HTML INTRODUCTION

What is HTML and How it Works?

HTML Stands for HyperText Markup Language, where HyperText stands for Link between web pages.Markup Language means Text between tags that define the structure.HTML is a markup language that is used to create web pages. It defines how the web page looks and how to display content with the help of elements. It forms or defines the structure of our Web Page, thus it forms or defines the structure of our Web Page. We must remember to save your file with .html extension. In this HTML Tutorial, we’ll understand all the basic concepts required to kick-start your journey HTML is the standard markup language for Web pages.HTML is used to create web pages and web applications.We can create a static website by HTML only.Technically, HTML is a Markup language rather than a programming language.HTML is widely used language on the web.

What is HTML

HTML is an acronym which stands for **Hyper Text Markup Language**

Hyper Text: HyperText simply means "Text within Text." A text has a link within it, is a hypertext. Whenever you click on a link which brings you to a new webpage, you have clicked on a hypertext. HyperText is a way to link two or more web pages (HTML documents) with each other.

Markup language: A markup language is a computer language that is used to apply layout and formatting conventions to a text document. Markup language makes text more interactive and dynamic. It can turn text into images, tables, links, etc.

Web Page: A web page is a document which is commonly written in HTML and translated by a web browser. A web page can be identified by entering an URL. A Web page can be of the static or dynamic type. **With the help of HTML only, we can create static web pages**.

Applications of HTML

HTML is used for various purposes. Let us take a look at them

1. **Web Pages Development**
   HTML is famously used for creating web pages on the world wide web. Every web page contains a set of HTML tags and hyperlinks which are used to connect other pages. Every page on the internet is written using HTML.
2. **Navigating the Internet**
   Navigating on the internet would have been quite a tedious task without HTML. The anchor tags of HTML allows us to link pages and navigate easily. Imagine our life without anchor tags, you would

literally have to enter URL everytime. Using achor tags, you can also navigate within a webpage.

3. **Embedding Images and Videos**
   HTML allows us to embed images and videos with ease and gives us features to adjust height, position and even rendering type. You can adjust controls, thumbnails, timestamps and much more for videos. Earlier this was done using Flash and HTML has made it easier with the help of **<video>** tag.
4. **Clinet-side storage**
   HTML5 has made client-side storage possible using localStorage and IndexD due to which we no longer need to reply on Cookies. Both of these tactics have their own set of rules and characteristics. String-based hash-table storage is provided by localStorage. Its API is straightforward, with setItem, getItem, and removeItem functions available to developers. On the other hand, IndexDB is a larger and more capable client-side data store. With the user's permission, the IndexDB database can be enlarged.
5. **Game development**
   Although you cannot create complex high-end video games with HTML, the **<canvas>** element of HTML can be used to make 2D and 3D games using CSS and JavaScript which can be run on browsers.

6. **Data entry support**
   With the usage of new HTML5 standards in all the latest browsers, developers can simply add the tags for required fields, text, data format, etc. and get the data. HTML5 now has several new attributes for data-entry and validation purposes.
7. **Interacting with Native APIs**
   With the help of HTML, you can interact with your Operating system. With this feature, you can easily drag files onto a web page to upload, full-screen a video, and much more.

Features Of HTML

- The learning curve is very easy (easy to modify)
- Creating effective presentations
- Adding Links wherein we can add references
- Can display documents on platforms like Mac, Windows, Linux, etc
- Adding videos, graphics, and audios making it more attractive
- Case insensitive language

HTML Editor

- Simple editor: Notepad
- Notepad++
- Atom
- Best editor: Sublime Text

HTML Skeleton

```
<!DOCTYPE html>

<html>

<head>

<title></title>

</head>

<body>

</body>

</html>
```

HTML Basic

<!DOCTYPE html>

Instruction to the browser about the HTML version.

<html>

- Root element which acts as a container to hold all the code
- The browser should know that this is an HTML document

- Permitted content: One head tag followed by one body tag

<head>

- Everything written here will never be displayed in the browser
- It contains general information about the document
- Title, definitions of CSS and script sheets
- Metadata(information about the document)

<body>

- Everything written here will be displayed in the browser
- Contains text, images, links that can be achieved through tags
- Examples:
- ○ <p> This is our first paragraph. </p>
- ○ <a href=”http://www.google.com”>Go To Google</a>
- ○ <img src=”photo.jpg”>

HTML Comment

- Comments don’t render on the browser
- Helps to understand our code better and makes it readable.
- Helps to debug our code
- Three ways to comment:

1. Single line
2. Multiple lines
3. Comment tag //Supported by IE

HTML Elements

- Elements are created using tags
- Elements are used to define the semantics
- Can be nested and empty

What is HTML Elements Definition

```
<p color=”red”> This is our first Paragraph </p>
```

Explanation:

- Start tag: <p>
- Attributes: color =” red”
- End tag : </p> // optional
- Content: This is our first Paragraph

Types of Elements

There are different types of elements in HTML. Before moving ahead in the HTML Tutorial, let us understand the types of elements.

- **Block Level**

○ Takes up full block or width and adds structure in the web page

○ Always starts from the new line

○ Always end before the new line

○ Example:

■ <p >

■ <div>

■ <h1>…<h6>

■ <ol>

■ <ul>

- **Inline Level**

○ Takes up what is requires and adds meaning to the web page

○ Always starts from where the previous element ended

○ Example :

■ <span>

■ <strong>

■ <em>

■ <img>

■ <a>

HTML Attributes

- Properties associated with each tag is called an Attribute.
- <tag name=”value”></tag> is the structure.
- There are some Global Attributes that can be applied to all the tags.
- Title: Add extra information (hover)
- Style: Add style information(font,background,color,size)
- There are some attributes that can be applied to specific tags.
- <img src=” URL” width=”100” height=”70” alt=” File cannot be loaded”>
- src is the attribute used in image tag to define the path
- Width is an attribute used to define width in pixels
- Height is an attribute used to define width in pixels
- Alt i.e alternate text if an image is not loaded
- <u>Name of the link</u>
    - href is used for defining the path of the link

HTML Tags

- Enclosed within <>
- Different tags render different meanings.

HTML TAGS

A HTML heading or HTML h tag can be defined as a title or a subtitle which you want to display on the webpage.

There are 6 types of heading tag

1. **<h1>**Heading no. 1**</h1>**
2. **<h2>**Heading no. 2**</h2>**
3. **<h3>**Heading no. 3**</h3>**
4. **<h4>**Heading no. 4**</h4>**
5. **<h5>**Heading no. 5**</h5>**
6. **<h6>**Heading no. 6**</h6>**

o/p : H **Heading no. 1**

Heading no. 2

Heading no. 3

Heading no. 4

Heading no. 5

Heading no. 6

HTML Paragraphs

The HTML <p> element defines a paragraph.

A paragraph always starts on a new line, and browsers automatically add some white space (a margin) before and after a paragraph

<p>welcome to mumbai</p>

o/p: welcome to mumbai

HTML Phrase tag

- Abbreviation tag : <abbr>
- Marked tag: <mark>
- Strong tag: <strong>
- Emphasized tag : <em>
- Definition tag: <dfn>
- Quoting tag: <blockquote>
- Short quote tag : <q>
- Code tag: <code>
- Keyboard tag: <kbd>
- Address tag: <address>

1. Text Abbreviation tag

1.<p>those are<abbr title = "Hypertext Markup language">Fron-tend Technology </abbr>languages is used to create web pages.

o/p: those are Front-end Technologylanguages is used to create web pages.

2. Marked tag:

<p>This tag will <mark>HTML</mark> the text.</p>

O/P: This tag will HTML the text.

3. Strong text:

This tag is used to display the important text of the content.

<p>In HTML,CSS,JS <strong>FRONT-END</strong>

O/P: In HTML,CSS,JS **FRONT-END**

4. Emphasized text

This tag is used to emphasize the text, and displayed the text in italic form.

7. **<p>**HTML is an **<em>**easy **</em>**to learn language.**</p>**

o/p: HTML is an *easy* to learn language.

5. Quoting text:

The HTML <blockquote> element shows that the enclosed content is quoted from another source. The Source URL can be given using the cite attribute, and text representation of source can display using **<cite> ..... </cite>element**.

```
<!DOCTYPE html>

<html>

<head>

</head>

  <body>

     <h2>Example of blockquote element</h2>

      <blockquote><p>"You are the one who can make your life You are the one who can distroy your life"</p></blockquote>

      <cite>-Anjali p</cite>

  </body>

</html>
```

o/p: "You are the one who can make your life You are the one who can distroy your life"

-Anjali p

6. Short Quotations:

An HTML <q> .... </q> element defines a short quotation. If you will put any content between <q> .... </q>, then it will enclose the text in double quotes.

```
<!DOCTYPE html>
<html>
  <head>
</head>
  <body>
     <p>Great Motivational quote</p>
      <p>sonal p: <q>hardwork is key of success</q>?</p>
   </body>
</html>
```

o/p:

```
Great Motivational quote
sonal p: hardwork is key of success?
```

7. Code tags

The HTML <code> </code> element is used to display the part of computer code. It will display the content in monospaced font.

8. Keyboard Tag

In HTML the keyboard tag, <kbd>, indicates that a section of content is a user input from keyboard.

9. Address tag

An HTML <address> tag defines the contact information about the author of the content. The content written between <address> and </address> tag, then it will be displayed in italic font.

```
<!DOCTYPE html>

<html>

  <head>

</head>

  <body>

     <p>Address Tag</p>

    <address> my email id is: <a href=" ">anaht123@hjahjh.com</a>

    <br> You can also visit at: <br>78 riverstone back side of temple near
sweet mart

   </address>
  </body>

</html>
```

o/p: Address Tag

my email id is: anaht123@hjahjh.com
You can also visit at:
78 riverstone back side of temple near sweet mart

For Example :

```
1. <!DOCTYPE html>
2. <html>
3.   <head>
4.     <title>html website</title>
5.  </head>
6.   <body>
7.        <h2>Thi is heading tag</h2>
8.        <p>This is a paragraph tag</p>
9.        <p style="color: red">The style is attribute of paragraph tag</p>
10.             <span>The element contains tag, attribute and content</span>
```

11. </body>

12. </html>

HEADER AND FOOTER TAG

<header>

The **<header>** HTML element represents introductory content, typically a group of introductory or navigational aids. It may contain some heading elements but also a logo, a search form, an author name, and other elements.

FOR EXAMPLE:

```
<header>
  <a class="logo" href="#">Cute Puppies Express!</a>
</header>
```

<article>

Usage notes

The <header> element has an identical meaning to the site-wide banner landmark role, unless nested within sectioning content. Then, the <header> element is not a landmark.

The <header> element can define a global site header, described as a banner in the accessibility tree. It usually includes a logo, company name, search feature, and possibly the global navigation or a slogan. It is generally located at the top of the page.

Otherwise, it is a section in the accessibility tree, and usually contain the surrounding section's heading (an h1 – h6 element) and optional subheading, but this is **not** required.

Page Header

```
<header>
  <h1>Main Page Title</h1>
  <img src="mdn-logo-sm.png" alt="MDN logo" />
</header>
```

Article Header

```
<article>
  <header>
    <h2>The Planet Earth</h2>
    <p>
      Posted on Wednesday, <time datetime="2017-10-04">4 October
2017</time> by
      Jane Smith
    </p>
  </header>
  <p>
    We live on a planet that's blue and green, with so many things still unseen.
  </p>
  <p><a href="https://example.com/the-planet-earth/">Continue reading…</a></p>
</article>
```

TABLE TAGS:

A table in HTML consists of table cells inside rows and columns.

HTML table tag is used to display data in tabular form (row * column). There can be many columns in a row.

Tag	**Description**
<table>	Defines a table
<th>	Defines a header cell in a table
<tr>	Defines a row in a table
<td>	Defines a cell in a table
<caption>	Defines a table caption
<colgroup>	Specifies a group of one or more columns in a table for formatting

<col>	Specifies column properties for each column within a <colgroup> element
<thead>	Groups the header content in a table
<tbody>	Groups the body content in a table
<tfoot>	Groups the footer content in a table

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible"content="IE=edge">
    <meta name="viewport"content="width=device-width, initial-scale=1.0">
    <title>Document</title>
<style>
    table{
        border-collapse: collapse;
        width: 100%;
    }
th,td{
    border: 2pxsolidgreen;
    padding: 15px;
    text-align: center;

}
</style>
</head>
<body>
<table>
    <tr>
     <th>name</th>
     <th>city</th>
     <th>roll</th>
    </tr>
    <tr>
        <td>anjali</td>
        <td>pune</td>
        <td>1</td>
    </tr>
    <tr>
        <td>sonali</td>
        <td>mumbai</td>
        <td>2</td>
    </tr>
</table>
</body>
</html>
```

1. The value of the rowspan attribute represents the number of rows to span.

If suppose I apply rowspan=2 then that td take the place of 2 rows

2. To make a cell span over multiple columns, use the colspan attribute:

It means if I apply colspan=2 then td take the place of 2 column

```
<!DOCTYPE>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible"content="IE=edge">
    <meta name="viewport"content="width=device-width, initial-scale=1.0">
    <title>Document</title>
<style>
    table{
        border-collapse: collapse;
        width: 100%;
      }
th,td{
    border: 2px solid green;
    padding: 15px;
    text-align: center;

}
</style>
</head>
<body>
<table>
    <tr>
     <th colspan="2">name</th>
     <th>city</th>
     <th rowspan="3">roll</th>
    </tr>
    <tr>
        <td>anjali</td>
        <td colspan="2">pune</td>

    </tr>
    <tr>
```

```
        <td>sonali</td>
        <td colspan="2">mumbai</td>

    </tr>
</table>
</body>
</html>
```

3)

```
<table>
  <thead>
    <tr>
      <th>Month</th>
      <th>Savings</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>January</td>
      <td>2000</td>
    </tr>
    <tr>
      <td>February</td>
      <td>1000</td>
    </tr>
  </tbody>
  <tfoot>
    <tr>
      <td>May</td>
      <td>3000</td>
    </tr>
  </tfoot>
</table>
```

HTML LIST:

There are 3 type of list

1.order list

Syntax:

```
<ol>
  <li>Cof</li>
  <li>Tea</li>
  <li>Mitta</li>
</ol>
```

2.unorder list

Syntax:

```
<ul>
  <li>Coee</li>
  <li>Table</li>
  <li>Meena</li>
</ul>
```

3.discription list

```
<dl>
  <dt>Coffee</dt>
  <dd>-             black             hot             drink</dd>
  <dt>Milk</dt>
  <dd>- white cold drink</dd>

</dl>
```

HTML iframes

An HTML iframe is used to display a web page within a web page.

HTML <iframe> tag defines an inline frame, hence it is also called as an Inline frame.

```
<!DOCTYPE html>
<html>
<body>
<h2>` Iframes</h2>
<p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Harum
debitis dolorum quis?</p>
<iframe src="https://www.facebook.com/" height="50%"
width="70%"></iframe>
</body>
</html>
```

Background Image on a HTML element

```
<!DOCTYPE html>
<html>
<head>
<style>
```

```
body {
  background-image: url('girl1.jpg');
  background-repeat: no-repeat;
  background-attachment: fixed;
  background-size: cover;
}
</style>
</head>
<body>
<h2>Background Image</h2>
</body>
</html>
```

HTML Semantic Elements

<article>	**Defines independent, self-contained content**
<aside>	**Defines content aside from the page content**
<details>	**Defines additional details that the user can view or hide**

<figcaption>	**Defines a caption for a <figure> element**
<figure>	**Specifies self-contained content, like illustrations, diagrams, photos, code**
<footer>	**Defines a footer for a document or section**
<header>	**Specifies a header for a document or section**
<main>	**Specifies the main content of a document**
<mark>	**Defines marked/highlighted text**
<nav>	**Defines navigation links**
<section>	**Defines a section in a document**
<summary>	**Defines a visible heading for a <details> element**
<time>	**Defines a date/time**

```
<!DOCTYPE html>
  <html lang="en">
  <head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible"content="IE=edge">
   <meta name="viewport"content="width=device-width, initial-scale=1.0">
   <title>Document</title>
  </head>
```

```
   <body>
    <article>
        <header>
          <h1>where is taj</h1>
          <p>how we can go</p>
        </header>
        <nav>
        <a href="">home</a>
        <a href="">about</a>
        <a href="">services</a>
        <a href="">contact</a>
        </nav>
        <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Modi
nam earum cumque?</p>
        </article>

 <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Tenetur
perspiciatis quidem provident quod atque odit ducimus est natus, sint iusto
magnam unde aliquam?</p>

<aside>
```

```
<p>Lorem, ipsum dolor sit amet consectetur adipisicing elit. Omnis error
earum veritatis. Delectus ab quia accusamus in minus maxime recusandae
quidem est.</p>
</aside>
      <footer>
        <h4>lorem12</h4>
      </footer>
   </body>
   </html>
```

Html Forms:

An HTML form is used to collect user input. The user input is sent to a server for processing.

For example: If a user want to purchase some items on internet, must fill the form such as name,email,shipping address and credit/debit card details so that item can be sent to the given address.

text	Defines a one-line text input field
password	Defines a one-line password input field
submit	Defines a submit button to submit the form to server
reset	Defines a reset button to reset all values in the form.

radio	Defines a radio button which allows select one option.
checkbox	Defines checkboxes which allow select multiple options form.
button	Defines a simple push button, which can be programmed to perform a task on an event.
file	Defines to select the file from device storage.
image	Defines a graphical submit button.
date	Defines an input field for selection of date.
datetime-local	Defines an input field for entering a date without time zone.
email	Defines an input field for entering an email address.
month	Defines a control with month and year, without time zone.
number	Defines an input field to enter a number.
url	Defines a field for entering URL
week	Defines a field to enter the date with week-year, without time zone.
search	Defines a single line text field for entering a search string.
tel	Defines an input field for entering the telephone number.
textarea	define for number of lines

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

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible"content="IE=edge">
  <meta name="viewport"content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <form action="HTML/fourt.html">
    <fieldset>
      <legend>User Information</legend>
    <label for="name">Name</label>
    <input type="text"id="name"placeholder="name"><br><br>
    <label for="email">email</label>
    <input type="email"id="name"placeholder="email"><br><br>
    <h4>choose the gender</h4>
    <input type="radio"name="color"value="red">Red <br>
  <input type="radio"name="color"value="blue">blue <br>
  <h4>Kindly Select your favourite sports</h4>
  <input type="checkbox"name="cricket"value="cricket">cricket<br>
  <input type="checkbox"name="khokho"value="khokho">khokho<br>
  <input type="checkbox"name="hoky"value="hoky">cricket<br>
  <input type="checkbox"name="tennis"value="tenis">tenis<br><br>
  <span>select date and time:</span><input type="datetime-local"><br><br>
  enter phone number:<input type="tel"><br><br>
  upload file:<input type="file"id=""><br><br>
  upload link:<input type="url"id=""><br><br>
  Address:<textarea
rows="2"cols="20"name="address"></textarea><br><br>
    <input type="button"value="submit">
    </fieldset>
  </form>
</body>
</html>
```

HTML Canvas Graphics

The <canvas> element is only a container for graphics. You must use JavaScript to actually draw the graphics.

Canvas has several methods for drawing paths, boxes, circles, text, and adding images.

The **HTML canvas element** provides HTML a bitmapped surface . It is used to draw graphics on the web page.

The **HTML 5 <canvas> tag** is used to draw graphics using scripting language like JavaScript

```
<!DOCTYPE>
<html>
<body>
<canvas id="canvas1" width="200" height="100" style="border:3px solid;">
Welcome to pune
</canvas>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>
<canvas id="Canvas1" width="200" height="100" style="border:1px solid red">
Your browser does not support the HTML canvas tag.</canvas>
<script>
```

```
var c = document.getElementById("Canvas1");
var x = c.getContext("2d");
x.moveTo(0,0);
x.lineTo(200,100);
x.stroke();
</script>
</body>
</html>
```

HTML SVG Graphics

SVG stands for Scalable Vector Graphics

The HTML <svg> element is a container for SVG graphics.

SVG has several methods for drawing boxes, text, circles, text, and graphic images.

SVG is mostly used for vector type diagrams like pie charts, 2-Dimensional graphs in an X,Y coordinate system etc.

```
<!DOCTYPE html>

<html>
```

```
<body>

  <svg width="200" height="200">

   <circle cx="100" cy="100" r="80" stroke="yellow" stroke-width="4"
fill="red"/>

</svg>

</body>

</html>

<!DOCTYPE html>

<html>

<body>

<svg width="200" height="100">

  <rect  width="200"  height="100"  stroke="yellow"  stroke-width="4"
fill="red"/>

</svg>

</body>

</html>
```

HTML Multimedia

HTML audio tag is used to define sounds such as music and other audio clips. Currently there are three supported file format for HTML 5 audio tag.

1. mp3
2. wav
3. Ogg

mp3	audio/mpeg
ogg	audio/ogg
wav	audio/wav

HTML5 supports <video> and <audio> controls

```
<!DOCTYPE>
<html>
<body>
<audio controls>
  <source src="cat.mp3" type="audio/mpeg">
    Sorry not supported
```

```
</audio>
</body>
</html>
```

HTML Video Tag

The HTML video tag is used for streaming video files such as a movie clip, song clip on the web page.

This 3 format support the html video

1. mp4
2. webM
3. Ogg

controls	defines the video controls which is displayed with play/pause buttons.
height	It is used to set the height of the video player.
width	It is used to set the width of the video player.
poster	It specifies the image which is displayed on the screen when the video is not played.
autoplay	It specifies that the video will start playing as soon as it is ready.

loop	It specifies that the video file will start over again, every time when it is completed.
muted	It is used to mute the video output.
preload	It specifies the author view to upload video file when the page loads.
src	It specifies the source URL of the video file.

```
<!DOCTYPE>
<html>
<body>
<video width="300" height="200" controls autoplay loop>
  <source src="kids.mp4" type="video/mp4">
   does not support the html video tag.
</video>
</body>
</html>
```

INTRODUCTION OF CSS

Introduction to CSS:

This first CSS article is designed to get your 'feet on the ground'. You should know at least a little about HTML and web design before you begin. Once you finish reading this page, you will be ready to jump into the tutorial!

My Popular CSS Video Course

CSS3 works hand-in-hand with HTML5 to create the beautiful websites that we see today. My front end web developer course teaches you everything you need to know to build amazing websites that easily cost $2000-$3000 to build … and for only $29!

An introduction to Cascading Style Sheets

CSS is the acronym for: 'Cascading Style Sheets'. CSS is an extension to basic HTML that allows you to style your web pages.

An example of a style change would be to make words bold. In standard HTML you would use the <b> tag like so:

```
<b>make me bold</b>
```

This works fine, and there is nothing wrong with it per se, except that now if you wanted to say change all your text that you initially made bold to underlined, you would have to go to every spot in the page and change the tag.

Another disadvantage can be found in this example: say you wanted to make the above text bold, make the font style Verdana and change its color to red, you would need a lot of code wrapped around the text:

```
<font color="#FF0000" face="Verdana, Arial,  Helvetica, sans-serif">
   <strong>This is  text</strong></font>
```

This is verbose and contributes to making your HTML messy. With CSS, you can create a custom style elsewhere and set all its properties, give it a unique name and then 'tag' your HTML to apply these stylistic properties:

```
<p class="myNewStyle">My CSS styled text</p>
```

And in between the tags at the top of your web page you would insert this CSS code that defines the style we just applied:

```
<style type="text/css">
.myNewStyle {
   font-family: Verdana, Arial, Helvetica, sans-serif;
   font-weight: bold;
   color: #FF0000;
}
</style>
```

In the above example we **embed the css code** directly into the page itself. This is fine for smaller projects or in situations where the styles you're defining will only be used in a single page. There are many times when you will be applying your styles to many pages and it would be a hassle to have to copy and paste your CSS code into each page.

Besides the fact that you will be cluttering up your pages with the same CSS code, you also find yourself having to edit each of these pages if you want to make a style change. Like with JavaScript, you can define/create your CSS styles in a separate file and then link it to the page you want to apply the code to:

```
<link href="myFirstStyleSheet.css" rel="stylesheet"  type="text/css">
```

The above line of code links your external style sheet called ‘myFirstStyleSheet.css’ to the HTML document. You place this code in between the <head> </head> tags in your web page.

How to create a linked external stylesheet

To create an external style sheet all you need to do is create a simple text document (on windows you simply right-click and select new -> text document) and then change the file from type .txt to .css.

You can change the file type by just changing the file’s extension. The file’s extension on windows tells the computer what kind of file it is and allows the computer to determine how to handle the file when for example you try to open it.

You probably guessed it; CSS files are just specially formatted text files, and much in the same way HTML pages are. There is nothing special or different in the file itself, rather it is the contents of the file that make an HTML document and a CSS page what they are.

When working with a external CSS document, there are a couple of points to remember:

1. You don’t add these tags in the CSS page itself as you would if you embedded the CSS code in your HTML:

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

Since the CSS link in your web page says that you are linking to a CSS page, you don’t need to declare (in the external CSS file) that the code in the CSS page is CSS. That is what the above tags do. Instead you would just add your CSS code directly to the page like so:

```
.myNewStyle {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-weight: bold;
  color: #FF0000;
}

.my2ndNewStyle {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-weight: bold;
  color: #FF0000;
}

.my3rdNewStyle {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-weight: bold;
  font-size: 12pt;
  color: #FF0000;
}
```

In the above example I have created a series CSS classes that can be applied to any HTML tag like so:

```
<p>My CSS styled text</p>
```

or

```
<h2 class=”my3rdNewStyle”>My CSS styled  text</h2>
```

You will notice that in the above example I applied a CSS style to a <h2> tag. Normally this tag sets the size of the text that it wraps to a size that is preset in the browser (ex: 10 pixels).

When you apply a CSS class to it, the CSS code overrides the default size that you would normally get with an <h2> tag in favor of the size specified in the CSS class. So now you can see that CSS can override default HTML tag behavior!

In the above examples, I have CSS code where I define my CSS classes and then ‘apply’ them to various elements in the page. Another way to apply CSS is to globally redefine an HTML tag to look a certain way:

```
h1 { font-family: Garamond, "Times New Roman",  serif; font-size: 200%; }
```

What this CSS code does is set the font style and size of all <h1> tags in one shot. Now you don’t have to apply a CSS class as we did before to any <h1> tags since they are automatically all affected by the CSS style rules.

Here is another example of where I give the whole page bigger margins:

```
body { margin-left: 15%; margin-right: 15%; }
```

As you can see, you can redefine any tag and change the way it looks! This can be very powerful:

```
div {
   background: rgb(204,204,255);
   padding: 0.5em;
   border: 1px solid #000000;
}
```

The above CSS code sets that any <div></div> tag will now have a background color of ‘rgb(204,204,255)’ and have a padding of 0.5em and a thin 1 pixel border that is solid black.

A few things to explain about the above:

Color in CSS can be expressed in a few ways:

1. In Hex -> for example: #000000 – this is black and this: #FF0000 is red.
2. In rgb -> rgb(204,204,255) is a light purple blue color.
3. With named colors like: ‘red’ or ‘blue’

I typically use hex color since I am familiar with them or I just use named colors. So the last example can be rewritten like so:

```
div {
   background: green;
   padding: 0.5em;
   border: 1px solid #FF0000;
}
```

So instead of ‘rgb(204,204,255)’ , I just specified ‘green’.

By using RGB (RGB is the acronym for: ‘Red Green Blue’) and Hex color, you can really get the exact color you want easily when you know your codes. Luckily many programs (like Dreamweaver) provide easy to use color pickers for you so you don’t need to know the values for the code.

In this last example I will show you the ‘super cool’ CSS code that allows you to create link roll-over affects without images:

```
a:link { color: rgb(0, 0, 153) }
a:visited { color: rgb(153, 0, 153) }
a:hover { color: rgb(0, 96, 255) }
a:active { color: rgb(255, 0, 102) }
```

The above CSS will cause your links to change color when someone hovers their mouse pointer over it, instant rollovers with no images! One important note with the above code, is that it is important that the style declarations be in the right order:

“link-visited-hover-active”,

… otherwise it may break in some browsers.

CSS is very powerful and allows you to do things that you can’t do with standard HTML. It is supported nicely now in all the modern browsers and is a must learn tool for web designers.

The above examples are just a small sample of what you can do with CSS, but it should be more than enough for you to start styling your pages nicely.

Like with many technologies CSS has a lot of capability that most people will not need to use often, if at all. So don’t get caught in the trap of thinking that if there is some functionality/feature available that you have to use it.

BTW: The killersites forum has a search engine that allows you to do some pretty powerful searches against what was discussed.

For example if you type ‘CSS’ in the search box, you will get a list of all the post regarding CSS. This feature will makes the forum archive a great source of information

CSS selectors

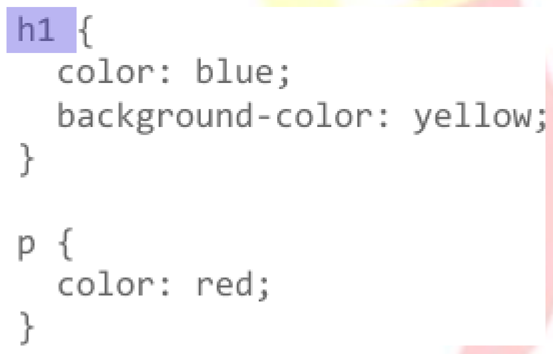
In CSS, selectors are used to target the HTML elements on our web pages that we want to style. There are a wide variety of CSS selectors available, allowing for fine-grained precision when selecting elements to style. In this article and its sub-articles we'll run through the different types in great detail, seeing how they work.

Prerequisites:	Basic computer literacy, basic software installed, basic knowledge of working with files, HTML basics (study Introduction to HTML), and an idea of how CSS works (study CSS first steps.)

Objective: To learn how CSS selectors work in detail.

What is a selector?

A CSS selector is the first part of a CSS Rule. It is a pattern of elements and other terms that tell the browser which HTML elements should be selected to have the CSS property values inside the rule applied to them. The element or elements which are selected by the selector are referred to as the *subject of the selector*.


```
h1 {
  color: blue;
  background-color: yellow;
}

p {
  color: red;
}
```

In other articles you may have met some different selectors, and learned that there are selectors that target the document in different ways — for example by selecting an element such as h1, or a class such as .special.

In CSS, selectors are defined in the CSS Selectors specification; like any other part of CSS they need to have support in browsers for them to work. The majority of selectors that you will come across are defined in the Level 3 Selectors specification and Level 4 Selectors specification, which are both mature specifications, therefore you will find excellent browser support for these selectors.

Selector lists

If you have more than one thing which uses the same CSS then the individual selectors can be combined into a *selector list* so that the rule is applied to all of the individual selectors. For example, if I have the same CSS for an h1 and also a class of .special, I could write this as two separate rules.

```
h1 {
  color: blue;
}


.special {
  color: blue;
}
```

Copy to Clipboard

I could also combine these into a selector list, by adding a comma between them.

```
h1, .special {
  color: blue;
}
```

Copy to Clipboard

White space is valid before or after the comma. You may also find the selectors more readable if each is on a new line.

```
h1,
.special {
  color: blue;
}
```

Copy to Clipboard

In the live example below try combining the two selectors which have identical declarations. The visual display should be the same after combining them.

When you group selectors in this way, if any selector is syntactically invalid, the whole rule will be ignored.

In the following example, the invalid class selector rule will be ignored, whereas the h1 would still be styled.

```
h1 {
  color: blue;
}

..special {
  color: blue;
}
```

Copy to Clipboard

When combined however, neither the h1 nor the class will be styled as the entire rule is deemed invalid.

```
h1, ..special {
  color: blue;
}
```

Copy to Clipboard

Types of selectors

There are a few different groupings of selectors, and knowing which type of selector you might need will help you to find the right tool for the job. In this article's subarticles we will look at the different groups of selectors in more detail.

Type, class, and ID selectors

This group includes selectors that target an HTML element such as an <h1>.

```
h1 {
}
```

Copy to Clipboard

It also includes selectors which target a class:

```
.box {
}
```

Copy to Clipboard

or, an ID:

```
#unique {
}
```

Copy to Clipboard

Attribute selectors

This group of selectors gives you different ways to select elements based on the presence of a certain attribute on an element:

```
a[title] {
}
```

Copy to Clipboard

Or even make a selection based on the presence of an attribute with a particular value:

```
a[href="https://example.com"]
{
}
```

Copy to Clipboard

Pseudo-classes and pseudo-elements

This group of selectors includes pseudo-classes, which style certain states of an element. The :hover pseudo-class for example selects an element only when it is being hovered over by the mouse pointer:

```
a:hover {
}
```

Copy to Clipboard

It also includes pseudo-elements, which select a certain part of an element rather than the element itself. For example, ::first-line always selects the first line of text inside an element (a <p> in the below case), acting as if a <span> was wrapped around the first formatted line and then selected.

```
p::first-line {
}
```

Copy to Clipboard

Combinators

The final group of selectors combine other selectors in order to target elements within our documents. The following, for example, selects paragraphs that are direct children of <article> elements using the child combinator (>):

article > p {

}

Border s in CSS

The **border** shorthand CSS property sets an element's border. It sets the values of border-width, border-style, and border-color.

Try it

Constituent properties

This property is a shorthand for the following CSS properties:

- border-color
- border-style
- border-width

Syntax

```
/* style */
border: solid;

/* width | style */
border: 2px dotted;
```

```
/* style | color */
border: outset #f33;

/* width | style | color */
border: medium dashed green;

/* Global values */
border: inherit;
border: initial;
border: revert;
border: revert-layer;
border: unset;
```

Copy to Clipboard

The border property may be specified using one, two, or three of the values listed below. The order of the values does not matter.

Note: The border will be invisible if its style is not defined. This is because the style defaults to none.

Values

<line-width>

Sets the thickness of the border. Defaults to medium if absent. See border-width.

<line-style>

Sets the style of the border. Defaults to none if absent. See border-style.

<color>

Sets the color of the border. Defaults to currentcolor if absent. See border-color.

Description

As with all shorthand properties, any omitted sub-values will be set to their initial value. Importantly, border cannot be used to specify a custom value for border-image, but instead sets it to its initial value, i.e., none.

The border shorthand is especially useful when you want all four borders to be the same. To make them different from each other, however, you can use the longhand border-width, border-style, and border-color properties, which accept different values for each side. Alternatively, you can target one border at a time with the physical (e.g., border-top) and logical (e.g., border-block-start) border properties.

Borders vs. outlines

Borders and outlines are very similar. However, outlines differ from borders in the following ways:

- Outlines never take up space, as they are drawn outside of an element's content.
- According to the spec, outlines don't have to be rectangular, although they usually are.

Formal definition

as each of the properties of the shorthand:

Initial value

- border-width: as each of the properties of the shorthand:
  - border-top-width: medium
  - border-right-width: medium
  - border-bottom-width: medium
  - border-left-width: medium
- border-style: as each of the properties of the shorthand:
  - border-top-style: none
  - border-right-style: none
  - border-bottom-style: none

	◦ border-left-style: none • border-color: as each of the properties of the shorthand: ◦ border-top-color: currentcolor ◦ border-right-color: currentcolor ◦ border-bottom-color: currentcolor ◦ border-left-color: currentcolor
Applies to	all elements. It also applies to ::first-letter.
Inherited	no
Computed value	as each of the properties of the shorthand: • border-width: as each of the properties of the shorthand: ◦ border-bottom-width: the absolute length or 0 if border-bottom-style is none or hidden ◦ border-left-width: the absolute length or 0 if border-left-style is none or hidden ◦ border-right-width: the absolute length or 0 if border-right-style is none or hidden ◦ border-top-width: the absolute length or 0 if border-top-style is none or hidden

- border-style: as each of the properties of the shorthand:
  - border-bottom-style: as specified
  - border-left-style: as specified
  - border-right-style: as specified
  - border-top-style: as specified
- border-color: as each of the properties of the shorthand:
  - border-bottom-color: computed color
  - border-left-color: computed color
  - border-right-color: computed color
  - border-top-color: computed color

as each of the properties of the shorthand:

border-color: as each of the properties of the shorthand:

Animation type

-
  - border-bottom-color: a color
  - border-left-color: a color
  - border-right-color: a color
  - border-top-color: a color
- border-style: discrete

- border-width: as each of the properties of the shorthand:
  - border-bottom-width: a length
  - border-left-width: a length
  - border-right-width: a length
  - border-top-width: a length

Formal syntax

```
border =
  <line-width>  ||
  <line-style>  ||
  <color>

<line-width> =
  <length [0,∞]>  |
  thin  |
  medium  |
  thick

<line-style> =
  none  |
  hidden  |
  dotted  |
  dashed
  solid
  double
  groove
  ridge
  inset
  outset
```

Examples

Setting a pink outset border

HTML

```
<div>I have a border, an outline, and a box shadow! Amazing, isn't it?</div>
```

Copy to Clipboard

CSS

```
div {
  border: 0.5rem outset pink;
  outline: 0.5rem solid khaki;
  box-shadow: 0 0 0 2rem skyblue;
  border-radius: 12px;
  font: bold 1rem sans-serif;
  margin: 2rem;
  padding: 1rem;
  outline-offset: 0.5rem;
}
```

Styling backgrounds in CSS

The CSS background property is a shorthand for a number of background longhand properties that we will meet in this lesson. If you discover a complex background property in a stylesheet, it might seem a little hard to understand as so many values can be passed in at once.

```
.box {
  background: linear-gradient(
      105deg,
      rgba(255, 255, 255, 0.2) 39%,
      rgba(51, 56, 57, 1) 96%
    ) center center / 400px 200px no-repeat, url(big-star.png) center
    no-repeat, rebeccapurple;
}
```

Copy to Clipboard

We'll return to how the shorthand works later in the tutorial, but first let's have a look at the different things you can do with backgrounds in CSS, by looking at the individual background properties.

Background colors

The background-color property defines the background color on any element in CSS. The property accepts any valid <color>. A background-color extends underneath the content and padding box of the element.

In the example below, we have used various color values to add a background color to the box, a heading, and a <span> element.

Play around with these, using any available <color> value.

Background images

The background-image property enables the display of an image in the background of an element. In the example below, we have two boxes — one has a background image which is larger than the box (balloons.jpg), the other has a small image of a single star (star.png).

This example demonstrates two things about background images. By default, the large image is not scaled down to fit the box, so we only see a small corner of it, whereas the small image is tiled to fill the box.

If you specify a background color in addition to a background image then the image displays on top of the color. Try adding a background-color property to the example above to see that in action.

Controlling background-repeat

The background-repeat property is used to control the tiling behavior of images. The available values are:

- no-repeat — stop the background from repeating altogether.
- repeat-x — repeat horizontally.
- repeat-y — repeat vertically.
- repeat — the default; repeat in both directions.

Try these values out in the example below. We have set the value to no-repeat so you will only see one star. Try out the different values — repeat-x and repeat-y — to see what their effects are.

Sizing the background image

The *balloons.jpg* image used in the initial background images example, is a large image that was cropped due to being larger than the element it is a background of. In this case we could use the background-size property, which can take length or percentage values, to size the image to fit inside the background.

You can also use keywords:

- cover — the browser will make the image just large enough so that it completely covers the box area while still retaining its aspect ratio. In this case, part of the image is likely to end up outside the box.
- contain — the browser will make the image the right size to fit inside the box. In this case, you may end up with gaps on either side or on the top and bottom of the image, if the aspect ratio of the image is different from that of the box.

In the example below I have used the *balloons.jpg* image along with length units to size it inside the box. You can see this has distorted the image.

Try the following.

- Change the length units used to modify the size of the background.
- Remove the length units and see what happens when you use background-size: cover or background-size: contain.
- If your image is smaller than the box, you can change the value of background-repeat to repeat the image.

Positioning the background image

The background-position property allows you to choose the position in which the background image appears on the box it is applied to. This uses a coordinate system in which the top-left-hand corner of the box is (0,0), and the box is positioned along the horizontal (x) and vertical (y) axes.

Note: The default background-position value is (0,0).

The most common background-position values take two individual values — a horizontal value followed by a vertical value.

You can use keywords such as top and right (look up the others on the background-position page):

```
.box {
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: top center;
}
```

Copy to Clipboard

And Lengths, and percentages:

```
.box {
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: 20px 10%;
}
```

You can also mix keyword values with lengths or percentages, in which case the first value must refer to the horizontal position or offset and the second vertical. For example:

```
.box {
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: 20px top;
}
```

Finally, you can also use a 4-value syntax in order to indicate a distance from certain edges of the box — the length unit, in this case, is an offset from the value that precedes it. So in the CSS below we are positioning the background 20px from the top and 10px from the right:

```
.box {
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: top 20px right 10px;
}
```

Copy to Clipboard

Use the example below to play around with these values and move the star around inside the box.

Note: background-position is a shorthand for background-position-x and background-position-y, which allow you to set the different axis position values individually.

Gradient backgrounds

A gradient — when used for a background — acts just like an image and is also set by using the background-image property.

You can read more about the different types of gradients and things you can do with them on the MDN page for the <gradient> data type. A fun way to play with gradients is to use one of the many CSS Gradient Generators available on the web, such as this one. You can create a gradient then copy and paste out the source code that generates it.

Try some different gradients in the example below. In the two boxes respectively, we have a linear gradient that is stretched over the whole box, and a radial gradient with a set size, which therefore repeats.

Multiple background images

It is also possible to have multiple background images — you specify multiple background-image values in a single property value, separating each one with a comma.

When you do this you may end up with background images overlapping each other. The backgrounds will layer with the last listed background image at the bottom of the stack, and each previous image stacking on top of the one that follows it in the code.

Note: Gradients can be happily mixed with regular background images.

The other background-* properties can also have comma-separated values in the same way as background-image:

```
background-image: url(image1.png), url(image2.png), url(image3.png),
  url(image4.png);
background-repeat: no-repeat, repeat-x, repeat;
background-position: 10px 20px, top right;
```
Copy to Clipboard

Each value of the different properties will match up to the values in the same position in the other properties. Above, for example, image1's background-repeat value will be no-repeat. However, what happens when different properties have different numbers of values? The answer is that the smaller numbers of values will cycle — in the above example there are four background images but only two background-position values. The first two position values will be applied to the first two images, then they will cycle back around again — image3 will be given the first position value, and image4 will be given the second position value.

Let's play. In the example below I have included two images. To demonstrate the stacking order, try switching which background image comes first in the list. Or play with the other properties to change the position, size, or repeat values.

Background attachment

Another option we have available for backgrounds is specifying how they scroll when the content scrolls. This is controlled using the background-attachment property, which can take the following values:

- scroll: causes the element's background to scroll when the page is scrolled. If the element content is scrolled, the background does not move. In effect, the background is fixed to the same position on the page, so it scrolls as the page scrolls.
- fixed: causes an element's background to be fixed to the viewport so that it doesn't scroll when the page or element content is scrolled. It will always remain in the same position on the screen.
- local: fixes the background to the element it is set on, so when you scroll the element, the background scrolls with it.

The background-attachment property only has an effect when there is content to scroll, so we've made a demo to demonstrate the differences between the three values — have a look at background-attachment.html (also see the source code here).

Using the background shorthand property

As I mentioned at the beginning of this lesson, you will often see backgrounds specified using the background property. This shorthand lets you set all of the different properties at once.

If using multiple backgrounds, you need to specify all of the properties for the first background, then add your next background after a comma. In the example below we have a gradient with a size and position, then an image background with no-repeat and a position, then a color.

There are a few rules that need to be followed when writing background image shorthand values, for example:

- A background-color may only be specified after the final comma.
- The value of background-size may only be included immediately after background-position, separated with the '/' character, like this: center/80%.

Accessibility considerations with backgrounds

When placing text on top of a background image or color, you should take care that you have enough contrast for the text to be legible for your visitors. If specifying an image, and if text will be placed on top of that image, you should also specify a background-color that will allow the text to be legible if the image does not load.

Screen readers cannot parse background images; therefore, they should be purely decoration. Any important content should be part of the HTML page and not contained in a background.

Borders

When learning about the Box Model, we discovered how borders affect the size of our box. In this lesson we will look at how to use borders creatively. Typically when we add borders to an element with CSS we use a shorthand property that sets the color, width, and style of the border in one line of CSS.

We can set a border for all four sides of a box with border:

```
.box {
  border: 1px solid black;
}
```

Copy to Clipboard

Or we can target one edge of the box, for example:

```
.box {
  border-top: 1px solid black;
}
```

Copy to Clipboard

The individual properties for these shorthands would be:

```
.box {
  border-width:   1px;
  border-style:   solid;
  border-color: black;
}
```

Copy to Clipboard

And for the longhands:

```
.box {
  border-top-width:   1px;
  border-top-style:   solid;
  border-top-color: black;
}
```

Copy to Clipboard

Note: These top, right, bottom, and left border properties also have mapped *logical* properties that relate to the writing mode of the document (e.g. left-to-right or right-to-left text, or top-to-bottom). We'll be exploring these in the next lesson, which covers handling different text directions.

There are a variety of styles that you can use for borders. In the example below we have used a different border style for the four sides of my box. Play with the border style, width, and color to see how borders work.

Rounded corners

Rounding corners on a box is achieved by using the border-radius property and associated longhands which relate to each corner of the box. Two lengths or percentages may be used as a value, the first value defining the horizontal radius, and the second the vertical radius. In a lot of cases, you will only pass in one value, which will be used for both.

For example, to make all four corners of a box have a 10px radius:

```
.box {
  border-radius: 10px;
}
```

Copy to Clipboard

Or to make the top right corner have a horizontal radius of 1em, and a vertical radius of 10%:

```
.box {
  border-top-right-radius: 1em 10%;
}
```

Copy to Clipboard

We have set all four corners in the example below and then changed the values for the top right corner to make it different. You can play with the values to change the corners. Take a look at the property page for border-radius to see the available syntax options.

CSS Text Effects

We can apply different effects on the text used within an HTML document. Some properties can be used for adding the effects on text.

Using CSS, we can style the web documents and affects the text. The properties of the text effect help us to make the text attractive and clear. There are some text effect properties in CSS that are listed below:

- word-break
- text-overflow
- word-wrap
- writing-mode

Let's discuss the above CSS properties along with illustrations.

word-break

It specifies how words should break at the end of the line. It defines the line break rules.

Syntax

```
word-break: normal |keep-all |  break-all | inherit ;
```

The default value of this property is normal. So, this value is automatically used when we don't specify any value.

Values

keep-all: It breaks the word in the default style.

break-all: It inserts the word break between the characters in order to prevent the word overflow.

Example

```
<!DOCTYPE html>
<html>
   <head>
      <title>word-break: break-all</title>
      <style>
         .jtp{
```

```
        width: 150px;
        border: 2px solid black;
        word-break: break-all;
            text-align: left;
            font-size: 25px;
    color: blue;
          }
          .jtp1{
            width: 156px;
            border: 2px solid black;
            word-break: keep-all;
            text-align: left;
            font-size: 25px;
    color: blue;
          }
      </style>
    </head>
     <center>
    <body>
      <h2>word-break: break-all;</h2>

      <p class="jtp">
          Welcome to the yess infotech.com
```

```
            </p>
            <h2>word-break: keep-all;</h2>
            <p class="jtp1">
            Welcome to the YS.com
            </p>
    </center>
        </body>
    </html>
```

word-wrap

CSS word-wrap property is used to break the long words and wrap onto the next line. This property is used to prevent overflow when an unbreakable string is too long to fit in the containing box.

Syntax

```
word-wrap: normal| break-word| inherit ;
```

Values

normal: This property is used to break words only at allowed break points.

break-word: It is used to break unbreakable words.

initial: It is used to set this property to its default value.

inherit: It inherits this property from its parent element.

Example

```
<!DOCTYPE html>
<html>
<head>
<style>
.test {
    width: 200px;
    background-color: lightblue;
    border: 2px solid black;
    padding:10px;
            font-size: 20px;

       }
       .test1 {
            width: 11em;
            background-color: lightblue;
            border: 2px solid black;
            padding:10px;
            word-wrap: break-word;
            font-size: 20px;
       }
       </style>
       </head>
```

```
<body>
<center>
<h1> Without Using word-wrap </h1>
<p class="test"> In this paragraph, there is a very long word:
 iamsooooooooooooooooooooooooooooooolonggggggggggggggg
. </p>
<h1> Using word-wrap: break-word; </h1>
<p class="test1"> In this paragraph, there is a very long word:
 iamsooooooooooooooooooooooooooooooolonggggggggggggggg
. The long word will break and wrap to the next line. </p>
 </center>
</body>
</html>
```

text-overflow

It specifies the representation of overflowed text, which is not visible to the user. It signals the user about the content that is not visible. This property helps us to decide whether the text should be clipped or show some dots (ellipsis).

This property does not work on its own. We have to use **white-space: nowrap;** and **overflow: hidden;** with this property.

Syntax

```
text-overflow: clip | ellipsis;
```

Property Values

clip: It is the default value that clips the overflowed text.

ellipsis: This value displays an ellipsis (…) or three dots to show the clipped text. It is displayed within the area, decreasing the amount of text.

Example

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .jtp{
        white-space: nowrap;
    height: 30px;
        width: 250px;
        overflow: hidden;
             border: 2px solid black;
             font-size: 25px;
             text-overflow: clip;
           }
       .jtp1 {
             white-space: nowrap;
   height: 30px;
             width: 250px;
             overflow: hidden;
             border: 2px solid black;
             font-size: 25px;
             text-overflow: ellipsis;
           }

   h2{
   color: blue;
   }
           div:hover {
             overflow: visible;
           }
   p{
   font-size: 25px;
   font-weight: bold;
   color: red;
   }
       </style>
   </head>
   <center>
```

```
    <body>
  <p> Hover over the bordered text to see the full content. </p>

      <h2>
        text-overflow: clip;
      </h2>

      <div class="jtp">
        Welcome to the yess infotech.com
      </div>
      <h2>
        text-overflow: ellipsis;
      </h2>

      <div class="jtp1">
        Welcome to the yess infotech.com
      </div>
      </center>
    </body>
  </html>
```

CSS text-align

This CSS property is used to set the horizontal alignment of a table-cell box or the block element. It is similar to the **vertical-align** property but in the horizontal direction.

Syntax

```
text-align: justify | center | right | left | initial | inherit;
```

Possible values

justify: It is generally used in newspapers and magazines. It stretches the element's content in order to display the equal width of every line.

center: It centers the inline text.

right: It is used to align the text to the right.

left: It is used to align the text to the left.

Let's see an example that will demonstrate the **text-align** property.

Example

```
<html>
  <head>
  </head>
<style>
h2{
```

```
color: blue;
}
</style>
  <body>
        <h1>Example of text-align proeprty</h1>

        <h2 style = "text-align: center;">
          text-align: center;
          </h2>

          <h2 style = "text-align: right;">
          text-align: right;
          </h2>

          <h2 style = "text-align: left;">
          text-align: left;
          </h2>
          <h2 style = "text-align: justify;">
            text-align: justify; To see its effect, it should be applied on large paragraph.
          </h2>

          </body>
```

```
28.     </html>
```

Navigation Bars

Having easy-to-use navigation is important for any web site.

With CSS you can transform boring HTML menus into good-looking navigation bars.

Navigation Bar = List of Links

A navigation bar needs standard HTML as a base.

In our examples we will build the navigation bar from a standard HTML list.

A navigation bar is basically a list of links, so using the <ul> and <li> elements makes perfect sense:

Example

```
<ul>
  <li><a href="default.asp">Home</a></li>
  <li><a href="news.asp">News</a></li>
  <li><a href="contact.asp">Contact</a></li>
  <li><a href="about.asp">About</a></li>
</ul>
```

Example

```
ul {
  list-style-type: none;
  margin: 0;
  padding: 0;
}
```

Example explained:

- list-style-type: none; - Removes the bullets. A navigation bar does not need list markers
- Set margin: 0; and padding: 0; to remove browser default settings

CSS Vertical Navigation Bar

Example

```
li a {
  display: block;
```

width: 60px;

}

Example explained:

- display: block; - Displaying the links as block elements makes the whole link area clickable (not just the text), and it allows us to specify the width (and padding, margin, height, etc. if you want)
- width: 60px; - Block elements take up the full width available by default. We want to specify a 60 pixels width

- ou can also set the width of <ul>, and remove the width of <a>, as they will take up the full width available when displayed as block elements. This will produce the same result as our previous example:

- Example

- ul {

```
list-style-type: none;
margin: 0;
padding: 0;
width: 60px;
}

li a {
```

```
display: block;
}
```

CSS Horizontal Navigation Bar

There are two ways to create a horizontal navigation bar. Using **inline** or **floating** list items.

Inline List Items

One way to build a horizontal navigation bar is to specify the <li> elements as inline, in addition to the "standard" code from the previous page:

Example

```
li {
    display: inline;
}
```

Example explained:

- display: inline; - By default, <li> elements are block elements. Here, we remove the line breaks before and after each list item, to display them on one line

Floating List Items

Another way of creating a horizontal navigation bar is to float the <li> elements, and specify a layout for the navigation links:

Example

```
li {
  float: left;
}

a {
  display: block;
  padding: 8px;
  background-color: #dddddd;
}
```

Example explained:

- float: left; - Use float to get block elements to float next to each other
- display: block; - Allows us to specify padding (and height, width, margins, etc. if you want)
- padding: 8px; - Specify some padding between each <a> element, to make them look good
- background-color: #dddddd; - Add a gray background-color to each <a> element

Tip: Add the background-color to <ul> instead of each <a> element if you want a full-width background color:

Horizontal Navigation Bar Examples

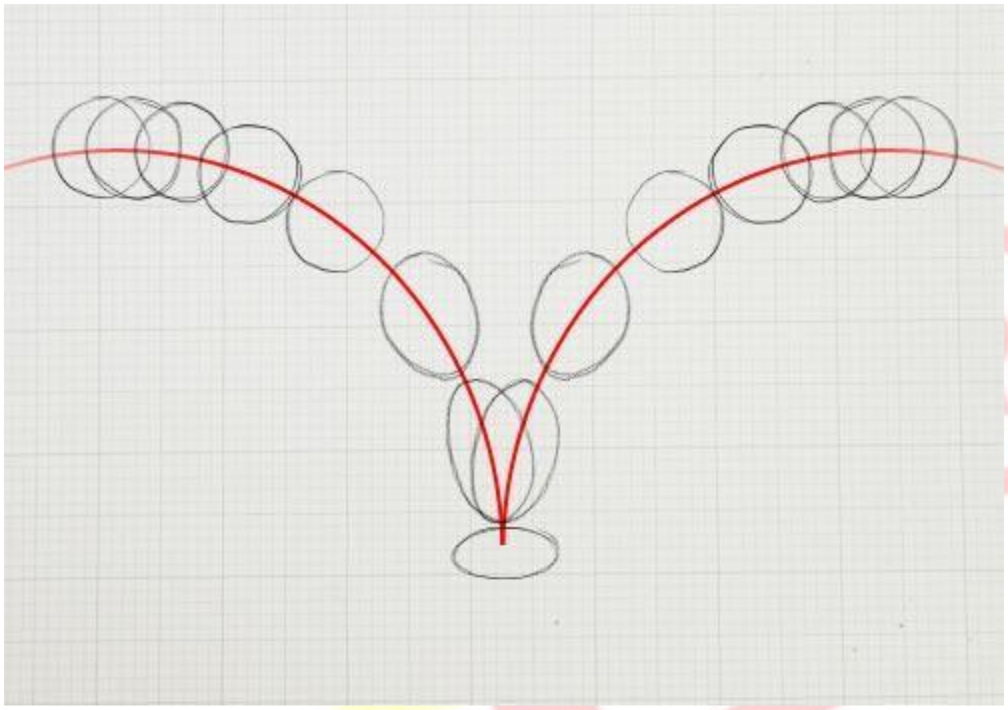
```
ul {
  list-style-type: none;
  margin: 0;
  padding: 0;
  overflow: hidden;
  background-color: #333;
}

li {
  float: left;
}

li a {
  display: block;
  color: white;
  text-align: center;
  padding: 14px 16px;
  text-decoration: none;
}
```

```
/* Change the link color to #111 (black) on hover */
li a:hover {
background-color: #111;
}
```

CSS Animation

With **CSS animation** now supported in both Firefox and Webkit browsers, there is no better time to give it a try. Regardless of its technical form, whether traditional, computer-generated 3-D, Flash or CSS, animation always follows the same basic principles.

In this article, we will take our first steps with CSS animation and consider the main guidelines for creating animation with CSS. We'll be working through an example, building up the animation using the principles of traditional animation. Finally, we'll see some real-world usages.



CSS Animation Properties #

Before diving into the details, let’s **set up the basic CSS**:

Animation is a new CSS property that allows for animation of most HTML elements (such as div, h1 and span) without JavaScript or Flash. At the moment, it’s supported in Webkit browsers, including Safari 4+, Safari for iOS (iOS 2+), Chrome 1+ and, more recently, Firefox 5. Unsupported browsers will simply ignore your animation code, so ensure that your page doesn’t rely on it!

Because the technology is still relatively new, prefixes for the browser vendors are required. So far, the syntax is exactly the same for each browser, with only a prefix change required. In the code examples below, we use the -webkit syntax.

All you need to get some CSS animation happening is to attach an animation to an element in the CSS:

```
/* This is the animation code. */
@-webkit-keyframes example {
   from { transform: scale(2.0); }
   to   { transform: scale(1.0); }
}


/* This is the element that we apply the animation to. */
div {
   -webkit-animation-name: example;
   -webkit-animation-duration: 1s;
   -webkit-animation-timing-function: ease; /* ease is the default */
   -webkit-animation-delay: 1s;             /* 0 is the default */
   -webkit-animation-iteration-count: 2;    /* 1 is the default */
   -webkit-animation-direction: alternate;  /* normal is the default */
}
```

When assigning the animation to your element, you can also use the shorthand:

```
div {
-webkit-animation: example 1s ease 1s 2 alternate;
}
```

What Are CSS Animations?

CSS Animation is the process of animating the objects (or elements) on a web page. Earlier to CSS Animations, it was done with the help of JavaScript and its libraries which as a developer, you would know that it unnecessarily complicated the matters. Animation in CSS bring pre-defined properties that are easier to apply and faster to establish the end-goals with multiple options packed inside it.

Moving the animation in CSS provides better performance than JavaScript as the browser takes control and optimizes the animation for the performance. The properties that come with the CSS animations can be divided into three major categories:

1. **Transformation** – Transforming the dimensions, rescaling the objects, moving them from point A to B, etc.
2. **Transitions** – Performing the transformations smoothly.
3. **Keyframes** – Changing the animation (property, value, etc.) at a given time or state.

Once we have implemented the above three properties, we are done with the animation in CSS. In this CSS Animations tutorial, we look at each one of them with relevant examples on CSS Animations.

But Primarily Are Your CSS Animations Responsive?

In this CSS Animations tutorial, we will learn how to create animation in CSS and incorporate them into websites and web pages. However, when we create animations, we want them to be compatible with a multitude of screen sizes like smartphones, tablets, desktops, laptops, etc. In order to build a responsive web design with animations, we must perform a responsiveness test using a responsive web checker tool.

Also Read – Responsive Web Design Testing Checklist: All You Need to Know

Frequently testing responsiveness with manual techniques can be cumbersome and expensive. Therefore, you should invest in robust and free responsive web checker tools like LT Browser to save time and quickly test web designs. With LT Browser you can perform mobile view debugging across 50+ pre-built device viewports, create custom device resolutions, network simulation, hot reloading, generate performance reports powered by Google Lighthouse, and much more.

CSS Animations – Transformations

The transform property in CSS animation makes transformations to the size (rescaling), moving them (translating), rotating them (rotations), or skewing them on the web page. The CSS transform requires four types of values:

1. **none**: To define that no transformation is intended on the object.
2. **value**: To assign a value such as rotate.
3. **inherit**: Inherit the property from the parent element.
4. **initial**: Set the value as the default value for the property assigned to the object.

The “**value**” part defined in the above pointer determines what we want to do with the object.
You can replace the value with one of the four options:

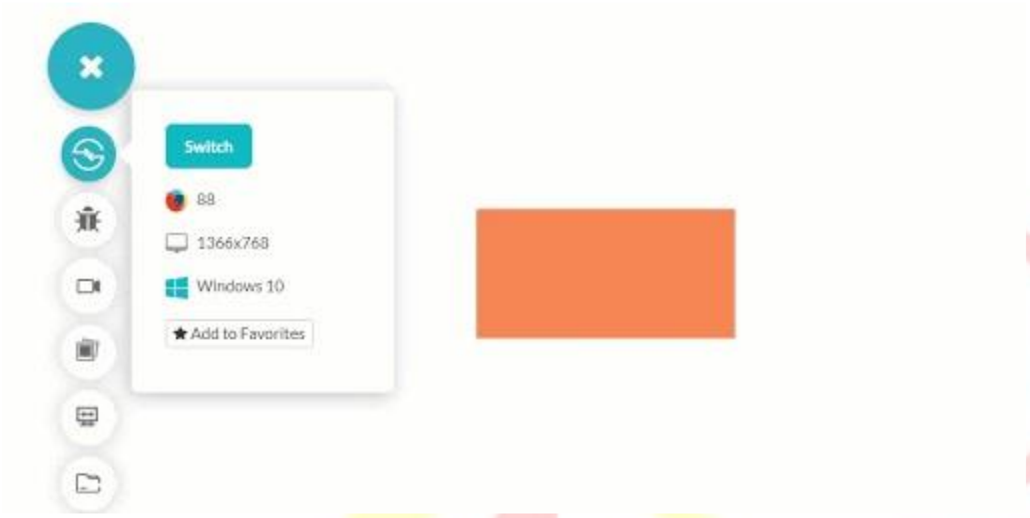
Translate

The “translate” value is used to move the object from point A to point B. The translation can happen either on the X-axis or the Y-axis.

Syntax:
transform: translate_option(values);

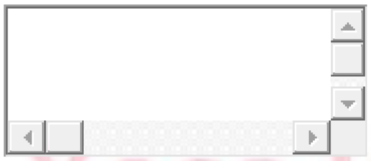
Here are the translate options as follows provide by animation in CSS:

- **translate(value):** To translate only along the X-axis.



- **translate(value, value)**: To translate along the X and Y axis.
- **translateX(value)**: To translate along the X axis.
- **translateY(value)**: To translate along the Y axis.

You can use the following common code and use syntax to replace the values.



```
<html lang="en" dir="ltr">
   <head>
     <meta charset="utf-8">
     <title>Flexbox demo</title>
     <style>
```

```
      .transform_animation {
        width: 200px;
        height: 100px;
        margin-top:200px;
        background-color: rgb(235, 122, 84);
      }

      .transform_animation:hover {
        transform: translate(100px, 100px);
      }

    </style>
  </head>
  <body>
      <br>
<center>
          <div class = "transform_animation">
          </div>
  </center>
  </body>
</html>
```

The “value” can also take negative values that would move in the opposite direction (i.e. along the negative X and Y-axis).

Scale

The next value that the transform property takes in CSS animations is “scale”. With the “scale” property, the object’s dimension can be scaled up or down without changing it’s coordinates.

Syntax:

```
transform: scaling_option(values);
```

The scaling options are similar to the translate option and provide the following values:

- **scale(value, value)**: To scale the object along X-axis and Y-axis.
- **scale(value)**: To take a single value and scale the object to X-axis and Y-axis as per the value.
- **scaleX(value)**: To scale the object on the X-axis only.
- **scaleY(value)**: To scale the object on the Y-axis only.

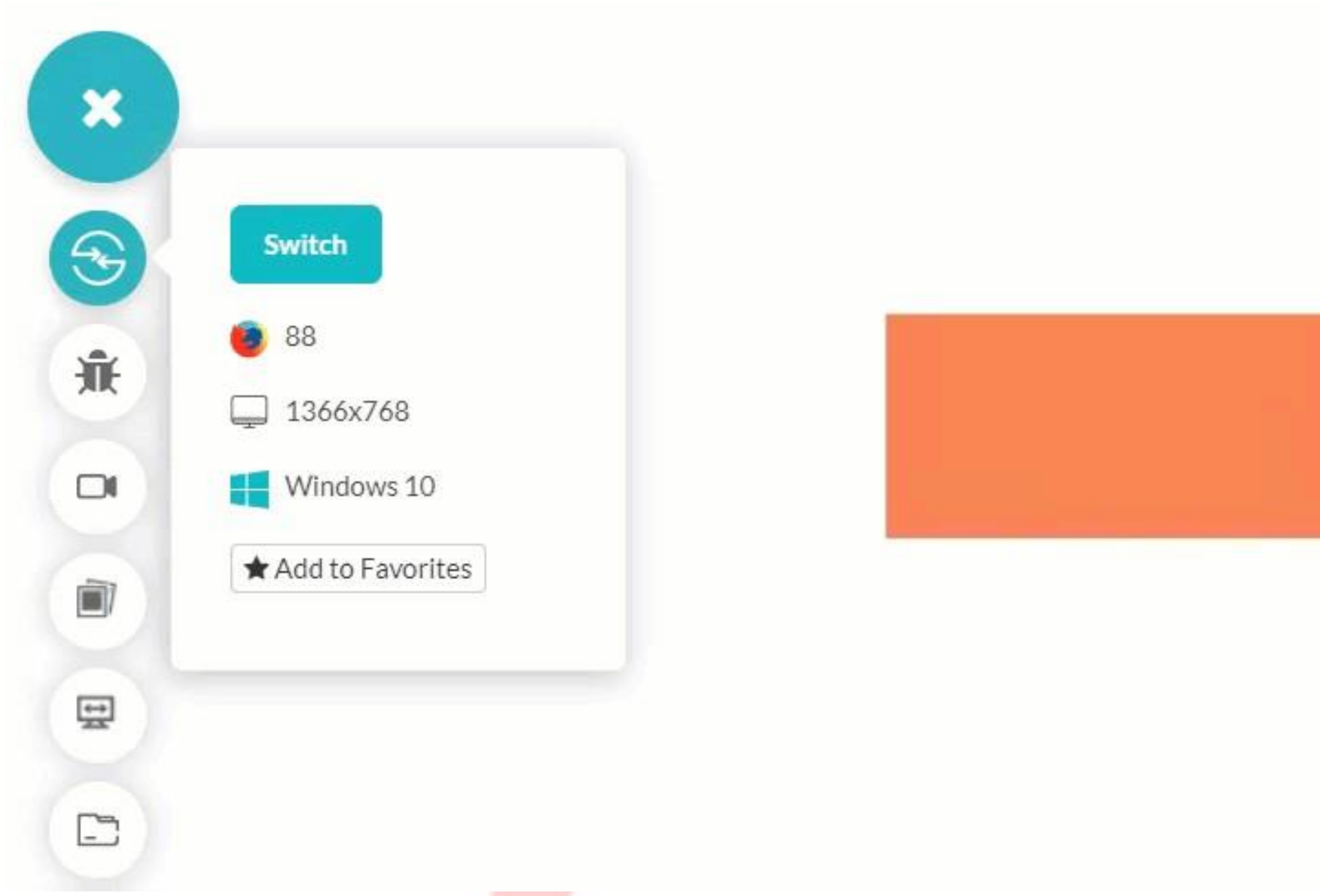
Unlike translate, we cannot specify the “px” value in the scale function. Instead, we need to specify the scaling factor by which we want to expand or shrink the current dimensions. For example, a scaling factor of 2 would expand the dimension to twice the current value. A scaling factor of 0.5 would shrink the dimensions to half the current value.

The following code will scale the dimensions to twice the size.

```
.transform_animation:hover {
        transform: scale(2,2);
      }
```

The rest of the code remains the same as described in the above section (Translate).

Output:



You can also try other values and check their outputs.

Rotate

Another transform attribute is “rotate” which works on the object’s rotation to the specified angle. The angle in “rotate” is specified in either degrees, gradians, radians, or turns. The unit chosen by the user has to be specified along with the numerical number. For example, rotate(30deg) for degrees etc. The rotate function comes with similar functions as follows:

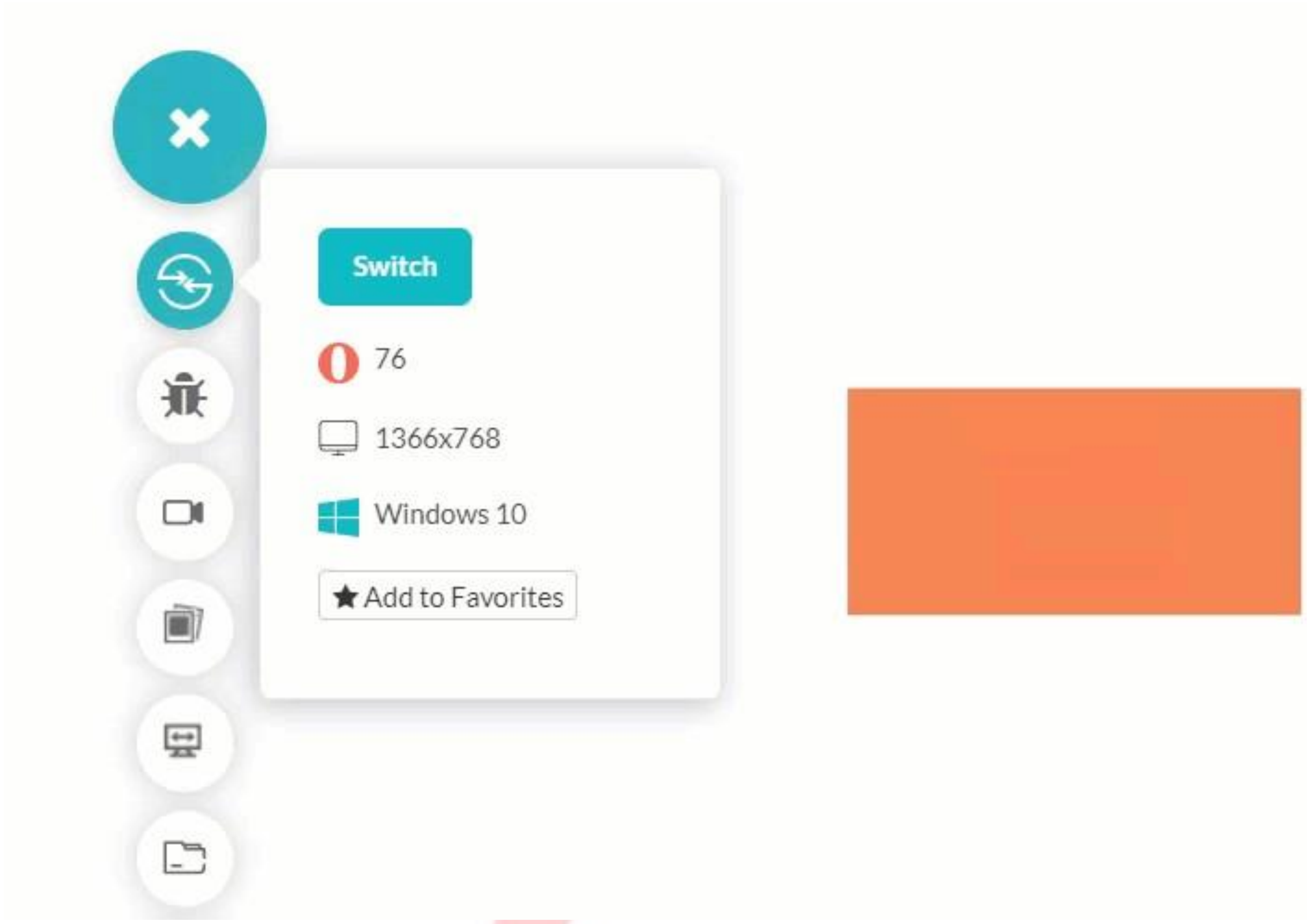
- **rotate(angle)**: To rotate the object at “angle” degrees.
- **rotateX(angle)**: To rotate the object at “angle” degrees along the X-axis.
- **rotateY(angle)**: To rotate the object at “angle” degrees along the Y-axis.

We have changed the code in the above section of this animation in CSS tutorial to realize the rotation of the object at 75 degrees.

Syntax:

```
.transform_animation:hover {
        transform: rotate(75deg);
      }
```

Output:



The following output will be seen when we rotate the same object at 75 degrees along the X-axis.

Output:

Skew

The final transformation in CSS animations is the “skew” attribute. The word “skew” refers to being slanted in English and performing the same job in CSS. The skew function comes in the following variations:

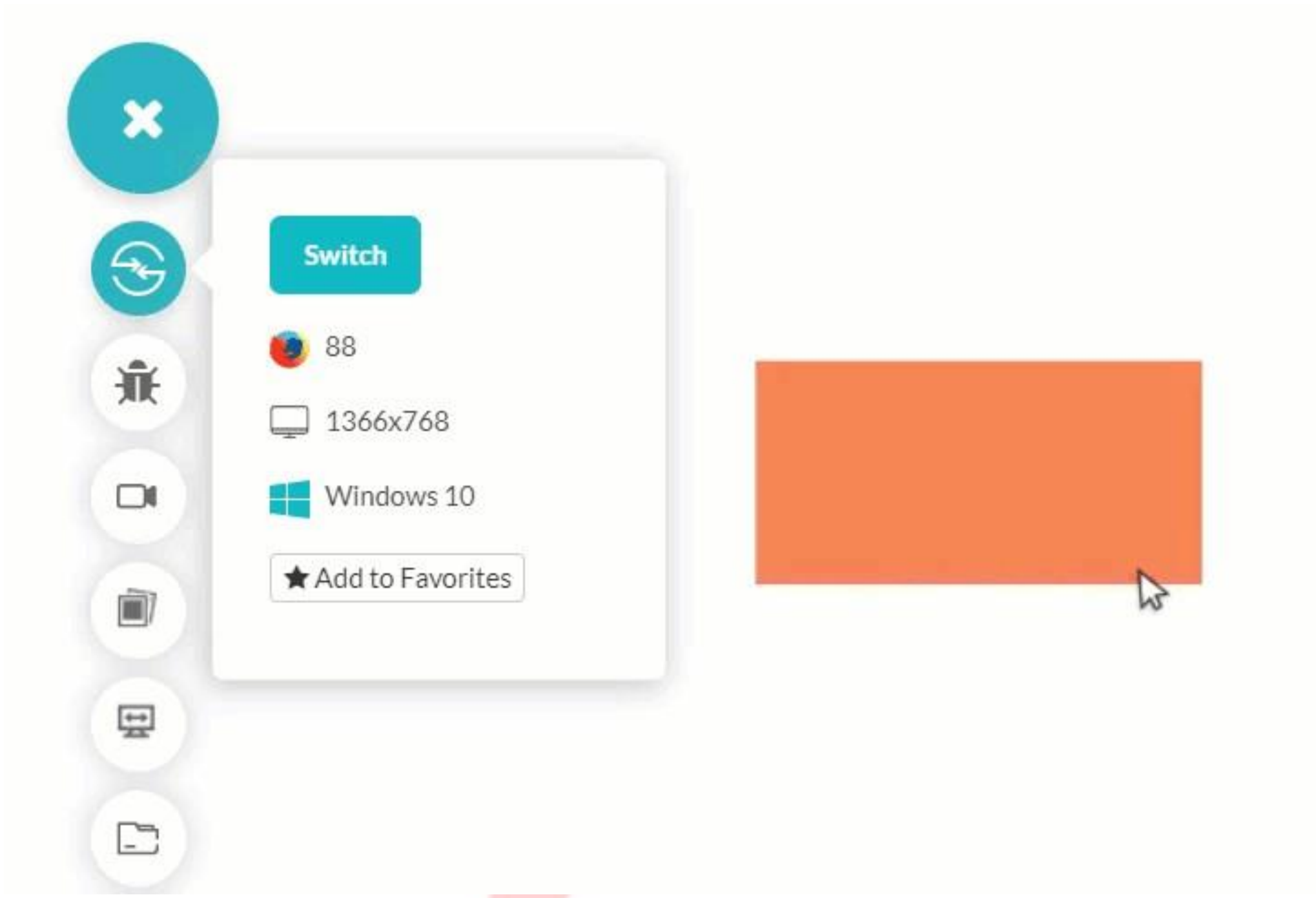
- **skew(X, Y)**: To skew the object at an angle X on the X-axis and Y on the Y-axis.
- **skewX(X)**: To skew the object at an angle X on the X-axis.
- **skewY(Y)**: To skew the object at an angle Y on the Y-axis.

The following code would skew the object at an angle of 30 degrees on the X-axis and an angle of 30 degrees on the Y-axis.

Syntax:

```
.transform_animation:hover {
          transform: skew(30deg, 30deg);
      }
```

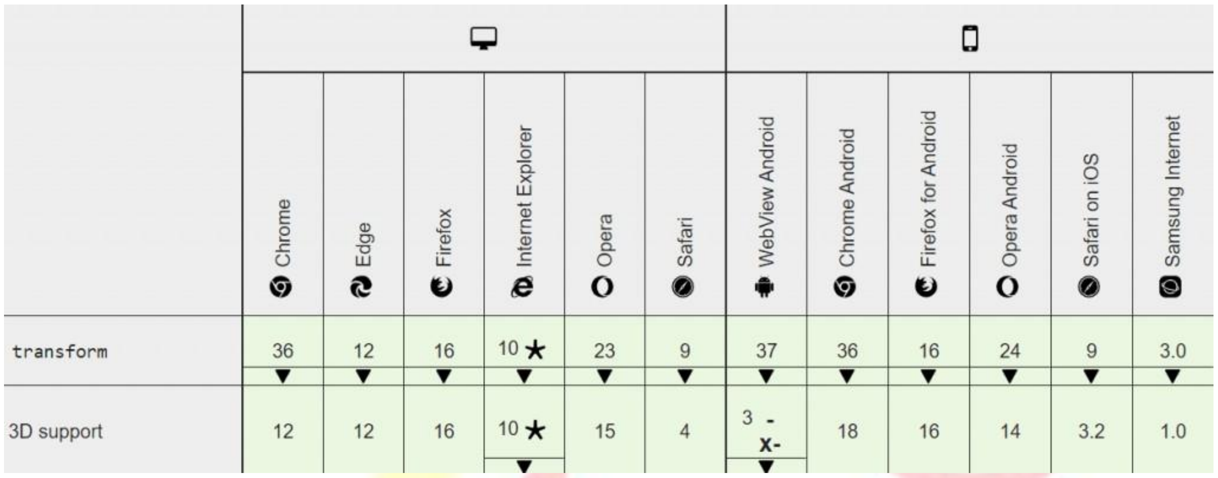
The rest of the code remains the same as in previous sections.



Similar to the rotate function, a user can use the angle in their favorite units by specifying the shorthand (such as deg for degrees). We can now end our transformation section and keep these properties in mind to use them later in the CSS Animations tutorial.

Browser Compatibility For CSS transform

The browser compatibility for CSS transform is great and the function is accepted in all browsers.



	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
transform	36 ▼	12 ▼	16 ▼	10 ★ ▼	23 ▼	9 ▼	37 ▼	36 ▼	16 ▼	24 ▼	9 ▼	3.0 ▼
3D support	12	12	16	10 ★ ▼	15	4	3 - x- ▼	18	16	14	3.2	1.0

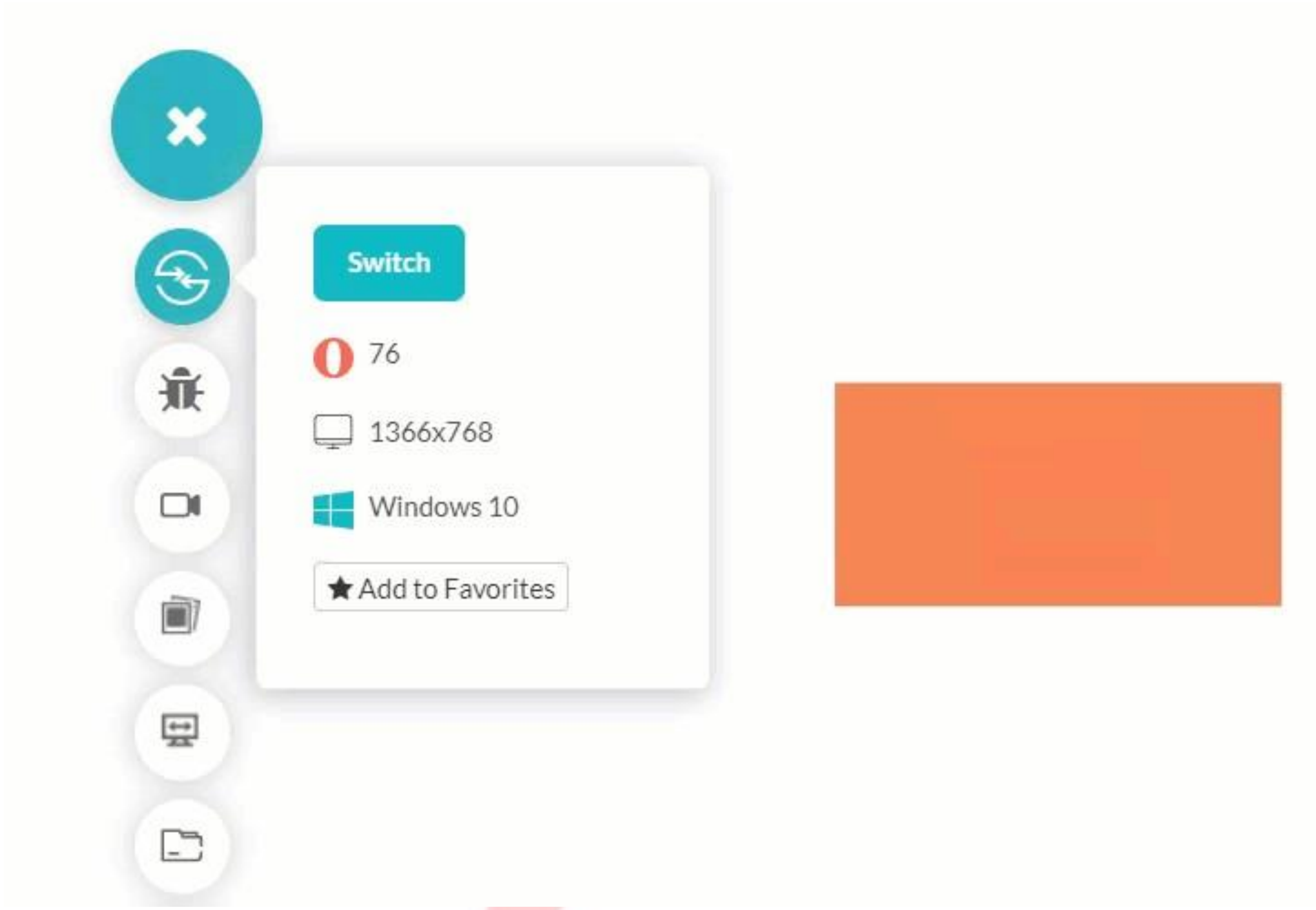
Source

So, you can go ahead and transform a few objects on the web page without worrying about cross-browser compatibility issues.

Read – 12 Modern CSS Techniques For Older CSS Problems

CSS Animations – Transitions

In the above section about the transformations in CSS Animations, we achieved the required goal (such as scaling the dimensions and moving objects) but the transition from the starting state to the end state is abrupt as shown in the below image.



Although the skew function is working fine, it looks as if a video with two different images is combined to make an animated video. A slower stop motion animation perhaps is not what we want from CSS animations. An “animation” will look like an “animation” when the transition from the starting state to the end state is smooth. We will achieve this using the transition property in CSS Animations.

The transition feature in CSS comprises of four properties:

- transition-property

- transition-duration
- transition-timing-function
- transition-delay

transition-property In CSS Transitions

The transition-property attribute specifies what property needs to apply to the transition. For example, transition-property: width will apply the transitions when there is a change in the width of the elements.

It is important to note that not all properties can be animated and applied in the transition-property section. CSS provides a list of animatable properties and they keep on changing with every version of CSS. Therefore, it is important to check your property in CSS Animations and perform cross browser testing from time to time.

Also, a special case occurs in “transition-property” when the developer uses a shorthand property as a value. For example, transition-property: background is one such property. Here, the background is a shorthand for multiple attributes and sub-properties of background. When such a shorthand is used, the animation is applied to all the sub-properties which are under the segment.

The transition-property takes the following values:

- **none**: To signify that no property should make transitions.

- **all**: To signify all the properties that can transit or should transit.
- **property_name**: The name of the property that you want the transition to take place.

Syntax:

`transition-property: property_name;`

transition-duration In CSS Transitions

The “transition-duration” in CSS tells about the time within which the transition needs to be completed. The more time you specify, the slower the transition will happen. If this property is not defined, the default value of zero is taken into consideration.

Syntax:

`transition-property: duration with units;`

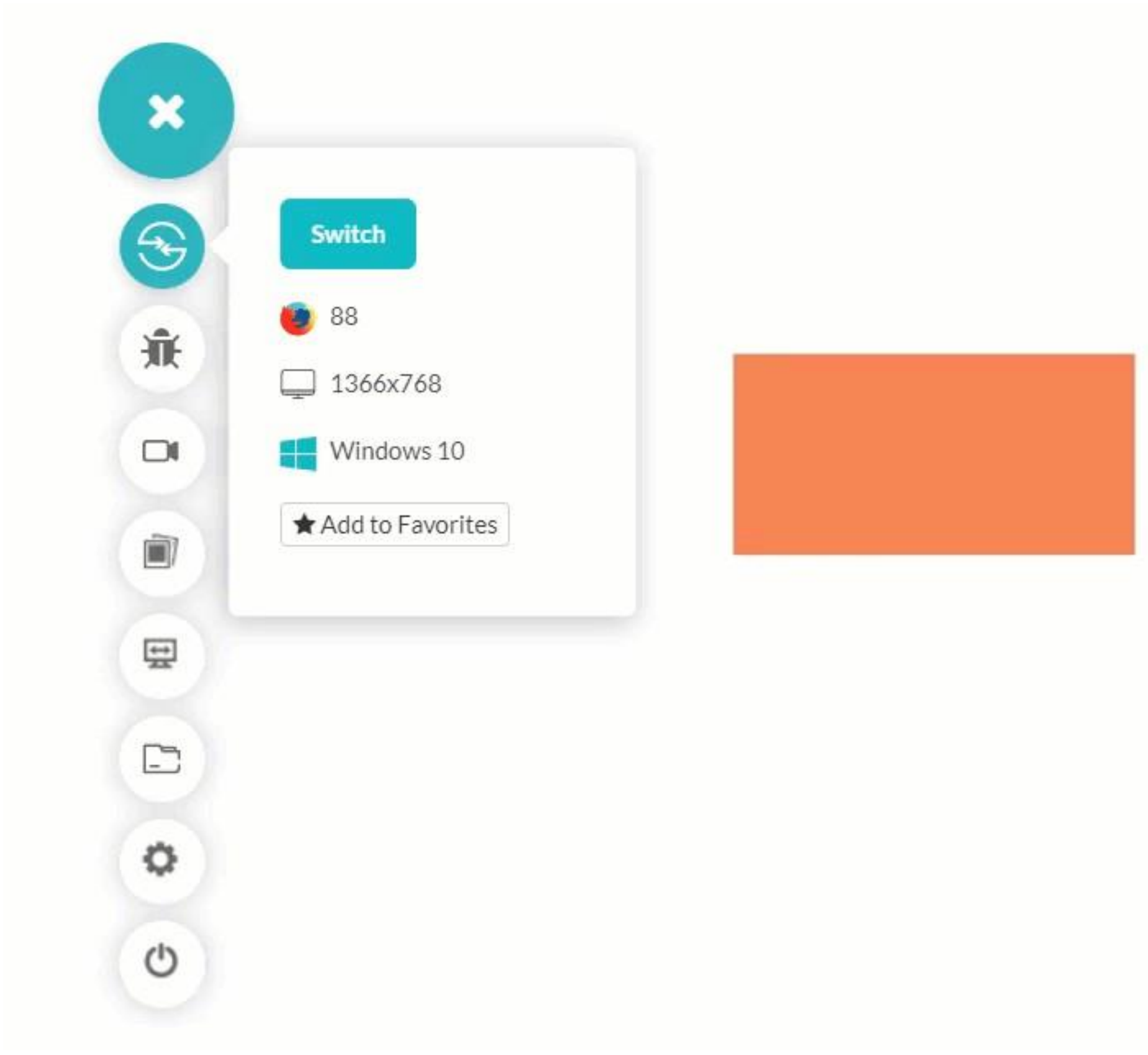
Let us construct the same web page of skewing the object with “transition-duration”:

```
<html lang="en" dir="ltr">
    <head>
```

```
  <meta charset="utf-8">
  <title>Flexbox demo</title>
  <style>
    .transform_animation {
      width: 200px;
      height: 100px;
      margin-top:200px;
      background-color: rgb(235, 122, 84);
    }

    .transform_animation:hover {
      transform: skew(30deg, 30deg);
      transition-duration: 2s;
    }

  </style>
</head>
<body>
```

```
        <br>
<center>
            <div class = "transform_animation">

            </div>
  </center>
  </body>
</html>
```

Output:



In the CSS Transition property, we can also specify multiple values separated by a comma. The same principle can be applied to the transition-duration property as well. But, that has to be calculated properly to match with the property.

- If there are two properties and two time-durations, then they match one on one to each other (i.e. the first property will run for the first duration and the second for the other).
- If the time durations are lesser in number than the properties, they are repeated to match the number of properties.
- If the time durations are more in number than the properties, the extra durations are not considered (and deleted further).

Syntax:

```
transition-duration: x;
```

transition-timing-function In CSS Transitions

The third CSS Transition property in CSS animations is the "transition-timing-function" that lets you decide the speed at which the transition should happen. The developer can use various speeds such as slow start, gradual increase in speed, or use a calculated function of their own. To define the transition-timing, it offers the following values:

- **ease**: Start the transition slowly then increase the speed as time passes till the middle of the animation. From the middle point, the animation slows down gradually. The value "ease" is equivalent to the (0.25, 0.1, 0.25, 1) cubic-bezier function.

- **ease-in**: Start the transition slowly and keep on increasing the speed till the end. The value “ease-in” is equivalent to the (0.42, 0, 1 , 1) cubic-bezier function.
- **ease-out**: Start the transition quickly and keep on decreasing the speed till the end. The value “ease-out” is equivalent to the (0, 0, 0.58, 1) cubic-bezier function.
- **ease-in-out**: Start the transition and keep on increasing the speed towards the middle of the animation. From middle to end, the transition keeps on decreasing at a gradual pace. The ease-in-out and functions are similar except that the ease starts slightly slower than it ends.
- **linear**: The transition starts, moves, and ends at the same speed. The value “linear” is equivalent to the (0, 0, 1, 1) cubic-bezier function.
- **steps(x, jump_term)**: The steps function performs the CSS transitions in a series of steps. The number of steps to perform is denoted by “x” while jump_term denotes how to divide these steps between 0% to 100% of the CSS transition. The “jump_term” can be replaced with one of the following values:

  - **jump-start**: The first jump starts from the beginning of the transition.
  - **jump-end**: The last jump ends at the end of the animation/transition.
  - **jump-none**: The jump doesn’t happen at both ends of the animation.

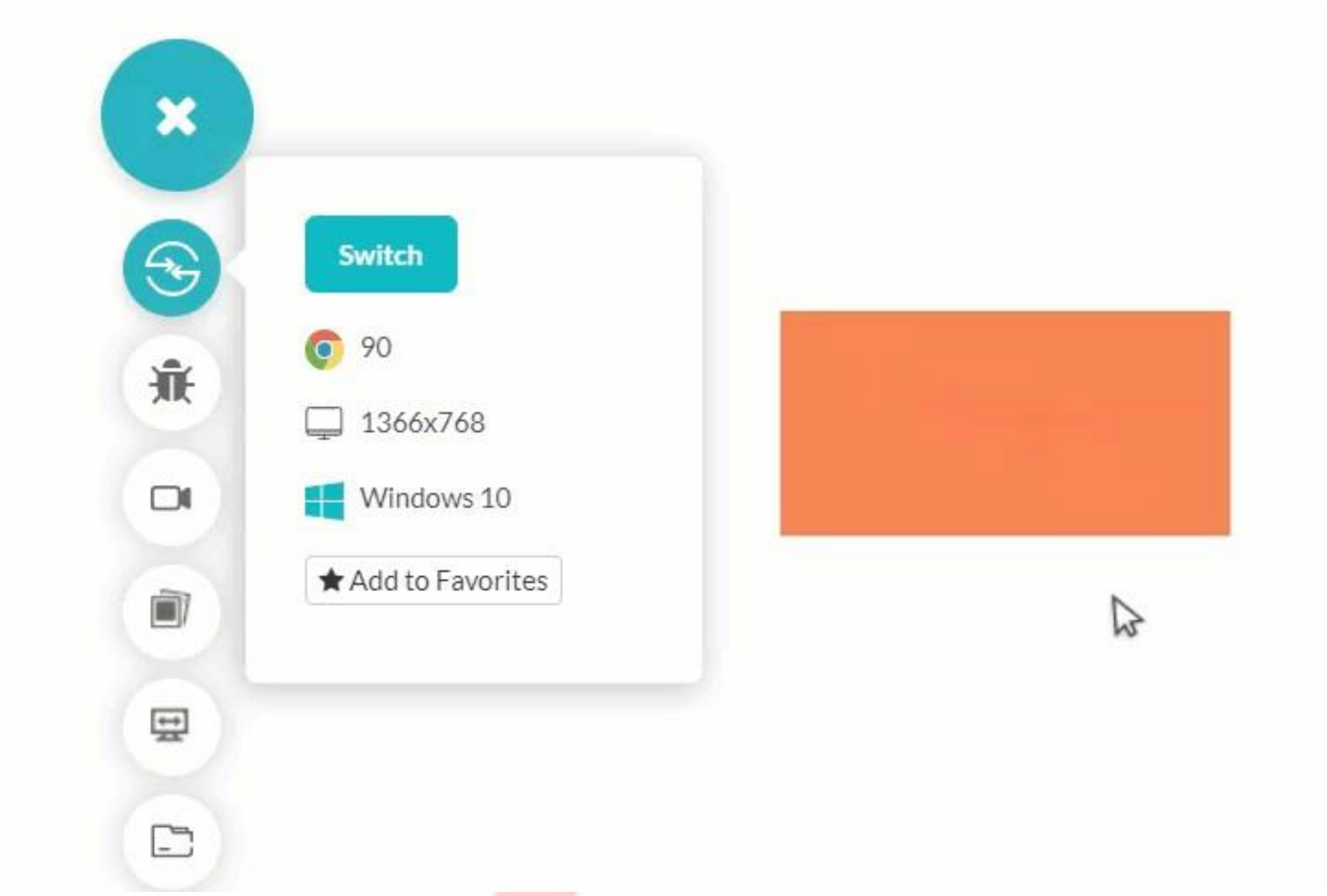
- **jump-both**: The value includes pauses at the start as well as end of the animation.

- **start**: This is similar to jump-start.
- end: This is similar to jump-end.
- **step-start:** This is equivalent to steps(1, jump_start).
- **step-end**: This is equivalent to steps(1, jump-end).

Use this property as follows:

```
.transform_animation:hover {
        transform: skew(30deg, 30deg);
        transition-timing-function: steps(4, jump-end);
        transition-duration: 2s;
      }
```

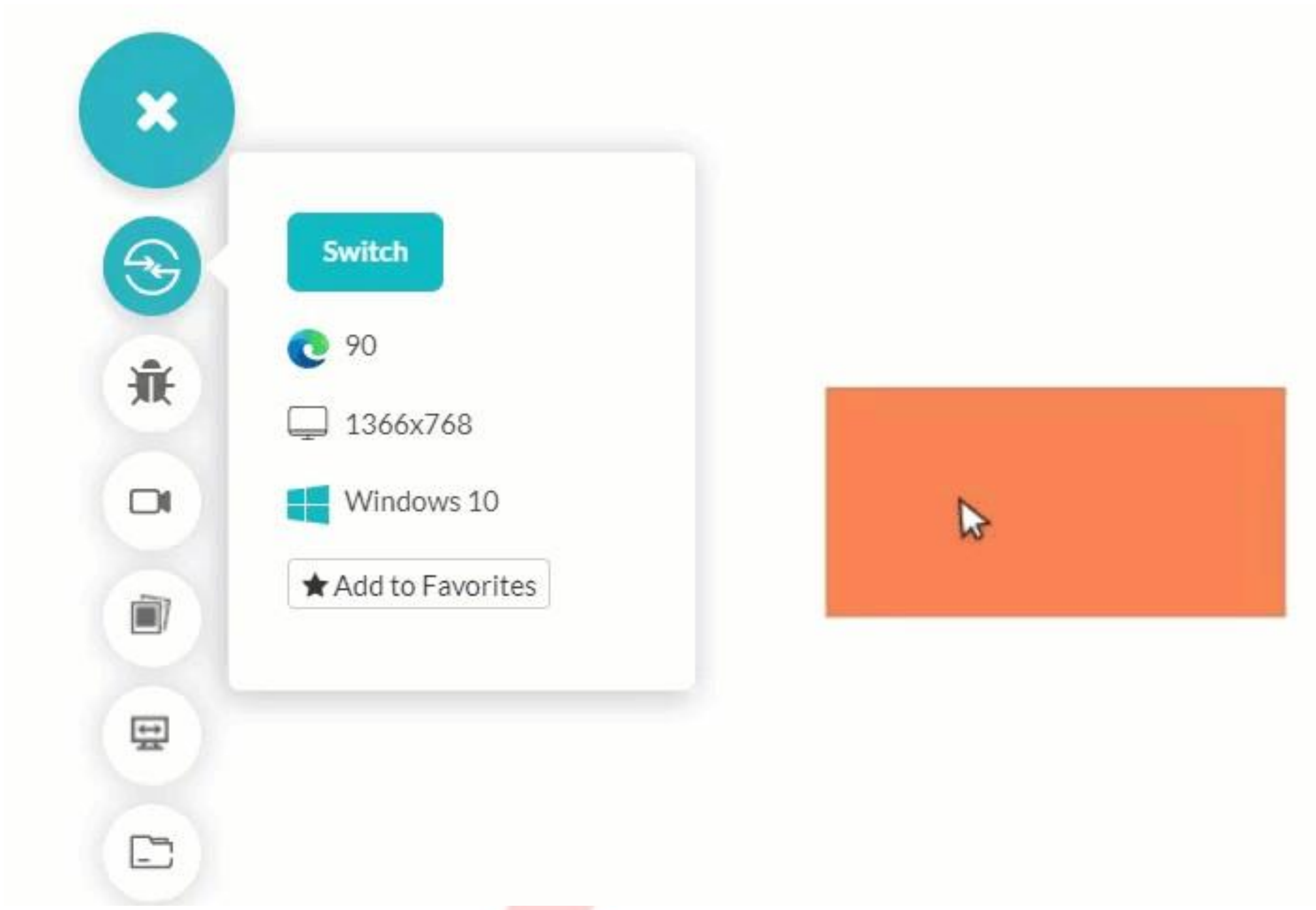
Output:



Notice the difference between the skew methods used in the above two functions and how the steps function has changed it.

transition-delay In CSS Transitions

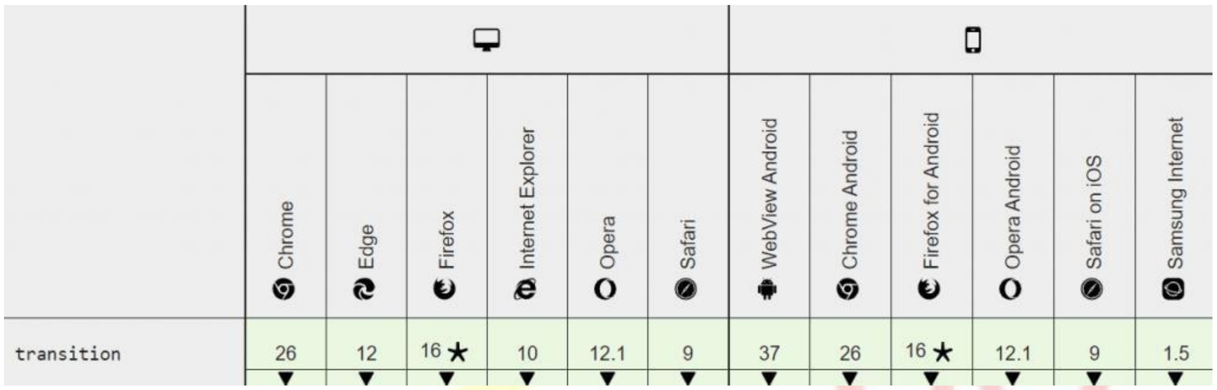
transition-delay is a simple property that defines the wait time before starting the CSS transition. The user needs to define the time units along with numerical values (e.g. 230ms and 2s).



This is the end of CSS Transition properties that provides us with smooth animations.

Browser Support for CSS Transitions

The browser support for CSS Transition is great and all the versions of all the browsers support every sub-property.

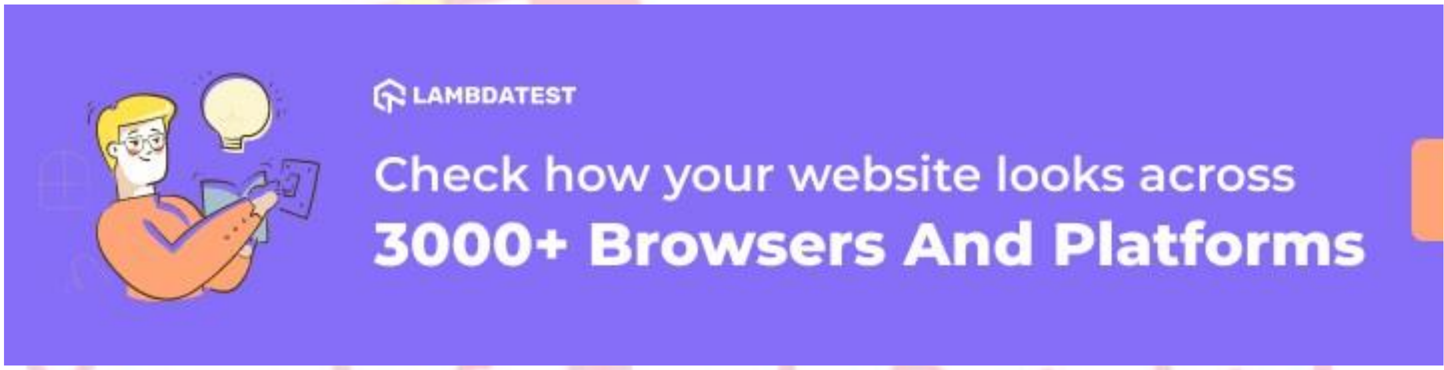


	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
transition	26	12	16 ★	10	12.1	9	37	26	16 ★	12.1	9	1.5

Source

The values starting with the "jump" keyword cannot be implemented in Internet Explorer though. Instead, you can use "start" and "end".

Read – Does Browser Testing On Internet Explorer Still Make Sense?


LAMBDATEST
Check how your website looks across
3000+ Browsers And Platforms

CSS Animations – Keyframes

With this, you would be able to do transformations from point A to point B in a smooth manner. But this is just a single animation we are talking about. We can either choose to expand the width or rotate the object. Doing a single thing will not bring out effective animations that please our eyes. It also challenges the developers!

This is why keyframes were introduced in CSS Animations. It also helps in changing the animations from animation 1 to animation 2 to animation n. On top of it, the developer has the freedom to choose the time (or interval) at which the animation should move from 1 to 2.

There are two ways to achieve the animations using keyframes:

- Using from-to keywords
- Using the percentage assignment

Let’s look at examples that demonstrate examples of CSS Animations using special scenarios.

Using CSS keyframes Using from-to Keywords

from-to keywords are used to define an animation that will move from “from” animations to the “to” animations.

A simple example is as follows:

```
from {
        transform: skew(0deg);
    }
    to {
      transform: skew(30deg);
    }
```

This indicates to the CSS Animation that the object needs to be moved from 0 degrees to 30 degrees.

The only thing missing here is assigning a name to it so that we can call it using CSS or JavaScript. The parser would know the animation that you are interested in. The naming is done with the "@keyframes" keyword (defining that we are defining a keyframe) and a name (also called identifier) after it.

```
@keyframes skew_animation {
    from {
        transform: skew(0deg);
    }
    to {
      transform: skew(30deg);
    }
}
```

This is a complete definition of a keyframe with the terms from and to.

Using CSS keyframes Using Percentages

Using CSS keyframes with percentages to achieve the animation gives us enough freedom and flexibility in constructing complex animations. The percentage defines the part of animation at which the animation needs to start. For example, 0% means implementation at the start of the transition. While 50% would mean applying certain transformations in the middle.

The following example defines three animations that take place from start to middle, middle to end, and at the end of the transition (i.e. at 100%).

```
@keyframes skew_animation {
    0% {
        transform: skew(0deg);
    }
    50% {
      transform: skew(30deg);
    }

    100% {
      background-color: cyan;
    }
  }
```

Once our keyframe is defined, we need to attach it to an object in the HTML so that the parser knows at which place this animation has to be applied. We use the identifier (the keyframe name) to attach it to the object as shown below:

```
animation-name: skew_animation;
```

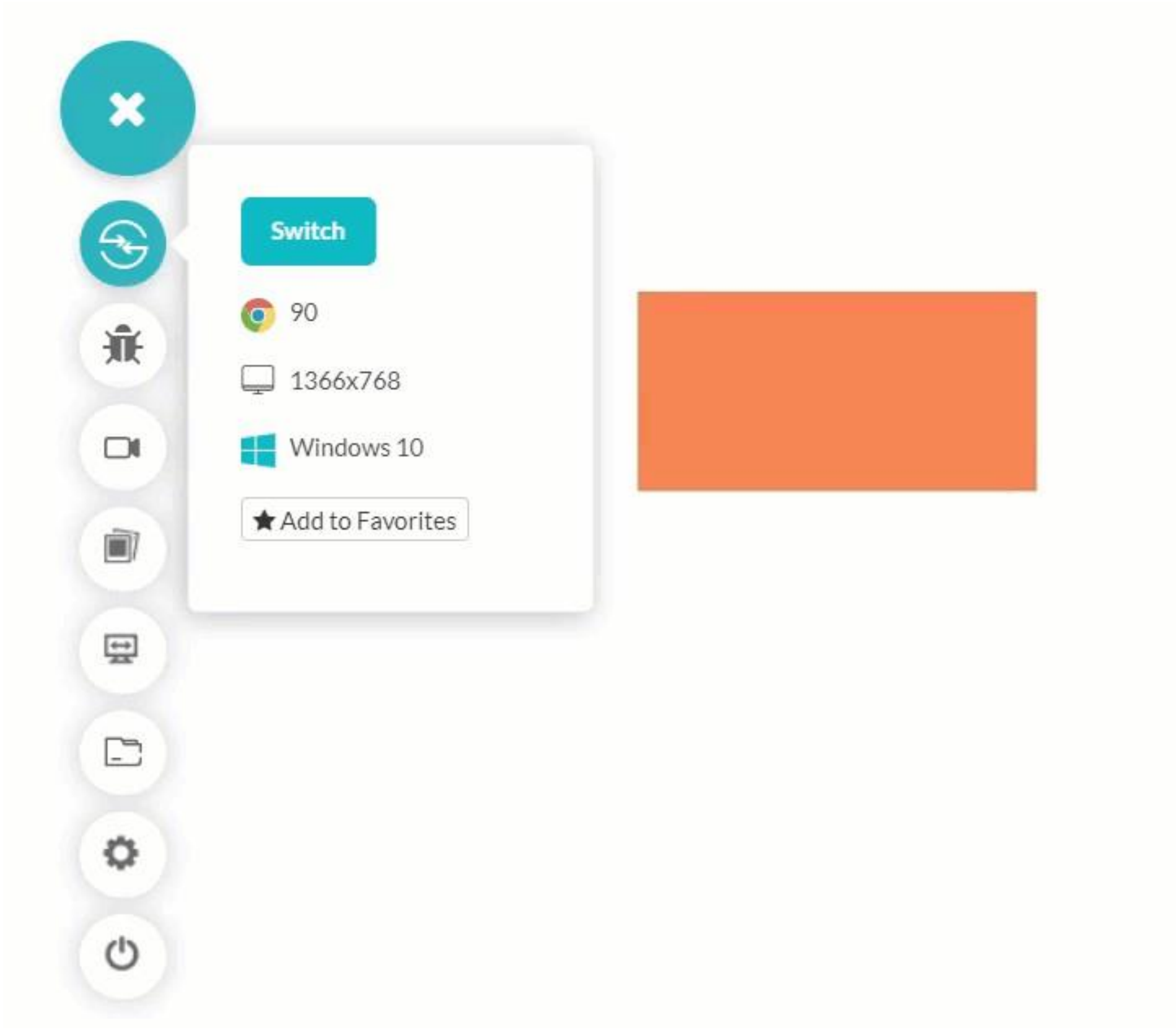
Here is the complete implementation:

```
<html lang="en" dir="ltr">
  <head>
```

```
<meta charset="utf-8">
<title>Flexbox demo</title>
<style>
  .transform_animation {
    width: 200px;
    height: 100px;
    margin-top:200px;
    background-color: rgb(235, 122, 84);
    animation-duration: 2s;
    animation-name: skew_animation;
  }

  @keyframes skew_animation {
  0% {
      transform: skew(0deg);
  }
  50% {
    transform: skew(30deg);
```

```
      }

      100% {
        background-color: cyan;
      }
    }
    </style>
  </head>
  <body>
      <br>
<center>
          <div class = "transform_animation">

          </div>
  </center>
  </body>
</html>
```

Output:



The attribute “animation-duration” can be changed to a higher value if the developer wants to retain the animation for a longer duration.

Animation in CSS And Its Sub Properties

In the above section of the CSS Animations tutorial, we applied the animations to an object using various methods. The last section introduced “animation-duration” and “animation-name” that attach these animations to an object. These are the sub-properties of “animation” that can be used for controlling the animation configurations.

Here are different animation properties:

animation-name

The animation-name property defines the name of the animation (the keyframe identifier) that is required to be attached to the object. Multiple animations can be used with comma-separated values.

```
animation-name: test;
```

animation-duration

animation-duration property specifies the time of completion of one animation cycle. Negative values are considered as zero in the “animation-duration”.

```
animation-duration: 4s;
```

animation-timing-function

Similar to “transition-timing-function”, the “animation-timing-function” works on the complete keyframe (i.e. how the animation should proceed). It takes the same values as defined in the “translation-timing-function”. Please refer to the CSS transition function to know more.

```
animation-timing-function: linear;
```

animation-delay

The animation-delay property specifies the time to wait before starting the animation. This property is similar to the “transition-delay” property but it works on the complete animation cycle. The “animation-delay” can also take a negative value but the animation starts immediately for negative delays.

```
animation-delay: 2s;
```

animation-iteration-count

The “animation-iteration-count” specifies the number of times the animation should be played before coming to a halt. You can either use a positive integer number or the keyword “infinite” to play animation infinitely.

Decimal positive numbers such as 0.5 are also valid specifying that half of the animation needs to be played. Negative numbers however are not considered valid.

```
animation-iteration-count: 4;
```

animation-direction

The “animation-direction” property specifies how the animation should be played when triggered. The “animation-direction” takes the following values:

- **normal**: The animation is played as it should without any changes and returns to the start state after each cycle.
- **reverse**: The animation plays from the end state to the start state (i.e. in reverse).
- **alternate**: The animation moves from normal to reverse every alternate time. So for the first cycle animation is played in the “normal” direction and the next cycle happens in reverse.
- **alternate-reverse**: This is the reverse of the alternate value and the animation moves backward in alternate-reverse.

```
animation-direction: reverse;
```

animation-fill-mode

The “animation-fill-mode” is used for applying the style to the element before start of the animation, after the end of the animation, when the animation is not playing, or at the start & end of the animation. The animation-fill-mode takes the following values:

- **none**: This is the default value of “animation-fill-mode” and does not apply any style to the element.
- **forwards**: The last keyframe styles are retained and applied to the element. The last played keyframe depends on the “animation-direction” and the “animation-iteration-count”.
- **backwards**: The first relevant keyframe’s styles are applied to the element. The keyframe depends on the “animation-direction” which is also retained during the “animation-delay” period.
- **both**: The animation styles are applicable in forward and backward direction.

```
animation-fill-mode: forwards;
```

animation-play-state

The “animation-play-state” property sets the state of the animation to either running or paused state. When the animation is paused, it resumes from the same point when the state is changed to “running”.

The animation-play-state accepts only two values: “running” and “paused”.

`animation-play-state: running;`

CSS Animation Shorthand Property

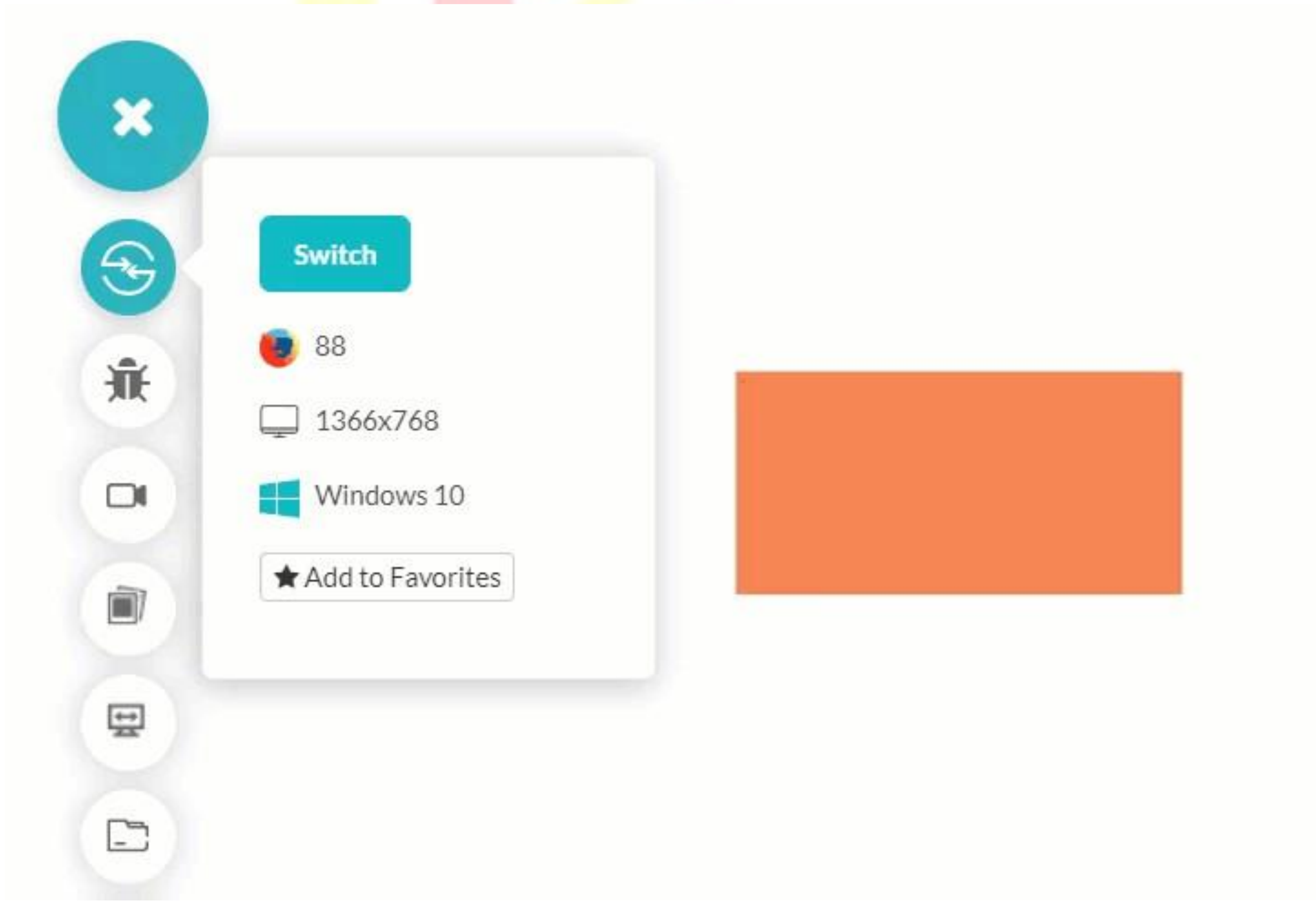
Shorthand properties are a great way to save space and implement all the useful specifications in a single line. You can refer to our earlier blog on CSS Flexbox guide where we demonstrated flex using three different properties. The property “animate” in CSS constitutes the following property:

- animation-name
- animation-duration
- animation-timing-function
- animation-delay
- animation-iteration-count
- animation-direction
- animation-fill-mode
- animation-play-state

Here is a simple example that demonstrates the usage of animation shorthand:

```
.transform_animation {
     width: 200px;
     height: 100px;
     margin-top:200px;
     background-color: rgb(235, 122, 84);
     animation: skew_animation 2s linear 0s 2 normal running;
    }

    @keyframes skew_animation {
    0% {
      transform: skew(0deg);
    }
    50% {
     transform: skew(30deg);
    }

    100% {
      background-color: cyan;
    }
  }
```

Output:

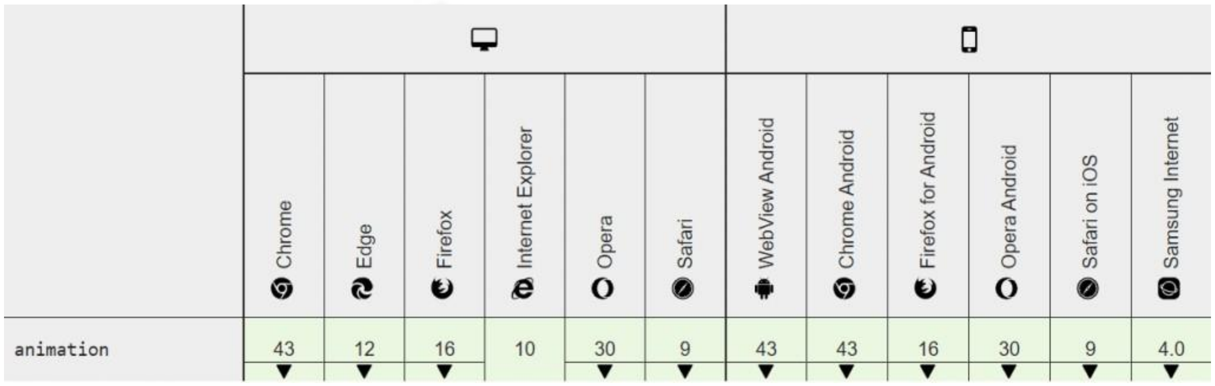


It is important to specify the “animation-duration” property else the default value ‘0’ is considered. With this, the animation will not start. If the properties are similar for two or more segments, you can combine them with a comma-separated value as follows:

```
0%, 100% {
 properties;
}
50% {
properties
}
```

Browser Support For Animation Property

Animation property has great support among the browsers.



	🖥						📱					
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet
animation	43 ▼	12 ▼	16 ▼	10	30 ▼	9 ▼	43 ▼	43 ▼	16 ▼	30 ▼	9 ▼	4.0 ▼

Source

Since the animation property contains different animation sub-properties, all of them enjoy similar browser support among the browsers.

INTRODUCTION OF JAVASCRIPT

What is JavaScript?

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.). There are also more advanced server side versions of JavaScript such as Node.js, which allow you to add more functionality to a website than downloading files (such as realtime collaboration between multiple computers). Inside a host environment (for example, a web browser), JavaScript can be connected to the objects of its environment to provide programmatic control over them.

JavaScript contains a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- *Client-side JavaScript* extends the core language by supplying objects to control a browser and its *Document Object Model* (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- *Server-side JavaScript* extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

This means that in the browser, JavaScript can change the way the webpage (DOM) looks. And, likewise, Node.js JavaScript on the server can respond to custom requests sent by code executed in the browser.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in some others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript follows most

Java expression syntax, naming conventions and basic control-flow constructs which was the reason why it was renamed from LiveScript to JavaScript.

In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Java is a class-based programming language designed for fast execution and type safety. Type safety means, for instance, that you can't cast a Java integer into an object reference or access private memory by corrupting the Java bytecode. Java's class-based model means that programs consist exclusively of classes and their methods. Java's class inheritance and strong typing generally require tightly coupled object hierarchies. These requirements make Java programming more complex than JavaScript programming.

In contrast, JavaScript descends in spirit from a line of smaller, dynamically typed languages such as HyperTalk and dBASE. These scripting languages offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation.

JavaScript	**Java**
Object-oriented. No distinction between types of objects. Inheritance is through the prototype mechanism, and properties and methods can be added to any object dynamically.	Class-based. Objects are divided into classes and instances with all inheritance through the class hierarchy. Classes and instances cannot have properties or methods added dynamically.
Variable data types are not declared (dynamic typing, loosely typed).	Variable data types must be declared (static typing, strongly typed).
Cannot automatically write to hard disk.	Can automatically write to hard disk.

Getting started with JavaScript

Getting started with JavaScript is easy: all you need is a modern Web browser. This guide includes some JavaScript features which are only currently available in the latest versions of Firefox, so using the most recent version of Firefox is recommended.

The *Web Console* tool built into Firefox is useful for experimenting with JavaScript; you can use it in two modes: single-line input mode, and multi-line input mode.

Single-line input in the Web Console

The Web Console shows you information about the currently loaded Web page, and also includes a JavaScript interpreter that you can use to execute JavaScript expressions in the current page.

To open the Web Console (Ctrl+Shift+I on Windows and Linux or Cmd-Option-K on Mac), open the **Tools** menu in Firefox, and select "**Developer ▶ Web Console**".

The Web Console appears at the bottom of the browser window. Along the bottom of the console is an input line that you can use to enter JavaScript, and the output appears in the panel above:

The console works the exact same way as eval: the last expression entered is returned. For the sake of simplicity, it can be imagined that every time something is entered into the console, it is actually surrounded by console.log around eval, like so:

```
console.log(eval('3 + 5'))
```

Copy to Clipboard

Multi-line input in the Web Console

The single-line input mode of the Web Console is great for quick testing of JavaScript expressions, but although you can execute multiple lines, it's not very convenient for that. For more complex JavaScript, you can use the multi-line input mode.

Hello world

To get started with writing JavaScript, open the Web Console in multi-line mode, and write your first "Hello world" JavaScript code:

```
(function(){
  "use strict";
  /* Start of your code */
  function greetMe(yourName) {
  alert(`Hello ${yourName}`);
  }

  greetMe('World');
  /* End of your code */
})();
```

Copy to Clipboard

Press Cmd+Enter or Ctrl+Enter (or click the **Run** button) to watch it unfold in your browser!

In the following pages, this guide introduces you to the JavaScript syntax and language features, so that you will be able to write more complex applications.

But for now, remember to always include the (function(){"use strict"; before your code, and add })(); to the end of your code. The strict mode and IIFE articles explain what those do, but for now they can be thought of as doing the following:

1. Prevent semantics in JavaScript that trip up beginners.
2. Prevent code snippets executed in the console from interacting with one another (e.g., having something created in one console execution being used for a different console execution).

Control flow and error handling

Block statement

The most basic statement is a *block statement*, which is used to group statements. The block is delimited by a pair of curly brackets:

```
{
  statement1;
  statement2;
  // …
  statementN;
}
```

Copy to Clipboard

Example

Block statements are commonly used with control flow statements (if, for, while).

```
while (x < 10) {
  x++;
}
```

Copy to Clipboard

Here, { x++; } is the block statement.

Note: var-declared variables are not block-scoped, but are scoped to the containing function or script, and the effects of setting them persist beyond the block itself. For example:

```
var x = 1;
{
  var x = 2;
}
console.log(x); // 2
```

Copy to Clipboard

This outputs 2 because the var x statement within the block is in the same scope as the var x statement before the block. (In C or Java, the equivalent code would have output 1.)

This scoping effect can be mitigated by using let or const.

Conditional statements

A conditional statement is a set of commands that executes if a specified condition is true. JavaScript supports two conditional statements: if...else and switch.

if...else statement

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false.

An if statement looks like this:

```
if (condition) {
  statement1;
} else {
  statement2;
}
```

Copy to Clipboard

Here, the condition can be any expression that evaluates to true or false. (See Boolean for an explanation of what evaluates to true and false.)

If condition evaluates to true, statement_1 is executed. Otherwise, statement_2 is executed. statement_1 and statement_2 can be any statement, including further nested if statements.

You can also compound the statements using else if to have multiple conditions tested in sequence, as follows:

```
if (condition1) {
  statement1;
} else if (condition2) {
  statement2;
} else if (conditionN) {
  statementN;
} else {
  statementLast;
}
```

Copy to Clipboard

In the case of multiple conditions, only the first logical condition which evaluates to true will be executed. To execute multiple statements, group them within a block statement ({ /* … */ }).

Best practice

In general, it's good practice to always use block statements—*especially* when nesting if statements:

```
if (condition) {
  // Statements for when condition is true
  // …
} else {
  // Statements for when condition is false
  // …
}
```

Copy to Clipboard

In general it's good practice to not have an if...else with an assignment like x = y as a condition:

```
if (x = y) {
  /* statements here */
}
```

However, in the rare case you find yourself wanting to do something like that, the while documentation has a Using an assignment as a condition section with guidance on a general best-practice syntax you should know about and follow.

Falsy values

The following values evaluate to false (also known as Falsy values):

- false
- undefined
- null
- 0
- NaN
- the empty string ("")

All other values—including all objects—evaluate to true when passed to a conditional statement.

Note: Do not confuse the primitive boolean values true and false with the true and false values of the Boolean object!

For example:

```
const b = new Boolean(false);
if (b) {
  // this condition evaluates to true
}
if (b == true) {
  // this condition evaluates to false
}
```

Copy to Clipboard

Example

In the following example, the function checkData returns true if the number of characters in a Text object is three. Otherwise, it displays an alert and returns false.

```
function checkData() {
  if (document.form1.threeChar.value.length === 3) {
    return true;
  } else {
    alert(`Enter exactly three characters.
${document.form1.threeChar.value} is not valid.`);
```

```
    return false;
  }
}
```

Copy to Clipboard

switch statement

A switch statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement.

A switch statement looks like this:

```
switch (expression) {
  case label1:
  statements1;
    break;
  case label2:
    statements2;
    break;
  // …
  default:
    statementsDefault;
}
```

Copy to Clipboard

JavaScript evaluates the above switch statement as follows:

- The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements.
- If no matching label is found, the program looks for the optional default clause:
  - If a default clause is found, the program transfers control to that clause, executing the associated statements.
  - If no default clause is found, the program resumes execution at the statement following the end of switch.
  - (By convention, the default clause is written as the last clause, but it does not need to be so.)

break statements

The optional break statement associated with each case clause ensures that the program breaks out of switch once the matched statement is executed, and then continues execution at the statement following switch. If break is omitted, the program continues execution inside the switch statement (and will evaluate the next case, and so on).

EXAMPLE

In the following example, if fruitType evaluates to 'Bananas', the program matches the value with case 'Bananas' and executes the associated statement. When break is encountered, the program exits the switch and continues execution from the statement following switch. If break were omitted, the statement for case 'Cherries' would also be executed.

```
switch (fruitType) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
    break;
  case 'Apples':
    console.log('Apples are $0.32 a pound.');
    break;
  case 'Bananas':
    console.log('Bananas are $0.48 a pound.');
    break;
  case 'Cherries':
    console.log('Cherries are $3.00 a pound.');
    break;
  case 'Mangoes':
    console.log('Mangoes are $0.56 a pound.');
    break;
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
    break;
  default:
    console.log(`Sorry, we are out of ${fruitType}.`);
}
console.log("Is there anything else you'd like?");
```
Copy to Clipboard

Exception handling statements

You can throw exceptions using the throw statement and handle them using the try...catch statements.

- throw statement
- try...catch statement

Exception types

Just about any object can be thrown in JavaScript. Nevertheless, not all thrown objects are created equal. While it is common to throw numbers or strings as errors, it is frequently more effective to use one of the exception types specifically created for this purpose:

- ECMAScript exceptions
- DOMException and DOMError

throw statement

Use the throw statement to throw an exception. A throw statement specifies the value to be thrown:

```
throw expression;
```

Copy to Clipboard

You may throw any expression, not just expressions of a specific type. The following code throws several exceptions of varying types:

```
throw 'Error2';   // String type
throw 42;         // Number type
throw true;       // Boolean type
throw {toString() { return "I'm an object!"; } };
```

Copy to Clipboard

try...catch statement

The try...catch statement marks a block of statements to try, and specifies one or more responses should an exception be thrown. If an exception is thrown, the try...catch statement catches it.

The try...catch statement consists of a try block, which contains one or more statements, and a catch block, containing statements that specify what to do if an exception is thrown in the try block.

In other words, you want the try block to succeed—but if it does not, you want control to pass to the catch block. If any statement within the try block (or in a function called from within the try block) throws an exception, control *immediately* shifts to the catch block. If no exception is thrown in the try block, the catch block is skipped. The finally block executes after the try and catch blocks execute but before the statements following the try...catch statement.

The following example uses a try...catch statement. The example calls a function that retrieves a month name from an array based on the value passed to the function. If the value does not correspond to a month number (1 – 12), an exception is thrown with the value 'InvalidMonthNo' and the statements in the catch block set the monthName variable to 'unknown'.

```
function getMonthName(mo) {
  mo--; // Adjust month number for array index (so that 0 = Jan, 11 = Dec)
  const months = [
    'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
    'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
  ];
  if (months[mo]) {
    return months[mo];
  } else {
    throw new Error('InvalidMonthNo'); // throw keyword is used here
  }
}

try { // statements to try
```

```
  monthName = getMonthName(myMonth); // function could throw exception
} catch (e) {
  monthName = 'unknown';
  logMyErrors(e); // pass exception object to error handler (i.e. your own function)
}
```

Copy to Clipboard

The catch block

You can use a catch block to handle all exceptions that may be generated in the try block.

```
catch (catchID) {
  statements
}
```

Copy to Clipboard

The catch block specifies an identifier (catchID in the preceding syntax) that holds the value specified by the throw statement. You can use this identifier to get information about the exception that was thrown.

JavaScript creates this identifier when the catch block is entered. The identifier lasts only for the duration of the catch block. Once the catch block finishes executing, the identifier no longer exists.

For example, the following code throws an exception. When the exception occurs, control transfers to the catch block.

```
try {
  throw 'myException'; // generates an exception
} catch (err) {
  // statements to handle any exceptions
  logMyErrors(err);    // pass exception object to error handler
}
```

Copy to Clipboard

Note: When logging errors to the console inside a catch block, using console.error() rather than console.log() is advised for debugging. It formats the message as an error, and adds it to the list of error messages generated by the page.

The finally block

The finally block contains statements to be executed *after* the try and catch blocks execute. Additionally, the finally block executes *before* the code that follows the try…catch…finally statement.

It is also important to note that the finally block will execute *whether or not* an exception is thrown. If an exception is thrown, however, the statements in the finally block execute even if no catch block handles the exception that was thrown.

You can use the finally block to make your script fail gracefully when an exception occurs. For example, you may need to release a resource that your script has tied up.

The following example opens a file and then executes statements that use the file. (Server-side JavaScript allows you to access files.) If an exception is thrown while the file is open, the finally block closes the file before the script fails. Using finally here *ensures* that the file is never left open, even if an error occurs.

```
openMyFile();
try {
  writeMyFile(theData); // This may throw an error
} catch (e) {
  handleError(e); // If an error occurred, handle it
} finally {
  closeMyFile(); // Always close the resource
}
```

Copy to Clipboard

If the finally block returns a value, this value becomes the return value of the entire try…catch…finally production, regardless of any return statements in the try and catch blocks:

```
function f() {
  try {
    console.log(0);
    throw 'bogus';
  } catch (e) {
    console.log(1);
    return true;    // this return statement is suspended
                    // until finally block has completed
    console.log(2); // not reachable
  } finally {
    console.log(3);
    return false;   // overwrites the previous "return"
    console.log(4); // not reachable
  }
  // "return false" is executed now
  console.log(5);   // not reachable
}
console.log(f()); // 0, 1, 3, false
```

Copy to Clipboard

Overwriting of return values by the finally block also applies to exceptions thrown or re-thrown inside of the catch block:

```
function f() {
  try {
    throw 'bogus';
  } catch (e) {
    console.log('caught inner "bogus"');
    throw e; // this throw statement is suspended until
             // finally block has completed
  } finally {
    return false; // overwrites the previous "throw"
  }
  // "return false" is executed now
}

try {
  console.log(f());
} catch (e) {
  // this is never reached!
  // while f() executes, the `finally` block returns false,
  // which overwrites the `throw` inside the above `catch`
  console.log('caught outer "bogus"');
}
```

```
// Logs:
// caught inner "bogus"
// false
```

Copy to Clipboard

Nesting try...catch statements

You can nest one or more try...catch statements.

If an inner try block does *not* have a corresponding catch block:

1. it *must* contain a finally block, and
2. the enclosing try...catch statement's catch block is checked for a match.

For more information, see nested try-blocks on the try...catch reference page.

Utilizing Error objects

Depending on the type of error, you may be able to use the name and message properties to get a more refined message.

The name property provides the general class of Error (such as DOMException or Error), while message generally provides a more succinct message than one would get by converting the error object to a string.

If you are throwing your own exceptions, in order to take advantage of these properties (such as if your catch block doesn't discriminate between your own exceptions and system ones), you can use the Error constructor.

For example:

```
function doSomethingErrorProne() {
  if (ourCodeMakesAMistake()) {
  throw new Error('The message');
  } else {
    doSomethingToGetAJavaScriptError();
  }
}

try {
  doSomethingErrorProne();
} catch (e) {
  // Now, we actually use `console.error()`
  console.error(e.name); // 'Error'
  console.error(e.message); // 'The message', or a JavaScript error message
}
```

JavaScript Datatypes

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows you to work with three primitive data types −

- **Numbers,** eg. 123, 120.50 etc.
- **Strings** of text e.g. "This text string" etc.
- **Boolean** e.g. true or false.

JavaScript also defines two trivial data types, **null** and **undefined,** each of which defines only a single value. In addition to these primitive data types, JavaScript supports a composite data type known as **object**. We will cover objects in detail in a separate chapter.

Note − JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.

JavaScript Variables

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the **var** keyword as follows.

```
<script type = "text/javascript">
   <!--
      var money;
      var name;
   //-->
</script>
```

You can also declare multiple variables with the same **var** keyword as follows −

```
<script type = "text/javascript">
   <!--
      var money, name;
   //-->
</script>
```

Storing a value in a variable is called **variable initialization**. You can do variable initialization at the time of variable creation or at a later point in time when you need that variable.

For instance, you might create a variable named **money** and assign the value 2000.50 to it later. For another variable, you can assign a value at the time of initialization as follows.

```
<script type = "text/javascript">
  <!--
    var name = "Ali";
    var money;
    money = 2000.50;
  //-->
</script>
```

Note − Use the **var** keyword only for declaration or initialization, once for the life of any variable name in a document. You should not re-declare same variable twice.

JavaScript is **untyped** language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

JavaScript Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variables have only two scopes.

- **Global Variables** − A global variable has global scope which means it can be defined anywhere in your JavaScript code.

- **Local Variables** − A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable. Take a look into the following example.

```
<html>
   <body onload = checkscope();>
      <script type = "text/javascript">
         <!--
            var myVar = "global";      // Declare a global variable
            function checkscope( ) {
               var myVar = "local";    // Declare a local variable
               document.write(myVar);
            }
         //-->
      </script>
   </body>
</html>
```

This produces the following result −

local

JavaScript Variable Names

While naming your variables in JavaScript, keep the following rules in mind.

- You should not use any of the JavaScript reserved keywords as a variable name. These keywords are mentioned in the next section. For example, **break** or **boolean** variable names are not valid.
- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, **123test** is an invalid variable name but **_123test** is a valid one.
- JavaScript variable names are case-sensitive. For example, **Name** and **name** are two different variables.

JavaScript Reserved Words

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

abstract	else	instanceof	switch
boolean	enum	int	synchronized

break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with

double	in	super

Functions

Defining functions

Function declarations

A **function definition** (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, { /* … */ }.

For example, the following code defines a simple function named square:

```
function square(number) {
  return number * number;
}
```

Copy to Clipboard

The function square takes one parameter, called number. The function consists of one statement that says to return the parameter of the function (that is, number) multiplied by itself. The statement return specifies the value returned by the function:

```
return number * number;
```

Copy to Clipboard

Parameters are essentially passed to functions **by value** — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, **the change is not reflected globally or in the code which called that function**.

When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}

const mycar = {
  make: 'Honda',
  model: 'Accord',
  year: 1998,
};

// x gets the value "Honda"
const x = mycar.make;

// the make property is changed by the function
myFunc(mycar);
// y gets the value "Toyota"
const y = mycar.make;
```
Copy to Clipboard

When you pass an array as a parameter, if the function changes any of the array's values, that change is visible outside the function, as shown in the following example:

```
function myFunc(theArr) {
  theArr[0] = 30;
}

const arr = [45];

console.log(arr[0]); // 45
myFunc(arr);
console.log(arr[0]); // 30
```

Copy to Clipboard

Function expressions

While the function declaration above is syntactically a statement, functions can also be created by a function expression.

Such a function can be **anonymous**; it does not have to have a name. For example, the function square could have been defined as:

```
const square = function (number) {
  return number * number;
}
const x = square(4); // x gets the value 16
```

Copy to Clipboard

However, a name *can* be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
}

console.log(factorial(3))
```

Copy to Clipboard

Function expressions are convenient when passing a function as an argument to another function. The following example shows a map function that should receive a function as first argument and an array as second argument:

```
function map(f, a) {
  const result = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
  result[i] = f(a[i]);
  }
  return result;
}
```

Copy to Clipboard

In the following code, the function receives a function defined by a function expression and executes it for every element of the array received as a second argument:

```
function map(f, a) {
  const result = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
  result[i] = f(a[i]);
  }
  return result;
}
```

```
const f = function (x) {
  return x * x * x;
}

const numbers = [0, 1, 2, 5, 10];
const cube = map(f, numbers);
console.log(cube);
```
Copy to Clipboard

Function returns: [0, 1, 8, 125, 1000].

In JavaScript, a function can be defined based on a condition. For example, the following function definition defines myFunc only if num equals 0:

```
let myFunc;
if (num === 0) {
  myFunc = function (theObject) {
    theObject.make = 'Toyota';
  }
}
```
Copy to Clipboard

In addition to defining functions as described here, you can also use the Function constructor to create functions from a string at runtime, much like eval().

A **method** is a function that is a property of an object. Read more about objects and methods in Working with objects.

Calling functions

Defining a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

```
square(5);
```
Copy to Clipboard

The preceding statement calls the function with an argument of 5. The function executes its statements and returns the value 25.

Functions must be *in scope* when they are called, but the function declaration can be hoisted (appear below the call in the code). The scope of a function declaration is the function in which it is declared (or the entire program, if it is declared at the top level).

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function. The showProps() function (defined in Working with objects) is an example of a function that takes an object as an argument.

A function can call itself. For example, here is a function that computes factorials recursively:

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Copy to Clipboard

You could then compute the factorials of 1 through 5 as follows:

```
const a = factorial(1); // a gets the value 1
const b = factorial(2); // b gets the value 2
const c = factorial(3); // c gets the value 6
const d = factorial(4); // d gets the value 24
const e = factorial(5); // e gets the value 120
```

Copy to Clipboard

There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime.

It turns out that *functions are themselves objects* — and in turn, these objects have methods. (See the Function object.) The call() and apply() methods can be used to achieve this goal.

Function hoisting

Consider the example below:

```
console.log(square(5)); // 25


function square(n) {
  return n * n;
}
```

Copy to Clipboard

This code runs without any error, despite the square() function being called before it's declared. This is because the JavaScript interpreter hoists the entire function declaration to the top of the current scope, so the code above is equivalent to:

```
// All function declarations are effectively at the top of the scope
```

```
function square(n) {
  return n * n;
}

console.log(square(5)); // 25
```

Copy to Clipboard

Function hoisting only works with function *declarations* — not with function *expressions*. The code below will not work.

```
console.log(square); // ReferenceError:  Cannot  access 'square' before initialization
const square = function (n) {
  return n * n;
}
```

Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.

```
// The following variables are defined in the global scope
const num1 = 20;
const num2 = 3;
const name = 'Chamakh';

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}

multiply(); // Returns 60

// A nested function example
function getScore() {
  const num1 = 2;
  const num2 = 3;

  function add() {
    return `${name} scored ${num1 + num2}`;
  }
  return add();
}

getScore(); // Returns "Chamakh scored 5"
```

Copy to Clipboard

Scope and the function stack

Recursion

A function can refer to and call itself. There are three ways for a function to refer to itself:

1. The function's name
2. arguments.callee
3. An in-scope variable that refers to the function

For example, consider the following function definition:

```
const foo = function bar() {
  // statements go here
}
```

Copy to Clipboard

Within the function body, the following are all equivalent:

1. bar()
2. arguments.callee()

3. foo()

A function that calls itself is called a *recursive function*. In some ways, recursion is analogous to a loop. Both execute the same code multiple times, and both require a condition (to avoid an infinite loop, or rather, infinite recursion in this case).

For example, consider the following loop:

```
let x = 0;
while (x < 10) { // "x < 10" is the loop condition
  // do stuff
  x++;
}
```

Copy to Clipboard

It can be converted into a recursive function declaration, followed by a call to that function:

```
function loop(x) {
  // "x >= 10" is the exit condition (equivalent to "!(x < 10)")
  if (x >= 10) {
    return;
  }
  // do stuff
  loop(x + 1); // the recursive call
}
loop(0);
```
Copy to Clipboard

However, some algorithms cannot be simple iterative loops. For example, getting all the nodes of a tree structure (such as the DOM) is easier via recursion:

```
function walkTree(node) {
  if (node === null) {
  return;
  }
  // do something with node
  for (let i = 0; i < node.childNodes.length; i++) {
    walkTree(node.childNodes[i]);
  }
}
```
Copy to Clipboard

Compared to the function loop, each recursive call itself makes many recursive calls here.

It is possible to convert any recursive algorithm to a non-recursive one, but the logic is often much more complex, and doing so requires the use of a stack.

In fact, recursion itself uses a stack: the function stack. The stack-like behavior can be seen in the following example:

```
function foo(i) {
  if (i < 0) {
  return;
  }
  console.log(`begin: ${i}`);
  foo(i - 1);
  console.log(`end: ${i}`);
}
foo(3);

// Logs:
// begin: 3
// begin: 2
// begin: 1
// begin: 0
// end: 0
// end: 1
// end: 2
// end: 3
```

Copy to Clipboard

Nested functions and closures

You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.

It also forms a *closure*. A closure is an expression (most commonly, a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

To summarize:

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
function addSquares(a, b) {
  function square(x) {
  return x * x;
  }
  return square(a) + square(b);
}
const a = addSquares(2, 3); // returns 13
const b = addSquares(3, 4); // returns 25
const c = addSquares(4, 5); // returns 41
```

Copy to Clipboard

Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
  function inside(y) {
  return x + y;
  }
  return inside;
}
const fnInside = outside(3); // Think of it like: give me a function that adds
3 to whatever you give it
const result = fnInside(5); // returns 8
const result1 = outside(3)(5); // returns 8
```

Copy to Clipboard

Preservation of variables

Notice how x is preserved when inside is returned. A closure must preserve the arguments and variables in all scopes it references. Since each call provides potentially different arguments, a new closure is created for each call to outside. The memory can be freed only when the returned inside is no longer accessible.

This is not different from storing references in other objects, but is often less obvious because one does not set the references directly and cannot inspect them.

Multiply-nested functions

Functions can be multiply-nested. For example:

- A function (A) contains a function (B), which itself contains a function (C).
- Both functions B and C form closures here. So, B can access A, and C can access B.
- In addition, since C can access B which can access A, C can also access A.

Thus, the closures can contain multiple scopes; they recursively contain the scope of the functions containing it. This is called *scope chaining*. (The reason it is called "chaining" is explained later.)

Consider the following example:

```
function A(x) {
  function B(y) {
  function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // Logs 6 (which is 1 + 2 + 3)
```

Copy to Clipboard

In this example, C accesses B's y and A's x.

This can be done because:

1. B forms a closure including A (i.e., B can access A's arguments and variables).
2. C forms a closure including B.
3. Because C's closure includes B and B's closure includes A, then C's closure also includes A. This means C can access *both* B *and* A's arguments and variables. In other words, C *chains* the scopes of B and A, *in that order*.

The reverse, however, is not true. A cannot access C, because A cannot access any argument or variable of B, which C is a variable of. Thus, C remains private to only B.

Name conflicts

When two arguments or variables in the scopes of a closure have the same name, there is a *name conflict*. More nested scopes take precedence. So, the innermost scope takes the highest precedence, while the outermost scope takes the lowest. This is the scope chain. The first on the chain is the innermost scope, and the last is the outermost scope. Consider the following:

```
function outside() {
  const x = 5;
  function inside(x) {
  return x * 2;
  }
  return inside;
}

outside()(10); // returns 20 instead of 10
```

Copy to Clipboard

The name conflict happens at the statement return x * 2 and is between inside's parameter x and outside's variable x. The scope chain here is {inside, outside, global object}. Therefore, inside's x takes precedences over outside's x, and 20 (inside's x) is returned instead of 10 (outside's x).

Expressions and operators

- « Previous
- Next »

This chapter describes JavaScript's expressions and operators, including assignment, comparison, arithmetic, bitwise, logical, string, ternary and more.

At a high level, an *expression* is a valid unit of code that resolves to a value. There are two types of expressions: those that have side effects (such as assigning values) and those that purely *evaluate*.

The expression x = 7 is an example of the first type. This expression uses the = *operator* to assign the value seven to the variable x. The expression itself evaluates to 7.

The expression 3 + 4 is an example of the second type. This expression uses the + operator to add 3 and 4 together and produces a value, 7.

However, if it's not eventually part of a bigger construct (for example, a variable declaration like const z = 3 + 4), its result will be immediately discarded — this is usually a programmer mistake because the evaluation doesn't produce any effects.

As the examples above also illustrate, all complex expressions are joined by *operators*, such as = and +. In this section, we will introduce the following operators:

- Assignment operators
- Comparison operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- BigInt operators
- String operators
- Conditional (ternary) operator
- Comma operator
- Unary operators
- Relational operators

These operators join operands either formed by higher-precedence operators or one of the basic expressions. A complete and detailed list of operators and expressions is also available in the reference.

The *precedence* of operators determines the order they are applied when evaluating an expression. For example:

```
const x = 1 + 2 * 3;
const y = 2 * 3 + 1;
```
Copy to Clipboard

Despite * and + coming in different orders, both expressions would result in 7 because * has precedence over +, so the *-joined expression will always be evaluated first. You can override operator precedence by using parentheses (which creates a grouped expression — the basic expression). To see a complete table of operator precedence as well as various caveats, see the Operator Precedence Reference page.

JavaScript has both *binary* and *unary* operators, and one special ternary operator, the conditional operator. A binary operator requires two operands, one before the operator and one after the operator:

```
operand1 operator operand2
```
Copy to Clipboard

For example, 3 + 4 or x * y. This form is called an *infix* binary operator, because the operator is placed between two operands. All binary operators in JavaScript are infix.

A unary operator requires a single operand, either before or after the operator:

operator operand
operand operator
Copy to Clipboard

For example, x++ or ++x. The operator operand form is called a *prefix* unary operator, and the operand operator form is called a *postfix* unary operator. ++ and -- are the only postfix operators in JavaScript — all other operators, like !, typeof, etc. are prefix.

Assignment operators

An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = f() is an assignment expression that assigns the value of f() to x.

There are also compound assignment operators that are shorthand for the operations listed in the following table:

Name	**Shorthand operator**	**Meaning**
Assignment	x = f()	x = f()
Addition assignment	x += f()	x = x + f()

Name	Shorthand operator	Meaning
Subtraction assignment	x -= f()	x = x - f()
Multiplication assignment	x *= f()	x = x * f()
Division assignment	x /= f()	x = x / f()
Remainder assignment	x %= f()	x = x % f()
Exponentiation assignment	x **= f()	x = x ** f()
Left shift assignment	x <<= f()	x = x << f()
Right shift assignment	x >>= f()	x = x >> f()
Unsigned right shift assignment	x >>>= f()	x = x >>> f()
Bitwise AND assignment	x &= f()	x = x & f()
Bitwise XOR assignment	x ^= f()	x = x ^ f()
Bitwise OR assignment	x |= f()	x = x | f()
Logical AND assignment	x &&= f()	x && (x = f())
Logical OR assignment	x ||= f()	x || (x = f())
Nullish coalescing assignment	x ??= f()	x ?? (x = f())

Assigning to properties

If an expression evaluates to an object, then the left-hand side of an assignment expression may make assignments to properties of that expression. For example:

```
const obj = {};

obj.x = 3;
console.log(obj.x); // Prints 3.
console.log(obj); // Prints { x: 3 }.

const key = "y";
obj[key] = 5;
console.log(obj[key]); // Prints 5.
console.log(obj); // Prints { x: 3, y: 5 }.
```

Copy to Clipboard

For more information about objects, read Working with Objects.

If an expression does not evaluate to an object, then assignments to properties of that expression do not assign:

```
const val = 0;
val.x = 3;
```

```
console.log(val.x); // Prints undefined.
console.log(val); // Prints 0.
```
Copy to Clipboard

In strict mode, the code above throws, because one cannot assign properties to primitives.

It is an error to assign values to unmodifiable properties or to properties of an expression without properties (null or undefined).

Destructuring

For more complex assignments, the destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.

```
const foo = ['one', 'two', 'three'];

// without destructuring
const one   = foo[0];
const two   = foo[1];
const three = foo[2];

// with destructuring
const [one, two, three] = foo;
```

Copy to Clipboard

Evaluation and nesting

In general, assignments are used within a variable declaration (i.e., with const, let, or var) or as standalone statements).

```
// Declares a variable x and initializes it to the result of f().
// The result of the x = f() assignment expression is discarded.
let x = f();

x = g(); // Reassigns the variable x to the result of g().
```

Copy to Clipboard

However, like other expressions, assignment expressions like x = f() evaluate into a result value. Although this result value is usually not used, it can then be used by another expression.

Chaining assignments or nesting assignments in other expressions can result in surprising behavior. For this reason, some JavaScript style guides discourage chaining or nesting assignments). Nevertheless, assignment chaining and nesting may occur sometimes, so it is important to be able to understand how they work.

By chaining or nesting an assignment expression, its result can itself be assigned to another variable. It can be logged, it can be put inside an array literal or function call, and so on.

```
let x;
const y = (x = f()); // Or equivalently: const y = x = f();
console.log(y); // Logs the return value of the assignment x = f().

console.log(x = f()); // Logs the return value directly.

// An assignment expression can be nested in any place
// where expressions are generally allowed,
// such as array literals' elements or as function calls' arguments.
console.log([ 0, x = f(), 0 ]);
console.log(f(0, x = f(), 0));
```

Copy to Clipboard

The evaluation result matches the expression to the right of the = sign in the "Meaning" column of the table above. That means that x = f() evaluates into whatever f()'s result is, x += f() evaluates into the resulting sum x + f(), x **= f() evaluates into the resulting power x ** y, and so on.

In the case of logical assignments, x &&= f(), x ||= f(), and x ??= f(), the return value is that of the logical operation without the assignment, so x && f(), x || f(), and x ?? f(), respectively.

When chaining these expressions without parentheses or other grouping operators like array literals, the assignment expressions are **grouped right to left** (they are right-associative), but they are **evaluated left to right**.

Note that, for all assignment operators other than = itself, the resulting values are always based on the operands' values *before* the operation.

For example, assume that the following functions f and g and the variables x and y have been declared:

```
function f () {
  console.log('F!');
  return 2;
}
function g () {
  console.log('G!');
  return 3;
}
let x, y;
```

Copy to Clipboard

Consider these three examples:

```
y = x = f()
y = [ f(), x = g() ]
x[f()] = g()
```

Copy to Clipboard

Evaluation example 1

y = x = f() is equivalent to y = (x = f()), because the assignment operator = is right-associative. However, it evaluates from left to right:

1. The assignment expression y = x = f() starts to evaluate.

i. The y on this assignment's left-hand side evaluates into a reference to the variable named y.

ii. The assignment expression x = f() starts to evaluate.

i. The x on this assignment's left-hand side evaluates into a reference to the variable named x.

ii. The function call f() prints "F!" to the console and then evaluates to the number 2.

iii. That 2 result from f() is assigned to x.

iii. The assignment expression x = f() has now finished evaluating; its result is the new value of x, which is 2.

iv. That 2 result in turn is also assigned to y.

2. The assignment expression y = x = f() has now finished evaluating; its result is the new value of y – which happens to be 2. x and y are assigned to 2, and the console has printed "F!".

Evaluation example 2

y = [f(), x = g()] also evaluates from left to right:

1. The assignment expression y = [f(), x = g()] starts to evaluate.

i. The y on this assignment's left-hand evaluates into a reference to the variable named y.

ii. The inner array literal [f(), x = g()] starts to evaluate.

i. The function call f() prints "F!" to the console and then evaluates to the number 2.

ii. The assignment expression x = g() starts to evaluate.

i. The x on this assignment's left-hand side evaluates into a reference to the variable named x.

ii. The function call g() prints "G!" to the console and then evaluates to the number 3.

iii. That 3 result from g() is assigned to x.

iii. The assignment expression x = g() has now finished evaluating; its result is the new value of x, which is 3. That 3 result becomes the next element in the inner array literal (after the 2 from the f()).

iii. The inner array literal [f(), x = g()] has now finished evaluating; its result is an array with two values: [2, 3].

iv. That [2, 3] array is now assigned to y.

2. The assignment expression y = [f(), x = g()] has now finished evaluating; its result is the new value of y – which happens to be [2, 3]. x is now assigned to 3, y is now assigned to [2, 3], and the console has printed "F!" then "G!".

Evaluation example 3

x[f()] = g() also evaluates from left to right. (This example assumes that x is already assigned to some object. For more information about objects, read Working with Objects.)

1. The assignment expression x[f()] = g() starts to evaluate.

i. The x[f()] property access on this assignment's left-hand starts to evaluate.

i. The x in this property access evaluates into a reference to the variable named x.

ii. Then the function call f() prints "F!" to the console and then evaluates to the number 2.

ii. The x[f()] property access on this assignment has now finished evaluating; its result is a variable property reference: x[2].

iii. Then the function call g() prints "G!" to the console and then evaluates to the number 3.

iv. That 3 is now assigned to x[2]. (This step will succeed only if x is assigned to an object.)

2. The assignment expression x[f()] = g() has now finished evaluating; its result is the new value of x[2] – which happens to be 3. x[2] is now assigned to 3, and the console has printed "F!" then "G!".

Avoid assignment chains

Chaining assignments or nesting assignments in other expressions can result in surprising behavior. For this reason, chaining assignments in the same statement is discouraged).

In particular, putting a variable chain in a const, let, or var statement often does *not* work. Only the outermost/leftmost variable would get declared; other variables within the assignment chain are *not* declared by the const/let/var statement. For example:

```
const z = y = x = f();
```

Copy to Clipboard

This statement seemingly declares the variables x, y, and z. However, it only actually declares the variable z. y and x are either invalid references to nonexistent variables (in strict mode) or, worse, would implicitly create global variables for x and y in sloppy mode.

Comparison operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared based on standard lexicographical ordering, using Unicode values. In most cases, if the two operands are not of the same type, JavaScript attempts to convert them to an appropriate type for the comparison. This behavior generally results in comparing the operands numerically. The sole exceptions to type conversion within comparisons involve the === and !== operators, which perform strict equality and inequality comparisons. These operators do not attempt to convert the operands to compatible types before checking equality. The following table describes the comparison operators in terms of this sample code:

```
const var1 = 3;
const var2 = 4;
```

Copy to Clipboard

Comparison operators

Operator	**Description**	**Examples returning true**
Equal (==)	Returns true if the operands are equal.	3 == var1 "3" == var1 3 == '3'

Comparison operators

Operator	**Description**	**Examples returning true**
Not equal (!=)	Returns true if the operands are not equal.	var1 != 4 var2 != "3"
Strict equal (===)	Returns true if the operands are equal and of the same type. See also Object.is and sameness in JS.	3 === var1
Strict not equal (!==)	Returns true if the operands are of the same type but not equal, or are of different type.	var1 !== "3" 3 !== '3'
Greater than (>)	Returns true if the left operand is greater than the right operand.	var2 > var1 "12" > 2
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	var2 >= var1 var1 >= 3
Less than (<)	Returns true if the left operand is less than the right operand.	var1 < var2 "2" < 12
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	var1 <= var2 var2 <= 5

Note: => is not a comparison operator but rather is the notation for Arrow functions.

Arithmetic operators

An arithmetic operator takes numerical values (either literals or variables) as their operands and returns a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (*), and division (/). These operators work as they do in most other programming languages when used with floating point numbers (in particular, note that division by zero produces Infinity). For example:

```
1 / 2; // 0.5
1 / 2 === 1.0 / 2.0; // this is true
```

Copy to Clipboard

In addition to the standard arithmetic operations (+, -, *, /), JavaScript provides the arithmetic operators listed in the following table:

Arithmetic operators

Operator	**Description**	**Example**
Remainder (%)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
Increment (++)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand	If x is 3, then ++x sets x to 4 and returns 4,

Arithmetic operators

Operator	**Description**	**Example**
	after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	whereas x++ returns 3 and, only then, sets x to 4.
Decrement (--)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2.
Unary negation (-)	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.
Unary plus (+)	Unary operator. Attempts to convert the operand to a number, if it is not already.	+"3" returns 3. +true returns 1.
Exponentiation operator (**)	Calculates the base to the exponent power, that is, base^exponent	2 ** 3 returns 8. 10 ** -1 returns 0.1.

Bitwise operators

A bitwise operator treats their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.

The following table summarizes JavaScript's bitwise operators.

Operator	**Usage**	**Description**
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a zero in each bit position for which the corresponding bits of both operands are zeros.
Bitwise XOR	a ^ b	Returns a zero in each bit position for which the corresponding bits are the same. [Returns a one in each bit position for which the corresponding bits are different.]
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts a in binary representation b bits to the left, shifting in zeros from the right.
Sign-propagating right shift	a >> b	Shifts a in binary representation b bits to the right, discarding bits shifted off.

Operator	Usage	Description
Zero-fill right shift	a >>> b	Shifts a in binary representation b bits to the right, discarding bits shifted off, and shifting in zeros from the left.

Bitwise logical operators

Conceptually, the bitwise logical operators work as follows:

- The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones). Numbers with more than 32 bits get their most significant bits discarded. For example, the following integer with more than 32 bits will be converted to a 32-bit integer:
- Before: 1110 0110 1111 1010 0000 0000 0000 0110 0000 0000 0001
- After: 1010 0000 0000 0000 0110 0000 0000 0001

Copy to Clipboard

- Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.
- The operator is applied to each pair of bits, and the result is constructed bitwise.

For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111. So, when the bitwise operators are applied to these values, the results are as follows:

Expression	**Result**	**Binary Description**
15 & 9	9	1111 & 1001 = 1001
15 | 9	15	1111 | 1001 = 1111
15 ^ 9	6	1111 ^ 1001 = 0110
~15	-16	~ 0000 0000 … 0000 1111 = 1111 1111 … 1111 0000
~9	-10	~ 0000 0000 … 0000 1001 = 1111 1111 … 1111 0110

Note that all 32 bits are inverted using the Bitwise NOT operator, and that values with the most significant (left-most) bit set to 1 represent negative numbers (two's-complement representation). ~x evaluates to the same value that -x - 1 evaluates to.

Bitwise shift operators

The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.

Shift operators convert their operands to thirty-two-bit integers and return a result of either type Number or BigInt: specifically, if the type of the left operand is BigInt, they return BigInt; otherwise, they return Number.

The shift operators are listed in the following table.

Bitwise shift operators

Operator	**Description**	**Example**
Left shift (<<)	This operator shifts the first operand the specified number of bits to the left. Excess bits shifted off to the left are discarded. Zero bits are shifted in from the right.	9<<2 yields 36, because 1001 shifted 2 bits to the left becomes 100100, which is 36.
Sign-propagating right shift (>>)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Copies of the leftmost bit are shifted in from the left.	9>>2 yields 2, because 1001 shifted 2 bits to the right becomes 10, which is 2. Likewise, -9>>2 yields -3, because the sign is preserved.

Bitwise shift operators

Operator	Description	Example
Zero-fill right shift (>>>)	This operator shifts the first operand the specified number of bits to the right. Excess bits shifted off to the right are discarded. Zero bits are shifted in from the left.	19>>>2 yields 4, because 10011 shifted 2 bits to the right becomes 100, which is 4. For non-negative numbers, zero-fill right shift and sign-propagating right shift yield the same result.

Logical operators

Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators actually return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value. The logical operators are described in the following table.

Logical operators

Operator	Usage	Description
Logical AND (&&)	expr1 && expr2	Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.

Logical operators

Operator	Usage	Description
Logical OR (||)	expr1 || expr2	Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, || returns true if either operand is true; if both are false, returns false.
Logical NOT (!)	!expr	Returns false if its single operand that can be converted to true; otherwise, returns true.

Examples of expressions that can be converted to false are those that evaluate to null, 0, NaN, the empty string (""), or undefined.

The following code shows examples of the && (logical AND) operator.

```
const a1 = true && true; // t && t returns true
const a2 = true && false; // t && f returns false
const a3 = false && true; // f && t returns false
const a4 = false && (3 === 4); // f && f returns false
const a5 = 'Cat' && 'Dog'; // t && t returns Dog
const a6 = false && 'Cat'; // f && t returns false
const a7 = 'Cat' && false; // t && f returns false
```

Copy to Clipboard

The following code shows examples of the || (logical OR) operator.

```
const o1 = true || true; // t || t returns true
const o2 = false || true; // f || t returns true
const o3 =  true || false; // t || f returns true
const o4 = false || (3 === 4); // f || f returns false
const o5 = 'Cat' || 'Dog'; // t || t returns Cat
const o6 = false || 'Cat'; // f || t returns Cat
const o7 = 'Cat' || false; // t || f returns Cat
```

Copy to Clipboard

The following code shows examples of the ! (logical NOT) operator.

```
const n1 = !true; // !t returns false
const n2 = !false; // !f returns true
const n3 = !'Cat'; // !t returns false
```

Copy to Clipboard

Short-circuit evaluation

As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:

- false && anything is short-circuit evaluated to false.
- true || anything is short-circuit evaluated to true.

The rules of logic guarantee that these evaluations are always correct. Note that the *anything* part of the above expressions is not evaluated, so any side effects of doing so do not take effect.

Note that for the second case, in modern code you can use the Nullish coalescing operator (??) that works like ||, but it only returns the second expression, when the first one is "nullish", i.e. null or undefined. It is thus the better alternative to provide defaults, when values like '' or 0 are valid values for the first expression, too.

BigInt operators

Most operators that can be used between numbers can be used between BigInt values as well.

```
// BigInt addition
const a = 1n + 2n; // 3n
// Division with BigInts round towards zero
const b = 1n / 2n; // 0n
// Bitwise operations with BigInts do not truncate either side
const c = 40000000000000000n >> 2n; // 10000000000000000n
```

Copy to Clipboard

One exception is unsigned right shift (>>>), which is not defined for BigInt values. This is because a BigInt does not have a fixed width, so technically it does not have a "highest bit".

```
const d = 8n >>> 2n; // TypeError: BigInts have no unsigned right shift, use >> instead
```
Copy to Clipboard

BigInts and numbers are not mutually replaceable — you cannot mix them in calculations.

```
const a = 1n + 2; // TypeError: Cannot mix BigInt and other types
```

This is because BigInt is neither a subset nor a superset of numbers. BigInts have higher precision than numbers when representing large integers, but cannot represent decimals, so implicit conversion on either side might lose precision. Use explicit conversion to signal whether you wish the operation to be a number operation or a BigInt one.

```
const a = Number(1n) + 2; // 3
const b = 1n + BigInt(2); // 3n
```

You can compare BigInts with numbers.

```
const a = 1n > 2; // false
const b = 3 > 2n; // true
```
Copy to Clipboard

String operators

In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.

For example,

```
console.log('my ' + 'string'); // console logs the string "my string".
```

Copy to Clipboard

The shorthand assignment operator += can also be used to concatenate strings.

For example,

```
let mystring = 'alpha';
mystring += 'bet'; // evaluates to "alphabet" and assigns this value to mystring.
```

Copy to Clipboard

Conditional (ternary) operator

The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition. The syntax is:

```
condition ? val1 : val2
```

Copy to Clipboard

If condition is true, the operator has the value of val1. Otherwise it has the value of val2. You can use the conditional operator anywhere you would use a standard operator.

For example,

```
const status = age >= 18 ? 'adult' : 'minor';
```

Copy to Clipboard

This statement assigns the value "adult" to the variable status if age is eighteen or more. Otherwise, it assigns the value "minor" to status.

Comma operator

The comma operator (,) evaluates both of its operands and returns the value of the last operand. This operator is primarily used inside a for loop, to allow multiple variables to be updated each time through the loop. It is regarded bad style to use it elsewhere, when it is not necessary. Often two separate statements can and should be used instead.

For example, if a is a 2-dimensional array with 10 elements on a side, the following code uses the comma operator to update two variables at once. The code prints the values of the diagonal elements in the array:

```
const x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const a = [x, x, x, x, x];

for (let i = 0, j = 9; i <= j; i++, j--) {
//                                ^
  console.log(`a[${i}][${j}]= ${a[i][j]}`);
}
```

Copy to Clipboard

JavaScript Events

What is an Event ?

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

Please go through this small tutorial for a better understanding HTML Event Reference. Here we will see a few examples to understand a relation between Event and JavaScript −

onclick Event Type

This is the most frequently used event type which occurs when a user clicks the left button of his mouse. You can put your validation, warning etc., against this event type.

Example

Try the following example.

```
<html>
   <head>
      <script type = "text/javascript">
         <!--
            function sayHello() {
               alert("Hello World")
            }
         //-->
      </script>
   </head>
```

```
  <body>
     <p>Click the following button and see result</p>
     <form>
        <input type = "button" onclick = "sayHello()" value = "Say Hello"
/>
     </form>
  </body>
</html>
```

Output

onsubmit Event Type

onsubmit is an event that occurs when you try to submit a form. You can put your form validation against this event type.

Example

The following example shows how to use onsubmit. Here we are calling a **validate()** function before submitting a form data to the webserver. If **validate()** function returns true, the form will be submitted, otherwise it will not submit the data.

Try the following example.

```
<html>
  <head>
     <script type = "text/javascript">
```

```
         <!--
           function validation() {
              all validation goes here
              .........
              return either true or false
           }
         //-->
      </script>
   </head>

   <body>
      <form   method   =   "POST"   action   =   "t.cgi"   onsubmit   =   "return
validate()">
         .......
         <input type = "submit" value = "Submit" />
      </form>
   </body>
</html>
```

onmouseover and onmouseout

These two event types will help you create nice effects with images or even with text as well. The **onmouseover** event triggers when you bring your mouse over any element and the **onmouseout** triggers when you move your mouse out from that element. Try the following example.

```
<html>
   <head>
      <script type = "text/javascript">
         <!--
            function over() {
               document.write ("Mouse Over");
            }
            function out() {
               document.write ("Mouse Out");
            }
         //-->
      </script>
   </head>

   <body>
      <p>Bring your mouse inside the division to see the result:</p>
      <div onmouseover = "over()" onmouseout = "out()">
         <h2> This is inside the division </h2>
      </div>
   </body>
</html>
```

Output

HTML 5 Standard Events

The standard HTML 5 events are listed here for your reference. Here script indicates a Javascript function to be executed against that event.

Attribute	**Value**	**Description**
Offline	script	Triggers when the document goes offline
Onabort	script	Triggers on an abort event
onafterprint	script	Triggers after the document is printed
onbeforeonload	script	Triggers before the document loads
onbeforeprint	script	Triggers before the document is printed
Onblur	script	Triggers when the window loses focus
oncanplay	script	Triggers when media can start play, but might has to stop for buffering
oncanplaythrough	script	Triggers when media can be played to the end, without stopping for buffering

onchange	script	Triggers when an element changes
Onclick	script	Triggers on a mouse click
oncontextmenu	script	Triggers when a context menu is triggered
ondblclick	script	Triggers on a mouse double-click
Ondrag	script	Triggers when an element is dragged
ondragend	script	Triggers at the end of a drag operation
ondragenter	script	Triggers when an element has been dragged to a valid drop target
ondragleave	script	Triggers when an element is being dragged over a valid drop target
ondragover	script	Triggers at the start of a drag operation
ondragstart	script	Triggers at the start of a drag operation
Ondrop	script	Triggers when dragged element is being dropped

ondurationchange	script	Triggers when the length of the media is changed
onemptied	script	Triggers when a media resource element suddenly becomes empty.
onended	script	Triggers when media has reach the end
onerror	script	Triggers when an error occur
onfocus	script	Triggers when the window gets focus
onformchange	script	Triggers when a form changes
onforminput	script	Triggers when a form gets user input
onhaschange	script	Triggers when the document has change
oninput	script	Triggers when an element gets user input
oninvalid	script	Triggers when an element is invalid
onkeydown	script	Triggers when a key is pressed
onkeypress	script	Triggers when a key is pressed and released

onkeyup	script	Triggers when a key is released
Onload	script	Triggers when the document loads
onloadeddata	script	Triggers when media data is loaded
onloadedmetadata	script	Triggers when the duration and other media data of a media element is loaded
onloadstart	script	Triggers when the browser starts to load the media data
onmessage	script	Triggers when the message is triggered
onmousedown	script	Triggers when a mouse button is pressed
onmousemove	script	Triggers when the mouse pointer moves
onmouseout	script	Triggers when the mouse pointer moves out of an element
onmouseover	script	Triggers when the mouse pointer moves over an element
onmouseup	script	Triggers when a mouse button is released

onmousewheel	script	Triggers when the mouse wheel is being rotated
onoffline	script	Triggers when the document goes offline
Onoine	script	Triggers when the document comes online
ononline	script	Triggers when the document comes online
onpagehide	script	Triggers when the window is hidden
onpageshow	script	Triggers when the window becomes visible
onpause	script	Triggers when media data is paused
Onplay	script	Triggers when media data is going to start playing
onplaying	script	Triggers when media data has start playing
onpopstate	script	Triggers when the window's history changes
onprogress	script	Triggers when the browser is fetching the media data

onratechange	script	Triggers when the media data's playing rate has changed
onreadystatechange	script	Triggers when the ready-state changes
Onredo	script	Triggers when the document performs a redo
onresize	script	Triggers when the window is resized
onscroll	script	Triggers when an element's scrollbar is being scrolled
onseeked	script	Triggers when a media element's seeking attribute is no longer true, and the seeking has ended
onseeking	script	Triggers when a media element's seeking attribute is true, and the seeking has begun
onselect	script	Triggers when an element is selected
onstalled	script	Triggers when there is an error in fetching media data

onstorage	script	Triggers when a document loads
onsubmit	script	Triggers when a form is submitted
onsuspend	script	Triggers when the browser has been fetching media data, but stopped before the entire media file was fetched
ontimeupdate	script	Triggers when media changes its playing position
onundo	script	Triggers when a document performs an undo
onunload	script	Triggers when the user leaves the document
onvolumechange	script	Triggers when media changes the volume, also when volume is set to "mute"
onwaiting	script	Triggers when media has stopped playing, but is expected to resume

Debugging JavaScript in Browser

You can and will encounter errors while writing programs. Errors are not necessarily bad. In fact, most of the time, they help us identify issues with our code. It is essential that you know how to debug your code and fix errors.

Debugging is the process of examining the program, finding the error and fixing it.

There are different ways you can debug your JavaScript program.

---

1. Using console.log()

You can use the `console.log()` method to debug the code. You can pass the value you want to check into the `console.log()` method and verify if the data is correct.

The syntax is:

```
console.log(object/message);
```

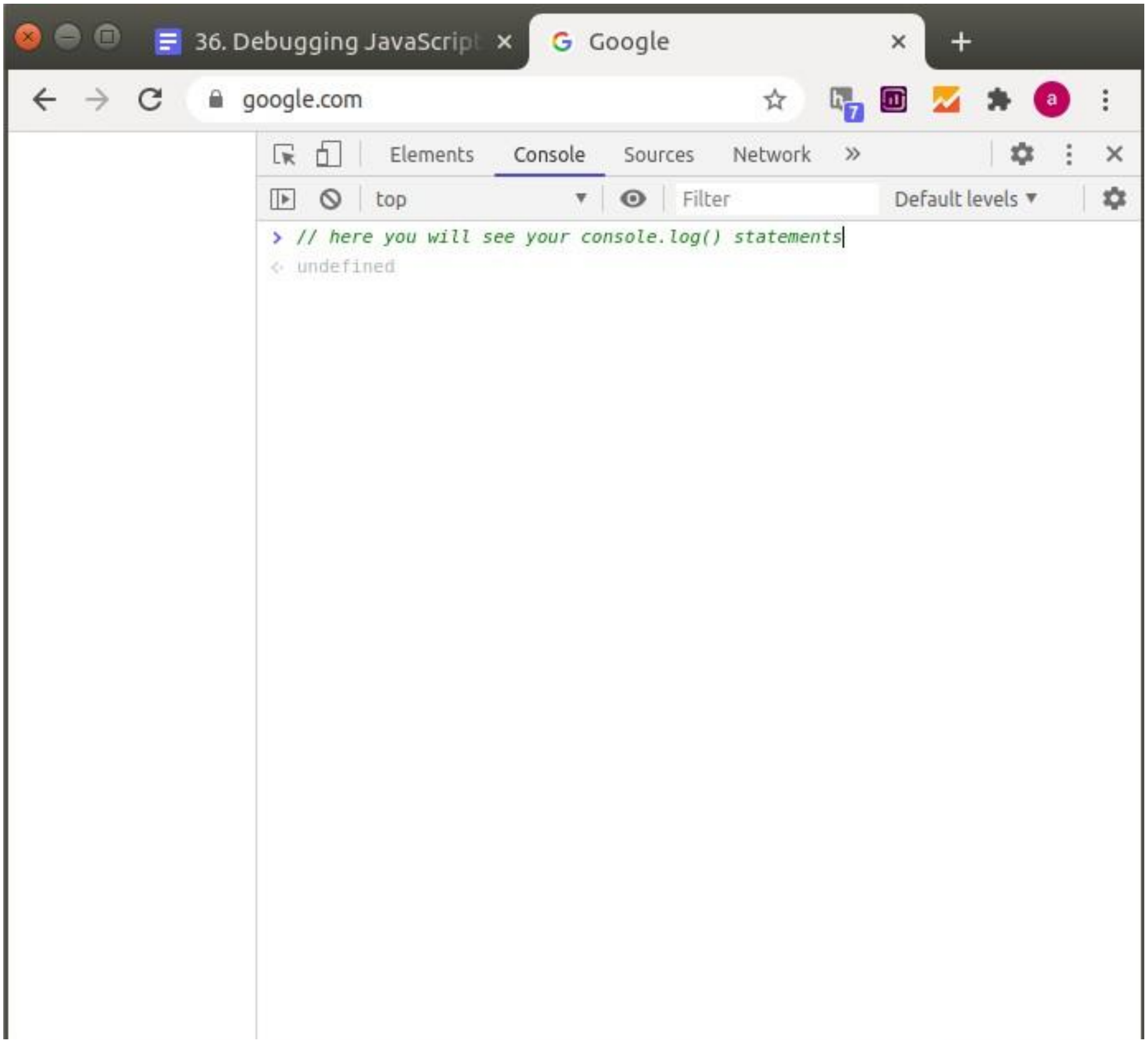
You could pass the object in `console.log()` or simply a message string.

In the previous tutorial, we used `console.log()` method to print the output. However, you can also use this method for debugging. For example,

```
let a = 5;

let b = 'asdf';

let c = a + b;
```

```
// if you want to see the value of c
console.log(c);

// then do other operations
if(c) {
    // do something
}
```

Using `console.log()` method in the browser opens the value in the debugger window.



```
> // here you will see your console.log() statements
<· undefined
```

Working of console.log() method in browser

The console.log() is not specific to browsers. It's also available in other JavaScript engines.

2. Using debugger

The debugger keyword stops the execution of the code and calls the debugging function.

The debugger is available in almost all JavaScript engines.

Let's see an example,

```
let a = 6;

let b = 9;

let c = a * b;


// stops the execution

debugger;


console.log(c);
```

JavaScript Objects

JavaScript object is a non-primitive data-type that allows you to store multiple collections of data.

Note: If you are familiar with other programming languages, JavaScript objects are a bit different. You do not need to create classes in order to create objects.

Here is an example of a JavaScript object.

```
// object
const student = {
    firstName: 'ram',
    class: 10
};
```

Here, student is an object that stores values such as strings and numbers.

JavaScript Object Declaration

The syntax to declare an object is:

```
const object_name = {
   key1: value1,
   key2: value2
}
```

Here, an object object_name is defined. Each member of an object is a **key: value** pair separated by commas and enclosed in curly braces {}.

For example,

```
// object creation
const person = {
    name: 'John',
    age: 20
};
console.log(typeof person); // object
```

Run Code

You can also define an object in a single line.

```
const person = { name: 'John', age: 20 };
```

In the above example, `name` and `age` are keys, and `John` and `20` are values respectively.

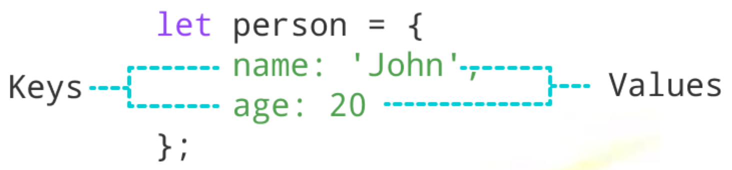
There are other ways to declare an object in JavaScript. To learn more, visit Different Ways to Declare JavaScript Objects.

JavaScript Object Properties

In JavaScript, "key: value" pairs are called **properties**. For example,

```
let person = {
    name: 'John',
    age: 20
};
```

Here, name: 'John' and age: 20 are properties.



JavaScript object properties

Accessing Object Properties

1. Using dot Notation

Here's the syntax of the dot notation.

```
objectName.key
```

For example,

```
const person = {
    name: 'John',
    age: 20,
};

// accessing property
console.log(person.name); // John
```

Run Code

---

2. Using bracket Notation

Here is the syntax of the bracket notation.

```
objectName["propertyName"]
```

For example,

```
const person = {
    name: 'John',
    age: 20,
};

// accessing property
console.log(person["name"]); // John
```

JavaScript and JSON

JSON stands for Javascript Object Notation. JSON is a text-based data format that is used to store and transfer data. For example,

```
// JSON syntax
```

```
{
    "name": "John",
    "age": 22,
    "gender": "male",

}
```

In JSON, the data are in **key/value** pairs separated by a comma `,`.

JSON was derived from JavaScript. So, the JSON syntax resembles JavaScript object literal syntax. However, the JSON format can be accessed and be created by other programming languages too.

Note: JavaScript Objects and JSON are not the same. You will learn about their differences later in this tutorial.

JSON Data

JSON data consists of **key/value** pairs similar to JavaScript object properties. The key and values are written in double quotes separated by a colon `:`. For example,

```
// JSON data
```

```
"name": "John"
```

Note: JSON data requires double quotes for the key.

JSON Object

The JSON object is written inside curly braces `{ }`. JSON objects can contain multiple **key/value** pairs. For example,

```
// JSON object
{ "name": "John", "age": 22 }
```

JSON Array

JSON array is written inside square brackets `[ ]`. For example,

```
// JSON array
[ "apple", "mango", "banana"]


// JSON array containing objects
```

```
[
   { "name": "John", "age": 22 },
   { "name": "Peter", "age": 20 }.
   { "name": "Mark", "age": 23 }
]
```

Note: JSON data can contain objects and arrays. However, unlike JavaScript objects, JSON data cannot contain functions as values.

Accessing JSON Data

You can access JSON data using the dot notation. For example,

```
// JSON object
const data = {
   "name": "John",
   "age": 22,
   "hobby": {
         "reading" : true,
         "gaming" : false,
         "sport" : "football"
   },
    "class" : ["JavaScript", "HTML", "CSS"]
}

// accessing JSON object
console.log(data.name); // John
console.log(data.hobby); // { gaming: false, reading: true, sport: "football"}

console.log(data.hobby.sport); // football
console.log(data.class[1]); // HTML
```

Run Code

We use the `.` notation to access JSON data. Its syntax is: `variableName.key`

You can also use square bracket syntax `[]` to access JSON data. For example,

```
// JSON object
const data = {
    "name": "John",
    "age": 22
}

// accessing JSON object
console.log(data["name"]); // John
```

Run Code

JavaScript Objects VS JSON

Though the syntax of JSON is similar to the JavaScript object, JSON is different from JavaScript objects.

JSON	JavaScript Object
The key in key/value pair should be in double quotes.	The key in key/value pair can without double quotes.
JSON cannot contain functions.	JavaScript objects can conta functions.
JSON can be created and used by other programming languages.	JavaScript objects can only be us in JavaScript.

Converting JSON to JavaScript Object

You can convert JSON data to a JavaScript object using the built-in JSON.parse() function. For example,

```
// json object
const jsonData = '{ "name": "John", "age": 22 }';

// converting to JavaScript object
const obj = JSON.parse(jsonData);

// accessing the data
console.log(obj.name); // John
```

Run Code

Converting JavaScript Object to JSON

You can also convert JavaScript objects to JSON format using the JavaScript built-in JSON.stringify() function. For example,

```
// JavaScript object
const jsonData = { "name": "John", "age": 22 };

// converting to JSON
```

```
const obj = JSON.stringify(jsonData);


// accessing the data
onsole.log(obj); // "{"name":"John","age":22}"
```