JS Basic Notes

1. Creation and Early Development (1995)

- **Brendan Eich** at Netscape Communications created JavaScript in just 10 days in May 1995. Initially, it was called **Mocha**.
- It was quickly renamed **LiveScript** before being finally branded as **JavaScript** to capitalise on the popularity of Java, despite the two languages having very different purposes and architectures.

2. Netscape Navigator and Early Adoption (1995-1996)

- JavaScript was first introduced in Netscape Navigator 2.0 in September 1995.
- Microsoft saw the potential of JavaScript and created a similar language called **JScript**, which was included in Internet Explorer 3.0 in August 1996.

3. Standardization (1997-1999)

- To avoid conflicts between different language implementations, Netscape submitted JavaScript to **ECMA International**.
- The first edition of the ECMAScript standard (ECMA-262) was published in June 1997.
- ECMAScript 2 was released in 1998 to keep the standard aligned with the ISO/IEC 16262 international standard.
- ECMAScript 3, released in 1999, introduced regular expressions, better string handling, new control statements, try/catch exception handling, and more.

4. Divergence and Browser Wars (2000-2005)

- During the early 2000s, web developers faced challenges due to browser incompatibilities, with JavaScript implementations differing across browsers.
- Despite this, JavaScript continued to grow in popularity, partly due to the rise of dynamic web pages and the need for interactive web content.

5. The Rise of AJAX (2005)

- In 2005, **Jesse James Garrett** coined the term **AJAX (Asynchronous JavaScript and XML)**, which revolutionized web development by allowing web pages to be updated asynchronously without reloading the entire page.
- This led to the development of more dynamic and responsive web applications, exemplified by Google Maps and Gmail.

6. Modern JavaScript and ECMAScript 5 (2009)

- ECMAScript 4 was a major update that was never fully realized due to its complexity and scope, leading to a scaled-back version.
- **ECMAScript 5 (ES5)** was released in December 2009, introducing features like strict mode, JSON support, improved object property definitions, and more.

7. The Era of ECMAScript 6 (ES6) and Beyond (2015-Present)

- **ECMAScript 6 (ES6)**, also known as **ECMAScript 2015**, was a significant update released in June 2015. It brought many new features, including:
    - **Arrow functions**, **classes**, **modules**, **let** and **const** keywords, **template literals**, and **promises**.
- Since ES6, new versions of ECMAScript have been released annually, each bringing incremental improvements and new features:
    - **ES7 (2016)**: Added `Array.prototype.includes` and the exponentiation operator (`**`).
    - **ES8 (2017)**: Added async/await, Object.entries(), Object.values(), and more.
    - **ES9 (2018)**: Introduced rest/spread properties, asynchronous iteration, and more.
    - **ES10 (2019)**: Brought Array.flat(), Array.flatMap(), Object.fromEntries(), and more.
    - **ES11 (2020)**: Added the nullish coalescing operator (??), optional chaining (?.), and more.
    - **ES12 (2021)**: Introduced logical assignment operators, numeric separators, and more.
    - **ES13 (2022)**: Brought top-level await, RegExp match indices, and more.

8. JavaScript Today

- JavaScript is now a versatile, high-level language that runs on both the client side and server side (thanks to Node.js).
- It's used in a variety of environments, from web browsers to servers to mobile devices.
- The ecosystem includes a vast number of frameworks and libraries such as React, Angular, Vue.js, and many more, making JavaScript one of the most important and widely-used programming languages in the world today.

ES6 (ECMA Script 2015) : ECMA stands for European Computer Manufacturers Association – It is the 6th version of ECMA script programming language. It introduced many new function to JS language like – Arrow functions, Classes, Modules, Template string, Default Parameters, Promises, Generators.

JavaScript is a high-level, interpreted scripting language designed to make web pages interactive. It is also used on the server-side with environments like **Node.js**. JavaScript allows developers to implement complex features on web pages, including dynamic content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc.

How to Execute JS?

- JS can be executed right inside one's browser. You can open the JS console and start writing JS there.
- Another wat to execute JS is a runtime like Node.js which can be installed and used to run JS code.
- Yet another way to execute JS is by inserting it inside <script> tag of an HTML document.

Note - JS is a dynamically typed language because it allow to change any variable type at the run time.

Advantages of JavaScript:

1. **Versatility:** JavaScript runs on nearly every platform, browser, and device, making it universally compatible.
2. **Speed:** Being an interpreted language, it can be executed directly in the browser, leading to faster user experience as it doesn't require compilation.
3. **Simplicity:** JavaScript is relatively easy to learn and implement.
4. **Interoperability:** It can be used alongside other languages and is supported by all modern web browsers.
5. **Rich interfaces:** With libraries and frameworks, developers can create rich interfaces with less effort.
6. **Community and ecosystem:** A vast community and a large selection of frameworks and libraries enhance its functionality.

Disadvantages of JavaScript:

1. **Client-side security:** Because the code executes on the user's machine, it can be exploited for malicious purposes.

2. **Browser support variability:** Different browsers interpret JavaScript differently, leading to inconsistencies in user experience.
3. **Performance limitations:** For complex or intensive tasks, it might not perform as well as lower-level languages.

The value of a JS variable can be changed during the execution of the program.

Var a = 7;

Let a = 7; (Declaring Variables)

*Here, a is **Identifier** and 7 is **literal** with assignment operator.*

Rules for choosing variables names:

- Letters, digits, underscores & $ sign allowed.
- Must begin with a $, _ , or a letter.
- JS reserved words cannot be used as a variable name. // let var = 7; (this will not work)
- Ronak & ronAK are different variables. (Case Sensitive)

Difference Between var & let in JS:

- Var is globally scoped while let and const are block scoped.
- Var can be updated & re-declared within its scoped.
- Let can be updated but not re-declared.
- Const can neither be updated nor be re-declared.
- Var variables are initialized with undefined whereas let and const variable are not initialized.
- Const must be initialized during declaration unlike let and var.

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;
    console.log(x);
  }
  console.log(x);
}
function letTest() {
```

```
    let y = 1;
    if (true) {
        let y = 2;
        console.log(y);
    }
    console.log(y);
}
varTest();
letTest();
```

Types of JavaScript:

JavaScript can be included in web pages in two ways: Internally and Externally.

- **Internal JavaScript** is written directly within the HTML document using <script> tags. This approach is useful for small scripts or scripts that are tightly coupled with a particular page's content.

  Example of Internal JS:

```
<!DOCTYPE html>
<html>
<head>
    <title>Internal JavaScript Example</title>
</head>
<body>

<div id="demo">Hello, World!</div>
<button onclick="changeText()">Change Text</button>

<script>
function changeText() {
    document.getElementById("demo").innerHTML = "Welcome to JavaScript!";
}
</script>

</body>
</html>
```

In this example, the **<script>** tag contains a JavaScript function called **changeText()** that changes the inner HTML of the ** ** with id **demo** when the button is clicked.

- **External JavaScript** involves writing the JavaScript code in separate .js files and then including these files in the HTML document using the <script src="filename.js"> tag. This method is preferred for larger projects as it promotes code reusability and makes the HTML and JavaScript easier to read and maintain.

  Example of External JS:

  HTML File-

```
<!DOCTYPE html>
<html>
<head>
  <title>External JavaScript Example</title>
</head>
<body>
        <div id="demo">Hello, World!</div>
        <button onclick="changeText()">Change Text</button>
        <script src="script.js"></script>
</body>
</html>
```

  JS File –

```
function changeText() {
  document.getElementById("demo").innerHTML = "Hello from external JavaScript!";
}
```

Introduction to Variables

Variables in JavaScript are used to store data values. JavaScript uses the var, let, and const keywords to declare variables.

- var: Historically used for variable declaration; it's function-scoped or globally-scoped.
- let: Introduced in ES6, it's block-scoped and is generally recommended over var for its clarity in scoping.

- const: Also block-scoped, but used for variables whose value should not change after assignment.

Operators in JS:

- Arithmetic : +, -, *, /, **, %, ++, --
- Assignment: =, +=,-=, *=, /=, %=, **=
- Comparison: ==, !=, ===, !==, >, <, >=, <=, ?
- Logical : &&, ||, !
- Bitwise Operators
- Ternary Operator

Basic Programming Concepts in JavaScript

Datatypes:

Primitive data types are immutable, meaning their values cannot be changed once assigned. Non-primitive data types are mutable and can be modified.

Primitive Datatypes: **Null, Number, String, Symbol, Undefined, Boolean, BigInt (NNSSBBU)**

```
let a = 11;
let b = "hello";
let c = null;
let d;
let e = true;
let f = BigInt("8374");
let g = Symbol("symbol");
```

Non-Primitive Datatype: **Object (key Value Pair)**

```
const  obj_1 =[ {
  "Ronak": true,
  "Shubh": false,
  "Love": 67,
  "Rohan": undefined
},
{
  "Ronak": true,
  "Shubh": false,
  "Love": 67,
  "Rohan": undefined
},
]


Let student1 = {
```

```
        “name” = ABC;
        “class” = 5
};
console.log(student1.name);
```

- **Scope of Variables:** Determines the accessibility of variables. Variables can be either in global scope (accessible everywhere) or local scope (accessible only within the function or block where they are declared).
- **Conditional Statements:** Allow the execution of code blocks based on certain conditions. JavaScript supports if, else if, else, and switch statements.
- **Loops:** Enable executing a block of code multiple times. JavaScript provides **for, while, do...while, for in** (Loops through the keys of an object)**, for of** (Loops through the Values of an object) loops.

Arrays:

Used to store multiple values in a single variable. JavaScript arrays are dynamic, and their elements can be accessed using indices.

Arrays in JavaScript are used to store multiple values in a single variable and offer various methods to access and manipulate these values. Below is an in-depth look at arrays and how to access their elements.

There are several ways to create an array in JavaScript:

Using an array literal is the simplest way to create an array:

```
let fruits = ["Apple", "Banana", "Cherry", “asscvg”, “dfdgd”];
```

Using the new Array () syntax:

```
let fruits = new Array("Apple", "Banana", "Cherry");
```

Accessing Array Elements:

Elements in an array can be accessed using their index (starting from 0). JavaScript also provides methods like map(), forEach(), filter(), and reduce() for more advanced iteration and manipulation of arrays.

- **Direct Index Access:** Access an element by its index, e.g., array[0].
- **Array Methods:** Use methods like array.forEach((element, index) => { /* ... */ }) to iterate over elements.

In summary, JavaScript's flexibility and compatibility make it an essential language for web development. Understanding these foundational concepts is crucial for anyone looking to become proficient in web programming.

Common Methods to Work with Arrays:

- **toString()**
- **join()**
- **push():** Adds one or more elements to the end of an array and returns the new length.

```
fruits.push("Mango"); // Adds "Mango" to the end
```

- **pop()**: Removes the last element from an array and returns that element.

```
let lastFruit = fruits.pop(); // Removes the last element
```

- **shift()**: Removes the first element from an array and returns that removed element.

```
let firstFruit = fruits.shift(); // Removes the first element
```

- **unshift()**: Adds one or more elements to the beginning of an array and returns the new length.

```
fruits.unshift("Orange"); // Adds "Orange" to the beginning
```

- **indexOf()**: Returns the first index at which a given element can be found in the array, or -1 if it is not present.

```
let index = fruits.indexOf("Banana"); // Finds the index of "Banana"
```

- **splice()**: Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

```
const numbers = [1, 2, 4, 5];
// Starting from index 2, delete 0 elements and add 3 and 3.5
numbers.splice(2, 1, 3, 3.5);
console.log(numbers); // Output: [1, 2, 3, 3.5, 4, 5]
```

- **concat()**
- **slice()**: Returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) without modifying the original array.

```
let citrus = fruits.slice(1, 3); // Creates a new array containing elements from index 1 to 2
```

Looping Over Array Elements:

To access each element in an array, you can use loops:

- **For Loop:**

```
for (let i = 0; i < fruits.length; i++) {
        console.log(fruits[i]);
}
```

- **forEach() method** provides a more concise syntax for executing a function once per each item in an array.

```
fruits.forEach(function(item, index) {
  console.log(item, index);
});
```

- **For in:** The **for...in** loop iterates over all enumerable properties of an object, including inherited enumerable properties. This loop is mostly used for **iterating over object properties.**

```
const person = { firstName: "John", lastName: "Doe", age: 30 };
        for (const key in person) {
                console.log(`${key}: ${person[key]}`);
        }
```

- **For of:** The **for...of** loop provides a simple way of iterating over iterable objects like arrays, strings, maps, NodeLists, and more. It allows you to directly access **the value of each item in the iterable.**

```
const fruits = ["Apple", "Banana", "Cherry"];
for (const fruit of fruits) {
        console.log(fruit);
}
```

Accessing Elements with map(), filter(), and reduce():

- **map()** creates a new array populated with the results of calling a provided function on every element in the calling array.

```
let lengths = fruits.map(function(item) {
  return item.length;
});
```

- **filter()** creates a new array with all elements that pass the test implemented by the provided function.

```
let longFruits = fruits.filter(function(item) {
  return item.length > 5;
});
```

- **reduce()** executes a reducer function on each element of the array, resulting in a single output value.

```
let fruitString = fruits.reduce(function(accumulator, currentValue) {
  return accumulator + ', ' + currentValue;
});
```

UNIT – II

Introduction to JSON:

JSON (JavaScript Object Notation) is a lightweight format for storing and transporting data, often used when data is sent from a server to a web page. JSON is "self-describing" and easy to understand, making it a favorite choice for APIs and configs, especially for frontend developers.

JSON Syntax

Basic Structure: JSON is structured as a combination of key/value pairs. The keys are strings, and the values can be strings, numbers, objects, arrays, booleans, or null.

Example:

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false
}
```

JSON vs. XML:

Feature	**JSON**	**XML**
Readability	More readable and concise.	Less concise, can be verbose.
Data Structure	Ideal for arrays and objects.	Supports complex hierarchical data. Includes attributes.
Parsing	Easier and faster. Native support in many languages.	Requires a parser. Generally more resource-intensive.
Interoperability	Preferred for web APIs, especially RESTful services.	Used in enterprise, legacy systems, and SOAP web services.
Metadata and Namespaces	No support for namespaces or comments.	Supports comments, namespaces, and schema definitions (XSD).
Network Performance	Generally uses less bandwidth.	Verbose syntax can lead to larger file sizes.
Use Cases	Web applications, APIs, configurations.	Document-centric applications, enterprise applications.

JSON:

- Less verbose, making it quicker to read and write.
- Easier to parse into JavaScript objects natively.
- **Example:** {"student": {"name": "John Doe", "age": 22}}

XML:

- More verbose and heavier than JSON.
- Requires an XML parser to be read by JavaScript.
- **Example:**

```
<student>
 <name>John Doe</name>
 <age>22</age>
</student>
```

Data Types:

Keys must be strings, and values must be a valid JSON data type:

JSON supports:

- Strings (e.g., **"Hello World"**),
- Numbers (e.g., **10**),
- Objects (**{"key": "value"}**),
- Arrays (**["Value1", "Value2"]**),
- Booleans (**true** or **false**),
- null.

Stringify and Parse are the methods in JS used to convert js object into string or any string into object.

JSON Stringify: JSON.stringify() is a method in JavaScript that converts a JavaScript object or value to a JSON (JavaScript Object Notation) string. This method is widely used in web development for serializing (converting to a string) complex data structures so that they can be easily transmitted over a network or stored in a file in a format that is text-based and easily readable.

Syntax:

```
JSON.stringify(value[, replacer[, space]])
```

- **value:** The JavaScript value (usually an object or array) to convert to a JSON string.

- **replacer (Optional):** Either a function that alters the behavior of the stringification process or an array of String and Number objects that serve as a whitelist for selecting/filtering the properties of the value object to be included in the JSON string.
- **space (Optional):** Specifies the indentation of nested structures. If it is omitted, the text will be packed without extra whitespace. If it is a number, it will specify the number of space characters to use as white space for indentation. If it is a string (such as '\t'), it will be used as white space.

Example:

```
const obj = {
  name: "John Doe",
  age: 30,
  isAdmin: false
};
const jsonString = JSON.stringify(obj);
console.log(jsonString);
// Output: {"name":"John Doe","age":30,"isAdmin":false}
```

Example with Array:

```
const colors = ["Red", "Green", "Blue"];
const jsonString = JSON.stringify(colors);
console.log(jsonString);
// Output: ["Red","Green","Blue"]
```

Objects and Arrays in JSON:

In JSON (JavaScript Object Notation), both objects and arrays are fundamental structures that allow you to represent data in a structured format. JSON is a lightweight data-interchange format that's easy for humans to read and write, and easy for machines to parse and generate. It's widely used for transmitting data in web applications between clients and servers.

- **Objects**: A JSON object is an unordered collection of key/value pairs. An object begins with a left brace { and ends with a right brace }. Each key is followed by a colon : and the key/value pairs are separated by commas ,. The keys are strings, and the values can be strings, numbers, objects, arrays, true, false, or null.

  Example:

```
Let Obj = {
 "name": "Viraj",
 "age": 37,
 "isEmployed": true,
        "address": {
           "street": "123 Main St",
           "city": "Anytown"
        },
 "phoneNumbers": ["8769101102", "9788453344"]
}
```

- **Arrays**: A JSON array is an ordered collection of values. An array begins with a left square bracket [and ends with a right square bracket]. Values are separated by commas. The values can be objects, arrays, numbers, strings, true, false, or null.

  Example:

```
Const Arr = [
 {
  "name": "John Doe",
  "age": 30
 },
 {
  "name": "Jane Smith",
  "age": 25
 }
]
Console.log(arr[1].name);
```

- **Using Objects and Arrays Together:** Objects and arrays can be nested inside each other. This allows you to represent complex data structures in a readable and efficient format.
  Example:

```
{
 "employees": [
  {
   "name": "John Doe",
   "age": 30,
   "skills": ["JavaScript", "React"]
  },
  {
   "name": "Jane Smith",
   "age": 25,
   "skills": ["HTML", "CSS"]
  }
 ],
 "company": "Tech Innovations",
 "founded": 2010
}
```

Integration with Server-side Languages:

JSON PHP:

The phrase "JSON PHP" combines two concepts, JSON (JavaScript Object Notation) and PHP (a popular server-side scripting language), to describe the process or technique of using PHP to generate, encode, or decode JSON data. Let's break down each component and their interaction:

JSON (JavaScript Object Notation)

- **JSON** is a lightweight data interchange format. It's easy for humans to read and write and easy for machines to parse and generate. JSON is language-independent but uses conventions familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. This makes JSON an ideal data-interchange language.
- **Structure**: JSON is built on two structures:
    1. A collection of name/value pairs (often realized as an object, record, struct, dictionary, hash table, keyed list, or associative array).
    2. An ordered list of values (more commonly known as an array, list, or sequence).

PHP

- **PHP** (Hypertext Preprocessor) is a widely-used open source general-purpose scripting language that is especially suited for web development and can be embedded into HTML. It's powerful enough to be at the core of the biggest blogging system on the web (WordPress) and deep enough to run the largest social network (Facebook). It is also easy enough to be a beginner's first server-side language.

JSON PHP Interaction

- In the context of web development, PHP is often used to handle data on the server side: accessing databases, processing data, and deciding what data to send to the client (browser). When PHP sends data to the client, it can use JSON format to serialize the data. JSON's format makes it easy to encode PHP arrays or objects into a string that can then be sent to the browser, where JavaScript (or another client-side language) can parse and use this data.
- PHP provides built-in functions for working with JSON data:
    - **json_encode()**: Converts a PHP value (array or object) into a JSON-encoded string.
    - **json_decode()**: Converts a JSON-encoded string into a PHP variable (array or object).

Example Usage

A typical use case involves a web application that needs to fetch data from a server. The server-side code (written in PHP) queries a database and gets some data that needs to be sent to the client-side (browser). PHP then uses **json_encode()** to turn the data into a JSON string and sends it to the client. The client-side code (written in JavaScript) then uses **JSON.parse()** (a native JavaScript function) to turn the JSON string back into an accessible JavaScript object or array.

Imagine you're writing letters (data) to a friend (another program) who speaks a different language. JSON is like a widely understood language that both of you agree to use for your letters, ensuring that your messages are easily understood by your friend, regardless of the language you each normally speak.

Example:

Let's say you have information about a book and you want to send this information from your PHP server to a web page where it can be displayed.

PHP Array :

```
$book = [
   'title' => 'The Great Gatsby',
   'author' => 'F. Scott Fitzgerald',
   'year' => 1925
];
```

Convert to JSON:

```
$jsonString = json_encode($book);
```

Now JSON string looks like this:

```
{"title":"The Great Gatsby","author":"F. Scott Fitzgerald","year":1925}
```

This is JSON! It's a string that represents the same data as your PHP array but in a format that's easy to send and understand by different programming languages and systems.

Use the JSON:

You can now send this JSON string from your PHP server to a web browser. The browser can use JavaScript to read this JSON string and display the book's information on a web page.

Receiving JSON in PHP:

If someone sends you JSON data, PHP can easily read it too. If you received the above JSON string, you could convert it back into a PHP array with **json_decode():**

```
$receivedJsonString = '{"title":"The Great Gatsby","author":"F. Scott Fitzgerald","year":1925}';
$bookArray = json_decode($receivedJsonString, true);
```

Now, $bookArray is back to being a PHP array, and you can work with it just like you did at the start.

In simple terms, JSON in PHP is a super handy tool for turning complex information into simple strings that can be easily shared and then turning those strings back into useful information when they reach their destination.

PHP (SERVER-SIDE):

```
<?php
        $data = array('name' => 'John Doe', 'age' => 30);
        header('Content-Type: application/json');
        echo json_encode($data);
?>
```

JS (Client Side):

```
fetch('your-php-script.php')
        .then(response => response.json()) // Parses the JSON string into a JavaScript object
        .then(data => console.log(data.name, data.age)); // John Doe 30
```

JSON HTML/JavaScript: Embed JSON within JavaScript to dynamically populate data.

Certainly! When talking about JSON and HTML together, we're often discussing how to use JSON data to dynamically generate HTML content on a webpage. This approach allows web developers to efficiently display structured data, like products in an online store, posts in a blog, or items in a to-do list, on their websites.

Example:

Imagine you have a JSON data set of book information you want to display on a webpage. Here's what the JSON might look like:

```
{
 "books": [
  {
   "title": "The Great Gatsby",
   "author": "F. Scott Fitzgerald",
   "year": 1925
  },
  {
   "title": "To Kill a Mockingbird",
   "author": "Harper Lee",
   "year": 1960
  }
 ]
}
```

Using JSON to Dynamically Generate HTML:

You can use JavaScript to parse this JSON and dynamically generate HTML content to display this information on a webpage. Here's a simple example using JavaScript:

```
<!DOCTYPE html>
<html>
<body>
        <div id="book-list"></div>
<script>
// Assuming this JSON data could be fetched from a server or is already present in your script
const jsonData = `{
 "books": [
  {
   "title": "The Great Gatsby",
```

```
    "author": "F. Scott Fitzgerald",
    "year": 1925
   },
   {
    "title": "To Kill a Mockingbird",
    "author": "Harper Lee",
    "year": 1960
   }
 ]
}`;

// Parse the JSON string into a JavaScript object
const dataObj = JSON.parse(jsonData);

// Get the div element where you want to display the book list
const bookListDiv = document.getElementById('book-list');

// Use a traditional for loop to iterate over the books array
for (let i = 0; i < dataObj.books.length; i++) {
  // Generate HTML content for each book and add it to the div's innerHTML
  bookListDiv.innerHTML += `<p>Title: ${dataObj.books[i].title}, Author: ${dataObj.books[i].author},
Year: ${dataObj.books[i].year}</p>`;
}
</script>

</body>
</html>
```

This approach allows data (like a list of books) to be dynamically inserted into a webpage's HTML content, making it visible to users. It's a powerful way to build dynamic, data-driven web applications.

Example:

```
<script>
    let data = {"name": "John", "age": 30};
    document.getElementById("demo").innerHTML = data.name;
</script>
```

JSONP (JSON with Padding)

JSONP (JSON with Padding) is a method used to request data from a server residing in a different domain than the client, circumventing the same-origin policy restrictions that are implemented in

web browsers for security reasons. The same-origin policy prevents a webpage from making requests to a different domain than the one that served the webpage, which can be a limitation when trying to access cross-domain resources or APIs.

- A technique used to overcome cross-domain requests limitations.
- Example:

```
<script>
function fetchData(json){
  console.log("Data:", json);
}
</script>
<script src="http://example.com/data.json?callback=fetchData"></script>
```

Classes:

Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are unique to classes.

Syntax :

```
Class MyClass{
        //class methods
        constructor( ) { …. }
        method 1( ) { … }
        method 2( ) { … }
}
```

Example :

```
class RailwayForm {
  submit() {
    alert(this.name + ": Your form is submitted for train number: " + this.trainno)
  }
  cancel() {
    alert(this.name + ": This form is cancelled for train number: " + this.trainno)
  }
  fill(givenname, trainno) {
    this.name = givenname
    this.trainno = trainno
  }
}

// Create a form for jiya
let jiyaForm = new RailwayForm()
// Fill the form with jiya's details
jiyaForm.fill("jiya", 145316)
```

```
// Create a forms for Rohan
let rohanForm1 = new RailwayForm()
let rohanForm2 = new RailwayForm()

// Fill the forms with Rohan's details
rohanForm1.fill("Rohan", 222420)
rohanForm2.fill("Rohan", 2229211)

jiyaForm.submit()
rohanForm1.submit()
rohanForm2.submit()
rohanForm1.cancel()
```

Constructor: In JavaScript, constructors are special functions used for creating and initializing objects created with a class. Before ES6, JavaScript used prototype-based inheritance, and constructors were defined by functions. With ES6 and later versions, the class syntax was introduced, making object creation more intuitive and closer to other object-oriented programming languages.

```
class RailwayForm {
  constructor(givenname, trainno, address) {
    console.log("CONSTRUCTOR CALLED..." + givenname + " " + trainno)
    this.name = givenname
    this.trainno = trainno
    this.address = address
  }

  preview() {
    alert(this.name + ": Your form is for Train number: " + this.trainno + " and your address is "
+ this.address)
  }
```

```
  submit() {
    alert(this.name + ": Your form is submitted for train number: " + this.trainno)
  }

  cancel() {
    alert(this.name + ": This form is cancelled for train number: " + this.trainno)
    this.trainno = 0
  }
}

let newForm = new RailwayForm("Ronak", 13488, "420, Pacific Ocean, Ocean, Bihar - 0000555")
newForm.preview()
newForm.submit()
newForm.cancel()
newForm.preview()
```

Arrow Function :

```
const addTwo = (num1, num2) => {
  return num1 + num2;
}
console.log(addTwo(3,4));
```

Implicit Return :

```
const addTwo = (num1, num2) => (num1 + num2);
console.log(addTwo(3,4))
```

If you're using { } brackets then you have to write Return keyword

And if you're using ( ) brackets then you don't have to write Return keyword

And if you want to return a object then:

```
const addTwo = (num1, num2) => ({username : “Ronak”});

console.log(addTwo(3,4);
```

Destructuring:

Destructuring in JavaScript is a convenient way to extract values from arrays or properties from objects into distinct variables. It simplifies the process of accessing and assigning the elements of arrays or the properties of objects to variables. Here's how it works for both arrays and objects:

Array Destructuring: With array destructuring, you can unpack values from array elements into distinct variables based on their position in the array.

```
// Example of array destructuring

const numbers = [1, 2, 3];


// Destructure the array

const [first, second, three] = numbers;


console.log(first); // 1

console.log(second); // 2

console.log(third); // 3
```

Object Destructuring: Object destructuring allows you to unpack values from object properties into variables. The variables can be named differently from the object properties.

```
// Example of object destructuring

const person = {

   name: 'John Doe',

   age: 30

};

console.log(name);
```

```
// Destructure the object
const { name: firstName, age: personAge } = person;
console.log(name);
console.log(personName); // John Doe
console.log(personAge); // 30

// Function using object destructuring in parameters
greet = (( name, age ) => {
    console.log(`Hello, my name is ${name} and I'm ${age} years old.`);
})

greet(person); // Hello, my name is John Doe and I'm 30 years old.
```

Array Methods :

forEach() :

Description: Executes a provided function once for each array element.

Example:

```
const fruits = ['apple', 'banana', 'orange'];
fruits.forEach(function(element, index) {
 console.log('Element at index ' + index + ': ' + element);
});
// Output: Element at index 0: apple
//         Element at index 1: banana
//         Element at index 2: orange
```

map( ) :

Description: Creates a new array populated with the results of calling a provided function on every element in the calling array.

Example:

```
const numbers = [1, 4, 9];
const roots = numbers.map(Math.sqrt);
console.log(roots); // Output: [1, 2, 3]
```

filter( ) :

Description: Creates a new array with all elements that pass the test implemented by the provided function.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(n => n % 2 === 0);
console.log(evenNumbers); // Output: [2, 4]
```

reduce( ) :

Description: Executes a reducer function on each element of the array, resulting in a single output value.

Example:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue,) => accumulator + currentValue, 0);
console.log(sum); // Output: 10
```

Spread Operators:

The spread operator (...) in JavaScript is a useful syntax that allows an iterable (like an array or string) to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected. It can also be used to spread out the properties of an object in object literals. The spread operator makes it easier to combine arrays or objects, pass elements to functions, and work with destructuring assignments.

Examples of Spread Operator:

1. **Combining Arrays**

You can use the spread operator to combine or concatenate multiple arrays without using the Array's concat() method.

```
const first = [1, 2, 3];
const second = [4, 5, 6];

// Combining first and second arrays
const combined = [...first, ...second];
console.log(combined); // Output: [1, 2, 3, 4, 5, 6]
```

2. **Copying Arrays**

The spread operator can create a shallow copy of an array.

```
const original = [1, 2, 3];
const copy = [...original];

console.log(copy); // Output: [1, 2, 3]
```

3. Function Calls

It's useful for applying an array of arguments to a function without using the apply() method.

```
function sum(x, y, z) {
  return x + y + z;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // Output: 6
```

4. **Spreading Elements Together with Individual Elements**

You can mix spread elements with individual elements in array literals.

```
const fruits = ['apple', 'banana'];
const moreFruits = ['orange', ...fruits, 'watermelon'];

console.log(moreFruits); // Output: ['orange', 'apple', 'banana', 'watermelon']
```

5. Spreading Objects

The Spread operator can be used to combine or clone objects. This is particularly useful for working with states in libraries like React.

```
const user = { name: 'John', age: 30 };

const updatedUser = { ...user, location: 'New York' };

console.log(updatedUser); // Output: { name: 'John', age: 30, location: 'New York' }
```

Note:

- When using the spread operator with objects, it only does a shallow copy. Deeply nested objects or arrays will not be deeply copied, which means nested objects or arrays will still be linked to the original reference.
- The spread operator works only with iterable objects. While you can spread elements in arrays or strings (because they are iterable), attempting to spread a non-iterable like a number would result in an error.

The spread operator simplifies various operations in JavaScript and makes code more readable and concise.

Modules in JS:

Modules in JavaScript are reusable pieces of code that can be exported from one program and imported for use in another program. Modules allow for a cleaner, more maintainable codebase by breaking up the code into separate, smaller pieces with their own context. This helps in avoiding namespace pollution, where different parts of a program might accidentally overwrite variables or functions with the same names.

Key Concepts of JavaScript Modules:

- Export: Modules can export functions, objects, or variables to be used in other files.
- Import: Other modules can import these functions, objects, or variables to use them.

Benefits of Using Modules:

1. Maintainability: By dividing a program into smaller, reusable components, modules make it easier to maintain and update the code.
2. Namespace Management: Modules help avoid global namespace pollution, reducing the chance of name conflicts.
3. Reusability: Modules allow developers to reuse code across different parts of a program or in different projects.
4. Dependency Management: Modules make it easier to track and manage the dependencies between different parts of a program.

How Modules Work in JavaScript:

JavaScript implements modules through two main syntaxes:

- ES6 Modules: The most recent standard for working with modules in JavaScript, using import and export statements.
- CommonJS Modules: CommonJS is primarily used in Node.js for server-side code. It uses require() to import modules and module.exports to export them.

Difference Between CommonJS & ES6 Modules:

Feature	**CommonJS**	**ES6 Modules**
Syntax	require() and module.exports	import and export statements
Loading Mechanism	Synchronous	Asynchronous
Module Loading	Dynamic loading is possible	Static structure; supports tree shaking
Execution Time	At runtime	Ahead of time (during parsing)
Interoperability	Designed for Node.js	Designed for web browsers; also supported in Node.js
Module Object	Modules are cached; can modify exports at runtime	Live read-only view of exports
Use Cases	Server-side development in Node.js	Front-end development; efficient for web applications
Advantages	- Easy to use in Node.js - Supports conditional loading	- Supports tree shaking for smaller bundles - Can load modules in parallel
Compatibility	Native to Node.js	Native to modern browsers; Node.js with .mjs or "type": "module"

DOM:

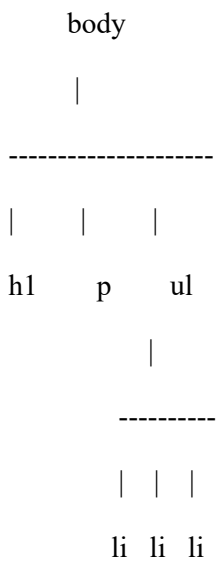
Imagine the Document Object Model (DOM) in JavaScript as a family tree of a webpage. Just like a family tree, where you have grandparents, parents, children, and so on, the DOM creates a structured representation or "tree" of all the elements on a webpage. Each element (like a paragraph, a link, or an image) is like a person in this family tree, and they are related to each other in a hierarchical manner.

Let's use a simple example: a webpage with a headline, a paragraph, and a list of items. In the family tree (or the DOM), the <body> tag is like the parent that contains three children - the headline (<h1>), the paragraph (<p>), and the list (<ul>). The list (<ul>) has its own children, the list items (<li>).

WebPage of HTML lookslike :

```
<body>
  <h1>My Simple Page</h1>
  <p>This is a paragraph about the page.</p>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</body>
```

Structure of WebPage:



Manipulating the DOM with JavaScript:

Now, let's say we want to change the text of the paragraph to something else, add another item to the list, and change the color of the headline. Here's how you could do it with JavaScript, which is like giving instructions to modify the family tree:

1. **Change the paragraph text:** Find the paragraph person in the tree and tell them to say something else.
   `document.querySelector('p').textContent = "This paragraph has been changed.";`

2. **Add another item to the list:** Find the list (<ul>) in the tree and add another child (<li>) to it with the text "Item 4".
   ```
   let newItem = document.createElement('li'); // Create a new list item
   newItem.textContent = 'Item 4'; // Set its text content
   document.querySelector('ul').appendChild(newItem); // Add it as a child to the list
   ```

3. **Change the headline color:** Find the headline (<h1>) in the family tree and change its style to make the text color blue.
   ```
   document.querySelector('h1').style.color = 'blue';
   ```

1. Introduction to DOM (Document Object Model)

Think of the DOM as a family tree of your webpage. Each part of your webpage (like paragraphs, headings, images) is a member of this family, and they relate to each other as parents, children, or siblings. The DOM lets JavaScript talk to this family tree, which means it can change the text on a button, add images, or even react to things the user does, like clicking or typing.

Example: Imagine your webpage is a simple tree:

- The whole page is the "root" of the tree.
- This root might have branches like "header", "main section", and "footer".
- Each of these branches can have smaller branches, like a header having a logo and navigation links.

2. Accessing Elements of HTML using DOM

Accessing elements is like finding a specific member in our family tree. For example, if you want to find a paragraph and change its text, you first need to locate it. JavaScript has tools to do exactly this, similar to how you might use a name or a description to find a person in a family reunion.

Example: If you have a paragraph in your HTML page marked by an ID "intro":

HTML:

```
<p id="intro">This is an introduction.</p>
```

You can find this paragraph in JavaScript by its ID and change its text:

JS:

```
document.getElementById('intro').textContent = 'Updated introduction text.';
```

3. Various Functions for Handling DOM

Once you've found the elements, you can do different things with them, like changing their text, adding styles, or making them react to clicks. JavaScript gives you many "tools" to manipulate these elements, just like a set of tools to repair or decorate a house.

Examples:

- **Creating a new element:** Let's say you want to add a new notice to your page.

JS:

```
var newParagraph = document.createElement('p').textContent = 'This is a new paragraph.';
document.body.appendChild(newParagraph);
```

- **Adding a class for styling:** If you want this new paragraph to be bold and italic.

JS:

```
newParagraph.className = 'bold italic';
```

4. Event Handling

Event handling is like setting up notifications for specific actions, such as when someone clicks a button or moves their mouse over a link. You can set up "listeners" that wait for these actions and then do something in response, like showing a message or changing what's displayed on the screen.

Example: If you have a button on your page and you want to show a message when it's clicked:

HTML:

```
<button id="myButton">Click me!</button>
```

JS:

```
let button = document.getElementById('myButton');
button.addEventListener("click", ( ) => alert('Thanks for clicking!'));
);
```

Another Explanation of Event Handling

Event Handling

Event handling in JavaScript is a fundamental aspect of interactive web development. It involves listening for and responding to events in the browser, such as clicks, key presses, mouse movements, form submissions, and more.

- **Selecting Elements:** First, select the HTML element you want to add the event listener to. This can be done using methods like getElementById, getElementsByClassName, querySelector, etc.
- **Adding Event Listeners:** Once you've selected an element, you can add an event listener to it. The addEventListener method is used for this purpose. It takes two main arguments: the name of the event to listen for and a function to call when the event occurs.

- **Defining Event Handler Functions:** This function, often called an "event handler" or "callback", defines what actions should be performed when the event occurs.

Let's consider a simple example where we have a button in HTML, and we want to display an alert when the button is clicked.

HTML File:

```
<!DOCTYPE html>
<html>
<head>
    <title>Event Handling Example</title>
</head>
<body>
    <button id="myButton">Click Me!</button>
    <script src="script.js"></script>
</body>
</html>
```

JavaScript File:

```
// Select the button element
var button = document.getElementById('myButton');
// Add a click event listener to the button
button.addEventListener('click', function() {
    // This function is the event handler
    alert('Button was clicked!');
});
```

When the button is clicked, the function provided to addEventListener executes, triggering an alert.

Example of DOM Manipulation:

HTML File:

```
<!DOCTYPE html>
<html>
    <head>
        <title>DOM Manipulation Example</title>
    </head>
    <body>
        <h1 id="main-heading">Hello, World!</h1>
        <button id="change-color">Change Color</button>
```

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

JavaScript File:

```
document.addEventListener('DOMContentLoaded', function() {
  // Select the button by its ID
  var button = document.getElementById('change-color');

  // Add a click event listener to the button
  button.addEventListener('click', function() {
    // Select the heading by its ID
var heading = document.getElementById('main-heading');
```

5. Ways of Handling the Events

There are a few different ways to listen for and respond to events:

- **Inline in HTML:** Directly in the HTML element, you can add an action.
  `<button onclick="alert('Clicked directly from HTML!')">Click me!</button>`

- **In JavaScript:** You can assign an event handler directly to the element or use a listener for more flexibility.

```
// Assigning directly
document.getElementById('myButton').onclick = function() { alert('Clicked!'); };
// Using a listener
document.getElementById('myButton').addEventListener('click', function() { alert('Clicked!');
});
```

6. Dynamic Handling of DOM Elements

This means making changes to the webpage on the fly, while it's being viewed. For instance, updating a news feed, changing colors based on user choices, or even adding new menu items based on what a user does.

Example: Imagine a user selects a theme from a dropdown, and you change the background color based on their choice:

JS:

```
var themeSelector = document.getElementById('themeSelector');

themeSelector.addEventListener('change', function() {

 document.body.style.backgroundColor = themeSelector.value;

});
```

7. Dynamic Creation of Different Elements of DOM

Creating elements dynamically is like building parts of a house while someone is already living in it. For instance, adding new photos to a gallery based on what users upload, or creating a list of search results based on what a user types.

Example: If you're making a to-do list where users can add items:

HTML:

```
<button id="addButton">Add Item</button>
<ul id="todoList"></ul>
```

Here's how you might add new items to the list:

JS:

```
document.getElementById('addButton').addEventListener('click', function() {

var newItem = document.createElement('li');

 newItem.textContent = 'New Todo Item';

 document.getElementById('todoList').appendChild(newItem);

});
```

****

1. Creating an HTML Element

To create an HTML element using JavaScript, you can use the document.createElement() method. This method creates the HTML element but does not add it to the page; you need to append it to an existing element for it to appear in the document.

Example:

// Create a new paragraph element var newParagraph = document.createElement("p"); // Add some text to the new paragraph newParagraph.textContent = "Hello, this is a new paragraph!"; // Append the new paragraph to the body of the document document.body.appendChild(newParagraph);

Explanation:

- **document.createElement("p") creates a new paragraph (<p>) element.**
- **newParagraph.textContent sets the text inside the paragraph.**
- **document.body.appendChild(newParagraph) adds the new paragraph to the end of the document body.**

2. Fetching HTML Elements

To fetch HTML elements, JavaScript provides several methods such as document.getElementById(), document.getElementsByClassName(), and document.querySelectorAll(). These methods allow you to select elements using their ID, class, or any CSS selector, respectively.

Example:

javascriptCopy code

// Get an element by its ID var header = document.getElementById("header"); // Get elements by class name var buttons = document.getElementsByClassName("btn"); // Get elements using a CSS selector var navItems = document.querySelectorAll("nav > ul > li");

Explanation:

- **document.getElementById("header") selects the element with the ID header.**
- **document.getElementsByClassName("btn") selects all elements with the class btn.**
- **document.querySelectorAll("nav > ul > li") selects all <li> elements that are direct children of <ul> which is a direct child of <nav>.**

3. Modifying HTML Elements

Once you have a reference to an HTML element, you can modify its properties and attributes. This includes changing its text content, style, and adding event listeners.

Example:

javascriptCopy code

// Fetch an element var title = document.getElementById("mainTitle"); // Modify its text content title.textContent = "Updated Title Text"; // Change its style title.style.color = "red"; // Add an event listener title.addEventListener("click", function() { alert("Title was clicked!"); });

Explanation:

- **title.textContent updates the text of the title element.**
- **title.style.color changes the color of the text to red.**
- **title.addEventListener() adds a click event listener that displays an alert when the title is clicked.**

4. Deleting HTML Elements

To remove an HTML element, you first need to select it and then use the remove() method or removeChild() method from its parent.

Example:

javascriptCopy code

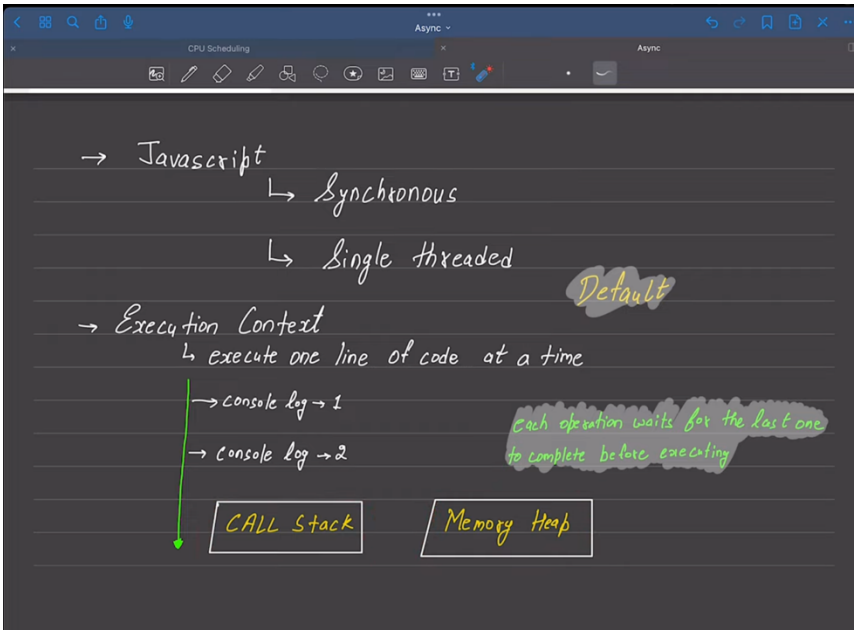
// Fetch an element var oldParagraph = document.getElementById("oldParagraph"); // Remove the element oldParagraph.remove(); // Modern approach // Alternatively, remove an element from its parent var parent = oldParagraph.parentNode; parent.removeChild(oldParagraph); // Older approach that works in older browsers

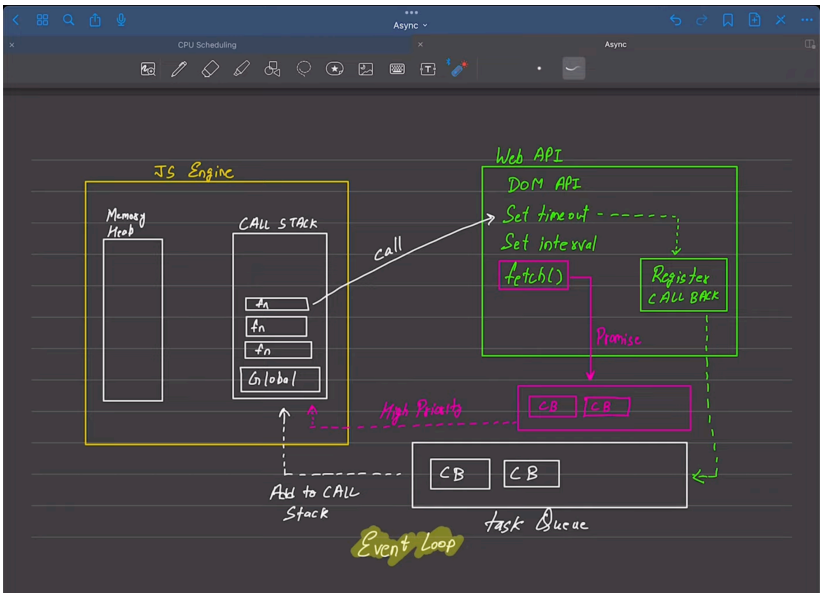
Explanation:

- **oldParagraph.remove() directly removes oldParagraph from the document.**
- **parent.removeChild(oldParagraph) removes oldParagraph from its parent element parent. This is a safer method if you're not sure about support for the remove() method in older browsers.**

****

Introduction to Async Programming:





Asynchronous programming in JavaScript is a technique that enables the execution of long-running operations without blocking the main thread, which is crucial for maintaining a responsive user interface. Before delving into the specifics, it's important to understand that JavaScript is single-threaded, which means it can only execute one operation at a time. Asynchronous programming allows JavaScript to handle tasks such as I/O operations, fetching data from an API, or any operation that might take time to complete, in the background.

Why Use Async Programming?

In a synchronous programming model, tasks are completed one after the other, i.e., each task must wait for the previous one to complete before it can begin. This model is straightforward but can lead to performance issues when tasks are long or depend on external resources like network requests or file IO.

Asynchronous programming helps overcome these drawbacks by:

1. **Improving application responsiveness**: By not blocking the main thread while waiting for tasks to be completed.
2. **Better resource utilization**: By enabling concurrency, making it is possible to handle multiple operations at the same time.
3. **Handling I/O more efficiently**: Particularly useful in network operations or large file processing.

JavaScript Methods for Background Tasks

JavaScript, particularly in the web context, provides several methods to handle tasks asynchronously, ensuring the user interface remains responsive. Here are the main techniques:

1. Callbacks

A callback is a function passed into another function as an argument to be executed after a certain event or condition. This was the earliest method to handle asynchronous operations in JavaScript.

Example:

```
function fetchData(data, callback) {

}

fetchData((data) => {

   console.log(data); // Data loaded

});
```

2. Promises

A Promise is an object representing the eventual completion or failure of an asynchronous operation. Promises provide a cleaner, more manageable approach to handling async operations compared to callbacks.

Example:

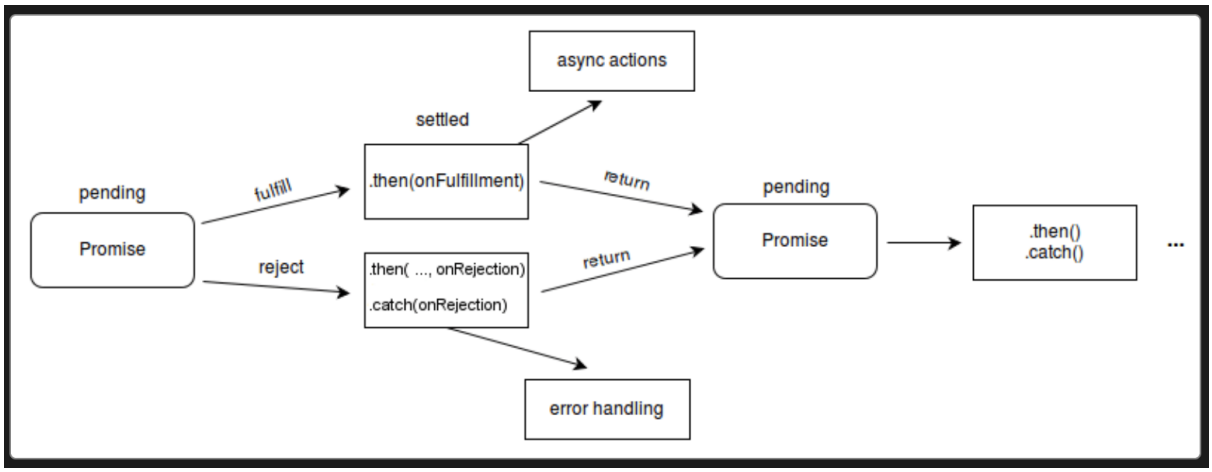
```
function fetchData() {

   return new Promise((resolve, reject) => {

      setTimeout(() => {
```

```
        resolve("Data loaded");
      }, 1000);
    });
}

fetchData().then(data => {
    console.log(data); // Data loaded
}).catch(error => {
    console.error(error);
});
```



3. Async/Await

Async/await is built on top of promises. It provides a more synchronous-looking syntax to manage asynchronous code, which makes it easier to read and maintain.

Example:

```
async function fetchData() {
    return await new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Data loaded");
        }, 1000);
```

```
    });
}
async function loadData() {
    const data = await fetchData();
    console.log(data); // Data loaded
}

loadData();
```

Timeout and Interval

setTimeout: Used to execute a block of code or a function after a specified delay. It's a way to schedule a single operation.

```
// Example: Log a message after 2 seconds
setTimeout(() => {
  console.log("This message appears after 2 seconds!");
}, 2000);
```

setInterval: Used to execute a block of code or function repeatedly at specified intervals.

```
// Example: Log a message every 3 seconds
const intervalId = setInterval(() => {
  console.log("This message appears every 3 seconds!");
}, 3000);

// To clear the interval, you can call clearInterval(intervalId);
```

Async and Await

async/await: These keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
// Example: Fetch data from an API
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error fetching data: ", error);
  }
}

fetchData();
```

Promises and Handling Promises

- **Promises:** Objects that represent the eventual completion or failure of an asynchronous operation and its resulting value.

```
// Example: A simple promise that resolves after 2 seconds
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data retrieved");
  }, 2000);
});

myPromise.then(data => {
  console.log(data); // "Data retrieved" after 2 seconds
}).catch(error => {
```

```
  console.error(error);
});
```

- **Handling Promises:** You can attach **.then()** for handling resolved promises and **.catch()** for handling rejected ones.

```
// Chaining then() and catch()
myPromise
  .then(data => { console.log(data); })
  .catch(error => { console.error(error); });
```

Callbacks and Exceptions

- **Callback Functions:** Functions that are passed to another function as arguments and are intended to be called at a later time to complete some kind of routine or action.

```
// Example: A simple asynchronous operation using a callback
function getData(callback) {
  setTimeout(() => {
    callback('Here is your data');
  }, 1000);
}

getData(data => {
  console.log(data); // Logs after 1 second
});
```

- **Exceptions in Callbacks:** Handling errors in callbacks typically involves passing the error as the first argument to the callback (Node.js convention).

```
// Example: Error-first callback pattern
function getData(callback) {
  setTimeout(() => {
```

```
    try {
      throw new Error('Something went wrong');
    } catch (error) {
      callback(error, null);
    }
  }, 1000);
}

getData((error, data) => {
  if (error) {
    console.error(error.message);
  } else {
    console.log(data);
  }
});
```

In callbacks, error handling is done through conventions (like the error-first callback pattern), whereas Promises and async/await have built-in mechanisms for handling errors through **.catch()** and **try/catch**, respectively, which makes them more powerful and easier to use for complex asynchronous operations.

GIT Version Control System:

Git is a distributed version control system that allows developers to track changes in source code during software development. Its features are designed to handle everything from small to very large projects with speed and efficiency.

1. Git Cycle

The Git cycle refers to the process of making changes to a repository, from modifying files to pushing those changes to a remote repository. The typical cycle includes:

- Modifying files in your working directory.
- Staging the files, indicating that you intend to commit them.

- Committing, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

2. Clone

git clone is used to copy an existing Git repository into a new directory. It is an initial step if you want to work on a project that is already stored in a Git repository. For example:

```
git clone https://github.com/user/project.git
```

This command clones the repository project.git into a local folder named project.

3. Git Log

git log displays a log of the commits in a repository's history. This command can show who made changes, when, and where. For example:

```
git log
```

It will show a list of commits, each with an associated commit ID, author, date, and commit message. You can use options to format the output, such as git log --oneline for a condensed view.

4. Diff

git diff shows the differences between files in your working directory and the staging area, or between any two sets of commits. For example:

```
git diff
```

This shows the changes in your working directory that have not been staged. If you want to see the differences between staged changes and the last commit, use:

```
git diff –staged
```

5. Commit

git commit takes your staged changes and commits them to your local repository. Each commit has a unique ID that allows you to keep a record of what changes were made, by whom, and when. For example:

```
git commit -m "Add new feature"
```

This command commits the staged changes with a message describing what was done.

6. Push

git push is used to send local repository changes to a remote repository. This makes your changes available to others. For example:

```
git push origin main
```

This pushes changes from your local main branch to the main branch on the remote repository named origin.

7. Pull

git pull updates your local line of development with updates from its corresponding remote branch. It's essentially a combination of **git fetch** and **git merge**, where Git fetches changes from the remote and then merges them into your current branch. For example:

```
git pull origin main
```

This command pulls changes from the main branch of the remote named origin and merges them into your current branch.

8. Branches

Branches in Git allow you to diverge from the main line of development and continue to work independently without affecting other work. For example, creating a new branch can be done with:

```
git branch new-feature
```

Switching to this branch before working on the features can be done using:

git checkout new-feature

Or in one step with:

git checkout -b new-feature

9. Contributions

Contributions to a project are typically made through branches and pull requests. You create a branch to work on a task, push the branch to a remote repository, and then create a pull request where project maintainers can review your work. After approval, your changes can be merged into the main branch of the project repository.

For example, after pushing a new branch to GitHub, you can open a pull request through GitHub's interface to begin the review process.

React.js

React is a popular JavaScript library developed by Facebook for building user interfaces, particularly for single-page applications where you need a fast response to user interactions.

(ReactJS is a JavaScript library (a collection of pre-written JavaScript code) used for building Complex user interfaces, primarily for Single-page applications. It allows developers to create reusable UI components.)

Single Page Applications (SPAs) are a type of web application or website that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from the server. This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application.

Component returns the element of the react

Another Definition of React with Installation Process:

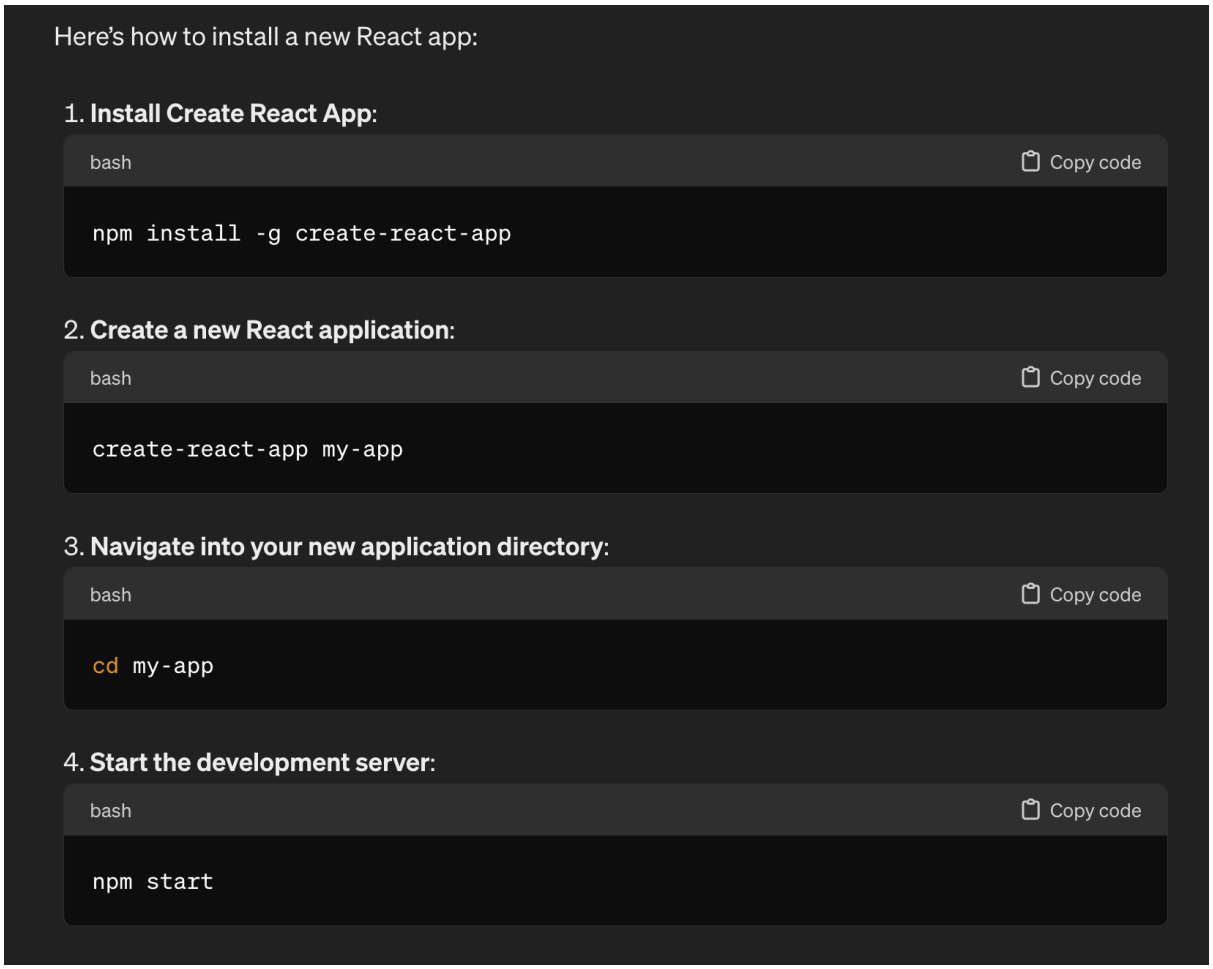
Notes by Ronak Duggar

React introduces the concept of the Virtual DOM, which offers a more efficient way of updating the user interface by re-rendering only parts of the page that need to be updated instead of the entire page. This makes it incredibly powerful for building complex, interactive web applications.

React is component-based, meaning you build encapsulated components that manage their own state, then compose them to make complex UIs. Each component has a lifecycle that you can manage according to the specific needs of your application.

Installation :

To get started with React, you need Node.js and npm (node package manager) installed on your computer. Once you have those, you can create a new React application using Create React App, which sets up everything you need for a React application.



Render HTML:

In React, you render HTML using the ReactDOM.render() method. This method takes the React element or component you want to render and a DOM node to render it into.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom';
function App() {
 return (
  <div>
   <h1>Hello, React!</h1>
  </div>
 );
}
ReactDOM.render(<App />, document.getElementById('root'));
```

This code defines a React component called App, which returns a simple div containing an h1 tag, and then renders it into a DOM element with the id root.

JSX:

JSX stands for JavaScript XML. It is a **Syntax Extension** for JavaScript recommended by React for describing what the UI should look like. JSX might remind you of a template language, but it comes with the full power of JavaScript.

JSX converts HTML tags into react elements. You are not required to use JSX, but it makes it easier to write React applications.

Here's how you use JSX to define components:

```
import React from 'react';
const App = () => {
 const greeting = 'Welcome to React';

 return (
  <div>
   <h1>{greeting}</h1>
  </div>
 );
```

```
}
```

In this example, greeting is a JavaScript variable declared in the function component App. In JSX, you can embed any JavaScript expression in curly braces {}. This expression will be evaluated as JavaScript, and its result will be rendered as part of the React DOM.

React VS ReactDOM:

1. **React:**
   - **Library for Building User Interfaces:** React is a JavaScript library developed by Facebook for building user interfaces. It is used to create reusable UI components.
   - **Declarative Components:** React allows developers to create components in a declarative style, which can improve readability and make the codebase easier to manage.
   - **State and Lifecycle:** It manages the internal state of components and handles the lifecycle processes of components, such as mounting, updating, and unmounting.
2. **ReactDOM:**
   - **DOM-specific Methods:** ReactDOM provides DOM-specific methods that can be used to interact with the DOM in a web browser. It acts as the glue between React components and the DOM.
   - **Rendering:** The main functionality of ReactDOM is to render React components to the DOM with `` `ReactDOM.render()` ``. This method takes a React element or component and a DOM container node, and renders a React tree into the DOM.
   - **Server-side Rendering:** ReactDOM also supports server-side rendering through `` `ReactDOMServer` ``, which can render components to static markup (useful for SEO and performance on initial page load).

Comparative Analysis of React & ReactDOM:

Feature/Function	React	ReactDOM
Primary Role	JavaScript library for building UI components	Provides DOM-specific methods for React
Usage	Creating and managing UI components	Rendering components to the DOM

Core Functionality	- Component creation - State management - Lifecycle methods	- Rendering React components to the web page - Server-side rendering support with ReactDOMServer
Methods and APIs	Uses JSX for defining components	ReactDOM.render() for mounting to the DOM
Integration	Integrated with other libraries and frameworks	Essential for web applications to interact with the browser
Environment	Can be used with mobile (React Native)	Primarily used in web browsers
Server-Side Rendering	Not directly involved	Provides ReactDOMServer for rendering components to HTML on the server
Performance	Focuses on efficient UI updates and virtual DOM management	Manages actual DOM updates, impacting performance especially in large applications

Why Use ReactJS?

- **Reusable Components:** Like LEGO blocks, components in React can be reused throughout the application. This makes development faster and code more manageable. For example, if you create a button component, you can use it in many places on your site.
- **Efficient and Fast:** React uses something called the 'Virtual DOM'. Imagine it as a lightweight copy of the actual webpage's structure. When something changes, React first updates the Virtual DOM. Then, it smartly updates only the parts of the real webpage that need to change. This makes the website faster and more efficient.
- **Flexible and Compatible:** React can be used with other libraries or frameworks, like Angular or Vue. This makes it a flexible choice for many developers.
- **Strong Community and Rich Ecosystem:** React is developed and maintained by Facebook. It has a large community of developers who contribute to its growth. This means lots of resources, tools, and third-party extensions are available.
- **Widely Used and Popular:** Many big companies use React, including Facebook, Instagram, and Netflix. This popularity means a lot of job opportunities for developers skilled in React.

➢ Benefits of using a front-end framework like React.js.

- **Simplifies Complex UI Development:** React.js breaks down the UI into reusable components, making it easier to manage and develop complex user interfaces.
- **Enhanced Performance:** React's Virtual DOM optimizes rendering, improving performance, especially in dynamic and interactive applications.
- **Component-Based Architecture:** Promotes reusability and modularity, leading to more organized and maintainable code.
- **React is Declarative:** This makes the code more readable and easier to debug, as you describe the UI and let React handle the actual rendering.

➢ Basic Concepts of React.js: Introduction to components, JSX, state, and props.

- **Components:** Fundamental building blocks of a React application. In React, components are like individual Lego blocks used to build a website. Each

component represents a part of the UI. You can have components for different things like a header, a footer, a button, or even a complex form.

Eg: A Button component might encapsulate all the HTML, JavaScript, and styling for a button that you can reuse wherever you need a button on your website.

- **JSX (JavaScript XML):** Syntax extension for JavaScript, allowing you to write HTML structures in the same file as JavaScript code. JSX allows you to write your UI structure in a declarative and readable way. It makes it easier to visualize the UI components while coding.

Eg: `const element = <h1>Hello, world!</h1>;`

- **State:** State is a special JavaScript object in a React component that holds data that might change over time. Think of it as a way to keep track of data that your component needs to render or use.

  State allows your components to be dynamic and interactive. For example, a state variable could keep track of whether a button has been clicked or not.

Eg: `const [count, setCount] = useState(0);`

- **Props (Properties):** Props are a way to **pass data from parent components to child** components. They are similar to arguments passed into a function.

  Props allow components to be more dynamic and reusable by giving them the ability to receive data from their parent component.

Eg: Child Component:

```
function Welcome(props) {
        return <h1>Hello, {props.name}</h1>;
}
```

Parent Component:

```
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
 return (
  <div>
   <ChildComponent name="Alice" />
  </div>
 );
```

```
}
```

Props	State
Props are immutable.	State can be changed
Props get passed to the component	State is managed within the component
Props are like the function parameter	State is like variable declaring in the function body
It can be access by props keyword in function component	It can be access by useState hook in function component

Types of Component

Function Component :

Functional components are a simpler way to write components in React. They are just JavaScript functions that take props as an argument and return React elements to describe what should appear on the screen. They don't have state or lifecycle methods. Functional components are typically used for presentational or UI-centric components.

- These are JavaScript functions that return elements (JSX) to be rendered on the page.
- They are simpler and more straightforward to write and understand.

  Eg :

```
import React from 'react';
const MyFunctionalComponent = (props) => {
  return <div>{props.message}</div>;
};

export default MyFunctionalComponent;
```

Class Component :

- These are ES6 classes that extend from **React.Component** and can hold and manage state.
- They can also contain lifecycle methods (like componentDidMount, componentDidUpdate, etc.).

  Eg :
```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Introduction to Hooks: Hooks are functions that let you use React state and lifecycle features from functional components. They enable you to use state, lifecycle methods, and other React features without writing a class.

useState: **useState** is a hook that allows functional components to manage state. It returns a stateful value and a function to update it.

Example:

```
import React, { useState } from 'react';

const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };
```

```
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};

export default Counter;
```

useEffect: **useEffect** hook adds the ability to perform side effects from a functional component. It's similar to **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount** combined in class-based components.

Example:

```
import React, { useState, useEffect } from 'react';

const DataFetcher = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data))
      .catch(error => console.error('Error fetching data: ', error));
  }, []);

  return <div>{data ? <p>{data}</p> : <p>Loading...</p>}</div>;
};
```

```
export default DataFetcher;
```

useContext: **useContext** hook allows you to access the value of a context within a functional component.

Example:

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

const MyComponent = () => {
  const value = useContext(MyContext);

  return <p>{value}</p>;
};

export default MyComponent;
```

useRef: **useRef** returns a mutable ref object whose **.current** property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

Example:

```
import React, { useRef } from 'react';

const InputComponent = () => {
  const inputRef = useRef();

  const handleFocus = () => {
    inputRef.current.focus();
  };
```

```
  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleFocus}>Focus Input</button>
    </div>
  );
};

export default InputComponent;
```

useReducer: **useReducer** is a hook for managing state in complex components. It is similar to **useState**, but instead of directly modifying the state, you dispatch actions to update it.

Example:

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
};
```

```
const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
    </div>
  );
};

export default Counter;
```

useCallback: **useCallback** memoizes a callback function and returns a memoized version of the callback that only changes if one of the dependencies has changed.

Example:

```
import React, { useState, useCallback } from 'react';

const MyComponent = () => {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return (
    <div>
```

```
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};

export default MyComponent;
```

useMemo: **useMemo** memoizes the result of a function, and re-runs the function only if one of the dependencies has changed.

Example:

```
import React, { useState, useMemo } from 'react';

const HeavyCalculationComponent = () => {
  const [count, setCount] = useState(0);

  const heavyCalculation = useMemo(() => {
    // Perform heavy calculation based on count
    return count * 2;
  }, [count]);

  return (
    <div>
      <p>Result of heavy calculation: {heavyCalculation}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};
```

```
export default HeavyCalculationComponent;
```

Creating Pages using React: In React, pages are created by composing multiple components together. Each component represents a part of the page's UI, and they can be organized and reused as needed to build complex page layouts. Typically, a React application will have a main component representing the entire page (e.g., App.js), which renders other components to create the complete UI.

Example:

```
import React from 'react';
import Header from './Header';
import Sidebar from './Sidebar';
import Content from './Content';

const MainPage = () => {
 return (
  <div>
   <Header />
   <div className="container">
    <Sidebar />
    <Content />
   </div>
  </div>
 );
};

export default MainPage;
```

Handling Events and State: In React, you can handle user events like clicks, input changes, etc., by defining event handlers in your components. State allows you to manage data that changes over time and affects the rendering of your components. By using **useState** hook or **this.state** in class components, you can manage state in React.

Example:

```
import React, { useState } from 'react';


const Counter = () => {
  const [count, setCount] = useState(0);


  const increment = () => {
    setCount(count + 1);
  };


  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
};


export default Counter;
```

Component-Driven Approach: A component-driven approach to development involves breaking down the user interface into smaller, reusable components. Each component represents a specific piece of UI or functionality, making it easier to manage and maintain your codebase. This approach promotes reusability, scalability, and easier collaboration among team members.

Smart vs Dumb Components: Smart components, also known as container components, are responsible for logic and data fetching. They manage state and pass data and actions down to dumb components. Dumb components, also known as presentational components, are concerned only with how things look. They receive data and callbacks via props and render UI elements accordingly.

State Management and Difference in Class and Function Components: In React, state management can be achieved in both class and functional components. However, class components use the **this.state** and **this.setState()** syntax for managing state, while functional components use hooks like **useState()** to manage state. Functional components with hooks offer a simpler and more concise way to manage state compared to class components.

Example of state management in functional component:

```
import React, { useState } from 'react';


const Counter = () => {
 const [count, setCount] = useState(0);


 const increment = () => {
  setCount(count + 1);
 };


 return (
  <div>
   <p>Count: {count}</p>
   <button onClick={increment}>Increment</button>
  </div>
 );
};


export default Counter;
```

Using the Components: Once the components are defined, they can be used anywhere within your application by importing and rendering them as needed. Whether it's a class component or a functional component, you can simply import it into another component or a page and include it in the JSX.

Example of using a component:

```
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <h1>My App</h1>
      <Counter />
    </div>
  );
};

export default App;
```

Event Handling in Components: Event handling in React components involves defining event handlers like **onClick**, **onChange**, etc., to respond to user interactions. These handlers are functions that get executed when a specific event occurs, such as a button click or input change.

Example:

```
import React, { useState } from 'react';

const MyComponent = () => {
  const handleClick = () => {
```

```
    console.log('Button clicked!');
  };

  return (
    <div>
      <button onClick={handleClick}>Click me</button>
    </div>
  );
};

export default MyComponent;
```

Properties in Components: Properties, also known as props, allow you to pass data from a parent component to a child component. Props are read-only, and they help make your components reusable and configurable.

Example:

```
import React from 'react';

const Greeting = (props) => {
  return <h1>Hello, {props.name}!</h1>;
};

export default Greeting;
```

Passing Values from One Component to Another: You can pass values from one component to another by using props. The parent component can pass data down to its child components via props, and child components can access this data as properties.

Example:

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const message = 'Hello from Parent';

  return <ChildComponent message={message} />;
};

export default ParentComponent;
// ChildComponent.js
import React from 'react';

const ChildComponent = (props) => {
  return <p>{props.message}</p>;
};

export default ChildComponent;
```

Lists and Keys: When rendering lists of elements in React, each item should have a unique **key** prop to help React identify which items have changed, added, or removed. Keys should be stable, predictable, and unique among siblings.

Example:

```
import React from 'react';
```

```
const MyList = () => {
  const items = ['Apple', 'Banana', 'Orange'];

  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
};

export default MyList;
```

NPM and Router: npm (Node Package Manager) is a package manager for JavaScript, and it's commonly used in React projects to install and manage dependencies. React Router is a popular routing library for React that allows you to create single-page applications with dynamic routing.

Component Lifecycle: React components have several lifecycle methods that you can override to run code at particular times in the component's lifecycle. These methods include **componentDidMount**, **componentDidUpdate**, **componentWillUnmount**, etc. However, with the introduction of hooks, most lifecycle methods have functional equivalents.

React Form: React forms allow users to input data and submit it to the server. React provides controlled components, where form data is handled by React, and uncontrolled components, where form data is handled by the DOM itself. Controlled components are usually preferred as they provide better control over form data.