

# HTML5

### KEY OBJECTIVES: \_\_\_\_\_

After completing this chapter readers will be able to—

- know new tags introduced in HTML5
- get an idea about what semantic elements are and how to use them
- know how to work with media using <audio> and <video> elements
- get sound knowledge on graphic elements such as <canvas> and <svg>
- understand and use JavaScript API such as Web Storage, IndexedDB, Geolocation, Drag and Drop, App Cache, Web Worker, Server Sent Events etc.

### 23.1 Introduction

The World Wide Web, from its introduction in 1989 by Tim Berners-Lee, has changed considerably and the HTML 4.01, which is being used for long, cannot fulfill today's need. Some components in HTML 4.01 have now become old, some are not utilized at all, and some are utilized in a different way than they were specified. Likewise, a new version HTML 5 was released in 2014 introducing a wide range of components to handle current situations. For example new elements and attributes for better semantics, canvas for drawings, playing media (audio, video etc.), drag-and-drop, handling location, new form elements were introduced. Many were previously dependent on third-part browser plug-ins.

### 23.2 Migrating to HTML5

Web pages that conform to HTML5 specification have different structure. We shall first discuss some of them.

An HTML5 document starts with a DOCTYPE declaration that tells the browsers that it is an HTML5 document. It looks like this:

```
<!DOCTYPE html>
```

It may not have any URL but must precede all other elements. The language used in an HTML5 page is specified using the `lang` attribute of `<html>` element, like this:

```
<!DOCTYPE html>
<html lang="en">

</html>
```

A list of language codes may be found in the following URL:

<http://webdesign.about.com/od/localization/l/bllanguagecodes.htm>

It is suggested to specify character encoding used in the document as follows:

```
<head>
  <meta charset="utf-8">
</head>
```

## 23.3 HTML5 Elements

Many new elements have been introduced in HTML5 specification. Table 23.1: provides a summary of them. Some tags have also been deprecated and should not be used in the future. Table 23.2: provides a list of deprecated tags. The living standard of HTML is provided in

<https://html.spec.whatwg.org/>

Table 23.1: List of tags added in HTML5

<article>	<aside>	<audio>	<canvas>	<command>	<datalist>	<details>
<embed>	<figure>	<figcaption>	<footer>	<header>	<hgroup>	<keygen>
<mark>	<meter>	<nav>	<output>	<progress>	<ruby>	<section>
<time>	<video>	<wbr>				

Table 23.2: List of tags deprecated in HTML5

<acronym>	<applet>	<basefont>	<center>	<dir>	<big>	<tt>
<frame>	<frameset>	<isindex>	<menu>	<noframes>	<plaintext>	<s>
<u>	<xmp>					

### 23.3.1 input types

HTML5 introduced 13 new input elements to be used for forms. These elements can accept variety of data such as date (and related), email, URL, color, search string etc. The benefits of these input types are two-fold: less development time and improved user experience. The new input types are:

- date
- datetime-local
- datetime
- month
- week

- time
- number
- tel
- range
- url
- color
- email
- search

New attributes for these input elements have also been introduced for better control of the data. Table 23.3: shows them with a brief description. Following sections demonstrate how to use new input elements and attributes.

Table 23.3: List of attributes added to the input elements HTML5 in

Attribute	Meaning
min	Used to specify the smallest possible value for an input field
max	Used to specify the largest possible value for an input field
pattern	Its value is a regular expression that matches the input value
required	Indicates if a value for an input field is mandatory (must be filled out) or not
step	Its value is the increment/decrement interval of the value of an input field

You may have come across date pickers while booking online tickets, reserving hotel etc. Before HTML5, either we were to write a complicated code using JavaScript to do this or load a whole library (such as jQuery) just to get a date picker. However, in HTML5, a new input element is provided and HTML5 compatible browsers can render such date picker. In fact, in addition to date, we can select a week, month, time, date and time using different input types.

We know that the type of an input element is specified using `type` attribute of `<input>` tag. New input elements are written in the same way. Following sections demonstrate it.

#### date

This type of input field can hold a date (day, month and year) in ISO 8601 format(YYYY-MM-DD). Here is an example:

```
date : <br>
<input type=date>
```

Note that an input element often has an attribute `name`. Here we are showing only relevant portion. The visual appearance may differ in different browsers. When it is clicked, a small window to select date may also pop up in some browsers. Here is a sample output of the above code in chrome:

date :

mm/dd/yyyy ▼

April, 2016 ▼

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Users may be enforced to select a date from a specific range:

```
date from 2016-04-21 to 2016-05-24 : <br>
<input type=date min=2016-04-21 max=2016-05-24>
```

Note that it does not allow users to specify time zone offset.

#### datetime-local

Similar to `date` except that it includes time (hour, minute and second). Here is an example:

```
datetime-local : <br>
<input type=datetime-local>
```

datetime-local :

mm/dd/yyyy --:-- -- ▼

April, 2016 ▼

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

It holds ISO 8601 encoded date and time (without time zone offset) in YYYY-MM-DDThh:mm format. When it is clicked, a small pop-up window to select a date may also appear in some browsers. Here is a sample output of the above code in chrome:

`datetime`

Similar to `datetime-local` except that it includes a Time Zone Designator (TZD) in YYYY-MM-DDThh:mmTZD format. Here is an example:

```
datetime : <br>
<input type=datetime name=datetime>
```

Note that it is removed from most recent version of HTML5 specification as it was not supported by most browsers. Here is a sample output in opera Presto/2.12.388 Version/12.17. Here time is in Coordinated Universal Time (UTC), with a special UTC designator "Z".

**datetime :**

2016-04-09 04:30 UTC

Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	1
2	3	4	5	6	7	8

Today

`month`

It allows users to provide a month and a year (in YYYY-MM format). Here is an example:

```
month : <br>
<input type=month>
```

It is useful to specify the expiry date of a credit card. Some browsers help users to select month and year from a pop-up window.

**month :**

-----, ---- submit

April, 2016

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

`week`

It holds a string representing a week of a year in YYYY-WNN format. Here NN is a two digit week number (1 to 52 or 53 depending on year) and W indicates that it is a week number. Following is an example:

```
week : <br>
<input type=week>
```

When it is clicked, a small pop-up window to select a week appears in chrome as shown here. Users may be enforced to select a week from a specific range. Following allows users to select a week from the first quarter of 2016.

**week :**

Week --, ---- submit

April, 2016

Week	Sun	Mon	Tue	Wed	Thu	Fri	Sat
13	27	28	29	30	31	1	2
14	3	4	5	6	7	8	9
15	10	11	12	13	14	15	16
16	17	18	19	20	21	22	23
17	24	25	26	27	28	29	30

```
week : <br>
<input type=week min=2016-W01 max=2016-W12>
```

`time`


The value of this element is a string representing a time (no time zone offset) in hh:mm format. It may be used to set time for alarm clock. Following shows an example.

```
time : <input type=time >
```

time : 01:30 PM X

Chrome renders this input element as shown here. We can select a specific field and click up or down arrows to change the value. Alternatively, we can write a value directly into the fields.

number

It, as its name indicates, can accept only numeric values. Here is an `number` : 

example:

```
number : <input type=number value=5>
```

Chrome, Safari render it as a spin box as shown here. We can click up or down arrows to change the value. Alternatively, we can write a value directly into the field. This element has attributes such as `min`, `max` and `step` to specify minimum, maximum and interval (default is 1) respectively.

```
number : <input type=number min=0 max=100 step=0.5 value=5>
```

Incompatible browsers render it as standard text input. In that case an appropriate value to the pattern attribute may be specified as shown below:

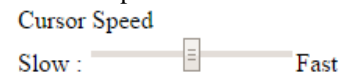
```
<input type=text pattern=[0-9]*>
```

It only accepts positive integers.

range

Similar to `number` but the numeric value has always a range (hence it's name) mentioned it or not. It is useful when relative value instead of absolute value should be specified. Here is an example:

```
Cursor Speed<br>Slow : <input type=range>Fast
```

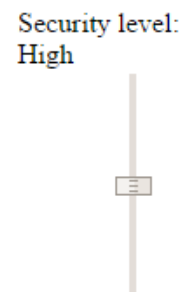


Chrome renders it as shown here. It also has attributes such as `min`, `max` and `step` to specify minimum (default is 0), maximum (default is 100) and interval (default is 1) respectively.

```
<input type=range min=10 max=20 step=0.5>
```

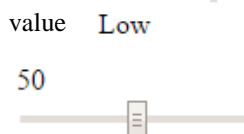
Following style may be used to me the slider vertically oriented:

```
Security level:<br>High<br><input type=range style='-webkit-appearance: slider-vertical'><br>Low
```



When the slider is moved, some browser such as IE 10 shows the value using a tooltip. For others, a small JavaScript code may be used as follows:

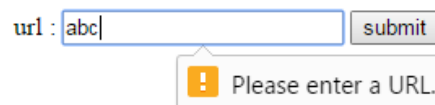
```
<div id=val></div>
<input type=range
onInput=document.getElementById('val').innerText=value>
```



url

This input field is used to provide web addresses i.e. URLs. It appears as just an ordinary text field. Use of `url` input type looks like:

```
url : <input type=url>
<input type=submit value=submit>
```



The value must conform to URL format. Browser performs validation for this field and shows error message when submitted. Here is a validation error example in Chrome when an ill formatted value 'abc' was submitted. Most browsers accept one or more letters followed by a colon as valid values for this field. However, restriction may be imposed using `pattern` attribute as follows:

```
<input type=url pattern=https?://.+>
```

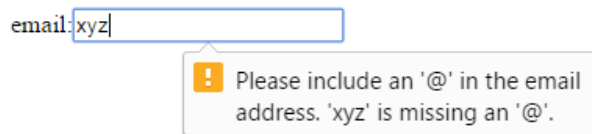
This input field now accepts only text starting with `http://` or `https://` followed by at least one additional character.

## email

Like `url`, it looks like a text box but accepts one or more e-mail addresses.

```
email:<input type=email>
```

Browser checks if a properly formatted value is provided and shows error message when submitted. This checking is rudimentary as every email address has at least a '@' character. Different browsers perform validation differently. For example, Chrome, Opera etc. accept anything in the form `*@*`. Chrome shows an error message as shown here.



The required attribute may be used to prevent supplying null (blank) value.

```
<input type=email required>
```

## tel

Like `url`, and `email` which accepts URL and email addresses, `tel` field accepts telephone numbers. Since for format of telephone numbers differs around the world widely, this field does not impose any restriction on values supplied. However, the field may be validated using its `pattern` attribute provided that a pattern is somehow determined. Stylistically is exactly same as text input. This field merely gives a hint that the value entered should be a telephone number.

## color

The input element `color`, as its name implies, allows users to select a color.

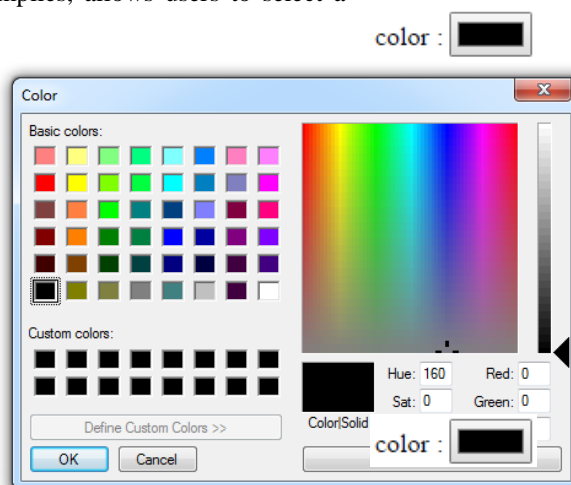
```
color : <input type=color>
```

The `color` is specified as the hex format as `#RRGGBB`. When the element is clicked, some browsers pop-up a color picker which may be operating system's color palette or a browser's own palette. Chrome uses a color palette as shown here.

## search

A search input field is same as a text input except styling. Here is an example:

```
search : <input type=search>
```

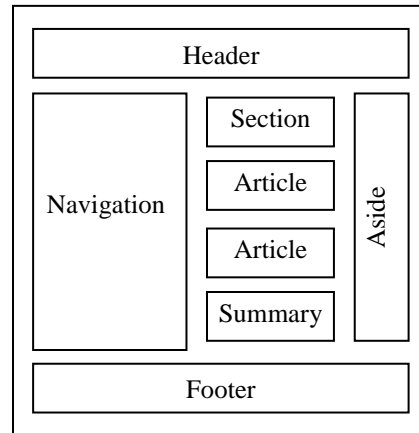


In fact specification does not require it to do anything special. However, some browsers place a small cross sign on the right side of the field as when we start typing. This helps users to clear the field quickly.

search :

## 23.4 Semantic Elements

A semantic element is essentially any element with concrete meaning. For example, `<form>`, `<table>`, `<img>` etc. clearly defines its contents. Non-semantic elements such as `<div>`, `<span>`, `<p>` do not hint anything about their contents. As a result it is impossible for the browser or search engines to conclude that the content of a `<p>` is an article or a header or footer or navigation section. To make pages more semantically structured and meaningful, HTML5 introduced a number of new elements often called *semantic elements*. These elements give a hint about their contents. Note that semantic elements don't offer anything except for clarity. They do not themselves include any styling. However CSS styles can be written for them. The different parts of web page are shown in the picture.



All semantic elements are block-level elements. Some older browser may not interpret them correctly. To get the correct behavior we set the CSS display property to block:

```

article, header, footer, section, aside, nav, main, figure {
    display: block;
}

```

`<article>`

An article is a something which is independently distributable or reusable. It should make sense of its own, and it should be read independently from the rest of the web site. This may be a newspaper article, a blog, a forum post, or any other segment of a page, or an entire page or even an application. This semantic concept is represented using `<article>` element. Here is an example:

```

<article>
  <h1>HTML5 Semantic Elements</h1>
  <p>
    Semantic elements enforce the semantics or meaning, of the information in
    web pages or applications instead of merely to specify its presentation
    or look.
    ...
  </p>
</article>
<article>
  <h1>Java</h1>
  <p>
    Java is an extremely versatile general-purpose programming language. It
    derives much of its syntax from C and C++.
    ...
  </p>
</article>

```

This has two independent articles; one on “Java” and another on “HTML5 Semantic Elements”. Note that an `<article>` is not supposed to contain a single `<p>` element; it may contain a mixture of arbitrary text and elements if it makes sense of its own.

`<header>`

This element is used to introduce something such as an article, section etc. So, our previous article could have been written using header as follows:

```
<article>
  <header><h1>HTML5 Semantic Elements</h1></header>
  <p>
    Semantic ...
  </p>
</article>
<article>
  <header><h1>Java</h1></header>
  <p>
    Java ...
  </p>
</article>
```

A page can contain several `<header>` elements.

`<footer>`

Similar to `<header>` except that it appears at the end of an article or section. Footer typically contains the author, contact, copyright information, terms and conditions etc.

```
<article>
  <header><h1>Java</h1></header>
  <p>
    Java ...
  </p>
  <footer>This article was last modified on June 11, 2016</footer>
</article>
```

Like `<header>` a page can contain several `<footer>` elements.

`<section>`

This element is used to represent a segment of thematically related contents of a document. For example chapters or the numbered sections of an e-book/thesis or tabbed pages in a tabbed dialog box etc. The theme is typically represented by a heading using `<header>` of `<h1>...<h6>` or any other suitable elements. It is similar to `<article>` but more general. Different sections need not be independent. Here is an example:

```
<section>
  <header><h1>HTML5 Web Storage </h1></header>
  <p>It one of the local (to client) storage facilities...</p>
</section>

<section>
  <header><h1>HTML5 Indexed DataBase</h1></header>
  <p>One of the powerful and recent web standards is Indexed Database ...</p>
</section>
```

The two sections describe two important features of HTML5. A section may have independent articles. So, `<article>` elements may be nested in `<section>` elements:

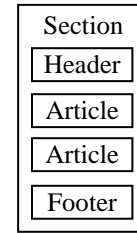
```
<section>
  <header><h1>Web Tutorial</h1></header>
  <article>
```



```

    <header><h1>HTML</h1></header>
    <p>...</p>
  </article>
  <article>
    <header><h1>JavaScript</h1></header>
    <p>...</p>
  </article>
  <footer>This article was last modified on June 11, 2016</footer>
</section>

```

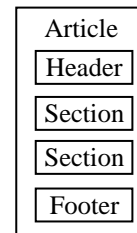


Similarly, an article may have different sections. So, <section> elements may be nested in <article> elements.

```

<article>
  <header><h1>Web Tutorial</h1></header>
  <section>
    <header><h1>HTML</h1></header>
    <p>...</p>
  </section>
  < section >
    <header><h1>JavaScript</h1></header>
    <p>...</p>
  </section>
  <footer>This article was last modified on June 11, 2016</footer>
</article>

```



In fact, there is no straightforward way to decide how to nest elements.

<aside>

The <aside> element is like a sidebar. It defines contents related to main content but slightly different.

```

<article>
  <header><h1>HTML5 Tutorial</h1></header>
  <p>...</p>
  <aside>
    For details about aside element, see W3C site
  </aside>
</article>

```

Here <aside> element is a part of <article> tag. However, <aside> may appear outside as well.

```

<article>
  <header><h1>HTML5 Tutorial</h1></header>
  <p>...</p>
</article>
<aside>
  See W3C site for details
</aside>

```

<figure> and <figcaption>

The element <figure> defines standalone content typically an image or a video. The <figcaption> defines the label of the figure. Here is an example:

```

<figure>
  
  <figcaption>Fig 1.1: W3C logo</figcaption>
</figure>

```

<summary>

As its name indicates, it defines a summary of detailed content. Typically embedded inside `<section>` or `<article>` element.

```
<article>
  <header><h1>HTML5 Tutorial</h1></header>
  <p>...</p>
  <summary>
    HTML5 introduced a number of powerful features. ...
  </summary>
</article>
```

`<details>`

It is used to define detail content of a document, or a part of a document. Here is an example:

```
<article>
  <header><h1>HTML5 Tutorial</h1></header>
  <details>
    <p>...</p>
  </details>
  <summary>
    HTML5 introduced a number of powerful features. ...
  </summary>
</article>
```

`<nav>`

The `<nav>` element is used to specify a set of related navigation links.

```
<h1>Web Technologies</h1>
<nav>
  <a href='html.html/'>HTML</a><br>
  <a href='jsp.html'>JSP</a><br>
  <a href='servlet.jsp'>Servlet</a><br>
  <a href='json.html/'>JSON</a>
</nav>
```

This defines a `<nav>` element with four navigation links.

`<main>`

The `<main>` element represents the main content of a document or application. Typically written within `<body>` or `<section>` tag.

```
<body>
  <main>
    Main content
  </main>
  <!--Other content-->
</body>
```

The `<main>` should not be a child of `<header>`, `<footer>`, `<article>`, `<aside>`, , or `<nav>` elements. There should be one `<main>` element per page.

## 23.5 Graphics

Before HTML5, web developers had only CSS and JavaScript in order to create animations or visual effects for their websites, or they would have to rely on a third-party plug-ins such as Flash. In HTML5, there are many new features which deal with graphics on the web. In the next two sections, we shall discuss two primary technologies for graphics; canvas and Scalable Vector Graphics (SVG)

### 23.5.1 Canvas

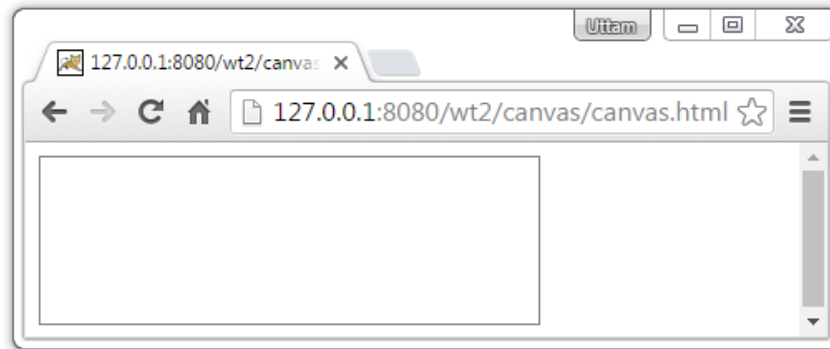
An important part of HTML 5 is `<canvas>` element that allows us to draw a variety of graphic elements and animate them very easily. A canvas is a drawable rectangular region (like painter's canvas) that JavaScript code can access and draw objects using HTML5 canvas API that provides a rich set of drawing functions.

The graphic elements in a canvas can respond to events such as mouse click or key press. Since JavaScript is used to draw them, virtually any kind of animation, simple to extremely complex, can be done. Canvas is a web standard and most current browsers support it, users don't have to install plug-ins such Flash or Silverlight to see the graphics. Applications of canvas include games, graphs, advertisements, art and decoration, education and training etc.

The `<canvas>` element is just an HTML tag like others except that its contents are populated using JavaScript and is created as follows:

```
<canvas id="myCanvas" width="300" height="100" style="border: 1px solid gray;">
  <!-- A message if browser doesn't support canvas -->
  Your browser doesn't support HTML5 Canvas.
</canvas>
```

It has typically two attributes `width` and `height` that defines the area of drawable region and `id` attribute so that it can be accessed via JavaScript code. It may not have a border. Here it is used to



see the boundary. It looks in the browser as shown here:

Inside the `<canvas>` element, a message often placed that is rendered if a browser does not support canvas. A reference to this canvas may be obtained using JavaScript as follows:

```
canvas = document.getElementById('myCanvas');
```

Now `canvas` refers to the drawing region. We then specify what kind of objects (2-D or 3-D) we are going to draw (i.e. which specific API we want to use). This is done by getting an API context using `getContext()` method specifying the kind of context (`"2d"` or `"webgl"` for 2-D or 3-D respectively) method as follows:

```
context = canvas.getContext('2d');
```

This gets a 2-D context which can be used to draw only 2-D graphic elements. This `context` object has a rich set of drawing methods that can be used to draw desired graphic objects. The script may also check if a browser supports HTML5 canvas API as follows:

```
if (canvas.getContext){
  context = canvas.getContext('2d');
  // draw objects
} else {
```

```

    alert('Your browser does not support canvas API');
}

```

To get the complete set of functions on `context` object, following piece of code is very useful:

```

for(i in context)
    document.writeln(i+' '+context[i]+'<br>');

```

Following sections describe how to draw different kind of graphic objects.

### 23.5.1.1 Rectangle

Rectangle is probably the easiest shape that we can draw using HTML5 canvas. Two methods are available: `strokeRect()` for rectangular outline and `fillRect()` for filled rectangles. Both take the coordinates of top-left corner, width and height of the rectangle. Here is an example:

```

context.strokeStyle = 'black';
context.strokeRect(10,10, 150,100);

context.fillStyle = 'gray';
context.fillRect(20,20, 130,80);

```



This first draws a rectangle with line color black and top-left corner at (10, 10) and width and height 150 and 100 pixels respectively. It then draws another rectangle filled with gray color. A sample result is shown here. The thickness of the line is changes using `lineWidth` attribute of the context.

```

context.lineWidth = 4;

```

This makes the outer rectangle as follows.

To clear a rectangular area, `clearRect()` is used.

```

context.clearRect(40, 40, 90, 40);

```



### 23.5.1.2 Paths

Although, HTML5 canvas API provides functions to draw rectangles, any other shape must be drawn using paths. A path is a series of line/curve segments having same or different properties. A path can create arbitrary shapes such as triangle, polygons, circles etc. A *subpath* is a segment of path having typically similar properties. For example, we can draw two triangles (two subpaths) with different line colors. A shape drawn using path takes the following form:

```

context.beginPath();
//... draw paths here
context.closePath();

```

This creates a logical segment. Drawing properties (stroke color, width etc) specified in a segment only apply to that subpath. Canvas uses an invisible pen/pointer that moves here and there from time to time. We can set the properties of the pen (color, thickness of pen-tip etc.) or can ask the pen to trace or move to get the desired shape. The pen lifted from the current position and placed at a new position (x, y) using `moveTo()` method. Following places the pen at (0, 100)

```

context.moveTo(0, 100);

```

#### Linear Paths

The `lineTo()` method instructs the pen to create a linear path from its current location to the specified location. Since, above code places the pen at (0, 100), the code below creates a linear path from (0, 100) to (50, 0)

```

context.lineTo(50, 0);

```

Note that a path is actually traced or a closed path is filled in by some color using other methods such as `stroke()`, `fill()` etc. Following creates a triangle:

```
context.beginPath();
context.moveTo(0,100);
context.lineTo(50, 0);
context.lineTo(100,100);
context.closePath();
context.stroke();
```



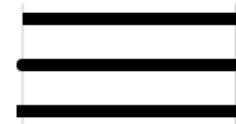
It starts from the point (0, 100), goes along a straight line to (50, 0) and again to (100, 100). The `closePath()` method automatically closes the path to form a triangle. The `stroke()` method actually draws those lines. It results in the shape shown to the left. Instead of `stroke()`, `fill()` would produce the shape shown to the right. The thickness of the stroke may be changed using the `lineWidth` property of the context. Following sets the stroke thickness of 2 pixels:

```
context.lineWidth=2;
```

Here is a picture showing 4 lines with line-width 2, 4, 6 and 8. Note that the `stroke()` method uses the properties set last. Therefore, the `stroke()` method must be invoked after setting a property to get the desired effect.

The end style of a line may be specified using the `lineCap` property which can have one of three values: 'butt' (default), 'round' and 'square'. Following specifies round end style:

```
context.lineCap='round';
```



The picture shows three lines with different end styles. For 'butt' and 'square', the end style looks the same except that for the latter the line extends by an amount `lineWidth/2` at each end. The vertical lines are shown for clarity.

There is another property called `lineJoin` that controls the appearance of the junction from where two or more thick lines emerge. Possible values are 'miter', 'bevel' and 'round'. Following specifies rounded junction point:

```
context.lineJoin = "round";
```

The picture shows three possible junction styles.



## Curved Paths

To draw curves, canvas provides two methods: `arc()` and `arcTo()`.

```
arc(cx, cy, radius, startAngle, endAngle, anticlockwise)
arcTo(cpx1, cpy1, cpx2, cpy2, radius)
```

The first draws an arc centered at (cx, cy) with radius r from startAngle radian to endAngle radian in the direction specified by anticlockwise (default is false). The second one draws an arc with two control points (cpx1, cpy1) and (cpx2, cpy2) having radius radius connected to the current point by a straight line. Following creates a circle centered at (100, 100) having radius 80 using `arc()`.

```
context.arc(100, 100, 80, 0, Math.PI, true);
```

A sample result is shown here.

Canvas also allows drawing Bezier curves. Two methods



`quadraticCurveTo()` and `bezierCurveTo()` have been provided to draw quadratic and cubic Bezier curves respectively.

```
quadraticCurveTo(cpx1, cpy1, toX, toY)
bezierCurveTo(cpx1, cpy1, cpx2, cpy2, toX, toY)
```

The former one draws a quadratic Bezier curve from the current point to `(toX, toY)` having with control point `(cpx1, cpy1)`. The second one draws a cubic Bezier curve from the current point to `(toX, toY)` with two control points `(cpx1, cpy1)` and `(cpx2, cpy2)`. Here is an example:

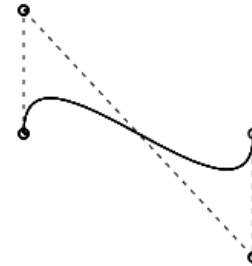
```
context.moveTo(100, 100);
context.quadraticCurveTo(150, 50, 300, 100);
```



This draws a quadratic Bezier curve from `(100, 100)` to `(300, 100)` with control point `(150, 50)`. The result is shown here. Note that dotted lines and circles are no part of the above code. They have been shown only for illustration.

Following shows how to draw a cubic Bezier curve.

```
context.moveTo(100, 200);
context.bezierCurveTo(100, 120, 250, 280, 250, 200);
```



This draws a cubic Bezier curve from `(100, 200)` to `(250, 200)` with control points `(100, 120)` and `(250, 280)`. The result is shown here. Note that dotted lines and circles are no part of the above code. They have been shown only for illustration.

### 23.5.1.3 Styled text

Canvas supports methods to draw text with custom font, style, sizes, colors etc. The property of the text is set using canvas's `font` property. Two methods are provided:

```
fillText(text, x, y [, maxWidth])
strokeText(text, x, y [, maxWidth])
```

Both take similar parameters. They draw given `text` as solid and outlined text respectively. The position of text is determined by coordinates `(x, y)` and `textAlign` and `textBaseline` properties. The optional `maxWidth` specifies the maximum horizontal space the text must occupy. Here are some examples:

```
context.font = "normal bold 32px Verdana";
context.fillStyle = "red";
context.fillText("Hello World!", 10, 40, 600);
```

```
context.font = "italic small-caps 36px Arial";
context.strokeStyle = "blue";
context.strokeText("Some Text", 20, 80);
```

**Hello World!**  
*SOME TEXT*

Note that the value of `font` property is specified in the same way as the 'font' property of CSS with few restrictions. It results styled text as shown here.

The `textAlign` and controls how `x` and `y` coordinates are interpreted respectively. Their possible values are:

`textAlign` — `start`, `end`, `left`, `right`, `center`. Default is `start`.


`textBaseline` — `alphabetic`, `top`, `hanging`, `middle`, `ideographic`, `bottom`. Default is `alphabetic`.

Following example uses different `textAlign` properties.

```
context.font = "normal bold 20px Verdana";
context.fillStyle = "red";

context.textAlign='start'; context.fillText("start", 200, 20);
context.textAlign='end'; context.fillText("end", 200, 40);
context.textAlign='left'; context.fillText("left", 200, 60);
context.textAlign='right'; context.fillText("right", 200, 80);
context.textAlign='center'; context.fillText("center", 200, 100);

context.beginPath();
context.moveTo(200, 0);
context.lineTo(200, 120);
context.stroke();
```



All texts are drawn with `x=200`. For clarity, a line is also drawn at `x=200`. This results styled text as shown here.

Following example illustrates the effect if `textBaseline` properties:

```
context.font = "normal normal 16px Verdana";
context.fillStyle = "red";

context.textBaseline = "top"; context.fillText("top", 0, 50);
context.textBaseline = "hanging"; context.fillText("hanging", 35, 50);
context.textBaseline = "middle"; context.fillText("middle", 110, 50);
context.textBaseline = "alphabetic"; context.fillText("alphabetic", 180, 50);
context.textBaseline = "ideographic"; context.fillText("ideographic", 270, 50);
context.textBaseline = "bottom"; context.fillText("bottom-glyph", 380, 50);

context.beginPath();
context.moveTo(0, 50);
context.lineTo(490, 50);
context.stroke();
```

All texts are drawn with `y=50`. It generates following styled text. For clarity, a line is also drawn at `y=50`.



#### 23.5.1.4 Images

One of the important features of canvas is the ability to draw images and control its properties dynamically. It can be used for different kind of tasks such as dynamic photo composition, setting background in games, backdrop of graphs etc.

Two steps are involved: create an `Image` object and draw it on canvas using its `drawImage()` method. The `Image` object may be created in different ways. Following creates one using `Image()` constructor:

```
img = new Image(); // Create new img element
img.src = 'W3C.jpg'; // specify image location
```

Alternatively, a reference to an existing image created using `<img>` tag may be obtained:

```
<img id='w3c' src='W3C.jpg'/>
<script language='JavaScript'>
  img = document.getElementById('w3c');
</script>
```

Once we have an `Image` object, we can use any one of the following three overloaded `drawImage()` methods to render it on the canvas.

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

The first one is the simplest one that draws the given `image` at the destination location `(dx, dy)`. The second one takes two additional parameters `dWidth` and `dHeight` parameters, which indicate the target size of the image. In the last one, we can specify area of the source image as a rectangle having top-left corner `(sx, sy)`, width `sWidth` and height `sHeight` to be used for rendering. For demonstration, we shall use first one.

However, before drawing the image, we have to make sure that the image is loaded fully. This can be checked using `onload` event listener as follows:

```
img.onload = function(){
    context.drawImage(img, 100, 100);
}
```

Alternatively, `addEventListener()` method may be used.

```
img.addEventListener('load', draw);
function draw() {
    context.drawImage(img, 100, 100);
}
```



The result is shown here.

#### 23.5.1.5 Gradient

A continuously smooth transition from one color to another is known as gradient. Gradients may be applied to properties (fill and stroke) of a shape. Canvas supports two types of gradients: *linear* and *radial*. Linear gradient changes color in one direction, either horizontally, vertically, or diagonally. Radial gradient changes color circularly in or out. To fill or stroke shapes using gradient, we first create a `CanvasGradient` object using one of the following methods on context object:

```
createLinearGradient(x1, y1, x2, y2)
createRadialGradient(x1, y1, r1, x2, y2, r2)
```

The former creates a linear gradient object from `(x1, y1)` to `(x2, y2)`. Later creates a radial gradient from one circle having center `(x1, y1)` and radius `r1` to another circle having center `(x2, y2)` and radius `r2`. Following respectively creates horizontal, vertical and diagonal linear gradients:

```
hg = context.createLinearGradient(0,25,150,25);
vg = context.createLinearGradient(75,0,75,50);
dg = context.createLinearGradient(0,0,50,50);
```

We then specify the colors of the gradient using `addColorStop()` method.

```
hg.addColorStop(0, 'white');
hg.addColorStop(1, 'black');
```

The first parameter is a number between 0 and 1 that represent start and stop position of the gradient. A value 0.5 indicates middle of the gradient range. Above specifies that the white color should applied to the beginning (first parameter 0) of the gradient area and black should be applied to the end (first parameter 1). We can then apply this gradient to fill or stroke shapes.

```
context.fillStyle = hg;
```





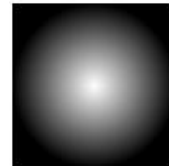
```
context.fillRect(0,0,150,50);

context.strokeStyle = hg;
context.strokeRect(0,60,150,60);
```

This results a picture as shown here.

The radial gradient is created using `createRadialGradient()` method. Following illustrates this:

```
rg = context.createRadialGradient(50, 50,0, 50, 50, 50);
rg.addColorStop(0, 'white');
rg.addColorStop(1, 'black');
context.fillStyle = rg;
context.fillRect(0,0,100,100);
```



It results the graphics as shown here.

#### 23.5.1.6 Transformation

We can apply certain transformations such as translation, rotation and scaling to the canvas. Note that canvas transformation is different from transformation applied to SVG (discussed later in this chapter) where specific drawing elements may be transformed. However, for canvas, what we transform is not an element but the coordinate system of the canvas. So, after a transformation, graphic elements are drawn according to the new coordinated system. Note further that the canvas area itself is not transformed.

Three kinds of transformations may be applied: *translation*, *rotation* and *scaling*. Two ways to do them: using respective methods such as `translate()`, `rotate()`, `scale()` or using a general method `transform()` that can perform all kind of transformations.

We shall first look at the `translate()` method that moves the origin of the canvas's coordinate system to new location. It takes the following form:

```
translate(dx, dy)
```

Here, `dx` and `dy` are respectively the horizontal and vertical distance to move. Here is an example:

```
context.translate(40, 20);
```

This moves the origin by 40 and 20 pixels along `x` and `y` axis respectively. To understand how it affects drawings, consider following code:

```
context.fillStyle = 'gray';
context.fillRect(20,20, 200, 100);
context.translate(40, 20);
context.fillStyle = 'lightgray';
context.fillRect(20,20, 200, 100);
```

We first draw a rectangle filled with gray color and translate the coordinate system. We then use the same coordinates to draw a new rectangle filled with light gray color. However, since the coordinate system has been translated in the meanwhile, it appears at the different position. The output looks like this. Note that shapes drawn after the `translate()` function are only affected.



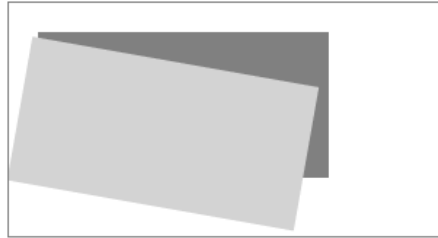
The coordinate system can be rotated using `rotate()`

that takes following form:

```
rotate(angle)
```

Here angle is the degree of rotation in radians.

```
context.fillStyle = 'gray';
context.fillRect(20,20, 200, 100);
context.rotate(Math.PI/18);
context.fillStyle = 'lightgray';
context.fillRect(20,20, 200, 100);
```



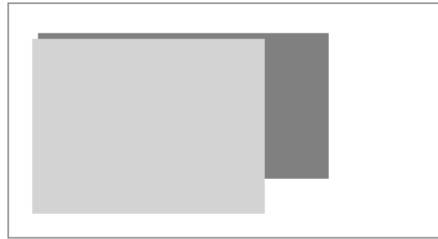
It is same as previous example except that instead of `translation()`, `rotation()` is used that rotate the coordinate system by 10°. It generates following output.

The `scale()` method scales the units of coordinate system up or down by the specified amount. It looks like this:

```
scale(x, y)
```

It scales the units respectively by the factors x and y horizontally and vertically. Both parameters are real numbers. Values less than and greater than 1 respectively increase and decrease the units. Let us take an example to illustrate this:

```
context.fillStyle = 'gray';
context.fillRect(20,20, 200, 100);
context.scale(0.8, 1.2);
context.fillStyle = 'lightgray';
context.fillRect(20,20, 200, 100);
```



This results a picture as shown here.

The `transform()` method may be used to perform any kind of transformation including translation, rotation and scaling. It takes following form:

```
transform(a, b, c, d, e, f)
```

It essentially transforms coordinates (x, y) to (x', y') according to the following equation:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

So, `translation(40, 20)` may be represented as `transform(1, 0, 0, 1, 40, 20)`. Other transformation may be figured out similarly. Note that the effect of `transform()` method is cumulative. Canvas provides another method `setTransform()` method that transforms always the absolute (initial) coordinate system.

## 23.5.2 SVG

SVG (Scalable Vector Graphics), key part of HTML5, is a XML based technology for specifying 2-D graphic elements (such as line, circle, rectangle, ellipse etc.) and their operations (rotation, translation etc.) for the web. A SVG viewer (modern browsers have such a component) reads SVG tags/file and renders the graphic element on the screen like JPEG/GIF/TIFF/PNG image. In fact the HTML5 logo originated as an SVG file.

Since SVG images are vector images, they have several advantages over other raster counterparts.

- Raster images store information about the image pixels. Vector images store mathematical formulas or procedures to render them. Hence the size of a vector image is very very less than that of a raster image. Since, file size is smaller, SVG images are not only loaded very quickly, they consume less bandwidth also.
- The SVG source file is an ordinary text file. So, it can be edited using an ordinary text editor. They can also be searched, indexed and scripted.
- SVG images can be created and manipulated (animated) programmatically (using say JavaScript, Servlet, JSP, PHP etc) in real time. So, they are suitable to generate graphs, charts, diagrams dynamically for the web.
- Scaling a vector image does not degrade the image quality. A raster image, on the other hand, store information about a fixed num of pixels. Scaling it up requires finding information of additional pixels from existing pixels. That's why for larger scaling factor, the quality of the resultant image is degraded.



SVG images are not suitable for irregular shapes, photos, movies. They are merely used to draw regular shapes.

### 23.5.2.1 SVG Coordinate system

Unlike Cartesian coordinate system whose (0, 0) is at bottom-left corner of the page/graph and x-axis pointing to the right and the y-axis pointing up, SVG uses a different convention. Here, (0, 0) point is at top-left corner and x-axis pointing to the right and the y-axis pointing down.

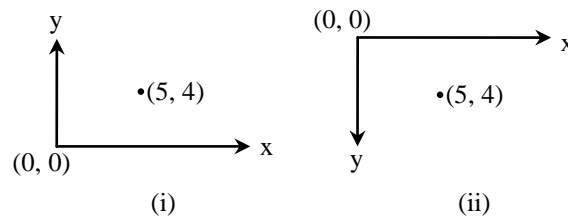


Figure 23.1: (i) Cartesian vs. (ii) SVG coordinate system

The unit of the length/coordinate may be specified just after the coordinate. Supported units are `px` (pixel), `pt` (`=1.25px`), `pc` (`=15px`), `cm`(centimeter), `mm`(millimeter), `in`(inch), `em`, `ex` and percentages. The units `em` and `ex` are relative to the current font's font-size and x-height, respectively. If none is specified, it is assumed to be `px`.

### 23.5.2.2 The <svg> element

The container of all SVG images is `<svg>` element that looks like:

```
<svg xmlns="http://www.w3.org/2000/svg">
...
</svg>
```

Nested `<svg>` elements are possible. They are used to create different coordinate system.

```
<svg xmlns="http://www.w3.org/2000/svg">
  <svg x="50" y="50">
    ...
```

```

</svg>
<svg x="50" y="100">
  ...
</svg>
</svg>

```

This creates two coordinate systems with origins at (50, 50) and (50, 100) respectively. Coordinates of elements under a `<svg>` are relative to that coordinate system.

### 23.5.2.3 Line

The most fundamental graphic element is line which is created using `<line>` tag that attributes `x1`, `x2`, `y1`, `y2` and draws a line from (`x1`, `y1`) to (`x2`, `y2`). Here is an example:

```

<svg height="100">
  <line x1="10" y1="10" x2="200" y2="50" style="stroke:red;stroke-width:4"/>
</svg>

```

This draws a line from (10, 10) to (200, 50). The graphic area has the height 100 pixels. The style attribute says that the line be in red color with thickness 4 pixels. A sample result is shown here.



### 23.5.2.4 Rectangles and squares

Rectangles (and squares) are drawn using `<rect>` element. It takes `x`, `y`, `height` and `width` attributes which are coordinates of the top-left corner, height and width of the rectangle respectively. Here is an example:

```

<svg width="300" height="100">
  <rect x="20" y="20" width="200" height="50" style="fill:rgb(0,255,0);stroke-
width:4;stroke:rgb(0,0,0)" />
</svg>

```

The rectangle is drawn with black color and filled in by green color. Sample output is shown here. Rounded rectangle can be drawn specifying `rx` and `ry` attributes. Here is an example:



```

<svg width="300" height="100">
  <rect x="20" y="20" width="200" height="50" rx="10" ry="10"
style="fill:rgb(0,255,0);stroke-width:4;stroke:rgb(0,0,0)" />
</svg>

```

The result is shown here:



### 23.5.2.5 Circles

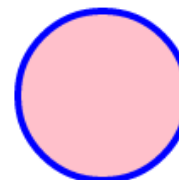
The `<circle>` element is used to draw a circle. It takes `cx`, `cy` and `r` attributes. These are coordinates of the center and radius respectively. Following shows how to create a circle and style it:

```

<svg width="150" height="150">
  <circle cx="60" cy="60" r="50" stroke="blue" stroke-width="4" fill="pink" />
</svg>
<br>

```

This draws a circle with center at (60, 60) and radius 50. If `cx` and `cy` are not specified, they are assumed to be (0, 0). Here is a sample output.



### 23.5.2.6 Ellipses

Ellipses are drawn using `<ellipse>` tag. It takes attributes `cx`, `cy`, `rx`, and `ry` which are coordinates of the center and x-axis radius and y-axis radius respectively. Here is an example:

```
<svg height="100">
  <ellipse cx="100" cy="50" rx="100" ry="50" fill="gray" />
</svg>
```

If `cx` and `cy` are not specified, they are assumed to be (0, 0). Here is a sample output.



### 23.5.2.7 Polyline

This drawing consists of multiple lines which is rendered using `<polyline>` element. It takes an attribute `points` whose value is a set of space separated points of the form `x,y`. Here is an example:

```
<svg height="50">
  <polyline points="0,50 0,0 40,0 40,50 80,50 80,0 120,0 120 50"
  style="fill:white;stroke:black"/>
</svg>
```

This generates a square wave as shown here. Note that a single point not enough to draw a line and hence nothing is displayed.



### 23.5.2.8 Polygon

A polygon, which has three or more sides is created using `<polygon>` element. Like `<polyline>`, it also takes a `points` attribute whose value is a set of space separated points of the form `x,y`. However, the last and the first point is connected to form a closed shape. Following creates hexagon.

```
<svg id="svgelem" height="200"
xmlns="http://www.w3.org/2000/svg">
  <polygon points="0,40 60,0 120,40 120,120 60,160 0,120"
  fill="red" />
</svg>
```



This results the following:

### 23.5.2.9 Path

A path in SVG is the outline of a shape and is created using `<path>` element. It can create fairly complicated shapes (closed or open) which are not possible using previously discussed elements. It takes an attribute `d` (path data) where data and commands for creating paths are specified. The `<path>` element uses *virtual pen* to draw outline of a shape. The commands (move the pen, draw a line, draw an arch etc.) are specified in the `d` attribute. Here is an example:

```
<svg>
  <path d="M0,0 L50, 50 L100, 0 z" fill="none" stroke="red" />
</svg>
```



This results as shown in the figure. Although, this shape could have been produced using `<polyline>` or `<polygon>` elements, we shall use it to understand how to use `<path>` element. Notice the value of `d` attribute. This instructs a *virtual pen* to first **M**ove at coordinate (0,0), then draw a **L**ine to (0, 100), then draw another **L**ine to (100, 0) and finally close (**z**) the path. Table 23.4: shows a list of possible command letters, their meaning and parameters taken. A command letter may be in uppercase or lowercase. Coordinates for uppercase commands are absolute whereas coordinates for lowercase commands are relative to the current position of the pen. For

example, if current position is (x1, y1), Lx2, y2 command draws a line from (x1, y1) to (x2, y2) and lx1, y1 draw a line from (x1, y1) to (x1+x2, y1+y2).

Table 23.4: SVG <path> element commands

Cmd	Function	Parameters	Description
M	Move to	(x y)+	Lift the pen and position at coordinate (x, y)
L	Line to	(x y)+	Draws a line from current point to specified point.
H	Horizontal Line to	x+	Draws a horizontal line from the current point (cpx, cpy) to (x, cpy).
V	Vertical Line to	y+	Draws a vertical line from the current point (cpx, cpy) to (cpx, y).
C	Curve to	(x1 y2 x2 y2 x y)+	Draws a cubic Bezier curve from current point to (x,y) with (x1,y1) and (x2,y2) as the control points
S	Smooth/Shorthand Curve to	x2 y2 x y)+	Draws a cubic Bezier curve from current point to (x,y) with (x2,y2) as second control point. First control point is the reflection of the second control point of previous command
Q	Quadratic Bezier curve to	(x1 y1 x y)+	Draws a quadratic Bezier curve from current point to (x,y) with (x1,y1) as the control point.
T	Smooth/Shorthand Quadratic Bezier Curve to	(x y)+	Draws a quadratic Bezier curve from current point to (x,y). the control point is assumed to be reflection of the control point of previous command
A	Elliptic curve to	(rx ry x-axis-rotation large-arc-flag sweep-flag x y)+	Draws an elliptical curve segment from the current point to (x, y).
Z/z	Close path		

### Positioning the pen

The pen is lifted and positioned to a new location using M (moveto) command. It must be the first command for any path segment. Following positions the pen at (0, 0):

```
<path d="M0,0"/>
```

For each subsequent pair of coordinates, a lineto command is assumed. Following draws a line from (0, 0) to (0, 100):

```
<path d="M0,0 L0,100" stroke="red"/>
```

It is equivalent to <path d="M0,0 L0,100" stroke="red"/>. The pen is set at the end point of previous drawing. So, the above sets the pen finally at (0, 100).

### Drawing Lines

The L(or l) command draws a line tracing the pen from current position to the specified location. Here is an example:

```
<path d="M0,0 L0,100" stroke="red"/>
```

It drawn a line from (0, 0) to (0, 100). Polyline may be drawn using a set of locations.

```
<path d="M0,0 L100,100 200, 100" stroke="red" fill="none"/>
```

Two specific kind of lines can be drawn using H (or l) and V(or v) commands which draw horizontal and vertical lines respectively. They take x and y coordinate respectively. Following draws a horizontal line from (0, 0) to (100, 0):

```
<path d="M0,0 H100" stroke="red"/>
```

Following draws a vertical line from (0, 0) to (0, 100):

```
<path d="M0,0 V100" stroke="red"/>
```

## Drawing Curves

Three kinds of curves can be drawn using `<path>` element.

- Cubic Bezier curves (defined by a start point, an end point, and two control points) using (C, c, S and s)..
- Quadratic Bezier (defined by a start point, an end point, and one control point) using commands (Q, q, T and t)..
- Elliptical curves (a segment of an ellipse) using (A and a).

Following draws a cubic Bezier curve from (100, 200) to (250, 200) having control points at (100, 150) and (250, 250):

```
<path d="M100,200 C100,150 250,250 250,200 "stroke="red" fill="none"/>
```

It results as shown in the picture. For clarification, the control points are also shown. A second Bezier curve can be joined smoothly using S command. It takes only one control point. The first control point is the reflection of the second control point on the previous command. Here is an example:

```
<path d="M100,200 C100,150 250,250 250,200 S400,300 400,200"
stroke="red" fill="none"/>
```

It results the curve as shown here:

The Q command on the other hand takes two points and draws a quadratic Bezier curve from current point to second parameter. The first point is the control point. The T command draws a smooth quadratic Bezier curve from current point to the specified point. The control point is assumed to the reflection of the control point of the previous command. Here is an example:

```
<path d="M100,100 Q100,50 200,100 T300,100"
fill="none" stroke="red" stroke-width="1" />
```

It first draws a quadratic Bezier curve from (100, 100) to (200, 100) with control point as (100, 50) and draws a second curve from (200, 100) to (300, 100) with control point (300, 150). The result is shown here. For convenience, all the points are shown.

An elliptic curve segment is drawn using A (or a) command that takes a series of parameters of the form

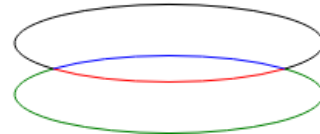
```
A(rx ry x-axis-rotation large-arc-flag sweep-flag x y)+
```

The first set (rx, ry) is the x-axis and y-axis radius. The x-axis-rotation (often set to 0) indicates rotation of the curve's x-axis w. r. t coordinate system's x axis. The binary large-arc-flag determines if the arc should be small (0) or big (1). The sweep-flag indicates which side to

bend. Following draws four arcs for all possible combinations of large-arc-flag and sweep-flag.

```
<path d="M 100,100 A100,25 0 0,0 250,100" style="fill:none; stroke:red; stroke-width:1"/>
<path d="M 100,100 A100,25 0 1,0 250,100" style="fill:none; stroke:green; stroke-width:1"/>
<path d="M 100,100 A100,25 0 0,1 250,100" style="fill:none; stroke:blue; stroke-width:1"/>
<path d="M 100,100 A100,25 0 1,1 250,100" style="fill:none; stroke:black; stroke-width:1"/>
```

It results following shapes:



### 23.5.2.10 Drawing text

SVG allows us to draw styled text in web pages. Text is rendered as graphic element. Therefore, all the graphic transformation can be applied to it. Its text property remains as it is. So, text related operation such as copying, pasting, selecting etc. can be done. The `<text>` element is used to draw a text.

```
<svg width="300" height="100">
  <text x="20" y="50" style="fill:olive; font-size:50px;">
    Hello World!
  </text>
</svg>
```

Hello World!

This takes x and y attributes which are the coordinates of the first character of the text. This result following result:

#### `<tspan>`

The different portion of text created using a `<text>` element may be reformatted/repositioned using its child `<tspan>` element. For example, a line of text may be split into two lines:

```
<text y="50" style="fill:olive; font-size:50px;">
  <tspan x="20">Hello</tspan>
  <tspan x="20" dy="50">World!</span>
</text>
```

Hello  
World!

It looks as shown here:

We can even create subscripted/superscripted text using CSS property:

```
<text x="20" y="50" style="fill:olive; font-size:50px;">
  <tspan>x</span>
  <tspan x="50" style="baseline-shift: super;font-size:30px;">i
  <tspan x="50" style="baseline-shift: sub;font-size:30px;">j
</text>
```

x<sup>i</sup><sub>j</sub>

This results following styled text:

Note that `<tspan>` is a child of `<text>` element. So, texts of different `<tspan>` under a single `<text>` element can be selected once and copied.

#### `<use>`

It allows us to create a named text segment using `<defs>` and then use it later as many times as we wish. Here is an example:

```
<defs>
  <text id="hello">Hello</text>
</defs>
<use x="50" y="20" xlink:href="#hello" /><text x="90" y="20">Tom</text>
```



```
<use x="50" y="50" xlink:href="#hello" /><text x="90" y="50">World</text>
```

The `<text>` element of `<defs>` has an `id` attribute which can be used to refer to the text. In the above example, the text "Hello" has been referred twice. Note that elements in the `<defs>` tag are not displayed. The result looks like this:

Hello Tom  
Hello World

### `<textpath>`

It allows us to draw a text along a path. The path is first created having an `id`. `<textpath>` refers to the path to be followed by specified text. Here is an example:

```
<path id="aPath" d="M100,200 C200,100 250,300 400,200" fill="none"/>
<text style="font-size:20px;">
  <textPath xlink:href="#aPath" >
    SVG stands for Scalable Vector Graphics.
  </textPath>
</text>
```

SVG stands for Scalable Vector Graphics.

We have first drawn an invisible cubic bezier curve from (100, 200) to (400, 200) with control points (200, 100) and (250, 300). The text is then drawn along this curve. A sample result is shown here.

### 23.5.2.11 Image

Although SVG images are vector images, we can refer to a raster image using `<image>` element as follows:

```
<svg viewBox="00 0 100 100" >
  <image xlink:href="W3C.jpg" x="10" y="0" height="50" width="50"/>
</svg>
```

The URL of the image is specified as the value of `xlink:href` attribute. The above example results following:



### 23.5.2.12 `<a>` tag

Like HTML `<a>` tag, any SVG graphic element can be made as a link using SVG `<a>` tag. The URL of the link is specified as the value of `xlink:href` attribute. Here is an example:

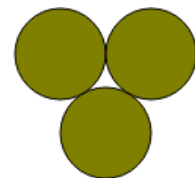
```
<a xlink:href="index.html">
  <text x="100" y="20">index.html</text>
</a>
```

The above example creates a text as hyperlink. However, any drawing element can be made as a link.

### 23.5.2.13 `<symbol>`, `<g>` `<defs>`

A graphic object may appear multiple times. Although, we can insert it multiple times, if we decide to change its properties, we have to do it for all elements. SVG `<symbol>` element avoids this problem by allowing us to formally declare one element or more elements under it having an `id` and using them as time times as we want. To change the property of the object, we change it in the element declaration. Here is an example:

```
<symbol id="aCircle" >
  <circle cx="50" cy="50" r="30" stroke-width="1" stroke="black" fill="olive"/>
</symbol>
<use xlink:href="#aCircle" x="0" y="0" />
<use xlink:href="#aCircle" x="60" y="0" />
<use xlink:href="#aCircle" x="30" y="52" />
```



It first declares a circle with center at (50, 50) and radius 30 and creates three such circles at designated place. It results as shown here:

The declaration of the circle could have been made using `<g>` (grouping) element as follows:

```
<g id="aCircle" >
  <circle cx="50" cy="50" r="30" stroke-width="1" stroke="black" fill="olive"/>
</g>
```

However, `<symbol>` element has `preserveAspectRatio` and `viewBox` attribute and seems a better way to group shapes and reuse them.

A related SVG element is `<defs>`. It allows us to declare named objects first and use them later. The above declaration could have been made using `<defs>` element as follows:

```
<defs>
  <circle id="aCircle" cx="50" cy="50" r="30" stroke-width="1" stroke="black"
    fill="olive"/>
</defs>
```

Note that `<defs>` element does not have any `id` attribute. It merely holds named element declarations to be used later.

#### 23.5.2.14 Viewport and viewBox

We know that default unit of lengths/coordinates is `px` (pixel). However, we redefine what coordinates/lengths really mean using `viewBox` attribute of `<svg>` element. This allows us to fit graphic elements within a given area. The value of the `viewBox` attribute is a list of four numbers `min-x, min-y, width` and `height`. Here is an example:

```
<svg width="400" height="300" viewBox="0 0 200 100" >
  <rect x="10" y="10" width="100" height="50"
    style="stroke: black; fill:none;stroke-width:1"/>
</svg>
```

This creates an SVG element of viewport 400 pixel X 300 pixel. However, the `viewBox` attribute maps the window that goes from (0, 0) to (200, 100). So, 1 unit now represents  $400/200 = 2$  pixels along axis and  $300/100 = 3$  pixels along y-axis. The rectangle is actually drawn from top-left corner (20, 30) to bottom-right corner (220, 180) in absolute coordinate system.

#### 23.5.2.15 Scripting

SVG elements may be accessed and manipulated using JavaScript in the ordinary way. They also respond to events. Following example shows how to increase the y-axis radius (`ry`) of an ellipse every time a button is pressed.

```
<svg width="500" height="400">
  <ellipse id="anEllipse" cx="100" cy="200" rx="100" ry="50" fill="gray" />
</svg>
<input type="button" value="Change" onclick="changeRy()" />
<script>
  function changeRy() {
    ry = parseInt(document.getElementById("anEllipse").getAttribute("ry"));
    document.getElementById("anEllipse").setAttribute("ry", (ry+10)+"");
  }
</script>
```

#### 23.5.2.16 Transformation

Several kind of transformations such as `translate`, `rotate`, `scale` etc. may be applied on SVG graphic elements or container elements such as `<g>`, `<symbol>` etc. If applied on container

elements, all the contained elements are transformed. SVG specification defines, four specific transformation functions `translate()`, `rotate()`, `scale()` and `skew()` and one general purpose function `matrix()`. Note that these functions actually do not transform the elements. Instead they transform the underlying coordinate system (and as a result the elements may be deformed).

## Translation

Simplest kind of transformation is translation that moves an element from one place to another and it takes the following form:

```
translate(<tx> [<ty>])
```

Here `tx`, and `ty` are the amount of translation along `x`, and `y` axes respectively. Last one is optional (default value is zero). Here is example:

```
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none" />
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none"
transform="translate(50, 20)"/>
```

The second ellipse is translated version of first one by an amount (50, 20). For clarity the first one is also shown. Also note that the line and two heavy points at its two ends are not the part of this code. Here is the result:



## Rotation

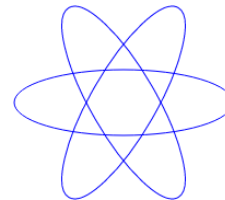
The `rotate()` function rotates an element and takes the following form:

```
rotate(<rotate-angle> [<cx> <cy>])
```

It rotates an element by `rotate-angle` degree about the point (`cx`, `cy`). Here is an example:

```
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none" />
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none"
transform="rotate(60 200 100)"/>
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none"
transform="rotate(120 200 100)"/>
```

This results a beautiful picture as shown here. This figure is created using three ellipses having center at (200, 100). The first one is created without any rotation. Last two are rotated by angles  $60^\circ$  and  $120^\circ$  w. r. t. the point (200, 100).



## Scale

Scaling is a kind of transformation that scales up (enlarges) or down (shrinks) shapes and is done using `scale()` function that takes following form:

```
scale(<sx> [<sy>])
```

Here `sx` and `sy` are scale factors along `x` and `y` axes respectively. If `sy` is not specified, it is assumed to equal to `sx`; hence will be a uniform scaling. Uniform scaling does not deform a shape, it simply increase or decreases the dimension. Here is an example:

```
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none" />
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none"
transform="scale(1.2, 1.2)"/>
```

It scales an ellipse, positioned at (200, 100) with `x` and `y` radius 100 and 30, up uniformly by a factor 1.2 in both `x` and `y` direction. The resultant ellipse will have center at (240, 120) with `x` and `y` radius 120 and 36 respectively and is shown in the picture. For clarity, both ellipses (without and with scaling) are



shown here. Note both the position and size of the ellipse are changed .

### Multiple transformation:

Multiple transformation may be applied repeatedly to have the desired effect. Here is an example:

```
<path d="M200,100 L300, 100 350, 150 250, 150 z" stroke="red" fill="none"/>
<path d="M200,100 L300, 100 350, 150 250, 150 z" stroke="red" fill="none"
transform="translate(350,0) scale(-1, 1)"/>
```

Following first reflects a parallelogram w. r. t. y axis and then shifts it 350 pixels along x axis. The resultant transformation is equivalent to reflection w. r. t. the line  $y=350$  and is shown in the figure. The line shown here is not a part of above code.



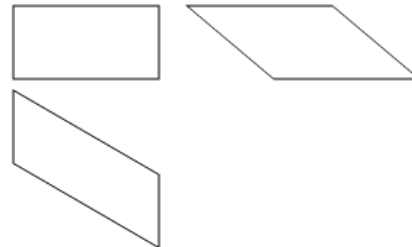
Note that not all pair of transformations is commutative. For example, in the above, translation and scaling pair is not commutative. So, order of operation is important.

### Skew

There are two function `skewX()` and `skewY()` for skewing (shearing) shapes along x and y axis respectively. Both take a single parameter `skew-angle` in degrees. Following demonstrates this:

```
<rect x="100" y="100" width="100" height="50" style="stroke-
width:1;stroke:black;fill:none"/>
<rect x="100" y="100" width="100" height="50" style="stroke-
width:1;stroke:black;fill:none" transform="skewX(50)"/>
<rect x="100" y="100" width="100" height="50" style="stroke-
width:1;stroke:black;fill:none" transform="skewY(30)"/>
```

The first one shows a rectangle without any skew. The second and third show the effect of skew of the first one by  $50^\circ$  along x and y axis respectively. The resultant picture is shown. Note that unlike scaling, skew deforms the original shape.



### Coordinate system transformations

The user space may also be transformed to create a new coordinate system by transforming a container element such as `<svg>`, `<symbol>`, `<g>`, `<marker>`, `<pattern>` etc.

```
<g fill="none" stroke="black" stroke-width="1">
  <line x1="0" y1="0" x2="200" y2="0" />
  <line x1="0" y1="0" x2="0" y2="100" />
  <text x="10" y="30" >Original coordinate system</text>
</g>

<g transform="translate(50,50)" fill="none" stroke="black" stroke-width="1">
  <line x1="0" y1="0" x2="200" y2="0" />
  <line x1="0" y1="0" x2="0" y2="100" />
  <text x="10" y="30" >New coordinate system</text>
</g>
```

The first `<g>` element uses the default coordinate system i.e. origin at (0, 0). The second, `<g>` element creates a new coordinate system by shifting the origin to absolute coordinate (50, 50). So although the same coordinates are used for the second `<text>` element, the text is actually rendered from absolute (50+10, 50+30) i.e. (60, 80) coordinates. Here is the sample result:

Multiple transformations may be applied in similar manner as we do for graphic elements. Nested transformations are possible where transformations apply cumulatively.

### Transformation matrix

In addition to those basic transformation, SVG provides a function `matrix()` that can be used to perform any kind of transformation (including those basic transformations). The function takes six values as

```
matrix(<a>, <b>, <c>, <d>, <e>, <f>)
```

It forms a transformation matrix as follows:

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$$

This maps coordinates (x, y) to (x', y') according to the following matrix equation:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax+cy+e \\ bx+dy+f \\ 1 \end{pmatrix}$$

So, `matrix(1, 0, 0, 1, tx, ty)` is basically represents a translation transformation. Following performs the same translation discussed earlier using `matrix()` function.

```
<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none"
transform="matrix(1,0,0,1,50,20)"/>
```

Following shows other basic transformations in the form of `matrix()` function.

```
rotate(θ) = matrix(cos(θ), sin(θ), -sin(θ), cos(θ), 0, 0)
rotate(θ, x, y) = translate(x, y) rotate(θ) translate(-x, -y) = matrix(1, 0, 0,
1, x, y) matrix(cos(θ), sin(θ), -sin(θ), cos(θ), 0, 0) matrix(1, 0, 0, 1, -x, -
y)
scale(sx, sy) = matrix(sx, 0, 0, sy, 0, 0)
skewX(θ) = matrix(1, 0, tan(θ), 1, 0, 0)
skewY(θ) = matrix(1, tan(θ), 0, 1, 0, 0)
```

### 23.5.2.17 Filters

Once SVG images are created, we can add special effects such as adding shadow, blurring or highlighting to those images. Filters may be applied successively to get the desired effect. SVG provided a rich set of elements for this purpose. These elements start with “fe” followed by the name of the effect. Following are filter elements:

<feDistantLight>	<fePointLight>	<feSpotLight>	<feBlend>
<feColorMatrix>	<feComponentTransfer>	<feComposite>	<feConvolveMatrix>
<feDiffuseLighting>	<feDisplacementMap>	<feFlood>	<feGaussianBlur>
<feImage>	<feMerge>	<feMorphology>	<feOffset>
<feSpecularLighting>	<feTile>	<feTurbulence>	

Following simple example shows how to use Gaussian blur on the input image

```
<defs>
  <filter id="gb" x="-10" y="-10" width="200" height="200">
```

```

    <feGaussianBlur in="SourceAlpha" stdDeviation="10" />
  </filter>
</defs>
<rect x="50" y="50" width="100" height="60" style="filter: url(#gb);" />

```

A Gaussian blur filter is first defined having id gb. A Gaussian blur is defined as

$$G(x, y) = H(x) I(y)$$

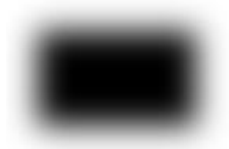
where

$$H(x) = \exp(-x^2 / (2s^2)) / \sqrt{2\pi s^2}$$

and

$$I(y) = \exp(-y^2 / (2t^2)) / \sqrt{2\pi t^2}$$

The parameters  $s$  and  $t$  are the standard deviation along  $x$  and  $y$  axis respectively. The effect of above code is shown here. The detailed description of individual filter element is out of this book.



### 23.5.2.18 Marker

SVG provides a facility to mark certain points (start, middle and end) of a line. The line may be an isolated line or a part of a path or polyline or polygon. The arrowhead is created using <marker> element. Like many other, it is nested inside a <defs> element. Once a marker is defined having an id, it can be referenced from <path>, <line>, <polyline> or <polygon> using marker-start or marker-mid or marker-end styles of respective element. Here is an example:

```

<defs>
  <marker id="point" markerWidth="8" markerHeight="8" refX="5" refY="5">
    <circle cx="5" cy="5" r="3" />
  </marker>
  <marker id="arrow" markerWidth="15" markerHeight="15" refX="3" refY="7"
orient="auto">
    <path d="M3,1 L3,14 15,7" />
  </marker>
</defs>

<path d="M10,10 L100,10" style="stroke: red; marker-start: url(#point); marker-
end: url(#arrow);" />

```



It creates two markers; one circle and another arrow. We then created a line from (10, 10) to (100, 10) and attached the circle marker to the starting point and arrow marker to the end point. A sample result is shown here:

### 23.5.2.19 Gradients

A continuously smooth transition from one color to another is known as gradient. Gradients may be applied to properties (fill and stroke) of a shape. Two kinds of gradient elements are provided by SVG: *linear gradient* and *radial gradient*. There are two steps to work with gradients: define one using <defs> element having an id and refer to it from a shape.

#### Linear Gradients

As the name implies, it changes the color linearly from one color to another and is done using <linearGradient> element. The change can occur along any direction designated by a vector from  $(x_1, y_1)$  to  $(x_2, y_2)$ .

```

<defs>
  <linearGradient id="lg" x1="0%" y1="0%" x2="100%" y2="0%" >
    <stop offset="0%" stop-color="white" />
    <stop offset="100%" stop-color="black" />
  </linearGradient>

```

```
</defs>
<rect x="10" y="10" width="150" height="80" style="fill:url(#lg);stroke:red;"/>
```

The four attributes `x1`, `y1`, `x2` and `y2` of `<linearGradient>` element controls the direction of the gradient. The above indicates a gradient along `x` axis. The two nested `<stop>`-elements controls the colors at the specified offset. In the above, it indicates that at offset 0% and 100% the color will white and black respectively. A sample result is shown here.



A gradient may be transformed before applying it using `gradientTransform` attribute as follows:

```
<defs>
  <linearGradient id="lg" x1="0%" y1="0%" x2="100%" y2="0%"
    gradientTransform="rotate(90)">
    <stop offset="0%" stop-color="white" />
    <stop offset="100%" stop-color="black" />
  </linearGradient>
</defs>
<rect x="10" y="10" width="150" height="80" style="fill:url(#lg);stroke:red;"/>
```

It is exactly same and previous one except the gradient occurs along `y` axis as shown in the picture:



A gradient may be applied on the stroke. Following is an example:

```
<defs>
  <linearGradient id="lg" x1="0%" y1="0%" x2="100%" y2="0%">
    <stop offset="0%" stop-color="pink" />
    <stop offset="100%" stop-color="black" />
  </linearGradient>
</defs>
<rect x="10" y="10" width="150" height="80"
  style="stroke:url(#lg);fill:red;stroke-width: 5" />
```

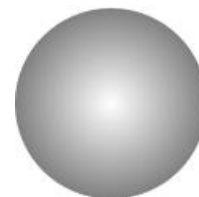


It is rendered on the screen as follows:

### Radial Gradients

The radial gradient starts from a point (focal point) and spreads along all directions. It is achieved using `<radialGradient>` element. Here is an example:

```
<defs>
  <radialGradient id="rg" fx="50%" fy="50%" r="100%">
    <stop offset="0%" stop-color="white" />
    <stop offset="100%" stop-color="black" />
  </radialGradient>
</defs>
<circle cx="60" cy="60" r="60" style="fill:url(#rg); />
```



The focal point is specified by `fx` and `fy` attribute in terms of % of width and height of the area to fill. These are optional with default value 50%. The radius of the gradient is specified by `r` attribute. In the above example, the gradient starts from the center of the circle with white color and changes towards black color.

### 23.5.2.20 Patterns

Like gradients, the `<pattern>` element is used to apply custom patterns on shapes. It is also used in `<defs>` element first and referenced from shapes. Here is an example:

```
<defs>
  <pattern id="aPattern" x="20" y="20" width="20" height="20"
    patternUnits="userSpaceOnUse">
```

```

    <circle cx="10" cy="10" r="10" style="stroke: none; fill: gray" />
  </pattern>
</defs>
<circle cx="60" cy="60" r="60" style="fill:url(#aPattern); />

```

Here one circle is filled with a series of other circles. A sample result is shown here:



### 23.5.3 Animation

SVG animations are similar to CSS animations. However, SVG animations can do something more than CSS animations. The primary animation elements in SVG: `<set>`, `<animate>`, `<animateTransform>` and `<animateMotion>`. They work almost in the same way as they manipulate the attributes of shapes over time to get the animation effect.

#### `<set>`

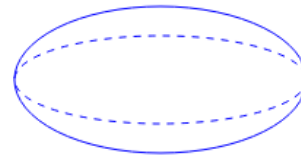
This is the simplest among others. It just sets an attribute to the specified value after the specified duration. Note that the attribute value does not change progressively over time. Following shows an example:

```

<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none">
  <set attributeName="ry" attributeType="XML" to="50" begin="1s" />
</ellipse>

```

The `<set>` element is nested inside the element (here ellipse) to be animated. It sets the y-axis radius `ry` (initial value is 30) to 50 (specified using `to` attribute) after 1 second (specified using `begin` attribute). The attribute to be set is specified using `attributeName` attribute. Note that a SVG shape can have both XML as well as CSS attributes. The value of `attributeType` indicates that `ry` is an XML attribute. So, after one second the ellipse looks as shown here. The initial shape (will not be present finally) is shown for clarity.



#### `<animate>`

It is used to progressively change a single attribute of a shape over time. It is also nested inside the element to be animated. Following demonstrates it:

```

<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none">
  <animate attributeName="ry" attributeType="XML"
    from="30" to="100"
    begin="0s" dur="1s"
    repeatCount="indefinite"/>
</ellipse>

```

It repeatedly changes `ry` attribute of ellipse from 30 (using `from` attribute) to 100 (using `to` attribute). It starts at time 0 (`begin` attribute) and takes 1 second (`dur` attribute). After 1 second, the `ry` is set to 30 and the entire procedure is started over again and again indefinitely.

#### `<animateTransform>`

This sophisticated animation element, as its name indicates, changes the transformation attributes of a shape over time. Following continuously rotates an ellipse with respect to its center.

```

<ellipse cx="200" cy="100" rx="100" ry="30" stroke="blue" fill="none">
  <animateTransform
    attributeName="transform"
    begin="0s"

```

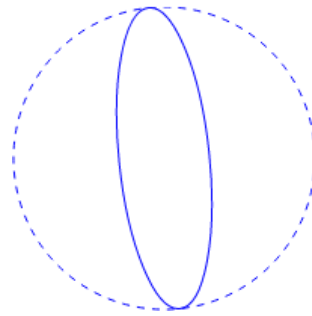


```

        dur="1s"
        type="rotate"
        from="0 200 100"
        to="360 200 100"
        repeatCount="indefinite"
    />
</ellipse>

```

It rotates (specified by `type` attribute) an ellipse from  $0^\circ$  (from attribute) to (to attribute)  $360^\circ$  w. r. t. its center i.e. (200, 100). The rotation starts (begin attribute) at time 0 and takes (dur attribute) 1 second. The entire procedure is repeated indefinitely. The possible values of `type` attribute are `translate`, `scale`, `rotate`, `skewX` and `skewY`. The snapshot of the animation is shown here: Note that the dotted circle is not a part of above code.



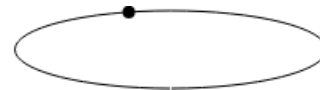
### <animateMotion>

This element may be used to move a shape along a specified path. Here is an example:

```

<circle cx="0" cy="0" r="4" >
  <animateMotion
    path="M 100,100 A100,25 0 1,1 101,100" stroke="blue"
    begin="0s" dur="5s" repeatCount="indefinite"
  />
</circle>

```



This rotates a small filled circle around an ellipse creates using path element. The snapshot of the animation is shown here:

## 23.6 HTML Media

Previously, developers used to use `<object>` and `<embed>` elements to insert media contents in web pages. The code not only looked ugly, it relied on third-party plug-in (such as flash) and end users had to download and install correct version of that plug-in. HTML5 has a competing, open standard for easily and cleanly embedding media into web pages with its native media elements (`<video>` and `<audio>`) and APIs. Now, CSS styles are provided for media elements. For example, we can resize video area on hover using CSS transitions. The elements can be accessed and manipulated using JavaScript.

However, HTML5 video and audio are as easy to download to a hard drive as an `<img>` is now. So, to protect copy some plug-ins are still the best option. There are still some other use-cases HTML5 media cannot handle and plug-ins are still used.

Following two sections provide a detail discussion on audio and video API.

### 23.6.1 HTML Video

The `<video>` element is used to embed a video in a web page. In its simplest form, it looks like this:

```
<video src='movie.mp4' />
```

The `src` attribute points to the video file (an MP4 file here) as relative or absolute path. HTML5 currently only supports MP4, WebM, and Ogg videos.

For browsers that do not support `<video>` tag, we can insert a fallback message directly between opening and closing tags.

```
<video src='movie.mp4'>Your browser does not support video element</video>
```

This code does not do anything except showing the first frame of the movie. We can supply more information such as height, width, controls to play/pause etc.

```
<video src='wildlife.mp4' height='180' width='300' controls>
  Your browser does not support video element
</video>
```

The height and width attributes specify the size of the video's display area in pixels. The Boolean controls attribute provides a control bar typically containing play/pause toggle, seek bar, minimize/maximize button volume control. The appearance differs in different browsers. Here is sample result in chrome browser. The control bar appears/disappears when the mouse pointer enters/leaves the video. We can add more and more attributes (all are optional) as desired. The list of available attributes is shown in Table 23.5:

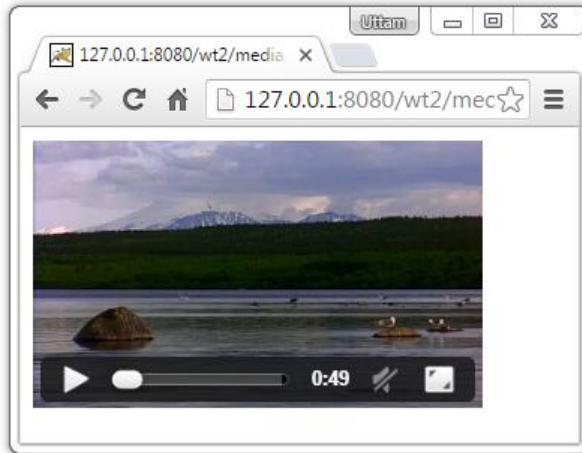


Table 23.5: <video> attributes

Attribute	Description
src	Optional; specifies the URL of the video to insert. The <source> element may be nested to specify the URL.
height	Optional; specifies the height in CSS pixels of the video's display area.
width	Optional; specifies the width in CSS pixels of the video's display area.
controls	Optional; this Boolean attribute adds video playback controls such as adjusting volume, seeking, play/pause etc.
autoplay	Optional; if this Boolean attribute is specified, the video starts playing back automatically soon as it can do and continues loading the data.
poster	Optional; this specifies the URL of an image to display until the playback begins.
preload	Optional; if this attribute gives a hint to browser that it should download video data/metadata. Possible values are auto (or blank), metadata and none. auto - optimistically download the entire video. metadata - Download only metadata (dimensions, first frame, track list, duration, and so on) but not the video itself. none - download is not necessary.
autobuffer	Optional; if this Boolean attribute is present, the video starts buffering automatically even if autoplay is not specified.
loop	Optional; if Boolean attribute is present, the video starts over again every time it completes.

We can optionally include additional information when specifying URL to specify the portion of the video to play. If information is appended to URL in the following format

```
#t=[starttime][,endtime]
```

The time can be specified in seconds as a floating-point number or colon separated hours, minutes and seconds. Here is an example:

```
<video src='wildlife.mp4#t=10,30' height='180' width='300' controls/>
```

It specifies that the video should play from 10<sup>th</sup> second to 30<sup>th</sup> second. The seek button of the control bar also shifts to the appropriate position. Here is a sample output. Following are some other range specification:

#t=,10.5— from the beginning through 10.5 seconds.

#t=,02:00:00—from the beginning through two hours.

#t=,02:30:00—from the beginning through two and half hours.

#t=10,02:30:00—from 10<sup>th</sup> second through two and half hours.

#t=60—60<sup>th</sup> second to the end

Instead of src attribute, we can specify the URL of the video using a nested <source> tag.

```
<video height='180' width='300' controls>
  <source src='wildlife.mp4' />
</video>
```

The advantage is that we can also specify the format of video file if it does not have usual file extension:

```
<video height='180' width='300' controls>
  <source src='wildlife.vid' type='video/mp4' />
</video>
```

We can also have the flexibility to specify multiple video sources. The browser inspects them one by one and plays as soon as a supported one is found.

```
<video height='180' width='300' controls>
  <source src='wildlife.mp4' />
  <source src='wildlife.ogg' />
</video>
```

The two <source> tags specify the MP4 and Ogg video respectively.

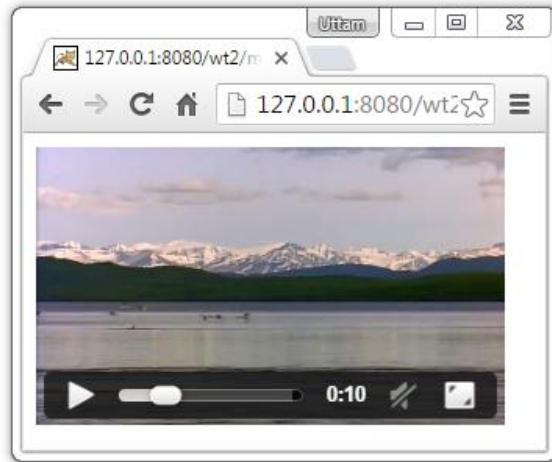
### 23.6.1.2 Programming <video>

We can programmatically control (such as play/pause) the embedded media using JavaScript. HTML5 defines DOM additional methods, properties, and events for the <video> element. A list of methods, properties and events with a brief description can be found in Table 23.6:

A reference to a <video> element may be obtained as

```
<video id='myVideo' height='180' width='300'>
  <source src='wildlife.mp4' />
</video>

<script>
```



```

aVideo = document.getElementById('myVideo');
//...
</script>

```

We can now use `play()/pause()` methods to play/pause the vide. A complete program is shown here:

```

<!DOCTYPE html>
<html>
<body>
  <video id='myVideo' height='180' width='300' >
    <source src='wildlife.mp4' />
  </video><br>
  <button id='playpause' onclick="playPause()">&#x25BA;</button>

  <script>
    aVideo = document.getElementById('myVideo');
    btn = document.getElementById('playpause');
    function playPause() {
      if (aVideo.paused) {
        btn.innerHTML='&#10074;&#10074;';
        aVideo.play();
      }else {
        btn.innerHTML='&#x25BA;';
        aVideo.pause();
      }
    }
  </script>
</body>
</html>

```

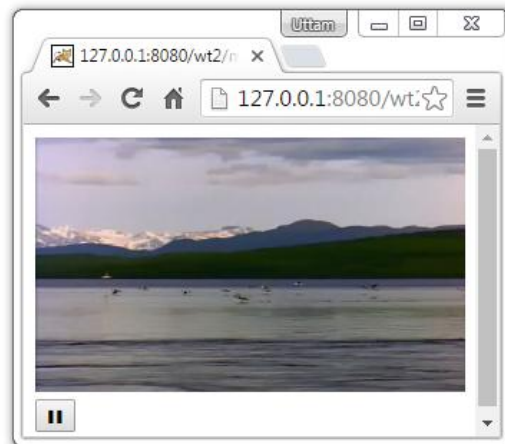
This has one video element and one button. When the button is clicked, `playPause()` gets executed. The function simple toggles the state of the video from play to pause and vice versa. The `pause` property indicates if the video is paused or not. The caption of the button is also toggled from ► to ■■ and vice versa. The former symbol corresponds to HTML entity `&#x25BA;` and later corresponds to entity pair `&#10074;&#10074;`. Now we are able to play/pause video using our own custom button and code. A snapshot when the video is playing is shown here.

However, the caption of the button remains as ■■ even after the video completes its playback. However, we can just attach a handler to `ended` event and make the caption as ►.

```

aVideo.onended = function() {
  btn.innerHTML='&#x25BA;';
}

```



A list of other events on `<video>` element is given in Table 23.6:

Table 23.6: Common `<audio>/<video>` properties, methods and events

Name	Description
------	-------------

Methods	
canPlayType()	Returns "" or "maybe" or "probably" based on the browser can play the specified audio/video type
load()	Causes the element to reset and start selecting and loading media
play()	Starts playing the media and sets paused attribute to false
pause()	Pauses the currently playing media and sets the pause attribute to true
addTextTrack()	Adds a new text track to the media
Properties	
audioTracks	It is an AudioTrackList object representing available audio tracks
autoplay	A Boolean property that indicates if the media should start playing as soon as it is loaded
buffered	It is a TimeRanges object representing the buffered parts of the media
controller	A MediaController object representing the current media controller of the media
controls	A Boolean property that indicates if the media should display control bar
crossOrigin	Its purpose is to allow images from third-party sites that allow cross-origin access to be used with canvas.
currentSrc	Holds the URL of the current media
currentTime	Represents the current playback position in seconds of the media
defaultMuted	Indicates if the media should be muted by default
defaultPlaybackRate	Holds the default speed of the media playback
duration	Returns the duration in seconds of the current media
ended	Indicates if the playback of the media has ended or not
error	A MediaError object representing the error state of the media
loop	Indicates if the media should start over again when completed
mediaGroup	Used to link multiple media elements together
muted	Indicates if the media is muted or not
networkState	Holds the current network state of the media
paused	Indicates if the media is paused or not
playbackRate	Holds the current speed of the media playback
played	A TimeRanges object representing the parts of the media that has already been played
preload	Indicates if the media should be loaded when the page loads
readyState	Holds the current ready state of the audio/video
seekable	A TimeRanges object representing the seekable parts of the media
seeking	Indicates if the user is currently seeking in the media
src	Holds the current source of the media element
textTracks	A TextTrackList object representing the available text tracks
videoTracks	A VideoTrackList object representing the available video tracks
volume	Holds the current volume of the media
Events	
abort	Occurs when the loading of a media is aborted
canplay	Occurs when the browser identifies that can play the media
canplaythrough	Occurs when the browser determines that it can play through the audio/video

	without stopping for buffering
durationchange	Occurs when the duration of the media is changed
emptied	Occurs when the current playlist is empty
ended	Occurs when the current playlist is completed
error	Occurs when an error occurred during the loading of an media
loadeddata	Occurs when the browser has loaded the current frame of the media
loadedmetadata	Occurs when the browser has loaded meta data for the media
loadstart	Occurs when the browser starts looking for the media
pause	Occurs when the media has been paused
play	Occurs when the media has been started or is no longer paused
playing	Occurs when the media is restarted playing after a pause or stop due to buffering
progress	Occurs when the browser is downloading the media
ratechange	Occurs when the playing speed of the media is changed
seeked	Occurs when the user is finished moving/skipping to a new position in the media
seeking	Occurs when the user starts moving/skipping to a new position in the media
stalled	Occurs when the browser is trying to get media data, but data is not available
suspend	Occurs when the browser is intentionally not getting media data
timeupdate	Occurs when the current playback position has changed
volumechange	Occurs when the volume has been changed
waiting	Occurs when the video stops because it needs to buffer the next frame

### 23.6.2 HTML Audio

A companion to a `<video>` element is an `<audio>` element that has many similar features. In section we shall outline `<audio>` elements noting key differences. An audio media is embedded in a page using `<audio>` element.

```
<audio src="SleepAway.mp3" controls>
  Your browser does not support the HTML5 audio element.
</audio>
```

The `<audio>` element exactly similar except that it has no `height` and `width` attribute. The list of attributes that we can use can be found in Table 23.5: except `height` and `width` attribute.

Programming `<audio>` element is also similar. In fact, the API for both audio and video descend from the same media API, so they are nearly the same. The only difference in these elements is that the video element has `height` and `width` attributes and a `poster` attribute. The events, the methods, and all other attributes are the same (see Table 23.6:)

## 23.7 JavaScript APIs

HTML5 also provides some beautiful JavaScript APIs that can be used to build powerful web applications. Following sections deal some of the important JavaScript APIs

### 23.7.1 Geolocation

Geolocation API, as the name indicates, allows a web page to find the geographic location (latitude and longitude) of the user navigating the page if the user desires to provide so. The page

can then do useful things such as showing the location on the map, geo-tagging, guiding someone to reach to a destination, finding local business etc.

The API it doesn't care how the browser determines location. The underlying mechanism might be via GPS, A-GPS, GSM/CDMA, WiFi, RFID, Bluetooth, IP-based or user input. Therefore, lookup may take some time and its accuracy depends on the underlying technology. It is more accurate for GPS-enabled devices.

The API is provided through the `navigator.geolocation` object. Most modern browsers support Geolocation API. However, we can check browser's support as follows:

```
if (navigator.geolocation) {  
    // Geolocation API is supported  
} else {  
    // Geolocation API is not supported  
}
```

Three methods are provided to work with Geolocation: `getCurrentPosition()`, `watchPosition()` and `clearWatch()`. We shall discuss them one by one.

The `getCurrentPosition()` method allows to find 'one-shot' location of the user. Since, finding location may take some time, the API is asynchronous and accepts callback method(s) to it. It looks like this:

```
void getCurrentPosition(PositionCallback successCallback,  
                        optional PositionErrorCallback errorCallback,  
                        optional PositionOptions options);
```

When invoked, this method returns the control immediately. It continues asynchronously and queries the positioning hardware to get up-to-date location information. If everything is fine and the location is obtained, it invokes the `successCallback` method with a `Position` object containing Geolocation information. The typical code to find one shot location looks like this:

```
function success(position) {  
    // do something with position object.  
}  
// One-shot position request.  
navigator.geolocation.getCurrentPosition(success);
```

The callback method should take first parameter as `Position` object, which has some useful properties which are described in Table 23.7: If the implementation cannot provide the value of some attribute, it is set to null.

Table 23.7: Properties of Position object

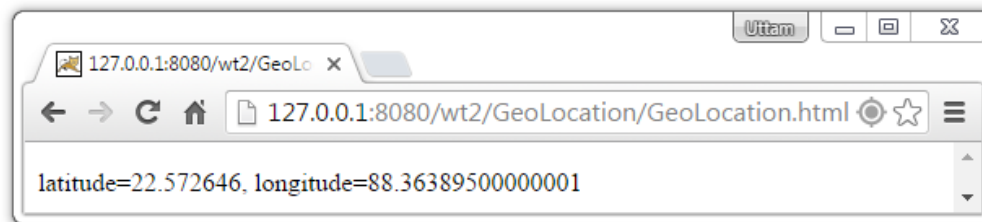
Event	Description
coords	An object that contains latitude and longitude
timestamp	The time when the Position object was acquired
coords.latitude	A real number from -90 to +90 that represents the latitude
coords.longitude	A real number from -180 to +180 that represents the longitude
coords.accuracy	Denotes the accuracy level in meters of the latitude and longitude coordinates. It is a non-negative real number.
coords.altitude	Denotes the height of the position, specified in meters above the WGS84 ellipsoid
coords.altitudeAccuracy	Denotes the accuracy level in meters of the altitude. It is a non-negative real number.

coords.heading	Denotes the direction of travel of the hosting device and is specified in degrees between 0° to 360°
coords.speed	Denotes the magnitude of the horizontal component of the hosting device's current velocity and is specified in meters per second.

The following example displays only latitude and longitude of the user:

```
<!DOCTYPE html>
<html>
  <body>
    <p id="loc"></p>
    <script>
      function success(position) {
        var p = document.getElementById("loc");
        p.innerHTML = 'latitude='+position.coords.latitude+',
longitude='+position.coords.longitude;
      }
      if (navigator.geolocation)
        navigator.geolocation.getCurrentPosition(success);
      else alert('Geolocation is not supported by your browser.');
```

A sample output is shown below:



It is good to print the values of all properties using for-in construct as follows:

```
for(i in position) document.writeln(i+' : '+position[i]+'<br>');
for(i in position.coords) document.writeln(i+' : '+position.coords[i]+'<br>');
```

It generated following output:

```
coords: [object Coordinates]
timestamp: 1464851163862
latitude: 22.561127
longitude: 88.4139124
altitude: null
accuracy: 76
altitudeAccuracy: null
heading: null
speed: null
```

We can access the map service of Google to show the location in a map.

```
<!DOCTYPE html>
<html>
  <body>
    <p id="map"></p>
    <script>
      function success(position) {
```

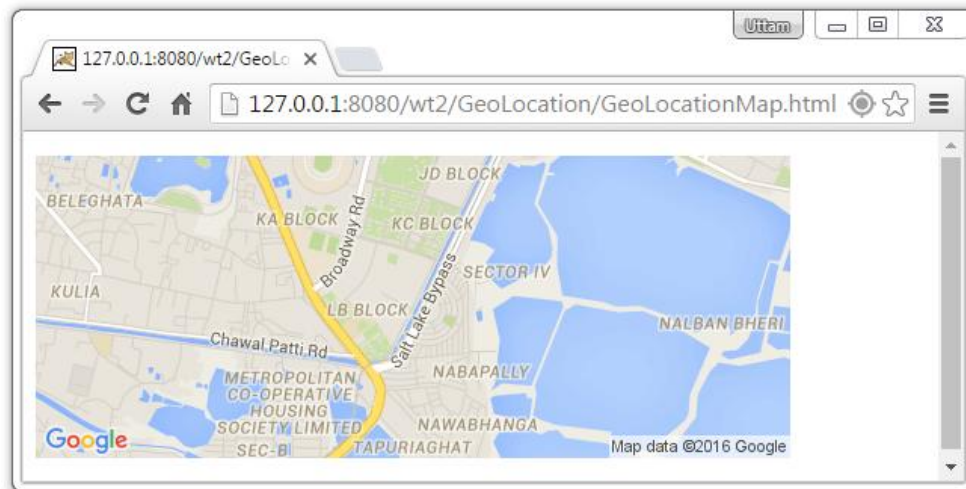


```

        url =
"http://maps.googleapis.com/maps/api/staticmap?zoom=14&size=500x300&sensor=false&center="+position.coords.latitude+","+position.coords.longitude;
        document.getElementById("map").innerHTML = "<img src='"+url+"'>";
    }
    if (navigator.geolocation)
navigator.geolocation.getCurrentPosition(success);
    else alert('Geolocation is not supported by your browser.');
```

</script>  
</body>  
</html>

This shows the location of the user in the center of the map. A sample result is shown below:



We can optionally pass a second callback function to `getCurrentPosition()` method as follows:

```

function success(position) {
    // do something with position object.
}
function failure(error) {
    // do something with error object.
}
// One-shot position request.
navigator.geolocation.getCurrentPosition(success, failure);
```

If there is anything wrong fetching location, a `PositionError` object created, populated with detail error information and `failure()` is invoked passing this object. This object has two properties `code` and `message` that respectively contains a numeric code and a description of the error. The possible codes are represented by constants `PERMISSION_DENIED`, `POSITION_UNAVAILABLE` and `TIMEOUT` in the error object. Following shows how to implement the error callback function:

```

function failure(error) {
    switch(error.code) {
        case error.PERMISSION_DENIED:
            alert('Denied to access Geolocation.');
```

break;

```
        case error.POSITION_UNAVAILABLE:
            alert('Location information is unavailable.');
```

break;

```
        case error.TIMEOUT:
            alert('Timeout occurs during accessing Geolocation');
```

break;

```

    }
}

```

The `getCurrentPosition()` method may take a third parameter `PositionOptions` object that customizes the way location information is obtained. It has following properties:

`enableHighAccuracy`—It is a Boolean type property used to indicate if the application wants accurate (true) location estimate or not (false). Default this is false.

`timeout`— Specifies the maximum duration in milliseconds the callback method must be invoked..

`maximumAge`— Indicates the longest age in milliseconds of cached position the application can accept.

Here is one example that shows how to use this `PositionOptions` parameter.

```

var options = { enableHighAccuracy: true, timeout: 1000, maximumAge : 0};
navigator.geolocation.watchPosition(success, failure, options);

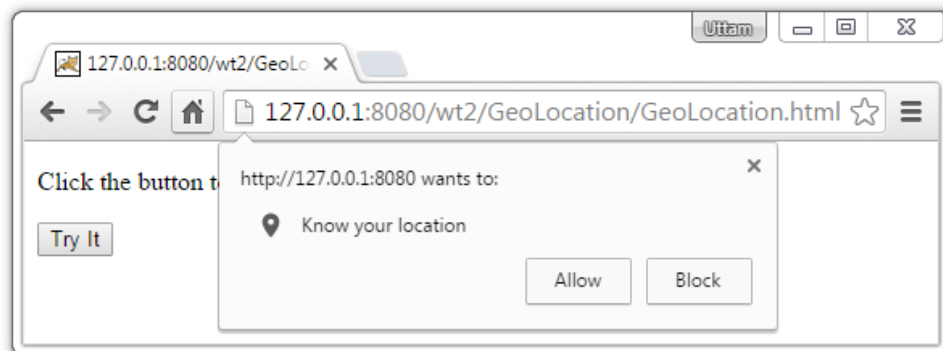
```

A value 0 of `maximumAge` indicates that no cached value is accepted; a fresh value must be obtained.

Unlike `getCurrentPosition()` that gets the location only once, `watchPosition()` gets the location repeatedly. It invokes the callback method only when position data changes due to device movement or if more accurate geo information arrives. So it can be used to trace the position of a moving objects very easily. The `watchPosition()` method takes similar parameters as `getCurrentPosition()` and is used exactly in the same way.

Note that HTML5 Geolocation specification does not specify how frequently `watchPosition()` should get the location. It depends upon the implementation. If it is required to get location after certain interval, `getCurrentPosition()` may be called periodically instead.

To protect user's privacy, the Geolocation API is not allowed to find user's location without user's permission. A browser often get such permission using a dialog as follows:



The user may also set the permission level in browser's privacy setting tab.

### 23.7.2 HTML Drag/Drop

**Drag and Drop (DnD)** means pressing and holding down the mouse button over an element, dragging it to another location, and releasing the mouse button to drop the element there. It may be used for a wide range of situations such as for moving, copying, reordering, deletion of elements with the help of mouse. Before HTML5, developers had to write complicated JavaScript code to

achieve DnD functionality. In HTML5, it is just a matter of fun. HTML5 provides a simple event-based JavaScript API and a few markup extensions that can be used to get DnD facility very easily.

The idea behind the DnD is very simple. An element, called draggable element,

Two elements are involved in DnD. An element where the dragging starts from, called *draggable* element and an element to catch it, called *drop target/point*. Typical source elements are images, lists, links, file object, block of HTML etc. The targets are those that can accept data such as `<div>`, `<span>`, `<p>` and many more. However, some elements such as images cannot be targets. The thing that we want to drop is called *payload*. The payload is typically the element the drag starts from. However, it may be anything else.

An element is made *draggable* by setting its `draggable` attribute to `true` as follows:

```

```

The entire DnD operation is described by series of events. Some are fired on draggable element and some are fired on drop target. A summary of these events are described in Table 23.8: The first three are fired on source element

Table 23.8: DnD event handlers

Event	Handler	Description
dragstart	ondragstart	Fires on source when the dragging is started
drag	ondrag	Fires on source repeated when the drag continues
dragend	ondragend	Fires on source element when mouse button is released at the end of dragging
dragenter	ondragenter	Fires when the mouse pointer enters the area of target element while dragging
dragover	ondragover	Fires repeatedly then dragging continues over the target element
dragleave	ondragleave	Fires when the mouse pointer leaves the area of target element while dragging
drop	ondrop	Fires on target element when mouse button is released at the end of dragging

All of these event handlers take a `DragEvent` (an extension to `MouseEvent`) object that has an useful property `dataTransfer`, which is a `DataTransfer` object. This `DataTransfer` object is the heart of the DnD API and holds information such as the type of drag (copy, move etc.), the drag's payload and its MIME type. It also provides methods to append items to the payload or remove items.

An application that uses DnD, first defines the handler for `dragstart` event on source element.

```
<script>
  function dragStart(de) {
    de.dataTransfer.setData("text/plain", de.target.id);
  }
</script>

```

The `dragStart()` method fires when the dragging `<img>` element is started. Its parameter `de` is a `DragEvent` object. The event handler for `dragstart` implements what is to happen when the user starts dragging the element. Typically, some information is stored in the `DataTransfer` object using its `setData()` method. This information is retrieved and used by `drop` event handler to do something; e.g. moving an element from one position to another. Note that this information need not be the data actually being transferred; it is something that can help us finding data to be transferred. For example, here we have just stored the id of the source `<img>` element in the

`de.dataTransfer` object. Note that `de.target` is the object on which the event fired and `de.target.id` is the value of its `id` attribute.

Any number of items may be included using `setData()` method. However, they must be string of some types such as `text/plain`, `text/html`, `text/uri-list` etc. These types are essentially labels of the items and help other handlers to find specific items by these labels.

To catch a drop we must define target and implement a set of handlers.

```
</script>
function dragEnter(de) {
    de.preventDefault();
}
function dragOver(de) {
    de.preventDefault();
}
function drop(de) {
    de.preventDefault();
    var id = de.dataTransfer.getData("text/plain");
    de.target.appendChild(document.getElementById(id));
}
</script>
<div id="target" ondrop="drop(event)" ondragenter="dragEnter(event)"
ondragover="dragOver(event)" ></div>
```

To handle drag and drop events, a browser may have its own mechanism that may prevent object from bubbling further and we may not get the behavior we want. Therefore, all event handlers first suppress browser's default behavior by calling `preventDefault()` method. This tells the browser to use the code we have written here. The drop handler then gets the id the source element that was set in `dragstart` handler and appends the element having that id to the target `<div>` element. The complete source code is given below:

```
<!DOCTYPE HTML>
<html>
<head>
<style>
    #target {width:100px;height:70px;padding:10px;border:1px solid gray;}
</style>
<script>
    function dragEnter(de) {
        de.preventDefault();
    }
    function dragOver(de) {
        de.preventDefault();
    }
    function dragStart(de) {
        de.dataTransfer.setData("text/plain", de.target.id);
    }
    function drop(de) {
        de.preventDefault();
        var id = de.dataTransfer.getData("text/plain");
        de.target.appendChild(document.getElementById(id));
    }
</script>
</head>
<body>
    <div id="target" ondrop="drop(event)" ondragenter="dragEnter(event)"
ondragover="dragOver(event)" ></div>
    
</body>
</html>
```

A sample out is shown in Figure 23.2:

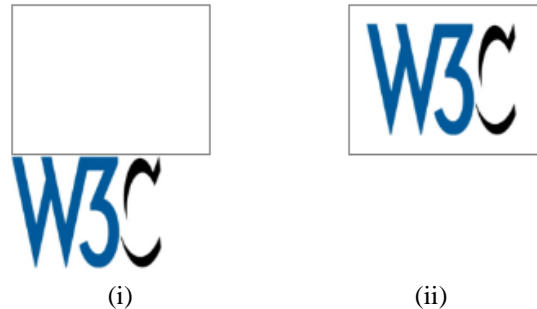


Figure 23.2: Drag and drop demonstration (i) before (ii) after

For clarity, a border is added to the target `<div>` element. We can also implement `dragleave` handler if desired. In the `dragenter`, `dragover` and `dragleave` handler, instead of just suppressing browser's default drag and drop behavior, we can do a number of interesting things. For example, we can revert the boundary color of target when a draggable object enters/leaves or continuously shows some text when an element dragged over the target etc.

Following example shows how to drag an image back and forth:

```
<!DOCTYPE HTML>
<html>
  <head>
    <style>
      #target1, #target2
      {float:left;width:100px;height:70px;padding:10px;border:1px solid gray;
margin:10px;}
    </style>
    <script>
      function dragOver(de) {
        de.preventDefault();
      }
      function dragStart(de) {
        de.dataTransfer.setData("text", de.target.id);
      }
      function drop(de) {
        de.preventDefault();
        var id = de.dataTransfer.getData("text");
        de.target.appendChild(document.getElementById(id));
      }
    </script>
  </head>
  <body>
    <div id="target1" ondrop="drop(event)" ondragover="dragOver(event)">
      
    </div>
    <div id="target2" ondrop="drop(event)" ondragover="dragOver(event)"></div>
  </body>
</html>
```

This example is very much similar to the previous one except that instead of one `<div>` element, two `<div>` targets are defined.

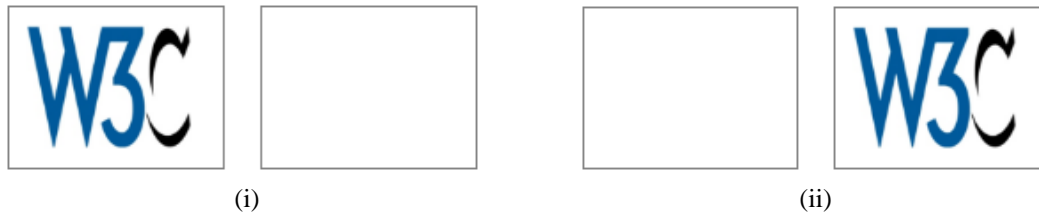


Figure 23.3: Drag back and forth (i) before (ii) after

When an element is dragged, the browser by default, shows a semi-transparent image of it that appears beside the mouse pointer. However, we can specify a custom image to shown at the specified location using `setDragImage()` method of `DataTransfer` object as follows:

```
var img = new Image();
img.src = 'sample.gif';
de.dataTransfer.setDragImage(img, 10, 10);
```

The `setDragImage()` method takes an `Image` object and offsets of the mouse pointer relative to the top-left corner of the image.

To provide better visual hint when an element is dragged the attributes `effectAllowed` and `dropEffect` of `DataTransfer` object may be used. The former is used to ask the browser to use specific mouse cursor to indicate the kind of action is performed when the dragged element is over a drop target. Possible values are `none`, `copy`, `copyLink`, `copyMove`, `link`, `linkMove`, `move`, `all` and `uninitialized`. The later controls the drag-and-drop feedback that the user is given during a drag-and-drop operation. Possible values are: `none`, `copy`, `link`, and `move`. Here is an example:

```
de.dataTransfer.effectAllowed = "copy";
```



### 23.7.3 File API

HTML5 File API provides a standard way to work with local files. We can load files and compress, encode, encrypt, or even upload them in smaller chunks using JavaScript. Albeit it can be a security hole, but the API itself can not do anything unless a user allows to do so.

The specification defines following primary interfaces for accessing files from a local file system:

**File** – This kind of object represents a single file. A file has useful read-only properties such as `name`, `size`, `type`, `lastModifiedDate` etc.

**FileList** – It is an array of `File` objects.

**FileReader** – Provides methods to read a `File` or a `Blob`, and an event model to obtain the results of these reads.

**Blob** - Represents a **B**inary **L**arge **O**bject (BLOB) to hold immutable raw binary data of a file. It also allows access to ranges of bytes within the `Blob` object as a separate `Blob`.

#### 23.7.3.1 Checking File API support

Before working with File API, we should test if the browser fully supports it.

```
if (window.File && window.FileReader && window.FileList && window.Blob) {
```

```

    // All the File APIs are supported, Work with File API
  }
  else {
    alert('Your browser does not fully support the File APIs');
  }
}

```

### 23.7.3.2 Selecting Files

To access a file local file system, user has to select the file to allow access to. There two ways user can select file(s); using `<input>` tag or using Drag and Drop (DnD). Following tag allows the user to select one or more files.

```
<input type="file" multiple />
```

We then attach a handler for change event. There are many ways we can do this. The simplest way, we can add `onchange` attribute specifying the handler:

```
<input type="file" onchange='readFiles(event)' multiple />
```

Alternatively, we can use following code:

```

<input type="file" id='files' multiple />
<script>
  document.getElementById('files').onchange = readFiles;
</script>

```

The file input has a browse button which when clicked a file dialog appears where user can select file(s). Then the handler for change event is invoked passing an event object containing selected files. Here is typical handler function:

```

function readFiles(evt) {
  // files is a FileList object which is an array of File objects.
  var files = evt.target.files;
  for (var i = 0; i < files.length; i++) {
    afile = files[i];      //aFile is a File Object
    //process aFile
  }
}

```

The `evt.target.files` is a `FileList` object that contains a list of the selected files as `File` objects. We can then iterate the file list, get individual file and process it. Here is a complete page that shows some attributes of selected file(s):

```

<!DOCTYPE html>
<html>
<body>
  <input type="file" onchange='readFiles(event)' multiple />
  <output id="info"></output>
  <script>
    // Check for File API support.
    if (window.File && window.FileReader && window.FileList && window.Blob) {
      function readFiles(evt) {
        var files = evt.target.files; // files is a FileList of File objects.
        str='<br>';
        for (var i = 0; i < files.length; i++) {
          str += 'name = '+files[i].name;
          str += ', size = '+files[i].size;
          str += ', type = '+files[i].type+'<br>';
        }
        document.getElementById('info').innerHTML = str;
      }
    } else alert('Your browser does not fully support the File APIs');
  </script>

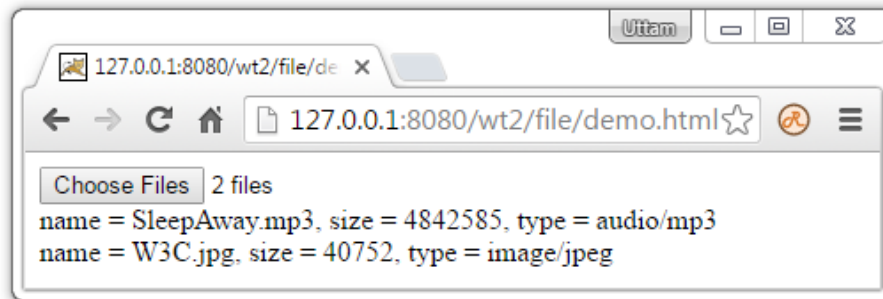
```

```

    </body>
</html>

```

When the user selects one or more files, it displays their name, size and type. A snapshot of the page is shown here:



The list of selected files can also be accessed via file input field's files property. Here is an example of that:

```

var inputTag = document.getElementById('files');
var files = inputTag.files;
for(var i=0; i<files.length; i++) {
    var file = files[i];
    //process file
}

```

### 23.7.3.3 Using Drag and Drop

We can use HTML5 Drag and Drop API to allow users to drag files from explorer and drop them on a zone. The dropped files can be detected using JavaScript. Here is a complete example:

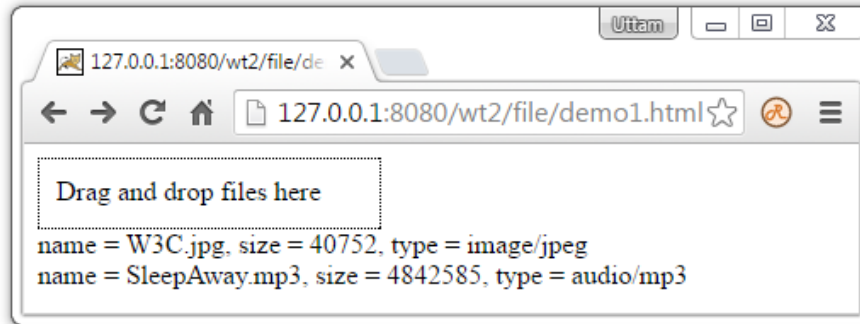
```

<!DOCTYPE html>
<html>
<body>
    <style>
        #file_area {width:180px;height:20px;padding:10px;border:1px dotted
black;}
    </style>
    <div id="file_area">Drag and drop files here</div>
    <output id="info"></output>
    <script>
        var filearea = document.getElementById('file_area');
        filearea.ondragover = function(evt) {
            evt.preventDefault();
            evt.dataTransfer.dropEffect = 'copy';
        }
        filearea.ondrop= function (evt) {
            evt.preventDefault();
            var files = evt.dataTransfer.files;
            str='';
            for (var i = 0; i <files.length; i++) {
                str += 'name = '+files[i].name;
                str += ', size = '+files[i].size;
                str += ', type = '+files[i].type+'<br>';
            }
            document.getElementById('info').innerHTML = str;
        }
    </script>
</body>
</html>

```



It defines a drop zone as a `<div>` element. When files are dropped here, drop event occurs and handler gets an event object. The list of files dropped onto `<div>` element can be found in the `event.dataTransfer.files` object. A sample snapshot of the program is shown here.



#### 23.7.3.4 Reading Files

The `FileReader` provides a way to read `File` content asynchronously through JavaScript event handling mechanism. The `FileReader` object contains a number of methods to read file content:

`readAsBinaryString()` - Takes a `File` or a `Blob` object and an optional encoding. It returns file/blob's data as a binary string. By default the string is encoded as 'UTF-8'. The encoding parameter may be passed to use a different encoding

`readAsDataURL()` - Takes a `File` or a `Blob` object. It returns file/blob's data encoded as a data URL.

`readAsArrayBuffer()` - Takes a `File` or a `Blob` object. It returns file/blob's data as an `ArrayBuffer` object.

To read file content, we must first create a `FileReader` object as follows:

```
var reader = new FileReader();
```

We then invoke one of the above methods passing a `File` or `Blob` object. However, result is returned in the load event handler. So, we add a handler as follows:

```
reader.onload = function(evt) {
    fileData = evt.target.result;
    //process file data
}
```

Here is complete example that shows how to read file content as data URL and use it in the `src` attribute of `<img>` tag.

```
<!DOCTYPE html>
<html>
<body>
    <input type="file" id="files" name="files[]" /><br>
    <output id="info"></output>
    <script>
        if (window.File && window.FileReader && window.FileList && window.Blob) {
            document.getElementById('files').onchange = function(evt) {
                files = evt.target.files;
                for (i = 0; i < files.length; i++) {
                    var reader = new FileReader();
                    reader.onload = function(evt) {
```

```

        document.getElementById('info').innerHTML = "<img width='200'
height='100' src='"+evt.target.result+"'>";
    }
    reader.readAsDataURL(files[i]);
}
} else alert('Your browser does not fully support the File APIs');
</script>
</body>
</html>

```

A snapshot of the above page is shown here.

### 23.7.3.5 Progress bar

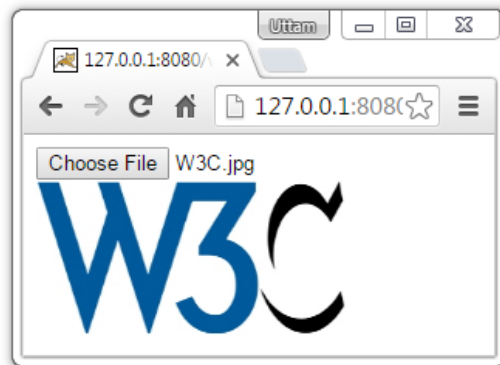
It is possible to monitor the progress of file read, catch errors, and determine when a load is complete. A progress bar will be good enough to show the progress. Following page demonstrates this:

```

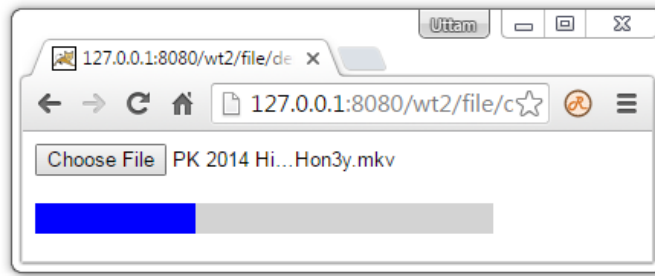
<!DOCTYPE html>
<html>
<body>
<style>
    #outer {
        width: 300px;
        height: 20px;
        background-color: lightgray;
    }
    #inner {
        width: 0%;
        height: 20px;
        background-color: blue;
    }
</style>
<input type="file" id="files" name="files[]" /><br><br>
<div id="outer" >
    <div id="inner" ></div>
</div>
<script>
    if (window.File && window.FileReader && window.FileList && window.Blob) {
        document.getElementById('files').onchange = function(evt) {
            files = evt.target.files;
            for (i = 0; i < files.length; i++) {
                var reader = new FileReader();
                reader.onprogress = function(evt) {
                    var percentLoaded = Math.round((evt.loaded / evt.total) * 100);

                    document.getElementById('inner').setAttribute("style","width:"+percentLoaded+"%");
                }
                reader.readAsDataURL(files[i]);
            }
        }
    } else alert('Your browser does not fully support the File APIs');
</script>
</body>
</html>

```



A snapshot in the middle of the file reading is shown here.



### 23.7.4 Web Storage

It is one of the local (to client) storage facilities. In short, it's a persistent storage within the web browser where web pages can store key-value pairs, retrieve/use them later. This immediately reminds us Cookies that are used for similar purposes. In fact, it is similar to cookies; data persist even after the browser window is closed. However, web storage has some distinct advantages over cookies:

- The value is never sent to the server by the browser unless we explicitly copy them out of the storage and append them to a request.; thereby hiding data from intruders and increasing speed of the web application.
- Cookies can store maximum of 4KB data whereas web storage can store much more data (typically 5MB to 10 MB depending on the browser).
- Security-conscious people and companies may turn Cookies off. However, web storage remains opened all the time.
- Events are fired when data are changed/set. Thus necessary action may be taken by adding event listeners.
- The data stored in the storage can only be read by pages from the same domain that stored the data.

Most current browsers support HTML5 web storage.

There are two types of storage available: *local storage* and *session storage*. Data stored in local storage are never expired and removed unless they are deleted explicitly. The data will be available even the browser/tab is closed or reopened. However, data stored in session storage by a page in a window/tab are only available as long as the window/tab remain opened. So, as soon as a browser window/tab is closed, the session storage associated with it is removed.

The local and session storages are provided via two `Storage` objects: `window.localStorage` and `window.sessionStorage`. These are used in the same way; only their life span and visibility is different. They provide methods to allow us to add, modify or delete stored data items. A summary of properties and methods available on storage objects is provided in Table 23.9:

Table 23.9: Web storage object properties and methods

Property	
Name	Description
length	An integer that represents the number of data items stored in the Storage object
Methods	
Name	Description

setItem()	Takes a key and a value and stores them in the storage.
getItem()	Returns the value corresponding to the specified key.
removeItem()	Deletes an item associated with the specified key
key()	Accepts an integer n and returns nth key from the storage.
clear()	Makes the storage empty deleting all items from the storage

We can check if a browser supports web storage as follows:

```
if (typeof(Storage) !== "undefined") {
    // use localStorage/sessionStorage.
} else {
    // Browser does not support web storage
}
```

Working with web storage is extremely easy. Since, `localStorage` and `sessionStorage` are objects; we can store values by setting its properties. Following stores a string value in the local and session storage respectively:

```
localStorage.name = "Tom";
sessionStorage.name = "Jerry";
```

Retrieving them is also pretty straightforward:

```
name1= localStorage.name;           //name1 = "Tom"
name2 = sessionStorage.name;       //name2 = "Jerry"
```

JavaScript array notation may also be used as follows:

```
//Store values
localStorage["name"] = "Tom";
sessionStorage["name"] = "Jerry";

//Retrieve them
name1= localStorage["name"];       //name1 = "Tom"
name2 = sessionStorage["name"];   //name2 = "Jerry"
```

This notation is useful is a property name contains characters which are not allowed in JavaScript variable names or property itself a variable. The `setItem()` and `getItem()` may also be used (probably more obvious) to store and retrieve values as follows:

```
//Store values
localStorage.setItem("name", "Tom");
sessionStorage.setItem("name", "Jerry");

//Retrieve them
name1= localStorage.getItem("name"); //name1 = "Tom"
name2 = sessionStorage.getItem("name"); //name2 = "Jerry"
```

The number of elements stored in storage may be obtained using its `length` property;

```
n1 = localStorage.length;
n2 = sessionStorage.length;
```

We can iterate the keys and get corresponding values as follows:

```
for(var n=0; n < localStorage.length; n++){
    var key = localStorage.key(n);
    document.writeln(key+' ' +localStorage.getItem(key)+'<br>');
}
```

This prints all the key/value pairs stored in local storage.

Note that web storage can only store strings. So, if we try to store an object, it will not be stored in right way.

```
info = {loginTime:new Date(), from>window.navigator.appName};
localStorage["Tom"]=info;
info = localStorage["Tom"]; //info = "[object Object]"
for(i in info) document.write(i+' '+info[i]+'<br>');
```

This shows that data are stored as "[object Object]" not a real object. However, can use `JSON.stringify()` and `JSON.parse()` methods to store and retrieve objects properly as follows:

```
info = {loginTime:new Date(), from>window.navigator.appName};
localStorage["Tom"]=JSON.stringify(info);
info = JSON.parse(localStorage["Tom"]);
for(i in info) document.write(i+' '+info[i]+'<br>');
```

We delete items from storage as follows:

```
delete sessionStorage.name;
delete localStorage.name;
```

Alternately, `removeItem()` method may also be used:

```
localStorage.removeItem ("name");
sessionStorage.removeItem ("name");
```

Entire storage is made empty using `clear()` method:

```
localStorage.clear();
sessionStorage.clear();
```

Following example shows how to keep track of number of times a page visited from a computer.

```
<!DOCTYPE HTML>
<html>
<body>
<script type="text/javascript">
  if (typeof(Storage) !== "undefined") {
    if( localStorage.count )
      localStorage.count = Number(localStorage.count) +1;
    else localStorage.count = 1;
    document.write('Visit count : ' + localStorage.count);
  } else {
    alert('Browser does not support web storage');
  }
</script>
</body>
</html>
```

Following how to keep track of time a user accessed a page last.

```
<!DOCTYPE HTML>
<html>
<body>
<script type="text/javascript">
  if (typeof(Storage) !== "undefined") {
    if( sessionStorage.count )
      document.write('Last access time : ' + sessionStorage.count);
    sessionStorage.count = new Date();
  }
  else {
```

```

        alert('Browser does not support web storage');
    }
</script>
</body>
</html>

```

We can also attach storage event handler that gets fired when data are changed/set/removed. We can take necessary actions inside the handler. Here is how we can do that:

```

function storageHandler(se){
    alert("storage event occurs");
}
window.addEventListener('storage', storageHandler, true);

```

The handler is passed a `StorageEvent` object that has a set of useful properties:

- `key` – name of the property that has changed/set
- `oldValue` – value of the property before change
- `newValue` – value of the property after change/set
- `url` – complete url of the page from where the event originated
- `storageArea` – `localStorage` or `sessionStorage` object where the change occurs.

Note that not all browsers support the storage event. Some supports but call the handler only when the storage is changed by a different window. So, to see the effect you may need to open two or more windows.

Although web storage provided a simple but elegant way to store information locally, it has some serious security holes. Since, data can always be stored in web storage without the user's permission; it is vulnerable to be exploited. So, HTML5 working group must immediate look into the matter so that web pages can't record any information without the user's knowledge. Like Geolocation, web storage should enforce web pages to seek permission before storing any information locally.

### 23.7.5 Indexeddb

One of the powerful and recent web standards is Indexed Database (or simply IndexedDB). It allows us to store huge data (maximum of 50% of free disk space) persistently in the browser (client-side) and an efficient indexed-based searching API to retrieve the data. Note that W3C has deprecated WebSQL, a technology similar to IndexedDB. Therefore web developer should no longer use WebSQL. In this section, an introduction to this technology is given.

Anyway, how is it different from Cookies or web storage or database? Cookies can store limited amount of data and they are sent back and forth to the server with every request. IndexedDB can store very large amount of data in the client machine; thus reducing the load on server and network. Although, web storage can store relatively more data, they can simply store key-value string pair; IndexedDB, on the other hand can store arbitrary data. However, IndexedDB is not that powerful as a full-fledge relational database. IndexedDB can store objects. Moreover, it does not provide any query language such as SQL to work with database. In fact, IndexedDB does not have many features a sophisticated relational database has.

The API is provided a child object of `window` object. Let us first check if a browser supports IndexedDB. Note that the W3C specification for IndexedDB is not yet finalized; it is still evolving. Some browsers have not yet implemented it, some have implemented it but uses prefix tag. Some browsers such as IE 10, Chrome 24, Firefox 16 have however removed prefix and provide full implementation as `indexedDB` object.

Implementations that use a prefix may be incomplete or may have bugs, or following an old version of the specification and should be avoided. It is not probably good to use a browser that claims to support IndexedDB but fails. So, we only check the existence of `indexedDB` property of `window` object.

```
if(window.indexedDB) {  
    // Work with IndexedDB  
}  
else {  
    // Browser does support a stable version of IndexedDB  
}
```

### 23.7.5.1 Opening a database

To open a database, we make a request to `indexedDB` object using its `open()` method.

```
request = indexedDB.open("myDB", 1);
```

The first parameter is the name (string) of database and second parameter is version (integer) of database. It opens the database having the specified name or creates one if no such database already exists. The second parameter is an integer, which allows us to update the schema of the database. A database created by a domain is visible by other domains. Different domains may create databases with same name without any conflict.

All API methods including `open()` are asynchronous and return immediately without blocking. To get results, event listeners are attached. Four kinds of events can occur: `success`, `error`, `upgradeneeded` and `blocked`. If `open()` fails to handler for `error` event gets called passing an `Event` object to it. Similarly, if everything succeeds, handler for `success` event gets called. The `upgradeneeded` event occurs when the database is created (i.e. `open()` is called for first time) or a higher version number is used. Higher version number may be used to upgrade (such as adding more object stores) the database. A version number less than the current version will throw an exception. Typically, when the page is loaded for the first time, `upgradeneeded` and `success` events will fire, otherwise `success` event will fire.

To know what really happened in `open()` method, we attach handlers to `success` and `error` events.

```
var db;  
request.onerror = function(event) {  
    //Something was wrong during open  
};  
request.onsuccess = function(event) {  
    //opening was successful, save it for other's use  
    db = event.target.result;      //db = request.result;  
};
```

If database is opened successfully, an `IDBDatabase` object that represents the database is created and is stored in `request.result` (or `event.target.result`). If we succeed, we save this object in the global `db` variable for other's use such as adding/retrieving data. Let us now define some simple but convenient data to store in the database:

```
const bookData = [  
    { isbn: '0198066228', title: 'Web Technologies', price: 565 },  
    { isbn: '0199455503', title: 'Advanced Java Programming', price: 695 }  
];
```

The array contains two objects each having three properties: `isbn`, `title` and `price`. Note that ISBN of a book is unique.

### 23.7.5.2 Object Store

An IndexedDB has *object stores* instead of tables. One IndexedDB can have any number of object stores. An object store serves as the primary unit of storage and can contain objects or other values depending on its settings. A value stored in an object store is associated with a unique key. We can supply the key explicitly or tell the object store how to find keys using key path or key generator.

The database that we have created is so far empty. Before adding any object/data, we create an object store first. We can only create object stores in `upgradeneeded` handler. Here is a sample code:

```
request.onupgradeneeded = function(event) {  
  db = event.target.result;  
  if (!db.objectStoreNames.contains('books'))  
    var bookStore = db.createObjectStore('books', { keyPath: 'isbn' });  
};
```

Here, `db` refers to the database opened by `open()` method. We then check if an object store having name 'books' already exists. If it does not exist, we create a new object store using the `createObjectStore()` method, passing the name ('books') of the object store. We also pass a JavaScript object containing some settings for the object store. The `keyPath` property of setting object indicates that the key associated with an object to be stored may be found in its `isbn` property. So, every object must have an `isbn` property. Indeed ISBN numbers are unique and may be used as keys. The significance of key path and key generator is summarized in **Error! Reference source not found.**

Table 23.10: Effect of key path and key generator.

autoIncrement	keyPath	Description
No	No	Object store can contain any type of value. We must supply a key when we store a value.
No	Yes	Object store can only contain JavaScript objects. The key is obtained from the object's property as indicated by <code>keyPath</code> . So, objects must have a property with the same name as the <code>keyPath</code> .
Yes	No	Object store can contain any type of value. The key is generated automatically. We can provide a key explicitly if we want..
Yes	Yes	Object store can only contain JavaScript objects. Key is generated and stored in the object in a property with the same name as the <code>keyPath</code> . However, if such a property already exists, it is used instead of generating a new key.

An object store can generate unique keys and use them if it is configured properly. The initial key number is set to 1 and newly auto-generated key is increased by 1. Here is an example that created an object store with key generation facility:

```
var bookStore = db.createObjectStore('books', { autoIncrement: true });
```

### 23.7.5.3 Storing data

Like relational databases, in IndexedDB, an operation (add/remove/update data) on database (more specifically object store) occurs in the form of a transaction represented by `IDBTransaction` object which is created using `transaction()` method.

```
var trans = db.transaction(['books'], 'readwrite');
```



The `transaction()` method takes two arguments. The first is argument determines the scope of the transaction; parts of the database can be affected through the transaction. The scope is specified as an array of names of object stores we want the transaction to span. Often it is a single object store. For our case it is only 'books' that one we created earlier.

The second argument is the type of transaction i.e. if any change to the database is going to happen or just reading from database is done. There are three available modes:

`readonly`—Only “read” operation on the objects that are in transaction’s scope is allowed.

`readwrite`—Both “read” and “write” operations on the objects that are in transaction’s scope are allowed.

`versionchange`— Allows “read” and “write” operations as well as creation and deletion of object stores and indexes.

So, to read the data the transaction can either be in `readonly` or `readwrite` mode and to make changes the transaction must be in `readwrite` mode. If nothing specified for the second argument, a read-only transaction is returned. Since, we want to add some data, we specify `readwrite` mode.

Now we specify the specific object store we are going to work with using `objectStore()` method on `IDBTransaction` object :

```
var bookStore = trans.objectStore('books');
```

The `bookStore` now refers to the object store that we created before and we can add data to it using `add()` method.

```
for(i in bookData) bookStore.add(bookData[i]);
```

The `add()` method takes an object to be stored. Note that no keys have been supplied as object store was configured with key path. The keys are taken from `isbn` property of the objects. The book objects really contain the `isbn` property. If `keyPath` is not specified during object store creation, we can explicitly specify keys as follows:

```
for(i in bookData) bookStore.add(bookData[i], i);
```

Here the keys are nothing by array indices starting from 0. We can figure out the status of the transaction by attaching event handlers and can take appropriate actions.

```
trans.oncomplete = function(event) {  
    //Transaction completed successfully  
};  
trans.onerror = function(event) {  
    //An error occurred during transaction  
};  
trans.onabort = function(event) {  
    //Transaction aborted  
};
```

Here is a complete example that creates a database and adds some data:

```
<html>  
<body>  
    <script>  
        const bookData = [  
            { isbn: '0198066228', title: 'Web Technologies', price: 565 },  
            { isbn: '0199455503', title: 'Advanced Java Programming', price: 695 }  
        ];  
        if(window.indexedDB) {  
            var db;  
            request = indexedDB.open("myDB", 1);
```

```

request.onupgradeneeded = function(event) {
    db = event.target.result;
    if (!db.objectStoreNames.contains('books'))
        var bookStore = db.createObjectStore('books', { keyPath: 'isbn' });
};
request.onerror = function(event) {
    alert("Some error occurred");
};
request.onsuccess = function(event) {
    db = event.target.result;
    var trans = db.transaction(['books'], 'readwrite');
    var bookStore = trans.objectStore('books');
    trans.oncomplete = function(event) {
        alert("Transaction completed successfully");
    };
    trans.onerror = function(event) {
        alert('Error occurred during transaction');
    };
    trans.onabort = function(event) {
        alert('Transaction aborted');
    };
    for(i in bookData) bookStore.add(bookData[i]);
};
} else alert('Browser does support a stable version of IndexedDB');
</script>
</body>
</html>

```

#### 23.7.5.4 Retrieving data

To read data from an object store, we must have a reference to it using the steps used when we added data:

```

var trans = db.transaction(['books'], 'readonly');
var bookStore = trans.objectStore('books');

```

We can then get the desired object using `get()` method specifying the key of the object.

```
req = bookStore.get('0199455503');
```

The actual object can be obtained and processed by attaching `success` event handler:

```

req.onsuccess = function(event) {
    obj = event.target.result;
    for(i in obj) document.writeln(i+': '+obj[i]+'<br>');
};

```

The entire code is typically written in `oncomplete` event handler of transaction object. To avoid verbosity, we can chain a series of calls like this:

```

db.transaction(['books'],
'readonly').objectStore('books').get('0199455503').onsuccess = function(event)
{
    obj = event.target.result;
    for(i in obj) document.writeln(i+': '+obj[i]+'<br>');
};

```

#### 23.7.5.5 Using cursor

To retrieve a value using `get()` method, we must know the corresponding key. Cursor lets us get all data by iterating through the object store. Filter may be used to get only desired data. Following fetch all the data from our book store using cursor:

```

var bookStore = db.transaction(['books'], 'readonly').objectStore('books');
bookStore.openCursor().onsuccess = function(event) {
    var cursor = event.target.result;
    if (cursor) {
        document.writeln('<br>' + cursor.key);
        for(i in cursor.value) document.writeln(i + ': ' + cursor.value[i] + ',');
        cursor.continue();
    }
    else alert("No more data!");
};

```

A request to get a cursor on book store, we use `openCursor()` method that like other works asynchronously and returns result in success event handler. The cursor is saved a variable `cursor`. Then the key and data are stored in `key` and `value` properties of the cursor object. The `continue()` method keeps the cursor moving. When it reaches at the end, it becomes null. Here is a complete example:

```

<html>
<body>
<script>
const bookData = [
{ isbn: '0198066228', title: 'Web Technologies', price: 565 },
{ isbn: '0199455503', title: 'Advanced Java Programming', price: 695}
];
if(window.indexedDB) {
    var db;
    var req = window.indexedDB.deleteDatabase("myDB");
    request = indexedDB.open("myDB", 1);
    request.onupgradeneeded = function(event) {
        db = event.target.result;
        if (!db.objectStoreNames.contains('books'))
            var bookStore = db.createObjectStore('books', { keyPath: 'isbn' });
    };
    request.onerror = function(event) {
        alert("Some error occurred");
    };
    request.onsuccess = function(event) {
        db = event.target.result;
        var trans = db.transaction(['books'], 'readwrite');
        var bookStore = trans.objectStore('books');
        trans.oncomplete = function(event) {
            var bookStore = db.transaction(['books'],
'readonly').objectStore('books');
            bookStore.openCursor().onsuccess = function(event) {
                var cursor = event.target.result;
                if (cursor) {
                    document.writeln('<br>' + cursor.key);
                    for(i in cursor.value) document.writeln(i + ':
'+cursor.value[i] + ',');
                    cursor.continue();
                }
            };
        };
        trans.onerror = function(event) {
            alert('Error occurred during transaction');
        };
        trans.onabort = function(event) {
            alert('Transaction aborted');
        };
        for(i in bookData) bookStore.add(bookData[i]);
    };
} else alert('Browser does support a stable version of IndexedDB');

```

```

</script>
</body>
</html>

```

The `openCursor()` method may take a key range of the values to be retrieved as first parameter and direction of the cursor movement as second parameter . Here is an example:

```

bookStore.openCursor(IDBKeyRange.bound('1', '10'),'next').onsuccess =
function(event) {
//...
};

```

The cursor moves in the backward direction during iteration and retrieves those data that have key value from 1 to 10.

### 23.7.5.6 Update and delete

Updating some data in the object store is done by first getting it, changing it and putting it back to the object store. It looks like this:

```

var bookStore = db.transaction(['books'], 'readwrite').objectStore('books');
bookStore.get('0199455503').onsuccess = function(event) {
    obj = event.target.result;
    obj.price = 799;
    bookStore.put(obj).onsuccess = function(event) {
        alert('Data updated successfully!');
    };
};

```

It changes the price of the book having issn number 0199455503 to 799.

Deleting an object is pretty easy. We use `delete()` method specifying the key of the object to be deleted. Here is an example:

```

var bookStore = db.transaction(['books'], 'readwrite').objectStore('books');
req = bookStore.delete('0199455503');
req.onsuccess = function(event) {
    //Deleted successfully
};

```

It deletes the book having the isbn number 0199455503 from the book store.

An object store is deleted (in upgradeneeded handler) using `deleteObjectStore()` method passing the name of the object store as follows:

```

db.deleteObjectStore('name');

```

All items from an object store is deleted using `clear()` method:

```

bookStore.clear();

```

An entire database is deleted using `deleteDatabase()` method passing the name of the database to be deleted.

```

var req = window.indexedDB.deleteDatabase("myDB");

```

### 23.7.6 Application Cache

The Application Cache (or simply AppCache) helps a web application developer to selectively cache in the web browser everything from pages to images to scripts to cascading style sheets and make available to users during offline. The application will work correctly with the help of cached resources. The execution of application using AppCache runs significantly faster since the files

come from local cache memory. Moreover, network load is reduced as few data are pulled down from the server.

We tell the browser which resources it must cache or must not cache for offline access in a text file called *cache manifest file*. The manifest file is specified in the `<html>` tag using its `manifest` attribute as follows:

```
<!DOCTYPE HTML >
<!--demo.html-->
<html manifest="sample.appcache">
<!-- ... -->
</html>
```

When this page is loaded for the first time, the `manifest` attribute instructs the browser to cache the page `demo.html` and cache or not cache other resources as specified in `sample.appcache` file. A page without `manifest` attribute is not cached unless manifest file used in some other page tell the browser to cache it.

The `manifest` attribute refer to an absolute or relative URL of the manifest file. If absolute URL is used, it must be belong to the same web application. A manifest file may have any file extension but `.appcache`. Some older browsers requires that mime-type of manifest file must be specified as `text/cache-manifest`. The procedure to specify this may be different in different web server. For example, in apache it specified in `.htaccess` configuration file as follows:

```
AddType text/cache-manifest .appcache
```

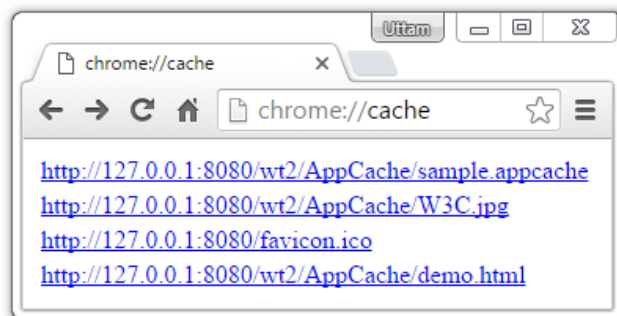
In Tomcat, it is specified in `conf/web.xml` file as follows:

```
<mime-mapping>
  <extension>appcache</extension>
  <mime-type>text/cache-manifest</mime-type>
</mime-mapping>
```

Here is sample manifest file (`sample.appcache`):

```
CACHE MANIFEST
CACHE:
W3C.jpg
```

This tells the browser to download and cache the image `w3c.jpg` when the page that specifies this manifest file. The cached pages may be viewed in chrome using URLs `chrome://cache/` or `chrome://appcache-internals/`. The later also allows removing cached items. A sample snapshot when used `chrome://cache/` is shown here:



This shows three files, `demo.html` (specifies the manifest file), `sample.appcache` (manifest file) and `w3c.jpg` (specified in manifest file) have been cached.

### 23.7.6.1 Format of manifest file

A manifest file must start with a line `CACHE MANIFEST` that indicates that file is a cache manifest file. In general, a manifest file may have three sections `CACHE`, `NETWORK` and `FALLBACK`:

**CACHE:** - Files listed in this section are cached after they are downloaded for the first time

**NETWORK:** - Files listed in this section are NEVER cached; the contents of this pages are probably generated dynamically (say a .jsp file) and hence should always be downloaded from the server.

**FALLBACK:** - Files listed in this section are fallback pages if a page is not accessible.

Here is sample manifest file:

```
CACHE MANIFEST
#9th June, 2016 v0.0
#It is a comment

CACHE:
#Files listed here are cached
W3C.jpg
mystyle.css

NETWORK:
#Files listed here are not cached
login.jsp

FALLBACK:
#serve error.html if main is not accessible
main.html error.html
```

Just after `CACHE MANIFEST`, we often write the date and version of manifest file with a comment. Lines started with ‘#’ are comment lines and are ignored. However, a comment line can serve another purpose. A browser re-cached the files listed in manifest file if content of manifest file changes. However, if contents of files listed in the manifest file change, those files should be re-cached. If we add/delete any resource to/from manifest file, it gets automatically changed and the browser will re-cache the resources. However, what if we just modify already cached resource(s)? In that case, comments come into play. This can easily be done by adding/deleting/modifying one or more comment lines (often date comment after `CACHE MANIFEST` line) .

In the `CACHE` section, we write list of files in separate lines we want the browser to cache. In the above example, we ask the browser to cache the files `W3C.jpg` and `mystyle.css`.

Each resource to be cached must be listed explicitly in a single line using relative or absolute path. Wild cards are not allowed. So, ‘/images/\*’ will be treated as single resource having URL ‘/images/\*’.

Files listed under `NETWORK` section require network connection. These are the files we want the browser not to cache. The pages that access data from databases or pages whose contents often change should be listed here. For example, JSP page `login.jsp` should not be cached. The \* indicates all resources except listed in `CACHE` section require the user to be online and should not be cached. Note that, asterisk is not a wildcard character; it is a special flag.

The `FALLBACK` section defines a page to be served if one or more pages fail to load due to network error. Above example says that `error.html` page will served if `main.html` fails to load. Note that `error.html` is also cached as it a fallback page. Following entry asks the browser to serve `error.html` if any page fails to load.

```
FALLBACK:  
/ error.html
```

Section names may appear any order and each section can even appear multiple times in a single manifest. The `NETWORK` and `FALLBACK` sections are optional. If no section name is present, anything after `CACHE MANIFEST` line will be considered under `CACHE` section. So,

```
CACHE MANIFEST  
CACHE:  
W3C.jpg
```

and

```
CACHE MANIFEST  
W3C.jpg
```

are same.

The `window.applicationCache` object represents the browser's app cache and can be manipulated programmatically. The cache goes through a series of events:

- checking — This event occurs when browser tries to download the manifest for the first time, or is testing if an newer version of the manifest file exists.
- noupdate — This event occurs it there is no updated version of the manifest file on the server, after checking.
- downloading—This event occurs if browser finds a new manifest file and fetches it, or downloads the resources listed by the cache manifest for the first time.
- progress — This event occurs for each file downloaded as part of the `AppCache`.
- cached — his event occurs when all the resources are downloaded and cached.
- updateready— This event occurs when browser completes re-downloading an updated cached file.
- obsolete — This is fired if the manifest file cannot be found (HTTP error code 404 or 410).
- error — This is fired if there is anything wrong such as manifest file is not found, or the manifest file is present, but any of the files mentioned in the manifest file can't be downloaded properly or manifest file changes during update or in any other case where there is a fatal error.

These events are represented by `window.applicationCache.status` with following values:

```
0 - UNCACHED  
1 - IDLE  
2 - CHECKING  
3 - DOWNLOADING  
4 - UPDATEREADY  
5 - OBSOLETE
```

We can add listeners to `updateready` event to test to see if an application has an updated cache manifest file. To programmatically testing for a new manifest file, we use `window.applicationCache.update()` method. This attempts to update the browser's cache (which requires the manifest file to have changed). Here is an example:

```
function onUpdateReady() {  
    //new version of manifest file detected  
    appCache.swapCache();  
}
```

```

window.applicationCache.addEventListener('updateready', onUpdateReady);
window.applicationCache.update();

```

We use `swapCache()` to make the browser to use this newly updated cache.

### 23.7.7 Web workers

Before HTML5, JavaScript supported only single-threaded programming i.e. multiple scripts can not run simultaneously. As a result some tasks such as handling UI events, responding to queries, computation of data, manipulating DOM which should occur concurrently for convenience could not be possible. Although, we can mimic parallelism using AJAX's XMLHttpRequest, event handlers, `setInterval()`, `setTimeout()` etc. they are really not multi-threaded but asynchronous.

Fortunately, HTML5 introduced a powerful concept called *Web worker* that can run scripts concurrently with the main thread without interfering the main thread. Workers run in a context other than the main thread's context and typically execute a computationally intensive task. Since an isolated (runs without interacting with others and terminate) worker is not that useful, API also specifies a messaging framework where main thread and worker can exchange messages.

Two kinds of workers are very common: *dedicated worker* and *shared worker*. A dedicated worker is only accessible from the script that spawned it. A shared worker can be accessed from multiple scripts. This section first discusses how to work with dedicated workers and then shared workers.

#### 23.7.7.1 An example

We shall consider a very simple but elegant example to understand how to work with Web Workers. In this example, an HTML page sends an integer number (5 say) to the worker, the worker calculates the factorial of the number and returns it. Although, it looks quite simple, it is good to understand the basic working principle of Web workers. Once we understand the basics, we shall discuss other issues.

In the main file (say `fact.html`), a dedicated Worker object using `Worker()` constructor (available in window object) is first created. Since, workers run in a separate thread, its code must be placed in a separate file. The constructor takes URL of this JavaScript file:

```
var fw = new Worker("fact.js");
```

The browser downloads the script and spawns a new thread that executes the script. It is a good idea to check if the browser supports it.

```

if (window.Worker) { // or if(typeof(Worker) !== "undefined") {
    var fw = new Worker("fact.js");
} else alert('Your browser does not support Worker');

```

The worker code (`fact.js`) and the JavaScript code created can now exchange data using HTML5 messaging API. Both send their messages using the `postMessage()` method, and uses a `onmessage` event listener to receive the message. Our `fact.js` looks like this:

```

//fact.js
onmessage = function (event){
    prod = 1;
    for(i = 2; i <= event.data; i++){
        prod = prod*i;
    }
    postMessage(prod);
}

```



This function gets called every time the main thread posts a message (5 in our case). The function gets an event object that has useful properties and methods. The message itself is available in the `data` property. The worker then calculates the factorial of the number and sends the result back to the main thread using `postMessage()` method.

The main thread, on the other hand, sends a number to the worker using the same `postMessage()` methods on worker object.

```
fw.postMessage(5);
```

However, it is always a good idea to set up a `onmessage` listener before sending a message.

```
fw.onmessage = function(e) {
    document.getElementById("result").innerHTML = e.data;
};
```

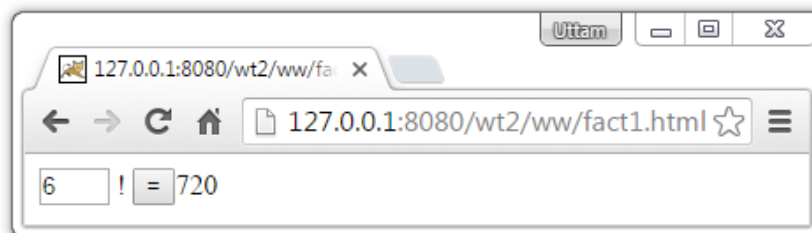
This function gets invoked when the worker posts messages. When the result comes back, it is shown in the element having id `"result"`. The complete code of the main thread is shown below:

```
<!--fact.html-->
<!DOCTYPE html>
<html>
<body>
<script>
    if (window.Worker) {
        var fw = new Worker("fact.js");
        fw.onmessage = function(e) {
            document.getElementById("result").innerHTML = e.data;
        };
        fw.postMessage(5);
    } else alert('Your browser does not support Worker');
</script>
<span id="result"></span>
</body>
</html>
```

Following is another example that allows you to enter an integer. The number is sent to the factorial worker and result obtained is displayed.

```
<!--fact1.html-->
<!DOCTYPE html>
<html>
<body>
<script>
    function process() {
        if (window.Worker) {
            var fw = new Worker("fact.js");
            fw.onmessage = function(e) {
                document.getElementById("result").innerHTML = e.data;
            };
            fw.postMessage(document.getElementById("no").value);
        }
        else alert('Your browser does not support Worker');
    }
</script>
<input type="text" id="no" value="1" size="1"> !
<input type="button" value="=" onClick=process()><span id="result"></span>
</body>
</html>
```

It has on text input element and

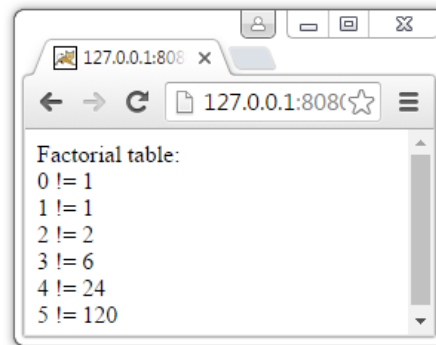


one button. A sample output is shown here:

There are two ways to stop a worker. The creator of the worker may invoke `terminate()` on Worker object or the worker itself can terminate it calling `close()` (or `self.close()`).

```
<!DOCTYPE html>
<html>
  <body>
    Factorial table: <output id="result"></output>
    <script>
      function create() {
        for(i=0;i<10;i++)
          w.postMessage(i);
      }
      var w = new Worker("fact.js");
      w.onmessage = function(event) {
        e = document.getElementById("result");
        e.innerHTML=e.innerHTML+'<br> '+event.data;
      }
      create();
    </script>
  </body>
</html>

onmessage = function (event){
  prod = 1;
  for(i = 2; i <= event.data; i++)
    prod = prod*i;
  postMessage(event.data + ' != ' + prod);
}
```



### 23.7.7.2 Loading External Scripts

A worker has the access to use `importScripts()` method to bring external scripts or libraries in its local scope. It accepts zero or more URLs (from the same domain) of the resources to import.

```
importScripts(); // imports nothing
importScripts('func.js'); // imports "func.js"
importScripts('func.js', 'tools.js'); // imports "func.js" and "tools.js"
```

### 23.7.7.3 Transferable object

When an object is passed from main thread to a worker thread (and vice versa), a copy of the object is passed to the target context keeping original object intact. This is done using *structured clone algorithm* defined by HTML5 specification. Using this algorithm, the object is essentially *serialized* at the sender end and serialized data is passed to the receiver where a copy of the original object is reconstructed (de-serialized) using serialized data.

The algorithm recursively traverses all the fields of the original object and duplicates the values of each field into a new object. Note that field objects may refer to other objects which in turn may refer to other object and so on. Moreover objects may also refer to themselves. Although, everything is done correctly, it takes significant amount of time for larger objects. To verify that let us write a very simple program that creates a 100 MB `ArrayBuffer`, passes it to a worker which simply sends it back, calculates round trip time and displays the result. Only the JavaScript code is shown here:

```

var w = new Worker("echo.js");
ab = new ArrayBuffer(1024*1024*100);
w.onmessage = function(event) {
    alert('Took '+(new Date()-start) +' ms.');
```

The echo worker is shown here:

```

//echo.js
onmessage = function (event){
    postMessage(event.data);
}
```

If we execute the program, it may take several milliseconds (it took 359 ms when executed). If we pass a large object several times, the program execution is expected to be very slow. However, often an object is not used by multiple contexts simultaneously.

Fortunately, chrome 17+ and firefox 18+ provide a facility where we can just transfer objects from one context to another context instead of copying which may improve the performance when sending large data. Objects are transferred using `postMessage()` in slightly different way.

```
w.postMessage(ab, [ ab ]);
```

This version of `postMessage()` method takes two arguments: first argument is the data and the second argument is the list of items from the data to be transferred. The above example says that the entire data `ab` has to be transferred.

Note that not all kind of data can be transferred. Objects that implement *Transferable* interface (currently `ArrayBuffer` and `MessagePort`) can only be transferred. The abstract interface *Transferable* does not define any methods, it merely indicate that an object implementing this is eligible for transfer. The first argument may be of any kind, but second argument must be an array of transferable objects.

```

obj = {data1: new ArrayBuffer(20), data2: new Int8Array(10)};
w.postMessage(obj, [obj.data1, obj.data2.buffer]);
```

It is possible to transfer a part of the object:

```
w.postMessage(obj, [obj.data2.buffer]);
```

Here, only second field of the object is transferred. Probably first field is still required by the caller and hence is not transferred, but copied.

Object transfer may be thought of as passing a *reference* to a method. However, unlike reference, the object is rather detached (closed) and moved (ownership changes) to the new context and will *not* be available (i.e. literally transferred) in the calling context thereafter. So, according to the specification “Transferring is an irreversible and non-idempotent operation. Once an object has been transferred, it cannot be transferred, or indeed used, again.” Following illustrates this:

```

var w = new Worker("echo.js");
ab = new ArrayBuffer(1024*1024*100);
alert(ab.byteLength);           //displays 104857600
w.postMessage(ab);              //object copy
alert(ab.byteLength);           //displays 104857600
w.postMessage(ab, [ab]);        //object is moved
alert(ab.byteLength);           //displays 0
w.postMessage(ab, [ab]);        //error no further transfer
```

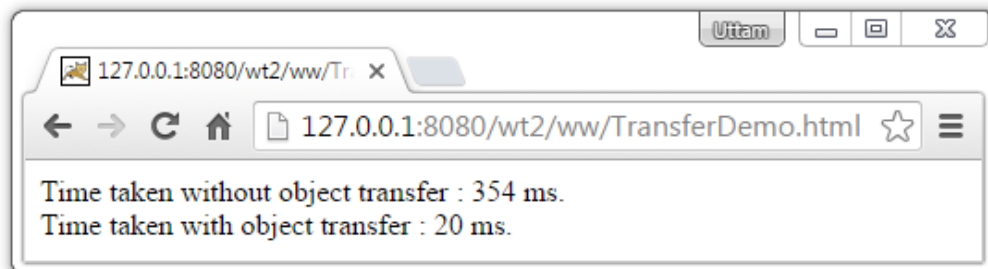
Assuming your browser supports transferable objects, following program demonstrates the effect of object copy and object transfer.

```
<!-- TransferDemo.html-->
<html>
<body>
  Time taken without object transfer : <output id="result"></output> ms.<br>
  Time taken with object transfer : <output id="result1"></output> ms.
  <script>
    var w = new Worker("echo.js");
    var w1 = new Worker("echo1.js");
    ab = new ArrayBuffer(1024*1024*100);      //100 MB object
    w.onmessage = function(event) {
      document.getElementById("result").innerHTML=(new Date()-start);
    }
    w1.onmessage = function(event) {
      document.getElementById("result1").innerHTML=(new Date()-start1);
    }
    start = new Date();
    w.postMessage(ab);
    start1 = new Date();
    w1.postMessage(ab, [ab]);
  </script>
</body>
</html>

//echo.js
onmessage = function (event){
  postMessage(event.data);
}

//echo1.js
onmessage = function (event){
  postMessage(event.data, [event.data]);
}
```

A sample snapshot is shown below:



#### 23.7.7.4 Shared Workers

The workers that we have created so far are called *dedicated workers* as they can be accessible from the script that creates them. There is another kind of worker called shared workers that can be accessed by multiple scripts having same origin (from different frame/iframe/window or workers ). A shared worker is created using `SharedWorker()` constructor:

```
sw = new SharedWorker('hello.js');
```

The script that creates it essentially gets a connection to shared worker. Scripts that use same URL will get connected to the same shared worker. The communication between scripts and

worker takes place according to Channel Messaging API. When a script gets connected to a shared worker, a message channel is established. Either party has the access to one of the two end points which are represented by `MessagePort` objects. Indeed `SharedWorker` instance has a property `port` of this kind and it has useful methods and properties to work with a shared worker such as sending and receiving messages. It is similar to HTML messaging API.

The endpoints of a channel created between a script and a shared worker does become active automatically. The script and the worker must activate it either by calling `start()` method or by registering `onmessage` handler that implicitly activates the end point (port). Note that `addEventListener()` does not implicitly activate the port.

Here we shall develop an HTML file (`outer.html`) which has a script and uses an `iframe` (having URL `inner.html`) which has another script. These two scripts simply send their names to a shared worker, which simply sends a greeting back to the scripts and scripts display them in their respective output field.

Here is out `outer.html` file:

```
<!--outer.html-->
<span id="result">Outer: </span>
<script>
  var sw = new SharedWorker('hello.js');
  sw.port.onmessage = function(e) {
    document.getElementById('result').innerHTML += '<br>' + e.data;
  }
  sw.port.postMessage('Tom');
</script>
<br><iframe src="inner.html" height="80"></iframe>
```

Observe that messaging API very similar except that `port` is used. Since, it uses `onmessage` handler, there is no need to use `start()` method. Alternatively, following could have been used:

```
sw.port.addEventListener('message', function(e) {
  document.getElementById('result').innerHTML += '<br>' + e.data;
}, false);
sw.port.start(); // necessary for addEventListener()
```

The code for inner is similar except it posts its name as “Jerry”.

```
<!--inner.html-->
<span id="result">Inner:</span>
<script>
  var sw = new SharedWorker('hello.js');
  sw.port.onmessage = function(e) {
    document.getElementById('result').innerHTML += '<br>' + e.data ;
  }
  sw.port.postMessage('Jerry');
</script>
```

The code for `hello.js` shared worker is shown below:

```
//hello.js
var count = 0;
onconnect = function(e) {
  count += 1;
  var port = e.ports[0];
  port.postMessage('Connected: ' + count);
  port.onmessage = function(e) {
    port.postMessage('Hello ' + e.data);
  }
}
```

The worker keeps track of the number of connections opened in a variable `count` which is incremented every time a connection is established i.e. `onconnect` handler gets called. Again, there is no need to use `start()` method as `onmessage` handler is used. A sample output is shown here:



#### 23.7.7.5 Restrictions

A script can only use a worker URL which has same protocol, port (if one is specified), and host. Its security mechanism prevents potentially malicious scripts from accessing arbitrary documents.

Web Workers having the URL starting with `file://` (i.e. from local file system) do not work.

Workers are relatively heavy-weight, and should not be created in large numbers.

A web worker has limited access to JavaScript features. For example it does not have direct access to `window`, `window.document`, `window.parent` objects. However, it has access to `navigator`, `location` objects. In short, a worker can not affect the creator page directly. So, a worker can't directly manipulate DOM tree. If it is really required, a message may be sent to the parent taking appropriate actions from there. Following is a list a web worker can use:

- `onmessage` event listener and `postMessage()` function.
- `navigator`, `location` object (read-only)
- `XMLHttpRequest` and send AJAX requests.
- `setTimeout()/clearTimeout()` and `setInterval()/clearInterval()` methods
- `atob()` and `btoa()` methods
- Web Sockets
- Web SQL Databases
- Application Cache
- external scripts using the `importScripts()` method
- other web workers

#### 23.7.8 WebSocket

This is a beautiful technology introduced to eliminate some of the problems of traditional HTTP. Before discussing how web sockets work, let us understand if the HTTP is adequate for implementing real time applications.

Note that HTTP is a client initiated protocol. A client, after establishing a connection to the server (using time-hungry three-way handshaking), makes a request for resource, server serves it and the closes the connection (non-persistent) using 4-way handshaking (even worse). For another request-response pair, exactly the same thing happens again(stateless). Although persistent HTTP (version 1.1) allows several request/response cycles to occur through one connection, the server has to keep the connection opened; thereby making the resources busy.

In HTTP, both the communicating parties (client and server) can send data, but not simultaneously; client first requests a resource, server then sends the response. However, many web applications (such as chat) require both-way-simultaneous data transfer (full-duplex).

Another problem of HTTP is the unnecessary transfer of some data. Every time client makes a request, some headers and cookies are sent to the server even much of these headers and cookies is not used. This may degrade the performance of real time web applications (e.g. games) where low-latency is very much desirable to keep them running smoothly.

It was always a dream to have a low-latency, persistent protocol where either one can initiate the data transfer. WebSocket exactly does it.

#### 23.7.8.1 How WebSocket works?

The WebSocket API allows setting up *socket* connections between a web server and a browser. The connection is essentially an enhanced persistent HTTP connection where either party, client or server, can send data at any time. In fact, they agree upon the protocol through a special HTTP request-response pair. Client includes a header `Upgrade` (having value `websocket`) in a regular HTTP request that notifies the server that the client wants a WebSocket protocol to be followed hereafter. Here is a sample HTTP request:

```
GET /wt2/echo HTTP/1.1
Host: localhost:6789
Connection: Upgrade
Upgrade: websocket
Origin: http://127.0.0.1:8080
Sec-WebSocket-Version: 13
...
```

If the server supports WebSocket protocol and agrees to switch to it, it informs the client using the same `Upgrade` header as follows:

```
HTTP/1.1 101 Switching Protocols
Server: Apache-Coyote/1.1
Upgrade: websocket
Connection: upgrade
...
```

At this point, both know that the other party has agreed upon the WebSocket protocol and continue speaking in WebSocket protocol i. e they can send and receive data (called *messages*) simultaneously. A message consists of one or more *frames*, each of which consists of a payload and a small header (4-12 bytes) so that the message can be reconstructed by the receiver. This way WebSocket reduces the amount of overhead (non-payload data) significantly; thereby reducing latency.

#### 23.7.8.2 Using WebSocket

A browser opens a connection using WebSocket constructor (available in the `window` object of a HTML5 compatible browser) that takes URL of the server as follows:

```
var ws = new WebSocket(url, [sub-protocol(s)]);
```

Here, first argument `url` is the URL of the server to connect to and optional second argument is the sub-protocol(s) [string/array of strings], at least one of which the server must accept. Here is an example:

```
var ws = new WebSocket("ws://localhost:8080/wt2/echo");
```

Note that the URL starts with `ws` indicating the **WebSocket** protocol to be followed. For secured WebSocket connection, `wss` is used in the same way `https` is used for secured HTTP connections. Since not all browsers support WebSocket, it is good idea to check it before use as follows:

```
if(window.WebSocket) {  
    ws = new WebSocket("ws://localhost:8080/wt2/echo");  
    //...work with ws  
}else alert('Your browser does not support Web socket');
```

The browser sends the data using `send()` method. Following sends a "Hello" message to the server:

```
ws.send("Hello");
```

Starting from establishing connection to closing the connection, the `WebSocket` object goes through different events such as `open`, `message`, `error`, `close` etc. Programmer may add handlers [Table 23.11:] to the interested events.

Table 23.11: WebSocket events and handlers

Event	Handler	Description
open	onopen	Occurs when the WebSocket connection is established
message	onmessage	Occurs when the browsers receives a message from server
error	onerror	Occurs if any communication error happens
close	onclose	Occurs when the connection is closed

Following adds a handler that gets called when the connection is established:

```
ws.onopen = function(evt) {  
    ws.send("Hello");  
};
```



As soon as the connection is established, the client can send data. The above sends a string "Hello" to the server. Binary data may also be transferred in `blob` or `arrayBuffer` format. Use `binaryType` property of `WebSocket` object to either 'blob' (default) or 'arraybuffer'.

```
ws.binaryType = 'arraybuffer';
```

Since server may also send messages, client should ideally set up a `onmessage` handler that gets called when the message arrives.

```
ws.onmessage = function(evt) {
    alert('Server said: '+evt.data);
};
```

The callback function gets an event object whose `data` property holds the actual message. The complete html file is shown here:

```
<html>
<body>
  <script language="javascript" type="text/javascript">
    function init() {
      if(window.WebSocket) {
        ws = new WebSocket("ws://localhost:8080/zt2/echo");
        ws.onopen=function(evt) {
          ws.send("Hello");
        }
        ws.onmessage = function(evt) {
          alert('Server said: '+evt.data);
        };
      }
      else alert('Your browser does not support WebSocket');
    }
    init();
  </script>
</body>
</html>
```

```
import java.io.IOException;
import java.nio.ByteBuffer;

import javax.websocket.OnMessage;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/echo")
public class Echo {
    @OnMessage
    public void echoTextMessage(Session session, String msg) {
        try {
            if (session.isOpen()) {
                System.out.println("Received from client: "+msg);
                session.getBasicRemote().sendText(msg);
            }
        } catch (IOException e) {
            try {
                session.close();
            } catch (IOException e1) {}
        }
    }
}
```

### 23.7.8.3 Server-side WebSockets

So far we have discussed how to use WebSocket at the client side. However a server must support this protocol to talk to the client. Since, large number of connections may be kept opened, a server must be designed very carefully to provide high concurrency at a low performance cost. Such designs often are governed by either threading or non-blocking I/O. Fortunately, WebSocket libraries have been implemented in many languages. Here are some examples:

- C++: libwebsockets
- Erlang: Shirasu.ws
- Java: Jetty
- Node.JS: ws, Socket.IO, WebSocket-Node
- Ruby: em-websocket
- Python: Tornado, pywebsocket
- PHP: Ratchet, phpws
- .NET: SuperWebSocket

The latest version of Tomcat also implements the Java WebSocket 1.1 API and can be found as jar file `websocket-api.jar` in its `lib` folder. We have only to define only an endpoint (a class), which is nothing but a Plain Old Java Object (POJO). Tomcat will handle the plumbing to route the request to the endpoint. The class can be written using annotations or without annotations. However, it is convenient to use annotations to define endpoint's methods. Here is simple implementation for our echo client:

```
import java.io.IOException;
import java.nio.ByteBuffer;

import javax.websocket.OnMessage;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint("/echo")
public class Echo {
    @OnMessage
    public void echoTextMessage(Session session, String msg) {
        try {
            if (session.isOpen()) {
                System.out.println("Received from client: "+msg);
                session.getBasicRemote().sendText(msg);
            }
        } catch (IOException e) {
            try {
                session.close();
            } catch (IOException e1) {}
        }
    }
}
```

The `@ServerEndpoint` annotation is used to give a relative name for the end point that will be accessed `ws://localhost:8080/wt2/echo`. Here `wt2` is the web application root directory and `echo` is the name of the endpoint. The `@OnMessage` annotation defines the method that gets called when a client sends a message. Other annotations such as `@OnOpen`, `@OnClose` allow us to define methods to intercept opening and closing new connections.

WebSocket is often used to developed chat application. Following the is a very simple client interface.

```

<html>
<body>
  <script lang="JavaScript">
    function start(name) {
      if (window.WebSocket) {
        socket = new WebSocket('ws://' + window.location.host +
'/wt2/chat/'+name);
      } else {
        show('Error: Your browser does not support WebSocket.');
```

This has two input text fields (name and chat), one button and one output field. The first input field accepts the name of the person want to chat. The chat messages are typed in the second text field. The output field shows the messages. The onopen handler sets up a onkeydown handler for chat text field. When the user types something in this field and hits enter, the typed text is sent to the server. The onmessage handler for WebSocket object simply shows the incoming messages in the output field.

The server side code is given below:

```

import java.io.IOException;
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import javax.websocket.server.PathParam;
import java.util.*;

@ServerEndpoint(value = "/chat/{name}")
public class ChatServer {
    static final List<ChatServer> handlers = new ArrayList<>();
    private String name;
    private Session session;
```

```

@OnOpen
public void onOpen(@PathParam("name") String name, Session session) {
    this.session = session;
    this.name=name;
    handlers.add(this);
    sendToClients(name+" joined");
    System.out.println(name);
}

@OnClose
public void onClose() {
    handlers.remove(this);
    sendToClients(name+" has left");
}

@OnMessage
public void onMessage(String msg) {
    sendToClients(name+": "+msg);
}

private static void sendToClients(String msg) {
    for (ChatServer handler : handlers) {
        try {
            synchronized (handler) {
                handler.session.getBasicRemote().sendText(msg);
            }
        } catch (IOException e) {
            handlers.remove(handler);
            try {
                handler.session.close();
            } catch (IOException e1) {}
            sendToClients(handler.name+" has discontinued.");
        }
    }
}
}

```

The {name} in the @ServerEndpoint annotation is a parameter (name of the person involving chat) and is passed by the client in the URL form

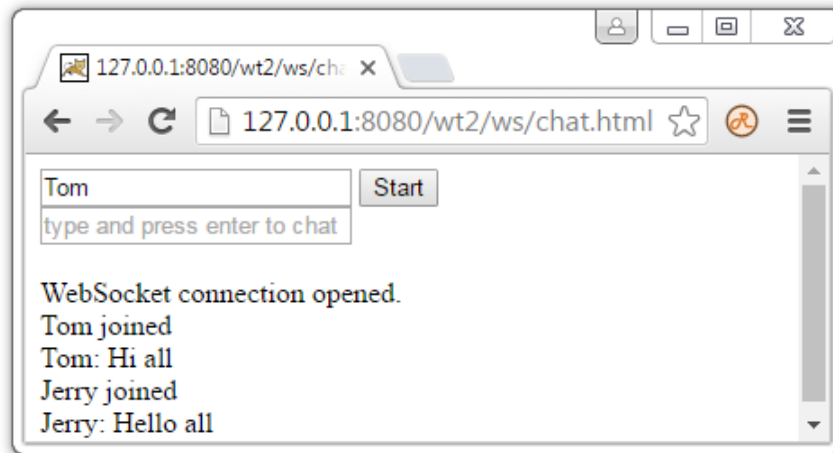
```
ws://host/webAppRoot/chat/[name].
```

The class maintains a static list of chat servers and defines three handlers onOpen, onMessage and onClose where appropriate messages are delivered to all the registered clients.

Put this class file in WEB-INF/classes directory of our web application. To compile this class, we need WebSocket API that can be found as a single JAR file websocket-api.jar in Tomcat's lib folder. Use -cp option to include this JAR during compilation. Assuming you are in WEB-INF/classes directory, use following command to compile:

```
javac -cp ../../../../lib/websocket-api.jar;. ChatServer.java
```

Restart the Tomcat so that the Tomcat can now load this class file. A snapshot of the client interface is shown below:



### 23.7.9 Server Sent Events(SSE)

Note that HTTP is a client-initiated protocol where client asks a resource and server serves it. It was always a demand for server-initiated data transfer towards client. Previously use used to use polling, long polling etc. to get data from server. However, each one has its one set of disadvantages. For example, for polling an important question arises, “how frequently a client should poll the server?” Since, a client does not know when data arrives to the server, finding an appropriate interval of polling is impossible. This means that a client may either get duplicate data (overhead) or miss (even worse) some data.

Although, WebSocket allows server to send data to the client at any time, it requires special support from the web server. Needless to say not all web servers automatically come with WebSocket support. Moreover, WebSocket opens a bidirectional, full-duplex channel and hence costly. However, there are situations where client need not send any data; simply need updates from server. Example includes score of a game (such as cricket/football match), stock value, friend’s status, news etc. So, we need one that works under HTTP and supports server-initiated data transfer. Server Sent Events (SSE) does exactly that. A server can push data to client whenever it wants, without any request. Server-Sent Events have also some features such that automatic reconnection, event IDs, and the ability to send arbitrary events which WebSockets lack.

#### 23.7.9.1 SSE API

The heart of SSE is the EventSource object that abstracts all the low-level connection establishment and message parsing. To subscribe to a server event, a client simply creates an EventSource (available in the window object) object passing the URL of the data generator.

```
es = new EventSource("time.jsp");
```

Here time.jsp is a JSP program that generates some data in the format as specified by SSE. In fact the program may a PHP, PERL, Python or any other script that generates data appropriately. We shall discuss server side program very soon. It is good idea to check if a browser supports SSE as follows:

```
if(typeof(EventSource) !== "undefined") {
    es = new EventSource("time.jsp");
    //...
}
else alert('Your browser does not support SSE.')
```

The client then registers event listeners on this object:

```
es.onmessage = function (evt) {  
    //process evt.data  
};
```

This is essentially a subscription to all events i.e. the handler gets called every time the server pushes any message. The data is available in the `data` property of `evt` object. If the connection is lost, client tries to reconnect to the event source after 3 seconds, which the server may customize. Alternatively, a listener can be registers using `addEventListener()` method as follows:

```
es.addEventListener("message", function(evt) {  
    //process evt.data  
}, false);
```

The client can optionally register `onopen` and `onerror` handlers. They are invoked when the connection is opened or any error occurs respectively.

```
es.onopen = function (evt) {  
    // do something when the connection opens  
};  
  
es.onerror = function (evt) {  
    // do something if an error occurs  
}
```

Here is a complete HTML code that shows server time.

```
<!DOCTYPE html>  
<html>  
<body>  
<div id="time"></div>  
<script>  
if(typeof(EventSource) != "undefined") {  
    es = new EventSource("time.jsp");  
    es.onmessage = function(evt) {  
        document.getElementById("time").innerHTML = evt.data;  
    };  
} else  
document.getElementById("time").innerHTML = "Your browser does not support  
SSE...";  
</script>  
</body>  
</html>
```

This creates an `EventSource` object specifying URL (`time.jsp`) of the time data generator. When the data is received, `onmessage` handler is called. In the handler, data (event `evt.data`) is placed in `<div>` element having id `"time"`.

### 23.7.9.2 Server Side Program for SSE

Any server side program may be used to push data. The only requirement is data must be plaintext (UTF-8 encoded) in SSE format and there must be a `Content-type` header having value `text/event-stream`. The data must start with the word `data:` followed by actual plaintext message followed by two `'\n'` characters. Here is out program (`time.jsp`) in JSP.

```
<%  
response.setContentType("text/event-stream");  
response.setCharacterEncoding("UTF-8");  
out.print("retry: 1000\n");  
out.print("data: " + new java.util.Date() + "\n\n");  
%>
```

The program first sets the Content-type header to text/event-stream. Since Java characters are Unicode characters and SSE supports only UTF-8 encoding data, the character encoding is set as UTF-8 in the second line. Third line asks the client to reconnect after 1000 milliseconds (1 second) and poll so that the client sees the time continuously. The last line actually sends the current date and time preceded by 'data: ' followed by two new line characters. To avoid polling every after ~3seconds, we can use a while loop as follows:

```
<%
response.setContentType("text/event-stream");
response.setCharacterEncoding("UTF-8");
//out.print("retry: 1000\n");
try {
    while(true) {
        out.print("data: " + new java.util.Date() + "\n\n");
        out.flush();
        Thread.sleep(1000);
    }
} catch (Exception e) {out.println(e);}
%>
```

The same code in PHP is here:

```
<?php
header('Content-Type: text/event-stream');
while(true) {
    echo "data: " . date("Y/m/d h:i:sa") . "\n\n";
    flush();
    sleep(1);
}
?>
```

### 23.7.9.3 Event Stream Format

Server can push multiple messages separated by a blank line. A SSE message consists of one or more lines. Each line starts with a field in the form "field-name: " followed by text of the field. The SSE specification defines following field names:

data

The text following it is data

event

Text following it is the name of the event. Data following this event belong to this event

id

Text following it is the identifier of the message

retry

Text following it is an integer represents the reconnection time (in milliseconds) for the client

A line with the absence of a field name is treated as comment at the client side and is not dispatched to any listener. If a server sends messages very infrequently, it may send comments to prevent connections from timing out. Here is an example:

```
: This is a comment\n\n
```

A very long text data may be sent as multiple lines preceded by "data: ". Each line of the data chunk ends with a single "\n" character except the last line which ends with "\n\n". Here is an example:

```
data: This is a multi-line message.\n
data: It has two lines.\n\n
```

At the receiving end, all data (including non-trailing newline characters) are merged and passed to the client. The newline characters may be deleted from the data as follows:

```
data = evt.data.replace(/\n/g, "");
```

Data may contain complicated semantic information such as JSON data as follows:

```
data: {"user": "Tom", "loginTime": "10:35:03"}\n\n
```

The client can then parse and extract data as follows:

```
var data = JSON.parse(evt.data);
//process data.user and data.loginTime
```

Server-sent events are message events by default, i.e. if "event: " is not present for some message, the event is assumed to be message event. Accordingly, these messages are dispatched to onmessage listeners. A message may be stamped by a custom event name using "event: " field followed by the name of the event.

```
event: login\n
data: {"user": "Tom", "loginTime": "10:35:03"}\n\n
```

This sends a message for login event. To receive this message, client will register a listener using addEventListener() as follows:

```
es.addEventListener('login', function(evt) {
    var data = JSON.parse(evt.data);
    var user = data.user;
    var loginTime = data.loginTime;
    //process user and loginTime
}, false);
```

We can mix up unnamed and named messages arbitrarily:

```
: This is a comment\n\n

data: Hello from server\n\n

event: login\n
data: {"user": "Tom", "loginTime": "10:35:03"}\n\n

data: This is a multi-line message.\n
data: It has two lines.\n\n

event: logout\n
data: {"user": "Tom", "logoutTime": "11:15:30"}\n\n
```

A message can be given an optional ID:

```
id: 1\n
data: Hello from server\n\n

id: 2\n
data: Hello from server\n\n
```



Browser remembers last id seen and sends it as the value of Last-Event-ID HTTP head to the server during reconnection. This helps the server to identify which messages were missed by the client and to send appropriate message thereafter.

The EventSource object has a default reconnection interval typically 3 to 6 seconds. However, server can instruct the client to use a custom interval using retry field as follows:

```
retry: 5000\n
```

A client unsubscribe using `close()` method on EventSource object as follows:

```
es.close();
```

## 23.8 Keywords

**DOCTYPE**—A declaration that tells the browsers that it is an HTML5 document

**Semantic Elements**—A semantic element is essentially any element with concrete meaning

**Graphics elements**—New feature of HTML5 which deal with graphics on the web

**Canvas**—An element that allows us to draw a variety of graphic elements and animate them very easily

**SVG**—It is a XML based technology introduced in HTML5 for specifying 2-D graphic elements

**Media elements**—Elements introduced in HTML5 used to insert media contents in web pages.

**Video element**—The `<video>` element used to insert video content in the web pages

**Audio element**—The `<audio>` element used to insert audio content in the web pages

**Geolocation**—An API used find the geographic location (latitude and longitude) of the user navigating the page

**Drag and Drop**—An API used to provide drag and drop facility to the users

**File API**—An API that provides a standard way to work with local files

**Web Storage**—It is one of the local (to client) storage facilities introduced in HTML5

**Indexeddb**—It allows us to store huge data persistently in the browser (client-side) and an efficient indexed-based searching API to retrieve the data.

**Application Cache**— A technology that helps a web application to selectively cache data in the web make available to users during offline.

**Web workers**—Web worker that can run scripts concurrently with the main thread without interfering the main thread

**WebSocket**—A low-latency, persistent communication protocol between two parties where either one can initiate the data transfer.

**Server Sent Events**—A one way communication protocol where server can push data to client.

## 23.9 Summary

HTML 5 was released in 2014 introducing a wide range of components to handle current situations. For example new elements and attributes for better semantics, canvas for drawings, playing media (audio, video etc.), drag-and-drop, handling location, new form elements were introduced.

HTML5 introduced 13 new input elements to be used for forms. These elements can accept variety of data such as date (and related), email, URL, color, search string etc. In addition, a number of new semantic elements were introduced for better structuring.

There are two primary technologies for graphics; canvas and Scalable Vector Graphics (SVG). The element `<canvas>` allows us to draw a variety of graphic elements and animate them very easily. SVG is a XML based technology for specifying 2-D graphic elements (such as line, circle, rectangle, ellipse etc.) and their operations (rotation, translation etc.) for the web. The media API is a competing, open standard for easily and cleanly embedding media into web pages with its native media elements (`<video>` and `<audio>`).

HTML5 also provides some beautiful JavaScript APIs such as Geolocation, Drag and Drop, File API, Web Storage, IndexedDB, AppCache, Web workers, WebSocket, SSE that can be used to build powerful web applications.

Geolocation API, as the name indicates, allows a web page to find the geographic location (latitude and longitude) of the user navigating the page if the user desires to provide so.

The Drag and Drop (DnD) feature provides facilities to drag and element and drop it over another element. File API provides a standard way to work with local files. We can load files and compress, encode, encrypt, or even upload them in smaller chunks using JavaScript.

Web Storage is a persistent storage within the web browser where web pages can store key-value pairs, retrieve/use them later. IndexedDB allows us to store huge data (maximum of 50% of free disk space) persistently in the browser (client-side) and an efficient indexed-based searching API to retrieve the data.

AppCache helps a web application developer to selectively cache in the web browser everything from pages to images to scripts to cascading style sheets and make available to users during offline.

Web workers can run scripts concurrently with the main thread without interfering the main thread. Workers run in a context other than main thread's context and typically execute a computationally intensive task. WebSocket is a low-latency, persistent communication protocol between two parties where either one can initiate the data transfer. SSE on the other hand is a one way communication protocol where server can push data to client.

## 23.10 Web Resources

<https://www.w3.org/TR/html5/>

HTML5, A vocabulary and associated APIs for HTML and XHTML

<https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>

HTML5

[http://www.w3schools.com/html/html5\\_intro.asp](http://www.w3schools.com/html/html5_intro.asp)

HTML5 Introduction

<http://tutorials.jenkov.com/html5/index.html>

HTML5 Tutorial

<http://www.tutorialspoint.com/html5/>  
HTML5 Tutorial

<http://www.html5-tutorials.org/>  
HTML5 Tutorial

<http://www.codeproject.com/Learn/HTML/>  
Learn HTML5 and CSS3

## 23.11 Exercises

### 23.11.1 Objective Type Questions

1. Which of the following input type represents a date according to ISO format?
  - a) datetime
  - b) datetime-local
  - c) date
  - d) month
2. Which of the following element holds some content that is slightly related to the rest of the page in HTML5?
  - a) section
  - b) article
  - c) header
  - d) aside
3. Which of the following is true about localStorage in HTML5?
  - a) Deleted after a window or tab is closed
  - b) Persists event after all windows/tabs are closed.
  - c) Deleted after a session is closed
  - d) None of the above.
4. Which of the following is true about sessionStorage in HTML5?
  - a) Available only in a page.
  - b) Persists event after all windows/tabs are closed.
  - c) The Session Storage Data would be deleted by the browsers immediately after the session gets terminated.
  - d) None of the above.
5. Which of the following attribute triggers event when media has reached the end?
  - a) oncompleted
  - b) ondurationchange
  - c) onfinished
  - d) onended
6. Which of the following attribute triggers event when the document goes offline?
  - a) ondisconnected
  - b) onoffline
  - c) onended

- d) ononline
- 7. Which of the following tag represents an independent piece of content of a document in HTML5?
  - a) section
  - b) article
  - c) aside
  - d) header
- 8. Which of the following tag represents a section of the document intended for navigation in HTML5?
  - a) nav
  - b) footer
  - c) section
  - d) dialog
- 9. Which of the following input type accepts only numerical value?
  - a) week
  - b) time
  - c) number
  - d) range
- 10. Which value of Socket.readyState attribute of WebSocket indicates that the connection is established and communication is possible?
  - a) 0
  - b) 1
  - c) 2
  - d) 3
- 11. Which of the following tags is used to hold some result?
  - a) output
  - b) placeholder
  - c) holder
  - d) result
- 12. Which of the following method returns a geolocation object in HTML5?
  - a) window.geolocation
  - b) document.geolocation
  - c) navigator.geolocation
  - d) None of the above.
- 13. Which of the following events occur when an element is dropped on another element?
  - a) drag
  - b) leave
  - c) push
  - d) drop
- 14. Which of the following feature(s) is/are does HTML 5 have?
  - a) Geolocation
  - b) Canvas

- c) Audio & Video
  - d) All of the above
15. Which of the following methods deletes entire localStorage?
- a) clear()
  - b) delete()
  - c) removeAll()
  - d) empty()
16. Which of the following method retrieves the current geographic location of the user?
- a) geolocation.currentPosition()
  - b) geolocation.returnCurrentPosition()
  - c) geolocation.watchPosition()
  - d) geolocation.getCurrentPosition()
17. Which of the following is correct content-type of server-side data for SSE?
- a) text/stream-event
  - b) text/event-stream
  - c) text/event-type
  - d) text/event-data
18. Which of the following tags represents a generic document or application section in HTML5?
- a) <section>
  - b) <article>
  - c) <aside>
  - d) <header>
19. The correct DOCTYPE declaration in HTML5 is
- a) <!DOCTYPE html:>
  - b) <!DOCTYPE html!>
  - c) <!DOCTYPE html;>
  - d) <!DOCTYPE html>
20. Which of the following is used to draw graphic elements in HTML5?
- a) <canvas>
  - b) <svg>
  - c) Both a) and b)
  - d) None of the above
21. The tag to specify illustrations, diagrams, photos is<br/>
- a) <mark>
  - b) <figure>
  - c) <draw>
  - d) <diagram>
22. The element to specify a list of predefined options for input controls is<br/>
- a) <datalist>
  - b) <embed>
  - c) <source>

- d) <list>
23. What is the full form of SVG?
- a) Scalable Vector Graph
  - b) Scaled Vector Graphics
  - c) Scaled Vector Graph
  - d) Scalable Vector Graphics
24. What is the full form of SSE?
- a) Secured Server Events
  - b) Server-Sent Events
  - c) Scalable Streaming Events
  - d) Server Secured Events
25. What the storages available in HTML5?
- a) localStorage and sessionStorage
  - b) sessionStorage and pageStorage
  - c) dataStorage and sessionStorage
  - d) localStorage and sessionStorage
26. Which of the following is not true about HTML5?
- a) Reduces the need for external plug-in
  - b) Improves error handling
  - c) Uses more markups to replace scripting language
  - d) All of the above
27. Which of the following is not a new feature of HTML5?
- a) <canvas>
  - b) <video>
  - c) <audio>
  - d) <div>
28. Which of the following is not form controls elements in HTML5?
- a) currency
  - b) url
  - c) search
  - d) email
29. Which of the following is not new element of HTML5?
- a) <details>
  - b) <summary>
  - c) <figure>
  - d) <colgroup>
30. Which of the following is not method of <video> element?
- a) load()
  - b) play()
  - c) pause()
  - d) stop()

31. Which of the following is true about video in HTML5?
- a) HTML5 doesn't support video.
  - b) To play video we must have to use some plug-in.
  - c) It defines a new element which specifies a standard way to embed a video.
  - d) Both the options (a) and (b)
32. Which of the following is not property of <video> element?
- a) duration
  - b) abort
  - c) currentTime
  - d) currentSrc
33. What is correct behavior of preventDefault() function:
- a) Prevent the drop event
  - b) Prevent the load data in browser.
  - c) Prevent the browser default handling of the data
  - d) Prevent the drag event
34. Which of the following is true about the following?  
<video src="song.mp4" controls preload="none" />
- a) It should not load the video when the page loads.
  - b) It should load the entire video when the page loads.
  - c) It should load the part of video when the page loads.
  - d) It should load only metadata when the page loads.
35. The type attribute of <source> element for a video is:
- a) Media compression type
  - b) Media source type
  - c) MIME type
  - d) Media extension
36. How to make an element draggable:
- a) <div type="draggable"/>
  - b) <div type="drag" />
  - c) <div draggable="true" />
  - d) <div drag="true" />
37. Which of the following is not a valid event of Drag and Drop?
- a) ondrop
  - b) ondragstart
  - c) ondragend
  - d) ondragover
38. Which method is used to get the dragged data?
- a) getData()
  - b) getDragData()
  - c) returnData()
  - d) dragData()

39. Which of the following is not event of <video> element?
  - a) play
  - b) error
  - c) pause
  - d) progress
40. Which of the following is not property of <video> element?
  - a) abort
  - b) seeking
  - c) width
  - d) volume

### 23.11.2 Subjective Type Questions

1. What is the difference between HTML and HTML5 ?
2. What are some of the key issues with previous HTML versions that HTML5 is addressing?
3. Name some of the new features of HTML5.
4. How would you typical HTML5 document look, and what are the primary differences between that, and an older version of HTML?
5. What are the new Form elements made available in HTML5
6. What is the purpose of <output> tag in HTML5?
7. What are the differences between localStorage and sessionStorage?
8. Describe the steps required to add and remove data to/from local storage?
9. What is the difference between web storage and cookies?
10. What is the difference between web storage and IndexedDB?
11. Describe with examples the purpose of HTML5 semantic elements.
12. What is Server Sent Events and how is it different from Web Socket?
13. Write the basic steps to write a server side scripts for SSE.
14. Explain the notion of WebSocket. Where Web Sockets can be used?
15. Explain with example the concept of Drag and Drop feature.
16. Describe with examples the structure of cache manifest file.
17. Write relative merits and demerits of <canvas> and <svg>.
18. What are the different types of storage in HTML 5?
19. How to draw a rectangle using Canvas and SVG using HTML 5
20. What is Geolocation API in HTML?
21. What are the various elements provided by HTML 5 for media content?
22. What are Web Workers?
23. What is the difference between getCurrentPosition() and watchPosition() methods in Geolocation API?
24. Demonstrate how to use SVG to draw graphics?
25. What is the purpose of <section> tag in HTML5?
26. What are the drawbacks of cookies?
27. What is the purpose of datetime input control in
28. What is the purpose of <nav> tag in HTML5?
29. What are the new APIs provided by HTML 5 standard?
30. How to use HTML5 details and summary elements?
31. What are the limitations of HTML5 Web Worker?
32. What is the difference between HTML5 Application cache and regular HTML browser cache?
33. What is the <datalist> element in HTML5? Use it in an example. Are there any limitations to it?



34. What is SVG and what are its advantages of over canvas?
35. How to play/pause video using JavaScript?