Inception

Example :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <div id="root">
        <!-- used plain html -->
        <h1>Hello World from plain HTML!</h1>
    </div>

    <script>
        const heading = document.createElement("h1");
        heading.innerHTML = "Hello World from Javascript!"

        const root = document.getElementById("root");
        root.appendChild(heading)
    </script>
</body>
</html>
```

1. **How does a browser understand what Document, createElement, innerHTML things are?**
    - Document, createElement, innerHTML, all these are super powers which browsers already have in it.
    - Browsers have a javascript engine in it that executes this javascript.
2. **Does the browser understand React code?**
    - No, browsers do not understand so we have to get React into our project.
3. **How can we get React to our project?**

- ***1st way :*** CDN, just get the CDN links and put it down into our index.html file under the body tag and just before the body closing tag.
    - CDN are content delivery networks, these are the websites where this React has been hosted and we are just pulling React from there, into our project.

    OR

    - CDN is a place where that React library is hosted. So we are just fetching that React & putting it into our project.

- When we are importing these CDN links, we are importing this React code into our project, This is how we get React into our Project.
- **What are those 2 links?**
- 1st file: this is the core file of React or This is the core React framework algorithm which is written inside it.
    - React Development Javascript :-

```
<script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js">
</script>
```

- 2nd file: This is the React Library which is useful for DOM operations
    - React DOM Development Javascript :-

```
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

- **Why do we need 2 links, can't we use a single link?**
    - No, React does not only work on browsers, React also works on mobile phones as React Native. Also something called React 3d So, There are different types of React, types of places where React is used so there are different functions, methods which are being used inside React native or browsers or different places that is why there are two files.
    - The main file `react.development.js` is the core react thing and the other one is kind of like a bridge between these react and browsers So it's like React DOM.
- **Why do we need crossorigin Attribute?**
    - If you serve React from a CDN, we recommend to keep the crossorigin attribute set.
- **What is crossorigin attribute ?**
    - The "crossorigin" attribute is commonly used in web development when loading external resources, such as images, scripts, or stylesheets. It specifies how the browser should handle a cross-origin request when fetching the resource.

      When you include the "crossorigin" attribute, you can specify different values:

        1. "anonymous": This value indicates that the resource should be fetched without sending any credentials (such as cookies or HTTP authentication).
        2. "use-credentials": This value indicates that the request should include credentials, such as cookies or HTTP authentication information.

    - Using the "crossorigin" attribute can help prevent certain types of web-based attacks, such as cross-site scripting (XSS) and cross-site request forgery (CSRF), by controlling how resources from different origins are handled by the browser.
    - The crossorigin attribute in the script tag enables Cross- Origin Resource Sharing (CORS) for loading external JavaScript files from different origin than the hosting web page. This allows the script to access resources from the server hosting the script, such as making HTTP requests or accessing data.

- ***2nd way :*** Another way to get react into our app is via `NPM`.

> This is recommended way.

4. **What is React?**
    - React is a Javascript Library.
    - React comes with a philosophy of writing or manipulating the DOM using javascript or using React.
5. **What is React Element?**
    - DOM elements are HTML elements.
    - So, just like we have DOM elements, in React, we have React elements which are kind of equivalent to DOM elements.
    - React element is basically an `Object`.
    - When we `render` this element onto DOM then it becomes HTML element.
6. **What is :**

```
const heading = React.createElement("h1", {id : "heading"}, "Hello world
from React!");
```

    - ***React.createElement() :*** it is used to create React elements. The function returns a React element object, not an HTML element.
        - It takes 3 arguments
            - 1st : which element you want to create
            - 2nd : it is an object, this is the place where we will give **attributes** to our tags like id, class etc.
            - 3rd : what content you want to show inside the element.
    - The ***heading*** is neither html nor tags, it is just an object.
        - This object is a react element.
        - This react object becomes HTML that the browser understands
    - ***ReactDOM.createRoot() :*** is used to create a root for a React application where you can render your React elements.
        - We just need to create a `id = “root”` in the **index.html** file i.e

```
<div id="root"></div>
```

- Whatever we will do, we can just **render** inside the this root `root.render()`
  - Used to render React elements inside the root.
  - **root.render** function job is basically to take a React element and render it as an HTML element, inside the root element.
- This **root** is the place where all the React code will run.
- Everything we will render inside this root.

Example :

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

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>

<body>
    <div id="root"></div>

    <script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
    <script>
        const heading = React.createElement("h1", {id : "heading"},
"Hello world from React!");
        const root =
ReactDOM.createRoot(document.getElementById("root"));
        root.render(heading);
    </script>
</body>

</html>
```

7. Can we create a nested elements & siblings (more than 1 children)?
   - Yes

> But, remember one thing, it will become messy, complicated, tedious so use ***JSX*** instead of it.

for example I want to **create a nested elements** like below one :

```
<div class="parent">
    <div class="child">
        <h1>I am a child H1 Tag</h1>
    </div>
</div>
```

then code should be like this :

```
const parent = React.createElement(
"div",
{ id: "parent" },
React.createElement(
    "div",
    { id: "child" },
    React.createElement("h1", {}, "I am a child H1 Tag")
)
);
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(parent);
```

- Now, **Create a siblings** (using an array)

example :

```
const parent = React.createElement(
"div",
{ id: "parent" },
React.createElement("div", { id: "child" }, [
    React.createElement("h1", {}, "I am a child H1 Tag"),
    React.createElement("h2", {}, "I am a child H2 Tag"),
])
);
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(parent);
```

8. **What will happen if *root* has already element?**

```
<div id="root">
    <h1>Hello World from plain HTML!</h1>
</div>
```

- It will be replaced by React element but not appended.
  - Because first of all when the browser reads our HTML file so it will print ***"Hello World from plain HTML!"*** on the browser but as soon as goes to the **script** tag, it will load react into our app, then it will load **React-DOM** into our app, then it will load **app.js** into our

app & finally it will start executing code line by line which is inside the **App.js** when it will reach **root.render()**, it will just render ***parent*** inside the **root**.

> It will just replace everything inside the **root** with whatever I am passing in from **React**.

9. **Is order of files matter?**
    - Yes, it is matters. Correct order of files are:

```
<script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>

<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>

<script src="./App.js"></script>
```

10. **Will React can work inside a "div" only?**
    - Yes, React can be a big HTML Page but It will work inside a small div only.
11. **What is diference between React and ReactDOM?**
    - React is the library for building components and managing state, while ReactDOM is the package that provides the rendering capabilities to render those components to the DOM.
    - Therefore, you'll typically use React to build your components, and ReactDOM to render those components in the DOM.
12. **Can React be used without ReactDOM?**
    - No, React cannot be used without ReactDOM. ReactDOM is specifically used for rendering React components to the DOM. While React itself is responsible for managing the components and their state, it requires ReactDOM for the final step of actually rendering those components to the browser DOM. Therefore, ReactDOM is a necessary part of the React ecosystem when you are building web applications.
13. **What is parsing?**
    - Parsing means analyzing and converting a program into an internal format that a runtime environment can actually run, for example the JavaScript engine inside browsers.
    - The browser parses HTML into a DOM tree. HTML parsing involves tokenization and tree construction. HTML tokens include start and end tags, as well as attribute names and values. If the document is well-formed, parsing it is straightforward and faster. The parser parses tokenized input into the document, building up the document tree.
    - When the HTML parser finds non-blocking resources, such as an image, the browser will request those resources and continue parsing. Parsing can continue when a CSS file is encountered, but

script tags, particularly those without an ***async or defer*** attribute, blocks rendering and pauses parsing of HTML.

14. **What is async and defer attributes?**

    - Generally, when we use a script tag to load any JavaScript code, the HTML parsing is paused by the browser when it encounters the script tag and it starts to download the JavaScript file first. The HTML elements script tag will not be executed until the browser is done with downloading the script and executing it. The browser waits till the script gets downloaded, and executed, and after this, it processes the rest of the page. In modern browsers, the script tends to be larger than the HTML files, hence their download size is large and processing time will be longer. This increases the load time of the page and restricts the user to navigate through the website. To solve this problem, async and defer attributes come into play.

      Syntax: A normal script is included on the page in the following way.

      ```
      <script src = "script.js"></script>
      ```

      When the HTML parser finds this element, a request is sent to the server to get the script.

      Asynchronous: When we use the async attribute the script is downloaded asynchronously with the rest of the page without pausing the HTML parsing and the contents of the page are processed and displayed. Once the script is downloaded, the HTML parsing will be paused and the script's execution will happen. Once the execution is done, HTML parsing will then resume. The page and other scripts don't wait for async scripts and async scripts also don't wait for them. It is great for independent scripts and externally located scripts.

      Syntax:

      ```
      <script async src = "script.js"></script>
      ```

      Deferred: The defer attribute tells the browser not to interfere with the HTML parsing and only to execute the script file once the HTML document has been fully parsed. Whenever a script with this attribute is encountered, the downloading of the script starts asynchronously in the background and when the scripts get downloaded, it is executed only after the HTML parsing is finished.

      Syntax:

      ```
      <script defer src = "script.js"></script>
      ```


<script>
Script fetch
Script Execution
Parsing Paused
<script async>
Script fetch
Script Execution
Parsing Paused
<script defer>
Script fetch
Script Execution