Chapter 1: Basics of HTML

HTML stands for Hyper Text Markup Language. It is the standard markup language for documents designed to be displayed in a web browser. It is not a programming language; it is a *markup language* used to tell your browser how to structure the web pages you visit. HTML consists of a series of elements, which you use to enclose, wrap, or *mark up* different parts of the content to make it appear or act a certain way. The enclosing tags can make a bit of content into a hyperlink to link to another page on the web, italicize words, and so on. Browsers do not display the HTML tags, but use them to render the content of the page.

HTML was created by Berners-Lee in late 1991 but "HTML 2.0" was the first standard HTML specification which was published in 1995. HTML 4.01 was a major version of HTML and it was published in late 1999. Though HTML 4.01 version is widely used but currently we are having HTML-5 version which is an extension to HTML 4.01, and this version was published in 2012.

A Simple HTML Document

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>My First Heading</h1>
<p>My first paragraph.</p>

</body>
</html>
```

The <!DOCTYPE html> declaration defines this document to be HTML5

- The <html> element is the root element of an HTML page
- The <head> element contains meta information about the document
- The <title> element specifies a title for the document
- The <body> element contains the visible page content
- The <h1> element defines a large heading
- The <p> element defines a paragraph

HTML Tags

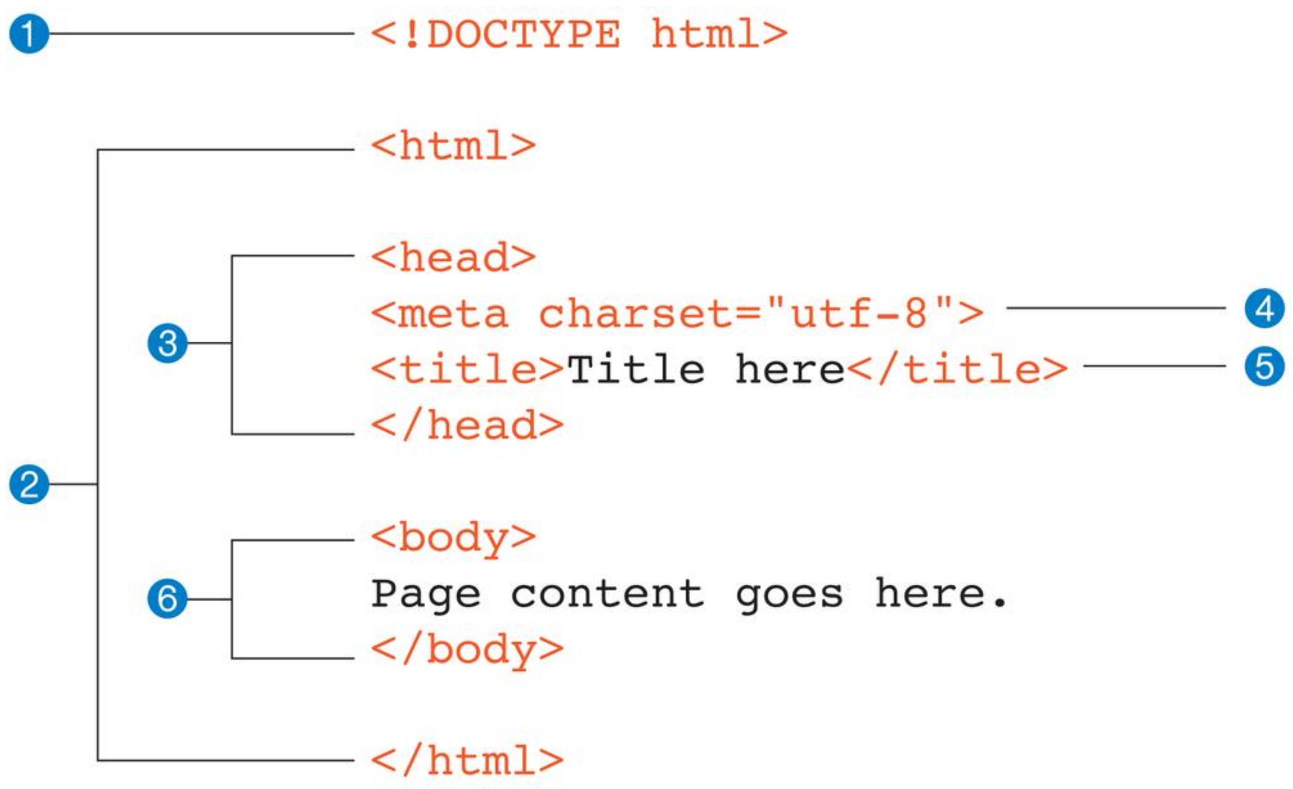
HTML is a markup language and makes use of various tags to format the content. These tags are enclosed within angle braces **<Tag Name>**. Except few tags, most of the tags have their corresponding closing tags. For example, **<html>** has its closing tag **</html>** and **<body>** tag has its closing tag **</body>** tag etc. HTML tags are element names surrounded by angle brackets:

<tagname> some content goes here ... </tagname>

- HTML tags normally come **in pairs** like <p> and </p>
- The first tag in a pair is the **start tag,** the second tag is the **end tag**

- The end tag is written like the start tag, but with a **forward slash** inserted before the tag name

HTML Document Structure



```
<!DOCTYPE html>

<html>

<head>
<meta charset="utf-8">
<title>Title here</title>
</head>

<body>
Page content goes here.
</body>

</html>
```

Source: https://www.oreilly.com/library/view/learning-web-design/9781449337513/ch04.html

1. The first line in the example isn't an element at all; it is a **document type declaration** (also called **DOCTYPE declaration**) that identifies this document as an HTML5 document. It lets modern browsers know they should interpret the document as written according to the HTML5 specification.
2. The entire document is contained within an html element. The html element is called the **root element** because it contains all the elements in the document, and it may not be contained within any other element.
3. Within the html element, the document is divided into a **head** and a **body**. The head element contains descriptive information about the document itself, such as its title, the style sheet(s) it uses, scripts, and other types of "meta" information.
4. The meta elements within the head element provide information *about* the document itself. A meta element can be used to provide all sorts of information, but in this case, it specifies the **character encoding** (the standardized collection of letters, numbers, and symbols) used in the document.
5. Also in the head is the mandatory title element. According to the HTML specification, every document must contain a descriptive title.
6. Finally, the body element contains everything that we want to show up in the browser window.

Write and view your HTML document

The steps to write and view your HTML document is given below:

1. Open a text editor of your choice (nano, gedit, atom etc.)

2. Write your HTML code in the file

3. Save the file with .html extension

4. To view, open the HTML file in a web browser

Nested HTML elements

HTML elements can be nested (elements can contain elements).

All HTML documents consist of nested HTML elements.

This example contains four HTML elements:

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>
<p>My first paragraph.</p>

</body>
</html>
```

The <html> element defines the **whole document**. It has a **start** tag <html> and an **end** tag </html>.The element **content** is another HTML element (the <body> element). This is nesting of elements.

HTML Basic Tags

1. Heading Tags

Any document starts with a heading. You can use different sizes for your headings. HTML also has six levels of headings, which use the elements **<h1>, <h2>, <h3>, <h4>, <h5>,** and **<h6>**. While displaying any heading, browser adds one line before and one line after that heading.

Example:

```
<!DOCTYPE html>
<html>

  <head>
    <title>Heading Example</title>
  </head>

  <body>
    <h1>This is heading 1</h1>
    <h2>This is heading 2</h2>
    <h3>This is heading 3</h3>
    <h4>This is heading 4</h4>
    <h5>This is heading 5</h5>
    <h6>This is heading 6</h6>
  </body>

</html>
```

This will produce following output:

This is heading 1

This is heading 2

This is heading 3

This is heading 4

This is heading 5

This is heading 6

2. Paragraph Tag: HTML paragraphs are defined with the <p> tag:

```
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
```

3. Line break Tag:

Whenever you use the **<br />** element, anything following it starts from the next line. This tag is an example of an **empty** element, where you do not need opening and closing tags, as there is nothing to go in between them.

4. HTML Formatting Elements:

HTML also defines special **elements** for defining text with a special **meaning**.

HTML uses elements like <b> and <i> for formatting output, like **bold** or *italic* text.

Formatting elements were designed to display special types of text:

- <b> - Bold text
- <strong> - Important text
- <i> - Italic text
- <em> - Emphasized text
- <mark> - Marked text
- <small> - Small text
- <del> - Deleted text
- <ins> - Inserted text
- <sub> - Subscript text
- <sup> - Superscript text

5. HTML Comments

Comment tags are used to insert comments in the HTML source code. You can add comments to your HTML source by using the following syntax:

<!-- *Write your comments here* -->

6. Links

Links are found in nearly all web pages. Links allow users to click their way from page to page. HTML links are hyperlinks. In HTML, links are defined with the <a> tag:

<a href="url">link text</a>

Example:

<a href="https://www.example.com"> Example Site </a>

The href attribute specifies the destination address of the link.

The **link text** is the visible part (Example Site)

7. HTML Images

In HTML, images are defined with the <img> tag. The <img> tag is empty, it contains attributes only, and does not have a closing tag. The src attribute specifies the URL (web address) of the image. alt attribute provides an alternate text for an image, if the user for some reason cannot view it (because of slow connection, an error in the src attribute, or if the user uses a screen reader).

Example:

<img src="img_demo.jpg" alt="Demo Image">

If a browser cannot find an image, it will display the value of the alt attribute:

Image Size

<img src="img_girl.jpg" alt="Girl in a jacket" style="width:500px;height:600px;">

8. HTML List

HTML offers web authors three ways for specifying lists of information. All lists must contain one or more list elements. Lists may contain −

- **<ul>** − An unordered list. This will list items using plain bullets.
- **<ol>** − An ordered list. This will use different schemes of numbers to list your items.
- **<dl>** − A definition list. This arranges your items in the same way as they are arranged in a dictionary.

Unordered HTML List:

An unordered list starts with the <ul> tag. Each list item starts with the <li> tag. The list items will be marked with bullets (small black circles) by default:

```
ul>
 <li>Coffee</li>
 <li>Tea</li>
 <li>Milk</li>
</ul>
```

You can use **type** attribute for <ul> tag to specify the type of bullet you like. By default, it is a disc. Following are the possible options −

```
<ul type = "square">
<ul type = "disc">
<ul type = "circle">
```

Ordered HTML List:

An ordered list starts with the <ol> tag. Each list item starts with the <li> tag. The list items will be marked with numbers by default:

```
<ol>
 <li>Coffee</li>
 <li>Tea</li>
 <li>Milk</li>
</ol>
```

The type attribute of the <ol> tag, defines the type of the list item marker:

Type	Description
type="1"	The list items will be numbered with numbers (default)
type="A"	The list items will be numbered with uppercase letters
type="a"	The list items will be numbered with lowercase letters
type="I"	The list items will be numbered with uppercase roman numbers
type="i"	The list items will be numbered with lowercase roman numbers

HTML Definition List:

The definition list is the ideal way to present a glossary, list of terms, or other name/value list.

Definition List makes use of following three tags.

- <dl> − Defines the start of the list
- <dt> − A term
- <dd> − Term definition
- </dl> − Defines the end of the list

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

9. HTML Table

An HTML table is defined with the <table> tag.

Each table row is defined with the <tr> tag. A table header is defined with the <th> tag. By default, table headings are bold and centered. A table data/cell is defined with the <td> tag.

```
<table style="width:100%">
<tr>
 <th>Firstname</th>
 <th>Lastname</th>
 <th>Age</th>
</tr>
<tr>
 <td>Jill</td>
 <td>Smith</td>
 <td>50</td>
</tr>
<tr>
 <td>Eve</td>
 <td>Jackson</td>
 <td>94</td>
</tr>
</table>
```

You can use colspan attribute if you want to merge two or more columns into a single column. Similar way you will use rowspan if you want to merge two or more rows.

You can set table background using one of the following two ways −

- **bgcolor** attribute − You can set background color for whole table or just for one cell.
- **background** attribute − You can set background image for whole table or just for one cell.

You can also set border color also using **bordercolor** attribute.

> **Note** − The *bgcolor*, *background*, and *bordercolor* attributes deprecated in HTML5. Do not use these attributes.

10. HTML Forms

HTML Forms are required, when you want to collect some data from the site visitor. For example, during user registration you would like to collect information such as name, email address, credit card, etc.

A form will take input from the site visitor and then will post it to a back-end application such as CGI, ASP Script or PHP script etc. The back-end application will perform required processing on the passed data based on defined business logic inside the application.

There are various form elements available like text fields, textarea fields, drop-down menus, radio buttons, checkboxes, etc.

The HTML **<form>** tag is used to create an HTML form and it has following syntax −

```
<form action = "URL" method = "get or post">
   form elements like input, textarea etc.
</form>
```

The following details about the most frequently used form attributes −

action: The linke of the backend script that is ready to process your passed data.

method: Method to be used to upload data. The most frequently used are GET and POST methods.

The Get Method

GET is used to request data from a specified resource. GET is one of the most common HTTP methods.

Note that the query string (name/value pairs) is sent in the URL of a GET request:

/test/demo_form.php?name1=value1&name2=value2

Some other notes on GET requests:

- GET requests can be cached
- GET requests remain in the browser history
- GET requests can be bookmarked
- GET requests should never be used when dealing with sensitive data
- GET requests have length restrictions
- GET requests is only used to request data (not modify)

The Post Method

POST is used to send data to a server to create/update a resource. The data sent to the server with POST is stored in the request body of the HTTP request:

```
POST /test/demo_form.php HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2
```

Some other notes on POST requests:

- POST requests are never cached
- POST requests do not remain in the browser history
- POST requests cannot be bookmarked
- POST requests have no restrictions on data length

HTML Form Control

There are different types of form controls that you can use to collect data using HTML form −

Text Input Control

There are three types of text input used on forms −

- **Single-line text input controls** − This control is used for items that require only one line of user input, such as search boxes or names. They are created using HTML **<input>** tag.

```
<form >
        First name: <input type = "text" name = "first_name" />
        <br>
        Last name: <input type = "text" name = "last_name" />
</form>
```

  type: Indicates the type of input control and for text input control it will be set to text.

  name: Used to give a name to the control which is sent to the server to be recognized and get the value.

  value: This can be used to provide an initial value inside the control.

  size: Allows to specify the width of the text-input control in terms of characters.

  maxlength: Allows to specify the maximum number of characters a user can enter into the text box.

- **Password input controls** − This is also a single-line text input but it masks the character as soon as a user enters it. They are also created using HTMl <input> tag.

```
<form >
  User ID : <input type = "text" name = "user_id" />
 <br>
 Password: <input type = "password" name = "password" />
</form>
```

- **Multi-line text input controls** − This is used when the user is required to give details that may be longer than a single sentence. Multi-line input controls are created using HTML **<textarea>** tag.

```
<form>
        Description : <br />
          <textarea rows = "5" cols = "50" name = "description">
            Enter description here...
         </textarea>
 </form>
```

  rows: Indicate number of rows in textarea cols: Indicate number of columns in textarea

Here are the different input types you can use in HTML:

- <input type="button">
- <input type="checkbox">
- <input type="color">
- <input type="date">
- <input type="datetime-local">
- <input type="email">
- <input type="file">
- <input type="hidden">
- <input type="image">
- <input type="month">
- <input type="number">
- <input type="password">
- <input type="radio">
- <input type="range">
- <input type="reset">
- <input type="search">
- <input type="submit">
- <input type="tel">
- <input type="text">
- <input type="time">
- <input type="url">
- <input type="week">

The <select> element:

The <select> element defines a **drop-down list**:

```
<select name="cars">
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="fiat">Fiat</option>
  <option value="audi">Audi</option>
</select>
```

The <option> elements defines an option that can be selected.

By default, the first item in the drop-down list is selected.

To define a pre-selected option, add the selected attribute to the option:

```
<option value="fiat" selected>Fiat</option>
```

Use the size attribute to specify the number of visible values:

```
<select name="cars" size="3">
<option value="volvo">Volvo</option>
<option value="saab">Saab</option>
<option value="fiat">Fiat</option>
<option value="audi">Audi</option>
</select>
```

Use the multiple attribute to allow the user to select more than one value:

```
<select name="cars" size="4" multiple>
<option value="volvo">Volvo</option>
<option value="saab">Saab</option>
<option value="fiat">Fiat</option>
<option value="audi">Audi</option>
</select>
```

Input Type Radio

<input type="radio"> defines a **radio button**.

Radio buttons let a user select ONLY ONE of a limited number of choices:

```
<form>
        <input type="radio" name="gender" value="male" checked> Male<br>
        <input type="radio" name="gender" value="female"> Female<br>
        <input type="radio" name="gender" value="other"> Other
</form>
```

Input Type Checkbox

<input type="checkbox"> defines a **checkbox**.

Checkboxes let a user select ZERO or MORE options of a limited number of choices.

```
<form>
        <input type="checkbox" name="vehicle1" value="Bike"> I have a bike<br>
        <input type="checkbox" name="vehicle2" value="Car"> I have a car
</form>
```

Input Type Button

<input type="button"> defines a **button**:

```
<input type="button" onclick="alert('Hello World!')" value="Click Me!">
```

Input Type Date

The <input type="date"> is used for input fields that should contain a date.

Depending on browser support, a date picker can show up in the input field.

```
<form>
        Birthday:
        <input type="date" name="bday">
</form>
```

Input Type Email

The <input type="email"> is used for input fields that should contain an e-mail address. Depending on browser support, the e-mail address can be automatically validated when submitted. Some smartphones recognize the email type, and add ".com" to the keyboard to match email input.

```
<form>
        E-mail:
        <input type="email" name="email">
</form>
```

Input Type Number

The <input type="number"> defines a numeric input field. You can also set restrictions on what numbers are accepted. The following example displays a numeric input field, where you can enter a value from 1 to 5:

```
<form>
        Quantity (between 1 and 5):
        <input type="number" name="quantity" min="1" max="5">
</form>
```

11. Other Tags

HTML <div> Tag:

The <div> tag defines a division or a section in an HTML document. The <div> element is often used as a container for other HTML elements to style them with CSS or to perform certain tasks with JavaScript.

Tip: The <div> element is very often used together with CSS, to layout a web page.

Note: By default, browsers always place a line break before and after the <div> element. However, this can be changed with CSS.

```
<div style="background-color:lightblue">
        <h3>This is a heading</h3>
        <p>This is a paragraph.</p>
</div>
```

HTML class attribute:

The HTML `class` attribute is used to define equal styles for elements with the same class name. So, all HTML elements with the same `class` attribute will have the same format and style. Here we have three `<div>` elements that point to the same class name:

```
<div class="cities">
        <h2>London</h2>
        <p>London is the capital of England.</p>
</div>
```

HTML id attribute:

The id attribute specifies a unique id for an HTML element (the value must be unique within the HTML document).

The id value can be used by CSS and JavaScript to perform certain tasks for a unique element with the specified id value. In CSS, to select an element with a specific id, write a hash (#) character, followed by the id of the element:

```
<h1 id="myHeader">My Header</h1>
```

Difference between class and id:

An HTML element can only have one unique id that belongs to that single element, while a class name can be used by multiple elements:

References:

1. https://www.w3schools.com/html/default.asp

2. https://www.tutorialspoint.com/html/index.htm

3. https://developer.mozilla.org/en-US/docs/Learn/HTML

4. *https://www.oreilly.com/library/view/learning-web-design/9781449337513/ch04.html*

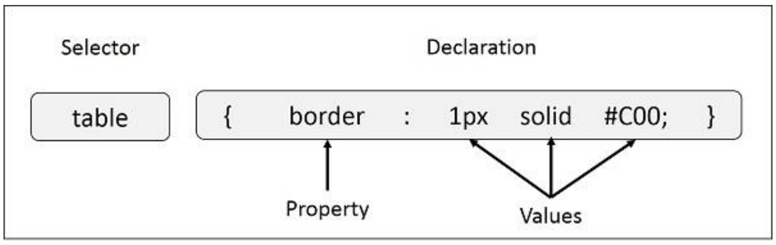
Chapter 2: Basics of CSS

CSS is used to control the style of a web document in a simple and easy way. CSS is the acronym for "Cascading Style Sheet". It is is designed to enable the separation of presentation and content, including layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file, and reduce complexity and repetition in the structural content. In a nutshell, CSS describes how HTML elements should be displayed.

A CSS comprises of style rules that are interpreted by the browser and then applied to the corresponding elements in your document. A style rule is made of three parts −

- **Selector** − A selector is an HTML tag at which a style will be applied. This could be any tag like <h1> or <table> etc.
- **Property** − A property is a type of attribute of HTML tag. Put simply, all the HTML attributes are converted into CSS properties. They could be *color*, *border* etc.
- **Value** − Values are assigned to properties. For example, *color* property can have value either *red* or *#F1F1F1* etc.

You can put CSS Style Rule Syntax as follows − ***selector { property: value }***



Here table is a selector and border is a property and given value *1px solid #C00* is the value of that property.

The element Selector

The element selector selects elements based on the element name. You can select all <p> elements on a page like this (in this case, all <p> elements will be center-aligned, with a red text color):

```
p {
        text-align: center;
        color: red;
}
```

The id Selector

The id selector uses the id attribute of an HTML element to select a specific element.

The id of an element should be unique within a page, so the id selector is used to select one unique element!

To select an element with a specific id, write a hash (#) character, followed by the id of the element.

The style rule below will be applied to the HTML element with id="para1":

```
#para1 {
	text-align: center;
	color: red;
}
```

The class Selector

The class selector selects elements with a specific class attribute. To select elements with a specific class, write a period (.) character, followed by the name of the class. In the example below, all HTML elements with class="center" will be red and center-aligned:

```
.center {
	text-align: center;
	color: red;
}
```

You can also specify that only specific HTML elements should be affected by a class. In the example below, only <p> elements with class="center" will be center-aligned:

```
p.center {
	text-align: center;
	color: red;
}
```

HTML elements can also refer to more than one class. In the example below, the <p> element will be styled according to class="center" and to class="large":

```
<p class="center large">This paragraph refers to two classes.</p>
```

Grouping Selectors

To group selectors, separate each selector with a comma. In the example below we have grouped the selectors h1,h2 and p

```
h1, h2, p {
	text-align: center;
	color: red;
}
```

CSS Comments

Comments are used to explain the code, and may help when you edit the source code at a later date. Comments are ignored by browsers. A CSS comment starts with /* and ends with */. Comments can also span multiple lines:

```
p {
	color: red;
	/* This is a single-line comment */
}
```

```
/* This is a multi-line
comment */
```

Three Ways to Insert CSS:

There are three ways of inserting a style sheet:

- External style sheet
- Internal style sheet
- Inline style

External Style Sheet

With an external style sheet, you can change the look of an entire website by changing just one file!

Each page must include a reference to the external style sheet file inside the <link> element. The <link> element goes inside the <head> section:

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

An external style sheet can be written in any text editor. The file should not contain any html tags. The style sheet file must be saved with a .css extension.

Here is how the "mystyle.css" looks:

```
body {
        background-color: lightblue;
}

h1 {
        color: navy;
        margin-left: 20px;
}
```

Internal Style Sheet

An internal style sheet may be used if one single page has a unique style. Internal styles are defined within the <style> element, inside the <head> section of an HTML page:

```
<head>
<style>
body {
        background-color: linen;
}

h1 {
        color: maroon;
        margin-left: 40px;
```

}
</style>
</head>

Inline Styles

An inline style may be used to apply a unique style for a single element. To use inline styles, add the style attribute to the relevant element. The style attribute can contain any CSS property. The example below shows how to change the color and the left margin of a <h1> element:

<h1 style="color:blue;margin-left:30px;">This is a heading</h1>

Cascading Order

What style will be used when there is more than one style specified for an HTML element?

All the styles in a page will "cascade" into a new "virtual" style sheet by the following rules, where number one has the highest priority:

1. Inline style (inside an HTML element)
2. External and internal style sheets (in the head section)
3. Browser default

So, an inline style has the highest priority, and will override external and internal styles and browser defaults.

CSS Colors

CSS uses color values to specify a color. Typically, these are used to set a color either for the foreground of an element (i.e., its text) or else for the background of the element. They can also be used to affect the color of borders and other decorative effects. You can specify your color values in various formats. Following table lists all the possible formats −

Format	**Syntax**	**Example**
Hex Code	#RRGGBB	p{color:#FF0000;}
Short Hex Code	#RGB	p{color:#6A7;}
RGB %	rgb(rrr%,ggg%,bbb%)	p{color:rgb(50%,50%,50%);}
RGB Absolute	rgb(rrr,ggg,bbb)	p{color:rgb(0,0,255);}
keyword	aqua, black, etc.	p{color:teal;}

CSS Backgrounds

You can set the following background properties of an element −

- The **background-color** property is used to set the background color of an element.
- The **background-image** property is used to set the background image of an element.
- The **background-repeat** property is used to control the repetition of an image in the background.

- The **background-position** property is used to control the position of an image in the background.
- The **background-attachment** property is used to control the scrolling of an image in the background.
- The **background** property is used as a shorthand to specify a number of other background properties.

Example:

```
<style>
  body {
    background-image: url("/css/images/css.jpg");
    background-repeat: repeat-y;
  }
</style>
```

CSS Border Properties

The CSS border properties allow you to specify the style, width, and color of an element's border.

The border-style property specifies what kind of border to display.

The following values are allowed:

- dotted - Defines a dotted border
- dashed - Defines a dashed border
- solid - Defines a solid border
- double - Defines a double border
- groove - Defines a 3D grooved border. The effect depends on the border-color value
- ridge - Defines a 3D ridged border. The effect depends on the border-color value
- inset - Defines a 3D inset border. The effect depends on the border-color value
- outset - Defines a 3D outset border. The effect depends on the border-color value
- none - Defines no border
- hidden - Defines a hidden border

Border Width

The border-width property specifies the width of the four borders.

The width can be set as a specific size (in px, pt, cm, em, etc) or by using one of the three pre-defined values: thin, medium, or thick.

The border-width property can have from one to four values (for the top border, right border, bottom border, and the left border).

Border Color

The border-color property is used to set the color of the four borders.

The color can be set by:

- name - specify a color name, like "red"
- Hex - specify a hex value, like "#ff0000"
- RGB - specify a RGB value, like "rgb(255,0,0)"
- transparent

The border-color property can have from one to four values (for the top border, right border, bottom border, and the left border).

If border-color is not set, it inherits the color of the element.

Border - Shorthand Property

As you can see from the examples above, there are many properties to consider when dealing with borders.

To shorten the code, it is also possible to specify all the individual border properties in one property.

The border property is a shorthand property for the following individual border properties:

- border-width
- border-style (required)
- border-color

```
p {
        border: 5px solid red;
}
```

CSS Margin

The CSS margin properties are used to create space around elements, outside of any defined borders.With CSS, you have full control over the margins. There are properties for setting the margin for each side of an element (top, right, bottom, and left). CSS has properties for specifying the margin for each side of an element:

- margin-top
- margin-right
- margin-bottom
- margin-left

All the margin properties can have the following values:

- auto - the browser calculates the margin
- *length* - specifies a margin in px, pt, cm, etc.
- *%* - specifies a margin in % of the width of the containing element
- inherit - specifies that the margin should be inherited from the parent element

Tip: Negative values are allowed.

The following example sets different margins for all four sides of a <p> element:

```
p {
        margin-top: 100px;
        margin-bottom: 100px;
        margin-right: 150px;
        margin-left: 80px;
}
```

Margin Shorthand:

margin: 25px 50px 75px 100px;

- top margin is 25px
- right margin is 50px
- bottom margin is 75px
- left margin is 100px

CSS Padding

The CSS padding properties are used to generate space around an element's content, inside of any defined borders. With CSS, you have full control over the padding. There are properties for setting the padding for each side of an element (top, right, bottom, and left).

Example:

```
div {
        padding-top: 50px;
        padding-right: 30px;
        padding-bottom: 50px;
        padding-left: 80px;
}
```

Shorthand:

```
div {
        padding: 50px 30px 50px 80px;
}
```

CSS Height/Width

The height and width properties are used to set the height and width of an element. The height and width can be set to auto (this is default. Means that the browser calculates the height and width), or be specified in *length values*, like px, cm, etc., or in percent (%) of the containing block.

```
div {
        height: 200px;
        width: 50%;
        background-color: powderblue;
}
```

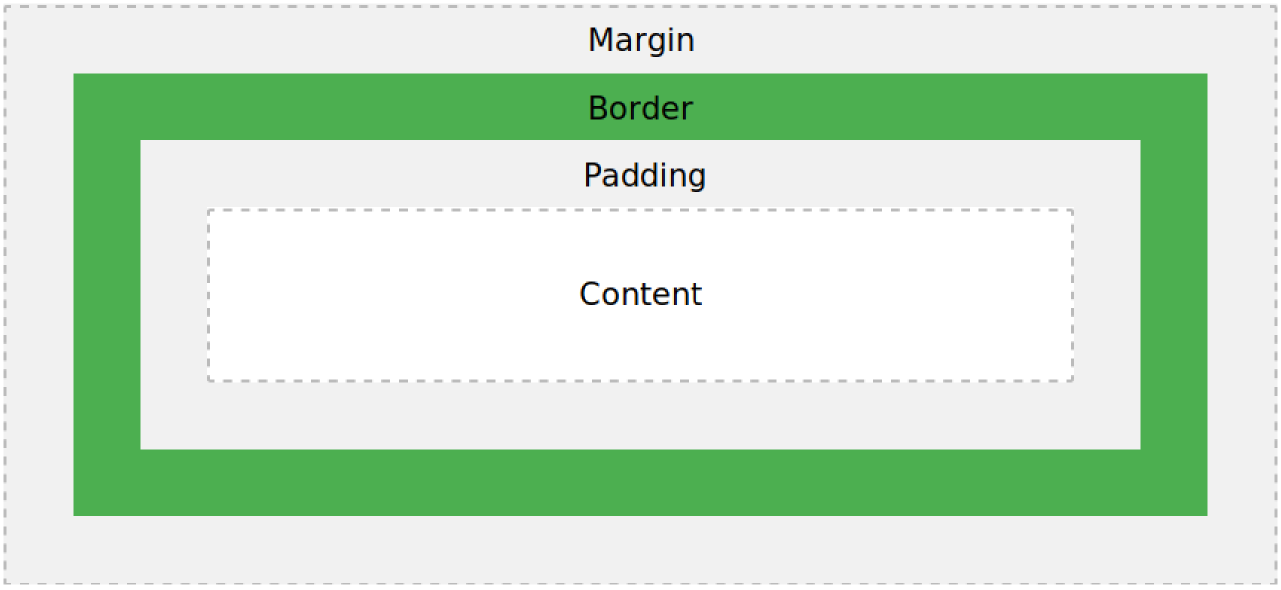
The CSS Box Model

All HTML elements can be considered as boxes. In CSS, the term "box model" is used when talking about design and layout. The CSS box model is essentially a box that wraps around every HTML element. It consists of: margins, borders, padding, and the actual content. The image below illustrates the box model:

Explanation of the different parts:

- **Content** - The content of the box, where text and images appear
- **Padding** - Clears an area around the content. The padding is transparent
- **Border** - A border that goes around the padding and content
- **Margin** - Clears an area outside the border. The margin is transparent

The box model allows us to add a border around elements, and to define space between elements.



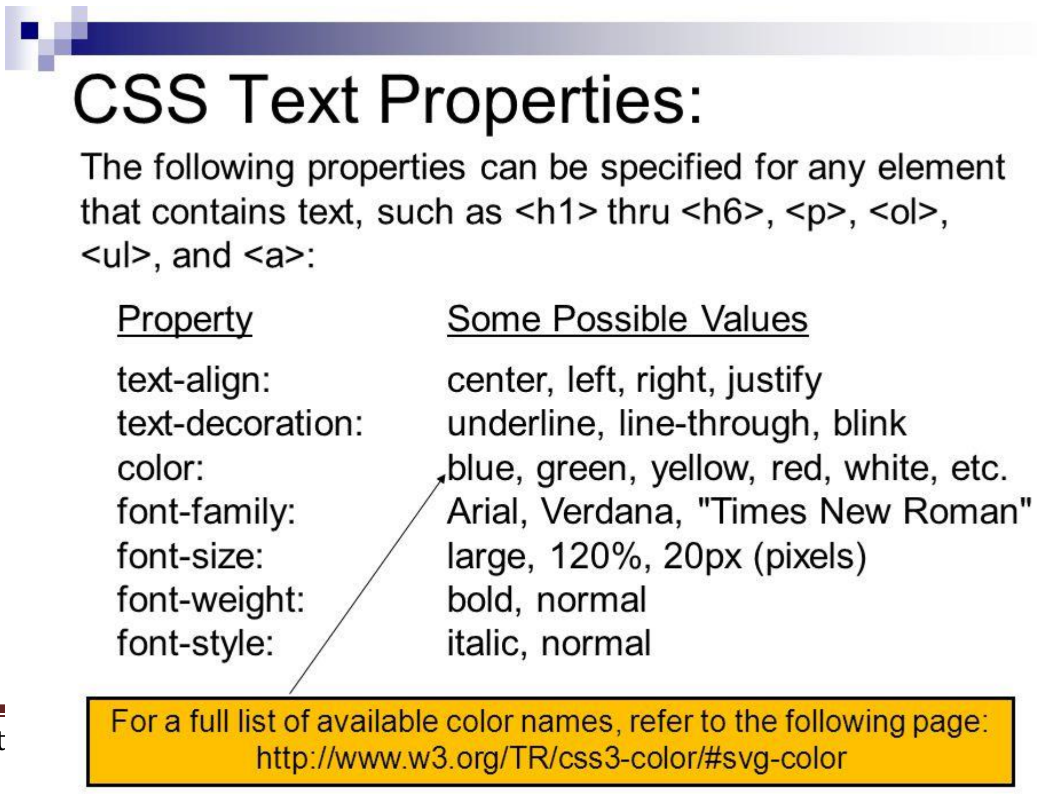
Example:

div {

width: 300px;
border: 15px solid green;
padding: 50px;
margin: 20px;
}

CSS Text Property

CSS Position Property



CSS Text Properties:

The following properties can be specified for any element that contains text, such as <h1> thru <h6>, <p>, <ol>, <ul>, and <a>:

Property	Some Possible Values
text-align:	center, left, right, justify
text-decoration:	underline, line-through, blink
color:	blue, green, yellow, red, white, etc.
font-family:	Arial, Verdana, "Times New Roman"
font-size:	large, 120%, 20px (pixels)
font-weight:	bold, normal
font-style:	italic, normal

For a full list of available color names, refer to the following page:
http://www.w3.org/TR/css3-color/#svg-color

The position property specifies the type of positioning method used for an element (static, relative, fixed, absolute or sticky). The position property specifies the type of positioning method used for an element.

There are five different position values:

- static
- relative
- fixed
- absolute
- sticky

Elements are then positioned using the top, bottom, left, and right properties. However, these properties will not work unless the position property is set first. They also work differently depending on the position value.

position: static;

HTML elements are positioned static by default. Static positioned elements are not affected by the top, bottom, left, and right properties. An element with position: static; is not positioned in any special way; it is always positioned according to the normal flow of the page:

position: relative;

An element with position: relative; is positioned relative to its normal position. Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

position: fixed;

An element with position: fixed; is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element. A fixed element does not leave a gap in the page where it would normally have been located.

position: absolute;

An element with position: absolute; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed). However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

position: sticky;

An element with position: sticky; is positioned based on the user's scroll position. A sticky element toggles between relative and fixed, depending on the scroll position. It is positioned relative until a given offset position is met in the viewport - then it "sticks" in place (like position:fixed).

CSS Overflow

The CSS overflow property controls what happens to content that is too big to fit into an area.

The overflow property specifies whether to clip the content or to add scrollbars when the content of an element is too big to fit in the specified area.

The overflow property has the following values:

- visible - Default. The overflow is not clipped. The content renders outside the element's box
- hidden - The overflow is clipped, and the rest of the content will be invisible
- scroll - The overflow is clipped, and a scrollbar is added to see the rest of the content
- auto - Similar to scroll, but it adds scrollbars only when necessary

overflow: visible

By default, the overflow is visible, meaning that it is not clipped and it renders outside the element's box:

overflow: hidden

With the hidden value, the overflow is clipped, and the rest of the content is hidden:

overflow: scroll

Setting the value to scroll, the overflow is clipped and a scrollbar is added to scroll inside the box. Note that this will add a scrollbar both horizontally and vertically (even if you do not need it):

overflow: auto

The auto value is similar to scroll, but it adds scrollbars only when necessary:

overflow-x and overflow-y

The overflow-x and overflow-y properties specifies whether to change the overflow of content just horizontally or vertically (or both):

overflow-x specifies what to do with the left/right edges of the content.
overflow-y specifies what to do with the top/bottom edges of the content.

CSS Layout -float and clear

The CSS float property specifies how an element should float.

The CSS clear property specifies what elements can float beside the cleared element and on which side.

The float property is used for positioning and formatting content e.g. let an image float left to the text in a container.

The float property can have one of the following values:

- left - The element floats to the left of its container
- right- The element floats to the right of its container

- none - The element does not float (will be displayed just where it occurs in the text). This is default
- inherit - The element inherits the float value of its parent

In its simplest use, the float property can be used to wrap text around images. The following example specifies that an image should float to the **right** in a text:

```
img {
        float: right;
}
```

In the following example the image will be displayed just where it occurs in the text (**float: none**;):

The clear property specifies what elements can float beside the cleared element and on which side.

The clear property can have one of the following values:

- none - Allows floating elements on both sides. This is default
- left - No floating elements allowed on the left side
- right- No floating elements allowed on the right side
- both - No floating elements allowed
- on either the left or the right side
- inherit - The element inherits the clear value of its parent

The most common way to use the clear property is after you have used a float property on an element.

When clearing floats, you should match the clear to the float: If an element is floated to the left, then you should clear to the left. Your floated element will continue to float, but the cleared element will appear below it on the web page.

The following example clears the float to the left. Means that no floating elements are allowed on the left side (of the div):

```
div {
        clear: left;
}
```

References:

1. https://www.w3schools.com/css/default.asp

2. https://www.tutorialspoint.com/css/index.htm

3. https://www.csstutorial.net/css-intro/introductioncss-part1.php

Chapter 3: Basics of JavaScript

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

- JavaScript is a programming language for use in HTML pages
- Invented in 1995 at Netscape Corporation (LiveScript)
- JavaScript has nothing to do with Java
- JavaScript programs are run by an interpreter built into the user's web browser (not on the server)

JavaScript Syntax

JavaScript can be implemented using JavaScript statements that are placed within the **<script>... </script>** HTML tags in a web page.

You can place the **<script>** tags, containing your JavaScript, anywhere within you web page, but it is normally recommended that you should keep it within the **<head>** tags.

The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of your JavaScript will appear as follows.

```
<script ...>
  JavaScript code
</script>
```

The script tag takes two important attributes −

- **Language** − This attribute specifies what scripting language you are using. Typically, its value will be javascript. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
- **Type** − This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".

So your JavaScript segment will look like −

```
<script language="javascript" type="text/javascript">
```

```
  JavaScript code
</script>
```

Your First JavaScript Script

Let us take a sample example to print out "Hello World". We added an optional HTML comment that surrounds our JavaScript code. This is to save our code from a browser that does not support JavaScript. The comment ends with a "//-->". Here "//" signifies a comment in JavaScript, so we add that to prevent a browser from reading the end of the HTML comment as a piece of JavaScript code. Next, we call a function **document.write** which writes a string into our HTML document.

This function can be used to write text, HTML, or both. Take a look at the following code.

```
<html>
  <body>
    <script language="javascript" type="text/javascript">
        document.write("Hello World!");
    </script>
  </body>
</html>
```

This code will produce the following result −

```
Hello World!
```

- JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.
- It is a good programming practice to use semicolons.
- JavaScript is a case-sensitive language.

Comments in JavaScript

JavaScript supports both C-style and C++-style comments, Thus −

- ✓ Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- ✓ Any text between the characters /* and */ is treated as a comment. This may span multiple lines.
- ✓ JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.

- ✓ The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

All the modern browsers come with built-in support for JavaScript. Frequently, you may need to enable or disable this support manually. This chapter explains the procedure of enabling and disabling JavaScript support in your browsers: Internet Explorer, Firefox, chrome, and Opera.

JavaScript Output

JavaScript can "display" data in different ways:

- Writing into an HTML element, using `innerHTML`.
- Writing into the HTML output using `document.write()`.
- Writing into an alert box, using `window.alert()`.
- Writing into the browser console, using `console.log()`.

Using innerHTML

To access an HTML element, JavaScript can use the `document.getElementById(id)` method. The `id` attribute defines the HTML element. The `innerHTML` property defines the HTML content:

Example:

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My First Paragraph</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
</html>
```

Using document.write()

For testing purposes, it is convenient to use `document.write()`:

```
<script>
document.write(5 + 6);
</script>
```

Using window.alert()

You can use an alert box to display data:

```
<script>
window.alert(5 + 6);
</script>
```

Using console.log()

For debugging purposes, you can use the `console.log()` method to display data.

```
<script>
console.log(5 + 6);
</script>
```

Placement in HTML FILE

There is a flexibility given to include JavaScript code anywhere in an HTML document. However the most preferred ways to include JavaScript in an HTML file are as follows −

- ✓ Script in <head>...</head> section.
- ✓ Script in <body>...</body> section.
- ✓ Script in <body>...</body> and <head>...</head> sections.
- ✓ Script in an external file and then include in <head>...</head> section.

In the following section, we will see how we can place JavaScript in an HTML file in different ways.

JavaScript in <head>...</head> section

If you want to have a script run on some event, such as when a user clicks somewhere, then you will place that script in the head as follows −

```
<html>
  <head>
    <script type="text/javascript">
        function sayHello() {
          alert("Hello World")
        }
    </script>
  </head>
 <body>
    <input type="button" onclick="sayHello()" value="Say Hello" />
  </body>
</html>
```

JavaScript in <body>...</body> section

If you need a script to run as the page loads so that the script generates content in the page, then the script goes in the <body> portion of the document. In this case, you would not have any function defined using JavaScript. Take a look at the following code.

```
<html>
  <head>
  </head>
  <body>
      <script type="text/javascript">
        document.write("Hello World")
    </script>
    <p>This is web page body </p>
  </body>
</html>
```

JavaScript in <body> and <head> Sections

You can put your JavaScript code in <head> and <body> section altogether as follows −

```
<html>
  <head>
    <script type="text/javascript">
        function sayHello() {
          alert("Hello World")
        }
    </script>
  </head>
  <body>
    <script type="text/javascript">
        document.write("Hello World")
    </script>
    <input type="button" onclick="sayHello()" value="Say Hello" />
  </body>
</html>
```

JavaScript in External File

As you begin to work more extensively with JavaScript, you will be likely to find that there are cases where you are reusing identical JavaScript code on multiple pages of a site.

You are not restricted to be maintaining identical code in multiple HTML files. The **script** tag provides a mechanism to allow you to store JavaScript in an external file and then include it into your HTML files.

Here is an example to show how you can include an external JavaScript file in your HTML code using **script** tag and its **src** attribute.

```
<html>
  <head>
    <script type="text/javascript" src="filename.js" ></script>
  </head>
  <body>
    .......
  </body>
</html>
```

To use JavaScript from an external file source, you need to write all your JavaScript source code in a simple text file with the extension ".js" and then include that file as shown above.

For example, you can keep the following content in **filename.js** file and then you can use **sayHello** function in your HTML file after including the filename.js file.

```
function sayHello() {
  alert("Hello World")
}
```

JavaScript - Variables

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.

```
var money;
var name;
```

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.

```
var name = "Ali";
var money;
money = 2000.50;
```

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.

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	

JavaScript - Operators

What is an operator?

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. JavaScript supports the following types of operators.

- ✓ Arithmetic Operators
- ✓ Comparision Operators
- ✓ Logical (or Relational) Operators
- ✓ Assignment Operators
- ✓ Conditional (or ternary) Operators

JavaScript - if...else Statement

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow your program to make correct decisions and perform right actions. JavaScript supports conditional statements which are used to perform different actions based on different conditions. Here we will explain the **if..else** statement. JavaScript supports the following forms of **if..else** statement −

- if statement

- if...else statement
- if...else if... statement.

if statement

The **if** statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.

Syntax

The syntax for a basic if statement is as follows −

```
if (expression){
  Statement(s) to be executed if expression is true
}
```

Here a JavaScript expression is evaluated. If the resulting value is true, the given statement(s) are executed. If the expression is false, then no statement would be not executed. Most of the times, you will use comparison operators while making decisions.

Example

Try the following example to understand how the **if** statement works.

```
<html>
  <body>
    <script type="text/javascript">
        var age = 20;
        if( age > 18 ){
          document.write("<b>Qualifies for driving</b>");
        }
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

Output

```
Qualifies for driving
Set the variable to different value and then try...
```

if...else statement:

The **'if...else'** statement is the next form of control statement that allows JavaScript to execute statements in a more controlled way.

Syntax

```
if (expression){
  Statement(s) to be executed if expression is true
}
else{
  Statement(s) to be executed if expression is false
}
```

Here JavaScript expression is evaluated. If the resulting value is true, the given statement(s) in the 'if' block, are executed. If the expression is false, then the given statement(s) in the else block are executed.

Example

Try the following code to learn how to implement an if-else statement in JavaScript.

```
<html>
  <body>
    <script type="text/javascript">
        var age = 15;
        if( age > 18 ){
          document.write("<b>Qualifies for driving</b>");
        }
        else{
          document.write("<b>Does not qualify for driving</b>");
        }
    </script>
    <p>Set the variable to different value and then try...</p>
  </body>
</html>
```

Output

```
Does not qualify for driving
Set the variable to different value and then try...
```

if...else if... statement

The **if...else if...** statement is an advanced form of **if…else** that allows JavaScript to make a correct decision out of several conditions.

Syntax

The syntax of an if-else-if statement is as follows −

```
if (expression 1){
   Statement(s) to be executed if expression 1 is true
}

else if (expression 2){
   Statement(s) to be executed if expression 2 is true
}

else if (expression 3){
   Statement(s) to be executed if expression 3 is true
}

else{
   Statement(s) to be executed if no expression is true
}
```

There is nothing special about this code. It is just a series of **if** statements, where each **if** is a part of the **else** clause of the previous statement. Statement(s) are executed based on the true condition, if none of the conditions is true, then the **else** block is executed.

JavaScript - Switch Case

You can use multiple **if...else…if** statements, as in the previous chapter, to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Starting with JavaScript 1.2, you can use a **switch** statement which handles exactly this situation, and it does so more efficiently than repeated **if...else if** statements.

Syntax

The objective of a **switch** statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each **case** against the value of the expression until a match is found. If nothing matches, a **default** condition will be used.

```
switch (expression)
{
  case condition 1: statement(s)
  break;
  case condition 2: statement(s)
  break;
  ...
  case condition n: statement(s)
  break;
  default: statement(s)
}
```

Break statements play a major role in switch-case statements.

JavaScript - While Loops

While writing a program, you may encounter a situation where you need to perform an action over and over again. In such situations, you would need to write loop statements to reduce the number of lines. JavaScript supports all the necessary loops to ease down the pressure of programming.

The while Loop

Syntax

The syntax of **while loop** in JavaScript is as follows −

```
while (expression){
  Statement(s) to be executed if expression is true
}
```

Example

Try the following example to implement while loop.

```
<html>
  <body>
    <script type="text/javascript">
          var count = 0;
```

```
            document.write("Starting Loop ");
            while (count < 10){
              document.write("Current Count : " + count + "<br />");
              count++;
            }
            document.write("Loop stopped!");
        </script>
        <p>Set the variable to different value and then try...</p>
    </body>
</html>
```

The do...while Loop

The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is **false**.

Syntax

The syntax for **do-while** loop in JavaScript is as follows −

```
do{
  Statement(s) to be executed;
} while (expression);
```

Note − Don’t miss the semicolon used at the end of the do...while loop.

JavaScript - For Loop

The '**for**' loop is the most compact form of looping. It includes the following three important parts −

- The **loop initialization** where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.
- The **test statement** which will test if a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.
- The **iteration statement** where you can increase or decrease your counter.

You can put all the three parts in a single line separated by semicolons.

Syntax

The syntax of for loop is JavaScript is as follows −

```
for (initialization; test condition; iteration statement){
   Statement(s) to be executed if test condition is true
}
```

JavaScript *for...in* loop

The **for...in** loop is used to loop through an object's properties. As we have not discussed Objects yet, you may not feel comfortable with this loop. But once you understand how objects behave in JavaScript, you will find this loop very useful.

Syntax

```
for (variablename in object){
   statement or block to execute
}
```

In each iteration, one property from **object** is assigned to **variablename** and this loop continues till all the properties of the object are exhausted.

JavaScript - Functions

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Like any other advanced programming language, JavaScript also supports all the features necessary to write modular code using functions. You must have seen functions like **alert()** and **write()** in the earlier chapters. We were using these functions again and again, but they had been written in core JavaScript only once.

JavaScript allows us to write our own functions as well. This section explains how to write your own functions in JavaScript.

Function Definition

Before we use a function, we need to define it. The most common way to define a function in JavaScript is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

```
    function functionname(parameter-list)
    {
      statements
    }
```

Example

Try the following example. It defines a function called sayHello that takes no parameters −

```
<script type="text/javascript">
    function sayHello()
    {
      alert("Hello there");
    }
</script>
```

Calling a Function

To invoke a function somewhere later in the script, you would simply need to write the name of that function as shown in the following code.

```
<html>
  <head>
    <script type="text/javascript">
      function sayHello()
      {
        document.write ("Hello there!");
      }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="sayHello()" value="Say Hello">
    </form>
    <p>Use different text in write method and then try...</p>
  </body>
```

```
</html>
```

Function Parameters

Try the following example. We have modified our **sayHello** function here. Now it takes two parameters.

```
<html>
  <head>
    <script type="text/javascript">
      function sayHello(name, age)
      {
        document.write (name + " is " + age + " years old.");
      }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
        <orm>
      <input type="button" onclick="sayHello('Zara', 7)" value="Say Hello">
    </form>
        <p>Use different parameters inside the function and then try...</p>
  </body>
</html>
```

The return Statement

A JavaScript function can have an optional **return** statement. This is required if you want to return a value from a function. This statement should be the last statement in a function.

For example, you can pass two numbers in a function and then you can expect the function to return their multiplication in your calling program.

There is a lot to learn about JavaScript functions, however we have covered the most important concepts in this tutorial.

- JavaScript Nested Functions
- JavaScript Function( ) Constructor

- JavaScript Function Literals

JavaScript - Events

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>
```

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>
```

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

void is an important keyword in JavaScript which can be used as a unary operator that appears before its single operand, which may be of any type. This operator specifies an expression to be evaluated without returning a value.

Syntax

The syntax of **void** can be either of the following two −

```
<head>
  <script type="text/javascript">
    <!--
      void func()
      javascript:void func()
      or:
          void(func())
      javascript:void(func())
    //-->
  </script>
  </head>
```

Example 1

The most common use of this operator is in a client-side *javascript:* URL, where it allows you to evaluate an expression for its side-effects without the browser displaying the value of the evaluated expression.

Here the expression **alert ('Warning!!!')** is evaluated but it is not loaded back into the current document −

```
<html>
  <head>
        <script type="text/javascript">
        <!--
        //-->
     </script>

  </head>
  <body>
     <p>Click the following, This won't react at all...</p>
     <a href="javascript:void(alert('Warning!!!'))">Click me!</a>
  </body>
</html>
```

References:
1. https://www.w3schools.com/js/js_intro.asp

2. http://cglab.ca/~morin/teaching/2405/notes/javascript1.pdf

3. https://www.tutorialspoint.com/javascript/

Chapter 4: AngularJS

Introduction
AngularJS is a very powerful JavaScript Framework. It is used in Single Page Application (SPA) projects. It extends HTML DOM with additional attributes and makes it more responsive to user actions. AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 2.0.

- It can be added to an HTML page with a <script> tag.
- AngularJS extends HTML attributes with Directives, and binds data to HTML with Expressions.

AngularJS is distributed as a JavaScript file, and can be added to a web page with a script tag:
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>

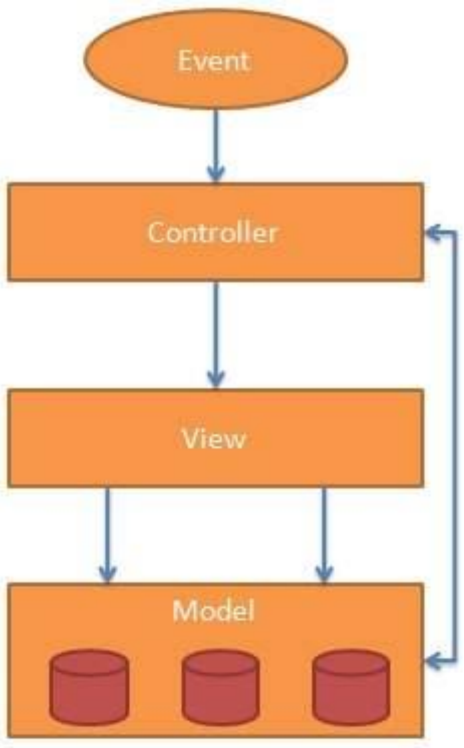
The core features of AngularJS are as follows −

- **Data-binding** − It is the automatic synchronization of data between model and view components.
- **Scope** − These are objects that refer to the model. They act as a glue between controller and view.
- **Controller** − These are JavaScript functions bound to a particular scope.
- **Services** − AngularJS comes with several built-in services such as $http to make a XMLHttpRequests. These are singleton objects which are instantiated only once in app.
- **Filters** − These select a subset of items from an array and returns a new array.
- **Directives** − Directives are markers on DOM elements such as elements, attributes, css, and more. These can be used to create custom HTML tags that serve as new, custom widgets. AngularJS has built-in directives such as ngBind, ngModel, etc.
- **Templates** − These are the rendered view with information from the controller and model. These can be a single file (such as index.html) or multiple views in one page using *partials*.
- **Routing** − It is concept of switching views.
- **Model View Whatever** − MVW is a design pattern for dividing an application into different parts called Model, View, and Controller, each with distinct responsibilities. AngularJS does not implement MVC in the traditional sense, but rather something closer to MVVM (Model-View-ViewModel). The Angular JS team refers it humorously as Model View Whatever.
- **Deep Linking** − Deep linking allows to encode the state of application in the URL so that it can be bookmarked. The application can then be restored from the URL to the same state.
- **Dependency Injection** − AngularJS has a built-in dependency injection subsystem that helps the developer to create, understand, and test the applications easily.

Model View Controller or MVC as it is popularly called, is a software design pattern for developing web applications. A Model View Controller pattern is made up of the following three parts −

- **Model** − It is the lowest level of the pattern responsible for maintaining data.
- **View** − It is responsible for displaying all or a portion of the data to the user.
- **Controller** − It is a software Code that controls the interactions between the Model and View.

MVC is popular because it isolates the application logic from the user interface layer and supports separation of concerns. The controller receives all requests for the application and then works with the model to prepare any data needed by the view. The view then uses the data prepared by the controller to generate a final presentable response. The MVC abstraction can be graphically represented as follows.



AngularJS Extends HTML

AngularJS extends HTML with **ng-directives**. The ng-app directive defines an AngularJS application. The ng-model directive binds the value of HTML controls (input, select, textarea) to application data. The ng-bind directive binds application data to the HTML view.

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p ng-bind="name"></p>
</div>
</body>
</html>
```

Example explained:

AngularJS starts automatically when the web page has loaded. The ng-app directive tells AngularJS that the <div> element is the "owner" of an AngularJS application. The ng-model directive binds the value of the input field to the application variable name. The ng-bind directive binds the content of the <p> element to the application variable name.

First Application

Before creating actual *Hello World !* application using AngularJS, let us see the parts of a AngularJS application. An AngularJS application consists of following three important parts −

- **ng-app** − This directive defines and links an AngularJS application to HTML.
- **ng-model** − This directive binds the values of AngularJS application data to HTML input controls.
- **ng-bind** − This directive binds the AngularJS Application data to HTML tags.

Creating AngularJS Application

Step 1: Load framework

Being a pure JavaScript framework, it can be added using <Script> tag.

```
<script
  src = "https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
</script>
```

Step 2: Define AngularJS application using ng-app directive

```
<div ng-app = "">
  ...
</div>
```

Step 3: Define a model name using ng-model directive

```
<p>Enter your Name: <input type = "text" ng-model = "name"></p>
```

Step 4: Bind the value of above model defined using ng-bind directive

```
<p>Hello <span ng-bind = "name"></span>!</p>
```

Executing AngularJS Application: Use the above-mentioned three steps in an HTML page.

testAngular.html

```
<html>
   <head>
      <title>AngularJS First Application</title>
   </head>

   <body>
      <h1>Sample Application</h1>

      <div ng-app = "">
         <p>Enter your Name: <input type = "text" ng-model = "name"></p>
         <p>Hello <span ng-bind = "name"></span>!</p>
      </div>

      <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
      </script>

   </body></html>
```

Output

Open the file *testAngularJS.html* in a web browser. Enter your name and see the result.

How AngularJS Integrates with HTML

- The ng-app directive indicates the start of AngularJS application.
- The ng-model directive creates a model variable named name, which can be used with the HTML page and within the div having ng-app directive.
- The ng-bind then uses the name model to be displayed in the HTML <span> tag whenever user enters input in the text box.
- Closing</div> tag indicates the end of AngularJS application.

AngularJS Directives

The AngularJS framework can be divided into three major parts −

- **ng-app** − This directive defines and links an AngularJS application to HTML.
- **ng-model** − This directive binds the values of AngularJS application data to HTML input controls.
- **ng-bind** − This directive binds the AngularJS application data to HTML tags.

It is seen that AngularJS directives are HTML attributes with an ng prefix. The ng-init directive initializes AngularJS application variables.

Example

```
<div ng-app="" ng-init="firstName='John'">
<p>The name is <span ng-bind="firstName"></span></p>
</div>
```

Alternatively with valid HTML:

Example

```
<div data-ng-app="" data-ng-init="firstName='John'">
<p>The name is <span data-ng-bind="firstName"></span></p>
</div>
```

We can use data-ng-, instead of ng-, if we want to make our page HTML valid.

AngularJS Expressions

AngularJS expressions are written inside double braces: {{ expression }}.

AngularJS will "output" data exactly where the expression is written:

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <p>My first expression: {{ 5 + 5 }}</p>
</div>

</body>
</html>
```

AngularJS expressions bind AngularJS data to HTML the same way as the ng-bind directive.

AngularJS Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p>{{name}}</p>
</div>

</body>
</html>
```

AngularJS Applications

AngularJS modules define AngularJS applications. AngularJS controllers control AngularJS applications. The ng-app directive defines the application, the ng-controller directive defines the controller.

Example

```
<div ng-app="myApp" ng-controller="myCtrl">

First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{firstName + " " + lastName}}

</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.firstName= "John";
  $scope.lastName= "Doe";
});
```

```
</script>
AngularJS modules define applications:

AngularJS Module
var app = angular.module('myApp', []);
AngularJS controllers control applications:

AngularJS Controller
app.controller('myCtrl', function($scope) {
  $scope.firstName= "John";
  $scope.lastName= "Doe";
});
```

2. AngularJS Expressions

AngularJS binds data to HTML using Expressions. AngularJS expressions can be written inside double braces: {{ expression }}. AngularJS expressions can also be written inside a directive: ng-bind="expression". AngularJS will resolve the expression, and return the result exactly where the expression is written.

AngularJS expressions are much like JavaScript expressions: They can contain literals, operators, and variables.

Example *{{ 5 + 5 }} or {{ firstName + " " + lastName }}*

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="">
  <p>My first expression: {{ 5 + 5 }}</p>
</div>

</body>
</html>
```

If we remove the ng-app directive, HTML will display the expression as it is, without solving it:

Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div>
  <p>My first expression: {{ 5 + 5 }}</p>
</div>

</body>
```

```
</html>
```

We can write expressions wherever we like, AngularJS will simply resolve the expression and return the result.

Example: Let AngularJS change the value of CSS properties like changing the color of the input box below, by changing its value to lightblue.

Example

```
<div ng-app="" ng-init="myCol='lightblue'">
<input style="background-color:{{myCol}}" ng-model="myCol">
</div>
```

AngularJS Numbers
AngularJS numbers are like JavaScript numbers:

Example

```
<div ng-app="" ng-init="quantity=1;cost=5">
<p>Total in dollar: {{ quantity * cost }}</p>
</div>
```

Same example can be solved using ng-bind:

Example

```
<div ng-app="" ng-init="quantity=1;cost=5">
<p>Total in dollar: <span ng-bind="quantity * cost"></span></p>
</div>
```

Using ng-init is not very common. We will learn a better way to initialize data in the chapter about controllers.

AngularJS Strings

AngularJS strings are like JavaScript strings:

Example

```
<div ng-app="" ng-init="firstName='John';lastName='Doe'">
<p>The name is {{ firstName + " " + lastName }}</p>
</div>
```

Same example using ng-bind:

Example

```
<div ng-app="" ng-init="firstName='John';lastName='Doe'">
<p>The name is <span ng-bind="firstName + ' ' + lastName"></span></p>
</div>
```

AngularJS Objects

AngularJS objects are like JavaScript objects:

Example

```
<div ng-app="" ng-init="person={firstName:'John',lastName:'Doe'}">
<p>The name is {{ person.lastName }}</p>
</div>
```

Same example using ng-bind:

Example

```
<div ng-app="" ng-init="person={firstName:'John',lastName:'Doe'}">
<p>The name is <span ng-bind="person.lastName"></span></p>
</div>
```

AngularJS Arrays

AngularJS arrays are like JavaScript arrays:

Example

```
<div ng-app="" ng-init="points=[1,15,19,2,40]">
<p>The third result is {{ points[2] }}</p>
</div>
```

Same example using ng-bind:

Example

```
<div ng-app="" ng-init="points=[1,15,19,2,40]">
<p>The third result is <span ng-bind="points[2]"></span></p>
</div>
```

AngularJS Expressions vs. JavaScript Expressions

Like JavaScript expressions, AngularJS expressions can contain literals, operators, and variables.
Unlike JavaScript expressions, AngularJS expressions can be written inside HTML.
AngularJS expressions do not support conditionals, loops, and exceptions, while JavaScript expressions do.
AngularJS expressions support filters, while JavaScript expressions do not.

3. AngularJS Modules

An AngularJS module defines an application. The module is a container for the different parts of an application. The module is a container for the application controllers. Controllers always belong to a module.

Creating a Module
A module is created by using the AngularJS function angular.module

```
<div ng-app="myApp">...</div>
<script>
var app = angular.module("myApp", []);
</script>
```

The "myApp" parameter refers to an HTML element in which the application will run. We can add controllers, directives, filters, and more, to your AngularJS application.

Adding a Controller

Let us add a controller to our application, and refer to the controller with the ng-controller directive:

Example

```
<div ng-app="myApp" ng-controller="myCtrl">
{{ firstName + " " + lastName }}
</div>

<script>
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
  $scope.firstName = "John";
  $scope.lastName = "Doe";
});
</script>
```

Adding a Directive
AngularJS has a set of built-in directives which we can use to add functionality to our application. In addition we can use the module to add your own directives to our applications:

Example
```
<div ng-app="myApp" w3-test-directive></div>
<script>
var app = angular.module("myApp", []);
app.directive("w3TestDirective", function() {
  return {
    template : "I was made in a directive constructor!"
  };
});
</script>
```

Modules and Controllers in Files

It is common in AngularJS applications to put the module and the controllers in JavaScript files.
n this example, "myApp.js" contains an application module definition, while "myCtrl.js" contains the controller:

Example
```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>

<div ng-app="myApp" ng-controller="myCtrl">
{{ firstName + " " + lastName }}
```

```
</div>

<script src="myApp.js"></script>
<script src="myCtrl.js"></script>

</body>
</html>
```

myApp.js

```
var app = angular.module("myApp", []);
```

The [] parameter in the module definition can be used to define dependent modules. Without the [] parameter, we are not creating a new module, but retrieving an existing one.

myCtrl.js

```
app.controller("myCtrl", function($scope) {
  $scope.firstName = "John";
  $scope.lastName= "Doe";
});
```

Functions can Pollute the Global Namespace

Global functions should be avoided in JavaScript. They can easily be overwritten or destroyed by other scripts. AngularJS modules reduces this problem, by keeping all functions local to the module.

When to Load the Library

While it is common in HTML applications to place scripts at the end of the <body> element, it is recommended that you load the AngularJS library either in the <head> or at the start of the <body>.

This is because calls to angular.module can only be compiled after the library has been loaded.

Example

```
<!DOCTYPE html>
<html>
<body>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<div ng-app="myApp" ng-controller="myCtrl">
{{ firstName + " " + lastName }}
</div>
<script>
var app = angular.module("myApp", []);
app.controller("myCtrl", function($scope) {
  $scope.firstName = "John";
  $scope.lastName = "Doe";
});
</script>
</body>
```

```
</html>
```

4. AngularJS Directives

AngularJS lets you extend HTML with new attributes called Directives. It has a set of built-in directives which offers functionality to our applications. AngularJS also lets us define our own directives.

AngularJS directives are extended HTML attributes with the prefix ng-.
The ng-app directive initializes an AngularJS application.
The ng-init directive initializes application data.
The ng-model directive binds the value of HTML controls (input, select, textarea) to application data.

Example

```
<div ng-app="" ng-init="firstName='John'">
<p>Name: <input type="text" ng-model="firstName"></p>
<p>You wrote: {{ firstName }}</p>
</div>
```

The ng-app directive also tells AngularJS that the <div> element is the "owner" of the AngularJS application.

Data Binding
The {{ firstName }} expression, in the example above, is an AngularJS data binding expression.
Data binding in AngularJS binds AngularJS expressions with AngularJS data.
{{ firstName }} is bound with ng-model="firstName".

In the next example two text fields are bound together with two ng-model directives:

Example
```
<div ng-app="" ng-init="quantity=1;price=5">
Quantity: <input type="number" ng-model="quantity">
Costs:    <input type="number" ng-model="price">
Total in dollar: {{ quantity * price }}
</div>
```

The ng-repeat directive repeats an HTML element:

Example
```
<div ng-app="" ng-init="names=['Jani','Hege','Kai']">
  <ul>
    <li ng-repeat="x in names">
      {{ x }}
    </li>
  </ul>
</div>
```
The ng-repeat directive actually clones HTML elements once for each item in a collection.
The ng-repeat directive used on an array of objects:

Example

```
<div ng-app="" ng-init="names=[
{name:'Jani',country:'Norway'},
{name:'Hege',country:'Sweden'},
{name:'Kai',country:'Denmark'}]">
<ul>
  <li ng-repeat="x in names">
    {{ x.name + ', ' + x.country }}
  </li>
</ul>
</div>
```

AngularJS is perfect for database CRUD (Create Read Update Delete) applications.
Just imagine if these objects were records from a database.

The ng-app Directive
The ng-app directive defines the root element of an AngularJS application.
The ng-app directive will auto-bootstrap (automatically initialize) the application when a web page is loaded.

The ng-model Directive
The ng-model directive binds the value of HTML controls (input, select, textarea) to application data. The ng-model directive can also:

Provide type validation for application data (number, email, required).
Provide status for application data (invalid, dirty, touched, error).
Provide CSS classes for HTML elements.
Bind HTML elements to HTML forms.

Creating New Directives
In addition to all the built-in AngularJS directives, you can create your own directives. New directives are created by using the .directive function. To invoke the new directive, make an HTML element with the same tag name as the new directive.

When naming a directive, we must use a camel case name, w3TestDirective, but when invoking it, we must use - separated name, w3-test-directive:
Example

```
<body ng-app ="myApp">
<w3-test-directive></w3-test-directive>
<script>
var app = angular.module("myApp", []);
app.directive("w3TestDirective", function() {
  return {
    template : "<h1>Made by a directive!</h1>"
  };
});
</script> </body>
```

We can invoke a directive by using:

Element name
Attribute
Class
Comment
The examples below will all produce the same result:

Element name
<w3-test-directive></w3-test-directive>

Attribute
<div w3-test-directive></div>

Class
<div class="w3-test-directive"></div>

Comment
<!-- directive: w3-test-directive -->

Restrictions
You can restrict your directives to only be invoked by some of the methods.

Example
By adding a restrict property with the value "A", the directive can only be invoked by attributes:

```
var app = angular.module("myApp", []);
app.directive("w3TestDirective", function() {
  return {
    restrict : "A",
    template : "<h1>Made by a directive!</h1>"
  };
});
```

The legal restrict values are:

E for Element name
A for Attribute
C for Class
M for Comment
By default the value is EA, meaning that both Element names and attribute names can invoke the directive.

5. AngularJS ng-model Directive

The ng-model directive binds the value of HTML controls (input, select, textarea) to application data.With the ng-model directive we can bind the value of an input field to a variable created in AngularJS.

Example
```
<div ng-app="myApp" ng-controller="myCtrl">
  Name: <input ng-model="name">
```

```
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.name = "John Doe";
});
</script>
```

Two-Way Binding
The binding goes both ways. If the user changes the value inside the input field, the AngularJS property will also change its value:

Example
```
<div ng-app="myApp" ng-controller="myCtrl">
  Name: <input ng-model="name">
  <h1>You entered: {{name}}</h1>
</div>
```

Validate User Input
The ng-model directive can provide type validation for application data (number, e-mail, required):

Example
```
<form ng-app="" name="myForm">
  Email:
  <input type="email" name="myAddress" ng-model="text">
  <span ng-show="myForm.myAddress.$error.email">Not a valid e-mail address</span>
</form>
```

In the example above, the span will be displayed only if the expression in the ng-show attribute returns true. If the property in the ng-model attribute does not exist, AngularJS will create one for us.

Application Status
The ng-model directive can provide status for application data (valid, dirty, touched, error):

Example

```
<form ng-app="" name="myForm" ng-init="myText = 'post@myweb.com'">
  Email:
  <input type="email" name="myAddress" ng-model="myText" required>
  <h1>Status</h1>
  {{myForm.myAddress.$valid}}
  {{myForm.myAddress.$dirty}}
  {{myForm.myAddress.$touched}}
</form>
```

CSS Classes
The ng-model directive provides CSS classes for HTML elements, depending on their status:

Example
```
<style>
input.ng-invalid {
  background-color: lightblue;
}
</style>
<body>

<form ng-app="" name="myForm">
  Enter your name:
  <input name="myName" ng-model="myText" required>
</form>
```
The ng-model directive adds/removes the following classes, according to the status of the form field:
ng-empty
ng-not-empty
ng-touched
ng-untouched
ng-valid
ng-invalid
ng-dirty
ng-pending
ng-pristine

6. AngularJS Data Binding

Data binding in AngularJS is the synchronization between the model and the view.

Data Model
AngularJS applications usually have a data model. The data model is a collection of data available for the application.

Example
```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.firstname = "John";
  $scope.lastname = "Doe";
});
```

HTML View
The HTML container where the AngularJS application is displayed, is called the view.
The view has access to the model, and there are several ways of displaying model data in the view.
You can use the ng-bind directive, which will bind the innerHTML of the element to the specified model property:

Example
```
<p ng-bind="firstname"></p>
```

We can also use double braces {{ }} to display content from the model:

Example

```
<p>First name: {{firstname}}</p>
```

Or we can use the ng-model directive on HTML controls to bind the model to the view.

The ng-model Directive

Use the ng-model directive to bind data from the model to the view on HTML controls (input, select, textarea)

Example

```
<input ng-model="firstname">
```

The ng-model directive provides a two-way binding between the model and the view.

Two-way Binding

Data binding in AngularJS is the synchronization between the model and the view.

When data in the model changes, the view reflects the change, and when data in the view changes, the model is updated as well. This happens immediately and automatically, which makes sure that the model and the view is updated at all times.

Example

```
<div ng-app="myApp" ng-controller="myCtrl">
  Name: <input ng-model="firstname">
  <h1>{{firstname}}</h1>
</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.firstname = "John";
  $scope.lastname = "Doe";
});
</script>
```

AngularJS Controller

Applications in AngularJS are controlled by controllers. Because of the immediate synchronization of the model and the view, the controller can be completely separated from the view, and simply concentrate on the model data. Thanks to the data binding in AngularJS, the view will reflect any changes made in the controller.

Example

```
<div ng-app="myApp" ng-controller="myCtrl">
  <h1 ng-click="changeName()">{{firstname}}</h1>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.firstname = "John";
  $scope.changeName = function() {
    $scope.firstname = "Nelly";
  }
});
</script>
```

7. AngularJS Controllers
AngularJS controllers control the data of AngularJS applications. These are regular JavaScript Objects. AngularJS applications are controlled by controllers. The ng-controller directive defines the application controller. A controller is a JavaScript Object, created by a standard JavaScript object constructor.

Example

```
<div ng-app="myApp" ng-controller="myCtrl">

First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{firstName + " " + lastName}}

</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.firstName = "John";
  $scope.lastName = "Doe";
});
</script>
```

Explanation:

The AngularJS application is defined by ng-app="myApp". The application runs inside the <div>.
The ng-controller="myCtrl" attribute is an AngularJS directive. It defines a controller.
The myCtrl function is a JavaScript function.
AngularJS will invoke the controller with a $scope object.
In AngularJS, $scope is the application object (the owner of application variables and functions).
The controller creates two properties (variables) in the scope (firstName and lastName).
The ng-model directives bind the input fields to the controller properties (firstName and lastName).

Controller Methods
The example above demonstrated a controller object with two properties: lastName and firstName.
A controller can also have methods (variables as functions):

Example

```
<div ng-app="myApp" ng-controller="personCtrl">

First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{fullName()}}
```

```
</div>

<script>
var app = angular.module('myApp', []);
app.controller('personCtrl', function($scope) {
  $scope.firstName = "John";
  $scope.lastName = "Doe";
  $scope.fullName = function() {
    return $scope.firstName + " " + $scope.lastName;
  };
});
</script>
```

Controllers In External Files
In larger applications, it is common to store controllers in external files. Just copy the code between the <script> tags into an external file named personController.js:

Example

```
<div ng-app="myApp" ng-controller="personCtrl">
First Name: <input type="text" ng-model="firstName"><br>
Last Name: <input type="text" ng-model="lastName"><br>
<br>
Full Name: {{fullName()}}
</div>
<script src="personController.js"></script>
```

Another Example

For the next example we will create a new controller file:

```
angular.module('myApp', []).controller('namesCtrl', function($scope) {
  $scope.names = [
    {name:'Jani',country:'Norway'},
    {name:'Hege',country:'Sweden'},
    {name:'Kai',country:'Denmark'}
  ];
});
```

Save the file as namesController.js:
And then use the controller file in an application:

Example

```
<div ng-app="myApp" ng-controller="namesCtrl">
<ul>
  <li ng-repeat="x in names">
    {{ x.name + ', ' + x.country }}
  </li>
</ul>
</div>
<script src="namesController.js"></script>
```

8. AngularJS Scope

The scope is the binding part between the HTML (view) and the JavaScript (controller). The scope is an object with the available properties and methods. The scope is available for both the view and the controller.
How to Use the Scope?
When you make a controller in AngularJS, you pass the $scope object as an argument:

Example
Properties made in the controller, can be referred to in the view:

```
<div ng-app="myApp" ng-controller="myCtrl">
<h1>{{carname}}</h1>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.carname = "Volvo";
});
</script>
```

When adding properties to the $scope object in the controller, the view (HTML) gets access to these properties. In the view, you do not use the prefix $scope, you just refer to a property name, like {{carname}}.

Understanding the Scope
If we consider an AngularJS application to consist of:
View, which is the HTML.
Model, which is the data available for the current view.
Controller, which is the JavaScript function that makes/changes/removes/controls the data.
Then the scope is the Model.

The scope is a JavaScript object with properties and methods, which are available for both the view and the controller.

Example
If we make changes in the view, the model and the controller will be updated:

```
<div ng-app="myApp" ng-controller="myCtrl">
<input ng-model="name">
<h1>My name is {{name}}</h1>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.name = "John Doe";
});
</script>
```

Know Your Scope
It is important to know which scope you are dealing with, at any time.

In the two examples above there is only one scope, so knowing your scope is not an issue, but for larger applications there can be sections in the HTML DOM which can only access certain scopes.

Example
When dealing with the ng-repeat directive, each repetition has access to the current repetition object:

```
<div ng-app="myApp" ng-controller="myCtrl">
<ul>
  <li ng-repeat="x in names">{{x}}</li>
</ul>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.names = ["Emil", "Tobias", "Linus"];
});
</script>
```

Each <li> element has access to the current repetition object, in this case a string, which is referred to by using x.

Root Scope
All applications have a $rootScope which is the scope created on the HTML element that contains the ng-app directive. The rootScope is available in the entire application. If a variable has the same name in both the current scope and in the rootScope, the application uses the one in the current scope.

Example
A variable named "color" exists in both the controller's scope and in the rootScope:

```
<body ng-app="myApp">
<p>The rootScope's favorite color:</p>
<h1>{{color}}</h1>
<div ng-controller="myCtrl">
  <p>The scope of the controller's favorite color:</p>
  <h1>{{color}}</h1>
</div>
<p>The rootScope's favorite color is still:</p>
<h1>{{color}}</h1>
<script>
var app = angular.module('myApp', []);
app.run(function($rootScope) {
  $rootScope.color = 'blue';
});
app.controller('myCtrl', function($scope) {
  $scope.color = "red";
});
</script>
</body>
```

9. AngularJS Filters

AngularJS provides filters to transform data:

currency Format a number to a currency format.
date Format a date to a specified format.
filter Select a subset of items from an array.
json Format an object to a JSON string.
limitTo Limits an array/string, into a specified number of elements/characters.
lowercase Format a string to lower case.
number Format a number to a string.
orderBy Orders an array by an expression.
uppercase Format a string to upper case.
Adding Filters to Expressions
Filters can be added to expressions by using the pipe character |, followed by a filter.

The uppercase filter format strings to upper case:

Example
```
<div ng-app="myApp" ng-controller="personCtrl">
<p>The name is {{ lastName | uppercase }}</p>
</div>
```
The lowercase filter format strings to lower case:

Example
```
<div ng-app="myApp" ng-controller="personCtrl">
<p>The name is {{ lastName | lowercase }}</p>
</div>
```

Adding Filters to Directives
Filters are added to directives, like ng-repeat, by using the pipe character |, followed by a filter:

Example
The orderBy filter sorts an array:

```
<div ng-app="myApp" ng-controller="namesCtrl">
<ul>
  <li ng-repeat="x in names | orderBy:'country'">
    {{ x.name + ', ' + x.country }}
  </li>
</ul>
</div>
```

The currency Filter
The currency filter formats a number as currency:

Example
```
<div ng-app="myApp" ng-controller="costCtrl">
<h1>Price: {{ price | currency }}</h1>
</div>
```

The filter Filter
The filter filter selects a subset of an array. The filter filter can only be used on arrays, and it returns an array containing only the matching items.

Example
Return the names that contains the letter "i":

```
<div ng-app="myApp" ng-controller="namesCtrl">
<ul>
  <li ng-repeat="x in names | filter : 'i'">
    {{ x }}
  </li>
</ul>
</div>
```

Filter an Array Based on User Input
By setting the ng-model directive on an input field, we can use the value of the input field as an expression in a filter.

Type a letter in the input field, and the list will shrink/grow depending on the match:

Jani
Carl
Margareth
Hege
Joe
Gustav
Birgit
Mary
Kai

Example
```
<div ng-app="myApp" ng-controller="namesCtrl">
<p><input type="text" ng-model="test"></p>
<ul>
  <li ng-repeat="x in names | filter : test">
    {{ x }}
  </li>
</ul>
</div>
```

10. AngularJS Events
AngularJS has its own HTML events directives. You can add AngularJS event listeners to your HTML elements by using one or more of these directives:
ng-blur
ng-change
ng-click
ng-copy
ng-cut
ng-dblclick

ng-focus
ng-keydown
ng-keypress
ng-keyup
ng-mousedown
ng-mouseenter
ng-mouseleave
ng-mousemove
ng-mouseover
ng-mouseup
ng-paste
The event directives allows us to run AngularJS functions at certain user events. An AngularJS event will not overwrite an HTML event, both events will be executed.

Mouse Events
Mouse events occur when the cursor moves over an element, in this order:
ng-mouseover
ng-mouseenter
ng-mousemove
ng-mouseleave

Or when a mouse button is clicked on an element, in this order:
ng-mousedown
ng-mouseup
ng-click

We can add mouse events on any HTML element.

Example
Increase the count variable when the mouse moves over the H1 element:

```
<div ng-app="myApp" ng-controller="myCtrl">
<h1 ng-mousemove="count = count + 1">Mouse over me!</h1>
<h2>{{ count }}</h2>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.count = 0;
});
</script>
```

The ng-click Directive
The ng-click directive defines AngularJS code that will be executed when the element is being clicked.
Example

```
<div ng-app="myApp" ng-controller="myCtrl">
<button ng-click="count = count + 1">Click me!</button>
<p>{{ count }}</p>
</div>
<script>
```

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.count = 0;
});
</script>
```

A function can also be referred here:

Example

```
<div ng-app="myApp" ng-controller="myCtrl">
<button ng-click="myFunction()">Click me!</button>
<p>{{ count }}</p>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.count = 0;
  $scope.myFunction = function() {
    $scope.count++;
  }
});
</script>
```

Toggle, True/False
If you want to show a section of HTML code when a button is clicked, and hide when the button is clicked again, like a dropdown menu, make the button behave like a toggle switch:

Click Me

Example

```
<div ng-app="myApp" ng-controller="myCtrl">
<button ng-click="myFunc()">Click Me!</button>
<div ng-show="showMe">
  <h1>Menu:</h1>
  <div>Pizza</div>
  <div>Pasta</div>
  <div>Pesce</div>
</div>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.showMe = false;
  $scope.myFunc = function() {
    $scope.showMe = !$scope.showMe;
  }
});
</script>
```

The showMe variable starts out as the Boolean value false.
The myFunc function sets the showMe variable to the opposite of what it is, by using the ! (not) operator.

$event Object

You can pass the $event object as an argument when calling the function.
The $event object contains the browser's event object:

Example
```
<div ng-app="myApp" ng-controller="myCtrl">
<h1 ng-mousemove="myFunc($event)">Mouse Over Me!</h1>
<p>Coordinates: {{x + ', ' + y}}</p>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.myFunc = function(myE) {
    $scope.x = myE.clientX;
    $scope.y = myE.clientY;
  }
});
</script>
```

11. AngularJS Forms

Forms in AngularJS provides data-binding and validation of input controls.

Input Controls
Input controls are the HTML input elements:
input elements
select elements
button elements
textarea elements

Data-Binding
Input controls provides data-binding by using the ng-model directive.
```
<input type="text" ng-model="firstname">
```
The application does now have a property named firstname. The ng-model directive binds the input controller to the rest of your application. The property firstname, can be referred to in a controller:

Example
```
<script>
var app = angular.module('myApp', []);
app.controller('formCtrl', function($scope) {
  $scope.firstname = "John";
});
</script>
```
It can also be referred to elsewhere in the application:
Example
```
<form>
  First Name: <input type="text" ng-model="firstname">
</form>
<h1>You entered: {{firstname}}</h1>
```
Checkbox
A checkbox has the value true or false. Apply the ng-model directive to a checkbox, and use its value in your application.

Example
Show the header if the checkbox is checked:

```
 <form>
  Check to show a header:
  <input type="checkbox" ng-model="myVar">
</form>
<h1 ng-show="myVar">My Header</h1>
```

Radiobuttons
Bind radio buttons to your application with the ng-model directive. Radio buttons with the same ng-model can have different values, but only the selected one will be used.

Example
Display some text, based on the value of the selected radio button:

```
<form>
  Pick a topic:
  <input type="radio" ng-model="myVar" value="dogs">Dogs
  <input type="radio" ng-model="myVar" value="tuts">Tutorials
  <input type="radio" ng-model="myVar" value="cars">Cars
</form>
```

The value of myVar will be either dogs, tuts, or cars.

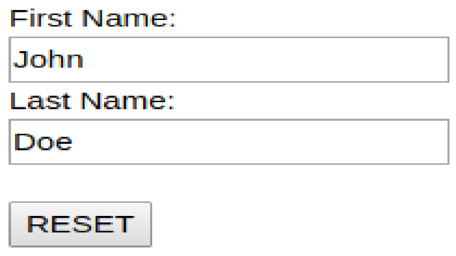
Selectbox
Bind select boxes to your application with the ng-model directive. The property defined in the ng-model attribute will have the value of the selected option in the selectbox.

Example
Display some text, based on the value of the selected option:

```
<form>
  Select a topic:
  <select ng-model="myVar">
    <option value="">
    <option value="dogs">Dogs
    <option value="tuts">Tutorials
    <option value="cars">Cars
  </select>
</form>
```

The value of myVar will be either dogs, tuts, or cars.

An AngularJS Form Example



First Name:

John

Last Name:

Doe

form = {"firstName":"John","lastName":"Doe"}
master = {"firstName":"John","lastName":"Doe"}

Application Code

```
<div ng-app="myApp" ng-controller="formCtrl">
  <form novalidate>
    First Name:<br>
    <input type="text" ng-model="user.firstName"><br>
    Last Name:<br>
    <input type="text" ng-model="user.lastName">
    <br><br>
    <button ng-click="reset()">RESET</button>
  </form>
  <p>form = {{user}}</p>
  <p>master = {{master}}</p>
</div>
<script>
var app = angular.module('myApp', []);
app.controller('formCtrl', function($scope) {
  $scope.master = {firstName: "John", lastName: "Doe"};
  $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
  };
  $scope.reset();
});
</script>
```

The novalidate attribute is new in HTML5. It disables any default browser validation.

Example Explained
The ng-app directive defines the AngularJS application.
The ng-controller directive defines the application controller.
The ng-model directive binds two input elements to the user object in the model.
The formCtrl controller sets initial values to the master object, and defines the reset() method.
The reset() method sets the user object equal to the master object.
The ng-click directive invokes the reset() method, only if the button is clicked.
The novalidate attribute is not needed for this application, but normally you will use it in AngularJS forms, to override standard HTML5 validation.

12. AngularJS Form Validation
AngularJS can validate input data. It offers client-side form validation. AngularJS monitors the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state. AngularJS also holds information about whether they have been touched, or modified, or not.

Standard HTML5 attributes can be used to validate input, or you can make your own validation functions. Client-side validation cannot alone secure user input. Server side validation is also necessary.

Required
Use the HTML5 attribute **required** to specify that the input field must be filled out:

Example
The input field is required:

```
<form name="myForm">
```

```
  <input name="myInput" ng-model="myInput" required>
</form>
<p>The input's valid state is:</p>
<h1>{{myForm.myInput.$valid}}</h1>
```

E-mail
Use the HTML5 type **email** to specify that the value must be an e-mail:

Example
The input field has to be an e-mail:

```
<form name="myForm">
  <input name="myInput" ng-model="myInput" type="email">
</form>
<p>The input's valid state is:</p>
<h1>{{myForm.myInput.$valid}}</h1>
```

Form State and Input State
AngularJS is constantly updating the state of both the form and the input fields.
Input fields have the following states:

$untouched The field has not been touched yet
$touched The field has been touched
$pristine The field has not been modified yet
$dirty The field has been modified
$invalid The field content is not valid
$valid The field content is valid
They are all properties of the input field, and are either true or false.

Forms have the following states:

$pristine No fields have been modified yet
$dirty One or more have been modified
$invalid The form content is not valid
$valid The form content is valid
$submitted The form is submitted
They are all properties of the form, and are either true or false.

You can use these states to show meaningful messages to the user. Example, if a field is required, and the user leaves it blank, you should give the user a warning:

Example
Show an error message if the field has been touched AND is empty:

```
<input name="myName" ng-model="myName" required>
<span ng-show="myForm.myName.$touched && myForm.myName.$invalid">The name is
required.</span>
```

CSS Classes
AngularJS adds CSS classes to forms and input fields depending on their states.
The following classes are added to, or removed from, input fields:

ng-untouched The field has not been touched yet
ng-touched The field has been touched
ng-pristine The field has not been modified yet
ng-dirty The field has been modified
ng-valid The field content is valid
ng-invalid The field content is not valid
ng-valid-key One key for each validation. Example: ng-valid-required, useful when there are more than one thing that must be validated
ng-invalid-key Example: ng-invalid-required

The following classes are added to, or removed from, forms:

ng-pristine No fields has not been modified yet
ng-dirty One or more fields has been modified
ng-valid The form content is valid
ng-invalid The form content is not valid
ng-valid-key One key for each validation. Example: ng-valid-required, useful when there are more than one thing that must be validated
ng-invalid-key Example: ng-invalid-required
The classes are removed if the value they represent is false.

Add styles for these classes to give your application a better and more intuitive user interface.

Example
Apply styles, using standard CSS:

```
<style>
input.ng-invalid {
  background-color: pink;
}
input.ng-valid {
  background-color: lightgreen;
}
</style>
```
Forms can also be styled:

Example
Apply styles for unmodified (pristine) forms, and for modified forms:

```
<style>
form.ng-pristine {
  background-color: lightblue;
}
form.ng-dirty {
  background-color: pink;
}
</style>
```

Custom Validation

To create your own validation function is a bit more tricky; You have to add a new directive to your application, and deal with the validation inside a function with certain specified arguments.

Example
Create your own directive, containing a custom validation function, and refer to it by using my-directive.

The field will only be valid if the value contains the character "e":

```
<form name="myForm">
<input name="myInput" ng-model="myInput" required my-directive>
</form>
<script>
var app = angular.module('myApp', []);
app.directive('myDirective', function() {
  return {
    require: 'ngModel',
    link: function(scope, element, attr, mCtrl) {
      function myValidation(value) {
        if (value.indexOf("e") > -1) {
          mCtrl.$setValidity('charE', true);
        } else {
          mCtrl.$setValidity('charE', false);
        }
        return value;
      }
      mCtrl.$parsers.push(myValidation);
    }
  };
});
</script>
```

Example Explained:
In HTML, the new directive will be referred to by using the attribute my-directive.

In the JavaScript we start by adding a new directive named myDirective.

Remember, when naming a directive, you must use a camel case name, myDirective, but when invoking it, you must use - separated name, my-directive.

Then, return an object where you specify that we require ngModel, which is the ngModelController.

Make a linking function which takes some arguments, where the fourth argument, mCtrl, is the ngModelController,

Then specify a function, in this case named myValidation, which takes one argument, this argument is the value of the input element.

Test if the value contains the letter "e", and set the validity of the model controller to either true or false.

At last, mCtrl.$parsers.push(myValidation); will add the myValidation function to an array of other functions, which will be executed every time the input value changes.

Validation Example

```
<!DOCTYPE html>
<html>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
<body>
<h2>Validation Example</h2>
<form  ng-app="myApp"  ng-controller="validateCtrl"
name="myForm" novalidate>
<p>Username:<br>
  <input type="text" name="user" ng-model="user" required>
  <span style="color:red" ng-show="myForm.user.$dirty && myForm.user.$invalid">
  <span ng-show="myForm.user.$error.required">Username is required.</span>
  </span>
</p>
<p>Email:<br>
  <input type="email" name="email" ng-model="email" required>
  <span style="color:red" ng-show="myForm.email.$dirty && myForm.email.$invalid">
  <span ng-show="myForm.email.$error.required">Email is required.</span>
  <span ng-show="myForm.email.$error.email">Invalid email address.</span>
  </span>
</p>
<p>
  <input type="submit"
  ng-disabled="myForm.user.$dirty && myForm.user.$invalid ||
  myForm.email.$dirty && myForm.email.$invalid">
</p>
</form>
<script>
var app = angular.module('myApp', []);
app.controller('validateCtrl', function($scope) {
  $scope.user = 'John Doe';
  $scope.email = 'john.doe@gmail.com';
});
</script>
</body>
</html>
```

The HTML form attribute novalidate is used to disable default browser validation.

Example Explained

The AngularJS directive ng-model binds the input elements to the model.
The model object has two properties: user and email.
Because of ng-show, the spans with color:red are displayed only when user or email is $dirty and $invalid.

References:

1. https://www.tutorialspoint.com/angularjs/index.html

2. https://www.w3schools.com/angular/default.asp

3. https://www.udemy.com/angular-2-and-nodejs-the-practical-guide

Chapter 5: Node.js

Introduction

Node.js is a very powerful JavaScript-based platform built on Google Chrome's JavaScript V8 Engine. It is used to develop I/O intensive web applications like video streaming sites, single-page applications, and other web applications. Node.js is open source, completely free, and used by thousands of developers around the world.

Node.js = Runtime Environment + JavaScript Library

Download Node.js: The official Node.js website has installation instructions for Node.js: https://nodejs.org

First Application

Before creating an actual "Hello, World!" application using Node.js, let us see the components of a Node.js application. A Node.js application consists of the following three important components −

- **Import required modules** − We use the **require** directive to load Node.js modules.
- **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
- **Read request and return response** − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

Creating Node.js Application

Step 1 - Import Required Module

We use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows −

```
var http = require("http");
```

Step 2 - Create Server

We use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

```
http.createServer(function (request, response) {
  // Send the HTTP header
  // HTTP Status: 200 : OK
  // Content Type: text/plain
  response.writeHead(200, {'Content-Type': 'text/plain'});

  // Send the response body as "Hello World"
  response.end('Hello World\n');
}).listen(8081);
```

```
// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

The above code is enough to create an HTTP server which listens, i.e., waits for a request over 8081 port on the local machine.

Step 3 - Testing Request & Response

Let's put step 1 and 2 together in a file called **main.js** and start our HTTP server as shown below −

```
var http = require("http");

http.createServer(function (request, response) {
   // Send the HTTP header
   // HTTP Status: 200 : OK
   // Content Type: text/plain
   response.writeHead(200, {'Content-Type': 'text/plain'});

   // Send the response body as "Hello World"
   response.end('Hello World\n');
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Now execute the main.js to start the server as follows −

```
$ node main.js
```

Verify the Output. Server has started.

```
Server running at http://127.0.0.1:8081/
```

Make a Request to the Node.js Server

Open http://127.0.0.1:8081/ in any browser and observe the following result.

Node.js Modules

Consider modules to be the same as JavaScript libraries. A set of functions you want to include in your application.

Built-in Modules: Node.js has a set of built-in modules which you can use without any further installation.

Include Modules

To include a module, use the require() function with the name of the module:

```
var http = require('http');
```

Now your application has access to the HTTP module, and is able to create a server:

```
http.createServer(function (req, res) {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end('Hello World!');
}).listen(8080);
```

Create Your Own Modules

You can create your own modules, and easily include them in your applications.

The following example creates a module that returns a date and time object:

Example

Create a module that returns the current date and time:

```
exports.myDateTime = function () {
        return Date();
};
```

Use the exports keyword to make properties and methods available outside the module file.

Save the code above in a file called "myfirstmodule.js"

Include Your Own Module

Now you can include and use the module in any of your Node.js files.

Example:

Use the module "myfirstmodule" in a Node.js file:

```
var http = require('http');
var dt = require('./myfirstmodule');
http.createServer(function (req, res) {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.write("The date and time are currently: " + dt.myDateTime());
        res.end();
}).listen(8080);
```

Notice that we use ./ to locate the module, that means that the module is located in the same folder as the Node.js file.

Save the code above in a file called "demo_module.js", and initiate the file:

The Built-in HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP). To include the HTTP module, use the require() method:

```
var http = require('http');
```

Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the createServer() method to create an HTTP server:

Example

```
var http = require('http');

//create a server object:
http.createServer(function (req, res) {
        res.write('Hello World!'); //write a response to the client
        res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.

Save the code above in a file called "demo_http.js", and initiate the file:

FileSystem Module

The Node.js file system module allows you to work with the file system on your computer. To include the File System module, use the require() method:

```
var fs = require('fs');
```

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Read Files

The *fs.readFile()* method is used to read files on your computer.

Assume we have the following HTML file (located in the same folder as Node.js):

demofile1.html

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

Create a Node.js file that reads the HTML file, and return the content:

Example

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
fs.readFile('demofile1.html', function(err, data) {
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.write(data);
        res.end();
 });
}).listen(8080);
```

Save the code above in a file called "demo_readfile.js", and initiate the file

The Built-in URL Module

The URL module splits up a web address into readable parts. To include the URL module, use the require() method:

```
var url = require('url');
```

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

Example

Split a web address into readable parts:

```
var url = require('url');
var adr = 'http://localhost:8080/default.htm?year=2017&month=february';
var q = url.parse(adr, true);
console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2017&month=february'
var qdata = q.query; //returns an object: { year: 2017, month: 'february' }
console.log(qdata.month); //returns 'february'
```

Node.js File Server

Now we know how to parse the query string, and in the previous chapter we learned how to make Node.js behave as a file server. Let us combine the two, and serve the file requested by the client.

Create two html files and save them in the same folder as your node.js files.

summer.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Summer</h1>
<p>I love the sun!</p>
```

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

winter.html

```
<!DOCTYPE html>
<html>
<body>
<h1>Winter</h1>
<p>I love the snow!</p>
</body>
</html>
```

Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

demo_fileserver.js:

```
var http = require('http');
var url = require('url');
var fs = require('fs');
http.createServer(function (req, res) {
var q = url.parse(req.url, true);
var filename = "." + q.pathname;
fs.readFile(filename, function(err, data) {
if (err) {
 res.writeHead(404, {'Content-Type': 'text/html'});
 return res.end("404 Not Found");
 }
 res.writeHead(200, {'Content-Type': 'text/html'});
 res.write(data);
 return res.end();
 });
}).listen(8080);
```

Nodejs Event

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Events in Node.js

Every action on a computer is an event. Like when a connection is made or a file is opened. Objects in Node.js can fire events, like the readStream object fires events when opening and closing a file:

Example

```
var fs = require('fs');
var rs = fs.createReadStream('./demofile.txt');
rs.on('open', function () {
        console.log('The file is open');
});
```

Events Module

Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.

To include the built-in Events module use the require() method. In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

The EventEmitter Object

You can assign event handlers to your own events with the EventEmitter object.

In the example below we have created a function that will be executed when a "scream" event is fired.

To fire an event, use the emit() method.

Example

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

//Create an event handler:
var myEventHandler = function () {
 console.log('I hear a scream!');
}
//Assign the event handler to an event:
eventEmitter.on('scream', myEventHandler);
//Fire the 'scream' event:
eventEmitter.emit('scream');
```

References:

1. https://www.w3schools.com/nodejs/

2. https://www.tutorialspoint.com/nodejs/

Chapter 6: MongoDB

MongoDB is an open-source document database and leading NoSQL database. MongoDB is written in C++. This tutorial will give you great understanding on MongoDB concepts needed to create and deploy a highly scalable and performance-oriented database.

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

The following table shows the relationship of RDBMS terminology with MongoDB.

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)
	Database Server and Client
Mysqld/Oracle	mongod
mysql/sqlplus	mongo

Sample Document

Following example shows the document structure of a blog site, which is simply a comma separated key value pair.

```
{
  _id: ObjectId(7df78ad8902c)
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100,
  comments: [
    {
      user:'user1',
      message: 'My first comment',
      dateCreated: new Date(2011,1,20,2,15),
      like: 0
    },
    {
      user:'user2',
      message: 'My second comments',
      dateCreated: new Date(2011,1,25,7,45),
      like: 5
    }
  ]
}
```

_id is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide _id while inserting the document. If you don't provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

Example

Create a database called "mydb":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
        if (err) throw err;
        console.log("Database created!");
        db.close();
});
```

Creating a Collection

To create a collection in MongoDB, use the createCollection() method:

Example

Create a collection called "customers":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 dbo.createCollection("customers", function(err, res) {
 if (err) throw err;
 console.log("Collection created!");
 db.close();
 });
});
```

Insert Into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insertOne() method.

A **document** in MongoDB is the same as a **record** in MySQL

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

Example

Insert a document in the "customers" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 var myobj = { name: "Company Inc", address: "Highway 37" };
 dbo.collection("customers").insertOne(myobj, function(err, res) {
 if (err) throw err;
 console.log("1 document inserted");
 db.close();
 });
});
```

Mongodb Find

Find One

To select data from a collection in MongoDB, we can use the findOne() method.

The findOne() method returns the first occurrence in the selection.

The first parameter of the findOne() method is a query object. In this example we use an empty query object, which selects all documents in a collection (but returns only the first document).

Example

Find the first document in the customers collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 dbo.collection("customers").findOne({}, function(err, result) {
 if (err) throw err;
 console.log(result.name);
 db.close();
 });
});
```

Mongodb Query

Filter the Result

When finding documents in a collection, you can filter the result by using a query object.

The first argument of the find() method is a query object, and is used to limit the search.

Example

Find documents with the address "Park Lane 38":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 var query = { address: "Park Lane 38" };
 dbo.collection("customers").find(query).toArray(function(err, result) {
 if (err) throw err;
 console.log(result);
 db.close();
 });
});
```

Sort the Result

Use the sort() method to sort the result in ascending or descending order.

The sort() method takes one parameter, an object defining the sorting order.

Example

Sort the result alphabetically by name:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 var mysort = { name: 1 };
 dbo.collection("customers").find().sort(mysort).toArray(function(err, result) {
 if (err) throw err;
 console.log(result);
 db.close();
 });
});
```

Delete Document

To delete a record, or document as it is called in MongoDB, we use the deleteOne() method.

The first parameter of the deleteOne() method is a query object defining which document to delete.

Note: If the query finds more than one document, only the first occurrence is deleted.

Example

Delete the document with the address "Mountain 21":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 var myquery = { address: 'Mountain 21' };
 dbo.collection("customers").deleteOne(myquery, function(err, obj) {
 if (err) throw err;
 console.log("1 document deleted");
 db.close();
 });
});
```

Drop Collection

You can delete a table, or collection as it is called in MongoDB, by using the drop() method.

The drop() method takes a callback function containing the error object and the result parameter which returns true if the collection was dropped successfully, otherwise it returns false.

Example

Delete the "customers" table:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 dbo.collection("customers").drop(function(err, delOK) {
 if (err) throw err;
 if (delOK) console.log("Collection deleted");
 db.close();
 });
});
```

Update Document

You can update a record, or document as it is called in MongoDB, by using the updateOne() method. The first parameter of the updateOne() method is a query object defining which document to update.

Note: If the query finds more than one record, only the first occurrence is updated.

The second parameter is an object defining the new values of the document.

Example

Update the document with the address "Valley 345" to name="Mickey" and address="Canyon 123":

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 var myquery = { address: "Valley 345" };
 var newvalues = { $set: {name: "Mickey", address: "Canyon 123" } };
 dbo.collection("customers").updateOne(myquery, newvalues, function(err, res) {
 if (err) throw err;
 console.log("1 document updated");
 db.close();
 });
});
```

Limit the Result

To limit the result in MongoDB, we use the limit() method.

The limit() method takes one parameter, a number defining how many documents to return.

Consider you have a "customers" collection:

```
[
{ _id: 58fdbf5c0ef8a50b4cdd9a84 , name: 'John', address: 'Highway 71'},
{ _id: 58fdbf5c0ef8a50b4cdd9a85 , name: 'Peter', address: 'Lowstreet 4'},
{ _id: 58fdbf5c0ef8a50b4cdd9a86 , name: 'Amy', address: 'Apple st 652'},
{ _id: 58fdbf5c0ef8a50b4cdd9a87 , name: 'Hannah', address: 'Mountain 21'},
{ _id: 58fdbf5c0ef8a50b4cdd9a88 , name: 'Michael', address: 'Valley 345'},
{ _id: 58fdbf5c0ef8a50b4cdd9a89 , name: 'Sandy', address: 'Ocean blvd 2'},
{ _id: 58fdbf5c0ef8a50b4cdd9a8a , name: 'Betty', address: 'Green Grass 1'},
{ _id: 58fdbf5c0ef8a50b4cdd9a8b , name: 'Richard', address: 'Sky st 331'},
{ _id: 58fdbf5c0ef8a50b4cdd9a8c , name: 'Susan', address: 'One way 98'},
{ _id: 58fdbf5c0ef8a50b4cdd9a8d , name: 'Vicky', address: 'Yellow Garden 2'},
{ _id: 58fdbf5c0ef8a50b4cdd9a8e , name: 'Ben', address: 'Park Lane 38'},
{ _id: 58fdbf5c0ef8a50b4cdd9a8f , name: 'William', address: 'Central st 954'},
{ _id: 58fdbf5c0ef8a50b4cdd9a90 , name: 'Chuck', address: 'Main Road 989'},
{ _id: 58fdbf5c0ef8a50b4cdd9a91 , name: 'Viola', address: 'Sideway 1633'}
]
```

Limit the result to only return 5 documents:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 dbo.collection("customers").find().limit(5).toArray(function(err, result) {
 if (err) throw err;
 console.log(result);
 db.close();
 });
});
```

Join Collections

MongoDB is not a relational database, but you can perform a left outer join by using the $lookup stage.

The $lookup stage lets you specify which collection you want to join with the current collection, and which fields that should match.

Consider you have a "orders" collection and a "products" collection:

orders

```
[
{ _id: 1, product_id: 154, status: 1 }
]
```

products

```
[
{ _id: 154, name: 'Chocolate Heaven' },
 { _id: 155, name: 'Tasty Lemons' },
 { _id: 156, name: 'Vanilla Dreams' }
]
```

Example

Join the matching "products" document(s) to the "orders" collection:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("mydb");
 dbo.collection('orders').aggregate([
 { $lookup:
 {
 from: 'products',
 localField: 'product_id',
 foreignField: '_id',
 as: 'orderdetails'
 }
 }
 ]).toArray(function(err, res) {
 if (err) throw err;
 console.log(JSON.stringify(res));
 db.close();
 });
});
```

References:

1. https://www.w3schools.com/nodejs/nodejs_mongodb.asp

2. https://www.w3schools.com/nodejs/

3. https://www.tutorialspoint.com/nodejs/