Teaching Guidelines for

Web Programming Technologies

PG-DAC September 2021

Duration: 108 hours (48 classroom hours + 48 lab hours + 12 revision/practice hours)

Objective: To introduce the students to HTML, CSS, JavaScript, XML, JSON, Ajax, Node.js, Express.js, React, React-Redux, and practical relevance of all these technologies.

Evaluation: 100 marks
Weightage: Theory Exam – 40%, Lab exam – 40%, Internals – 20%

Text Books:

- Fundamentals of Web Development, 1e, by Randy Connolly, Ricardo Hoar / Pearson
- MERN Quick Start Guide – Build web applications with MongoDB, Express.js, React, and Node by Eddy Wilson Iriarte Koroliova / Packt

References:

- Internet & World Wide Web : How to Program by Paul Deitel, Henry Deitel & Abbey Deitel / Pearson Education
- XML - How to Program by Deitel et al / Pearson Education
- Ajax in Action by Dave Crane, Eric Pascarello / Dreamtech Press
- JavaScript: The Good Parts by Douglas Crockford / O'Reilly
- Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node by Vasan Subramanian / Apress
- Web Application Security: A Beginner's Guide by Bryan Sullivan & Vincent Liu / Tata McGraw Hill
- W3Schools Tutorials [https://www.w3schools.com/]
- Mozilla Developer Network Web Development Tutorials [https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web]
- Curated Tutorial Links on ES6, React, etc. [https://github.com/markerikson/react-redux-links]

(Note: Each Session is of 2 hours)

Session 1: Architecture of Web
Lecture:

- Brief history of the Internet
- How does the Internet work?
- Internet Protocol; HTTP
- Domain Names; Domain Name Service servers
- HTTP Protocols
  - Difference between HTTP 1.0, HTTP 1.1, and HTTP 2.0
  - Methods – GET, POST, HEAD, PUT, DELETE, etc.
  - Status codes
  - Stateless nature of the protocol and HTTP Session
  - HTTPS
- Architecture of the Web
- Web servers – IIS, Apache server

Lab:

- Exploring different browsers
  - Mozilla Firefox, Google Chrome, Safari
- Exploring different text editors
  - Windows: Notepad++, Linux: Gedit or Vim or Emacs

Session 2: HTML
Lecture:

- Introduction to HTML5
- Introduction to basic HTML Tags
  - Alignment, Headings, Anchor, Paragraph, Image, Lists, Tables, and iFrames
- HTML5
  - New features in HTML5
  - New elements, new attributes, link relations, microdata, ARIA accessibility, objects, events, and Canvas tags
  - HTML5 Validation
  - Audio & Video Support
  - Geo-location Support
- HTML Forms & Controls
  - Input, Text Area, Radio Button, Checkbox, Dropdown, Submit, Reset, Button, etc.
- Introduction to Document Object Model (DOM)

Lab:

- Create a HTML form for building your resume.

Session 3: Cascading Style Sheets (CSS)
Lecture:

- Introduction to CSS, Styling HTML with CSS, Structuring pages with CSS,
- Inline CSS, Internal CSS, External CSS, Multiple styles, CSS Fonts
- CSS Box Model
- id Attribute, class Attribute
- HTML Style Tags
- Linking a style to an HTML document

Lab:

- Apply inline, internal, and external CSS to change colors of certain text portions, bold, underline, and italics certain words in the previously created HTML resume form.

Session 4: JavaScript
Lecture:

- Introduction to JavaScript
- Variables in JavaScript
- Statements, Operators, Comments, Expressions, and Control Structures
- JavaScript Scopes
- Strings, String Methods
- Numbers, Number Methods
- Boolean Values
- Dates, Date Formats, Date Methods
- Arrays, Array Methods

Lab:

- Practice writing basic JavaScript programs for better understanding of the language constructs

Sessions 5 & 6: JavaScript
Lecture:

- Objects, Object Definitions, Object Properties, Object Methods, Object Prototypes
- Functions, Function Definitions, Function Parameters, Function Invocation, Function Closures
- Introduction to Object Oriented Programming in JS
  - Method, Constructor, Inheritance, Encapsulation, Abstraction, Polymorphism

Lab:

- Write a JavaScript program to sort a list of elements by implementing a sorting algorithm.
- Write a JavaScript program to list the properties of a JavaScript object.

Sessions 7 & 8: JavaScript
Lecture:

- Document Object Model (DOM)
  - Object hierarchy in JavaScript
  - HTML DOM, DOM Elements, DOM Events
  - DOM Methods, DOM Manipulation
- Forms, Forms API, Forms Validation
- Regular Expressions
- Errors, Debugging
- Introduction to Browser Dev Tool
- Pushing code quality via JSLint tool

Lab:

- Write a JavaScript function to get First and Last name from the previously created Resume form
- Validate the entire Resume form using client-side JavaScript
- Write a JavaScript function to validate whether a given value is RegEx or not.

Session 9: jQuery
Lecture:

- Introducing to jQuery
- jQuery selectors
- jQuery events
- jQuery animation effects
- jQuery DOM traversal and manipulation
- Data attributes and templates
- jQuery DOM utility functions
- jQuery plugins

Lab:

- Write a jQuery program to get a single element from a selection of elements of a HTML page.
- You are having sample data for the link. Write jQuery code to change the hyperlink and the text of an existing link.
- Write a jQuery program to attach a click and double-click events to all <p> elements.
- Write a jQuery program to hide all headings on a page when they are clicked.
  - Also find the position of the mouse pointer relative to the left and top edges of the document.

Sessions 10 & 11: JSON & Ajax
Lecture:

- JSON: JavaScript Object Notation (JSON)
  - Introduction and need of JSON
  - JSON Syntax Rules
  - JSON Data - a Name and a Value,
  - JSON Objects, JSON Arrays, JSON Files
  - JSON parsing
- Ajax
  - Introduction to Ajax
  - Ajax Framework
  - Ajax Architecture
  - Web services and Ajax
  - Ajax using JSON and jQuery

Lab:

- Create a page showing live score/feed using Ajax and JSON from a live sport/news service end-point given by the faculty

Session 12: Introduction to Node.js
Lecture:

- Introduction to Node.js
- Browser JS vs. Node.js
- ECMAScript 2015 (ES6)
- Node.js REPL

Lab:

- Install Node.js 12.x.x LTS version on your machine
- Write a recursive function in Node.js
- Write a Node program that prints all the numbers between 1 and 100, each on a separate line. A few caveats:
  - if the number is divisible by 3, print "foo"
  - if the number is divisible by 5, print "bar"
  - if the number is divisible by both 3 and 5, print "foobar"

Sessions 13 & 14: Node.js Asynchronous Programming
Lecture:

- Introduction to Asynchronous programming and callbacks
- Promises and async & await
- The Event Loop and Timers

Lab:

- Assignment on JavaScript callback functions
- Assignment on Timers, Promises, and Async & Await

Session 15: Node.js Modules
Lecture:

- Understanding Node modules, exports, and require
- Introduction to npm
  - package.json and package-lock.json files
  - Install, update, and manage package dependencies
  - Local and global packages

Lab:

- Create a module and import it in other programs
- Install a module/package using npm

Session 16: Node.js Modules – *fs* and *http*
Lecture:

- File I/O – Sync & Async Methods

- HTTP Module – Building an HTTP server
- Developing a Node web application

Lab:

- Write a program to create a new file and write some content to it in synchronous mode and read and display file contents on standard output in async mode
- Build a simple Node.js web application serving both HTTP GET and POST methods

Session 17: Introduction to Express

Lecture:

- Introduction to Express
- Getting started with Express
- Application, Request and Response Objects
- Routes and Middlewares
- Templates, Template Engines, and Rendering Views

Lab:

- Use Node and Express to write a simple web application that consists of at least 5 route implementations
- Rebuild any previous Node assignment using Express and a template engine

Session 18: Introduction to React

Lecture:

- Introduction to React
- Getting started with React
- React Elements and React Components
- Function and Class Components
- Working with React Components and Props
  - Compose components
  - Render components
  - Declutter components

Lab:

- Rebuild any previous plain HTML lab assignment using React
- Build a React Clock app showing time (hh:mm:ss) of any three countries

Sessions 19 & 20: React

Lecture:

- Introduction to State and Lifecycle
- Stateful components and lifecycle methods
- Props vs. State vs. Context
- Handling events
- Conditional rendering

Lab:

- Implement the following items in the React Clock app
  - Update the time (hh:mm:ss) using State and Lifecycle methods
  - Add a close function on each rendered clock component
  - Assign background color of rendered clock components based on AM, PM

Session 21: React

Lecture:

- Lists and Keys
  - Rendering Multiple Components
  - Basic List Component
- Working with forms and inputs
- Refs and the DOM
- Lifting state up

Lab:

- Implement and integrate a new feature in the React Clock app where one can select a country time zone from dropdown list and click on “Add” button to render it.

Session 22: React

Lecture:

- Error Boundaries
- Composition vs. Inheritance
  - Containment
  - Specialization
- Thinking in React

Lab:

- Implement error boundaries at appropriate places in the React Clock app

Session 23: Introduction to React-Redux

Lecture:

- Introduction to Redux
- Actions, Reducers, and Stores
- Usage with React

Lab:

- Make necessary changes in the design and implementation of React Clock app using React-Redux to maintain the application state.

Session 24: Responsive Web Design & Web Security

Lecture:

- Introduction of UI Scripting
- The Best Experience for All Users
  - Desktop, Tablet, Mobile
- Bootstrap
  - Overview of Bootstrap, Need to use Bootstrap
  - Bootstrap Grid System, Grid Classes, Basic Structure of a Bootstrap Grid
  - Typography
  - Components – Tables, Images, Jumbotron, Wells, Alerts, Buttons, Button Groups, Badges/Labels, Progress Bars, Pagination, List Groups, Panels, Dropdowns, Collapse, Tabs/Pills, Navbar
  - Forms, Inputs
  - Bootstrap Themes, Templates
- Web Security
  - Introduction to Web security
  - SQL Injection, Cross-Site Scripting (XSS)
  - JSON and Security Concerns, Cross Site Request Forgery (CSRF), Injection Attacks
  - Security Standards (OWASP)

Lab:

- Update the design of the Resume form using Bootstrap

HTML

What is HTML?

- HTML(Hyper Text Markup Language)
  - is a language for describing web pages.
  - not a programming language
  - uses markup tags to describe web pages.
- Most Web documents are created using HTML.
  - Documents are saved with extension .html or .htm.
- Markup?
  - Markup Tags are strings in the language surrounded by a < and a > sign.
  - Opening tag: <html> Ending tag: </html>
  - Not case sensitive.
  - Can have attributes which provide additional information about HTML elements on your page. Example
    - <body bgcolor="red">
    - <table border="0">

HTML

- An HTML document appears as follows:

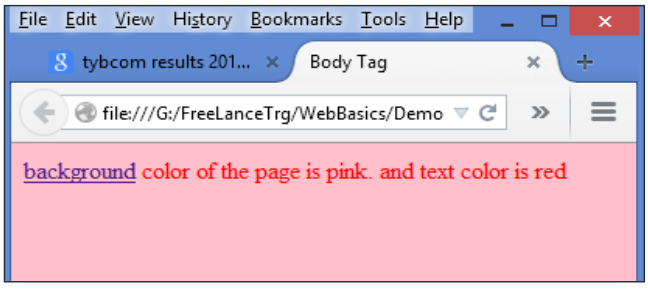
```
<!DOCTYPE HTML>
<html>
        <head>
            <title>Title of page</title>
        </head>
        <body>
                This is my first homepage. <b>This text is bold</b>
        </body>
</html>
```

  - HTML Head Section: contain information about the document. The browser does not display this information to the user. Following tags can be in the head section: <base>, <link>, <meta>, <script>, <style>, and <title>.
  - HTML Body Section: defines the document's body. Contains all the contents of the document (like text, images, colors, graphics, etc.).

HTML Body Section

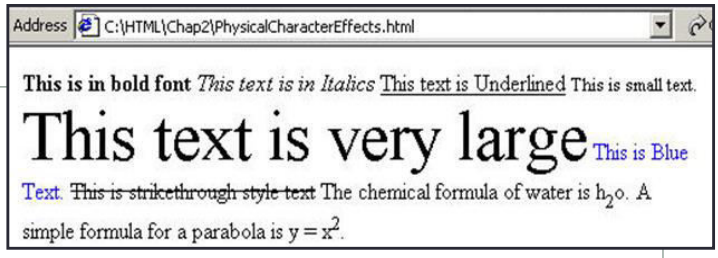
- <body> Element:
  - Each document can have at most one <body> element.
  - Attributes used in <body> element are:
    - BGCOLOR: Gives a background color to HTML page.
    - BACKGROUND: Use to specify images to the BACKGROUND.
    - TEXT: Specifies text color throughout the browser window
    - LINK, ALINK, VLINK: Used to specify link color, active link color & visited link color
  - Examples:
    - <body text="red"> OR <body text="#FF0000">
    - <body link="red" alink="blue" vlink="purple">
    - <body bgcolor="black"> OR <body bgcolor="#000000">
    - <body background="http://www.mysite.edu/img1.gif">
    - <body background="symbol.gif" text="red" link="blue" bgproperties="fixed">

```
<HTML>
 <HEAD>
   <TITLE>Body Tag</TITLE>
 </HEAD>
 <BODY  BGCOLOR="pink" text="red"
          alink="green" link="yellow">
    <a href="body.html">background</a>
       color of the page is pink. and text color is red
 </BODY>
</HTML>
```



Physical Character Effects

- Bold Font: …
- Italic: …
- Underline: <u>…</u>
- Strikethrough: <strike> or <s>
- Fixed width font: <tt> - normal text in fixed-width font
- Subscript: <sub>
- Superscript: <sup>




```
<html>
<body>
<b>This is in bold font</b>
<i>This text is in Italics</i>
<u>This text is Underlined</u>
<small>This is small text.</small>
<font size=7>This text is very large</font>
<font color="Blue">This is Blue Text.</font>
<strike>This is strikethrough style text</strike>
The chemical formula of water is h<sub>2</sub>o.
A simple formula for a parabola is y = x<sup>2</sup>.
</body>
</html>
```


Document (Body) Contents

- Body Text
  - HTML truncates spaces in your text.
  - Use
 to insert new lines.
  - Use <p> tag to create paragraphs.
  - Use to hold division or a section in an HTML document
  - Use as an inline container to mark up a part of a text, or a part of a document
- Comments in HTML Document
  - Increase code readability; Ignored by the browser.
  - Example of HTML comment: <!-- This is a Sample HTML Comment -->

Logical Character Effects

- Heading Styles:
  - **<hn>.....</hn>**
    - Value of n can range from **1 to 6**
  - <h1 align="center">This is level 1 heading</h1>

Using Heading styles

```
<html>
<head><title>This is the first html page</title>
<body>This is the first html page created
<h1 align="left">This is level 1 heading</h1>
<h2 align="right">This is level 2 heading</h2>
<h3 align="center">This is level 3 heading</h3>
</body>
</head>
</html>
```

Horizontal Lines in a Web Page

- <hr> - horixontal Rule. Attributes:
  - Size: Line thickness <hr size="5">
  - Width: Line width either in pixels or % of browser window
    - <hr width="100"> or <hr width="60%">
  - Align: Alignment values can be left, center or right <hr align="center">
  - Color: to display colored horizontal lines. Eg <hr color="red">

This paragraph contains a lot of lines in the source code, but the browser ignores it.

Notice the horizontal rule occuping 50 % of the window width.

This paragraph contains
line breaks in the source code
so this is the third line displayed within the paragraph.

```
<body> <p>
This paragraph contains a lot of lines
in the source code,
but the browser ignores it.  </p>
<hr size="2" width="50%" color="blue" align = "left" >  <p>
Notice the horizontal rule occupying 50 % of the window width.
</p> This paragraph contains <br> line breaks in the
source code <br> so this  is the third line displayed  within the paragraph.
</body>
```

Numbered List (Ordered List)

- Automatically generate numbers in front of each item in the list
  - Number placed against an item depends on the location of the item in the list.
  - Example:

```
<body>
  <ol>
    <li>INDIA</li>
    <li>SRILANKA</li>
  </ol>
</body>
```

1. INDIA
2. SRILANKA

  - To start an ordered list at a number other than 1, use **START** attribute:
    Example : <ol start=11>

- Select the type of numbering system with the **type** attribute. Example:
  - A-Uppercase letters. <OL TYPE=A>
  - a-Lowercase letters. <OL TYPE=a>
  - I-Uppercase Roman letters
  - i-Lowercase Roman letters
  - 1-Standard numbers, default

Bulleted List (Unordered List)

- Example:

```
  <ul>
    <li>Latte</li>
    <li>Expresso</li>
    <li>Mocha</li>
</ul>
```

By default you get bullet points(DISC)

- To change the type of bullet used throughout the list, place the TYPE attribute in the <UL> tag at the beginning of the list.
  - DISC (default) gives bullet in disc form.
  - SQUARE gives bullet in square form.
  - CIRCLE gives bullet in circle form.

```
<ul>
  <li type='disc'>Latte</li>
  <li type='square'>Expresso</li>
  <li type='circle'>Mocha</li>
</ul>
```

Adding Image

- Images are added into a document using <img> tag.
- Attributes of <img> tag are:
  - alt: Alternative text to display for non-graphical browsers.
  - align: Image horizontal alignment. Valid values: left, right, center.
  - width/Height: Sets the width and height of the image.
  - src: Specifies the image path or source.

```
<IMG src="home.png“  height="100“   width="200"   >
```

Table

- Use <table> tag to create a table.
- Table Attributes:
  - Border: <table border=“2”>……</table>
  - Align: defines the horizontal alignment of the table element.
    - Values of the align attribute are right, left and center. <table align=“center”>
  - Width: defines the width of the table element.
    - <table width=“75%”>……</table>
    - <table width=“400”>……</table>
- Example:

```
<table border="1">
    <tr border="1">
        <td>Row 1, cell 1</td>
         <td>Row 1, cell 2</td>
    </tr>
 </table>
```

Row 1, cell 1	Row 1, cell 2

Table Data

- An HTML table has two kinds of cells:
  - Header Cells <th>: Contain header information (created with <th> element) ; The text is bold and centered.
  - Standard Cells <td>: Contain data (created with the td element) ; The text is regular and left-aligned.

```
<table>
  <tr> <th>Column1 Header</th> <th>Column2 Header</th></tr>
  <tr> <td>Cell 1,1</td> <td>Cell 1,2</td> </tr>
  <tr> <td>Cell 2,1</td> <td>Cell 2,2</td> </tr>
</table>
```

- You can insert a **bgcolor** attribute in a <table>, <td>, <th> or <tr> tag to set the color of the particular element.

<table bgcolor="cyan">
<tr bgcolor="blue">

```
<table bgcolor="cyan">
  <tr bgcolor="blue">
     <th bgcolor="red">Header 1</th> <th>Header 2</th>
  </tr>
  <tr> <td bgcolor="green">data 1</td> <td>data 2</td> </tr>
</table>
```

Header 1	Header 2
data 1	data 2

How Does a Hyperlink Work?

- Hyperlinks access resources on the internet.
- Create a link with <a href=“”> (anchor)

Login Here
Hello, Welcome to My Site
I have some older information about this subject.

```
<a href="http://www.mysite.com/login.htm">Login Here</a>
Hello, Welcome to  <a href="welcome.htm">My Site</a>
I have some <a href="http://www.state.edu/info/info.htm">
older information</a> about this subject.
```

- Hyperlinks in Lists Items & Table Elements

```
<ul>
<li><a href=home.html>mumbai</a></li>
<li><a href=home.html>pune</a></li>
<li><a href=home.html>nasik</a></li>
</ul>
```

- mumbai
- pune
- nasik

```
<table border=1>
<tr><th>team<th>points<th>grade</tr>
<tr><td><a href=home.html>mumbai</a></td><td>90</td><td>a</td></tr>
<tr><td><a href=home.html>pune</a></td><td>86</td><td>b</td></tr>
<tr><td><a href=home.html>nasik</a></td><td>80</td><td>c</td></tr>
</table>
```

team	points	grade
mumbai	90	a
pune	86	b
nasik	80	c

HTML Forms : Types of Form Fields

```
<form method="get/post" action="URL">
        Field definitions
</form>
```

- <input> tag is used to create form input fields.
  - Type attribute of <input> tag specifies the field type
    - Single line text box <input type="text">
    - Password field <input type="password">
    - Hidden field <input type="hidden">
    - Radio button <input type="radio">
    - Checkbox <input type="checkbox">
    - File selector dialog box <input type="file">
    - Button <input type="button">
    - Submit/Reset <input type="submit/reset">
- <textarea>
- <select>
- <button>

Text Field

- **Single Line Text Fields**: used to type letters, numbers, etc. in a form.

<INPUT TYPE="type" NAME="name" SIZE="number" VALUE="value" maxlength=n>

  - Eg:

```
<form>
      First name: <input type="text" name="firstname" value="fname">
      Last name:<input type="text" name="lname">
</form>
```

- Text Area (Multiple Line Text Field)
  - A text area can hold an unlimited number of characters. Text renders in a fixed-width font (usually Courier).
  - You can specify text area size with cols and rows attributes.

<textarea name="name" rows="10" cols="50" [disabled] [readonly]>
        Default-Text
</textarea>

For example, <textarea rows="4" cols="50"> creates a textarea that can display 4 lines of text and 50 average-width characters per line1

```
<textarea name="address" rows=5 cols=10>
            Please write your address
</textarea>
<textarea rows="4" cols="20">
```

Check Box

- Lets you select one or more options from a limited number of choices.

  <input type="checkbox" name="name" value="value" [checked] [disabled]> Checkbox Label

  - Content of value attribute is sent to the form's action URL.

```
<input type="checkbox" name="color1" value="0"/>Red
<input type="checkbox" name="color3" value="1" checked/>Green
```

Radio Buttons

<input type="radio" name="name" value="value" [checked] [disabled]> Radio Button Label

- *Content of the value attribute is sent to the form's action URL.*

```
<form>
<input type="radio" name="sex" value="male"/>Male<br>
<input type="radio" name="sex" value="female"/> Female
</form>
```

Hidden Field

- Allows to pass information between forms:
  - <input type="hidden" name="name" size="n" value="value"/>
  - <input type="hidden" name="valid_user" size="5" value="yes"/>

Password Fields

- <input type="password" name="name" size=n value="value" [disabled]>
  - Eg: Enter the password:<input type="password" name="passwd" size=20 value="abc">

File Selector Dialog Box

- <input type="file" name="name" size="width of field" value="value">

```
<FORM>
   Please select file that you want to upload:
   <INPUT name="file" type="file"> <BR>
   <INPUT type="submit" >
</FORM>
```

Action Buttons

- To add a button to a form use:
  - <input type="button" name="btnCalculate" value="Calculate"/>
- To submit the contents of the form use:
  - <input type="submit" name="btnSubmit" value="Submit"/>
- To reset the field contents use:
  - <input type="reset" name="btnReset" value="Reset"/>

Drop-Down List

<select name="name" multiple="true/false" size=n [disabled]>
<option [selected] [disabled] [value]>Option 1</option>…
</select>

- Multiple: States if multiple element selection is allowed.
- Size: Number of visible elements.
- Disabled: States if the option is to be disabled after it first loads.

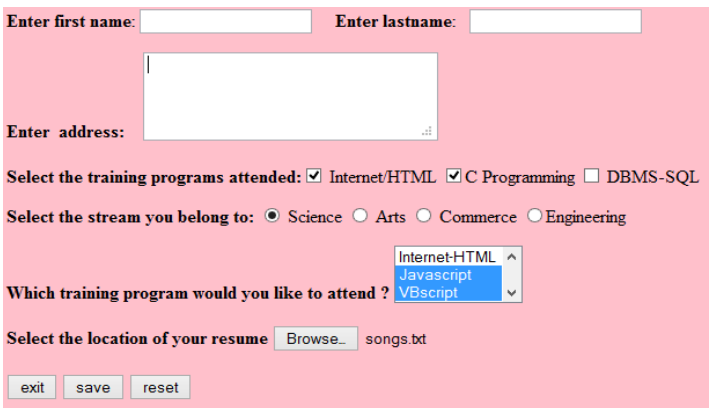
In this case, the button will be displayed, but the user will not be able to interact with it.

Eg:

```
<form>
<select multiple size="3" name="pref">
<option value="ih"   selected>Internet-HTML</option>
<option value="js">Javascript</option>
<option   value="vbs">VBscript</option>
<option value="as">ASP</option>
<option value="xm">XML</option>
<option value="jv">JAVA</option>
<option value="jsp">jsp</option>
</select>
</form>
```

Complete example

```
<html><head><title>form examples</title></head>
<body bgcolor="pink">
<form name="form1" action="store.html" method="post">
<p>
<strong>Enter first name</strong>: <input name="username">    
<strong>Enter lastname</strong>:   <input  maxlength="30" name="surname"></p>
<p><strong>Enter  address:</strong>    
<textarea name="addr" rows="3" readonly value="sjdshd"></textarea>
<br><br>
<strong>Select the training programs attended:</strong>
<input type="checkbox" value="internet/html" name="internet-html"> Internet/HTML
<input type="checkbox" checked value="c programming" name="c-programming">C Programming
<input type="checkbox" value="dbms-sql" name="dbms-sql"> DBMS-SQL </p>
<p><strong>Select the stream you belong to:</strong>
<input type="radio" value="science" name="s-grp"> Science
<input type="radio" value="arts" name="s-grp"> Arts
<input type="radio" value="commerce" name="s-grp"> Commerce
<input type="radio" value="oth2" name="s-grp">Engineering </p>
<strong>Which training program would you like to attend ?</strong>
 <select size="5" multiple name="pref">
   <option value="ih"   selected>Internet-HTML
   <option value="js">Javascript
   <option   value="vbs">VBscript
   <option value="as">ASP
   <option value="xm">XML
   <option value="jv">JAVA
   <option value="jsp">jsp</option>
</select> <br>
<input type="button" value="exit" name="but">
<input type="submit" value="save">
 <input type="reset" value="reset">
</form></body></html>
```



HTML5

What's new in HTML5?

- HTML5 offers new enhanced set of tags
  - New Content Tags : <nav>,<section>,<header>,<article>,<aside>,<summary>
  - New Media Tags : <video>,<audio>
  - New Dynamic drawing : <canvas>graphic tag
  - New form controls, like calendar, date, time, email, url, search
- Support for JavaScript APIs
  - Canvas element for 2D drawing API
  - Video and audio APIs
  - APIs to support offline storages
  - The Drag & Drop APIs
  - The Geolocation API
  - Web workers, WebSQL etc

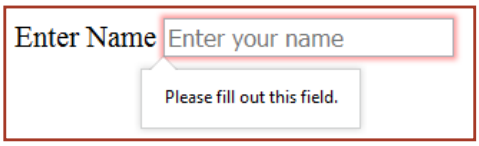
HTML5 Attributes for <input>

- A Form is one of the most basic and essential feature of any web site
  - HTML5 brings 13 new input types
  - HTML5 introduces these data types via the <input type="_NEW_TYPE_HERE_"/> format

- **Placeholder** - A placeholder is a textbox that hold a text in lighter shade when there is no value and not focused
  - <input id="first_name" placeholder="This is a placeholder">
  - Once the textbox gets focus, the text goes off and you shall input your own text

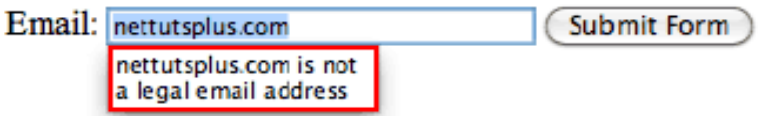
- **AutoFocus** - Autofocus is a Boolean attribute of form field that make browser set focus on it when a page is loaded
  - <input id ="Text2" type="text" autofocus/>

- **Required** - A "Required Field" is a field that must be filled in with value before submission of a form
- <input name="name" type="text" required />

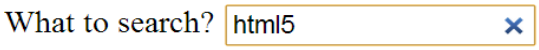


New Form elements

- **Email** - This field is used to check whether the string entered by the user is valid email id or not.
  - <input id="email" name="email" type="email" />



- **Search** - used for search fields (behaves like a regular text field).
  - <input id="mysearch" type="search" />



- **Tel** - used for input fields that should contain a telephone number.
  - <input type="tel" name="usrtel">
- **url** - is used for input fields that should contain a URL address.
  - Depending on browser support, the url field can be automatically validated when submitted.
- **color** – displays a color palette

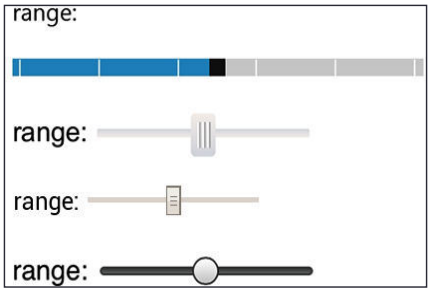
New Form elements

- **Number** - used for input fields that should contain a numeric value.
  - Min and max parameters provided to limit the values.
  - Browser will treat it as simple textfield if it doesn't support this type.
  - <input id="movie" type="number" value="0"/>
  - <input type="number" min="0" max="50" step="2" value="6">

value="6": The value attribute specifies the default value for the input. When the page loads, the number input will display 6 as its initial value.

step="2": The step attribute specifies the legal number intervals for the input. Here, it's set to 2, which means the numbers should increase or decrease in increments of 2.

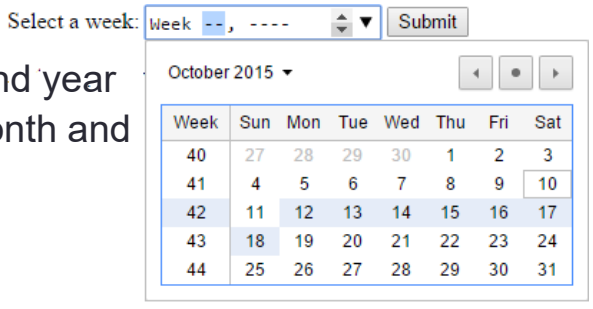
- **Range** - used for input fields that should contain a value within a range
  - Browser will treat it as simple textfield if it doesn't support this type
  - <input id="test" type="range"/>
  - <input type="range" min="1" max="20" value="0">

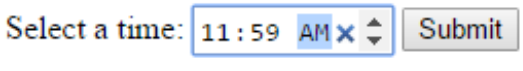


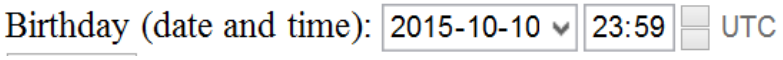
New Form elements

- **Date** - used for input fields that should contain a date.
  - Depending on browser support, a date picker can show up in the input field.
  - <input id="meeting" type="date" />

- **month** - Selects month and year
- **week** - Selects week and year
- **time** - Selects time (hour and minute)
- **datetime** - Selects time, date, month and year
- **datetime-local** - Selects time, date, month and year (local time)



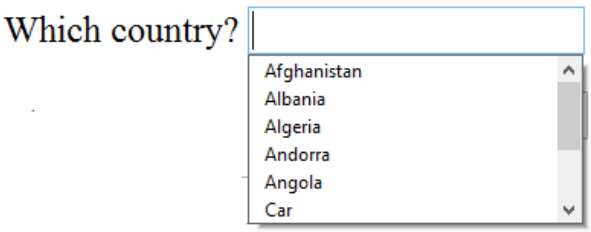




New Form elements

- Data List - specifies a list of options for an input field. The list is created with option elements inside the datalist.
  - seems like type-ahead auto suggest textbox as you can see in Google search box

```
<input id="country_name"
       name="country_name"
       type="text“
       list="country" />

<datalist id="country">
<option value="Australia">
<option value="Austria">
<option value="Algeria">
<option value="Andorra">
<option value="Angola">
</datalist>
```



Audio

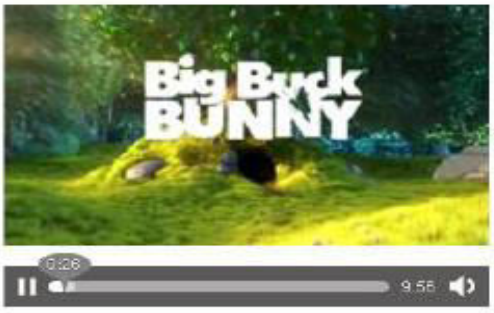
- Until now, there has never been a standard for playing audio on a web page.
  - Today, most audio is played through a audio plugin (like Microsoft Windows Media player, Microsoft Silverlight ,Apple QuickTime and the famous Adobe Flash).
  - However, not all browsers have the same plugins.
  - HTML5 specifies a standard way to include audio, with the audio element; The audio element can play sound files, or an audio stream.
  - Currently, there are 3 supported formats for the audio element: Ogg Vorbis, MP3, .webm and WAV
  - Other properties like auto play, loop, preload area also available

```
<audio controls>
<source src="vincent.mp3" type="audio/mpeg"/>
<source src="vincent.ogg" type="audio/ogg"/>
</audio>
```

Video

- Until now, there hasn't been a standard for showing video on a web page.
  - Today, most videos are shown through a plugin (like Flash). However, not all browsers have the same plugins.
  - HTML5 specifies a standard way to include video with the video element
  - Supported video formats for the video element : Ogg, MP4, WebM, .flv, .avi
  - Attributes : width, height, poster, autoplay, controls, loop, src

```
<video controls="controls" width="640" height="480" src="bunny.mp4" />
  Your browser does not support the video element.
</video>
```



Canvas

- A canvas is a rectangle in your web page within which you can use JavaScript to draw shapes
  - Canvas can be used to represent something visually in your browser like Simple Diagrams, Fancy user interfaces, Animations, Charts and graphs, Embedded drawing applications, Working around CSS limitations
  - The canvas element has several methods for drawing paths, boxes, circles, characters, and adding images.
  - The canvas element has no drawing abilities of its own. All drawing must be done inside a JavaScript

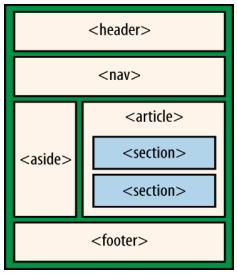
```
<canvas id="myCanvas" width="200" height="100">
</canvas>
```

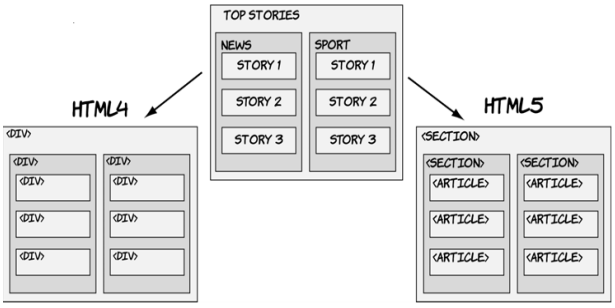
```
<canvas id="myCanvas"></canvas>
<script type="text/javascript">
  var canvas=document.getElementById('myCanvas');
  var ctx=canvas.getContext('2d');
  ctx.fillStyle='#FF0000'; Red
  ctx.fillRect(0,0,80,100);
</script>fillRect(x, y, width, height)
```

ctx=canvas.getContext('2d');:
This line gets the drawing context on the canvas. The getContext('2d') method returns a drawing context on the canvas, which is where the actual drawing will be done

New Semantic Elements

- **<section> :** can be used to thematically group content, typically with a heading.



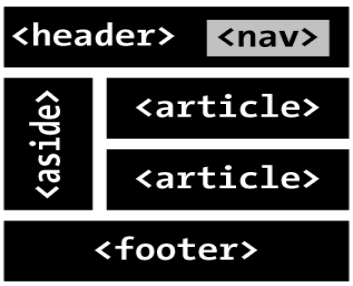


- **<article>:** element represents a self-contained composition in a document, page, application, or site that is intended to be independently distributable or reusable
  - Eg a forum post, a magazine or newspaper article, a blog entry, a user-submitted comment
- **<nav>**: Represents a major navigation block. It groups links to other pages or to parts of the current page.
- **<Header>:** tag specifies a header for a document or section. Can also be used as a heading of an blog entry or news article as every article has its title and published date and time

New Semantic Elements

- **<aside>:** The "aside" element is a section that somehow related to main content, but it can be separate from that content header and footer element in an article.
- **<footer>:** Similarly to "header" element, "footer" element is often referred to the footer of a web page.
  - However, you can have a footer in every section, or every article too

- **<figure>:** The <figure> tag specifies self-contained content, like illustrations, diagrams, photos, code listings, etc.
  - This element can optionally contain a *figcaption* element to denote a caption for the figure.



Fat footer on the w3c site

What is CSS?

- Cascading Styles Sheets - a way to style and present HTML.
  - HTML deals with content and structure, stylesheet deals with formatting and presentation of that document.
  - Allows to control the style and layout of multiple Web pages all at once.
- Example:

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

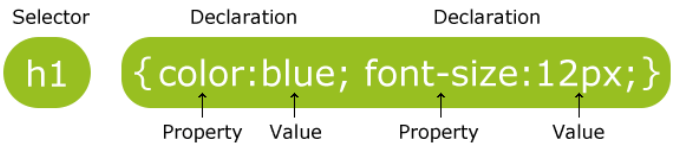
Messy HTML

```
<p class="MyStyle">My CSS styled text</p>
…….
<style>
   .myStyle {
       font-family: Verdana, Arial, Helvetica, sans-serif; font-weight: bold; color: #FF0000; }
</style>
```

- Why CSS?
  - saves time
  - Pages load faster
  - Easy maintenance
  - Superior styles to HTML

CSS Syntax

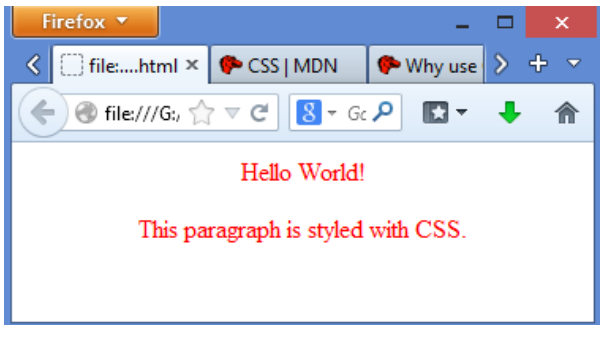
- A CSS rule has two parts: a selector, and one or more declarations:



- "HTML tag" { "CSS Property" : "Value" ; }

```
<head>
<style>
p
 {
   color:red;
   text-align:center;
 }
</style>
</head>
<body>
<p>Hello World!</p>
<p>This paragraph is styled with CSS.</p>
</body>
```

Selects all the paragraphs in HTML document and gives them style



Three Ways to Insert CSS

- Embedded Style Sheets
  - Style defined between <STYLE>..</STYLE> tags. <STYLE> tags appear either in <HEAD> section or between </HEAD> and <BODY> tags.
  - Example in previous slide
- Linked Style Sheets
  - Separate files (extn .CSS) that are linked to a page with the <LINK> tag. Are referenced with a URL. Placing a single <LINK> tag within the <HEAD> tags links the page that needs these styles.
  - <head> <link rel="stylesheet" type="text/css" href="mystyle.css"> </head>
- Inline Style Sheets
  - Only applies to the tag contents that contain it. Used to control a single tag element. Tag inherits style from its parent. Egs.
  - This is a paragraph.
  - A new background and font color with inline CSS

Demo: Link Style Sheet

```
<html>
<head>
<link rel=stylesheet href="linked_ex.css"
type="text/css">
</head>
<body>
<h2>This is Level 2 Heading, with style</h2>
<h1>This is Level 1 Heading, with style</h1>
<h3>This is Level 3 Heading, with style</h3>
<h4>This is Level 4 Heading, without style</h4>
</body>
</html>
```

```
body {   background: black;
         color:green
      }
h1 {    background: orange;
        font-family: Arial, Impact;
        color: blue;
        font-size:30pt;
         text-align: center
    }
h2, h3 {   background: gold;
           font-family: Arial, Impact;
           color:red   }
```

linked_ex2.css

Inline Style Sheet

- All style attribute are specified in the tag it self. It gives desired effect on that tag only. Doesn't affect any other HTML tag.

```
<body style="background: white; color:green">
<h2 style="background: gold; font-family: Arial, Impact; color:red">
This is Level 2 Heading, with style</h2>
<h1 style="background: orange; font-family: Arial, Impact; color: blue;font-size:30pt; text-align: center">This is Level 1 Heading, with style</h1>
<h3 style="background: gold; font-family: Arial, Impact;color:red">
This is Level 3 Heading, with style</h3>
<h4>This is Level 4 Heading, without style</h4>
<h1>This is again Level 1 heading with default styles</h1>
</body>
```

Types of selectors

- HTML selectors
  - Used to define styles associated to HTML tags. – already seen!!!!
- Class selectors
  - Used to define styles that can be used without redefining plain HTML tags.
- ID selectors
  - Used to define styles relating to objects with a unique id

ID Selector & Style Sheet Classes

```
<style>
    H1.myClass {color: blue}
    .myOtherClass { color: red; text-align:center}
</style>
<body style="background: white; color:green">
<H1 class="myClass">This text would be blue</H1>
<p class="myOtherClass">The text would be in red</P>
<H3 class="myOtherClass">This is level 2 heading</H3>
<table class=myotherClass border width=100%>
<td>Data 1</td><td>Data 2</td></table>
<h3>This is Level 4 Heading, without style</h3>
<h1>This is again Level 1 heading with default styles</h1>
```

```
<html> <head>
<style>
#para1 {
text-align:center;
color:red;
}
</style> </head>
<body>
<p id="para1">Hello World!</p>
<p>This paragraph is not affected by the
style.</p>
</body>
</html>
```

Hello World!

This paragraph is not affected by the style.

CSS Box Model

- All HTML elements can be considered as boxes.
- The CSS box model is essentially a box that wraps around HTML elements, and it consists of:
  - Margin - Clears an area around the border. The margin does not have a background color, it is completely transparent
  - Border - A border that goes around the padding and content. The border is affected by the background color of the box
  - Padding - Clears an area around the content. The padding is affected by the background color of the box
  - Content - The content of the box, where text and images appear

CSS Border

- The border property is a shorthand for the following individual border properties: border-width, border-style (required), border-color

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

This is some text in a paragraph.

```
<style type="text/css">
.box {
    width: 100px;
    height: 100px;
    border-color: Blue;
    border-width: 2px;
    border-style: solid;
}
</style>

<div class="box">
        Hello, world!
</div>
```

```
<div class="box" style="border-style: dashed;">Dashed</div>
<div class="box" style="border-style: dotted;">Dotted</div>
<div class="box" style="border-style: double;">Double</div>
<div class="box" style="border-style: groove;">Groove</div>
<div class="box" style="border-style: inset;">Inset</div>
<div class="box" style="border-style: outset;">Outset</div>
<div class="box" style="border-style: ridge;">Ridge</div>
<div class="box" style="border-style: solid;">Solid</div>
```

Hello, world!

CSS Styling

CSS Background

- CSS background properties are used to define the background effects of an element.
- CSS properties used for background effects:

background-color	Sets the background color of an element
background-image	Sets the background image for an element
background-position	Sets the starting position of a background image
background-repeat	Sets how a background image will be repeated

With CSS, a color is specified by:
- *a HEX value - like "#ff0000"*
- *an RGB value - like "rgb(255,0,0)"*
- *a color name - like "red"*

- Example:
  - div {background-color:#b0c4de;}
  - body {background-image:url('paper.gif');}
  - body {background-image:url('gradient2.png');background-repeat:repeat-x;}
  - body {background-image:url('img_tree.png'); background-repeat:no-repeat; background-position:right top; }

- The background-repeat property sets if/how a background image will be repeated.
  - By default, (repeat) : a background-image is repeated both vertically and horizontally.
  - no-repeat : The background-image is not repeated. The image will only be shown once

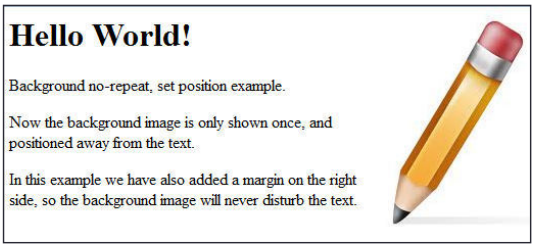
Demo : CSS Background

```
body {
    background-image: url("img_tree.gif"), url("img_flwr.gif");
    background-color: #cccccc;
}
```

```
body {
     background: #00ff00 url("smiley.gif") no-repeat fixed center;
}
```

See here we add style tag in the head tag

```
<html>
<head><style>
body {
background-image:url('img_tree.png');
background-repeat:no-repeat;
background-position:right top;
margin-right:200px;
}
</style> </head>
<body>
<h1>Hello World!</h1>
<p>Background no-repeat, set position example.</p>
<p>Now the background image is only shown once, and positioned away from the text.</p>
<p>In this example we have also added a margin on the right side, so the background image will never disturb the text.</p>
</body></html>
```



CSS Text

- CSS Text Properties

color	Sets the color of text
direction	Specifies the text direction/writing direction
letter-spacing	Increases or decreases the space between characters in a text
text-align	Specifies the horizontal alignment of text
text-decoration	Specifies the decoration added to text
text-indent	Specifies the indentation of the first line in a text-block
text-shadow	Specifies the shadow effect added to text
text-transform	Controls the capitalization of text
white-space	Specifies how white-space inside an element is handled
word-spacing	Increases or decreases the space between words in a text

```
<style>
h1 {text-decoration:overline;}
h2 {text-decoration:line-through;}
h3 {text-decoration:underline;}
</style>
<body>
<h1>This is heading 1</h1>
<h2>This is heading 2</h2>
<h3>This is heading 3</h3>
```

similar to strikethrough

This is heading 1

~~**This is heading 2**~~

This is heading 3

Demo :

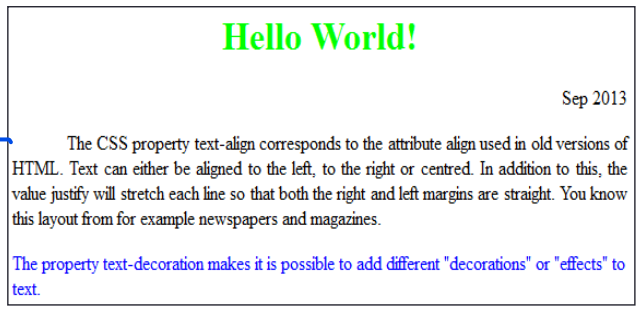
```
<style>
p.uppercase {text-transform:uppercase;}
p.lowercase {text-transform:lowercase;}
p.capitalize {text-transform:capitalize;}
</style>
<body>
<p class="uppercase">This is some text.</p>
<p class="lowercase">This is some text.</p>
<p class="capitalize">This is some text.</p>
</body>
```

THIS IS SOME TEXT.

this is some text.

This Is Some Text.

```
<head> <style>
h1 {text-align:center;color:#00ff00;}
p.date {text-align:right;}
p.main {text-align:justify;}
p.ex {color:rgb(0,0,255);}
p.indent {text-indent:50px;}
</style> </head>
<body>
<h1>Hello World!</h1>
<p class="date"> Sep 2013</p>
<p class="main indent">The CSS property text-align corresponds to the attribute align used in old versions of HTML. Text can either be aligned to the left, to the right or centred. In addition to this, the value justify will stretch each line so that both the right and left margins are straight. You know this layout from for example newspapers and magazines. </p>
<p class="ex">The property text-decoration makes it is possible to add different "decorations" or "effects" to text. </p>
</body>
```



More examples

- P { word-spacing:30px; }

This is some text.

- h1 {text-shadow:2px 2px #FF0000;}

Text-shadow effect

It has +ve and -ve values.
+ve values = shadow right
-ve values = shadow left

gap between two sentences

- p.small {line-height:70%;}
- p.big {line-height:200%;}

This is a paragraph with a standard line-height.
This is a paragraph with a standard line-height.
The default line height in most browsers is about 110% to 120%.

This is a paragraph with a smaller line-height.
This is a paragraph with a smaller line-height.
This is a paragraph with a smaller line-height.
This is a paragraph with a smaller line-height.

This is a paragraph with a bigger line-height.

This is a paragraph with a bigger line-height.

This is a paragraph with a bigger line-height.

This is a paragraph with a bigger line-height.

- h1 {letter-spacing:2px;}
- h2 {letter-spacing:-3px;}

This is heading 1

This is heading 2

CSS : Styling Fonts

- CSS font properties define the font family, boldness, size, and the style of a text.

font-family	Specifies the font family for text
font-size	Specifies the font size of text
font-style	Specifies the font style for text
font-variant	Specifies if a text should be displayed in a small-caps font
font-weight	Specifies the weight of a font

- Example:
  - p.normal {font-weight:normal;}
  - p{font-family:"Times New Roman", Times;}
  - p.italic {font-style:italic;}
  - h1 {font-size:40px;}
  - p.small { font-variant:small-caps; }

Styling lists

- The CSS list properties allow you to:
  - Set different list item markers for ordered lists
  - Set different list item markers for unordered lists
  - Set an image as the list item marker
- CSS List Properties

```
Example:
  ul.circle {list-style-type:circle}
  ol.upper-roman {list-style-type:upper-roman}
  ul { list-style-image: url('sqpurple.gif'); }
```

list-style-image	Specifies an image as the list-item marker
list-style-position	Should the list-item markers appear inside or outside the content flow
list-style-type	Specifies the type of list-item marker

```
<html>
<head>
<style>
ul { list-style-image:url('sqpurple.gif');   }
</style>
</head>
<body>
  <ul>
    <li>Coffee</li>
    <li>Tea</li>
    <li>Coca Cola</li>
    </ul>
</body>
```

- Coffee
- Tea
- Coca Cola

Image and table

```
img {
    float: right;
    margin: 0 0 10px 10px;
}
```

```
table{
    border: 1px solid black;
    border-collapse: collapse;
}
```

CSS Box Model

- The CSS box model is essentially a box that wraps around HTML elements; it consists of:
  - Margin - Clears an area around the border. The margin does not have a background color, it is completely transparent
  - Border - A border that goes around the padding and content. The border is affected by the background color of the box
  - Padding - Clears an area around the content; padding is affected by the background color of the box
  - Content - The content of the box, where text and images appear
- Padding defines the inner distance of elements to the end of the box.
- Example
  - padding-top:25px;
    padding-bottom:25px;
    padding-right:50px;
    padding-left:50px;

This is a paragraph with no specified padding.

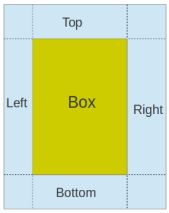
This is a paragraph with specified paddings.

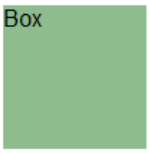
```
<head><style>
p {  background-color:yellow; }
p.padding {
    padding:25px 25px 50px 50px;
}
</style></head>
<body>
<p>This is a paragraph with no specified padding.</p>
<p class="padding">This is a paragraph with specified paddings.</p>
</body>
```

Margins

- Margins allow you to put a bigger distance between your elements
  - the margin is an outer, invisible border around your element.

```
<style type="text/css">
.box {
        background-color: DarkSeaGreen;
        width: 100px;
        height: 100px;
        margin-top: 10px;
        margin-right: 5px;
        margin-bottom: 10px;
        margin-left: 5px;
}
</style>
<div class="box"> Box   </div>
```

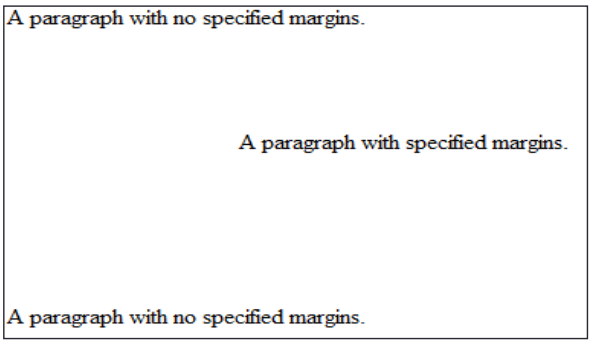




You can define the margins for a box individually or combine them into one statement.

```
.box {  ….; margin: 10px 10px 10px 10px;   }
```

```
<head>
<style>
   p.ex1 {margin:2cm 4cm 3cm 4cm}
</style>
</head>
<body>
<p>A paragraph with no specified margins.</p>
<p class="ex1">A paragraph with specified margins.</p>
<p>A paragraph with no specified margins.</p>
</body>
```

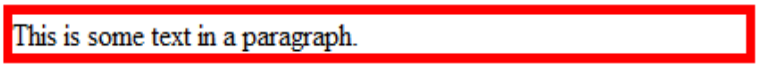


CSS Border

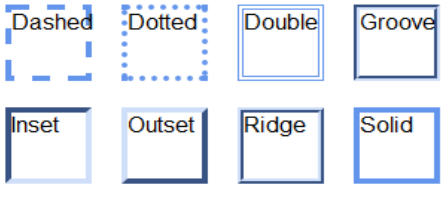
- The border property is a shorthand for the following individual border properties: border-width, border-style (required), border-color

```
<style type="text/css">
.box {
        width: 100px;
        height: 100px;
        border-color: Blue;
        border-width: 2px;
        border-style: solid;
}
</style>
<div class="box">
        Hello, world!
</div>
```


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




```
<div class="box" style="border-style: dashed;">Dashed</div>
<div class="box" style="border-style: dotted;">Dotted</div>
<div class="box" style="border-style: double;">Double</div>
<div class="box" style="border-style: groove;">Groove</div>
<div class="box" style="border-style: inset;">Inset</div>
<div class="box" style="border-style: outset;">Outset</div>
<div class="box" style="border-style: ridge;">Ridge</div>
<div class="box" style="border-style: solid;">Solid</div>
```



CSS3 border

- CSS 3 defines “border radius”, giving developers the possibility to make rounded corners on their elements.

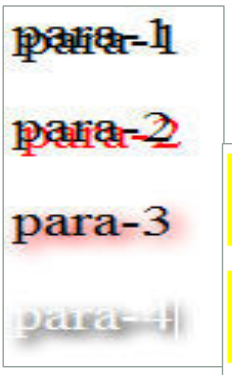
```
<style>
#rcorners1 {
    border-radius: 25px;
    background: #8AC007;
    padding: 20px;
    width: 100px;
    height: 100px;
}
#rcorners2 {
    border-radius: 25px;
    border: 2px solid #8AC007;
    padding: 20px;
    width: 100px;
    height: 100px;
}
```

```
#rcorners3 {
    border-radius: 25px;
    background: url(paper.gif);
    background-position: left top;
    background-repeat: repeat;
    padding: 20px;
    width: 100px;
    height: 100px;
}
</style>
<p id="rcorners1">Rounded corners!</p>
<p id="rcorners2">Rounded corners!</p>
<p id="rcorners3">Rounded corners!</p>
```

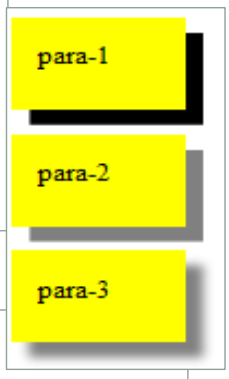
border-radius: 100px 100px 0 0;

Shadow Effects

- You can add shadow to text and to elements.
  - text-shadow property applies shadow to text.
  - box-shadow property applies shadow to elements.




```
<style>
.c1{text-shadow: 2px 2px;}
.c2{text-shadow: 2px 2px red;}
.c3{text-shadow: 2px 2px 5px red;}
.c4{color: white; text-shadow: 2px 2px 4px #000000;}
</style>
<p class="c1">para-1</p>
<p class="c2">para-2</p>
<p class="c3">para-3</p>
<p class="c4">para-4</p>
```




```
<style>
.c1{box-shadow: 10px 10px;}
.c2{box-shadow: 10px 10px grey;}
.c3{ box-shadow: 10px 10px 5px grey;}
p { width: 300px; height: 100px;  padding: 15px; background-color:
yellow; }
</style>
<p class="c1">para-1</p>
<p class="c2">para-2</p>
<p class="c3">para-3</p>
```

Javascript

Overview

- JavaScript is Netscape's cross-platform, object-based scripting language
  - JavaScript code is embedded into HTML pages
  - It is a lightweight programming language
  - Client-side JavaScript extends the core language by supplying objects to control a browser and its Document Object Model
- Why use Javascript?
  - Provides HTML designers a programming tool :
  - Puts dynamic text into an HTML page
  - Reacts to events
  - Reads and writes to HTML elements :
  - Can be used to perform Client side validation
  - The <SCRIPT> tag

```
<SCRIPT>
   JavaScript statements …
</SCRIPT>
```

```
<html>
<head> </head>
<body>
<script type="text/javascript">
      document.write("<H1>Hello World!</H1>");
      alert(“some message”);
      console.log(“some message”);
</script>
</body></html>
```

Embedding JavaScript in HTML

- Where to Write JavaScript?
  - Head Section
  - Body Section
  - External File

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

```
<html>
<head>
<script type="text/javascript">
  function message()  {
        alert("Hello World")
   }
</script>
</head>
<body onload="message()">
</body>
</html>
```

```
<head>
<script src="common.js">
      <!– no javascript statements →
   </ script>
</head>
<body>
   <script>
       document.write("display value of a variable"+msg)
   </script>
</body>
```

```
//common.js file contents
var msg
msg="<h1>in external file</h1>"
```

Data Types in JavaScript

- JavaScript is a free-form language. Need not declare all variables, classes, and methods
- Variables in JavaScript can be of type:
  - Number (4.156, 39)
  - String ("This is JavaScript")
  - Boolean (true or false)
  - Null (null) → usually used to indicate the absence of a value
- Defining variables. var variableName = value
- JavaScript variables are said to be un-typed or loosely typed
  - letters of the alphabet, digits 0-9 and the underscore (_) character and is case-sensitive.
  - Cannot include spaces or any other punctuation characters.
  - First character of name must be either a letter or the underscore character.
  - No official limit on the length of a variable name, but must fit within a line.

Javascript operators:

- Arithmetic Operators (+ , -, * , / , %)
- Assignment Operators(=,+=,-=,*=,/=,%=)
- Comparison Operators (==,!=,<,<=,>,>=)
- Boolean Operators(&&,||,!)
- Bitwise Operators(&,|,!,^,<<,>>,>>>)
- String Operators(=,+,+=)

```
txt1 = "What a very"
txt2 = "nice day!"
txt3 = txt1 + txt2
```

Output → What a verynice day!

```
txt1 = "What a very"
txt2 = "nice day!"
txt3 = txt1 + " " + txt2
```

Output → What a very nice day!

Control Structures and Loops

- JavaScript supports the usual control structures:
  - the conditionals:
    - if,
    - if...else
    - If … else if … else
    - Switch

```
if(condition) {
    statement 1
} else {
    statement 2
}
```

```
if(a>10) {
document.write("Greater than 10")
} else {
document.write("Less than 10")
}
```

```
document.write( (a>10) ? "Greater than 10" : "Less than 10" );
```

  - iterations:
    - for
    - while

```
switch (variable) {
    case outcome1 :{
        //stmts for outcome 1
        break; }
    case outcome2 :{
        //stmts outcome 2
        break; }
    default: {
        //none of the outcomes is chosen
    }
}
```

```
for( [initial expression;][condition;][increment expression] ) {
        statements
}
```

```
for(var i=0;i<10;i++)
{
    document.write("Hello");
}
```

```
while(condition) {
    statements
}
```

```
while(i<10) {
    document.write("Hello");
    i++;
}
```

JavaScript Functions

```
function myFunction (arg1, arg2, arg3)
{
        statements
          return
}
```

```
Calling the function :
 myFunction( “abc”, “xyz”, 4 )
 myFunction()
```

```
function area(w1, w2, h) {
   var area=(w1+w2)*h/2;
   alert(area+" sq ft");
}
area(2,3,7);  //calling the function
```

```
function diameter(radius){
    return radius * 2;
}

var d=diameter(5); //calling the function
```

- **Function expressions -** functions are assigned to variables

```
var area = function (radius) {
      return Math.PI * radius * radius;
  };
alert(area(5));         // => 78.5
```

Global and Local Variables

```
<script language=“Javascript”>
  var companyName=“TechnoFlo”
  function f(){
    var employeeName=“Henry”
    document.write(“Welcome to ”+companyName+ ”, “ +employeeName)
  }
</script>
```

- Variables that exist only inside a function are called Local variables
- Variables that exist throughout the script are called Global variables
  - Their values can be changed anytime in the code and even by other functions

Predefined Functions

- isFinite: evaluates an argument to determine if it is a finite number.

```
isFinite (number)      //where number is the number to evaluate
```

- isNaN : Evaluates an argument to determine if it is “NaN” (not a number)
  - isNaN (testValue), where testValue is the value you want to evaluate

```
isNaN(0)          //false
isNaN('123')      //false
isNaN('Hello')    //true
```

- Parseint and parsefloat
  - Returns a numeric value for string argument.
  - parseInt (str)
  - parseFloat (str)

```
parseInt("3 blind mice")          // => 3
parseFloat(" 3.14 meters")        // => 3.14
parseInt("-12.34")                // => -12
parseInt("0xFF")                  // => 255
parseInt("0xff")                  // => 255
parseInt("-0XFF")                 // => -255
parseFloat(".1")                  // => 0.1
parseInt("0.1")                   // => 0
parseInt(".1")                    // => NaN: integers can't start with "."
parseFloat("$72.47");    // => NaN: numbers can't start with "$"
```

parseInt() interprets any number beginning with “0x” or “0X” as hexadecimal no

Predefined Core Objects

String Objects

- Creating a string object:
  - var myString = new String(“characters”)
  - var myString = “fred”

- Properties of a string object:
  - length: returns the number of characters in a string.

```
• “Lincoln”.length // result = 7
• “Four score”.length // result = 10
• “One\ntwo”.length // result = 7
• “”.length // result = 0
```

- String Object methods:
  - charAt(index) : returns the character at a specified position.
    - Eg : var str = "Hello world!";
      - str.charAt(0); //returns H
      - str.charAt(str.length-1)); //returns !
  - concat() : joins two or more strings
    - stringObject.concat(stringX,stringX,...,stringX)
    - Eg: var str1="Hello ";
      var str2="world!";
      document.write(str1.concat(str2));

String functions

- indexOf () : returns the position of the first occurrence of a specified string value in a string.
  - index values start their count with 0.
  - If no match occurs within the main string, the returned value is -1.
  - string.indexOf(searchString [, startIndex])

```
Eg : var str="Hello world, welcome";
str.indexOf("Hello");  //returns 0
str.indexOf("wor"));  //returns 6
str.indexOf("e",5); //returns 14
```

- toLowerCase() / toUpperCase()

```
Eg: var str="Hello World!";
str.toLowerCase()  //returns hello world
str.toUpperCase()  //returns HELLO WORLD
```

- slice(startIndex [, endIndex])
  - Extracts a part of a string and returns the extracted part in a new string

```
Eg : var str=“Hello World";
     str.slice(6)  //returns World
     str.slice(0,1)  //returns H
```

String functions

- split(“delimiterCharacter”[, limitInteger]) - Splits a string into array of strings
  - string.split(“delimiterCharacter”[, limitInteger])

```
var str = "zero one two three four";
var arr = str.split(" ");
for(i = 0; i < str.length; i++){  document.write(“<br>” + arr[i]); }
```

Output :
zero
one
two
three
four

```
var myString = “Anderson,Smith,Johnson,Washington”
var myArray = myString.split(“,”)
var itemCount = myArray.length // result: 4
```

- Complete Example:

```
var s = "hello, world"          // Start with some text.
s.charAt(0)                     // => "h": the first character.
s.charAt(s.length-1)            // => "d": the last character.
s.substring(1,4)                // => "ell": the 2nd, 3rd and 4th characters.
s.slice(1,4)                    // => "ell": same thing
s.slice(-3)                     // => "rld": last 3 characters
s.indexOf("l")                  // => 2: position of first letter l.
s.lastIndexOf("l")              // => 10: position of last letter l.
s.indexOf("l", 3)               // => 3: position of first "l" at or after 3
s.split(", ")                   // => ["hello", "world"] split into substrings
s.replace("h", "H")             // => "Hello, world": replaces all instances
s.toUpperCase()                 // => "HELLO, WORLD"
```


Date

- Date object allows the handling of date and time information.
  - All dates are in milliseconds from January 1, 1970, 00:00:00.
  - Dates before 1970 are invalid dates.
- There are different ways to define a new instance of the date object:

```
var d = new Date()       //Current date
var d = new Date(milliseconds)
var d = new Date(dateString)
var d = new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

```
<script>
   var d=new Date();
   document.write(d);
</script>
```

```
var d = new Date(86400000);
var d = new Date(99,5,24,11,33,30,0);
```

Date Object - Methods

• getDate( )	Date of the month (1 - 31)
• getDay( )	Day of the week (0 - 6, 0-Sunday)
• getMonth( )	The month (0 - 11, 0 - Jan.)
• getFullYear( )	The year (4 digits)
• getHours( )	Hour of the day (0 - 23)
• getMinutes( )	Minutes (0 - 59)
• getSeconds( )	Seconds (0 - 59)
• getTime( )	Milliseconds since 1/1/1970
• getTimezoneOffset( )	Offset between local time and GMT
• setDate(dayValue)	1-31
• setHours(hoursValue)	0-23
• setMinutes(minutesValue)	0-59
• setMonth(monthValue)	0-11
• setSeconds(secondsValue)	0-59
• setTime(timeValue)	>=0
• setYear(yearValue)	>=1970

Array

- An array is data structure for storing and manipulating ordered collections of data.
- An array can be created in several ways.
  - Eg1: Regular: ----------→
  - Eg 2: Condensed:
    - var cars=new Array("Spark","Volvo","BMW");
  - Eg 3: Literal:
    - var cars=["Spark","Volvo","BMW"];
  - Eg 4: var matrix = [[1,2,3], [4,5,6], [7,8,9]];
  - Eg 5 : var sparseArray = [1,,,,5];

```
var cars=new Array();
cars[0]="Spark";
cars[1]="Volvo";
cars[2]="BMW";
```

- Deleting an array element eliminates the index from the list of accessible index values
  - delete is a unary operator that attempts to delete the **object property** or **array element** specified
  - This does not reduce array's length

```
myArray.length// result: 5
delete myArray[2]
myArray.length// result: 5
myArray[2] // result: undefined
```

Array Object Methods

- arrayObject.reverse()
- arrayObject.slice(startIndex, [endIndex])
- arrayObject.join(separatorString) : array contents will be joined and placed into arrayText by using the comma separator“
- arrayObject.push(): add one or more values to the end of an array

```
arrayObject.slice(startIndex [, endIndex])          //Returns: Array
var solarSys = new Array (“Mercury”,”Venus”,”Earth”,”Mars”,”Jupiter”,”Saturn”)
var nearby = solarSys.slice(1,4)
// result: new array of “Venus”, “Earth”, “Mars”
```

```
arrayObject.concat(array2)
var a1 = new Array(1,2,3)
var a2 = new Array(“a”,”b”,”c”)
var a3 = a1.concat(a2)
// result: array with values 1,2,3,”a”,”b”,”c”
```

```
var arrayText = myArray.join(“,”)
```

```
a = []; // Start with an empty array
a.push("zero") // Add a value at the end. a = ["zero"]
a.push("one", "two") // Add two more values. a = ["zero", "one", "two"]
```

Creating New Objects

1. **Using Object Initializers**
   - Syntax : objName = {property1:value1, property2:value2, … }
   - person = { "name ":"amit", "age":23};
   - myHonda = {color:“red”, wheels:4, engine:{cylinders:4, size:2}}
2. **Using Constructors**
   - Define the object type by writing a constructor function.
   - Create an instance of the object with new.

```
function car(make, model, year) {
   this.make = make
   this.model = model
   this.year = year
}
…..
mycar = new car( “Ford” , “Mustang” , 2013)
```

```
function person(name, age) {
    this.name = name
    this.age = age
}
ken = new person( “Ken” , 33 )
```

```
function car(make, year, owner) {
   this.make = make
   this.year = year
   this.owner = owner
}
car1 = new car( “Mazda”, 1990, ken )
```

Creating New Objects (Contd.)

- Accessing properties

```
car2.owner.name
```

```
car1.make = “corvette”
```

- Defining methods
  - Example:

```
obj.methodName = function_name
obj.methodName(params)
```

```
function car(make, model, year, owner) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.displayCar = displayCar;
}
function displayCar() {
        document.writeln( “A beautiful” + this.year
         + “ ” +  this.make + “ ” + this.model
 }
….
car1.displayCar();   car2.displayCar()
```

Examples : Using Object Initializers

```
// Example 1
var myFirstObject = {};
myFirstObject.firstName = "Andrew";
myFirstObject.lastName = "Grant";
console.log(myFirstObject.firstName);
```

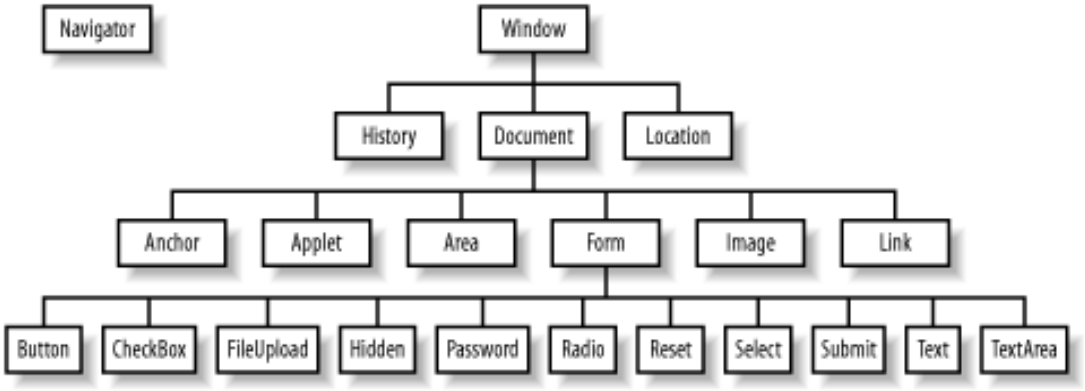
```
// Example 2
var mySecondObject = {
    firstName: "Andrew",
    lastName: "Grant"
};
console.log(mySecondObject.firstName);
```

```
// Example 3
var myThirdObject = new Object();
myThirdObject.firstName = "Andrew";
myThirdObject.lastName = "Grant";
console.log(myThirdObject.firstName);
```

```
var myFirstObject = {};
myFirstObject.firstName = "Andrew";
console.log(myFirstObject.firstName);
myFirstObject.firstName = "Monica";
console.log(myFirstObject.firstName);
myFirstObject["firstName"] = "Catie";
console.log(myFirstObject["firstName"]);
```

```
//Adding Methods to Objects
var person= {
  name: "Andrew",
  age: 21,
  info: function () {
    console.log(“Name" + this.name );
    console.log(“Age" + this.age );
   }
};
person.info();
for (var prop in person) {
     console.log(person[prop]);
}
```

JavaScript Document Object Model



Window Object Methods

- alert(message)
  - window.alert(“Display Message”)
- confirm(message)
  - window.confirm(“Exit Application ?”)
- prompt(message,[defaultReply])
  - var input=window.prompt(“Enter value of X”)

- window.open(*URL,name,specs*)
  - URL : Specifies the URL of the page to open. If no URL is specified, a new window with about:blank is opened
  - Name : Specifies the target attribute or the name of the window.
  - Specs : comma-separated list of items.

```
myWindow=window.open('','','width=200,height=100');
myWindow.document.write("<p>This is 'myWindow'</p>");
myWindow.focus();
```

example opens an about:blank page in a new browser window:

setInterval and setTimeout methods

```
<body>
<input type="text" id="clock" size="35" />
<script language=javascript>
var int=self.setInterval("clock()",50)
function clock() {
   var ctime=new Date()
   document.getElementById("clock").value=ctime
 }
</script>
<button onclick="int=window.clearInterval(int)">Stop interval</button>
</body>
```

```
<head> <script type="text/javascript">
function timedMsg()  {
    var t=setTimeout("alert('5 seconds!')",5000)
 }
</script> </head>
<body> <p>Click on the button. An alert box will be  displayed after 5 seconds.</p>
<form>
<input type="button" value="Display timed alertbox!“ onClick="timedMsg()">
</form>
</body>
```

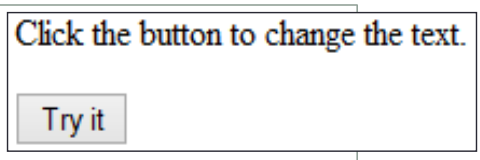
Document Object

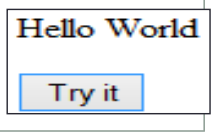
- When an HTML document is loaded into a web browser, it becomes a document object; root node of the HTML document and owns all other nodes

document.anchors	Returns a collection of all the anchors in the document
document.baseURI	Returns the absolute base URI of a document
document.cookie	Returns all name/value pairs of cookies in the document
document.forms	Returns a collection of all the forms in the document
document.getElementById()	Returns the element that has the ID attribute with the specified value
document.getElementsByName()	Accesses all elements with a specified name
document.getElementsByTagName()	Returns a NodeList containing all elements with the specified tagname
document.images	Returns a collection of all the images in the document
document.lastModified	Returns the date and time the document was last modified
document.links	Returns a collection of all the links in the document
document.referrer	Returns the URL of document that loaded current document
document.title	Sets or returns the title of the document
document.URL	Returns the full URL of the document
document.write()	Writes HTML expressions or JavaScript code to a document
document.writeln()	Same as write(), but adds a newline character after each statement

Examples : Modifying content

```
<body>
<p id="p1">Click the button to change the text.</p>
<button onclick="myFunction()">Try it</button>
<script>
function myFunction() {
   document.getElementById("p1").innerHTML="Hello World";
};
</script></body>
```

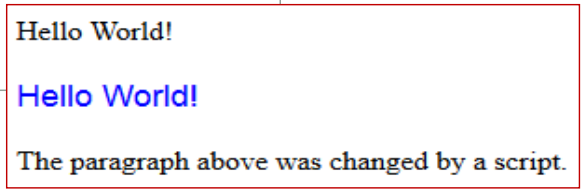




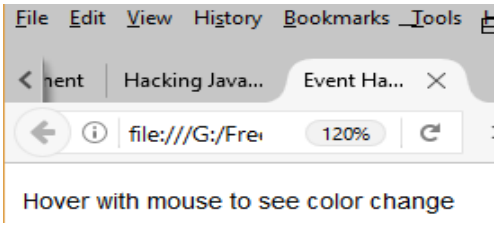

```
<body>
The title of the document is:
<script>
document.write(document.title);
document.title="another title"  //change the title
</script>
</body>
```

Example : Modifying styles

```
<html>
<body>

<p id="p1">Hello World!</p>
<p id="p2">Hello World!</p>

<script>
document.getElementById("p2").style.color = "blue";
document.getElementById("p2").style.fontFamily = "Arial";
document.getElementById("p2").style.fontSize = "larger";
</script>

<p>The paragraph above was changed by a script.</p>

</body>
</html>
```



Mouse events

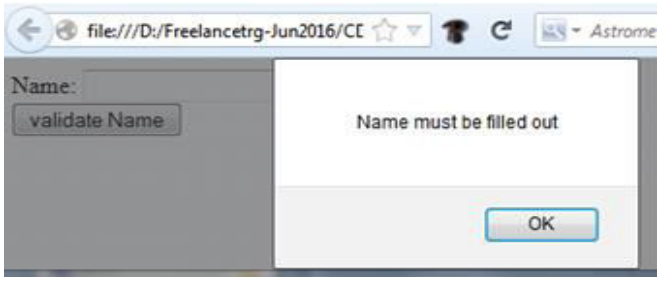


Hover with mouse to see color change

```
<SCRIPT>
function changeColor(para){
   para.style.color="blue";
   para.style.backgroundColor = "lightgray";
   para.style.font = "italic bold 30px arial,serif";
}
function revertColor(para){
    para.style.color="black";
    para.style.backgroundColor = "white";
    para.style.font = "12px arial,serif";
}
</SCRIPT>
<BODY>
   <p id="p1" onmouseover="changeColor(this)"
          onmouseout="revertColor(this)">Hover with mouse to see color change</p>
</BODY>
```

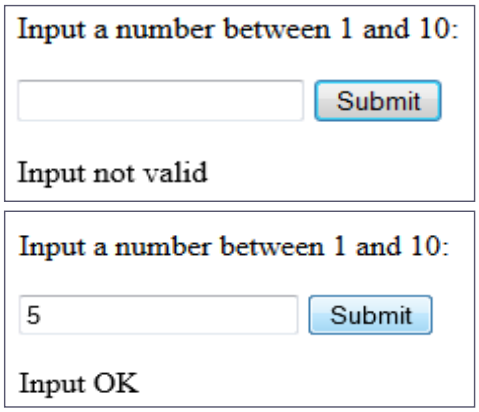
Form validation

```
<html>
<head>
<script >
function validate(){
    var x=document.getElementById("fname").value;
    if (x == null || x == "") {
         alert("Name must be filled out");
         return false;
     }
 }
</script>
</head>
<body>
<form id="form1"  onsubmit="return validate()">
Name: <input type="text" name="fname" id="fname" /><br />
<input type="submit" value="validate Name"  />
</form>
</body></html>
```



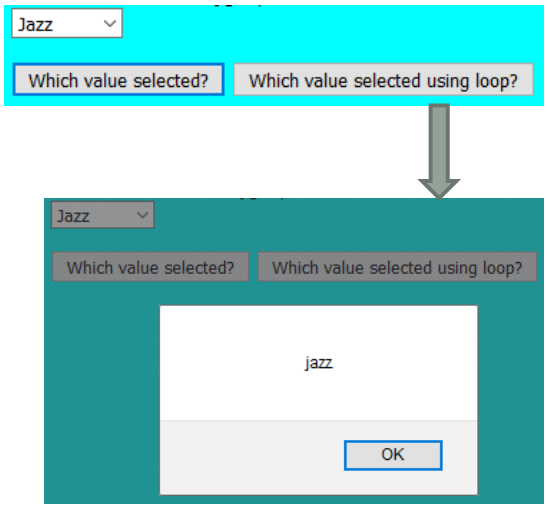
Example

```
<body>
< Input a number between 1 and 10:</p>
<input id="numb">
<button type="button" onclick="myFunction()">Submit</button>
<p id="demo"></p>
<script>
function myFunction() {
    var x, text;
   x = document.getElementById("numb").value;
   if (isNaN(x) || x < 1 || x > 10) {
        text = "Input not valid";
    } else {
        text = "Input OK";
    }
    document.getElementById("demo").innerHTML = text;
}
</script>
</body>
```

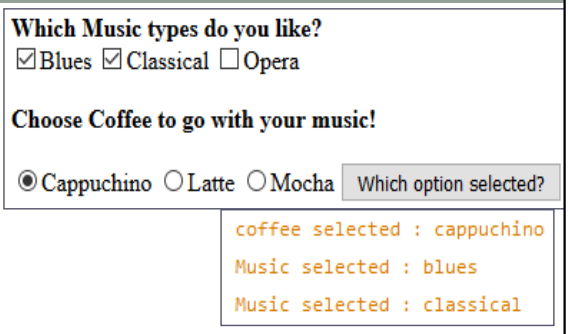


Example

```
<SCRIPT>
function valSelected1(){
  var sel = document.getElementById("musicTypes");
  alert(sel.value);       // prints value, not text
  var opt = sel.options[sel.selectedIndex];
  alert(opt.text);      //option.text prints text
}
function valSelected3(){
  var sel = document.getElementById("musicTypes");    var opt;
    for ( var i = 0, len =; i < sel.options.length; i++ ) {
        opt = sel.options[i];
        if ( opt.selected == true ) {   break;   }
    }
    alert(opt.value);
}
</SCRIPT>
<FORM NAME="selectForm">
  <SELECT name="musicTypes" id="musicTypes">
    <OPTION VALUE="rnb" SELECTED> R&B </OPTION>
    <OPTION VALUE="jazz"> Jazz </OPTION>
    <OPTION VALUE="blues"> Blues </OPTION>
</SELECT>
<INPUT TYPE="button" VALUE="value selected?"onClick="valSelected1()">
<INPUT TYPE="button" VALUE="value selected using loop?"onClick="valSelected3()">
</FORM>
</BODY>
```



Example




```
<SCRIPT>
function valSelected(){
  var radio = document.getElementsByClassName("r1");
   for(var i = 0; i < radio.length; i++){
        if(radio[i].checked)  console.log("coffee selected : " + radio[i].value);
   }
  var checklist = document.getElementsByClassName("c1");
  for(i=0;i<checklist.length;i++){
     if (checklist[i].checked == true)  console.log("Music selected : " + checklist[i].value);
}
</SCRIPT>
<FORM NAME="selectForm">
<B>Which Music types do you like?</B>
<input type="checkbox" class="c1" id="c1" value="blues">Blues</input>
<input type="checkbox" class="c1" id="c2" value="classical">Classical</input>
<input type="checkbox" class="c1" id="c3" value="opera">Opera</input>
<b>Choose Coffee to go with your music!</b><br>
<INPUT TYPE="radio" name="coffee" class="r1" id="coffee" VALUE="cappuchino">Cappuchino </input>
<INPUT TYPE="radio" name="coffee" class="r1" id="coffee" VALUE="latte">Latte</input>
<INPUT TYPE="radio" name="coffee" class="r1" id="coffee" VALUE="Mocha">Mocha</input>
<INPUT TYPE="button" VALUE="Which option selected?" onClick="valSelected()">
</FORM>
```

Regular Expressions

- A regular expression is an object that describes a pattern of characters.
  - Its matched against a text string, when you perform searches & replacements
  - Perform client-side data validations or any other extensive text entry parsing
  - RegExp objects may be created either with the RegExp() constructor or using a special literal syntax.
  - regular expression literals are specified as characters within a pair of slash (/)

var re = / / → simple pattern to match the space character

var re = / /g → matching a string on a global basis

var re = /web/i → a case-insensitive match

var re = /web/gi → expression is both case-insensitive and global

- Eg

```
str = "I love JavaScript!";
regexp = /love/;
alert( str.search(regexp) ); // 2
```

i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

RegEx – Special Characters

- \b Word Boundary:
  - Get a match at the beginning or end of a word in the string
    - /\bor/ matches “origami” and “or” but not “normal”.
    - /or\b/ matches “traitor” and “or” but not “perform”
    - /\bor\b/ matches full word “or” and nothing else

- \B Word Non-Boundary:
  - Get a match when it is not at the beginning or end of a word in the string
    - /\Bor/ matches “normal” but not “origami”
    - /or\B/ matches “normal” and “origami” but not “traitor”
    - /\Bor\B/ matches “normal” but not “origami” or “traitor”

RegEx – Special Characters (Contd.)

- \d Numeral: Find any single digit 0 through 9
  - /\d\d\d/ matches “212” and “415” but not “B17”

- \D Non-numeral: Find any non-digit
  - /\D\D\D/ matches “ABC” but not “212” or “B17”

- \s Single White Space: Find any single space character
  - /over\sbite/ matches “over bite” but not “overbite” or “over bite”

- \S Single Non-White Space:
  - /over\Sbite/ matches “over-bite” but not “overbite” or “over bite”

- \w Letter, Numeral, or Underscore:
  - /A\w/ matches “A1” and “AA” but not “A+”

- \W Not letter, Numeral, or Underscore:
  - /A\W/ matches “A+” but not “A1” and “AA”

RegEx – Special Characters (Contd.)

- "." Any Character Except Newline:
  - /…/ matches "ABC", "1+3", "A 3" or any 3 characters
- […] Character Set: any character in the specified character set
  - /[AN]BC/ matches "ABC" and "NBC"
- [^…] Negated Character Set: any character not in the specified character set
  - /[^AN]BC/ matches "BBC" and "CBC" but not "ABC" or "NBC"
- **Positional Metacharacters**
- "^" - At the beginning of a string or line
  - /^Fred/ matches "Fred is OK" but not "I'm with Fred" or "Is Fred here?"
- "$" - At the end of a string or line
  - /Fred$/ matches "I'm with Fred" but not "Fred is OK" or "Is Fred here?"

RegEx – Counting Metacharacters

- "*" - Zero or More Times:
  - /Ja*vaScript/ matches "JvaScript", "JavaScript" & "JaaavaScript" but not "JovaScript"
- "?" - Zero or One Time:
  - /Ja?vaScript/ matches "JvaScript" or "JavaScript" but not "JaaavaScript"
- "+" - One or More Times:
  - /Ja+vaScript/ matches "JavaScript" or "JaavaScript" but not "JvaScript"
- {n} - Exactly n Times:
  - /Ja{2}vaScript/ matches "JaavaScript" but not "JvaScript" or "JavaScript"
- {n,} - N or More Times:
  - /Ja{2,}vaScript/ matches "JaavaScript" or "JaaavaScript" but not "JavaScript"
- {n,m} - At Least n, At Most m Times:
  - /Ja{2,3}vaScript/ matches "JaavaScript" or "JaaavaScript" but not "JavaScript"

Regular Expression Object

- Create Regular Expression:

  regExpObject = /pattern/ [g | i | gi]

  regExpObject = new RegExp(pattern, flag)

  - Eg: re = new RegExp("pushing", "g");
  - Eg: var zipcode = new RegExp("\\d{6}", "g");

- Methods that use regular expressions

Method	Description
exec()	executes a search for a match in a string. It returns an array of information or null on a mismatch.
test()	tests for a match in a string. It returns true or false
match()	executes a search for a match in a string. It returns an array of information or null on a mismatch.
search()	tests for a match in a string; returns the index of the match, or -1 if search fails
replace()	executes a search for a match in a string, and replaces the matched substring with a replacement substring.
split()	uses a reg exp or a fixed string to break a string into an array of substrings.

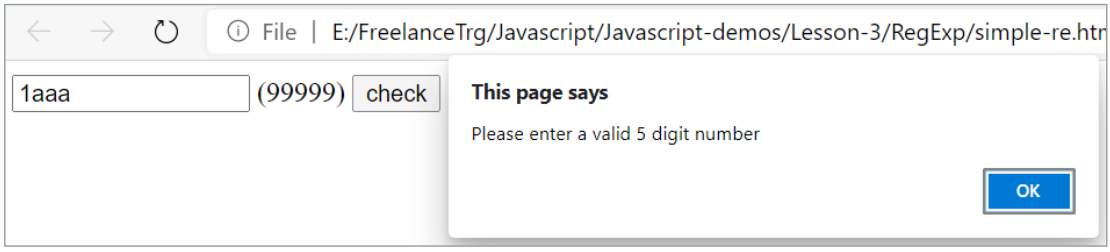
```
var pattern = /java/g;
var text = "JavaScript is more fun than java!";
var result = pattern.exec(text)
console.log(result)   //["java", index: 28, input: "JavaScript is more fun than java!"]
```

When there's a "g" flag, then str.match returns an array of all matches. There are no additional properties in that array, and parentheses do not create any elements.
With no "g" flag, looks for the first match only.

```
let str = "HO-Ho-ho!";
let result = str.match( /ho/ig );  //global search
alert( result ); // HO, Ho, ho (all matches, case-insensitive)
```

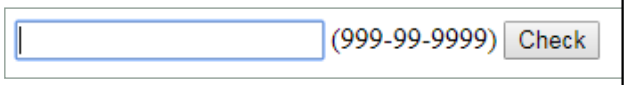
```
s1 = "how are you all doing";
var re = new RegExp("o","g");
console.log(s1.replace(re,"z"));  //hzw are yzu all dzing
```

```
<script >
function checkpostal(){
var re5digit=/^\d{5}$/
//regular expression defining a 5 digit number
if (document.myform.myinput.value.search(re5digit)==-1)
  alert("Please enter a valid 5 digit number")
}
</script>
<form name="myform">
   <input type="text" name="myinput" size=15> (99999)
   <input type="button" onClick="checkpostal()" value="check">
</form>
```



Demo

```
<head>
<title>ssn Checker</title>
<script type="text/javascript">
var RE_SSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;

function checkSsn(ssn){
 if (RE_SSN.test(ssn)) {
  alert("VALID SSN");
 } else {
  alert("INVALID SSN");
 }
}
</script>
</head>
<body>
 <form onsubmit="return false;">
  <input type="text" name="ssn" size="20"> (999-99-9999)
  <input type="button" value="Check"
   onclick="checkSsn(this.form.ssn.value);">
 </form>
</body>
```



AJAX & JSON

JSON (JavaScript Object Notation)

- JSON is a simple and easy to read and write data exchange format.
  - It is easy for humans to read and write and easy for machines to parse and generate.
  - It is based on a subset of the JavaScript, Standard ECMA-262
  - JSON is a text format that is completely language independent; can be used with most of the modern programming languages.
  - The filename extension is .json
  - JSON Internet Media type is application/json
  - It's popular and implemented in countless projects worldwide, for those don't like XML, JSON is a very good alternative solution.

```
var obj = {
    "name":"Amit",
    "age": 37.5,
    "married": true,
    "address":{ "city":"Pune" , "state":"Mah" },
    "hobbies":["swimming","reading","music"]
}
```

- Values supported by JSON
  - Strings : double-quoted Unicode, with backslash escaping
  - Numbers:
    - double-precision floating-point format in JavaScript
  - Booleans : true or false
  - Objects: an unordered, comma-separated collection of key:value pairs enclosed in curly braces, with the ':' character separating the key and the value; the keys must be strings and should be distinct from each other
  - Arrays : an ordered, comma-separated sequence of values enclosed in square brackets; the values do not need to be of the same type
  - Null : A value that isn't anything

Demo

```
<script language="javascript">
var JSONObject = {  "name" : "Amit",
       "address"  : "B-123 Bangalow",
       "age"  : 23,
       "phone"   : "011-4565763",
       "MobileNo"  : 0981100092
 };
var str =
"<h2><font color='blue'>Name </font>::“        +JSONObject.name+"</h2>“ +
 "<h2><font color='blue'>Address </font>::“   + JSONObject.address+"</h2>“ +
"<h2><font color='blue'>Age </font>::“            +JSONObject.age+"</h2>“ +
"<h2><font color='blue'>Phone No </font>::“  +JSONObject.phone+"</h2>“ +
"<h2><font color='blue'>Mobile No </font>::“  + JSONObject.MobileNo+"</h2>“;

document.write(str);
</script>
```

Name ::Amit

Address ::B-123 Bungalow

Age ::23

Phone No ::011-4565763

Mobile No ::981100092

Demo

```
<script >
var students = {
    "Students": [
            { "Name": "Amit Goenka",
              "Major": "Physics"
            },
            { "Name": "Smita Pallod",
             "Major": "Chemistry"
            },
            { "Name": "Rajeev Sen",
              "Major": "Mathematics"
            }] }
var i=0
document.writeln("students.Students.length : " + students.Students.length);
for(i=1;i<students.Students.length+1;i++)  {
    document.writeln("<b>Name : </b>" + students.Students[i].Name + "    ");
    document.writeln("<b>Majoring in : </b>" + students.Students[i].Major);
    document.writeln("</br>");
}
</script>
```

students.Students.length : 3
Name : Amit Goenka **Majoring in :** Physics
Name : Smita Pallod **Majoring in :** Chemistry
Name : Rajeev Sen **Majoring in :** Mathematics

What is Ajax ?

- “Asynchronous JavaScript And XML”
  - AJAX is not a programming language, but a technique for making the user interfaces of web applications more responsive and interactive
  - It provide a simple and standard means for a web page to communicate with the server without a complete page refresh.

Why Ajax?

- Intuitive and natural user interaction
  - No clicking required. Call can be triggered on any event
  - Mouse movement is a sufficient event trigger
- "Partial screen update" replaces the "click, wait, and refresh" user interaction model
  - Only user interface elements that contain new information are updated (fast response)
- The rest of the user interface remains displayed as it is without interruption (no loss of operational context)

XMLHttpRequest

- JavaScript object - XMLHttpRequest object for asynchronously exchanging the XML data between the client and the server

- XMLHttpRequest Methods
  - open(“method”, “URL”, syn/asyn) : Assigns destination URL, method, mode
  - send(content) : Sends request including string or DOM object data
  - abort() : Terminates current request
  - getAllResponseHeaders() : Returns headers (labels + values) as a string
  - getResponseHeader(“header”) : Returns value of a given header
  - setRequestHeader(“label”,”value”) : Sets Request Headers before sending

- XMLHttpRequest Properties
  - Onreadystatechange : Event handler that fires at each state change
  - readyState values – current status of request
  - Status : HTTP Status returned from server: 200 = OK
  - responseText : get the response data as a string
  - responseXML : get the response data as XML data

Creating an AJAX application

- Step 1: Get an instance of XHR object

```
if (window.XMLHttpRequest) { // Mozilla, Safari, IE7+ ...
    xhr = new XMLHttpRequest();
} else if (window.ActiveXObject) { // IE 6 and older
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
```

- Step 2: Make the request

```
xhr.open('GET', 'http://www.example.org/some.file', true);
xhr.send(null);
```

```
xhr.open("POST", "AddNos.jsp");
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhr.send("tno1=100&tno2=200");
```

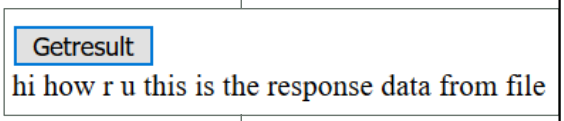
- Step 3 : Attach callback function to xhr object

```
httpRequest.onreadystatechange = function(){
      // process the server response
};
```

Ajax Demo

```
<script >
var xhr;
function getData(){
  getHTTPRequestObject();
  if(xhr){
    xhr.open("GET", "Sample.txt", true);
    xhr.send();
    xhr.onreadystatechange = function(){
      if(xhr.readyState == 4 &&  xhr.status == 200){
          document.getElementById("lblresult").innerHTML=xhr.responseText;
       }
    }  //end of callback function
}}
function getHTTPRequestObject() {
      if (window.XMLHttpRequest) { // Mozilla, Safari, IE7+ ...
          xhr = new XMLHttpRequest();
      } else if (window.ActiveXObject) { // IE 6 and older
          xhr = new ActiveXObject("Microsoft.XMLHTTP");
 }}
</script> <body>
   <input type="button" onclick="getData()"  value="Getresult"/>
   <div id="lblresult"></div>
</body>
```

```
//sample.txt
hi how r u this is the
response data from file
```



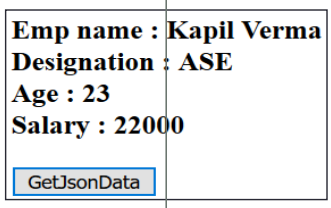
AJAX Demo with XML

```
<script language="javascript" type="text/javascript">
var xmlhttp;
function getData(){
  getHTTPRequestObject();
  if(xmlhttp){
     xmlhttp.open("GET", "Employee.xml", true);
    xmlhttp.send();
    xmlhttp.onreadystatechange = function(){
        if(xmlhttp.readyState == 4 && ixmlhttp.status == 200){
            xmlDoc=xmlhttp.responseXML;
            x=xmlDoc.getElementsByTagName("EmpName");
            document.write("<h3>--- Emp names ----</h3><br>");
            for (i=0;i<x.length;i++){
                document.write(x[i].childNodes[0].nodeValue  + "<br>");
             }
   } }
</script>
<body>
<input type="button" onclick=" getData()" value="Getresult"/>
<div id="lblresult"></div>
</body>
</html>
```

```
<Employees>
  <Employee>
    <empid>1001</empid>
    <EmpName>Vipul</EmpName>
    <Desig>Software Analyst</Desig>
  </Employee>
 <Employee>
  <Empid>1002</Empid>
    <EmpName>Vivek</EmpName>
    <Desig>Software Analyst</Desig>
  </Employee>
</Employees>
```

AJAX Demo with JSON

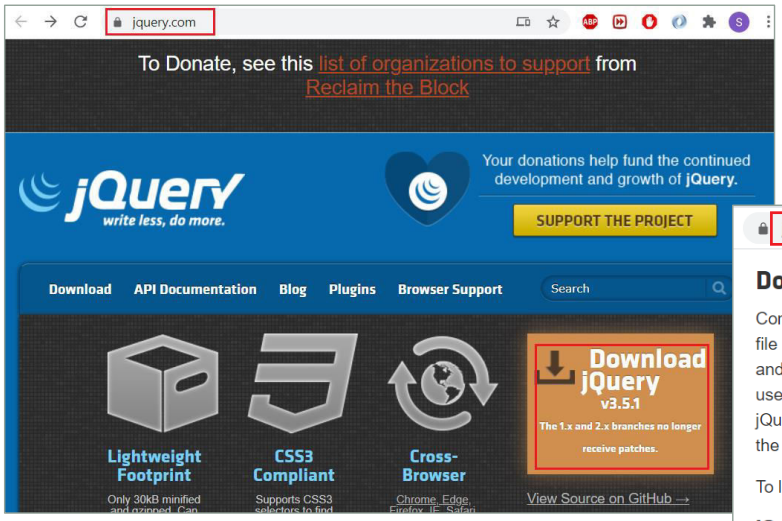
```
<script >
 var xmlhttp;
 function getData(){
   getHTTPRequestObject();
   if(xmlhttp){
      xmlhttp.open("GET", "EmpJSONData.txt", true);
     xmlhttp.onreadystatechange = function(){
       if(xmlhttp.readyState == 4 && xmlhttp.status == 200){
            var obj = JSON.parse(xmlhttp.responseText);
            var displaytext = "";
            displaytext += "Emp name : " + obj.name + "<br>" +
                        "Designation : " + obj.desig + "<br>" +
                        "Age : " + obj.age + "<br>" +
                        "Salary : " + obj.sal;
           document.getElementById("lblres").innerHTML = displaytext;
        }
} }
</script><body>
<h3 id="lblres">Result</h3>
<input type="button" id="btngetjsondata" onclick="getData()"  value="GetData">
</body>
```

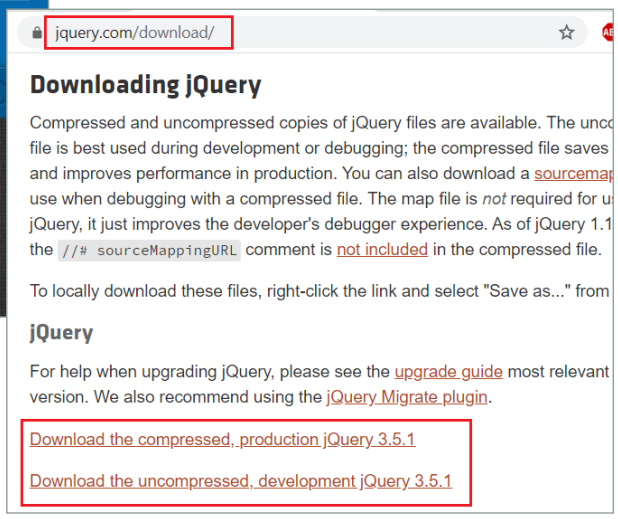



```
{ "name":"Kapil Verma",
  "desig":"ASE",
  "age":23,
  "sal":22000
}
```

JQUERY

Software





- **Choosing a Text Editor**
  - Text editors that support jQuery include Brackets, Sublime Text, Kwrite, Gedit, Notepad++, PSPad, or TextMate.

jQuery Introduction

- jQuery is a lightweight, cross browser and feature-rich JavaScript library which is used to manipulate DOM
  - Originally created by John Resig in early 2006.
  - The jQuery project is currently run and maintained by a distributed group of developers as an open-source project.

- Why jQuery
  - JavaScript is great for a lot of things especially manipulating the DOM but it's pretty complex stuff. DOM manipulation is by no means straightforward at the base level, and that's where jQuery comes in. It abstracts away a lot of the complexity involved in dealing with the DOM, and makes creating effects super easy.
  - It can locate elements with a specific class
  - It can apply styles to multiple elements
  - It solves the cross browser issues
  - It supports method chaining
  - It makes the client side development very easy

Including jQuery in HTML Document

- jQuery library can be included in a document by linking to a local copy or to one of the versions available from public servers.
- Eg : include a local copy of the jQuery library

```
<html>
<head>
<title>Test jQuery</title>
<script src="../scripts/jquery-3.5.1.min.js"></script>
</head>
<body>
   <!-- bidy of HTML -->
</body>
</html>
```

- Eg : include the library from a publicly available repository
  - There are several well-known public repositories for jQuery; these repositories are also known as Content Delivery Networks (CDNs).

```
<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.1.1.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
```

Using jQuery

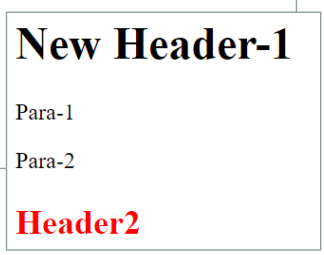
```
<html>
<head>
<title>Test jQuery</title>
<script type="text/javascript" src="jquery-3.5.1.js"></script>
<script type="text/javascript">
$(document).ready(function()  {
      alert('Hi');
 });
</script>
</head>
<body>
     Welcome to jQuery
</body>
</html>
```

```
$(function() {
   // jQuery code
});
```

Introduction to selectors

- jQuery uses same CSS selectors used to style html page to manipulate elements
  - CSS selectors select elements to add style to those elements where as jQuery selectors select elements to add behavior to those elements.
  - Selectors allow page elements (Single or Multiple) to be selected.
- Selector Syntax
  - $(selectorExpression)
  - jQuery(selectorExpression)
  - $(selector).action()

```
<html>
<head>
<title>Test jQuery</title>
<script src="../scripts/jquery-3.5.1.min.js"></script>
<script>
$(document).ready(function(){
      $("h2").css("color","red");
      jQuery("h1").html("New Header-1");
   });
</script>
</head>
<body>
    <h1>jQuery Enabled</h1>
   <p>Para-1</p>
   <p>Para-2</p>
   <h2>Header2</h2>
</body>
</html>
```



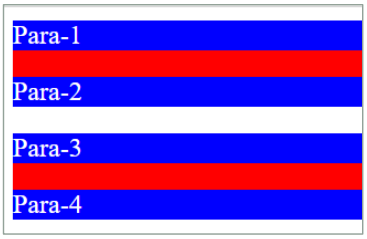
Selectors

- **Selecting by Tag Name:**
- Selecting single tag takes the following syntax
  - $('p') – selects all <p> elements
  - $('a') – selects all <a> elements
- To reference multiple tags, use the (,) to separate the elements
  - $('p, a, span') - selects all paragraphs, anchors and span elements

- **Selecting Descendants**
- $('ancestor descendant') - selects all the descendants of the ansector
  - $('table tr') - Selects all tr elements that are the descendants of the table element
- Descendants can be children, grand children etc of the designated ancestor element.

Demo

```
<style type="text/css">
.redDiv{background-color:red; color:white;}
</style>
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function() {
    var paragraphs = $('p');
    alert(paragraphs.length);   //4
    paragraphs.css('background-color','blue');
    paragraphs.each(function(){
       alert($(this).html());
    });
    var collections = $('div,p');
    alert(collections.length);    //6
    var bodydivs = $('body div');
    alert(bodydivs.length);    //2
});
</script>
</head>
<body>
<div class="redDiv" >
<p>Para-1</p>
<p>Para-2</p>
</div>
<div  class="redDiv">
<p>Para-3</p>
<p>Para-4</p>
</div>
</body>
```



Selecting by Element ID

- It is used to locate the DOM element very fast.
- Use the # character to select elements by ID
  - $("#first") — selects the element with id="first"
  - $('#myID') – selects the element with id=" myID "

```
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function()
{
    alert($('#testDiv').html());
    $('#testDiv').html("Changed Text in Div!!");
    $("#p1").hide();
});
</script>
</head>
<body>
<div id="testDiv">Test Div</div>
<p id="p1"> Para-1</p>
<p>Para-2</p>
</body>
```

Test Div

Para-1

Para-2

Changed Text in Div!!

Para-2

Selecting Elements by Class Name

- Use the (.) character to select elements by class name
  - $(".intro") — selects all elements with class="intro"
- To reference multiple tags, use the (,) character to separate class name.
  - $('.blueDiv, .redDiv') - selects all elements containing class blueDiv and redDiv
- Tag names can be combined with elements name as well.
  - $('div.myclass') – selects only those tags with class="myclass"

```
<script>
$(document).ready(function(){
    $(".bold").css("font-weight", "bold");
});
</script>
</head>
<body>
<ul>
   <li class="bold">Test 1</li>
   <li>Test 2</li>
   <li class="bold">Test 3</li>
</ul>
</body>
```

- **Test 1**
- Test 2
- **Test 3**

```
<style type="text/css">
.blueDiv{background-color:lightblue; color:darkblue;}
.redDiv{background-color:orange; color:red;}
</style>
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function(){
    var collection = $('.blueDiv');
    collection.css('border','5px solid blue');
    collection.css('padding','10px');
});
</script>
</head>
<body>
<div class="blueDiv">
<p>para-1</p>
</div>
<div class="redDiv">
<p>para-2</p>
</div>
<div class="blueDiv">
<p>para-3</p>
</div>
</body>
```

para-1

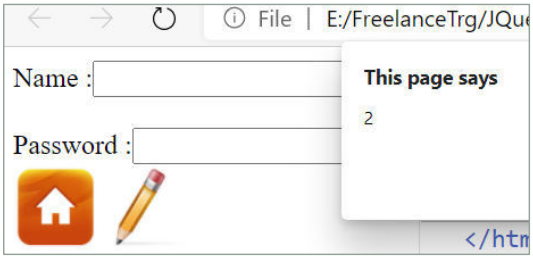
para-2

para-3

Selecting by attribute values

- Use brackets [attribute] to select on attribute name and/or attribute value
  - $('a[title]') - selects all anchor elements that have a title attribute
  - $('a[title="trainer"]') – selects all <a> elements that have a "trainer" title attribute value

```
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function()
{
    var list = $('div[title="first"],img[height]' );
    alert(list.length);   //2
});
</script>
</head>
<body>
<div title="first">
Name :<input type="text"/>
</div><br>
<div title="second" >
Password :<input type="password"/>
</div>
<img src="a.png" height="50" width="50">
<img src="b.png"  width="50">
</body>
```



Selecting by input elements

- To select input elements of type : <input>:
  - $('input[type="text"]').css("background", "yellow");

- To select all input elements
  - $(':input') - Selects all form elements (input, select, textarea, button).
  - $(':input[type="radio"]') – selects all radio buttons

  - $(":text") - All input elements with type="text"
  - $(":password") - All input elements with type="password"
  - $(":radio") - All input elements with type="radio"
  - $(":checkbox") - All input elements with type="checkbox"
  - $(":submit") - All input elements with type="submit"
  - $(":reset") - All input elements with type="reset"
  - $(":button") - All input elements with type="button"
  - $(":file") - All input elements with type="file"

```
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function()
{
    var inputs = $(':input');
    alert($(inputs[2]).val());  //Chennai
    $( ":text" ).css({ background: "yellow", border: "3px red solid" });
});
</script>
</head>
<body>
Name : <input id="txtName" type="text" value="Karthik"><br>
Age : <input id="txtAge" type="text" /><br>
City :
<select id="city">
<option value="Bangalore">Bangalore</option>
<option value="Chennai" selected="selected">Chennai</option>
<option value="Mumbai">Mumbai</option>
</select><br>
</body>
```

Basic Filters

- The index-related selectors (:eq(), :lt(), :gt(), :even, :odd) filter the set of elements that have matched the expressions that precede them.
  - They narrow the set down based on the order of the elements within this matched set.
  - Eg, if elements are first selected with a class selector (.myclass) and four elements are returned, these elements are given indices 0 through 3
- eq() - Select the element at index n within the matched set.
  - Eg : $("p:eq(1)") - Select the second <p> element
  - Eg : $("element:eq(0)") is same as $('element:first-child')
- $('element:odd') and $('element:even') selects odd and even positions respectively. 0 based indexing
  - Odd returns (1,3,5…) and Even returns (0,2,4…)
- :gt() and lt() - Select all elements at an index > or < index within the matched set
  - Eg : $("tr:gt(3)") : Select all <tr> elements after the 4 first
  - Eg : $("tr:lt(4)") : Select the 4 first <tr> elements

```
<table border="1">
  <tr><td>TD #0</td><td>TD #1</td><td>TD #2</td></tr>
  <tr><td>TD #3</td><td>TD #4</td><td>TD #5</td></tr>
  <tr><td>TD #6</td><td>TD #7</td><td>TD #8</td></tr>
</table>
<script>
  $( "td:eq( 2 )" ).css( "color", "red" );
  //$( "tr:first" ).css( "font-style", "italic" );  //is same as below line
  $( "tr:eq(0)" ).css( "font-style", "italic" );
  $( "td:gt(4)" ).css( "backgroundColor", "yellow" );
</script>
```

TD #0	TD #1	TD #2
TD #3	TD #4	TD #5
TD #6	TD #7	TD #8

```
<script type="text/javascript">
$(document).ready(function() {
    $('tr:odd').css('background-color','tomato');
    $('tr:even').css('background-color','bisque');
});
</script>
<table border=1 cellspacing=5 cellpadding=5>
  <th>column 1<th>column 2<th>column 3
  <tr><td>data 1</td><td>data 2</td><td>data 3
  <tr><td>data 4<td>data 5<td>data 6
  <tr><td>data 7<td>data 8<td>data 9
  <tr><td>data 10<td>data 11<td>data 12
</table>
```

column 1	column 2	column 3
data 1	data 2	data 3
data 4	data 5	data 6
data 7	data 8	data 9
data 10	data 11	data 12

Basic Filters continued

- :first , :last - Selects the first/last matched element.
  - :first is equivalent to :eq(0) and :lt(1). This matches only a single element, whereas, :first-child can match more than one; one for each parent.
- :header - Selects all elements that are headers, like h1, h2, etc

```
<script>
$(document).ready(function() {
    $( ":header" ).css({ background: "#ccc", color: "blue" });
    $('tr:first-child').css('background-color','tomato');
});
</script>
</head>
<body>
<h1>Table 1</h1>
<table>
  <tr><td>Row 1</td></tr>
  <tr><td>Row 2</td></tr>
  <tr><td>Row 3</td></tr>
</table>
<br/><br/>
<h2>Table 2</h2>
<table border=1>
<th>column 1<th>column 2<th>column 3
<tr><td>data 1</td><td>data 2</td><td>data 3
```

Table 1

Row 1
Row 2
Row 3

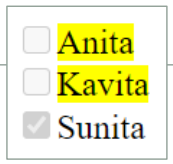
Table 2

column 1	column 2	column 3
data 1	data 2	data 3
data 4	data 5	data 6
data 7	data 8	data 9

Basic Filters continued

- :not - Selects all elements that do not match the given selector.
  - Eg : $("p:not(.intro") - Select all <p> elements except those with class="intro"
  - Eg : $('a:not(div.important a, a.nav)'); - // Selects anchors that do not reside within 'div.important' or have the class 'nav'

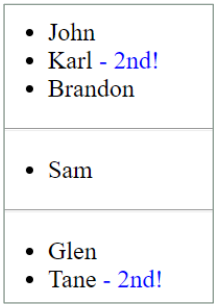
```
<body>
 <div><input type="checkbox"><span>Anita</span></div>
 <div><input type="checkbox"><span>Kavita</span></div>
 <div><input type="checkbox" checked="checked"><span>Sunita</span></div>
 <script>
   $( "input:not(:checked) + span" ).css( "background-color", "yellow" );
   $( "input").attr( "disabled", "disabled" );
</script>
 </body>
```



Child Filter

- $('element:first-child') and $('element:last-child') selects the first child & last child of its parent.
  - $('span:first-child') returns the span which is a first child for all the groups
- :nth-child() - Selects all elements that are the nth-child of their parent. 1-based indexing
  - Eg : $("p:nth-child(3)") - Select each <p> element that is the third child of its parent

```
  <style>
  span {color: blue; }
  </style>
</head>
<body>
  <ul>
    <li>John</li>
    <li>Karl</li>
    <li>Brandon</li>
  </ul> <hr>
  <ul><li>Sam</li></ul> <hr>
  <ul>
    <li>Glen</li>
    <li>Tane</li>
  </ul>
 <script>
$( "ul li:nth-child(2)" ).append( "<span> - 2nd!</span>" );
</script>
```



```
 <style>
   span {color: #008;}
   span.sogreen { color: green; font-weight: bolder;}
   span.solast {text-decoration: line-through;}
</style>
<script src="..\Scripts\jquery-3.5.1.min.js"></script>
</head>
<body>
<div> <span>John,</span><span>Karl,</span><span>Brandon</span></div>
<div> <span>Glen,</span> <span>Tane,</span><span>Ralph</span> </div>
<script>
$( "div span:first-child" ) .css( "text-decoration", "underline" )
      .hover(function() {
            $( this ).addClass( "sogreen" );
          }, function() {
              $( this ).removeClass( "sogreen" );
        });
$( "div span:last-child" ).css({ color:"red", fontSize:"80%" })
.hover(function() {
      $( this ).addClass( "solast" );
   }, function() {
      $( this ).removeClass( "solast" );
    });
$("div span:nth-child(2)").css("background-color","yellow");
</script>
```

John,Karl,Brandon
Glen, Tane,Ralph

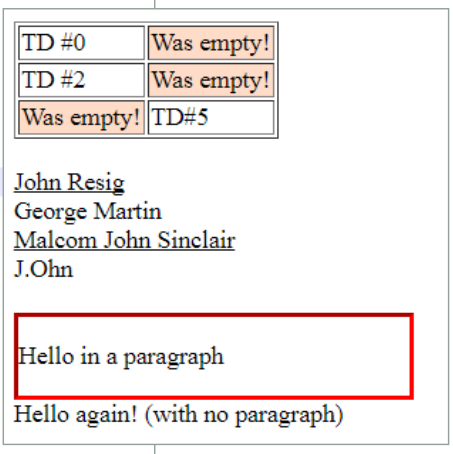
John,Karl,Brandon
Glen, Tane,Ralph

John,Karl,Brandon
Glen, Tane,~~Ralph~~

Content Filters

- :contains() will select elements that match the contents.
  - $('div:contains("hello")') - selects div's which contains the text hello (match is case sensitive)
- :empty - Select all elements that have no children (incl text nodes)
- :has : Selects elements which contain at least one element that matches the specified selector.
  - Eg : $("p:has(span)") - Select all <p> elements that have a element inside of them
  - Eg : $("div:has(p,span,li)").css("border","solid red"); - Select all elements that have at least one of the given elements inside
- :parent - Select all elements that have at least one child node (either an element or text).
  - Eg : $("td:parent") - Select all <td> elements with children, including text

```
<style>
.test { border: 3px inset red; width:250px }
</style>
</head>
<body>
<table border="1">
  <tr><td>TD #0</td><td></td></tr>
  <tr><td>TD #2</td><td></td></tr>
  <tr><td></td><td>TD#5</td></tr>
</table>
<br>
<div>John Resig</div>
<div>George Martin</div>
<div>Malcom John Sinclair</div>
<div>J.Ohn</div>
<br>
<div><p>Hello in a paragraph</p></div>
<div>Hello again! (with no paragraph)</div>
<script>
  $( "td:empty" ).text( "Was empty!" )
      .css( "background", "rgb(255,220,200)" );
  $( "div:contains('John')" ).css( "text-decoration", "underline" );
  $( "div:has(p)" ).addClass( "test" );
</script>
```



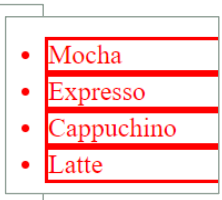
jQuery Traversing -> filtering

- .eq() - Reduce the set of matched elements to the one at the specified index.
  - Eg : $("p").eq(1).css("background-color","yellow") - Select the second <p> element (index number 1)
- .filter() - Reduce the set of matched elements to those that match the selector or pass the function's test
  - Eg : $("p").filter(".intro") - Return all <p> elements with class "intro“
- .first() / last()
  - Eg : $("div p").first() - Select first <p> element inside first element
- .has() - Reduce the set of matched elements to those that have a descendant that matches the selector or DOM element.
  - Eg : $("p").has("span") - Return all <p> elements that have element inside

jQuery Traversing -> Tree Traversal

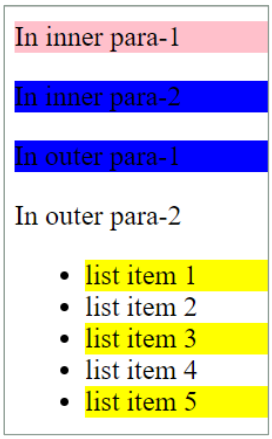
- .children() - Returns all direct children of the selected element
  - Eg : $("ul").children().css({"color":"red","border":"2px solid red"}) - Return elements that are direct children of

```
<ul class="ulclass">
  <li>Mocha</li>
  <li>Expresso</li>
  <li>Cappuchino</li>
  <li>Latte</li>
</ul>
<script>
$(".ulclass").children().css({"color":"red","border":"2px solid red"})
</script>
```



- .find() - Returns descendant elements of the selected element
  - Eg : $("ul").find("span").css({"color":"red","border":"2px solid red"}) - Return all elements that are descendants of
- .next() / prev() - Returns the next / previous sibling element of the selected element
- nextAll() - returns all next sibling elements of the selected element
- parent() - Returns the direct parent element of the selected element

```
<style>
 .blue {background: blue;}
 .highlight{background-color: pink}
</style>
<script src="..\Scripts\jquery-3.5.1.min.js"></script>
</head>
<body>
  <div>
    <p>In inner para-1</p>
    <p class="intro">In inner para-2</p>
  </div>
  <p id="outro">In outer para-1</p>
  <p>In outer para-2</p>
  <ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
    <li>list item 4</li>
    <li>list item 5</li>
  </ul>
<script>
 $( "li" ).eq(4).css( "background-color", "red" );
 $( "body" ).find( "li" ).eq( 2 ).addClass( "blue" );
 $("li").filter(":even").css("background-color","yellow");
 $("p").filter(".intro,#outro").css("background-color","blue");
 $( "div p" ).first().addClass( "highlight" );
</script>
```



Method Chaining

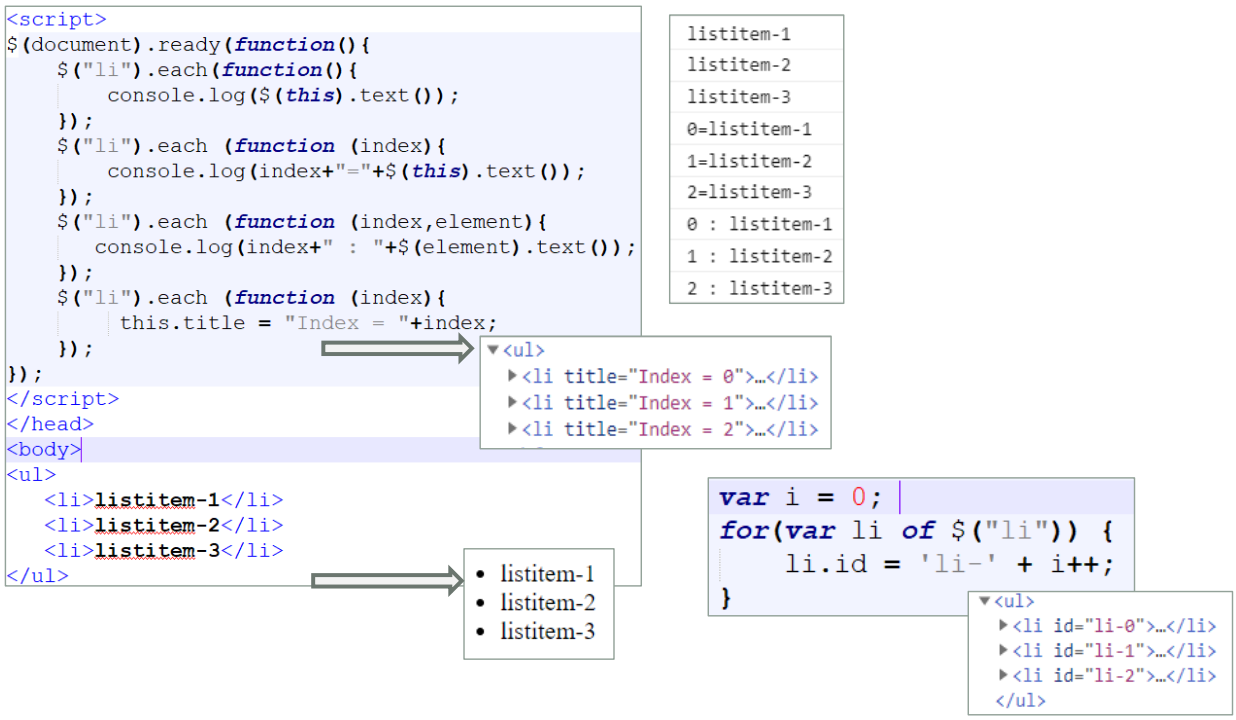
- Chaining is a good way to avoid selecting elements more than once. Eg:
  - $("div").fadeOut();
  - $("div").css("color", "red");
  - $("div").text("hello world");

- Instead of doing that and running $("div") three times, you could do this:

```
$("div").fadeOut().css("color", "red").text("hello world");
```

Iterating through Nodes

- .each(function(index,Element)) is used to iterate through jQuery objects.
  - jQuery 3 also allows to iterate over the DOM elements of a jQuery collection using the for...of loop.



```
<script>
$(document).ready(function(){
    var result = "";
    $('div.One,div.Two,div.Three').each(function(index) {
         result+=index+" "+$(this).text()+"<br/>";
    });
    $('#OutputDiv').html(result);
    //try with element
    $("li").each(function(){
       console.log($(this).text())
    });
});
</script>
</head>
<body>
<ul>
   <li>foo</li>
   <li>bar</li>
</ul>
<div class="One"></div>
<div class="Two">Second Div</div>
<div class="Three">Third Div</div>
------------- output from jQuery ---------------
<div id="OutputDiv" />
</body>
```

- foo
- bar

Second Div
Third Div
---------- output
0
1 Second Div
2 Third Div

Console
top
foo
bar

Working with Attributes

- Object attributes can be used using attr():
  - var val = $('#customDiv').attr('title'); - Retrieves the title attribute value
- .attr(attributeName,value) : accesses an object's attributes and modifies the values.
  - $('img').attr('title','Image title'); - changes the title attribute value to Image title.
- To modify multiple attributes, pass JSON object.

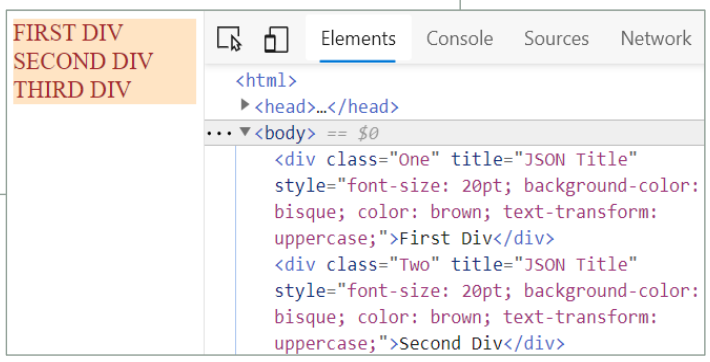
```
$("img").attr({
          "title":  "image title",
          "style" : "border:5px dotted red"

    });
```

```
<img src="paintbrush.gif" title="image title" style="border:5px dotted red">
```

- You can also remove attributes entirely using .remo

```
<script>
$(document).ready(function(){
        var result = "";
        $('div').each(function(index){
            //raw DOM object
            this.title="Custom title";

            //using jQuery Object
            //$(this).attr('title','Modified title');
        });
        //using JSON to Modify multiple attributes
        $('div').attr({
                    title:'JSON Title',
                    style:'font-size:20pt;background-color:bisque;'
                   }).css('color','brown')
                     .css('text-transform','uppercase');

});
</script>
</head>
<body>
<div class="One">First Div</div>
<div class="Two">Second Div</div>
<div class="Three">Third Div</div>
</body>
```

attributes-demo.html



Getting Content

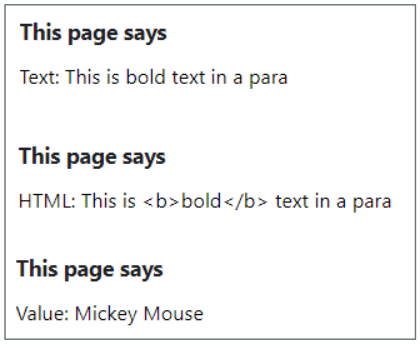
- text() - Sets or returns the text content of selected elements
- html() - Sets or returns the content of selected elements (including HTML markup)
- val() - Sets or returns the value of form fields

```
<script>
$(document).ready(function(){
   $("#div1").html('<a href="example.html">Link</a><b>hello</b>');
   $("#div2").text('<a href="example.html">Link</a><b>hello</b>');
});
</script>
</head>
<body>
<div id="div1"></div>
<div id="div2"></div>
</body>
```

$.html() treats the string as HTML, $.text() treats the content as text

Getting Content

```
<script type="text/javascript">
   $(document).ready(function(){
      $("#btn1").click(function(){
         alert("Text: " + $("#test").text());
      });
      $("#btn2").click(function(){
         alert("HTML: " + $("#test").html());
      });
      $("#btn3").click(function(){
         alert("Value: " + $("#test1").val())
       });
});
</script>
</head>
<body>
<p id="test">This is <b>bold</b> text in a para</p>
<input id="test1" value="Mickey Mouse"><br>
<button id="btn1">Button1</button>
<button id="btn2">Button2</button>
<button id="btn3">Button3</button>
```




```
var input = $( 'input[type="text"]' );
input.val( 'new value' );
input.val();    // returns 'new value'
```

Adding and Removing Nodes

- In traditional approach adding and removing nodes is tedious.
- To insert nodes four methods available:
- Appending adds children at the end of the matching elements
  - .append()
  - .appendTo()
- Prepending adds children at the beginning of the matching elements
  - .prepend()
  - .prependTo()
- To wrap the elements use .wrap()
  - Eg: $(“p”).wrap(“<h1></h1>”) //wraps “p” in <h1> tags
- To remove nodes from an element use .remove()

```
<script>
$(document).ready(function(){
  $("#btn1").click(function(){
    $("p").prepend(" <b>Prepended text</b>.");
    $("h2").wrap("<header></header>");
  });
  $("#btn2").click(function(){
    $("ol").append("<li>Appended item</li>");
  });
  $("#btn3").click(function(){
    $( "h2" ).text("h2 position changed")
             .appendTo( $( "p" ) )
  });
});
</script>
</head>
<body>
 <h2>Greetings</h2>
 <p>This is a paragraph.</p>
 <p>This is another paragraph.</p>
 <ol>
   <li>List item 1</li>
   <li>List item 2</li>
   <li>List item 3</li>
 </ol>
 <button id="btn1">Prepend text</button>
 <button id="btn2">Append list items</button>
 <button id="btn3">AppendTo text</button>
```

Greetings

This is a paragraph.

This is another paragraph.

1. List item 1
2. List item 3
3. List item 3

Prepend text | Append list items | AppendTo text

Greetings

Prepended text.This is a paragraph.

Prepended text.This is another paragraph.

1. List item 1
2. List item 3
3. List item 3

Prepend text | Append list items | AppendTo text

Greetings

Prepended text.This is a paragraph.

Prepended text.This is another paragraph.

1. List item 1
2. List item 3
3. List item 3
4. Appended item
5. Appended item

Prepend text | Append list items | AppendTo text

Prepended text.This is a paragraph.

h2 position changed

Prepended text.This is another paragraph.

h2 position changed

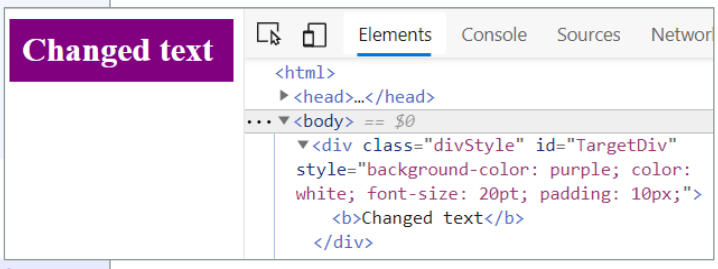
Modifying Styles

- .css() function is used to modify an object's style
  - $('div').css('color','red');
- Multiple styles can be modified by passing a JSON Object

```
$('div').css({
  "color":"red",
  "font-weight":"bold"
});
```

```
<script>
$(document).ready(function(){
    $('#TargetDiv').css({
        'background-color':'purple',
        'color':'white',
        'font-size':'20pt',
        'padding':'10px'
    })
    .html('<b>Changed text</b>');
});
</script>
</head>
<body>
<div id="TargetDiv">Target Div</div>
</body>
```

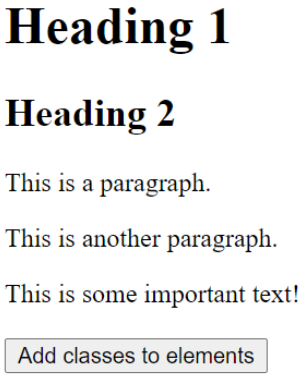
Changed text

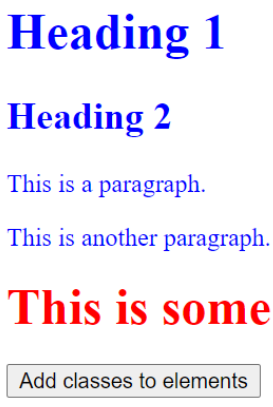


Working with Classes

- The four methods for working with css class attributes are
- .addClass() : adds one or more classes to the class attribute of each element.
  - $('p').addClass('classOne');
  - $('p').addClass('classOne classTwo');
- .hasClass() : returns true if the selected element has a matching class
  - if($('p').hasClass('classOne')) { //perform operation}
- removeClass() remove one or more classes
  - $('p').removeClass('classOne classTwo');
- To remove all class attributes for the matching selector
  - $('p').removeClass();
- .toggleClass() : alternates adding or removing a class based on the current presence or absence of the class.
  - $('#targetDiv').toggleClass('highlight');

```
<style>
.important{
  font-weight:bold;
  font-size:xx-large;
  color:red;
}
.blue {color:blue;}
</style>
<script>
$(document).ready(function() {
    $("button").click(function(){
        $("h1,h2,p").addClass("blue");
        $("div").addClass("important");
  });
});
</script>
</head>
<body>
<h1>Heading 1</h1>
<h2>Heading 2</h2>
<p>This is a paragraph.</p>
<p>This is another paragraph.</p>
<div>This is some important text!</div>
<button>Add classes to elements</button>
```





Working with Classes (Contd)

```
<style type="text/css">
.highlight{ background-color:yellow; };
</style>
<script>
  $(document).ready(function() {
    $('input[type="text"]').addClass('highlight');
    //$('#txtLastName').removeClass('highlight');
  });
  function FocusBlur(element){
    $(element).toggleClass('highlight');
  }
</script>
</head>
<body>
<table>
<tr>
<td>FirstName : </td>
<td><input id="txtFirstName" onFocus="FocusBlur(this)"
          onBlur="FocusBlur(this)" /></td>
</tr>
<tr>
<td>LastName : </td>
<td><input id="txtLastName" onFocus="FocusBlur(this)"
          onBlur="FocusBlur(this)"/></td>
</tr>
</table>
```

FirstName :

LastName :

jQuery Event Model Benefits

- Events notify a program that a user performed some type of action
- jQuery Events
  - click()
  - blur()
  - focus()
  - dblclick()
  - mousedown()
  - mouseup()
  - mouseover()
  - keydown()
  - keypress()
  - hover() : hover() method takes two functions and is a combination of the mouseenter() and mouseleave() methods.

```
$("img").mouseover(function () {
        $(this).css("opacity", "0.3");
});
$("img").mouseout(function () {
        $(this).css("opacity", "1.0");
});
```

```
$("#p1").hover(function(){
  alert("You entered p1!");
},
function(){
  alert("Bye! You now leave p1!");
});
```

Handling Click Events

- .click(handler([eventObject])) is used to listen for a click event or trigger a click event on an element
  - $('#submitButton').click(function() { alert('Clicked') });

```
<script>
$(document).ready(function(){
    $('#SubmitButton').click(function(){
        var fName = $('#txtFirstName').val();
        var lName = $('#txtLastName').val();
        $('#tgtDiv').html("<b>"+fName+" "+lName+"</b>");
    });
});
</script>
</head>
<body>
FirstName : <input id="txtFirstName" type="text" /><br>
LastName : <input id="txtLastName" type="text" /><br>
<input id="SubmitButton" type="submit" value="Submit"/><br>
<div id="tgtDiv" />
</body>
```

FirstName : Anil
LastName : Patil
Submit
Anil Patil

```
<style>
  p {
    color: red;
    margin: 5px;
    cursor: pointer;
  }
  p:hover {background: yellow;}
  </style>
</head>
<body>
<p>First Paragraph</p>
<p>Second Paragraph</p>
<p>Yet one more Paragraph</p>

<script>
 $( "p" ).click(function() {
   $( this ).slideUp();
});
</script>
</body>
```

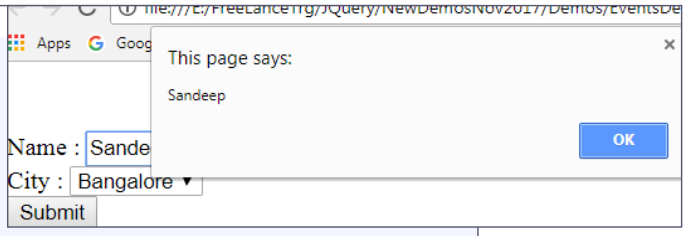
First Paragraph
Second Paragraph
Yet one more Paragraph

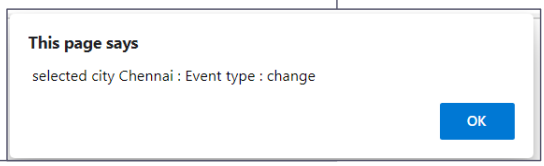
```
<style type="text/css">
.highlight{background-color:yellow;};
</style>
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
        $(document).ready(function(){
            $('#targetDiv').mouseenter(function(){
                toggle(this);
            });
            $('#targetDiv').mouseleave(function(){
                toggle(this);
            });
            $('#targetDiv').mouseup(function(e){
                $(this).text('X : '+e.pageX+' Y :'+e.pageY );
            });
        });
        function toggle(element){
            $(element).toggleClass('highlight');
        }
</script>
</head>
<body>
<div id="targetDiv">Welcome to events</div>
</body>
```

Welcome to events

X : 123 Y :17

```
<script>
$(document).ready(function() {
    $('.testClass').change(function(){
        alert($(this).val());
    });
    $('#City').change(disp);

    function disp(e){
       alert(" selected city " + $(this).val() + " : Event type : " + e.type)
    }
});
</script>
</head>
<body><BR><BR>
Name : <input id="txtName" class="testClass" autofocus/>
City :
<select id="City" >
  <option value="Bangalore">Bangalore</option>
  <option value="Chennai" >Chennai</option>
  <option value="Mumbai">Mumbai</option>
</select><br>
<input type="submit" value="Submit"/></td>
</body>
```



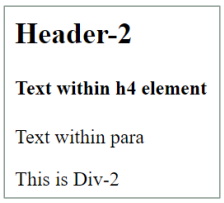


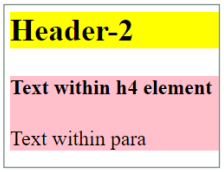
Using on() and off()

- The on() method attaches one or more event handlers for the selected elements.

```
$( "#dataTable tbody tr" ).on( "click", function() {
    console.log( $( this ).text() );
});
```

```
<script>
$(document).ready(function() {
  $("#div1").on("click",function(){
      $(this).css("background-color","pink");
  });
  $("h2").on("mouseover mouseout",function(){
      $(this).toggleClass("intro");
  });
  $("#div2").on({
    mouseover: function(){
      $(this).css("background-color", "lightgray");
    },
    mouseout: function(){
      $(this).css("background-color", "lightblue");
    },
    click: function(){
      $(this).hide();
    }
  });
});
</script>
```






```
<body>
<h2>Header-2</h2>
<div id="div1">
 <h4>Text within h4 element</h4>
 <p>Text within para</p>
</div>
<div id="div2"> This is Div-2 </div>
</body>
```

Showing and Hiding Elements

- To set a duration and a callback function
  - show(duration, callback)
  - duration is the amount of time taken (in milliseconds), and callback is a callback function jQuery will call when the transition is complete.
- The corresponding version of hide( )
  - hide(duration, callback)
- To toggle an element from visible to invisible or the other way around with a specific speed and a callback function, use this form of toggle( )
  - toggle(duration, callback)

```
<style type="text/css">
.blueDiv{
  background-color:blue;
  color:white;font-size:30pt;
  display:none
  }
</style>
<script src="..\Scripts\jquery-3.5.1.js"></script>
<script>
$(document).ready(function(){
   $('#btnShow').click(function(){
      //fast(200),normal(400) and slow(600) can also be s
      $('.blueDiv').show(5000,function(){
         $('#results').text('Show Animation Completed');
      });
    });
    $('#btnHide').click(function(){
      $('.blueDiv').hide(5000,function(){
        $('#results').text('Hide Animation Completed');
       });
    });
    $('#btnSH').click(function(){
      $('.blueDiv').toggle(5000,function(){
        $('#results').text('Animation Completed');
      });
    });
});
</script>
```


```
<body>
<input id="btnShow" type="button" value="Show"/>
<input id="btnHide" type="button" value="Hide"/>
<input id="btnSH" type="button" value="Show/Hide"/>
<div class="blueDiv">
<p>Welcome to animations!!</p>
</div>
<div id="results"></div>
</body>
```

jQuery Sliding Effects

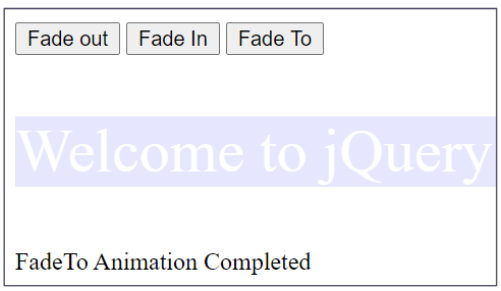
- The jQuery slide methods slide elements up and down.
- jQuery has the following slide methods:
  - $(selector).slideDown(speed,callback) : *Display the matched elements with a sliding motion*
  - $(selector).slideUp(speed,callback) : *Hide the matched elements with a sliding motion.*
  - $(selector).slideToggle(speed,callback)
- The speed parameter can take the following values: "slow", "fast", "normal", or milliseconds.
- The callback parameter is the name of a function to be executed after the function completes.

```
$("#flip").click(function(){
    $("#panel").slideDown();
});
```

jQuery Fading Effects

- The jQuery fade methods gradually change the opacity for selected elements.
- jQuery has the following fade methods:
  - $(selector).fadeIn(speed,callback)
  - $(selector).fadeOut(speed,callback)
  - $(selector).fadeTo(speed,opacity,callback)
- The speed parameter can take the following values: "slow", "fast", "normal", or milliseconds.
  - The opacity parameter in the fadeTo() method allows fading to a given opacity.
  - The callback parameter is the name of a function to be executed after the function completes.

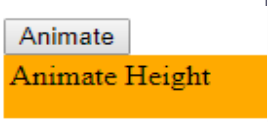
```
$('.blueDiv').fadeTo(1000,0.1,function(){
        $('#results').text('FadeTo Animation Completed');
});
```

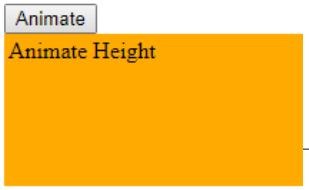


Creating Custom Animation

- Custom animation can be created in jQuery with the animate() function
  - animate(params, duration, callback)
    - params contains the properties of the object you're animating, such as CSS properties, duration is the optional time in milliseconds that the animation should take and callback is an optional callback function.

```
<style type="text/css">
#content { background-color:#ffaa00; width:300px; height:30px; padding:3px; }
</style>
<script type="text/javascript">
$(document).ready(function() {
      $("#animate").click(function() {
          $("#content").animate({"height": "100px", "width": "350px"}, "slow");
       });
});
</script>
</head>
<body>
<input type="button" id="animate" value="Animate"/>
<div id="content">Animate Height</div> </body>
```





jQuery Ajax features

- The jQuery library has a full suite of Ajax capabilities.
  - The functions and methods therein allow us to load data from the server without a browser page refresh.
  - $(selector).load() : Loads HTML data from the server
  - $.get() and $.post() : Get raw data from the server
  - $.getJSON() : Get / Post and return JSON data
  - $.ajax() : Provides core functionalityjQuery Ajax functions works with REST APIs, Webservices and more

Loading HTML content from server

- $(selector).load(url,data,callback) allows HTML content to be loaded from a server and added into DOM object.
  - $("#targetDiv").load('GetContents.html');
- A selector can be added after the URL to filter the content that is returned from the calling load().
  - $("#targetDiv").load('GetContents.html #Main');
- Data can be passed to the server using load(url,data)
  - $('#targetDiv').load('Add.aspx',{firstNumber:5,secondNumber:10})
- load () can be passed a callback function

```
$('#targetDiv').load('Notfound.html', function (res,status,xhr) {
        if (status == "errror") {
            alert(xhr.statusText);
        }
});
```

Using get(), getJSON() & post()

- $.get(url,data,callback,datatype) can retrieve data from a server.
  - datatype can be html, xml, json

```
$.get('GetContents.html',function(data){
                $('#targetDiv').html(data);
},'html');
```

- $.getJSON(url,data,callback) can retrieve data from a server.

```
$.getJSON('GetContents.aspx,{id:5},function(data){
                $('#targetDiv').html(data);
});
```

- $.post(url,data,callback,datatype) can post data to a server and retrieve results.

Using ajax() function

- ajax() function is configured by assigning values to JSON properties

```
$.ajax({
  url: “employee.asmx/GetEmployees",
  data : null,
  contentType: "application/json; charset=utf-8",
  datatype: ‘json’,
  success: function(data,status,xhr){
     //Perform success operation
  },
error: function(xhr,status,error) {
       //show error details
}
});
```

```
$("button").click(function(){
    $.ajax({
        url: "demo_test.txt",
        success: function(result){
            $("#div1").html(result);
        }
    });
});
```

NODE.JS

Server Side Javascript

Node.js – an intro

- In 2009 Ryan Dahl created Node.js or Node, a framework primarily used to create highly scalable servers for web applications.
  - Node.js is an open source, cross-platform runtime environment for server-side JavaScript.
  - Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute code.
  - It is written in C++ and JavaScript.
  - Node.js is a development framework that is based on Google's V8 JavaScript engine that powers Google's Chrome web browser.
  - You write Node.js code in JavaScript, and then V8 compiles it into machine code to be executed.

  - It's a highly scalable system that uses asynchronous, non-blocking I/O model (input/output), rather than threads or separate processes
  - It is not a framework like jQuery nor a programming language like C# or JAVA; Its primarily a Javascript engine

Node.js is really two things: a runtime environment and a library

Traditional Programming vs Event-driven programming

- In traditional programming I/O is performed in the same way as it does local function calls. i.e. Processing cannot continue until the operation is completed.
  - When the operation like executing a query against database is being executed, the whole process/thread idles, waiting for the response. This is termed as "Blocking"
  - Due to this blocking behavior we cannot perform another I/O operation, the call stack becomes frozen waiting for the response.

- Event-driven programming or Asynchronous programming is a programming style where the flow of execution is determined by events.
- Events are handled by event handlers or event callbacks
  - An event callback is a function that is invoked when something significant happens like when the user clicks on a button or when the result of a database query is available.

- This style of programming — whereby instead of using a return value you define functions that are called by the system when interesting events occur — is called event-driven or asynchronous programming.

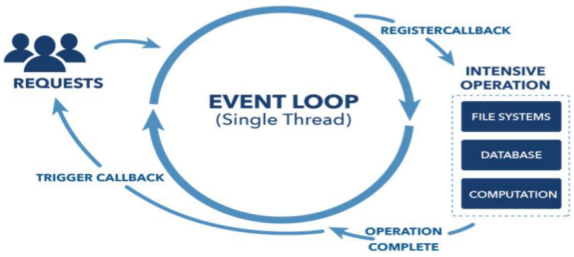
Typical blocking I/O programming

```
result = query('SELECT * FROM posts WHERE id = 1');
do_something_with(result);
```

```
query_finished = function(result) {
    do_something_with(result);
}
query('SELECT * FROM posts WHERE id = 1', query_finished);
```

Event loop

- An event loop is a construct that mainly performs two functions in a continuous loop — **event detection and event handler triggering**.
  - In any run of the loop, it has to detect which events just happened.
  - Then, when an event happens, the event loop must determine the event callback and invoke it.
- This event loop is just one thread running inside one process, which means that, when an event happens, the event handler can run without interruption. This means the following:
  - There is at most one event handler running at any given time.
  - Any event handler will run to completion without being interrupted.
  - Node.js uses the “Single Threaded Event Loop” architecture to handle multiple concurrent clients.



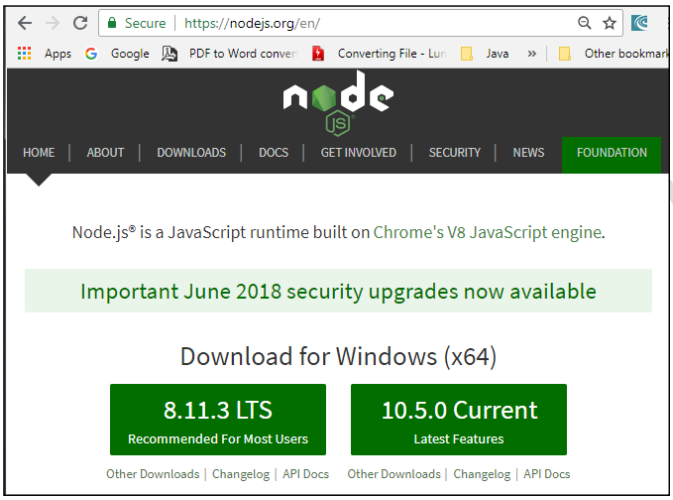
Asynchronous and Event Driven

- All APIs of Node.js library are asynchronous that is, non-blocking.
  - It essentially means a Node.js based server never waits for an API to return data.
  - The server moves to the next API after calling it and a notification mechanism of Node.js helps the server to get a response from the previous API call.
  - It is non-blocking, so it doesn't make the program wait, but instead it registers a callback and lets the program continue.
- Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.
- Node.js is great for data-intensive applications.
  - Using a single thread means that Node.js has an extremely low-memory footprint when used as a web server and can potentially serve a lot more requests.
  - Eg, a data intensive application that serves a dataset from a database to clients via HTTP
- **What Node is NOT!**

Node is **not** a webserver. By itself it doesn't do anything. It doesn't work like Apache. There is no config file where you point it to you HTML files. If you want it to be a HTTP server, you have to write an HTTP server (with the help of its built-in libraries). Node.js is just another way to execute code on your computer.
It is simply a JavaScript runtime.

Setting up Node

- To install and setup an environment for Node.js :
  - Download the latest version of Node.js installable archive file from https://nodejs.org/en/
  - Double click to run the msi file
  - Verify if the installation was successful : node –v in command window.




```
C:\Users\Administrator>node -v
v8.11.3
```

Using the Node CLI : REPL (Read-Eval-Print-Loop)

- There are two primary ways to use Node.js on your machines: by using the Node Shell or by saving JavaScript to files and running those.
  - Node shell is also called the Node REPL; a great way to quickly test things in Node.
  - When you run “node” without any command line arguments, it puts you in REPL

```
node
Your environment has been set up for
 using Node.js 4.4.0 (x64) and npm.

C:\Users\DELL>node
> REPL
```

```
node
C:\Users\DELL>node
> console.log("hello world");
hello world
undefined
>
```

```
node
> 10+20
30
> x=50
50
> x
50
>
```

```
> var foo = [];
undefined
> foo.push(123);
1
> foo
[ 123 ]
>
```

```
> function add(a,b){
... return (a+b);
... }
undefined
> add(10,20)
30
>
```

```
> var x = 10, y = 20;
undefined
> x+y
30
```

- You can also create a js file and type in some javascript.

```
C:\Users\DELL>node helloworld.js
Hello World!
```

```
//helloworld.js
console.log("Hello World!");
```

Using the REPL

- To view the options available to you in REPL type .help and press Enter.

```
C:\Users\DELL>node
> .help
break   Sometimes you get stuck, this gets you out
clear   Alias for .break
exit    Exit the repl
help    Show repl options
load    Load JS from a file into the REPL session
save    Save all evaluated commands in this REPL session to a file
>
```

```
> .load helloworld.js
> console.log('Hello World!!');
Hello World!!
undefined
>
```

```
for (var i = 1; i < 11; i++)
    console.log(i);
var arr1 = [10, 20, 30];
arr1.push(40,50);
console.log('arr length: ' + arr1.length); //5
console.log('arr contents: ' + arr1);  //10,20,30,40,50
```

```
//functionEx.js
function foo() {
   return 123;
   }
console.log(foo()); // 123

function bar() { }
console.log(bar()); // undefined
```

```
//JSObjEx.js
var person = {
    name: "Amit",
    age: 23,
    addr: {
        city: 'Pune',
        state: 'Mah'
    },
    hobbies: ['Reading', 'Swimming']
};
console.log(person);
```

```
E:\FreelanceTrg\Node.js\Demo\Intro>node JSObjEx.js
{
  name: 'Amit',
  age: 23,
  addr: { city: 'Pune', state: 'Mah' },
  hobbies: [ 'Reading', 'Swimming' ]
}
```

Node js Modules

- A module in Node.js is a logical encapsulation of code in a single unit.
  - Since each module is an independent entity with its own encapsulated functionality, it can be managed as a separate unit of work.
- Consider modules to be the same as JavaScript libraries.
  - A set of functions you want to include in your application.
  - Module in Node.js is a simple or complex functionality organized in JavaScript files which can be reused throughout a Node.js application.
  - A module is a discrete program, contained in a single file in Node.js. Modules are therefore tied to files, with one module per file.
- Node.js has a set of built-in modules which you can use without any further installation.
  - Built-in modules provide a core set of features we can build upon.
  - Also, each module can be placed in a separate .js file under a separate folder.
  - To include a module, use the **require()** function with the name of the module.
- In Node, modules are referenced either by file path or by name
  - For example, we can require some native modules:

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

```
var myFile = require('./myFile'); // loads myFile.js
```

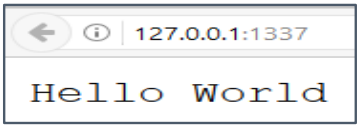
Node.js Web App

```
//RunServer.js
var http = require("http");

function process_request(req, res) {
   var body = 'Hello World\n';
   var content_length = body.length ;
   res.writeHead(200, {
            'Content-Length': content_length,
            'Content-Type': 'text/plain'   });

    res.end(body);
}

var srv = http.createServer(process_request);
srv.listen(1337, '127.0.0.1');

console.log('Server running at http://127.0.0.1:1337/');
```

Import required module using **require**; load http module and store returned HTTP instance into http variable

createServer() : turns your computer into an HTTP server
Creates an HTTP server which listens for request over 1337 port on local machine

```
G:\FreeLanceTrg\Node.js\Demo\Intro>node runserver.js
Server running at http://127.0.0.1:1337/
```




```
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
   })
  .listen(1337, '127.0.0.1');
```

Node.js Module

- Node.js includes three types of modules:
  - Core Modules
  - Local Modules
  - Third Party Modules

- Loading a core module
  - Node has several modules compiled into its binary distribution called core modules.
  - It is referred solely by the module name, not by the path and are preferentially loaded even if a third-party module exists with the same name.
  - var http = require('http');
- Some of the important core modules in Node.js

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods &events to work with file I/O.
util	util module includes utility functions useful for programmers.

Node.js Local Module

- The local modules are custom modules that are created locally by developer in the app
  - These modules can include various functionalities bundled into distinct files and folders
  - You can also package it and distribute it via NPM, so that Node.js community can use it.
  - For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

```
//Module1.js : exporting variable
exports.answer=50;
```

```
//UseModule1.js
var ans=require("./module1");
console.log(ans.answer);
```

exports object is a special ***object*** created by the Node module system which is returned as the value of the require function when you include that module.

```
E:\FreelanceTrg\Node.js\Demo\Modules>node module1.js

E:\FreelanceTrg\Node.js\Demo\Modules>node UseModule1.js
50
```

module2.js > ...

```
1   //exporting function
2   exports.sayHelloInEnglish = function(){
3   return "Hello";
4   };
5   exports.sayHelloInSpanish = function(){
6   return "Hola";
7   };
```

UseModule2.js > ...

```
1   var greetings=require("./module2");
2   console.log(greetings.sayHelloInSpanish());
3   console.log(greetings.sayHelloInEnglish());
4   /*
5   This is equivalent to:
6   var greetings=require("./module2").sayHelloInSpanish();
7   console.log(greetings);
8   */
```

Create Your Own Modules

module1a.js > ...

```
    exports.bname = 'Core Node.js';
    exports.read = function() {
        console.log('I am reading ' + exports.bname);
    }
```

UseModule1a.js > ...

```
1   var book = require('./module1a.js');
2   console.log('Book name: ' + book.bname);
3   book.read();
```

```
Book name: Core Node.js
I am reading Core Node.js
```

module2a.js > ...

```
1   exports.func1 = function() {
2       console.log('in function func1()');
3     };
4
5   module.exports.func2 = function() {
6       console.log('in function func2()');
7     };
```

UseModule2a.js > ...

```
1   const f1 = require('./module2a');
2   console.log(f1);
3
4   f1.func1();
5   f1.func2();
```

```
{ func1: [Function], func2: [Function] }
in function func1()
in function func2()
```

- Exports is just module.exports's little helper. Your module returns module.exports to the caller ultimately, not exports. All exports does is collect properties and attach them to module.exports

module2a.js > ...

```
1   exports.func1 = function() {
2       console.log('in function func1()');
3     };
4
5   module.exports.func2 = function() {
6       console.log('in function func2()');
7     };
```

UseModule2a.js > ...

```
1   const f1 = require('./module2a');
2   console.log(f1);
3
4   f1.func1();
5   f1.func2();
```

```
{ func1: [Function], func2: [Function] }
in function func1()
in function func2()
```

```
//module7.js
function sayHello(){
  console.log("Hello World!")
}

const username = 'Joan'

module.exports = {username, sayHello}

//or, using dot notation:
module.exports.username = username
module.exports.sayHello = sayHello

//alternatively
exports.username = username
exports.sayHello = sayHello
```

```
//import the assigned properties with a destructuring assignment:
const { username, sayHello } = require('./module7')

//Or with a regular assignment and dot notation:

const helloModule = require('./module7')

helloModule.sayHello()
console.log(helloModule.username)
```

- To Expose properties and functions we use exports
- On other hand, to expose user defined objects (class) and JSON objects, we use module.exports

```
class User {
  constructor(name, age, email) {
    this.name = name;
    this.age = age;
    this.email = email;
  }

  getUserStats() {
    return `
      Name: ${this.name}
      Age: ${this.age}
      Email: ${this.email}
    `;
  }
}

module.exports = User;
```

```
const User = require('./user');
const jim = new User('Jim', 37, 'jim@example.com');

console.log(jim.getUserStats());
```

module3a.js > ...

```
//exposing JSON object
module.exports = {
    firstName: 'James',
    lastName: 'Bond',
    display: function(){
      console.log(this.firstName);}
}
exports.ans=40;
```

UseModule3a.js > ...

```
var person =require("./module3a");
console.log("First Name : " + person.firstName);
console.log("Last Name : " + person.lastName);
person.display();
console.log(person.ans);
```

```
First Name : James
Last Name : Bond
James
undefined
```

module4.js > ...

```
//exporting result of a function that takes args
exports.add = function() {
    var sum = 0, i = 0;
    while (i < arguments.length) {
      sum += arguments[i++];
    }
    return sum;
};
```

UseModule4.js > ...

```
var result=require("./module4").add(10,20,30,40);
console.log(result);
/*
This is equivalent to:
var addNumbers=require("./module4");
console.log(addNumbers.add(10,20,30,40));
*/
```

module5.js > ...

```
function printA() {
console.log('A');
}
function printB() {
console.log('B');
}
function printC() {
console.log('C');
}

module.exports.pi = Math.PI;
module.exports.printA = printA;
module.exports.printB = printB;
```

UseModule5.js > ...

```
var module5 = require('./module5');
module5.printA(); // -> A
module5.printB(); // -> B
console.log(module5.pi); // -> 3.141592653589793
```

Example

module6.js > ...

```
//export methods and values as you go, not just at the end of the file.
/*exports.getName = () => {
    return 'Jim';
  };
exports.getLocation = () => {
    return 'Munich';
  };
exports.dob = '12.01.1982';
*/
//We can export multiple methods and values in the same way
const getName = () => {
    return 'Jim';
  };
  const getLocation = () => {
    return 'Munich';
  };
  const dateOfBirth = '12.01.1982';

  exports.getName = getName;
  exports.getLocation = getLocation;
  exports.dob = dateOfBirth;
```

UseModule6.js > ...

```
//we can cherry-pick what we want to import
const { getName, dob } = require('./module6');
console.log(
  `${getName()} was born on ${dob}.`
);  //Jim was born on 12.01.1982.
```

NPM (Node Package Manager)

- Loading a module(Third party) installed via NPM
  - To use the modules written by other people in the Node community and published on the Internet (**npmjs.com**).
  - We can install those third party modules using the **Node Package Manager** which is installed by default with the node installation.
  - Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages in your application.
  - It is also an online repository for open-source Node.js packages. The node community around the world creates useful modules and publishes them as packages in this repository.
  - NPM is a command line tool that installs, updates or uninstalls Node.js packages in your application.
  - After you install Node.js, verify NPM installation : **npm -v**

```
C:\Users\Shrilata>node -v
v14.16.0

C:\Users\Shrilata>npm -v
6.14.11
```

NPM (Node Package Manager)

- Installing Packages
  - In order to use a module, you must install it on your machine.
  - To install a package, type npm install, followed by the package name

- There are two ways to install a package using npm: globally and locally.
- **Globally** − This method is generally used to install development tools and CLI based packages. To install a package globally, use the following code.
  - **npm install -g <package-name>**
  - Eg to install expressJS : npm install -g express
  - Eg to install Typescript : npm install –g typescript
  - Eg to install Angular : npm install -g @angular/cli

- **Locally** − This method is generally used to install frameworks and libraries. A locally installed package can be used only within the directory it is installed.
  - To install a package locally, use the same command as above without the -g flag.
  - **npm install <package-name>**
  - Eg : To install cookie parser in Express : npm install --save cookie-parser
  - Eg: to install bootstrap : npm install bootstrap@4.1.1

NPM (Node Package Manager)

- When packages are installed, they are saved on local machine
- npm installs module packages to the node_modules folder.

- **Installing a package using NPM :** *$ npm install [g] <Package Unique Name>*

- **To remove an installed package :** *npm uninstall [g] < Package Unique Name>*

- **To update a package to its latest version :** *npm update [g] < Package Unique Name>*

Loading a third party module : package.json

- The package.json file in Node.js is the heart of the entire application.
  - It is basically the manifest file that contains the metadata of the project.
  - package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.
  - It must be located in project's root directory.
  - It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application.
  - Your project also must include a package.json before any packages can be installed from NPM.
  - Eg : a minimal package.json:

```
{
  "name" : "barebones",
  "version" : "0.0.0",
}
```

    - The name field should explain itself: this is the name of your project. The version field is used by npm to make sure the right version of the package is being installed.

Loading a third party module

- Lets say I want to create a ExpressJS application. I will install ExpressJs locally.
  - Step-1) choose a empty folder
  - Step-2) run npm init to create a package.json file
  - Step-3) install express : npm install express –save (to update package.json)
  - Step-4) check the updated json file to see new dependencies

  - See how package.json is installed in root folder along with node_modules folder
  - My Express app is dependent on a number of other modules
  - All these dependencies will have an entry in package.json

Loading a file module

- Loading a file module (User defined module)
  - We load non-core modules by providing the absolute path / relative path.
  - Node will automatically add the .js extension to the module referred.
  - var myModule = require('d:/shrilata/nodejs/module'); // Absolute path for module.js
  - var myModule = require('**../**module'); // Relative path for module.js (one folder up level)
  - var myModule = require('./module'); // Relative path for module.js (Exists in current directory)

If the given path does not exist, require() will throw an Error with its code property set to 'MODULE_NOT_FOUND'.

package.json

- The package.json file in Node.js is the heart of the entire application.
- It is basically the manifest file that contains the metadata of the project.
- package.json is a configuration file from where the npm can recognize dependencies between packages and installs modules accordingly.
  - It must be located in project's root directory.
- package-lock.json file
  - Introduced in version 5; keeps track of the exact version of every package that is installed so that a product is 100% reproducible in the same way even if packages are updated by their maintainers.
  - The package-lock.json sets your currently installed version of each package in stone, and npm will use those exact versions when running npm install.

Buffers

- A buffer is an area of memory; It represents a fixed-size chunk of memory (can't be resized) allocated outside of the V8 JavaScript engine.
  - You can think of a buffer like an array of integers, which each represent a byte of data.
  - It is implemented by the Node.js Buffer class.

- **Creating Buffer**
  - It is possible to create your own buffer! Aside from the one Node.js will automatically create during a stream, it is possible to create and manipulate your own buffer
  - A buffer is created using the : Buffer.alloc(), Buffer.allocUnsafe() , Buffer.from()

  - Buffer.alloc(size, fill, encoding);
    - Size: Desired length of new Buffer. It accepts integer type of data.
    - Fill: The value to prefill the buffer. The default value is 0.It accepts any of the following: integer, string, buffer type of data.
    - Encoding: It is Optional. If buffer values are string , default encoding type is utf8. Supported values are: ("ascii","utf8","utf16le","ucs2","base64","latin1", "binary", "hex")

  - Eg : Create a buffer of length 20, with initializing all the value to fill as 0 in hexadecimal format

```
var zerobuf = Buffer.alloc(20);
console.log(zerobuf);
//<Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>
```


Node.js fs (File System) Module

- The fs module provides a lot of very useful functionality to access and interact with the file system.
  - There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:
  - const fs = require('fs')
- This module provides a wrapper for the standard file I/O operations.
- All the methods in this module has asynchronous and synchronous forms.
  - synchronous methods in this module ends with 'Sync'. For instance renameSync() is the synchronous method for rename() synchronous method.
  - The asynchronous form always take a completion callback as its last argument.
  - The arguments passed to the completion callback depend on the method, but the first argument is always reserved for an exception. If the operation was completed successfully, then the first argument will be null or undefined.

  - When using the synchronous form any exceptions are immediately thrown. You can use try/catch to handle exceptions or allow them to bubble up.

Asynchronous method is preferred over synchronous method because it never blocks the program execution where as the synchronous method blocks.

Node.js File System

- Node fs module provides an API to interact with FileSystem and to perform some IO operations like create a file, read a File, delete a File etc..
  - fs module is responsible for all the async or synchronous file I/O operations.

```
var fs = require('fs');
// write
fs.writeFileSync('test.txt', 'Hello fs!');
// read
console.log(fs.readFileSync('test.txt')); //<Buffer 48 65 6c 6c 6f 20 66 73 21>
console.log(fs.readFileSync('test.txt').toString());   //Hello fs!

console.log(fs.readFileSync('test.txt','utf8'));  //Hello fs!
```

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

// Asynchronous read
fs.readFile('test.txt', function (err, data) {
   if (err) {
      return console.error(err);
   }
   console.log("Asynchronous read: " + data.toString());
});

// Synchronous read
var data = fs.readFileSync('test.txt');
console.log("Synchronous read: " + data.toString());

console.log("Program Ended");
```

```
Synchronous read: Hello fs!
Program Ended
Asynchronous read: Hello fs!
```

```
var fs = require('fs');
fs.writeFile('test.txt', 'Hello World!', function (err) {
        if (err)
           console.log(err);
        else
           console.log('Write operation complete.');
});
```

Node.js File System

- fs.readFile(fileName [,options], callback) : read the physical file asynchronously.
- fs.writeFile(filename, data [, options], callback) : writes data to a file
- fs.appendFile(): appends the content to an existing file
- fs.unlink(path, callback); delete an existing file

```
var fs = require("fs")
fs.unlink("test1.txt", function(err){
    if(err) console.log("Err : " , err);
    console.log("delete successful")
 })
```

- fs.exists(path, callback) : determines if specified file exists or not
- fs.close(fd, callback(err));
- fs.rename(oldPath, newPath, callback): rename a file or folder

```
fs.rename("src.json", "tgt.json", err => {
    if (err) {
      return console.error(err)
    }
    console.log('Rename operation complete.');
  });
```

Node.js File System

- The exists() and existsSync() methods are used to determine if a given path exists.
- Both methods take a path string as an argument.
  - If existsSync() is used, a Boolean value representing path's existence is returned.
  - If exists() is used, the same Boolean value is passed as an argument to the callback function.

```
var fs = require("fs");
var path = "/";

fs.exists(path, function(exists) {
    if (exists)
        console.log(path + " exists: " + exists);
    else
        console.error("Something is wrong!");
});
```

- Reading Directories
  - We can use the fs.readdir() method to list all the files and directories within a specified path:

```
const fs = require('fs')
fs.readdir('./', (err, files) => {
    if (err) {
        console.error(err)
        return
    }
    console.log('files: ', files)
})
```


Node.js File System

- fs.open(path, flags[, mode], callback) : opens a file for reading or writing in async
  - path - string having file name including path.
  - flags - tells the behavior of the file to be opened
  - mode - sets the file mode; defaults to 0666, readable and writeable.
  - callback - function which gets two arguments **(err, fd)**.

```
// Asynchronous - Opening File
console.log("Going to open file!");
fs.open(test.txt', 'r+', function(err, fd) {
   if (err) {
      return console.error(err);
   }
   console.log("File opened successfully!");
});
```

Flag	Description
r	Open file for reading. An exception occurs if the file does not exist.
r+	Open file for reading and writing. An exception occurs if the file does not exist.
rs	Open file for reading in synchronous mode.
w	Open file for writing. The file is created (if it does not exist) or overwritten (if it exists).
wx	Like 'w' but fails if path exists.
w+	Open file for read+write. The file is created (if it doesn't exist) or overwritten (if it exists).
wx+	Like 'w+' but fails if path exists.
a	Open file for appending. The file is created if it does not exist.
ax	Like 'a' but fails if path exists.
a+	Open file for reading and appending. The file is created if it does not exist.

Node.js File System

- fs.read(fd, buffer, offset, length, position, callback) : reads an opened file; from the file specified by fd.
  - fd <Integer> - is a file descriptor, a handle, to the file
  - buffer <String> | <Buffer> : buffer into which the data will be read
  - offset <Integer> : offset in the buffer to start reading at.
  - length <Integer> : specifies the number of bytes to read
  - position <Integer> : specifies where to begin reading from in the file
  - callback (err, bytesRead)

- fs.write(fd, string[, position[, encoding]], callback): write into an opened file; specified by fd
- Alternatively :fs.write(fd, buffer[, offset[, length[, position]]], callback)

  - offset : offset in the buffer to start writing at
  - length : specifies the number of bytes to write.
  - position : specifies where to begin writing
  - The callback will be given three arguments (err, bytesWritten, buffer) where bytesWritten specifies how many bytes were written from buffer.

Example : read file and write to file

```
var fs = require("fs");
fs.open('bigsample.txt', 'r', function (err, fd) {
   if (err)
      return console.error(err);
   var buffr = Buffer.alloc(1024);

   fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {
      if (err) throw err;
      // Print only read bytes to avoid junk.
      if (bytes > 0) {
          console.log(buffr.toString());
      }

        // Close the opened file.
       fs.close(fd, function (err) {
          if (err) throw err;
       });
   });
});
```

- __filename, is the absolute path of the currently executing file.
- __dirname is the absolute path to the directory containing the currently executing file

```
var fs = require("fs");
var path = __dirname + "/sampleout.txt";
var data = "Lorem ipsum dolor sit amet";
fs.open(path, "w", function(error, fd) {
   var buffer = Buffer.from(data);
   fs.write(fd, buffer, 0, buffer.length, null, function(error, written, buffer) {
      if (error) {
          console.error("write error: " + error.message);
      } else {
            console.log("Successfully wrote " + written + " bytes.");
            }
    });
});
```

File System

- fs.stat(path, callback) : gets the information about file on path
  - callback function gets two arguments (err, stats) where stats is an object of fs.Stats type

stats.isFile()	Returns true if file type of a simple file.
stats.isDirectory()	Returns true if file type of a directory.
stats.isBlockDevice()	Returns true if file type of a block device.
stats.isCharacterDevice()	Returns true if file type of a character device.
stats.isSymbolicLink()	Returns true if file type of a symbolic link.
stats.isFIFO()	Returns true if file type of a FIFO.
stats.isSocket()	Returns true if file type of asocket.

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

console.log("Going to get file info!");
fs.stat('test.txt', function (err, stats) {
   if (err) {
      console.log(err.code + " (" + err.message + ")");
      return console.error(err);
   }
   console.log(stats);
   console.log("Got file info successfully!");

   // Check file type
   console.log("isFile ? " + stats.isFile());
   console.log("isDirectory ? " + stats.isDirectory());
   console.log("File size ? " +stats.size  );
});
```

```
Stats {
  dev: 1154251169,
  mode: 33206,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: 4096,
  ino: 1407374884059506,
  size: 9,
  blocks: 0,
  atimeMs: 1605632713708.1055,
  mtimeMs: 1605632713708.1055,
  ctimeMs: 1605632713708.1055,
  birthtimeMs: 1605631297622.6255,
  atime: 2020-11-17T17:05:13.708Z,
  mtime: 2020-11-17T17:05:13.708Z,
  ctime: 2020-11-17T17:05:13.708Z,
  birthtime: 2020-11-17T16:41:37.623Z
}
Got file info successfully!
isFile ? true
isDirectory ? false
File size ? 9
```

File System : example

```
var fs = require("fs");
var fileName = "sample.txt";

fs.exists(fileName, function(exists) {
  if (exists) {
    fs.stat(fileName, function(error, stats) {
      fs.open(fileName, "r", function(error, fd) {
        var buffer = Buffer.alloc(stats.size);
        console.log(stats.size + "  " + fd);   //19  3
        fs.read(fd, buffer, 0, buffer.length, null, function(error, bytesRead, buffer) {
          var data = buffer.toString("utf8", 0, buffer.length);
          console.log(data);  //This is sample text
          fs.close(fd, function (err) {
            if (err) throw err;
          });
        });
      });
    });
  }
});
```

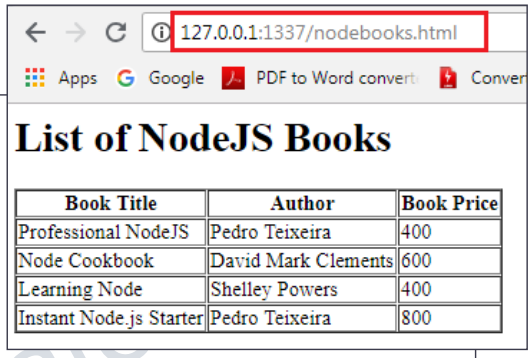
URL Core module

- The url module provides utilities for URL resolution and parsing.
  - It splits up a web address into readable parts.

```
var url = require('url');
//var adr = 'http://localhost:8080/MyApp/welcome.html?year=2017&month=february';
var adr = 'http://someserver.com/processLogin.jsp?username=soha&password=secret';

var q = url.parse(adr, true);    //Parse an address

console.log("Host : " , q.host); //returns 'someserver.com'
console.log("Pathname : " , q.pathname); //returns '/processLogin.jsp'
console.log("Search : " , q.search); //returns '?username=soha&password=secret'
console.log("Href : " , q.href);  //returns 'http://someserver.com/processLogin.jsp?username=soha
&password=secret'
console.log("Protocol : " , q.protocol);  //returns 'http:'

var qdata = q.query; //returns an object: { year: 2017, month: 'february' }
console.log(qdata);   //returns '{ username: 'soha', password: 'secret' }'
console.log(qdata.username, qdata.password); //returns 'soha secret'
```

```
E:\FreelanceTrg\Node.js\Demo\urlmodule>node CoreURLModuleEg.js
Host :  someserver.com
Pathname :  /processLogin.jsp
Search :  ?username=soha&password=secret
Href :  http://someserver.com/processLogin.jsp?username=soha&password=secret
Protocol :  http:
[Object: null prototype] { username: 'soha', password: 'secret' }
soha secret
```


URL Core module

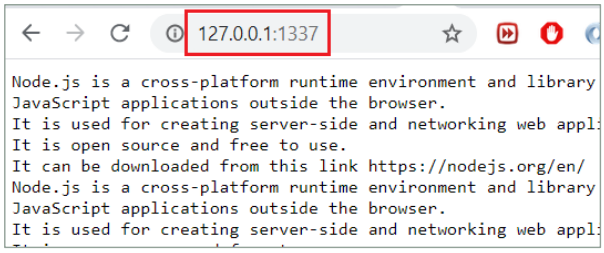



```
var http = require('http');
var url = require('url');
var fs = require('fs');
function process_request(req, res) {
  var q = url.parse(req.url, true);
  var filename = "." + q.pathname;
  fs.readFile(filename, function(err, data) {
  if (err) {
      res.writeHead(404, {'Content-Type': 'text/html'});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

var s = http.createServer(process_request);
s.listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

Web development with Node : http.ServerRequest

- When listening for request events, the callback gets an http.ServerRequest object as the first argument (function(req,res))
- This object contains some properties:
  - req.url: This property contains the requested URL as a string
    - It does not contain the schema, hostname, or port, but it contains everything after that.
    - Eg : if URL is :http://localhost:3000/about?a=20 then req.url will return /about?a=20
  - req.method: This contains the HTTP method used on the request. It can be, for example, GET, POST, DELETE, or HEAD.
  - req.headers: This contains an object with a property for every HTTP header on the request.
  - Eg : Serving file on request : Reading a file at server end and serving to browser!

```
var http = require('http');
var fs = require('fs');
function process_request(req, res) {
   fs.readFile('bigsample.txt', function(err, data) {
      res.writeHead(200, {'Content-Type': 'text/plain'});
      res.write(data);
      res.end();
   });
}
var s = http.createServer(process_request);
s.listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

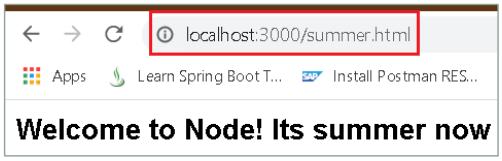


Serving file on request

```
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(3000);    //invoke as http://localhost:3000/summer.html+
```

./summer.html

```
<!-- summer.html -->
<html>
<head>
<title>Hello there</title>
</head>
<body>
<h3>Welcome to Node!
Its summer now</h3>
</body>
</html>
```



Routing

- Routing refers to the mechanism for serving the client the content it has asked

```
var http = require('http');
var server = http.createServer(function(req,res){
    //console.log(req.url);
var path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase();
switch(path) {
    case '' :
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end('<h1>Home Page</h1>');
        break;
    case '/about' :
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end('<h1>About us</h1>');
        break;
    case '/admin' :
        res.writeHead(200, {'Content-Type': 'text/html'});
        res.end('<h1>Admin page</h1>');
        break;
    default:
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('Not Found');
        break;
    }
});
server.listen(3000);
```

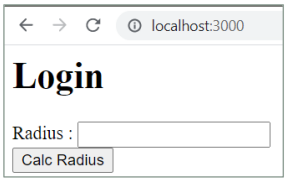
```
var http = require("http");
http.createServer(function(request, response) {
 if (request.url === "/" && request.method === "GET") {
      response.writeHead(200, { "Content-Type": "text/html" });
      response.end("Hello <strong>home page</strong>");
 }
  else if (request.url === "/foo" && request.method === "GET") {
      response.writeHead(200, { "Content-Type": "text/html" });
      response.end("Hello <strong>foo</strong>");
 }
 else if (request.url === "/bar" && request.method === "GET") {
      response.writeHead(200, { "Content-Type": "text/html" });
      response.end("Hello <strong>bar</strong>");
 }
 else {
      response.writeHead(404, { "Content-Type": "text/html" });
      response.end("404 Not Found");
}
}).listen(8000);
```

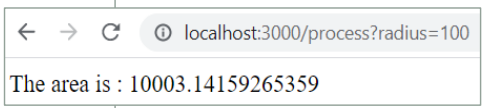


```
var http = require('http');
var url = require('url');
var fs = require('fs');

function process_req(req, res) {
  if (req.method == 'GET' && req.url == '/') {
    fs.readFile('radius.html', function(err, data) {
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.write(data);
      res.end();
   });
  }
  else if(req.method == 'GET' && req.url.substring(0,8) == '/process'){
     var q = url.parse(req.url, true);
     var qdata = q.query;
     var r = qdata.radius;

     var rad = Math.PI + r * r;
     res.writeHead(200, {'Content-Type': 'text/html'});
     res.write("The area is : " + rad);
     res.end();
   }
   else
     res.end("not found");
}
var server = http.createServer(process_req)
server.listen(3000);
console.log('server listening on localhost:3000');
```

{ radius: '100' }

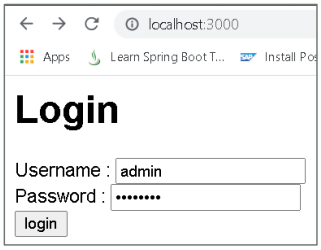
```
<html>
<body>
  <h1>Login</h1>
   <form action="process">
    Radius : <input name="radius">
   <input type="submit"
          value="Calc Radius">
   </form>
</body></html>
```

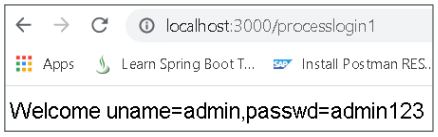




```
var http = require('http');
var fs = require('fs');

function process_req(req, res) {
  if (req.method == 'GET' && req.url == '/') {
     fs.readFile('loginPost.html', function(err, data) {
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.write(data);
      res.end();
   });
  }
  else if(req.method == 'POST'){
     var body = "";
     req.on("data",function(data){
        body += data;
        res.writeHead(200, {'Content-Type': 'text/html'});
        var arr = body.split("&");
        res.write("Welcome " + arr);
        res.end();
      })
  }
}
var server = http.createServer(process_req)
server.listen(3000);
console.log('server listening on localhost:3000');
```

```
<body>   //loginpost.html
  <h1>Login</h1>
   <form action="processlogin"
         method="post">
    Username :
      <input name="uname"><br>
    Password :
      <input type="password"
             name="passwd"><br>
      <input type="submit"
             value="login">
   </form>
</body>
```





Promise

- Promises are a new feature of ES6.
  - It's a method to write asynchronous code; it represents the completion of an asynchronous function.
  - It is a Javascript object that might return a value in the future.
  - It accomplishes the same basic goal as a callback function, but with many additional features and a more readable syntax.
- Creating a Promise
  - Promises are created by using a constructor called Promise and passing it a function that receives two parameters, resolve and reject, which allow us to indicate to it that it was resolved or rejected.

```
let promise = new Promise(function(resolve, reject) {
  // executor code - things to do to accomplish your promise
});
```

```
let promise = new Promise(function(resolve, reject) {
  // things to do to accomplish your promise

  if(/* everything turned out fine */) {
      resolve('Stuff worked')
  } else { // for some reason the promise doesn't fulfilled
      reject(new Error('it broke'))
  }
})
```

Consuming a Promise

- The promise we created earlier has fulfilled with a value, now we want to be able to access the value.
- Promises have a method called then() that will run after a promise reaches resolve in the code.
- The then() method returns a Promise. It takes up to two arguments: callback functions for the success and failure cases of the Promise.
- Syntax : p.then(onFulfilled[, onRejected]);
  - onFulfilled function called if the Promise is fulfilled. This function has one argument, the fulfillment value.
  - Example

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

```
promise.then(function(result) {
  console.log("Promise worked");
}, function(err) {
  console.log("Something broke");
});
```


Consuming a Promise

- Full example

```
const promise = new Promise((resolve, reject) => {
  if(true)
     resolve("resolved!!")
});

promise.then(msg => console.log("In then - "+ msg))
```

Console

```
"In then - resolved!!"
```

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve("done!"), 3000);
  });

  // resolve runs the first function in .then
  promise.then(
    result =>console.log(result), //shows "done!" after 3 seconds
    error =>console.log(error) // doesn't run
  );
```

```
D:\>node one.js
done!
```

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

Consuming a Promise

- Alternatively, instead of using this syntax of then(fulfilled, rejected), you can also use catch()
- The **then** and **catch** handlers are asynchronous.
  - Basically, then and catch will be executed once Javascript finished reading the code

```
promise.then(function(result) {
  console.log(result)
}).catch(function(err) {
  console.log(err)
})

console.log('Hello world')
```

```
const promise = new Promise((resolve, reject) => {
  // Note: only 1 param allowed
  return reject('Hi')
})

// Parameter passed into reject would be the arguments passed into
 catch.
promise.catch(err => console.log(err)) // Hi
```

```
function randomDelayed(max = 10, expected = 5, delay =  1000) {
  return new Promise((resolve, reject) => {
    const number = Math.floor(Math.random() * max)

    setTimeout(
      () => number > expected
        ? resolve(number)
        : reject(new Error('lower than expected number')), 1000
    );
  });
}
randomDelayed(100, 75, 2500)
    .then(number => console.log(number))
    .catch(error => console.error(error.toString()));
```

Console

```
× "Error: lower than expected number"
  78
× "Error: lower than expected number"
  89
× "Error: lower than expected number"
× "Error: lower than expected number"
× "Error: lower than expected number"
  85
× "Error: lower than expected number"
```


Chained Promises

- The methods promise.then(), promise.catch(), and promise.finally() can be used to associate further action with a promise that becomes settled.
  - Each .then() returns a newly generated promise object, which can optionally be used for chaining

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});

myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

```
const myPromise = new Promise((resolve) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});

myPromise
  .then((x)=>{console.log("in A - " + x);})
  .then((x)=>{console.log("in B - " + x);})
  .then((x)=>{console.log("in C - " + x);});
```

Console

```
"in A - foo"
"in B - undefined"
"in C - undefined"
```

```
import fs from 'fs';

function readAFile(path) {
  return new Promise((resolve, reject) => {
    fs.readFile(path, 'utf8', (error, data) => {
      if (error) return reject(error);
      return resolve(data);
    });
  });
}

readAFile('./file.txt')
  .then(data => console.log(data))
  .catch(error => console.error(error));
```

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest()
    xhr.open("GET", url)
    xhr.onload = () => resolve(xhr.responseText)
    xhr.onerror = () => reject(xhr.statusText)
    xhr.send()
  });
}
```

Ajax example with promise

```
<!DOCTYPE html>
<html>
<body>
  <div id="msg"></div>
  <button id="btnGet">Get Message</button>
</body>
</html>
```

- You'll see Promises used a lot when fetching data from an API
- Promises can be confusing, both for new developers and experienced programmers that have never worked in an asynchronous environment before.
- However, it is much more common to consume promises than create them. Usually, a browser's Web API or third party library will be providing the promise, and you only need to consume it

```
function load(url) {
   return new Promise(function (resolve, reject) {
     const request = new XMLHttpRequest();
     request.onreadystatechange = function (e) {
        if (this.readyState === 4) {
           if (this.status == 200) {
              resolve(this.response);
            } else {
              reject(this.status);
            }
         }
      }
      request.open('GET', url, true);
      request.send();
    });
  }
```

```
btn.onclick = function () {
  load('data.json')
    .then(
      response => {
        const result = JSON.parse(response);
         $("#msg").text(result.message);
      },
      error =>  $("#msg").text(`Error getting message,
 HTTP status: ${error}`
  );
}
```


Using Fetch

- One of the most useful and frequently used Web APIs that returns a promise is the Fetch API.
  - It allows you to make an asynchronous resource request over a network.
  - fetch() is a two-part process, and therefore requires chaining then()

```
// Fetch a user from the GitHub API
fetch('https://api.github.com/users/octocat')
  .then((response) => {
    return response.json()
  })
  .then((data) => {
    console.log(data)
  })
  .catch((error) => {
    console.error(error)
  })
```

Console

```
[object Object] {
  avatar_url: "https://avatars.githubus
  bio: null,
  blog: "https://github.blog",
  company: "@github",
  created_at: "2011-01-25T18:44:36Z",
  email: null,
  events_url: "https://api.github.com/u
```

async/await

- async/await added in ECMAScript 2017 (ES8)
- async/await is built on promises
- **async** keyword put in front of a function declaration turns it into an async function
- An async function is a function that knows how to expect the possibility of the **await** keyword being used to invoke asynchronous code.
  - We use the async keyword with a function to represent that the function is an asynchronous function.
  - The async function returns a promise.

```
//simple synchromous JS function and invocation
function hello() { return "Hello" };
hello();
```

```
//converting above JS function into async function
async function hello() { return "Hello" };
hello();
```

```
// You can also create an async function expression like this:
let hello = async function() { return "Hello" };
hello();
```

```
//you can also use arrow functions:
let hello = async () => { return "Hello" };
```


async/await

- To consume the value returned when the promise fulfills (since it is returning a promise) use a .then() block

```
hello().then((value) => console.log(value))
//or even just shorthand such as
hello().then(console.log)
```

```
//complete example
async function hello(){
  return "hello-1"
}
hello().then((x)=>console.log(x))
// returns"hello-1"
```

```
let f = async () => {
  console.log('Async function.');
  return Promise.resolve("Hello-1");  //function returns a promise
}

f().then(function(result) {
  console.log(result)
});
```

```
Console
"Async function."
"Hello-1"
```

The await keyword

- The **await** keyword is used inside the async function to wait for the asynchronous operation
  - await can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value.
  - Syntax: `let result = await promise;`
  - You can use await when calling any function that returns a Promise, including web API functions

```
// a promise
let promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
     resolve('Promise resolved')}, 4000);
});


async function asyncFunc() {// async function

  // wait until the promise resolves
  let result = await promise;

  console.log(result);
  console.log('hello');
}

asyncFunc(); // calling the async function
```

```
Console
"Promise resolved"
"hello"
```

```
function logFetch(url) {
  return fetch(url)
    .then(response => response.text())
    .then(text => {
      console.log(text);
    }).catch(err => {
      console.error('fetch failed', err);
    });
}
```

```
async function logFetch(url) {
  try {
    const response = await fetch(url);
    console.log(await response.text());
  }
  catch (err) {
    console.log('fetch failed', err);
  }
}
```


Next gen Javascript

- **const** : from JS 1.5 onwards.- to define constants
  - Eg :

```
const myBirthday = '18.04.1982';
myBirthday = '01.01.2001';          // error, can't reassign the constant!
```

```
const LANGUAGES = ['Js', 'Ruby', 'Python', 'Go'];
LANGUAGES = "Javascript"; // shows error.
LANGUAGES.push('Java'); // Works fine.
console.log(LANGUAGES); // ['Js', 'Ruby', 'Python', 'Go', 'Java']
```

- **let** : to define block-scoped variables; can be used in four ways:
  - as a variable declaration like var; in a for or for/in loop, as a substitute for var;
  - as a block statement, to define new variables and explicitly delimit their scope
  - to define variables that are scoped to a single expression.
  - Eg : let message = 'Hello!';
  - Eg : let user = 'John', age = 25, message = 'Hello';

```
if (true) {
 let a = 40;
 console.log(a); //40
}
console.log(a); // undefined
```

```
let a = 50;  let b = 100;
if (true) {
  let a = 60;
  var c = 10;
  console.log(a/c); // 6
  console.log(b/c); // 10
}
console.log(c); // 10
console.log(a); // 50
```

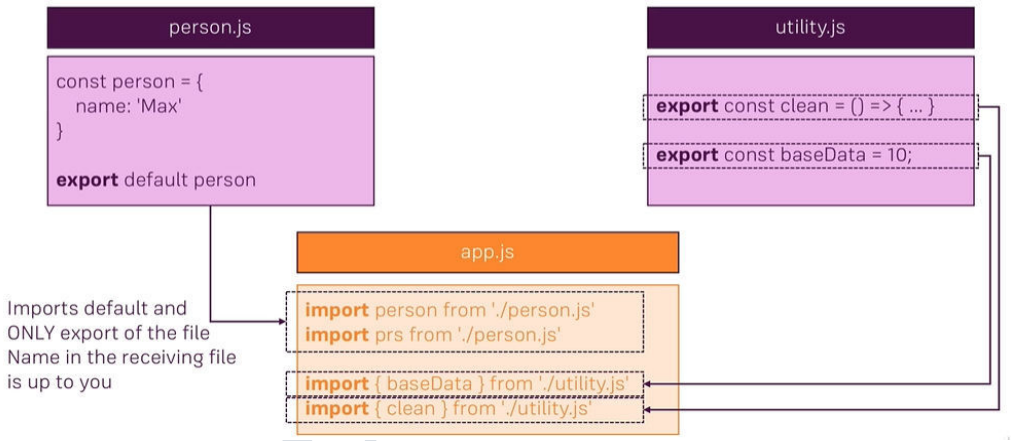
Next gen Javascript

- Arrow functions : same as lambda in TS

```
<script>
//non lambda
    function greet(name){
        console.log(name);
    }
    greet("shrilata");

    //lambda-eg1
    const greet1 = name => console.log(name);
    greet1("sandeep");

    //lambda-eg2
    const add = (a,b) => a + b;

    console.log(add(10,20));

    //lambda-eg3
    const strOp = str => {
                        console.log(str.length);
                        console.log(str.toUpperCase());
                        console.log(str.charAt(0));
                        };
     strOp("Hello");
```

Next gen Javascript

- Exports and imports




```
//  lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

//  someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))

//  otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

Next gen Javascript : Classes, properties and methods

```
class Person{
      fname = "Anil";
      lname = "Patil";
      getFullName = () => this.fname + " " + this.lname;
      }

      const p1 = new Person();
      console.log(p1.getFullName()); //Anil Patil


class Stud{
    constructor(name){
        this.sname=name;
    }
    getName(){
        console.log(this.sname); //sunita
     }
}
const s1 = new Stud("sunita");
s1.getName();
```

constructor method is always defined with the name "constructor"

Classes can have methods, which defined as functions, albeit without needing to use the function keyword.

```
class GradStud extends Stud{
    constructor(){
        super("Kavita");
        this.gpa = 4.9;
    }
    getDetails(){
         super.getName();    //Kavita
         console.log("GPA : " + this.gpa);
    }
}

const gs = new GradStud();
gs.getDetails();
```


Next gen Javascript : Classes, properties and methods

- To declare a class, you use the class keyword with the name of the class
- There can only be one "constructor" in a class; SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.

```
class Rectangle {
    constructor(height, width) {
      this.height = height;
      this.width = width;
    }
  }
```

```
class Rectangle {
    constructor(height, width) {
      this.height = height;
      this.width = width;
    }

    // Method
    calcArea() {
      return this.height * this.width;
    }
  }

  const square = new Rectangle(10, 10);

  console.log(square.calcArea()); // 100
```

Next gen Javascript : Spread and rest operators

- spread:

```
const numbers = [1,2,3];
console.log(numbers);

//spread
const newNumbers = [...numbers, 4]
console.log(newNumbers)

const newNumbers1 = [numbers, 4]
console.log(newNumbers1)
```

```
[1, 2, 3]
[1, 2, 3, 4]
[[1, 2, 3], 4]
>
```

- rest operator : used to merge a list of function arguments into an array

```
//rest
function sortArgs(...args){
  console.log(args.sort());
}

sortArgs(5,2,7,3,9,1)

function evenNos(...args){

  console.log(args.filter(ele => (ele%2 ==0)));
}

evenNos(5,2,7,4,9,1,8)
```

```
[1, 2, 3, 5, 7, 9]
[2, 4, 8]
>
```

Next gen Javascript

- Destructuring : allows to easily extract array elements or object properties and store them in variables
  - Destructuring is useful because it allows you to do in a single line, what would otherwise require multiple lines

```
var rect = { x: 0, y: 10, width: 15, height: 20 };

// Destructuring assignment
var {x, y, width, height} = rect;
console.log(x, y, width, height); // 0,10,15,20

rect.x = 10;

// assign to existing variables using outer parentheses
({x, y, width, height} = rect);
console.log(x, y, width, height); // 10,10,15,20
```

```
const arr=[1,2,3,4,5]
var [a,b] = arr
console.log(a,b)  //1,2
```

Easily extract array elements or object properties and store them in variables

Array Destructuring

```
[a, b] = ['Hello', 'Max']
console.log(a) // Hello
console.log(b) // Max
```

Object Destructuring

```
{name} = {name: 'Max', age: 28}
console.log(name) // Max
console.log(age) // undefined
```

Array functions

- Array.filter
  - You can filter arrays by using the .filter(callback) method.
  - The result will be another array that contains 0 or more elements based on the condition (or the "check") that you have in the callback.

```
var nums = [1, 2, 3, 21, 22, 30];
var evens = nums.filter(i => i % 2 == 0);
```

```
const grades = [10, 2, 21, 35, 50, -10, 0, 1];

//get all grades > 20
const result = grades.filter(grade => grade > 20); // [21, 35, 50];

// get all grades > 30
grades.filter(grade => grade > 30); // ([35, 50])
```


Array functions

- Array.map() : returns a new array containing the result of invoking the callback function for every item in the array.

```
const numbers = [1, 2, 3];

const doubleNumArray = numbers.map((num) => {
  return num * 2;
});

console.log(numbers);
console.log(doubleNumArray);
```

```
[1, 2, 3]
[2, 4, 6]
>
```

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
console.log("roots is : " + roots );  //1,2,3
```

```
var numbers = [1, 2, 3, 4];
var doubled = numbers.map(i => i * 2);
var doubled = [for (i of numbers) i * 2];  //same as above
console.log(doubled); // logs 2,4,6,8
```

Reference and primitive types

```
const person = {
  name: 'Max'
};

const secondPerson = person;

console.log(secondPerson);
```

```
[object Object] {
  name: "Max"
}
```

```
const person = {
  name: 'Max'
};

const secondPerson = person;

person.name = 'Manu';

console.log(secondPerson);
```

```
[object Object] {
  name: "Manu"
}
```

```
const person = {
  name: 'Max'
};

const secondPerson = {
  ...person
};

person.name = 'Manu';

console.log(secondPerson);
```

```
[object Object] {
  name: "Max"
}
```

Next gen Javascript

- Template strings provide us with an alternative to string concatenation. They also allow us to insert variables into a string.
  - Traditional string concatenation uses plus signs or commas to compose a string using variable values and strings.
  - console.log(lastName + ", " + firstName + " " + middleName)
- With a template, we can create one string and insert the variable values by surrounding them with ${variable}.
  - console.log(`${lastName}, ${firstName} ${middleName}`)
  - Any JavaScript that returns a value can be added to a template string between the ${ } in a template string.

```
//Javascript : generating an html string
var msg1 = 'Have a great day';
var html = '<div>' + msg1 + '</div>';
document.write(html)

//Using template strings
var msg2 = 'Never give up';
var html1 = `<div>${msg2}</div>`;
document.write(html1)
```

BOOTSTRAP 4

Pre-reqs:

- HTML
- CSS
- JavaScript

Introduction

- Bootstrap is an opensource frontend framework developed by Twitter.
  - It is the most popular ***HTML, CSS, and JavaScript*** framework for developing responsive, mobile first web sites.
  - Bootstrap is a free and open source collection of tools for creating websites and web applications.
  - Bootstrap contains a set of CSS- and HTML-based templates for styling forms, elements, buttons, navigation, typography, and a range of other UI components.
  - It also comes with optional JavaScript plugins to add interactivity to components.
- Bootstrap is promoted as being **One framework, every device**.
  - This is because websites built with Bootstrap will automatically scale between devices — whether the device is a mobile phone, tablet, laptop, desktop computer, screen reader, etc.
- Responsive web design is about creating web sites which automatically adjust themselves to look good on all devices, from small phones to large desktops.
  - Developers can then create a single design that works on any kind of device: mobiles, tablets, smart TVs, and PCs

Where to Get Bootstrap 4?

- There are two ways to start using Bootstrap 4 on your own web site.
  - Download Bootstrap 4 from getbootstrap.com : https://getbootstrap.com/docs/4.5/getting-started/download/
  - If you don't want to download and host Bootstrap 4 yourself, you can include it from a CDN (Content Delivery Network).
  - MaxCDN provides CDN support for Bootstrap's CSS and JavaScript. You must also include jQuery
  - The https://getbootstrap.com/docs/4.5/getting-started/introduction/ page gives CDN links for CSS and js files

```
<!-- Latest compiled and minified CSS -->
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">

<!-- jQuery library -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

<!-- Popper JS -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>

<!-- Latest compiled JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
```


Create First Web Page With Bootstrap 4

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap 4 Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js">
  </script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js">
  </script>
</head>
<body>
   //body comes here
</body>
</html>
```


To ensure proper rendering and touch zooming, add this <meta> tag

- The width=device-width part sets the width of the page to follow the screen-width of the device (which will vary depending on the device).
- The initial-scale=1 part sets the initial zoom level when the page is first loaded by the browser.

Create First Web Page With Bootstrap 4

- Bootstrap 4 also requires a containing element to wrap site contents.
- There are two container classes to choose from:
  - The .container class provides a responsive fixed width container
  - The .container-fluid class provides a full width container, spanning the entire width of the viewport

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Bootstrap 4 Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/boot
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/j
  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.1
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/b
</head>
<body>

<div class="container-fluid">
  <h1>My First Bootstrap Page</h1>
  <p>This is some text.</p>
</div>

</body>
</html>
```

Bootstrap Container

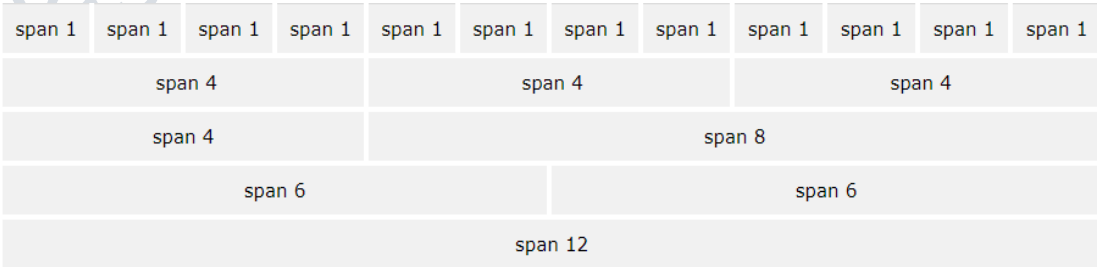
- Bootstrap container is basically used in order to create a centered area that lies within the page and generally deals with the margin of the content and the behaviors that are responsible for the layout.
  - It contains the grid system (row elements, which in turn are the container of columns).
- There are two container classes in Bootstrap:
  - .container: provides a fixed width container with responsiveness. It will not take the complete width of its viewport.
  - .container-fluid: provides a full width container of the viewport and its width will change (expand or shrink) on different screen sizes.

```
<body>
   <div class="container">
      <h1>Container</h1>
   </div>
</body>
```

Bootstrap Grid System

- Bootstrap grid system divides the screen into columns—up to 12 in each row. (rows are infinite)
  - The column widths vary according to the size of screen they're displayed in.
  - Bootstrap's grid system is responsive, as the columns resize themselves dynamically when the size of browser window changes.
  - If you do not want to use all 12 columns individually, you can group the columns together to create wider columns
  - it is a good practice to wrap all the contents within a container; create a row (with class row) inside a container, then start creating the columns.

```
<div class="container">
    <div class="row">
      //add desired number of cols here
    </div>
</div>
```

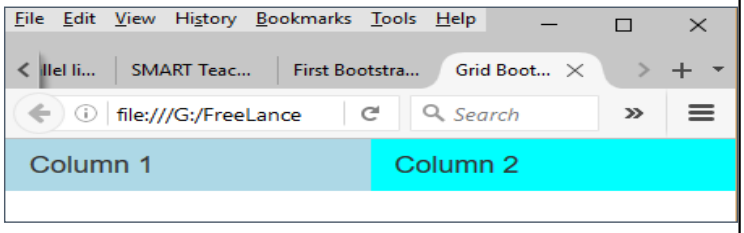


Grid Classes

- The Bootstrap 4 grid system has five classes:
  - .col- (extra small devices - screen width less than 576px)
  - .col-sm- (small devices - screen width equal to or greater than 576px)
  - .col-md- (medium devices - screen width equal to or greater than 768px)
  - .col-lg- (large devices - screen width equal to or greater than 992px)
  - .col-xl- (xlarge devices - screen width equal to or greater than 1200px)

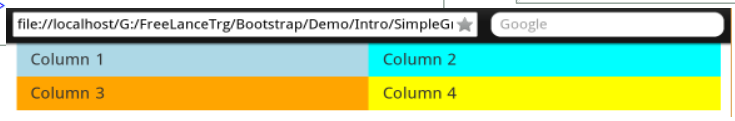

```
  <style>
    .mycol1{ background: lightblue;}
    .mycol2{ background: cyan;}
  </style>
</head>

    <body>
        <div class="container-fluid">
            <div class="row">
                <div class="col mycol1">
                    <h4>Column 1</h4>
                </div>
                <div class="col mycol2">
                    <h4>Column 2</h4>
                </div>
            </div>
        </div>

    </body>
```

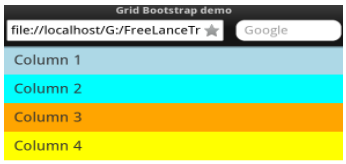
Example



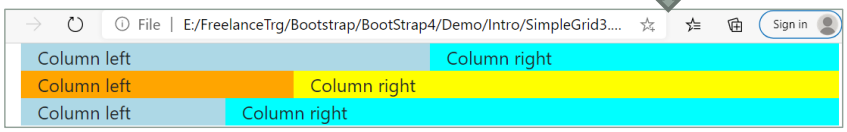
Building a Basic Grid

```
  <style>
        .col1{ background: lightblue;}
        .col2{ background: cyan;}
        .col3{ background: orange;}
        .col4{ background: yellow;}
        </style>
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-12 col-sm-6 col1">
        <h4>Column 1</h4>
      </div>
      <div class="col-12 col-sm-6 col2">
        <h4>Column 2</h4>
      </div>
      <div class="col-12 col-sm-6 col3">
        <h4>Column 3</h4>
      </div>
      <div class="col-12 col-sm-6 col4">
        <h4>Column 4</h4>
      </div>
    </div>
  </div>
```






```
  <style>
        .col1{ background: lightblue;}
        .col2{ background: cyan;}
        .col3{ background: orange;}
        .col4{ background: yellow;}
        </style>
    </head>
<body>
<div class="container">
    <!--Row with two equal columns-->
    <div class="row">
        <div class="col-md-6 col1">Column left</div>
        <div class="col-md-6 col2">Column right</div>
    </div>

    <!--Row with two columns divided in 1:2 ratio-->
    <div class="row">
        <div class="col-md-4 col3">Column left</div>
        <div class="col-md-8 col4">Column right</div>
    </div>

    <!--Row with two columns divided in 1:3 ratio-->
    <div class="row">
        <div class="col-md-3 col1">Column left</div>
        <div class="col-md-9 col2">Column right</div>
    </div>
</div>
```



MISC COMPONENTS

<h1> - <h6>

- Typography refers to the various styles present in Bootstrap style sheets which define how various text elements will appear on the web page.
  - HTML uses default font and style to create headings, paragraphs, lists and other inline elements.
  - Bootstrap overrides default and provides consistent styling across browsers for common typographic elements.
- Bootstrap 4 styles HTML headings (<h1> to <h6>) with a bolder font-weight and an increased font-size

```
<div class="container">
    <h1>h1 Bootstrap heading (2.5rem = 40px)</h1>
    <h2>h2 Bootstrap heading (2rem = 32px)</h2>
    <h3>h3 Bootstrap heading (1.75rem = 28px)</h3>
    <h4>h4 Bootstrap heading (1.5rem = 24px)</h4>
    <h5>h5 Bootstrap heading (1.25rem = 20px)</h5>
    <h6>h6 Bootstrap heading (1rem = 16px)</h6>
</div>
```

h1 Bootstrap heading (2.5rem = 40px)
h2 Bootstrap heading (2rem = 32px)
h3 Bootstrap heading (1.75rem = 28px)
h4 Bootstrap heading (1.5rem = 24px)
h5 Bootstrap heading (1.25rem = 20px)
h6 Bootstrap heading (1rem = 16px)

- Additionally, you can use the <small> tag with .text-muted class to display the secondary text of any heading in a smaller and lighter variation.

```
<h2>Fancy display heading
  <small class="text-muted">faded secondary text</small>
</h2>
```

Fancy display heading faded secondary text

Working with Paragraphs

- Bootstrap's global default font-size is 1rem (typically 16px), with a line-height of 1.5. This is applied to the <body> and all paragraphs
  - You can also make a paragraph stand out by adding the class .lead on it.
  - You can also transform the text to lowercase, uppercase or make them capitalize.

```
<div class="container">
  <p>This is how a normal paragraph looks like in Bootstrap.</p>
  <p class="lead">This is how a paragraph stands out in Bootstrap</p>
  <p class="text-left">Left aligned text.</p>
  <p class="text-center">Center aligned text.</p>
  <p class="text-right">Right aligned text.</p>
  <p class="text-lowercase">Text in lowercase</p>
  <p class="text-uppercase">Text in uppercase</p>
  <p class="text-capitalize">Text in capitalize</p>
```

This is how a normal paragraph looks like in Bootstrap.
This is how a paragraph stands out in Bootstrap.
Left aligned text.
Center aligned text.
Right aligned text.
text in lowercase
TEXT IN UPPERCASE
Text In Capitalize

- Text Coloring
  - Colors are the powerful method of conveying important information in website design.

```
<p class="text-muted">This text is muted.</p>
<p class="text-primary">This text is important.</p>
<p class="text-success">This text indicates success.</p>
<p class="text-info">This text represents some information.</p>
<p class="text-warning">This text represents a warning.</p>
<p class="text-danger">This text represents danger.</p>
<p>This content has <em>emphasis</em>, and can be <strong>bold

<p class="bg-primary">This text is important.</p>
<p class="bg-success">This text indicates success.</p>
<p class="bg-info">This text represents some information.</p>
<p class="bg-warning">This text represents a warning.</p>
<p class="bg-danger">This text represents danger.</p>
```

This text is muted.
This text is important.
This text indicates success.
This text represents some inform
This text represents a warning.
This text represents danger.
This content has *emphasis*, and can be **bold**

This text is important.
This text indicates success.
This text represents some information.
This text represents a warning.
This text represents danger.

Tables

- Bootstrap provides an efficient layout to build elegant tables
  - You can create tables with basic styling that has horizontal dividers and small cell padding, by just adding the Bootstrap's class .table to the <table> element.

```
<table class="table">
    <tr><th>Name</th><th>Age</th></tr>
    <tr><td>Kavita</td><td>23</td></tr>
    <tr><td>Anita</td><td>33</td></tr>
</table>
```

Name	Age
Kavita	23
Anita	33

  - The .table-striped class adds zebra-stripes to a table
  - The .table-bordered class adds borders on all sides of the table and cells
  - The .table-condensed class makes a table more compact by cutting cell padding in half
  - The .table-dark class create inverted version of this table, i.e. table with light text on dark backgrounds

<table class="table table-dark">

Name	Age
Kavita	23
Anita	33

```
<table class = "table table-striped table-bordered table-condensed table-sm">
    <caption>Basic Table Layout</caption>
    <thead class="thead-light">
        <tr><th>Name</th><th>City</th></tr>
    </thead>
    <tbody>
        <tr><td>Soha</td><td>Bangalore</td></tr>
        <tr><td>Shrilata</td><td>Pune</td></tr>
        <tr><td>Sandeep</td><td>Mumbai</td></tr>
        <tr class="table-success">
            <td>Sheela</td>
            <td>Delhi</td>
        </tr>
    </tbody>
</table>
```

Name	City
Soha	Bangalore
Shrilata	Pune
Sandeep	Mumbai
Sheela	Delhi

Jumbotron

- A jumbotron indicates a big box for calling extra attention to some special content or information.
  - A jumbotron is displayed as a grey box with rounded corners. It also enlarges the font sizes of the text inside it.
  - Just wrap your featured content like heading, descriptions etc. in a element and apply the class .jumbotron on it.
  - Tip: Inside a jumbotron you can put nearly any valid HTML, including other Bootstrap elements/classes.

```
<div class="jumbotron">
   <h1>Bootstrap Tutorial</h1>
   <p>Bootstrap is the most popular HTML, CSS, and JS framework for
       developing responsive, mobile-first projects on the web.</p>
 </div>
```

images

- To add images on the webpage use element <img> , it has **three** classes to apply simple styles to images.
  - .img-rounded : To add rounded corners around the edges of the image, radius of the border is **6px**.
  - .img-circle : To create a circle of radius is **500px**
  - .img-thumbnail : To add some padding with grey border , making the image look like a polaroid photo.

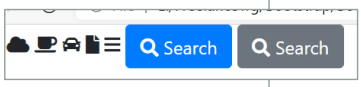
```
<img src="taj.jpg" class="img-rounded"> <!-- rounded  edges-->
<img src="taj.jpg." class="img-circle"> <!-- circle -->
<img src="taj.jpg" class="img-thumbnail"> <!-- thumbnail -->
```

Bootstrap 4 Icons

- Bootstrap 4 does not have its own icon library; but there are many free icon libraries to choose from, such as Font Awesome and Google Material Design Icons
  - To use Font Awesome icons, add the following CDN link to your HTML page
  - <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.6.3/css/all.css" integrity="sha384-UHRtZLI+pbxtHCWp1t77Bi1L4ZtiqrqD80Kn4Z8NTSRyMA2Fd33n5dQ8lWUE00s/" crossorigin="anonymous">

```
<i class="fas fa-cloud"></i>
<i class="fas fa-coffee"></i>
<i class="fas fa-car"></i>
<i class="fas fa-file"></i>
<i class="fas fa-bars"></i>
```

```
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css"
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.6.3/css/all.css" integrity="sh
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/popper.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>
</head>
<body>
    <div class="container">
      <i class="fas fa-cloud"></i>
      <i class="fas fa-coffee"></i>
      <i class="fas fa-car"></i>
      <i class="fas fa-file"></i>
      <i class="fas fa-bars"></i>

    <button type="submit" class="btn btn-primary"><span class="fa fa-search"></span> Search</button>
    <button type="submit" class="btn btn-secondary"><span class="fa fa-search"></span> Search</button>
```



Alerts

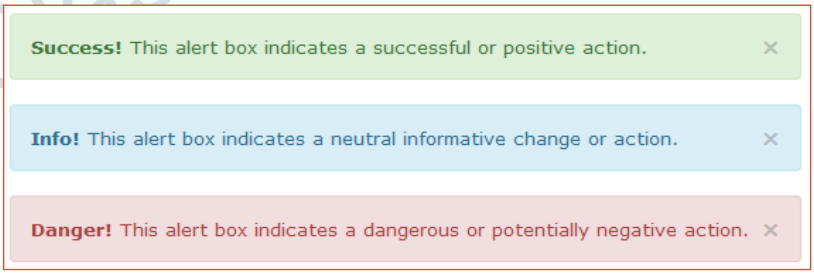
- Bootstrap comes with a very useful component for displaying alert messages in various sections of our website
  - You can use them for displaying a success message, a warning message, a failure message, or an information message.
  - These messages can be annoying to visitors, hence they should have dismiss functionality added to give visitors the ability to hide them.

```
<div class="alert alert-success">
    Amount has been transferred successfully.
</div>
```

contextual classes for alert messages: alert-success, alert-info, alert-danger, alert-warning

```
<div class="alert alert-success alert-dismissable">
    <button type="button" class="close" data-dismiss="alert">&times;</button>
        Amount has been transferred successfully.
</div>
```

dismissible alert box



Bootstrap Lists

- Unstyled Ordered and Unordered Lists
  - Sometimes you might need to remove the default styling form the list items. You can do this by simply applying the class .list-unstyled to the respective <ul> or <ol> elements

```
<!-- normal HTML List -->
<ul>
  <li>Home</li>
  <li>Products
    <ul>
      <li>Gadgets</li>
      <li>Accessories</li>
    </ul>
  </li>
  <li>About Us</li>
  <li>Contact</li>
</ul>
```

- Home
- Products
  - Gadgets
  - Accessories
- About Us
- Contact

```
<ul class="list-unstyled">
  <li>Home</li>
  <li>Products
    <ul>
      <li>Gadgets</li>
      <li>Accessories</li>
    </ul>
  </li>
  <li>About Us</li>
  <li>Contact</li>
</ul>
```

Home
Products
- Gadgets
- Accessories

About Us
Contact

- If you want to create a horizontal menu using ordered or unordered list you need to place all list items in a single line i.e. side by side.
  - You can do this by simply applying the class .list-inline to the respective <ul> or <ol>, and the class .list-inline-item to the <li> elements.

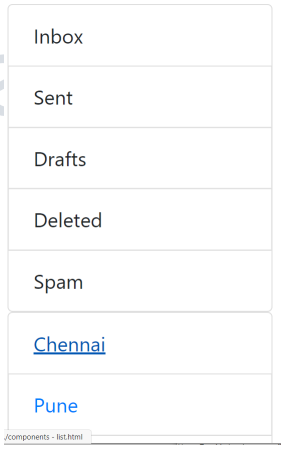
```
<!-- inline List -->
<ul class="list-inline">
  <li class="list-inline-item">Home</li>
  <li class="list-inline-item">Products</li>
  <li class="list-inline-item">About Us</li>
  <li class="list-inline-item">Contact</li>
</ul>
```

Home Products About Us Contact

Page Components : List Group

- List group is used for creating lists; eg a list of useful resources or a list of recent activities
  - Add class list-group to a <ul> or element to make its children appear as a list.
  - The children can be li or a element, depending on your parent element choice.
  - The child should always have the class list-group-item.

```
<!-- List group -->
<ul class="list-group">
    <li class="list-group-item">Inbox</li>
    <li class="list-group-item">Sent</li>
    <li class="list-group-item">Drafts</li>
    <li class="list-group-item">Deleted</li>
    <li class="list-group-item">Spam</li>
</ul>
<div class="list-group">
    <a href="#" class="list-group-item">Chennai</a>
    <a href="#" class="list-group-item">Pune</a>
    <a href="#" class="list-group-item">Mumbai</a>
</div>
```



Page Components : List Group

- We can display a number beside each list item using the badge component.
  - Add this inside each “list-group-item” to display badge; badges align to the right of each list item

```
<div class="list-group">
    <a href="#" class="list-group-item list-group-item-success">
        <i class="fa fa-home"></i> Home
        <span class="badge">14</span> </a>
    <a href="#" class="list-group-item">
        <i class="fa fa-camera"></i> Pictures
        <span class="badge badge-pill badge-primary pull-right">145</span></a>
    <a href="#" class="list-group-item">
        <i class="fa fa-music"></i> Music
        <span class="badge badge-pill badge-primary pull-right">50</span></a>
```

- We can also apply various colors to each list item by adding list-group-item-* classes along with list-group-item.

```
<ul class="list-group">
    <li class="list-group-item list-group-item-success">Success item</li>
    <li class="list-group-item list-group-item-secondary">Secondary item</li>
    <li class="list-group-item list-group-item-info">Info item</li>
    <li class="list-group-item list-group-item-warning">Warning item</li>
    <li class="list-group-item list-group-item-danger">Danger item</li>
    <li class="list-group-item list-group-item-primary">Primary item</li>
    <li class="list-group-item list-group-item-dark">Dark item</li>
    <li class="list-group-item list-group-item-light">Light item</li>
</ul>
```

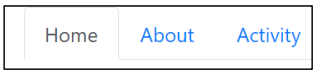
Bootstrap 4 Navs

- Navs : a group of links placed inline with each other to be used for navigation.
  - There are options to make this group of links appear either as tabs or small buttons, the latter known as pills in Bootstrap.
  - If you want to create a simple horizontal menu, add the .nav class to a <ul> element, followed by .nav-item for each <li> and add the .nav-link class to their links:

```
<nav class="nav">
  <a href="#" class="nav-item nav-link active">Home</a>
  <a href="#" class="nav-item nav-link">About</a>
  <a href="#" class="nav-item nav-link">Activity</a>
</nav>
```

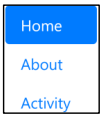
  - Add the class .nav-tabs to the basic nav to generate a tabbed navigation.
  - Similarly, you can create pill based navigation by adding the class .nav-pills on the basic nav instead of class .nav-tabs
  - Vertically stack these pills by attaching an additional class `flex-column`

```
<nav class="nav nav-tabs">
```




```
<nav class="nav nav-pills">
```


```
<nav class="nav nav-pills flex-column">
```

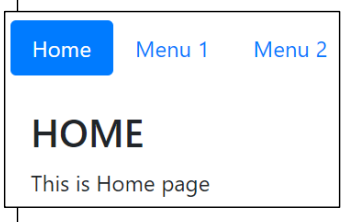


Toggleable / Dynamic Pills

- To make the tabs toggleable, add the data-toggle="tab" attribute to each link.
  - Then add a .tab-pane class with a unique ID for every tab and wrap them inside a element with class .tab-content.

```
<ul class="nav nav-pills" role="tablist">
  <li class="nav-item">
    <a class="nav-link active" data-toggle="pill" href="#home">Home</
  </li>
  <li class="nav-item">
    <a class="nav-link" data-toggle="pill" href="#menu1">Menu 1</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" data-toggle="pill" href="#menu2">Menu 2</a>
  </li>
</ul>
  <!-- Tab panes -->
  <div class="tab-content">
    <div id="home" class="container tab-pane active"><br>
      <h3>HOME</h3><p>This is Home page</p>
    </div>
    <div id="menu1" class="container tab-pane fade"><br>
      <h3>Menu 1</h3><p>This is Menu1</p>
    </div>
    <div id="menu2" class="container tab-pane fade"><br>
      <h3>Menu 2</h3><p>This is menu2</p>
    </div>
  </div>
```

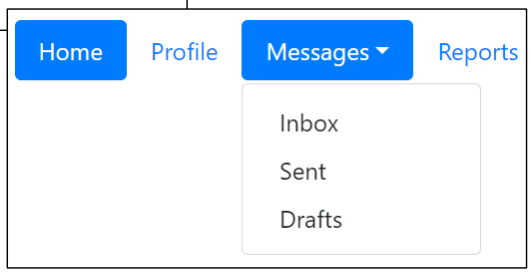
- To fade the tabs in and out when clicking on them, add the .fade class to .tab-pane



Navs with dropdown

- You can add dropdown menus to a link inside tabs and pills nav with a little extra markup.
  - The four CSS classes .dropdown, .dropdown-toggle, .dropdown-menu and .dropdown-item are required

```
<nav class="nav nav-pills">
  <a href="#" class="nav-item nav-link active">Home</a>
  <a href="#" class="nav-item nav-link">Profile</a>
  <div class="nav-item dropdown">
      <a href="#" class="nav-link dropdown-toggle"
                  data-toggle="dropdown">Messages</a>
      <div class="dropdown-menu">
          <a href="#" class="dropdown-item">Inbox</a>
          <a href="#" class="dropdown-item">Sent</a>
          <a href="#" class="dropdown-item">Drafts</a>
      </div>
  </div>
  <a href="#" class="nav-item nav-link">Reports</a>
</nav>
```



Navbar

- A navbar is a navigation header that is placed at the top of the page
  - A standard navigation bar is created with the .navbar class, followed by a responsive collapsing class: .navbar-expand-xl|lg|md|sm
  - To add links inside the navbar, use a <ul> element with class="navbar-nav".
  - Then add <li> elements with a .nav-item class followed by an <a> element with a .nav-link class

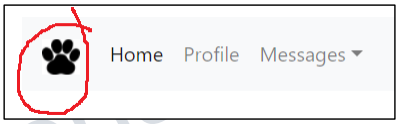
```
<!-- A grey horizontal navbar that becomes vertical on small screens -->
<nav class="navbar navbar-expand-sm bg-light">
<!-- Links -->
    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link" href="#">Link 1</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="#">Link 2</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="#">Link 3</a>
        </li>
    </ul>
</nav>
```

Remove .navbar-expand-xl|lg|md|sm class to create a vertical nav bar

Navbar

- Brand / Logo
  - The .navbar-brand class is used to highlight the brand/logo/project name of your page

```
<a href="#" class="navbar-brand">
   <img src="paws.png" height="28" alt="CoolBrand">
</a>
```



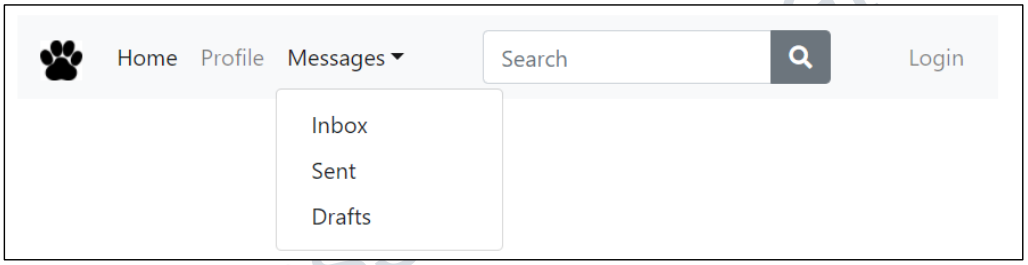
- Collapsing The Navigation Bar
  - Very often, especially on small screens, you want to hide the navigation links and replace them with a button that should reveal them when clicked on.
  - To create a collapsible navigation bar, use a button with class="navbar-toggler", data-toggle="collapse" and data-target="#thetarget".
  - Then wrap the navbar content (links, etc) inside a div element with class="collapse navbar-collapse", followed by an id that matches the data-target of the button: "thetarget".

```
<!-- Toggler/collapsibe Button -->
 <button class="navbar-toggler" type="button" data-toggle="collapse"
         data-target="#collapsibleNavbar">
  <span class="navbar-toggler-icon"></span>
 </button>
 <!-- Navbar links -->
 <div class="collapse navbar-collapse" id="collapsibleNavbar">
```

Navbar

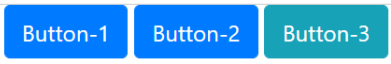
```
<nav class="navbar navbar-expand-md navbar-light bg-light">
<a href="#" class="navbar-brand">
   <img src="paws.png" height="28" alt="CoolBrand">
</a>
<button type="button" class="navbar-toggler" data-toggle="collapse"
                      data-target="#nb1">
    <span class="navbar-toggler-icon"></span>
</button>  <!-- when navbar collapses on small dev-->

<div class="collapse navbar-collapse justify-content-between" id="nb1">
    <div class="navbar-nav">
        <a href="#" class="nav-item nav-link active">Home</a>
        <a href="#" class="nav-item nav-link">Profile</a>
        <div class="nav-item dropdown">
            <a href="#" class="nav-link dropdown-toggle"
                        data-toggle="dropdown">Messages</a>
            <div class="dropdown-menu">
                <a href="#" class="dropdown-item">Inbox</a>
                <a href="#" class="dropdown-item">Sent</a>
                <a href="#" class="dropdown-item">Drafts</a>
            </div>
        </div>
    </div>
    <form class="form-inline">
    <div class="navbar-nav">
        <a href="#" class="nav-item nav-link">Login</a>
    </div>
```

```
<form class="form-inline">
    <div class="input-group">
        <input type="text" class="form-control" placeholder="Search">
        <div class="input-group-append">
            <button type="button" class="btn btn-secondary">
                <i class="fa fa-search"></i>
            </button>
        </div>
    </div>
</form>
```



Standing Out : Buttons

- Its easy to convert an a, button, or input element into a fancy bold button in Bootstrap; just have to add the btn class

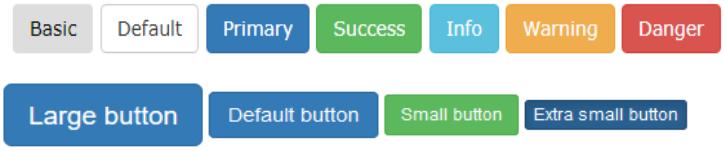



```
<a href="#" class="btn btn-primary">Button-1</a>
<button type="button" class="btn btn-primary">Button-2</button>
<input type="button" class="btn btn-info" value="Button-3">
```

  - You can also create outline buttons by replacing the button modifier classes


```
<button type="button" class="btn btn-outline-primary">Primary</button>
<button type="button" class="btn btn-outline-warning">Warning</button>
```

  - Buttons come in various color options:
    - btn-default for white
    - btn-primary for dark blue
    - btn-success for green
    - btn-info for light blue
    - btn-warning for orange
    - btn-danger for red
  - And in various sizes:
    - btn-lg for large buttons
    - btn-sm for small buttons
    - btn-xs for extra small button

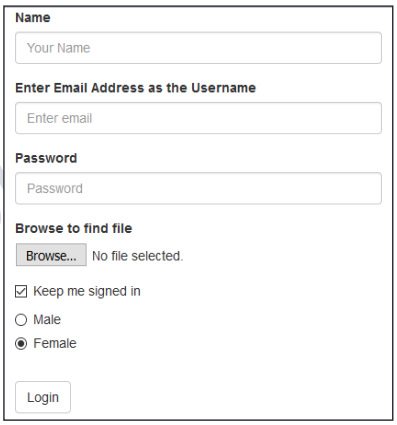



```
<button type="button" class="btn btn-primary btn-lg">Large button</button>
<button type="button" class="btn btn-primary">Default button</button>
<button type="button" class="btn btn-primary btn-sm btn-success">Small button</button>
```

Creating Forms

- Bootstrap provides three different types of form layouts:
  - Vertical Form (default form layout)
  - Horizontal Form
  - Inline Form

Eg ->



- Standard rules for all three form layouts:
  - Wrap labels & form controls in (for optimum spacing)
  - Add class .form-control to all textual <input>, <textarea>, and <select> elements

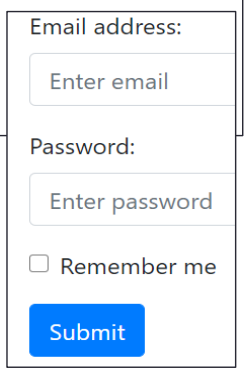
```
<form class="form">
    <div class="form-group">
        <label for="n1">Name</label>
        <input type="text" class="form-control" id="n1"  placeholder="Your Name" />
    </div>
</form>
```

class form-control in an input element will make it a full-width element

Creating Forms : Vertical Form Layout

- This is the default Bootstrap form layout in which styles are applied to form controls without adding any base class to the <form> element or any large changes in the markup.
- The form controls in this layout are stacked with left-aligned labels on the top.

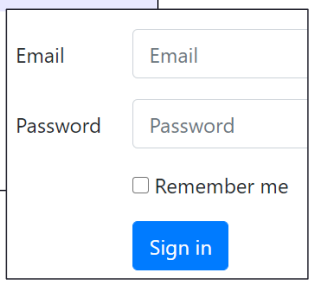
```
<form action="#">
    <div class="form-group">
        <label for="email">Email address:</label>
        <input type="email" class="form-control" placeholder="Enter email" id="email">
    </div>
    <div class="form-group">
        <label for="pwd">Password:</label>
        <input type="password" class="form-control" placeholder="Enter password" id="pwd">
    </div>
    <div class="form-group form-check">
        <label class="form-check-label">
            <input class="form-check-input" type="checkbox"> Remember me
        </label>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
```



Creating Forms : Horizontal Form Layout

- Labels and form controls are aligned side-by-side using the Bootstrap grid classes.
- To create this layout add the class .row on form groups and use the .col-*-* grid classes to specify the width of your labels and controls.

```
<form action="#">
    <div class="form-group row">
        <label for="mail" class="col-sm-2 col-form-label">Email</label>
        <div class="col-sm-10">
            <input type="email" class="form-control" id="mail" placeholder="Email">
        </div>
    </div>
    <div class="form-group row">
        <label for="pass" class="col-sm-2 col-form-label">Password</label>
        <div class="col-sm-10">
            <input type="password" class="form-control" id="pass" placeholder="Password">
        </div>
    </div>
    <div class="form-group row">
        <div class="col-sm-10 offset-sm-2">
            <label class="form-check-label"><input type="checkbox"> Remember me</label>
        </div>
    </div>
    <div class="form-group row">
        <div class="col-sm-10 offset-sm-2">
            <button type="submit" class="btn btn-primary">Sign in</button>
        </div>
    </div>
</form>
```



Creating Forms : Inline Form Layout

- Additional rule for an inline form: Add class .form-inline to the <form> element

```
<form class="form-inline">
    <div class="form-group mr-2">
        <label class="sr-only" for="inputEmail">Email</label>
        <input type="email" class="form-control"
               id="inputEmail" placeholder="Email">
    </div>
    <div class="form-group mr-2">
        <label class="sr-only" for="inputPassword">Password</label>
        <input type="password" class="form-control"
               id="inputPassword" placeholder="Password">
    </div>
    <div class="form-group mr-2">
        <label><input type="checkbox" class="mr-1"> Remember me</label>
    </div>
    <button type="submit" class="btn btn-primary">Sign in</button>
</form>
```

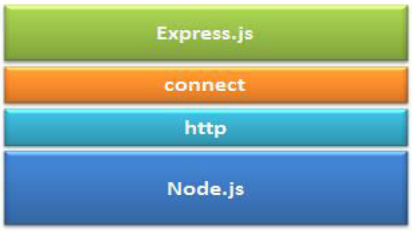
EXPRESS.JS

Node.js web application framework

Introduction

- If you write serious apps using only core Node.js modules you most likely find yourself reinventing the wheel by writing the same code continually for similar tasks, such as the following:
  - Parsing of HTTP request bodies and Parsing of cookies
  - Managing sessions
  - Organizing routes with a chain of if conditions based on URL paths and HTTP methods of the requests
  - Determining proper response headers based on data types

Express 4.17.1
Fast, unopinionated, minimalist web framework for Node.js



- Express is a web application framework for Node
  - Its built on top of Node.js.
  - It provides various features that make web application development fast and easy compared to Node.js.

Express.js Installation

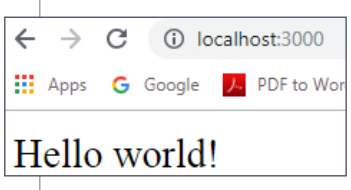
- Create a new folder: eg E:\Express-app
- E:\Express-app>npm init //for creating package.json
- E:\Express-app>npm install express --save

```
{
  "name": "express-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

First app

- An Express app is created by calling the express() function
  - express() : Creates an Express application; is a top-level function exported by the express module
  - The app object conventionally denotes the Express application.
  - This object, which is traditionally named app, has methods for routing HTTP requests, configuring middleware, rendering HTML views, registering a template engine, and modifying application settings that control how the application behaves

```
let express = require("express")  // import the express module

let app = express()  // create an Express application

app.get("/",function(req,resp){
    resp.send("Hello world")
})

app.listen(3000, function(){
    console.log("app running on port 3000")
})
```



The *app.listen()* method returns an http.Server object

Express Routes

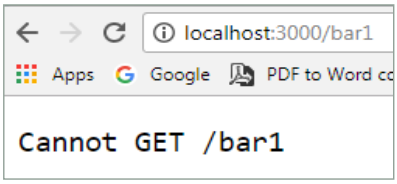
- HTTP verb and URL combinations are referred to as routes, and Express has efficient syntax for handling them.

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

app.get("/", function(req, res, next) {
      res.send("Hello <strong>home page</strong>");
});

app.get("/foo", function(req, res, next) {
      res.send("Hello <strong>foo</strong>");
});

app.get("/bar", function(req, res, next) {
     res.send("Hello <strong>bar</strong>");
});
//app is passed to the http.createServer() method

http.createServer(app).listen(8000);
```

Open browser and type:
http://localhost:3000
http://localhost:3000/foo
http://localhost:3000/bar
http://localhost:3000/admin - gives error

Express Routes

- If none of your routes match the request, you'll get a ***"Cannot GET <your-request-route>"*** message as response.



- This message can be replaced by a 404 not found page using this simple route

```
app.get('*', function(req, res){
      res.send('Sorry, this is an invalid URL.');
});
```

```
app.get('/course/:id', function(req, res){
      var course = //code to retrieve course
      If(!course){
           res.status(404).send("course not found");
});
```

Routing basics

```
app.get("/", function(req, res, next) {
      res.send("Hello <strong>home page</strong>");
});
```

- The get() method defines routes for handling GET requests.
- Express also defines similar methods for the other HTTP verbs (put(), post(), delete(), and so on).
  - app.method(path, handler) : This METHOD can be applied to any one of the HTTP verbs – get, post, put, delete.
  - All methods take a URL path and a sequence of middleware as arguments.
  - The path is a string or regular expression representing the URL that the route responds to. Note that the query string is not considered part of the route's URL.
- Also notice that we haven't defined a 404 route, as this is the default behavior of Express when a request does not match any defined routes.

Routing basics

```
app.get("/", function(req, res, next) {
      res.send("Hello <strong>home page</strong>");
});
```

- Express also augments the request and response objects with additional methods. Example response.send() .
  - send() is used to send a response status code and/or body back to the client.
  - If the first argument to send() is a number, then it is treated as the status code. If a status code is not provided, Express will send back a 200.
  - The response body can be specified in the first or second argument, and can be a string, Buffer, array, or object.
- send() also sets the Content-Type header unless you do so explicitly.
  - If the body is a string, Express will set the Content-Type header to text/html.
  - If the body is an array or object, then Express will send back JSON.
  - If the response body is a Buffer, the Content-Type header is also set to application/octet-stream

Routing basics

```
var express = require("express");
var app = express();
var path = require('path');

app.get("/buff", function(req, res, next) {
    var buff = Buffer.from("Hello World");
    res.send(buff.toString());
});
app.get("/string", function(req, res, next) {
    res.send("Hello <strong>String response</strong>");
});
app.get("/json", function(req, res, next) {
     res.send({name:'Soha',age:23});
});
app.get("/array", function(req, res, next) {
     res.send(['NodeJS','Angular','ExpressJS']);
});
app.get("/file", function(req, res, next) {
    res.sendFile(path.join(__dirname + '/summer.html'));
});
app.listen(3000);
```

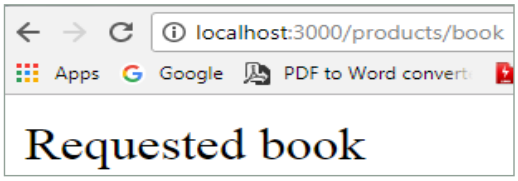
Route Parameters

- Route can be parameterized using a regular expression

```
var express = require("express");
var http = require("http");
var app = express();
app.get(/\/products\/([^\/]+)\/?$/, function(req, res, next) {
    res.send("Requested " + req.params[0]);
});
http.createServer(app).listen(8000);
```

Ignore multiple slash

/products?productId=sweater
/products/sweater



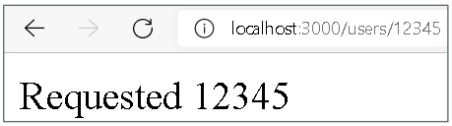
The above regular expression matches anything but /
/products/books
/products/books:aaa
/products/books/
/products/books?aaa
/products/books=aaa

Doesn't match:
/products/books/aaa
/products/books//
/products//

Eg 2 : Using regular expressions to match routes

- Assume you want to match things like /users/123 or /users/456 but not /users/anita. You can code this into a regular expression and also grab the number

```
app.get(/^\/users\/(\d+)$/, function(req, res) {
    var userId = parseInt(req.params[0], 10);
    res.send("Requested " + userId);
});
```



- **req.params -** An object containing parameter values parsed from the URL path.
  - For example, if you have the route /user/:name, then the “name” property is available as req.params.name. This object defaults to {}.
  - GET /user/shrilata
  - req.params.name // => “shrilata“
  - When you use a regular expression for the route definition, each capture group match from the regex is available as req.params[0], req.params[1]
  - GET /file/javascripts/jquery.js
  - req.params[0] // => "javascripts/jquery.js"

Route Parameters

- One of the most powerful features of routing is the ability to use placeholders to extract named values from the requested route, marked by the colon (:) character.
  - When the route is parsed, express puts the matched placeholders on the req.params object for you.

```
app.get('/user/:id', function(req, res) {
    res.send('user ' + req.params.id);
});

app.get(‘/product/:prodname', function(req, res) {
    res.send(‘Product :  ' + req.params.prodname);
});
```

Placeholders match any sequence of characters except for forward slashes.

Route Parameters

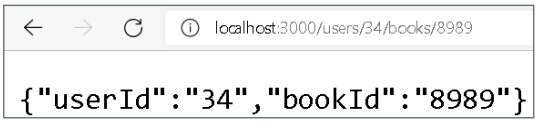
- Below example, creates a named parameter productId.

```
app.get("/product/:productId(\\d+)", function(req, res, next) {
    res.send("Requested " + req.params.productId );
});
```

productId can now only be made up of digits

Invoke as : http://localhost:3000/product/123

```
app.get('/users/:userId/books/:bookId', function (req, res) {
    res.send(req.params)
    //res.send(req.params.userId + ":" + req.params.bookId)
})
  //Route path: /users/:userId/books/:bookId
  //Request URL: http://localhost:3000/users/34/books/8989
  //req.params: { "userId": "34", "bookId": "8989" }
```

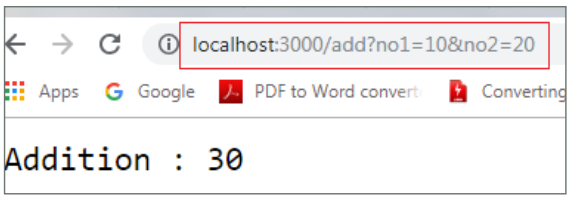


Working with parameters using get

- **req.query**
  - Express parses query string parameters by default, and puts them into the req.query property.
  - If the request is GET /search?username=Shrilata
    req.query.username returns “Shrilata"

  - Lets say the incoming url is : http://localhost:3000/add?no1=10&no2=20
  - Use req.query to query the request parameters

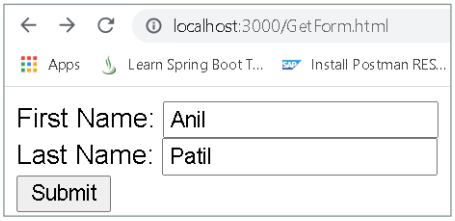
```
app.get("/add", function (req, res) {
    n1 = parseInt(req.query.no1);
    n2 = parseInt(req.query.no2);
    res.end("Addition : " +   (n1 + n2) );
});
```

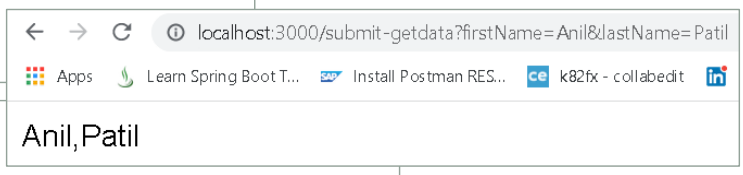


Handle GET Request

GetForm.html > ...

```
<!DOCTYPE html>
<html>
<body>
    <form action="/submit-getdata" method="get">
        First Name: <input name="firstName" type="text" /
        Last Name: <input name="lastName" type="text" />
        <input type="submit" />
    </form>
</body>
</html>
```






```
var express = require('express');
var app = express();

app.get('/GetForm.html', function (req, res) {
    res.sendFile('public/GetForm.html' , { root : __dirname});
});

app.get('/submit-getdata', function (req, res) {
    console.log(req.query.firstName);
    console.log(req.query.lastName);
    res.send(req.query.firstName+","+req.query.lastName);
});
app.listen(3000);
```

Multiple different methods

- We can also have multiple different methods at the same route.

```
var express = require('express');
var app = express();

app.get('/hello', function(req, res){
   res.send("Hello World!");
});

app.post('/hello', function(req, res){
   res.send("You just called the post method at '/hello'!\n");
});

app.delete('/hello', function(req, res){
   res.send("You just called the delete method at '/hello'!\n");
});

app.listen(3000);
```

Handle POST Request

- To handle HTTP POST request in Express.js version 4 and above, you need to install a middleware module called body-parser.
  - This is used to parse the body of requests which have payloads attached to them.
  - Install it using : **npm install** --save body-parser
  - Mount it by including the following lines in your js

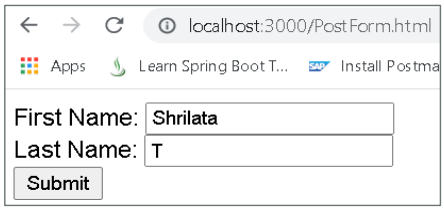
```
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
```

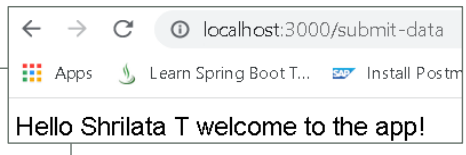
  - This body-parser module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request.
  - Eg :To parse json data: app.use(bodyParser.json())

bodyParser.urlencoded(): Parses the text as URL encoded data (which is how browsers tend to send form data from regular forms set to POST) and exposes the resulting object (containing the keys and values) on **req.body**

Handle POST Request

```
<!DOCTYPE html>
<html>  <!-- PostForm.html -->
<body>
    <form action="/submit-data" method="post">
        First Name: <input name="firstName" type="text" />
        Last Name: <input name="lastName" type="text" />
        <input type="submit" />
    </form>
</body>
</html>
```






```
var express = require('express');
var app = express();
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));

app.get('/PostForm.html', function (req, res) {
    //res.sendFile('E:\\Node.js\\Demo\\expressPrj\\PostForm.html');
    res.sendFile('public/PostForm.html' , { root : __dirname});
});

app.post('/submit-data', function (req, res) {
    var name = req.body.firstName + ' ' + req.body.lastName;
    res.send("Hello " + name + ' welcome to the app!');
});
app.listen(3000);
```

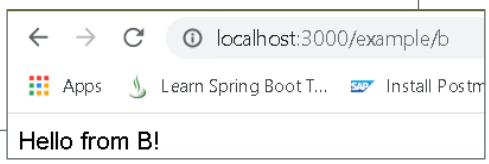
Routing handlers

- app.get(path, callback [, callback ...])
  - Routes HTTP GET requests to the specified path with the specified callback functions. Callback functions can be:
    - A middleware function.
    - A series of middleware functions (separated by commas).
    - An array of middleware functions.
  - The next() function gives you the opportunity to do some additional examinations on the incoming URL and still choose to ignore it

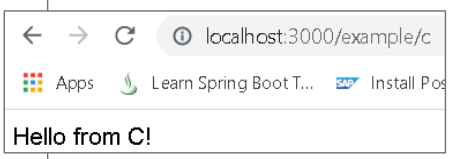
```
app.get('/example/b',  f1 ,  f2 , f3);

……..
function f1(){
    //handle the callback
   next();
}
```

Routing handlers

```
//More than one callback function can handle a route
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...')
  next()
}, function (req, res) {
  res.send('Hello from B!')
})
```




```
//An array of callback functions can handle a route.
var cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

var cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

var cb2 = function (req, res) {
  res.send('Hello from C!')
}
app.get('/example/c', [cb0, cb1, cb2])
```



Views, templates, template engines

- A template engine facilitates you to use static template files in your applications.
  - At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.
  - This approach makes it easier to design an HTML page.
  - Some popular template engines that work with Express are Pug, Mustache, Haml, Hogan, Swig and EJS.

- Using EJS:
  1. install ejs : npm install ejs –save
  2. Create a folder called “views” in main project folder

```
var express = require("express");
var path = require("path");

var app = express();

app.set("views", path.resolve(__dirname, "views"));
app.set("view engine", "ejs");
```

Tells Express that your views will be in a folder called views

Tells Express that you're going to use the EJS templating engine

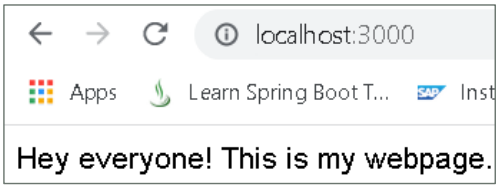
Views, templates, template engines

- Create a file called index.ejs and put it into the “views” directory

index.ejs > ...

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello, world!</title>
  </head>
  <body>
    <%= message %>
  </body>
</html>
```

```
//expresslocal -> view_eg.js
var express = require("express");
var path = require("path");
var app = express();

app.set("views", path.resolve(__dirname, "views"));
app.set("view engine", "ejs");

app.get("/", function(request, response) {
    response.render("index", {
        message: "Hey everyone! This is my webpage."
    });
});

app.listen(3000);
```

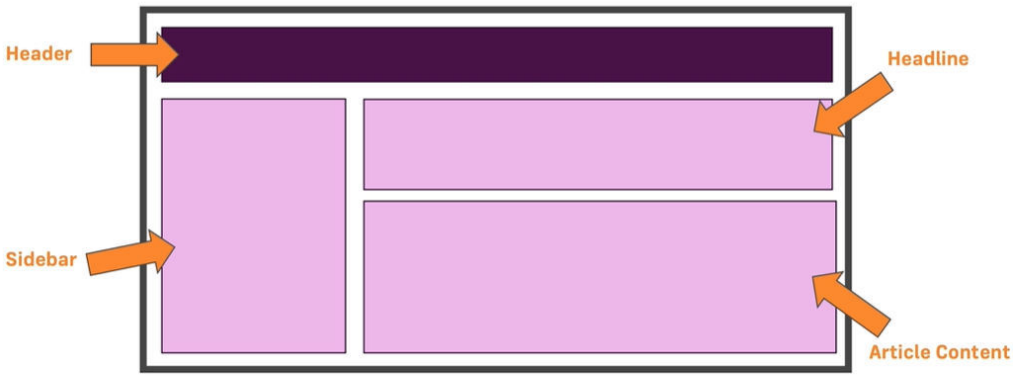


REACT.JS

JavaScript library for building user interfaces

What is React?

- From the official React page : A JavaScript library for building user interfaces
- Its not a framework; React does only one thing – create awesome UI!
- React is used to build single page applications.
- React.js is a JavaScript library. It was developed by engineers at Facebook.
- React is a declarative, efficient, and flexible JavaScript library for building user interfaces.
- It lets you compose complex UIs from small and isolated pieces of code called “components”.



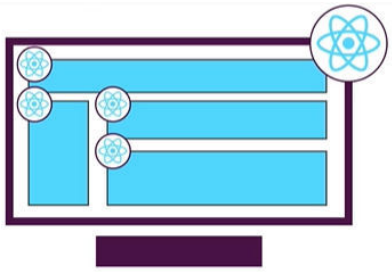
Client-side Javascript frameworks

- Ember : was initially released in December 2011. It is an older framework that has less users than more modern alternatives such as React and Vue
- Angular is an open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations.
- Vue : first released in 2014; is the youngest of the big four, but has enjoyed a recent uptick in popularity.
- React : released by Facebook in 2013. By this point, FB had already been using React to solve many of its problems internally.
  - React itself is not technically a framework; it's a library for rendering UI components.
  - React is used in combination with other libraries to make applications — React and React Native enable developers to make mobile applications; React and ReactDOM enable them to make web applications, etc.

Components

- A **Component** is one of the core building blocks of React.
- Its just a **custom HTML element**!

- Every application you will develop in React will be made up of pieces called components.
  - Components make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.



Why react

- Created and maintained by facebook
- Has a huge community on Github
- Component based architecture
- React is *fast*. Apps made in React can handle complex updates and still feel quick and responsive.
- React is *modular*. Instead of writing large, dense files of code, you can write many smaller, reusable files. React's modularity can be a beautiful solution to JavaScript's maintainability problems.
- React is *scalable*. Large programs that display a lot of changing data are where React performs best.
- React is *popular*.
- UI state becomes difficult to manage with vanilla Javascript

Requirements

- Ensure that NodeJS and typescript are installed
  - Install TypeScript as follows:
  - npm install –g typescript

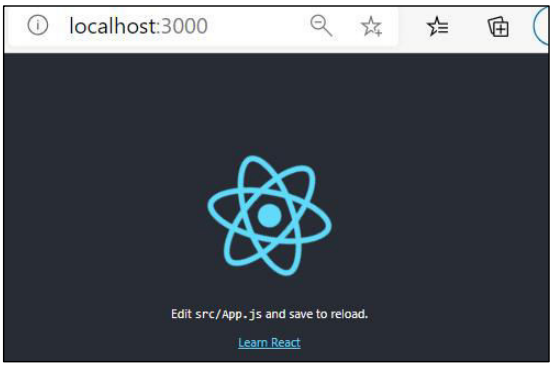
```
C:\Users\Shrilata>node --version
v14.16.0

C:\Users\Shrilata>npm --version
6.14.11

C:\Users\Shrilata>tsc --version
Version 4.4.3

C:\Users\Shrilata>npx --version
6.14.11
```

Using create-react app

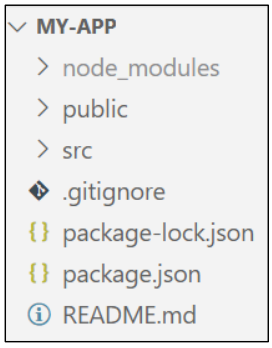
- npx create-react-app my-app
- cd my-app
- npm start

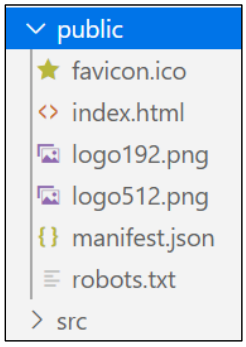
Create React App is a comfortable environment for **learning React**, and is the best way to start building **a new single-page application** in React.

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production. You'll need to have Node >= 14.0.0 and npm >= 5.6 on your machine. To create a project, run:

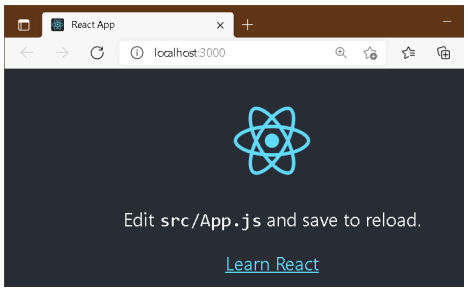


Understanding the folder structure






```
"dependencies": {
  "@testing-library/jest-dom": "^5.11.6",
  "@testing-library/react": "^11.2.2",
  "@testing-library/user-event": "^12.2.2",
  "react": "^17.0.1",
  "react-dom": "^17.0.1",
  "react-scripts": "4.0.0",
  "web-vitals": "^0.2.4"
},
Debug
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```




```
E:\FreelanceTrg\ReactJS\Demo\my-app>npm start

> my-app@0.1.0 start E:\FreelanceTrg\ReactJS\Demo\my-app
> react-scripts start

i ｢wds｣: Project is running at http://192.168.1.18/
i ｢wds｣: webpack output is served from
i ｢wds｣: Content not from webpack is served from E:\Freelar
i ｢wds｣: 404s will fallback to /
Starting the development server...
Compiled successfully!

You can now view my-app in the browser.
```

Index.html

index.html > ...

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--···
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <!--···
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--···
  </body>
</html>
```

- The root node is the HTML element where you want to display the result.
- It is like a container for content managed by React.
- It does NOT have to be a element and it does NOT have to have the id='root

Understanding the folder structure


```
function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}
export default App;
```

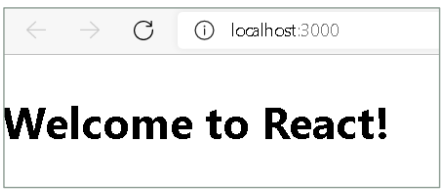
```
function App() {
  return (
    <div >
      <h2>Welcome to React!</h2>
    </div>
  );
}

export default App;
```

index.js

```
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root')
);
```

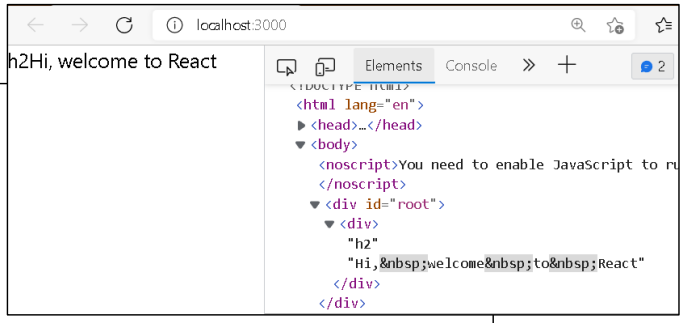
App.css > .App

```
.App {
  text-align: center;
}
```

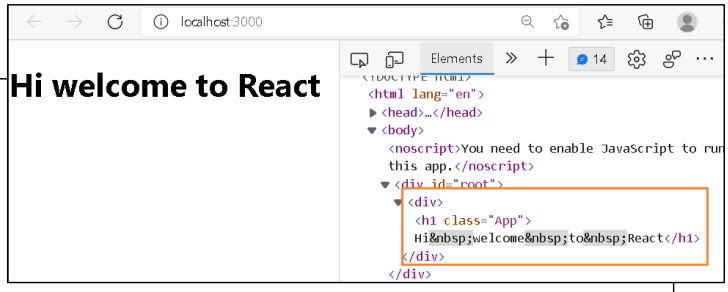


Understanding JSX

```
import React from 'react';

function App() {
  /*return (
    <div >
      <h2>Welcome to React!</h2>
    </div>
  );*/
  return React.createElement('div',null,'h2','Hi, welcome to React');
}
export default App;
```




```
function App() {
  /*return (
    <div >
      <h2>Welcome to React!</h2>
    </div>
  );*/
  return React.createElement('div',null,
          React.createElement('h1',{className:'App'},'Hi welcome to React'));
}
export default App;
```



Understanding JSX

- Eg : const mytag = <h1>Hello React!</h1>;

```
const myelement = <h1>Understanding JSX!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));
```

```
const myelement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);

ReactDOM.render(myelement, document.getElementById('root'));
```

```
function App() {
  return (JSX attribute) React.HTMLAttributes<HTMLDivElement>.className?: string
    <div className="App">
      <h2>Welcome to React!</h2>
    </div>
  );
}
export default App;
```

```
<div class="App">
  <h1> Hi, welcome to React</h1>
</div>
```

```
const myelement = (
  <div>
    <h1>I am a Header.</h1>
    <h1>I am a Header too.</h1>
  </div>
);

ReactDOM.render(myelement, document.getElementById('root'));
```

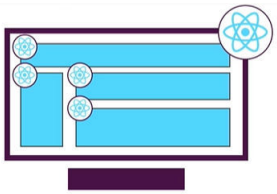
If we want to return more elements, we need to wrap it with one container element. Notice how we are using div as a wrapper for the two h1 elements.

Creating a functional component

- Components are the essential building blocks of any app created with React
  - A single app most often consists of many components.
- A component is in essence, a piece of the UI - splitting the user interface into reusable and independent parts, each of which can be processed separately.
  - Components are independent and reusable bits of code.
  - It's an encapsulated piece of logic.
  - They serve the same purpose as JavaScript functions, but work in isolation and return HTML via a render function.

```
function ExpenseItem(){
   return <h2>Expense Item</h2>
}
export default ExpenseItem;
```

```
const ExpenseItem = () => {
   return <h2>Expense Item</h2>
}
export default ExpenseItem;
```

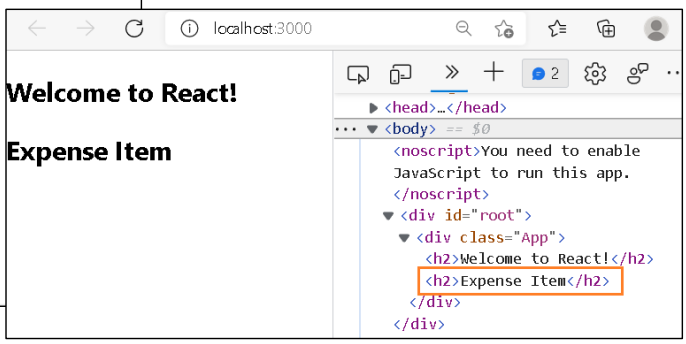


Creating a functional component

- Adding our component to main component
  - To use this component in your application, use similar syntax as normal HTML

App.js > ...

```
import React from 'react';
import ExpenseItem from './components/ExpenseItem';

function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem />
    </div>
  );
}
export default App;
```



- Creating components makes them reusable and configurable.
- Reusing is simple. Eg, simply copy paste <ExpenseItem/> multiple times in App.js.

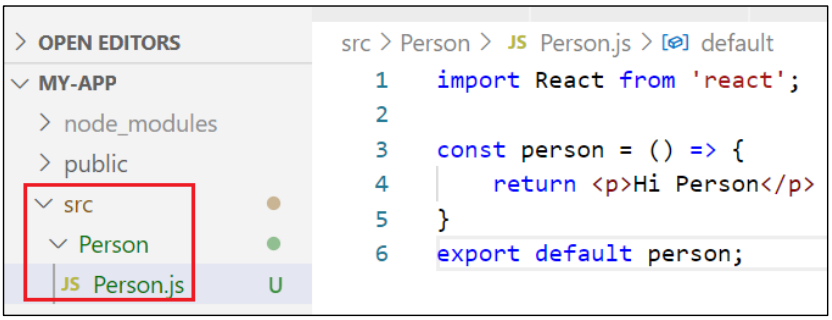
```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem />
      <ExpenseItem />
      <ExpenseItem />
    </div>
  );
}
```

Welcome to React!

Expense Item

Expense Item

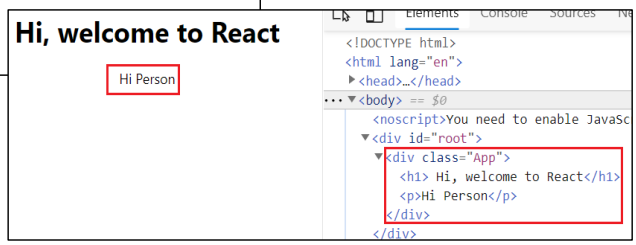
Expense Item



Another example

```
import React, {Component} from 'react';
import './App.css';
import Person from './Person/Person';
function App() {
  return (
    <div className="App">
      <h1> Hi, welcome to React</h1>
        <Person />
    </div>
  );
}
export default App;
```

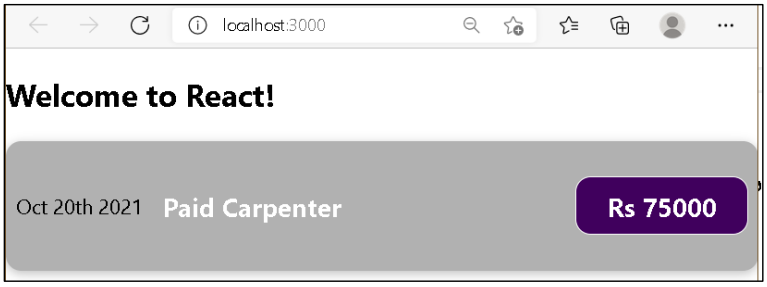
```
return (
  <div className="App">
    <h1> Hi, welcome to React</h1>
    <Person />
    <Person />
    <Person />
  </div>
);
```

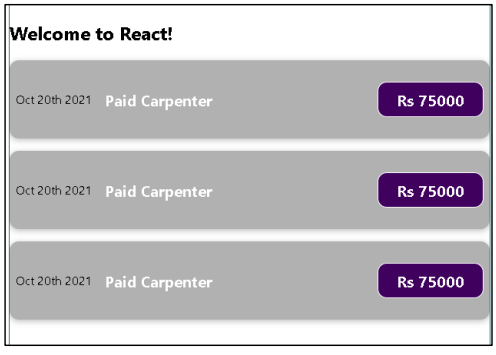


Making our functional component more complex

```
import "./ExpenseItem.css"

const ExpenseItem = () => {
   return (
       <div className="expense-item">
           <div>Oct 20th 2021</div>
           <div className="expense-item__description">
               <h2>Paid Carpenter</h2>
               <p className="expense-item__price">Rs 75000</p>
           </div>
       </div>
   )
}
export default ExpenseItem;
```

```
.expense-item {
    display: flex;
    justify-content: space-between;
    align-items: center;
    box-shadow: 0 2px 8px rgba(0,
    padding: 0.5rem;
    margin: 1rem 0;
    border-radius: 12px;
    background-color: #4b4b4b6e;
}

.expense-item__description {…
}

.expense-item h2 {…
}

.expense-item__price {…
}
```





Components & JSX

- When creating components, you have the choice between two different ways:
- Functional components (also referred to as "presentational", "dumb" or "stateless" components)

```
const cmp = () => {
    return <div>some JSX</div>
}
```

using ES6 arrow functions as shown here is recommended but optional

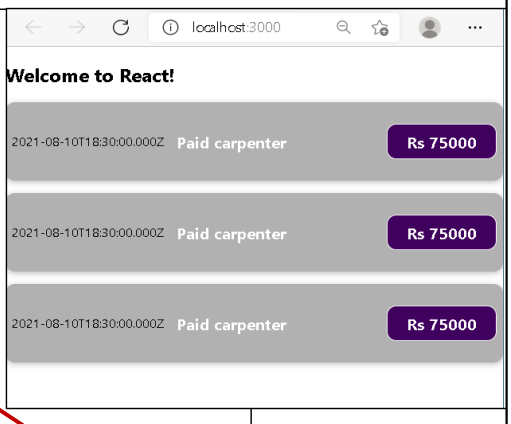
- class-based components (also referred to as "containers", "smart" or "stateful" components)

```
class Cmp extends Component {
    render () {
        return <div>some JSX</div>
    }
}
```

Outputting dynamic content

- If we have some dynamic content in our jsx part which we want to run as javaScript code and not interpret as text, we have to wrap it in single curly braces.

```
import "./ExpenseItem.css"

const ExpenseItem = () => {

    const expDate = new Date(2021, 7, 11);
    const expTitle = "Paid carpenter";
    const expAmount = 75000

    return (
        <div className="expense-item">

            {/* single and multiline comments in JSX */}

            <div>{expDate.toISOString()}</div>
            <div className="expense-item__description">
                <h2>{expTitle}</h2>
                <p className="expense-item__price">Rs {expAmount}</p>
            </div>
        </div>
    )
}
export default ExpenseItem;
```



Comments in JSX

Outputting dynamic content – another example

```
//Person.js
import React from 'react';

const person = () => {
  return <p>Hi Person i am {Math.floor(Math.random() * 30)} years old</p>
}
export default person;
```

Wrap dynamic content in JSX in {...}

```
function App() {
  return (
      <div className="App">
        <h1> Hi, welcome to React</h1>
        <Person />
        <Person />
        <Person />
      </div>
    );
  }
}
```

Hi, welcome to React

Hi Person i am 27 years old

Hi Person i am 29 years old

Hi Person i am 12 years old

PROPS

Passing data via 'Props'

- "Props" stands for properties.
  - It is a special keyword in React used for passing data from one component to another.
  - Props are arguments passed into React components.
  - props are read-only. So, the data coming from a parent component can't be changed by the child component.
  - Props are passed to components via HTML attributes.
  - Props can be used to pass any kind of data such as: String, Array, Integer, Boolean, Objects or, Functions

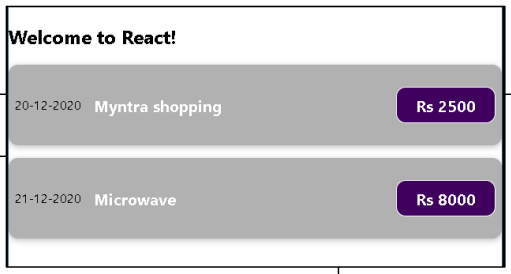
```
import HelloComponent from "./HelloComponent";

const MessageComponent = () => {
    return(
        <div>
            <HelloComponent name="Shrilata" />
        </div>
    );
}
export default MessageComponent;
```

```
const HelloComponent = (props) => {
    return (<h3>Hello, welcome {props.name}</h3>)
}
export default HelloComponent;
```

Passing data via 'Props'

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem expDate="20-12-2020" expTitle="Myntra shopping" expAmount="2500"/>
      <ExpenseItem expDate="21-12-2020" expTitle="Microwave" expAmount="8000"/>
    </div>
  );
}
export default App;
```

you can use as many props as you like

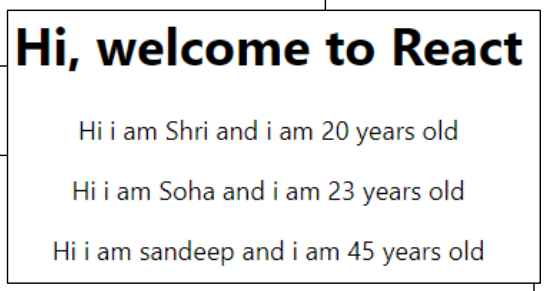



```
const ExpenseItem = (props) => {
   return (
       <div className="expense-item">
           <div>{props.expDate}</div>
           <div className="expense-item__description">
               <h2>{props.expTitle}</h2>
               <p className="expense-item__price">Rs {props.expAmount}</p>
           </div>
       </div>
   )
}
export default ExpenseItem;
```

React uses 'props' to pass attributes from 'parent' component to 'child' component.

Working with props

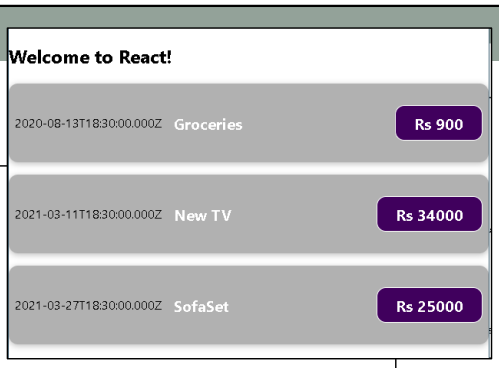
```
function App() {
    return (
      <div className="App">
        <h1> Hi, welcome to React</h1>
        <Person name="Shri" age="20"/>
        <Person name="Soha" age="23”>Hobbies : Coding</Person>
        <Person name="sandeep" age="45"/>
      </div>
    );
```




```
//Person.js
import React from 'react';

const person = (props) => {
   return <p>Hi i am {props.name} and i am {props.age} years old</p>
}
export default person;
```

React uses ‘props’ to pass attributes from ‘parent’ component to ‘child’ component.

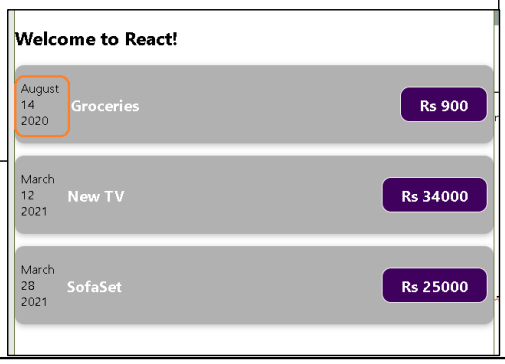
Working with props




```
function App() {
  const expenses = [
    {
      title: 'Groceries',
      amount: 900,
      date: new Date(2020, 7, 14),
    },
    { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
    { title: 'SofaSet', amount: 25000, date: new Date(2021, 2, 28),
    }
  ];

  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpenseItem expDate={expenses[0].date} expTitle={expenses[0].title}
expAmount={expenses[0].amount}/>
      <ExpenseItem expDate={expenses[1].date} expTitle={expenses[1].title}
expAmount={expenses[1].amount}/>
      <ExpenseItem expDate={expenses[2].date} expTitle={expenses[2].title}
expAmount={expenses[2].amount}/>
    </div>
  );
}
export default App;
```

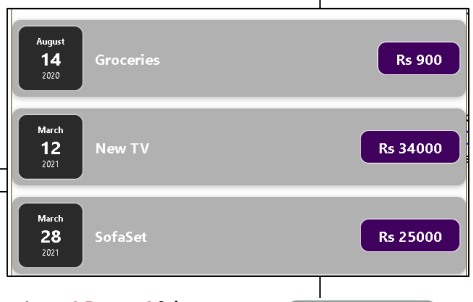
"Javascript" in components

```
const ExpenseItem = (props) => {
    const month = props.expDate.toLocaleString('en-US', {month: 'long'});
    const day = props.expDate.toLocaleString('en-US', {day: '2-digit'});
    const year = props.expDate.getFullYear();

    return (
       <div className="expense-item">
           <div>
               <div>{month}</div>
               <div>{day}</div>
               <div>{year}</div>
           </div>
           <div className="expense-item__description">
               <h2>{props.expTitle}</h2>
               <p className="expense-item__price">Rs {props.expAmount}</p>
           </div>
       </div>
   )
}
export default ExpenseItem;
```

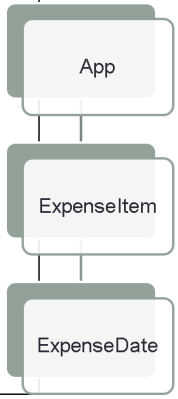


Splitting components further

```
import ExpenseDate from "./ExpenseDate";
const ExpenseItem = (props) => {
   return (
       <div className="expense-item">
           <ExpenseDate date={props.expDate}/>
           <div className="expense-item__description">
               <h2>{props.expTitle}</h2>
               <p className="expense-item__price">Rs {props.expAmount}</p>
           </div>
       </div>
   )
}
export default ExpenseItem;
```




```
import "./ExpenseDate.css"

const ExpenseDate = (props) => {
    const month = props.date.toLocaleString('en-US', {month: 'long'});
    const day = props.date.toLocaleString('en-US', {day: '2-digit'});
    const year = props.date.getFullYear();
    return (
        <div className="expense-date">
            <div className="expense-date__month">{month}</div>
            <div className="expense-date__day">{day}</div>
            <div className="expense-date__year">{year}</div>
        </div>
    );
}
export default ExpenseDate;
```

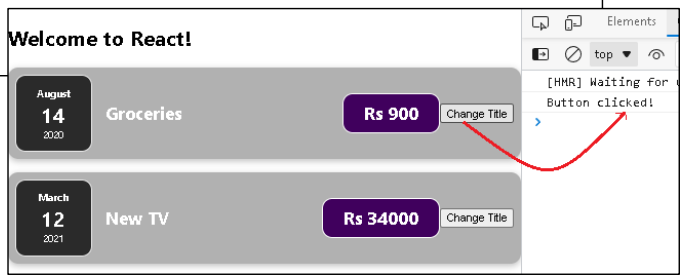


EVENTS AND EVENT HANDLING

Listening to events and working with event handlers

```
const ExpenseItem = (props) => {

    let btnHandler = () => {
       console.log("Button clicked!")
   }

   return (
       <div className="expense-item">
           <ExpenseDate date={props.expDate}/>
           <div className="expense-item__description">
               <h2>{props.expTitle}</h2>
               <p className="expense-item__price">Rs {props.expAmount}</p>
           </div>
           <button onClick={btnHandler}>Change Title</button>
       </div>
   )
}
export default ExpenseItem;
```

No parenthesis ()



STATEFUL COMPONENTS

React State

- The state is a built-in React object that is used to contain data or information about the component.
- A component's state can change over time; whenever it changes, the component re-renders.
  - The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.
- A component with state is known as stateful component.

- State allows us to create components that are dynamic and interactive.
  - State is private, it must not be manipulated from the outside.
  - Also, it is important to know when to use 'state', it is generally used with data that is bound to change.

Component without state

```
const ExpenseItem = (props) => {

    let title = props.expTitle;

    let btnHandler = () => {
        title = "updated expense"
       console.log("Button clicked!")
   }

   return (
       <div className="expense-item">
           <ExpenseDate date={props.expDate}/>
           <div className="expense-item__description">
               <h2>{title}</h2>
               <p className="expense-item__price">Rs {props.expAmount}</p>
           </div>
           <button onClick={btnHandler}>Change Title</button>
       </div>
   )
}
export default ExpenseItem;
```

React Hooks

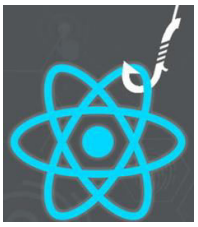
- Hooks allow us to "hook" into React features such as state and lifecycle methods
  - React Hooks are special functions provided by React to handle a specific functionality inside a React functional component.
  - Eg React provides *useState()* function to manage state in a functional component.
  - When a React functional component uses React Hooks, React Hooks attach itself into the component and provides additional functionality.

- You must import Hooks from react
  - Eg : import React, { useState } from "react"; Here - useState is a Hook to keep track of the application state.

- There are some rules for hooks:
  - Hooks can only be called inside React function components.
  - Hooks can only be called at the top level of a component.
  - Hooks cannot be conditional
  - Hooks will not work in React class components.
  - If you have stateful logic that needs to be reused in several components, you can build your own custom Hooks

Working with “state” in functional component

- The React useState Hook allows us to track state in a function component.
- To use the useState Hook, we first need to import it into our component.
  - import { useState } from "react";
  - We initialize our state by calling useState in our function component.



```
import React, {useState} from 'react';

const UseStateComponent = () => {
    useState(); //hooks go here
}
```

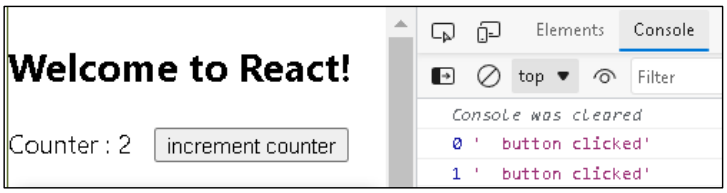
  - useState accepts an initial state and returns two values:
  1. The current state.
  2. A function that updates the state.
  - Eg:

```
function FavoriteColor() {
    const [color, setColor] = useState("");
}
```

    - The first value, color, is our current state.
    - The second value, setColor, is the fuction that is used to update our state.
    - Lastly, we set the initial state to an empty string: useState("")

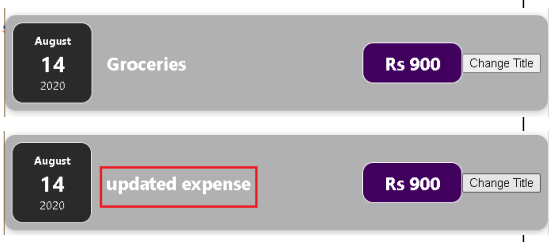
Working with “state” in functional component

```
import React, {useState} from 'react';

const UseStateComponent = () => {
    const [counter, setCounter] = useState(0);  //hooks go here

    const btnHandler = () => {
        setCounter(counter+1);
        console.log(counter, "  button clicked")
     }
    return(
           <div>
              Counter : {counter}   
              <button onClick={btnHandler}>increment counter</button>
           </div>
        );
}
export default UseStateComponent;
```



Working with “state” in functional component

```
import React, {useState} from 'react'

const ExpenseItem = (props) => {

    const [title, setTitle] = useState(props.expTitle);

    let btnHandler = () => {
        setTitle("updated expense")
       console.log("Button clicked!")
   }

   return (
       <div className="expense-item">
           <ExpenseDate date={props.expDate}/>
           <div className="expense-item__description">
               <h2>{title}</h2>
               <p className="expense-item__price">Rs {props.expAmount}</p>
           </div>
           <button onClick={btnHandler}>Change Title</button>
       </div>
   )
}
export default ExpenseItem;
```



props and state

- props and state are CORE concepts of React.
  - Actually, only changes in props and/ or state trigger React to re-render your components and potentially update the DOM in the browser
- Props : allow you to pass data from a parent (wrapping) component to a child component.
  - Eg : AllPosts Component : “title” is the custom property (prop) set up on the custom Post component.
  - Post Component: receives the props argument. React will pass one argument to your component function; an object, which contains all properties you set up on <Post ... /> .
  - {props.title} then dynamically outputs the title property of the props object - which is available since we set the title property inside AllPosts component

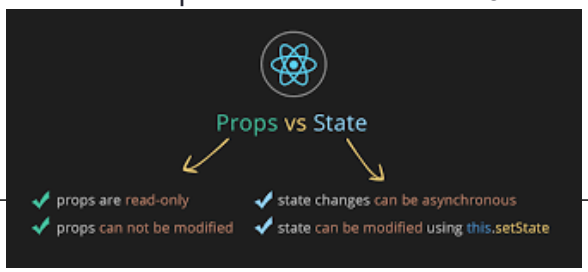
```
//AllPosts
const posts = () => {
    return (
        <div>
            <Post title="My first Post" />
            <Post title="My second Post" />
        </div>
    );
}
```

```
//Post
const post = (props) => {
    return (
        <div>
            <h1>{props.title}</h1>
        </div>
    );
}
```

props and state

- State : While props allow you to pass data down the component tree (and hence trigger an UI update), state is used to change the component's, well, state from within.
  - Changes to state also trigger an UI update.
  - Example: NewPost Component: this component contains state . Only class-based components can define and use state . You can of course pass the state down to functional components, but these then can't directly edit it.

```
class NewPost extends Component { // state can only be accessed in class-based components!
    state = {
        counter: 1
    };

    render () { // Needs to be implemented in class-based components! Return some JSX!
        return (
            <div>{this.state.counter}</div>
        );
    }
}
```



props and state

- Props are immutable.
- They should not be updated by the component to which they are passed.
- They are owned by the component which passes them to some other component.

- State is something internal and private to the component.
- State can and will change depending on the interactions with the outer world.
- State should store as simple data as possible, such as whether an input checkbox is checked or not or a CSS class that hides or displays the component

Simple example : props + state

```
import React, {useState} from 'react'
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
   const [uname, setUname] = useState('Shrilata')
   const [email, setEmail] = useState('shrilata@gmail.com')

   return(
       <ChildComponent uname={uname} email={email} />
   );
}
export default ParentComponent;
```

```
const ChildComponent = (props) => {
    return(
        <div>
            <div>Name : {props.uname}</div>
            <div>Email : {props.email}</div>
        </div>
    );
}
export default ChildComponent;
```

```
function App() {
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ParentComponent />
...
```

Welcome to React!

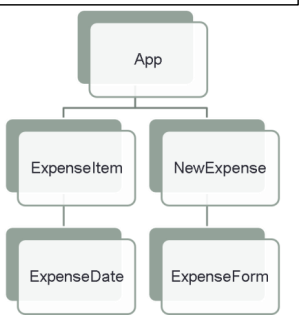
Name : Shrilata
Email : shrilata@gmail.com

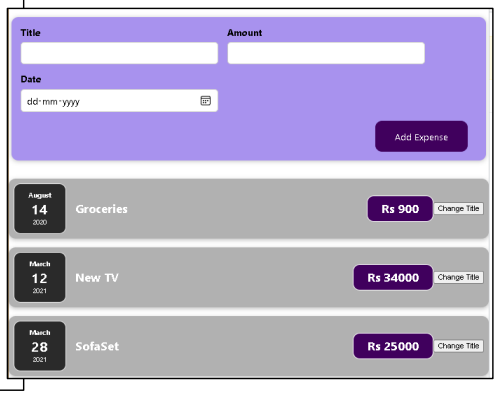
USING FORM FOR INPUT

Adding form inputs

```
import "./ExpenseForm.css"
const ExpenseForm = () => {
    return(
      <form>
          <div className="new-expense__controls">
             <div className="new-expense__control">
                 <label>Title</label>
                 <input />
             </div>
             <div className="new-expense__control">
                 <label>Amount</label>
                 <input type="number"/>
             </div>
             <div className="new-expense__control">
                 <label>Date</label>
                 <input type="date" min="2019-01-01"
                        max="2022-12-31" />
             </div>
          </div>
          <div className="new-expense__actions">
              <button type="submit">Add Expense</button>
          </div>
      </form>
    );
}
export default ExpenseForm;
```

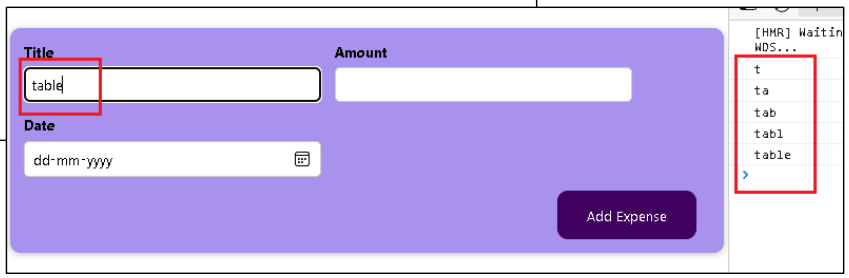
```
const NewExpense = () => {
    return(
        <div className="new-expense">
            <ExpenseForm />
        </div>
    );
}
export default  NewExpense;
```





Listening to user input

```
const ExpenseForm = () => {

    const titleChangeHandler = (event) => {
        console.log(event.target.value)
    }
    return(
      <form>
          <div className="new-expense__controls">
             <div className="new-expense__control">
                 <label>Title</label>
                 <input onChange={titleChangeHandler}/>
             </div>
             ...
          </div>
          <div className="new-expense__actions">
              <button type="submit">Add Expense</button>
          </div>
      </form>
    );
}
export default ExpenseForm;
```



Storing input into state – working with multiple states

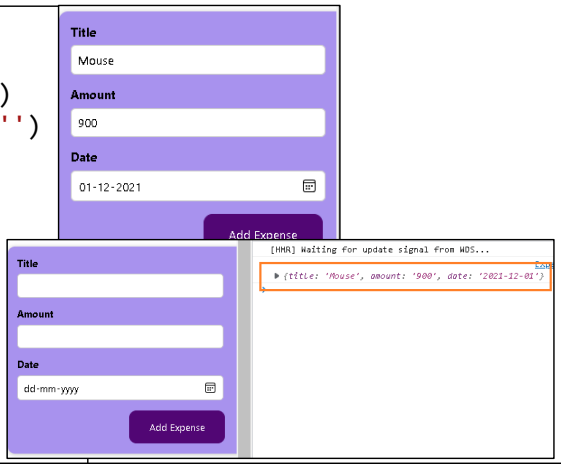
```
import React, {useState} from 'react';
const ExpenseForm = () => {

    const [inputTitle, setInputTitle] = useState('')
    const [inputAmount, setInputAmount] = useState('')
    const [inputDate, setInputDate] = useState('')

    const titleChangeHandler = (event) => {
        setInputTitle(event.target.value)
    }
    const amountChangeHandler = (event) => {
        setInputAmount(event.target.value)
    }
    const dateChangeHandler = (event) => {
        setInputDate(event.target.value)
    }
    return(
      <form>
          <div className="new-expense__controls">
             <div className="new-expense__control">
                 <label>Title</label> <input onChange={titleChangeHandler}/>
             </div>
             <div className="new-expense__control">
                 <label>Amount</label>
                 <input type="number" onChange={amountChangeHandler}/>
             </div> ...
```

Form submission

```
const ExpenseForm = () => {

    const [inputTitle, setInputTitle] = useState('')
    const [inputAmount, setInputAmount] = useState('')
    const [inputDate, setInputDate] = useState('')

    const titleChangeHandler = (event) => {…
    }
    const amountChangeHandler = (event) => {…
    }
    const dateChangeHandler = (event) => {…
    }
    const submitHandler = (event) => {
        event.preventDefault();
    }
    return(
      <form onSubmit={submitHandler}>
          <div className="new-expense__controls">…
          </div>
          <div className="new-expense__actions">
              <button type="submit">Add Expense</button>
          </div>
      </form>
    );
}
```

Form submission – extracting data, 2-way binding

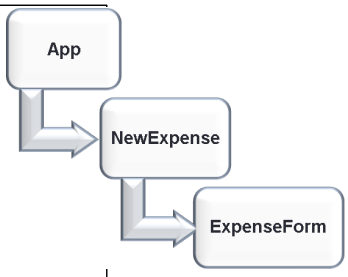
```
const ExpenseForm = () => {

    const [inputTitle, setInputTitle] = useState('')
    const [inputAmount, setInputAmount] = useState('')
    const [inputDate, setInputDate] = useState('')

    const titleChangeHandler = (event) => {...}
    const amountChangeHandler = (event) => {...}
    const dateChangeHandler = (event) => {...}

    const submitHandler = (event) => {
        event.preventDefault();
        const expenseData = {
            title:inputTitle,
            amount:inputAmount,
            date:inputDate
        }
        console.log(expenseData)
        setInputAmount('')
        setInputDate('')
        setInputTitle('')
    }
```




```
return(
      <form onSubmit={submitHandler}>
          <div className="new-expense__controls">
             <div className="new-expense__control">
                 <label>Title</label>
                 <input value={inputTitle}
                        onChange={titleChangeHandler}/>
             </div>
             ...
        </form>
    );
}
```

Passing data from child to parent component

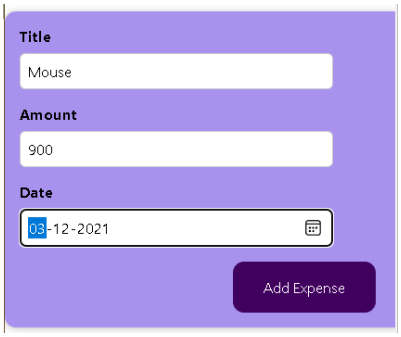
```
const NewExpense = () => {

    const saveExpenseDataHandler = (inputExpenseData) => {
        const expenseData = {
            ...inputExpenseData,
            id:Math.random().toString()
        }
        console.log("In NewExpense ",expenseData)
    }
    return(
        <div className="new-expense">
            <ExpenseForm onSaveExpenseData={saveExpenseDataHandler}/>
        </div>
    );
}

export default  NewExpense;
```




```
const ExpenseForm = (props) => {

    ...
    const submitHandler = (event) => {
        event.preventDefault();
        const expenseData = {
            title:inputTitle,
            amount:inputAmount,
            date: new Date(inputDate)
        }
        //console.log(expenseData)
        props.onSaveExpenseData(expenseData);
        ...
    }
```

Passing data from child to parent

```
function App() {
  const expenses = [...];

  const addExpenseHandler = expense => {
    console.log("In App component ", expense)
  }

  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <NewExpense onAddExpense={addExpenseHandler} />
      ...
  );
}
export default App;
```



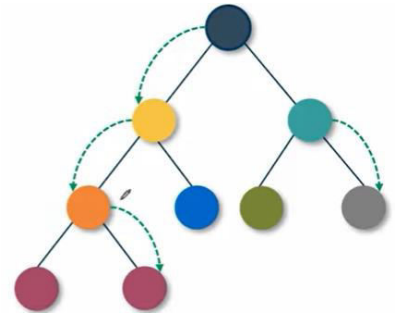

```
In App component                                    App.js:20
▶ {title: 'Mouse', amount: '900', date: '2021-12-03',
   id: '0.25759054649698765'}
```

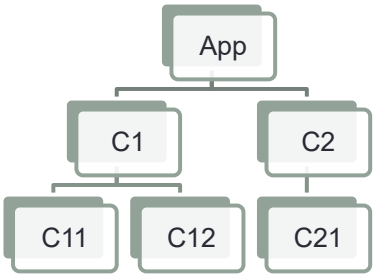
```
const NewExpense = (props) => {

    const saveExpenseDataHandler = (inputExpenseData) => {
        const expenseData = {
            ...inputExpenseData,
            id:Math.random().toString()
        }
        //console.log("In NewExpense ",expenseData)
        props.onAddExpense(expenseData)
    }
    return(...);
}
export default  NewExpense;
```

LIFTING STATE UP

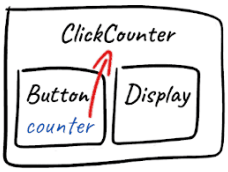
Lifting state up in React.js

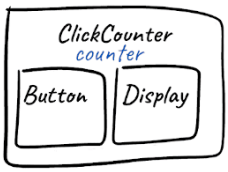
- In a typical application, you pass a lot of state down to child components as props.
  - These props are often passed down multiple component levels.
  - That's how state is shared vertically in your application.
- Often there will be a need to share state between different components.
  - The common approach to share state between two components is to move the state to common parent of the two components.
  - This approach is called as lifting state up in React.js
  - React components can manage their own state
  - Often components need to communicate state to others
  - Siblings do not pass state to each other directly
  - State should pass through a parent, then trickle down





Lifting state up – simple example





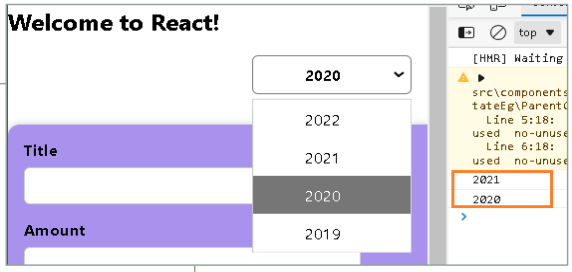

```
import React, {useState} from 'react'
import Display from './Display'
import Button from './Button'

const ClickCounter = () => {
   const [counter, setCounter] = useState(0)

   const incrCounter = () => {
       setCounter(counter + 1)
   }
   return(
       <div>
           <Button onClick={incrCounter}>Button</Button>
           <Display message={`Clicked ${counter} times`}/>
       </div>
   );
}
export default ClickCounter;
```

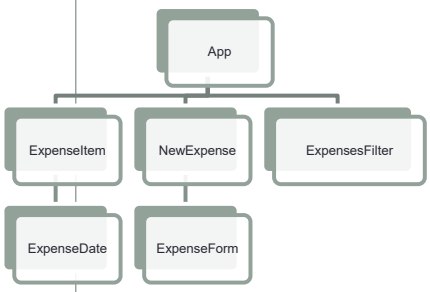
```
const Display = (props) => {
     return <p>{props.message}</p>;
}
export default Display;
```


```
const Button = (props) => {
    return <button onClick={props.onClick}>Click me</button>;
}
export default Button;
```

A small aside – add filter to date

```
const ExpensesFilter = (props) => {

  const selectChangeHandler = (event) => {
      console.log(event.target.value)
      props.onSelectYear(event.target.value)
  }
  return (
    <div className='expenses-filter'>
      <div className='expenses-filter__control'>
        <label>Filter by year</label>
        <select onChange={selectChangeHandler}>
          <option value='2022'>2022</option>
          <option value='2021'>2021</option>
          <option value='2020'>2020</option>
          <option value='2019'>2019</option>
        </select>
      </div>
    </div>
  );
};
export default ExpensesFilter;
```






```
function App() {
  const [filteredYear, setFilteredYear] = useState(2020);

  const selectYearHandler = filteredValue => {
    setFilteredYear(filteredValue)
  }
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <ExpensesFilter onSelectYear={selectYearHandler}/>
```

WORKING WITH LISTS AND CONDITIONALS

```
const SimpleListComponent = () => {
   const nums = [1,2,3,4,5]
   const updatedNums = nums.map((num)=>{
    return <li>{num*num}</li>;
  });

   const items =[{name:'Item-1'}, {name:'Item-2'}]
   const updatedItems = items.map(item => (
      <p>{item.name}</p>
    ))

   const students = [{name:"Anita",rollno:101},
                     {name:"Sunita",rollno:102},
                     {name:"Kavita",rollno:103}]
   const updatedStudents = students.map(student => (
      <tr>
         <td>Name {student.name} </td>
         <td>Rollno {student.rollno} </td>
       </tr>
   ))
return(
    <div>
        <h2> Numbers List</h2> {updatedNums}
        <h2> Items List</h2>   {updatedItems}
        <h2> Students List</h2>
        <table border="1">{updatedStudents}</table>
    </div>
);}
export default SimpleListComponent
```

Working with lists

Numbers List

- 1
- 4
- 9
- 16
- 25

Items List

Item-1

Item-2

Students List

Name Anita	Rollno 101
Name Sunita	Rollno 102
Name Kavita	Rollno 103

Working with lists in child component

```
import SimpleListChildComponent from "./SimpleListChildComponent";

const SimpleListComponent = () => {
   const nums = [1,2,3,4,5]

   const items =[{name:'Item-1'}, {name:'Item-2'}]

   const students = [{name:"Anita",rollno:101},
                     {name:"Sunita",rollno:102},
                     {name:"Kavita",rollno:103}]
return(
    <div>
        <SimpleListChildComponent
            nums={nums}
            items={items}
            students={students} />
    </div>
);
}
export default SimpleListComponent
```

```
const SimpleListChildComponent = (props) => {

   const nums = props.nums;
   const updatedNums = nums.map(...);
   const items = props.items
   const updatedItems = items.map(...)

   const students = props.students;
   const updatedStudents = students.map(...)

    return(...);
}
export default SimpleListChildComponent;
```

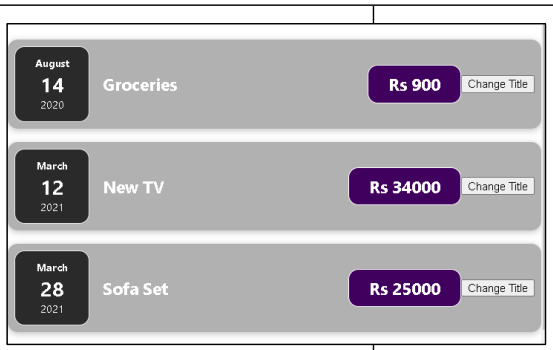
Working with the expenses list

```
const expenses = [
  { title: 'Groceries’, amount: 900, date: new Date(2020, 7, 14)},
  { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { title: 'SofaSet', amount: 25000, date: new Date(2021, 2, 28)}
];
return (
  <div className="App">
    <h2>Welcome to React!</h2>
    <ExpenseItem
         expDate={expenses[0].date}
         expTitle={expenses[0].title}
         expAmount={expenses[0].amount}
    />
    <ExpenseItem
         expDate={expenses[1].date}
         expTitle={expenses[1].title}
         expAmount={expenses[1].amount}
    />
    <ExpenseItem
         expDate={expenses[2].date}
         expTitle={expenses[2].title}
         expAmount={expenses[2].amount}
    />
  </div>
);
}
```

Working with the expenses list

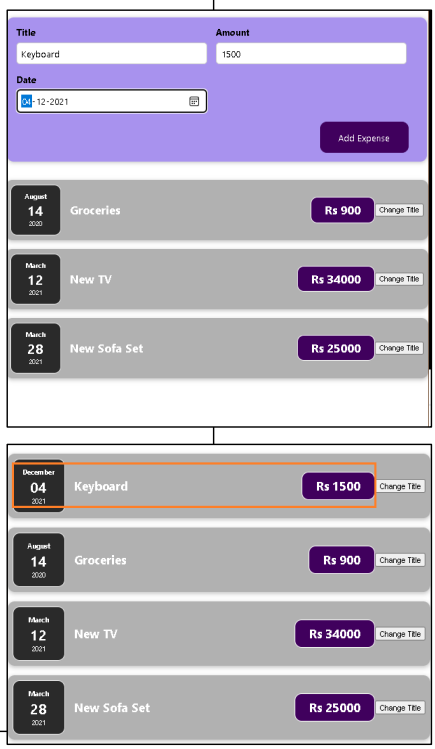
```
function App() {
  const expenses = [
    { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
    { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
    { title: 'Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
  ];
  return (
    <div className="App">
      <h2>Welcome to React!</h2>

       {expenses.map(expense => {
          return <ExpenseItem
                   expDate={expense.date}
                   expTitle={expense.title}
                   expAmount={expense.amount}
                 />
          })}
       {/* <ExpenseItem
            expDate={expenses[0].date}
            expTitle={expenses[0].title}
            expAmount={expenses[0].amount}
       />  ... */}
    </div>
  );
}
export default App;
```

```
{expenses.map(expense =>
         (<ExpenseItem
             expDate={expense.date}
             expTitle={expense.title}
             expAmount={expense.amount}
          />))
}
```



Using stateful lists

```
const DUMMY_EXP = [
  { title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
  { title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { title: 'New Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
];
function App() {
  const [expenses, setExpenses] = useState(DUMMY_EXP)

  const addExpenseHandler = expense => {
    setExpenses(prevArr => {return [expense, ...prevArr]})
  }
  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <NewExpense onAddExpense={addExpenseHandler} />

       {expenses.map(expense => (<ExpenseItem
                  expDate={expense.date}
                  expTitle={expense.title}
                  expAmount={expense.amount}
                 />))
       }
    </div>
  );
}
export default App;
```



Lists and keys


```
const DUMMY_EXP = [
  { id:101, title: 'Groceries', amount: 900, date: new Date(2020, 7, 14)},
  { id:102,title: 'New TV', amount: 34000, date: new Date(2021, 2, 12) },
  { id:103,title: 'New Sofa Set', amount: 25000, date: new Date(2021, 2, 28)}
];

function App() {

  const [expenses, setExpenses] = useState(DUMMY_EXP)
  ...
  return (
    <div className="App">
      ...
       {expenses.map(expense => (
                <ExpenseItem
                  key={expense.id}
                  expDate={expense.date}
                  expTitle={expense.title}
                  expAmount={expense.amount}
                 />))
       }
...
```

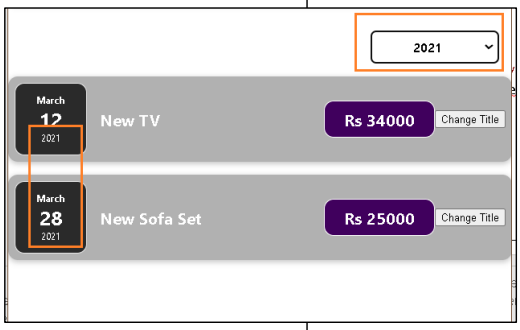
Lists and keys

- When creating a list in JSX, React may show you an error and ask for a key.
  - Keys are unique identifiers that must be attached to the top-level element inside a map.
  - Keys are used by React to know how to update a list whether adding, updating, or deleting items.
  - This is part of how React is so fast with large lists.
  - Keys are a way to help React know how to efficiently update a list.
  - We can add a key using the key prop like so:

```
<div>
 {people.map(person => (
   <p key={person.name}>{person.name}</p>
 ))}
</div>
```

Implementing filters

```
function App() {
  const [expenses, setExpenses] = useState(DUMMY_EXP)
  const [filteredYear, setFilteredYear] = useState(2020);

  const filteredExpensesArr = expenses.filter(expense =>{
    return expense.date.getFullYear().toString() === filteredYear;
  })

...
  const selectYearHandler = filteredValue => {
    setFilteredYear(filteredValue)
  }

  return (
    <div className="App">
      <h2>Welcome to React!</h2>
      <NewExpense onAddExpense={addExpenseHandler} />
      <ExpensesFilter onSelectYear={selectYearHandler}/>
       {filteredExpensesArr.map(expense => (
                <ExpenseItem
                  key={expense.id}
                  expDate={expense.date}
                  expTitle={expense.title}
                  expAmount={expense.amount}
                 />))
       }
```



Rendering content conditionally

- Conditional rendering means to render a specific HTML element or React component depending on a prop or state value.
  - In a conditional render, a React component decides based on one or several conditions which DOM elements it will return.
  - For instance, based on some logic it can either return a list of items or a text that says "Sorry, the list is empty".

```
if(condition_is_met) {
    renderSectionOfUI();
  }
```

Rendering content conditionally : example

```
/*const users = [
    { id: '1', firstName: 'Shrilata', lastName: 'T' },
    { id: '2', firstName: 'Anita', lastName: 'Patil' },
  ];*/
  const users = []

  function ListUsers() {
    return (
      <div>
        <List list={users} />
      </div>
    );
  }

 function List({ list }) {
    if (!list) {
      return null;
    }
    return (
      <ul>
        {list.map(item => (
          <Item key={item.id} item={item} />
        ))}
      </ul>
    );
  }
```

```
if (!list.length) {
        return <p>Sorry, the list is empty.</p>;
    }
```

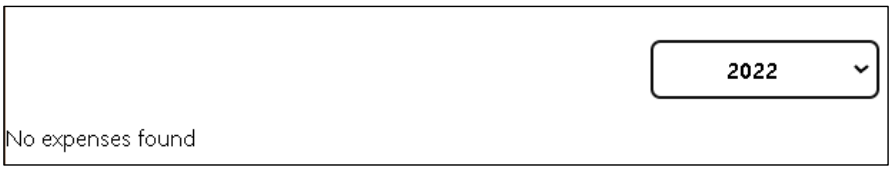
```
function Item({ item }) {
    return (
      <li>
        {item.firstName} {item.lastName}
      </li>
    );
  }
  export default ListUsers;
```

Hello Conditional Rendering

- Shrilata T
- Anita Patil

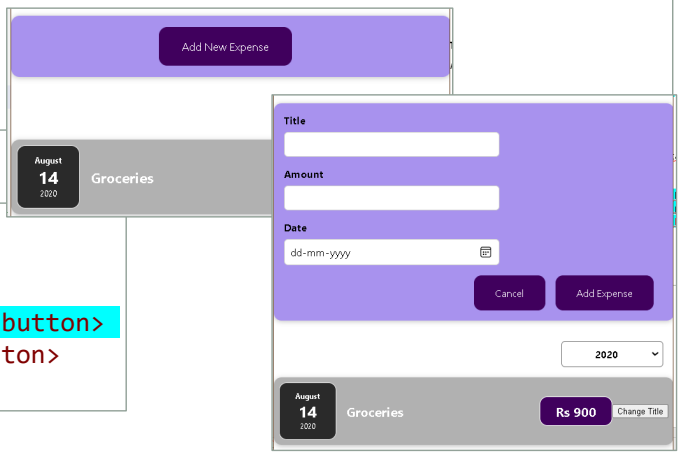
Rendering content conditionally : Expense tracker example

```
{filteredExpensesArr.length ==0 ? <p>No expenses found</p> :
          filteredExpensesArr.map(expense => (
               <ExpenseItem
                 key={expense.id}
                 expDate={expense.date}
                 expTitle={expense.title}
                 expAmount={expense.amount}
                />))
}
```



Rendering content conditionally : one more example

```
const NewExpense = (props) => {
    const [showForm, setShowForm] = useState(false)

    const showFormHandler = () => {
        setShowForm(true)
    }
    const saveExpenseDataHandler = (inputExpenseData) => {...}
    return(
        <div className="new-expense">
            {!showForm && <button onClick={showFormHandler}>Add New Expense </button>}
            {showForm && <ExpenseForm onCancel={stopShowForm}
                                      onSaveExpenseData={saveExpenseDataHandler}/>}
        </div>
    );
}
export default  NewExpense;
```




```
//ExpenseForm
<div className="new-expense__actions">
   <button type="button"
        onClick={props.onCancel}>Cancel</button>
   <button type="submit">Add Expense</button>
</div>
```

CLASS-BASED COMPONENTS

```
const HelloComponent = (props) => {
    return (<h3>Hello, welcome user!!</h3>)
}
export default HelloComponent;
```

Functional components are regular javascript functions which return renderable results (typically JSX)

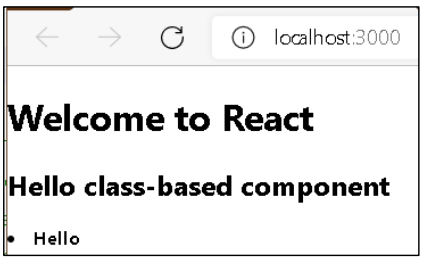
```
class HelloComponent extends Component{
    render(){
        return (<h3>Hello, welcome user!!</h3>)
    }
}
export default HelloComponent;
```

Class based components are defined using Javascript classes where a render method defines the to-be-rendered output

Class-based components : Examples

```
import React, {Component} from 'react';
class HelloComponent extends Component{
  render(){
     return (<h2>Hello class-based component</h2>)
  }
}
export default HelloComponent;
```

```
import './User.css';
import React, {Component} from 'react';

class User extends Component{
  render(){
      return <li className='user'>Hello User</li>
  }
};
export default User;
```




```
function App() {
  return (
    <div className="App">
      <h1> Welcome to React</h1>
      <HelloComponent />
      <User />
    </div>
  );
}
```

Class-based components : passing into props

```
import React, {Component} from 'react';
import User from './User'

const DUMMY_USERS = [
    { id: 'u1', name: 'Shrilata' },
    { id: 'u2', name: 'Soha' },
    { id: 'u3', name: 'Sia' },
  ];

class Users extends Component{
    render(){
        return(
        <div>
            <User name={DUMMY_USERS[0].name} />
            <User name={DUMMY_USERS[1].name} />
            <User name={DUMMY_USERS[2].name} />
        </div>
        );
    }
}
export default Users;
```

```
function App() {
  return (
    <div className="App">
      <h1> Welcome to React</h1>
      <Users />
    </div>
  );
}
```

Welcome to React

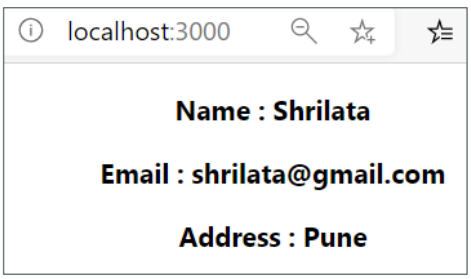
- Hello Max
- Hello Manuel
- Hello Julie

```
import './User.css';
import React, {Component} from 'react';

class User extends Component{
  render(){
      return (<li className='user'>Hello {this.props.name}</li>);
  }
};
export default User;
```

React State : Example

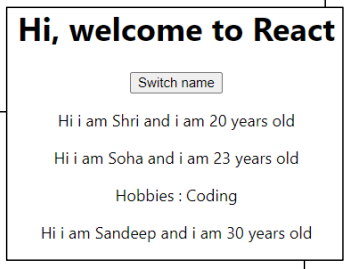
```
import React, {Component} from 'react';
import './App.css';
import StatefulComponent from "./StatefulComponent/StatefulComponent";

class App extends Component {
 render() {
   return (
     <div>
       <StatefulComponent />
     </div>
   );
 }
}
export default App;
```

```
import React,{Component} from 'react';

class statefulComponent extends Component{
    state = {
        name: "Shrilata",
        email: "shrilata@gmail.com",
        address:"Pune"
     }
    render(){
        return(
          <div>
            <h3>Name : {this.state.name}</h3>
            <h3>Email : {this.state.email}</h3>
            <h3>Address : {this.state.address}</h3>
          </div>
        );
    }
}
export default statefulComponent;
```



State and props

```
const person = (props) => {
return (
    <div>
       <p>Hi i am {props.name}
          and i am {props.age} years old</p>
    </div>
)
}
export default person;
```



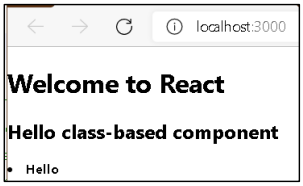

```
class App extends Component {
  state = {
    persons:[
      {name:"Shri",age:20},
      {name:"Soha",age:23},
      {name:"Sandeep",age:30},
    ]
  }
  render() {
    return (
      <div className="App">
        <h1> Hi, welcome to React</h1>
        <button>Switch name</button>
        <Person name={this.state.persons[0].name} age={this.state.persons[0].age}/>
        <Person name={this.state.persons[1].name} age={this.state.persons[1].age}>
             Hobbies : Coding
        </Person>
        <Person name={this.state.persons[2].name} age={this.state.persons[2].age}/>
      </div>
    );
  }
}
```

LIFECYCLE EVENTS

Lifecycle Events

- React class components can have hooks for several lifecycle events
  - During the lifetime of a component, there's a series of events that gets called, and to each event you can hook and provide custom functionality.
  - React lifecycle methods are a series of events that happen from the birth of a React component to its death.
  - There are 4 phases in a React component lifecycle:
    - initial
    - Mounting
    - Updating
    - Unmounting

```
class HelloComponent extends Component{
  constructor(){
      super();
       this.state = {message: "hello" }
  }
  render(){
     return (<h2>Hello World</h2>)
  }
}
```



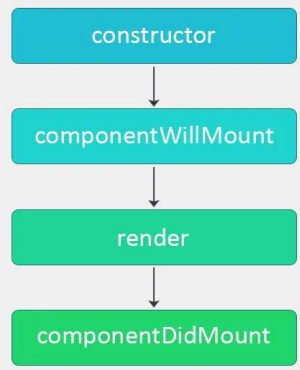
Lifecycle methods

- React lifecycle methods:
  - Each React lifecycle phase has a number of lifecycle methods that you can override to run code at specified times during the process.
  - These are popularly known as component lifecycle methods.

- Initialisation phase:
  - Constructor(props): This is a special function that is called when new components are created. In this, we initialize the state or props of the component.

Lifecycle methods

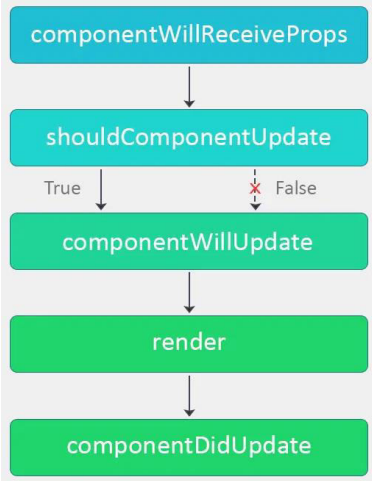
- The mounting phase refers to the phase during which a component is created and inserted to the DOM.
- The following methods are called in order.
  - **ComponentWillMount**(): This function is called immediately before mounting occurs. It is called right before the first rendering is performed.
  - **render**(): You have this method for all the components created. It returns the Html node.
  - **componentDidMount**(): This method is called right after the react component is mounted on DOM or you can say that it is called right after render method executed for the first time
  - Here we can make API call, foreg, interact with dom or perform any ajax call to load data.




```
class HelloComponent extends Component{
    componentDidMount(){
        console.log("Mounted component")
    }
    render(){
        return <h1>Hello</h1>
    }
}
```

Lifecycle methods

- Update: In this state, the dom is interacted by a user and updated. For example, you enter something in the textbox, so the state properties are updated.
  - The component is re-rendered whenever a change is made to react component's state or props, you can simply say that the component is updated.
- Following are the methods available in update state:
  - shouldComponentUpdate() : called when the component is updated.
  - componentDidUpdate() : after the component is updated.



- UnMounting: this state comes into the picture when the Component is not required or removed.
- Following are the methods available in unmount state:
  - ComponentWillUnmount(): called when the Component is removed or destroyed.

```
import React, {Component} from 'react';

class LifecycleComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: ''};

    this.UpdateName = this.UpdateName.bind(this);
    this.testclick = this.testclick.bind(this);
  }

  UpdateName(event) {
    this.setState({name: event.target.value});
  }

  testclick(event) {
    alert("The name entered is: "+ this.state.name);
  }

  componentDidMount() {
    console.log('Mounting State : calling method componentDidMount');
  }

  shouldComponentUpdate() {
    console.log('Update  State : calling method shouldComponentUpdate');
    return true;
  }
```

```
componentDidUpdate() {
    console.log('Update  State : calling method
componentDidUpdate')
  }
  componentWillUnmount() {
    console.log('UnMounting State : calling method
componentWillUnmount');
  }

  render() {
    return (
      <div>
         Enter Your Name:<input type="text"
value={this.state.name} onChange={this.UpdateName} /><br/>
        <h2>{this.state.name}</h2>
        <input type="button"
          value="Click Here"
          onClick={this.testclick} />
      </div>
    );
  }
}

export default LifecycleComponent;
```


```
Mounting State : calling method componentDidMount
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
Update  State : calling method shouldComponentUpdate
Update  State : calling method componentDidUpdate
```

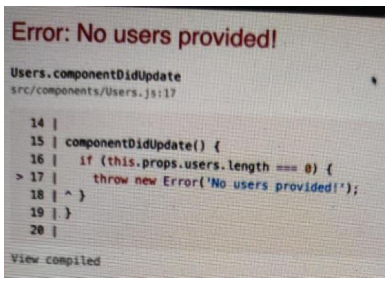
[Lifecycle Methods (kirupa.com)]

ERROR BOUNDARIES

Using Error Boundaries

```
const person = (props) => {
    const rnd = Math.random();
    if(rnd > 0.7){
       throw new Error("Something went wrong");
    }
return (
    ...
)
};
export default person;
```


```
class Users extends Component{

  componentDidUpdate(){
      if(this.props.users.length == 0)
         throw new Error("No users in list!")
  }
...
```



- A JavaScript error in a part of the UI shouldn't break the whole app.
- To solve this problem for React users, React 16 introduces a new concept of an "error boundary".

Using Error Boundaries

```
import React, {Component} from 'react';
class ErrorBoundary extends Component{
    state = {
        hasError:false,
        errorMessage:''
    }
    componentDidCatch = (error, info) => {
        this.setState({hasError:true, errorMessage:error});
    }
    render(){
        if(this.state.hasError)
            return <h1>{this.state.errorMessage}</h1>
        else
            return this.props.children
    }
}
export default ErrorBoundary;
```

```
import React from 'react'
function Artists({artistName}) {
  if (artistName === 'peruzzi') {
   throw new Error ('not performing tonight!')
   }
  return (
   <div>
    {artistName}
   </div>
  )
 }
export default Artists
```

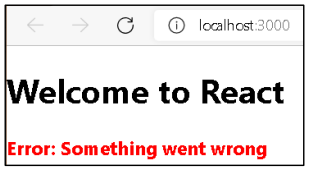
Use it like this:

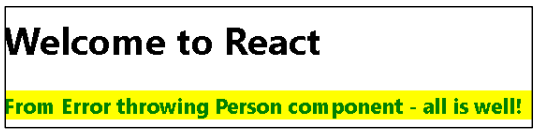
```
<ErrorBoundary>
   <Artists />
</ErrorBoundary>
```

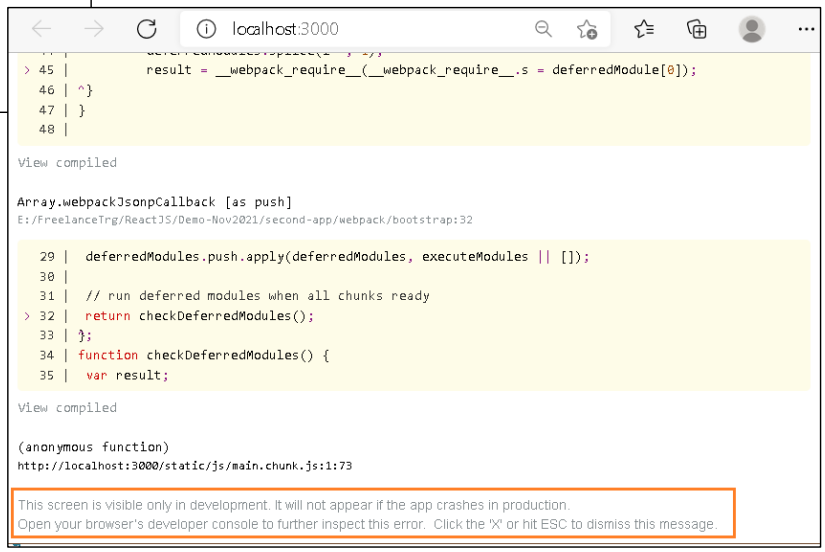
- Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering
- https://reactjs.org/docs/error-boundaries.html
- React.Component – React (reactjs.org)

Using Error Boundaries

```
function App() {
  return (
    <div className="App">
      <h1> Welcome to React</h1>
        <ErrorBoundary >
            <Person />
        </ErrorBoundary>
    </div>
  );
}
```







FORMS AND FORMS VALIDATION

Handling user input the right way

Controlled Components and Uncontrolled components

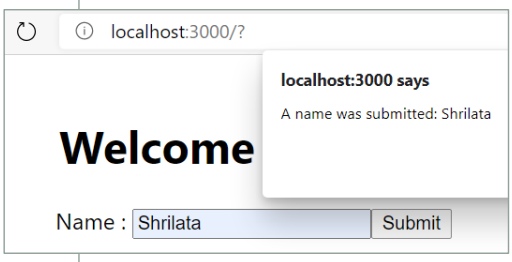
- React forms present a unique challenge because you can either allow the browser to handle most of the form elements and collect data through React change events, or you can use React to fully control the element by setting and updating the input value directly.
  - The first approach is called an uncontrolled component because React is not setting the value.
  - The second approach is called a controlled component because React is actively updating the input.

- In HTML, form data is usually handled by the DOM.
- In React, form data is usually handled by the components.
  - When the data is handled by the components, all the data is stored in the component state.

Controlled Inputs in class components

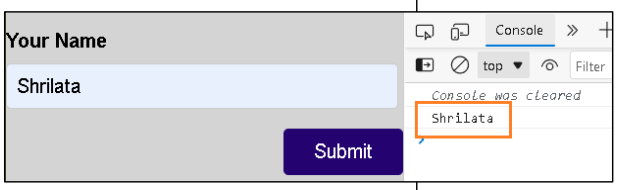
- An input is said to be “controlled” when React is responsible for maintaining and setting its state.
  - The state is kept in sync with the input’s value, meaning that changing the input will update the state, and updating the state will change the input.

```
class ControlledInput extends React.Component {
   state = { name: '' };

    handleInput = (event) => {
      this.setState({name: event.target.value });
    }
    handleSubmit = (event) => {
        alert('A name was submitted: ' + this.state.name);
        event.preventDefault();
      }
    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          Name : <input value={this.state.name}
                        onChange={this.handleInput} />
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```



Controlled Inputs using React hooks (functional components)

```
import React, {useState} from 'react'

const SimpleInput = (props) => {
    const [inputName, setInputName] = useState('')

    const inputNameHandler = (event) => {
        setInputName(event.target.value)
    }

    const formSubmitHandler = event => {
         event.preventDefault();
         console.log(inputName)
    }
    return (
      <form onSubmit={formSubmitHandler}>
        <div className='form-control'>
          <label htmlFor='name'>Your Name</label>
          <input type='text' id='name' onChange={inputNameHandler}/>
        </div>
        <div className="form-actions">
          <button>Submit</button>
        </div>
      </form>
    );
  };
  export default SimpleInput;
```



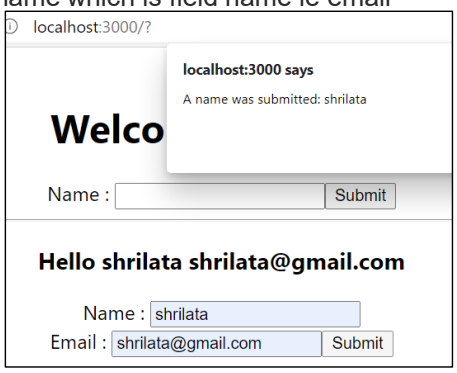
Controlled Inputs

- Controlled inputs open up the following possibilities:
  - **instant input validation**: we can give the user instant feedback without having to wait for them to submit the form (e.g. if their password is not complex enough)
  - **instant input formatting**: we can add proper separators to currency inputs, or grouping to phone numbers on the fly
  - **conditionally disable form submission**: we can enable the submit button after certain criteria are met (e.g. the user consented to the terms and conditions)
  - **dynamically generate new inputs**: we can add additional inputs to a form based on the user's previous input (e.g. adding details of additional people on a hotel booking)

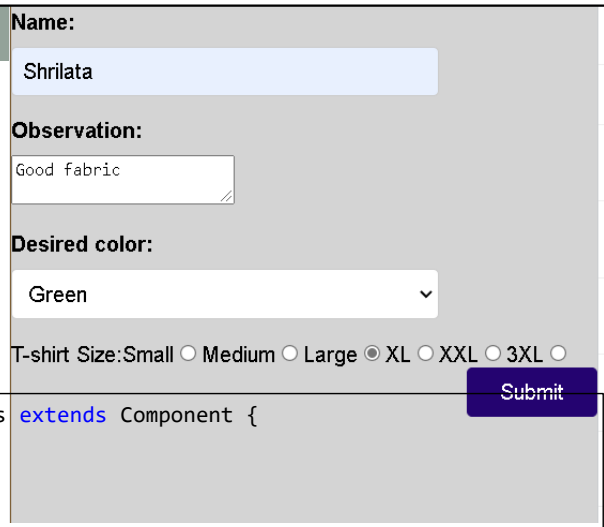
Handling Multiple Form Inputs

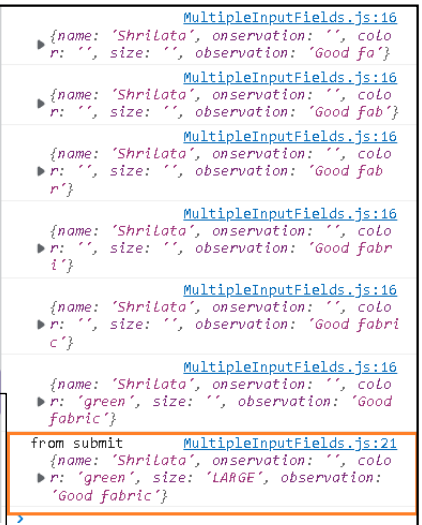
```
class ControlledLoginForm extends React.Component {
   state = {
    username: '',
    email: ''
  };
    handleInput = (event) => {
      let name = event.target.name;
      let val = event.target.value;
      this.setState({[name]: val});
      console.log(this.state)
    }
    handleSubmit = (event) => {
        alert('A name was submitted: ’
              + this.state.username);
        event.preventDefault();
      }
    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <h3>Hello {this.state.username} {this.state.email}</h3>
          Name : <input  name="username" onChange={this.handleInput} /><br />
          Email : <input  name="email" onChange={this.handleInput} />
          <input type="submit" value="Submit" />
        </form>
      ); }
  }
```

- {username: 'shrilata'} //where username is event.target.name which is field name ie username
- {email: 'Shrilata@gmail.com'} //where email is event.target.name which is field name ie email



One more example






```
class MultipleInputFields extends Component {

    state = {
        name:'',
        onservation:'',
        color:'',
        size:'',
    }
    handleChanges = (event) => {
         let name = event.target.name;
         let val = event.target.value;
         this.setState({[name]: val});
         console.log(this.state)
    };
    submitFormHandler = (event) => {
        event.preventDefault();
        console.log("from submit ", this.state);
    };
    render(){
      const colors = ['Blue', 'Red', 'Green', 'Yellow'];
      const sizes = ['Small', 'Medium', 'Large', 'XL', 'XXL', '3XL'];
```

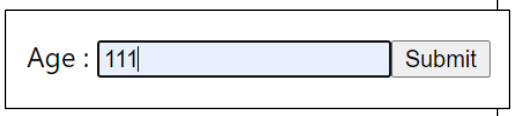
```
return (
    <form onSubmit={this.submitFormHandler}>
        <div className='form-control'>
          <label>Name:</label>
          <input name="name" type="text" value={this.state.name} onChange={this.handleChanges} />
          </div>
          <div className='form-control'>
            <label>Observation:</label>
            <textarea name="observation" value={this.state.observation} onChange={this.handleChanges} />
           </div>
           <div className='form-control'>
          <label>Desired color:</label>
          <select name="color" value={this.state.color} onChange={this.handleChanges}>
            {colors.map((color, i) => <option key={i} value={color.toLowerCase()}>{color}</option>)}
          </select>
          </div>
          <div >
          <label>T-shirt Size:</label>
          {sizes.map((size, i) =>
            <label key={i}> {size}
              <input
                name="size"  value={size.toUpperCase()}  checked={this.state.size === size.toUpperCase()}
                onChange={this.handleChanges}    type="radio" />
            </label>
          )}
       </div>
       <div className="form-actions">
       <button type="submit">Submit</button>
       </div>
    </form>
  )}
}
export default MultipleInputFields;
```

Controlled components : Summary

- A controlled component is bound to a value, and its adjustments will be handled in code by using event-based callbacks.
  - Here, the input form variable is handled by the react itself rather than the DOM.
  - In this case, the mutable state is maintained in the state property and modified using setState().
- Controlled components have functions which regulate the data that occurs at each on Change event.
  - This data is subsequently saved in the setState() method and updated. It helps components manage the elements and data of the form easier.
-
- You can use the controlled component when you create:
  - Forms validation so that when you type, you always have to know the input value to verify whether it is true or not!
  - Disable submission icon, except for valid data in all fields
  - If you have a format such as the input for a credit card

Validation

```
class ControlledInputValidation1 extends React.Component {
   state = { age: '' };

    handleInput = (event) => {
      let nam = event.target.name;
      let val = event.target.value;
      if (nam === "age") {
        if (!Number(val))
          alert("Age must be a number");
      }
      this.setState({[nam]: val});
    }
    handleSubmit = (event) => {
        alert('A age was submitted: ' + this.state.age);
        event.preventDefault();
      }
    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          Age : <input name="age" value={this.state.age} onChange={this.handleInput}
/>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```



Uncontrolled Inputs

- “uncontrolled” form inputs: React doesn’t track the input’s state.
  - HTML input elements naturally keep track of their own state as part of the DOM, and so when the form is submitted we have to read the values from the DOM elements themselves.

- “uncontrolled” form inputs: React doesn’t track the input’s state.
  - If the DOM handles the data, then the form is uncontrolled, and if the state of the form component manages the data, then the form is said to be controlled

  - Uncontrolled components are inputs that do not have a value property. In opposite to controlled components, it is the application's responsibility to keep the component state and the input value in sync.

  - In order to do this, React allows us to create a “**ref**” (reference) to associate with an element, giving access to the underlying DOM node

Uncontrolled Inputs

- In **uncontrolled** components form data is being handled by DOM itself.
- For example here we can reference form values by name
- This is quick and dirty way of handling forms. It is mostly useful for simple forms or when you are *just learning React*.

- HTML input elements keep track of their own state
  - When the form is submitted we typically read the values from the DOM elements ourselves

First Name:

Last Name:

Submit

```
class ProfileForm extends Component {
  handleSubmit = (event) => {
    event.preventDefault();

    const firstName = event.target.firstName.value;
    const lastName = event.target.lastName.value;

    // Here we do something with form data
    console.log(firstName, lastName)
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input name="firstName" type="text" />
        </label>
        <label>
          Surname:
          <input name="lastName" type="text" />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Uncontrolled Inputs

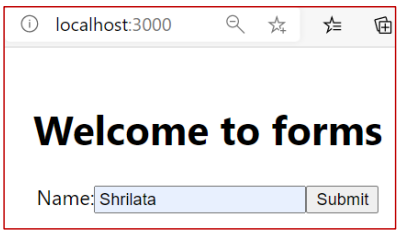
- "**ref** is used to receive the form value from DOM.
  - To enable this, React allows us to create a "ref" (reference) to associate with an element, giving access to the underlying DOM node.
  - Refs provide a way to access DOM nodes or React elements created in the render method.
  - Refs are created using React.createRef() and attached to React elements via the ref attribute.
  - Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

Uncontrolled Inputs

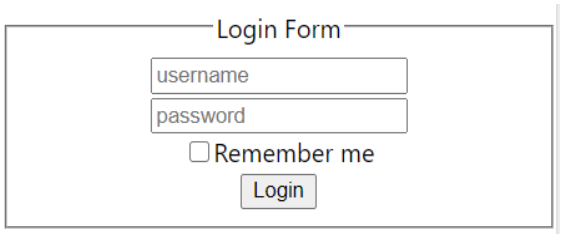
```
import React, {Component} from 'react';
class SimpleForm extends Component {
    constructor(props) {
        super(props);
        // create a ref to store the DOM element
        this.nameEl = React.createRef();
    }

    handleSubmit = (e) => {
      e.preventDefault();
      alert(this.nameEl.current.value);
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>Name:
            <input type="text" ref={this.nameEl} />
          </label>
          <input type="submit" name="Submit" />
        </form>
      )
    }
  }
export default SimpleForm;
```

- You initialize a new ref in the constructor by calling React.createRef, assigning it to an instance property so it's available for the lifetime of the component.
- In order to associate the ref with an input, it's passed to the element as the special ref attribute.
- Once this is done, the input's underlying DOM node can be accessed via this.nameEl.current.



Another example : Login form

```
class LoginForm extends Component {
   constructor(props) {
      super(props);
      this.nameEl = React.createRef();
      this.passwordEl = React.createRef();
      this.rememberMeEl = React.createRef();
    }
   handleSubmit = (e) => {
        e.preventDefault();
        const data = {
            username: this.nameEl.current.value,
            password: this.passwordEl.current.value,
            rememberMe: this.rememberMeEl.current.checked,
          }
        console.log(data)
      }
    render(){
      return (
       <form onSubmit={this.handleSubmit}>
         <fieldset><legend>Login Form</legend>
         <input type="text" placeholder="username" ref={this.nameEl} /><br></br>
         <input type="password" placeholder="password" ref={this.passwordEl} /><br></br>
         <label><input type="checkbox" ref={this.rememberMeEl} />Remember me
         </label><br></br>
         <button type="submit" className="myButton">Login</button>
         </fieldset>
        </form>
     );
}}
```



COMPOSITION VS. INHERITANCE

Composition over Inheritance

- Composition and inheritance are the approaches to use multiple components together in React.js .
- This helps in code reuse.
- React recommend using composition instead of inheritance as much as possible and inheritance should be used in very specific cases only.
- Composition works with functions as well as classes both.

Inheritance in JS

```
class Automobile {
    constructor() {
        this.vehicleName = automobile;
        this.numWheels = null;
    }
    printNumWheels() {
        console.log(`This ${this.vehicleName} has ${this.numWheels} wheels`);
    }
}
class Car extends Automobile {
    constructor() {
        super(this);
        this.vehicleName = 'car';
        this.numWheels = 4;
    }
}
class Bicycle extends Automobile {
    constructor() {
        super(this);
        this.vehicleName = 'bike';
        this.numWheels = 2;
    }
}
const car = new Car();
const bike = new Bicycle();
car.printNumWheels() // This car has 4 wheels
bike.printNumWheels() // This bike has 2 wheels
```

```
class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.methodA = this.methodA.bind(this);
  }

  methodA() {
    console.log("methodA in parent class");
  }

  render() {
    return false;
  }
}
```

Console output

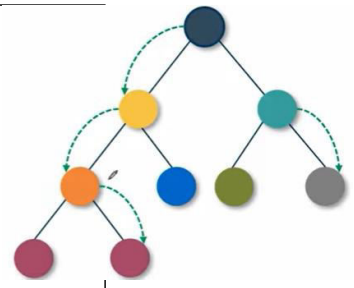
```
In child class, calling parent method...   child.js:9
methodA in parent class                    parent.js:10
```

```
import Parent from "./parent";

 class Child extends Parent {
    constructor() {
      super();
    }
    render() {
      console.log("In child class, calling parent method...");
      this.methodA();
      return false;
    }
  }
```

Composition

- Composition is a code reuse technique where a larger object is created by combining multiple smaller objects.

```
class App extends React.Component {
    render() {
      return <Toolbar theme="dark" />;
    }
  }

  function Toolbar(props) {
    return (
      <div>
        <ThemedButton theme={props.theme} />
      </div>
    );
  }

  class ThemedButton extends React.Component {
    render() {
      return <Button theme={this.props.theme} />;
    }
  }
```



```
class App extends Component {
state = {
   date:new Date()
...
}
...
return (
  <div className="container">
    <NewsHeader className="news" subject="Sports"
            date={this.state.date.toString()} />
```

News for Sun Jun 1

News Heading : Sports

News Title Content Title-1

News Content : Lots of Content-1

Author for Content Title-1 - Shrilata

News Title Content Title-2

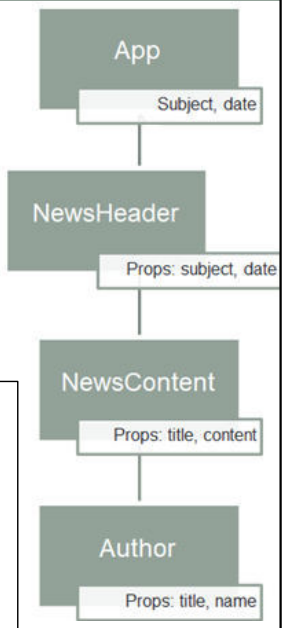
News Content : Lots of Content-2

Author for Content Title-2 - Shrilata

News Title Content Title-3

News Content : Lots of Content-3

Author for Content Title-3 - Shrilata




```
const newsHeader = (props) => {
 return(
 <div>
    <h1>News for {props.date}</h1>
    <h2>News Heading : {props.subject}</h2>
    <NewsContent title="Content Title-1" content="Lots of Content-1" />
    <NewsContent title="Content Title-2" content="Lots of Content-2" />
    <NewsContent title="Content Title-3" content="Lots of Content-3" />
```

```
const newsContent = (props) => {
 return(
  <div>
    {/*complex code that filters out news based on subject*/}
    <h4><b><i>News Title {props.title}</i></b></h4>
    <h4>News Content : {props.content}</h4>
    <Author title={props.title} name="Shrilata" />
```

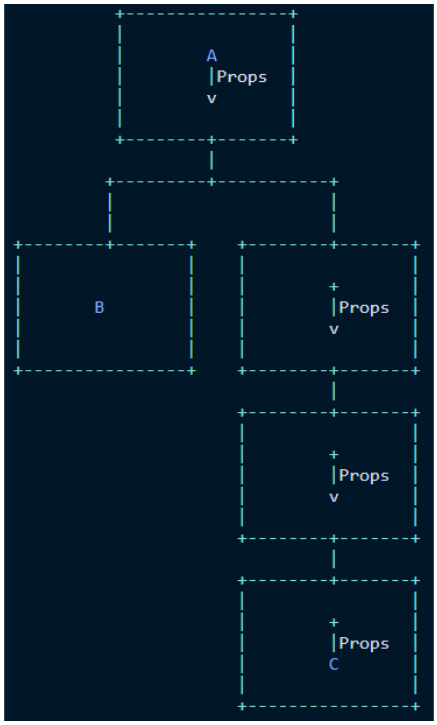
```
const author = (props) => {
 return(
  <h6>Author for {props.title} - {props.name}</h6>
```

CONTEXT

Context

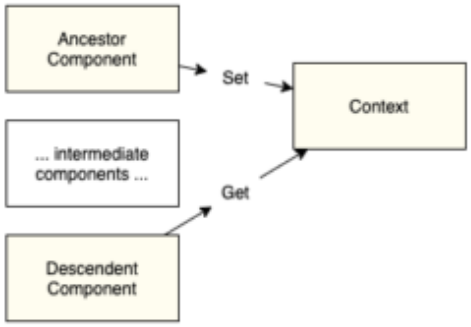
- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language

```
class App extends React.Component {
    render() {
      return <Toolbar theme="dark" />;
    }
  }

  function Toolbar(props) {
    return (
      <div>
        <ThemedButton theme={props.theme} />
      </div>
    );
  }

  class ThemedButton extends React.Component {
    render() {
      return <Button theme={this.props.theme} />;
    }
  }
```



React Context API

- Store the state in a Context value in the common ancestor component (called the Provider Component), and access it from as many components as needed (called Consumer Components), which can be nested at any depth under this ancestor.
- This solution has the same benefits as the Props solution, but because of what could be called “hierarchical scoping”, it has the added benefit that any component can access the state in any Context that is rooted above itself in React’s hierarchy, without this state needing to be passed down to it as props.
- React.js takes care of all the magic behind the scenes to make this work.
- Primary situations where the React Context API really shines are:
  - When your state needs to be accessed or set from deeply nested components.
  - When your state needs to be accessed or set from many child components.



Three aspects to using React Contexts

- 1) Defining the Context object so we can use it.
  - If we wanted to store data about the current user of a web app, we could create a UserContext that can be used in the next two steps:

```
// Here we provide the initial value of the context
const UserContext = React.createContext({
    currentUser: null,
});
```

Note: It doesn't matter where this Context lives, as long as it can be accessed by all components that need to use it in the next two steps.

- 2) Providing a value for a Context in the hierarchy.
  - Assuming you had an AccountView component, you might provide a value like this

```
const AccountView = (props) => {
    const [currentUser, setCurrentUser] = React.useState(null);
    return (
        {/* Here we provides the actual value for its descendents */}
        <UserContext.Provider value={{ currentUser: currentUser }}>
        <AccountSummary/>
        <AccountProfile/>
        </UserContext.Provider>
    );
};
```

Three aspects to using React Contexts

- 3) Accessing the current Context value lower in the hierarchy.
  - If the AccountSummary component needed the user, we could have just passed it as a prop. But let's assume that it doesn't directly access the user data, but rather contains another component that does:

```
// Here we don't use the Context directly, but render children that do.
const AccountSummary = (props) => {
  return (
    <AccountSummaryHeader/>
    <AccountSummaryDashboard/>
    <AccountSummaryFooter/>
  );
};
```

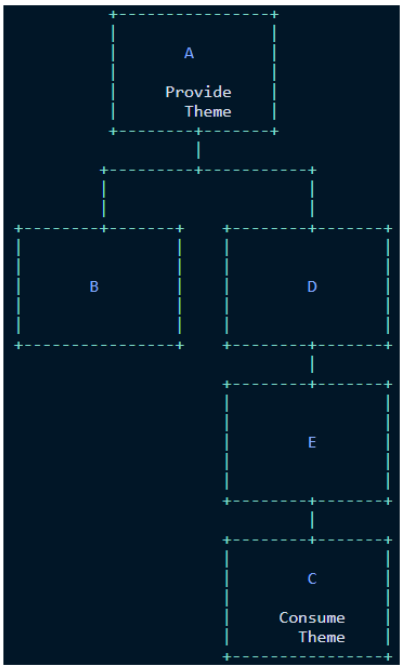
  - All three of these child components may not want to access the current user's data. But as an example, let's just look at the AccountSummaryHeader component:

```
const AccountSummaryHeader = (props) => {
// Here we retrieve the current value of the context
    const context = React.useContext(UserContext);
     return (
         <section><h2>{context.currentUser.name}</h2> </section>
    );
};
```

Context

- Using context, we can avoid passing props through intermediate elements

```
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
// A component in the middle doesn't have to pass the theme d
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```



Bringing Bootstrap into your React App

- After creating your app project:
- ..Demo\second-app>npm install --save react-bootstrap bootstrap@3
- Then start the app : npm start

- In Index.js, put these as the 1st 2 lines:
- import 'bootstrap/dist/css/bootstrap.css';
- import 'bootstrap/dist/css/bootstrap-theme.css';

- Now use Bootstrap classes in App.js or any other component:
- Eg :

```
render(){
  return(
      <div className="container">
        <h1 className="text-danger">TODO LIST </h1>
```

REDUX

Because state management can be hard

What is state

- Eg:

```
const state = {
  posts: [],
  signUpModal: {
    open: false
  }
}
```

```
<div className={this.state.signUpModal.open ? 'hidden' : ''}>
  Sign Up Modal
</div>
```

- state references the condition of something at a particular point in time, such as whether a modal is open or not.
- In a React component the state holds data which can be rendered to the user.
- The state in React could also change in response to actions and events: in fact you can update the local component's state with this.setState().
- So, in general a typical JavaScript application is full of state. For example, state is:
  - what the user sees (data)
  - the data we fetch from an API
  - the URL
  - the items selected inside a page
  - eventual errors to show to the user

State can be complex

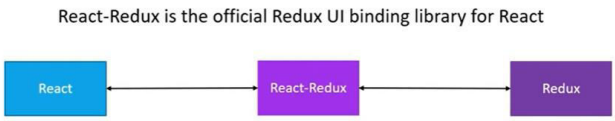
- Even an innocent single page app could grow out of control without clear boundaries between every layer of the application. This holds particularly true in React.
  - You can get by with keeping the state within a parent React component (or in context) as long as the application remains small.
  - Then things will become tricky especially when you add more behaviours to the app. At some point you may want to reach for a consistent way to keep track of state changes.

Create a Redux application

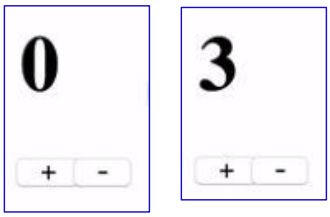
npx create-react-app redux-app

cd redux-app

npm install redux react-redux




```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

//STORE -> GLOBALISED STATE

//ACTION -> INCREMENT

//REDUCER

//DISPATCH

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```



Create a Redux application

- Step-1 : Create the store:

```
import {createStore} from 'redux';
const myStore = createStore(reducer-name)
```

- Step-2 : Create action

```
const increment = () => {
  return {
    type :'INCREMENT'  //name of the action
  }
}
const decrement = () => {
  return {
    type :'DECREMENT'  //name of the action
  }
}
```

- Step-3 : Create reducer

```
function reducer(state=initial-state, action){}
```

```
const counter = (state=0, action) => {
  switch(action.type){
    case "INCREMENT":
      return state + 1;
      case "DECREMENT":
      return state - 1;
  }
}
let store = createStore(counter)
```

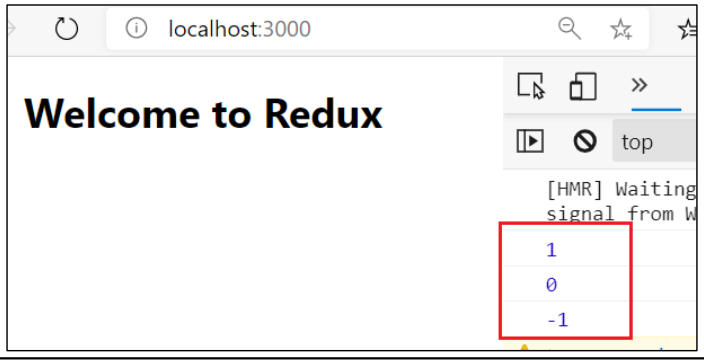
Create a Redux application

- Step-4 : display store on console
- `store.subscribe(() => console.log(store.getState()))`

- Step-5 : dispatch the action
- `store.dispatch(increment()); //dispatches the increment action`

```
//DISPATCH
store.dispatch(increment()); //dispatches the increment action
store.dispatch(decrement()); //dispatches the decrement action
store.dispatch(decrement()); //dispatches the decrement action again
```

- Step-6 : Execute the app. Start server
- npm start



Need for Redux

- Redux offers a solution to storing all your application state in one place called "**Store**"
- Components then "**dispatch**" state changes to store, not directly to other components
- Components that need to be aware of state changes can "**subscribe**" to the store
- The center of every Redux application is the store. A "store" is a container that holds your application's global state.
  - A store is a JavaScript object with a few special functions and abilities that make it different than a plain global object:
  - You must never directly modify or change the state that is kept inside the Redux store
  - Instead, the only way to cause an update to the state is to create a plain action object that describes "something that happened in the application", and then dispatch the action to the store to tell it what happened.
  - When an action is dispatched, the store runs the root reducer function, and lets it calculate the new state based on the old state and the action
  - Finally, the store notifies subscribers that the state has been updated so the UI can be updated with the new data.

Actions, reducers and dispatchers

- An **action** is a plain JavaScript object that has a type field. You can think of an action as an event that describes something that happened in the application.
  - The type field should be a string that gives this action a descriptive name. Eg "todoAdded" or "depositFunds" or "incrementCounter"

- A **reducer** is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state: (state, action) => newState.
  - You can think of a reducer as an event listener which handles events based on the received action (event) type.

- The Redux store has a method called **dispatch**. The only way to update the state is to call store.dispatch() and pass in an action object.
  - The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value:

My original index.js

```
//original React imports
import {createStore} from 'redux';

//STORE -> GLOBALISED STATE

//ACTION -> INCREMENT
const increment = () => {
  return {
    type :'INCREMENT'  //name of the action
  }
}
const decrement = () => {
  return {
    type :'DECREMENT'  //name of the action
  }
}

//REDUCER
const counter = (state=0, action) => {
  switch(action.type){
    case "INCREMENT":
      return state + 1;
      case "DECREMENT":
      return state - 1;
  }
}
```

```
let store = createStore(counter);

//Display it in console
store.subscribe(() =>
     console.log(store.getState()));

//DISPATCH
store.dispatch(increment());
store.dispatch(decrement());

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

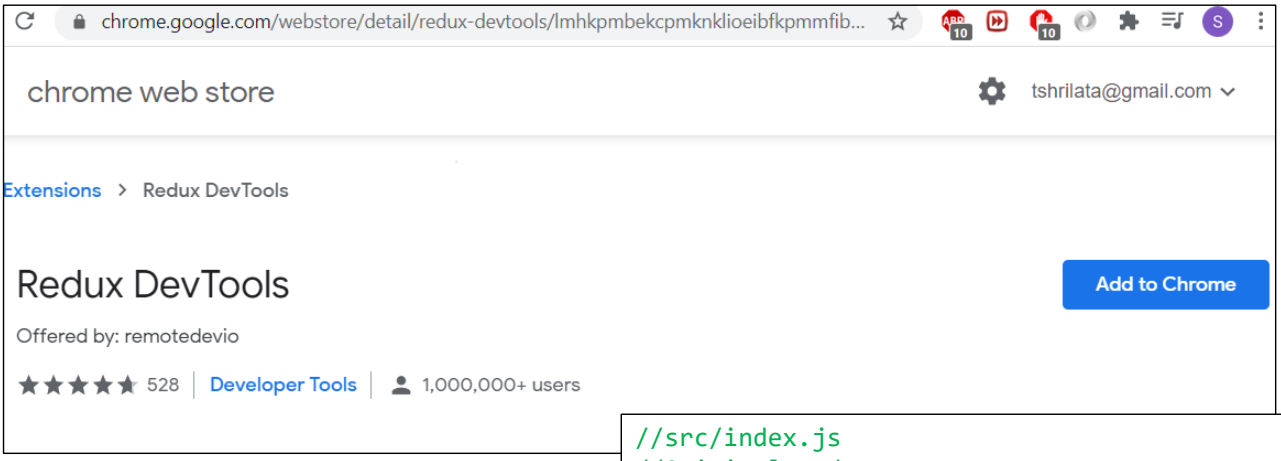
Create a Redux app

```
//reducers/counter.js
const counterReducer = (state=0, action) => {
    switch(action.type){
      case "INCREMENT":
        return state + 1;
      case "DECREMENT":
        return state - 1;
      default: return null;
    }
  }
  export default counterReducer;
```

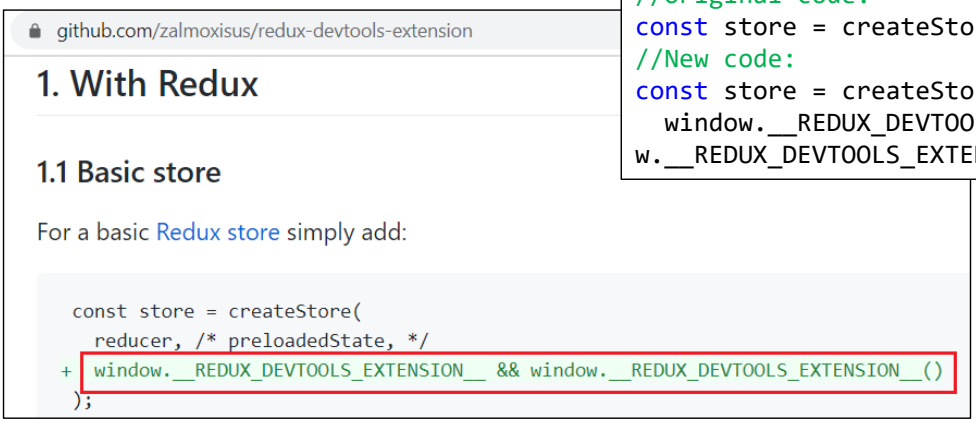
```
//src/index.js
import {createStore} from 'redux';
import allReducers from "./reducers";

const store = createStore(allReducers);
```

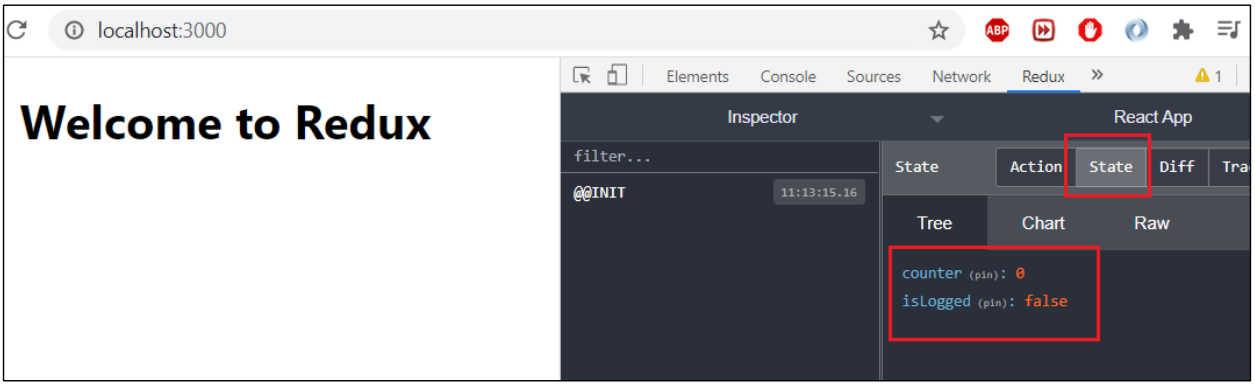
```
//reducers/isLogged.js
const loggedReducer = (state=false, action) => {
    switch(action.type){
      case "SIGNIN":
        return !state;
      default:
         return state;
    }
  }
  export default loggedReducer;
```

```
//reducers/index.js
import counterReducer from "./counter";
import loggedReducer from "./isLogged";
import {combineReducers} from 'redux';

const allReducers = combineReducers({
    counter : counterReducer,
    isLogged:loggedReducer
})
export default allReducers;
```




```
//src/index.js
//Original code:
const store = createStore(allReducers);
//New code:
const store = createStore(allReducers,
  window.__REDUX_DEVTOOLS_EXTENSION__ && windo
w.__REDUX_DEVTOOLS_EXTENSION__());
```






```
//src/index.js
import {createStore} from 'redux';
import allReducers from "./reducers";
import {Provider} from 'react-redux';

const myStore = createStore(allReducers,
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__());

ReactDOM.render(
  <React.StrictMode>
    <Provider store={myStore}>
        <App />
    </Provider>
  </React.StrictMode>,
```

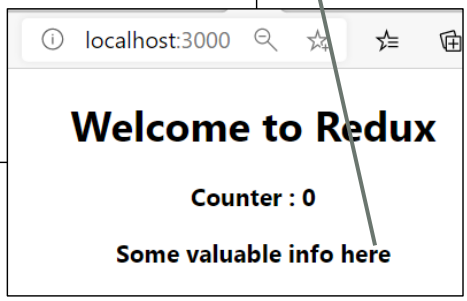
```
const counterReducer =
(state=0, action) => {…}
```

App.js : displaying state

Allows you to extract data from the Redux store state, using a selector function.

```
import './App.css';
import {useSelector} from 'react-redux';

 function App() {
   const counter = useSelector(state => state.counter);
   const isLogged = useSelector(state => state.isLogged);


  return (
    <div className="App">
      <h1>Welcome to Redux</h1>
      <h3>Counter : {counter}</h3>
       {isLogged ? <h3> Some valuable info here</h3> : ''}
    </div>
  );
}

export default App;
```

I set the isLogged state to true



Modifying state

```
//actions/index.js
export const increment = () => {
      return {
        type :'INCREMENT'
      }
    }
 export const decrement = () => {
      return {
        type :'DECREMENT'
      }
    }
```

```
import './App.css';
import {useSelector, useDispatch} from 'react-redux';
import {increment} from './actions';
import {decrement} from './actions';

 function App() {
   const counter = useSelector(state => state.counter);
   const isLogged = useSelector(state => state.isLogged);
   const dispatch = useDispatch();

  return (
    <div className="App">
      <h1>Welcome to Redux</h1>
      <h3>Counter : {counter}</h3>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
       {isLogged ? <h3> Some valuable info here</h3> : ''}
    </div>
  );
}
export default App;
```