format, and set it in the variable "msg." Then set the variable as the text content of the element "result."

```
function(err) {
  var msgs = [
    err.message,
    'Permission denied.',
    'Position unavailable.',
    'Timeout.'
  ];
  alert(msgs[err.code]);
},
```

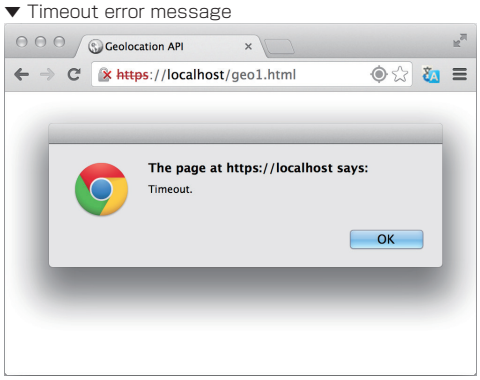
The second parameter of the getCurrentPosition method is the error handling in the case of the failure to retrieve the device's position. This is also called as "Error Callback."
The argument "err" of the error callback refers to error information (PositionError object) of the Geolocation API.

The PositionError object has the following properties:

Properties	Descriptions	
code	Error codes	
	Codes	Descriptions
	0	Unknown
	1	Permission denied
	2	Position unavailable
	3	Timeout
message	Error message	

In the example code, the desired error messages corresponding to these error codes are prepared as the array "msgs" in advance so that you can retrieve these messages by "msgs[err.code]." For example, when the return value of err.code is 1, you will get the message of msgs[1].
In addition, when the error code is 0, which means some unknown error, you will get the standard message of the Geolocation API by "err.message."

▼ Timeout error message




```
{
  timeout : 10000,
  maximumAge : 0,
  enableHighAccuracy: true
}
```

The third parameter of the getCurrentPosition method is the options for the Geolocation API (PositionOptions object). These options are set in the form of a JavaScript object like { name1:value1, name2:value2, name3:value3 }.
The meaning of each option is as follows:

Options	Descriptions
timeout	The maximum length of time (in milliseconds) the device is allowed to take in order to return a position
maximumAge	The maximum age in milliseconds of a possible cached position that is acceptable to return
enableHighAccuracy	The application would like to receive the best possible results

If the timeout error occurs so often, you can try setting the timeout property longer than the current setting.
The maximumAge option indicates the length of time the position information (Position object) can be cached and regarded as valid.
If you give "true" to the enableHighAccuracy option, the Web application can retrieve the most accurate position information. On your smartphone, it uses the GPS information. Note that the enableHighAccuracy option consumes the device's battery too much.

Ⅲ - 2
Location monitoring

In the last section, we've learned the getCurrentPosition method to retrieve the device's current position at the very moment. While in this section, we will learn how to monitor the device's position continuously by using the watchPosition method. This method is basically for the mobile device whose position is changed so often.

▼ Example Code

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>Geolocation API</title>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    var result = document.querySelector('#result');
    var stop = document.querySelector('#stop');
    if (navigator.geolocation) {
      var id = navigator.geolocation.watchPosition(
        function(pos) {
          msg = 'Latitude: ' + pos.coords.latitude + '<br>' +
            'Longitude: ' + pos.coords.longitude + '<br>' +
            'Direction: ' + pos.coords.heading;
          result.innerHTML = msg;
        },
        function(err) {
          var msgs = [
            err.message,
            'Permission denied.',
            'Position unavailable.',
            'Timeout.'
          ];
          alert(msgs[err.code]);
        },
        {
          timeout : 10000,
          maximumAge : 0,
          enableHighAccuracy: true
        }
      );
      stop.addEventListener('click',
        function() {
          navigator.geolocation.clearWatch(id);
        }
      );
    } else {
      alert("Your browser doesn't support the Geolocation API.");
    }
  }, false
);
</script>
</head>
<body>
<div id="result">
</div>
<input id="stop" type="button" value="Stop monitoring">
</body>
</html>
```

▼ Numbers change in accordance with device's movement

Let's check the source code.

```
var id = navigator.geolocation.watchPosition(
  function(pos) {
    msg = 'Latitude: ' + pos.coords.latitude + '<br>' +
      'Longitude: ' + pos.coords.longitude + '<br>' +
      'Direction: ' + pos.coords.heading;
    result.innerHTML = msg;
  },
  function(err) {
    var msgs = [
      err.message,
      'Permission denied.',
      'Position unavailable.',
      'Timeout.'
    ];
    alert(msgs[err.code]);
  },
  {
    timeout : 10000,
    maximumAge : 0,
    enableHighAccuracy: true
  }
);
```

The syntax of the watchPosition method is the same as that of the getCurrentPosition method. The first parameter is a success callback, the second is an error callback, and the last one is an object for some options.
The getCurrentPosition() method returns nothing, while the watchPosition() method returns a unique integer (a long value) which is called a watchId that can be used to stop the watch operation.

```
      stop.addEventListener('click',
        function() {
          navigator.geolocation.clearWatch(id);
        }
      );
```

This process is invoked when "Stop monitoring" button is clicked. The clearWatch() method stops the watch operation which corresponds to the given watchId argument. You have to pass the watchId to the clearWatch() method to stop the corresponding watch operation.

Ⅲ - 3 Collaboration with Google Maps API

You can pass the position information of the device which is retrieved from the Geolocation API to the Google Maps API. Both APIs are basically very similar so they are often used altogether especially for smartphone apps.
This is the example code to display the current position of the device in the Google Map.

▼ Example Code:

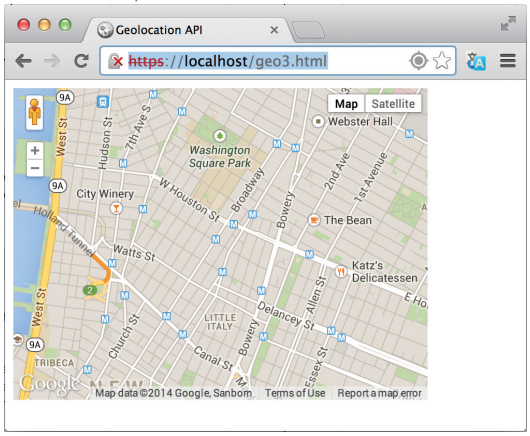
```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Geolocation API</title>
<script src="http://maps.google.com/maps/api/js?sensor=true"></script>
<script>
window.addEventListener('DOMContentLoaded',
  function() {
    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(
        function(pos) {
          var gmap = new google.maps.Map(
            document.querySelector('#gmap'),
            {
              zoom: 14,
              center: new google.maps.LatLng(
                pos.coords.latitude, pos.coords.longitude
              ),
              mapTypeId: google.maps.MapTypeId.ROADMAP
            }
          );
        },
        function(err) {
          var msgs = [
            err.message,
            'Permission denied.',
            'Position unavailable.',
            'Timeout.'
          ];
          alert(msgs[err.code]);
        },
        {
          timeout : 10000,
          maximumAge : 0,
          enableHighAccuracy: true
```

Google Maps

The mapping service provided by Google. It has gained explosive popularity in the world because their map can be moved or zoomed smoothly without any redirection. By using the Google Maps API, we can make or customize our own Google Maps contents with JavaScript.

```
        }
      );
    } else {
      alert("Your browser doesn't support the Geolocation API.");
    }
  }, false
);
</script>
</head>
<body>
<div id="gmap" style="width:400px; height:300px;"></div>
</body>
</html>
```

▼ The current position is located in the center.



Let's check the source code.

```
<script src="http://maps.google.com/maps/api/js?sensor=true"></script>
```

Firstly, we have to import the Google Maps API itself with the script tag. The query string "sensor=true" as in the URL means you want your device to use its CPS feature. With smartphone apps, "true" must be the best setting.

```
<div id="gmap" style="width:400px; height:300px;"></div>
```

This code defines the area which will display a map graphics. Set the size like 300px x 400px for . Notice that if you forget to set the size (width and height), you may get an unfit map for the area.

The following process is the process which shows a map in the area as a result of collaboration between the Geolocation API and Google Maps API. The part of the Geolocation API is the same as we have learned before, so let's check the remaining code which is related to the Google Maps API.

The below code is a fragment of the success callback process, which really displays a map.

```
        function(pos) {
          var gmap = new google.maps.Map(
            document.querySelector('#gmap'),
            {
              zoom: 14,
              center: new google.maps.LatLng(
                pos.coords.latitude, pos.coords.longitude
              ),
              mapTypeId: google.maps.MapTypeId.ROADMAP
            }
          );
```

The Google Map itself is referred as the google.maps.Map object. We pass the reference to to the first parameter of its constructor as a destination of displaying a map. In the second parameter, we set some Google Maps options in the form of an object { name1:value1, name2:value2, ···, nameX:valueX }. You can set the following options in it:

Options	Descriptions	
mapTypedId	Display mode of a map	
	Values	Descriptions
	google.maps.MapTypeId.ROADMAP	Normal road map (default)
	google.maps.MapTypeId.SATELLITE	Satellite view
	google.maps.MapTypeId.HYBRID	Mixed of a road map and satellite view
	google.maps.MapTypeId.TERRAIN	Map with terrain information
center	Center of a map	
zoom	Magnification rate (0:Min - 21:Max)	

We will set the center option with the coordinates information of the center position as the google.maps.LatLng object. The LatLng constructor gets the two parameters: the one is latitude and the other is longitude.
In the sample code, these coordinates is retrieved from the current position information of the Geolocation API. That means the current device's position will be set as the center of the map.