11/02/24.

* Parse: Parsing means analyzing & converting a program into an internal format that a runtime environment can actually run, for ex JS engine inside a browser.

↳ The browser pass HTML into a DOM tree.

↳ In Script tag when we don't have async or defer attribute blocks rendering & pause parsing of HTML

↓

Rendering is the process of combining all our code according to some instruction and to translate it into HTML, code that our browser understands and displays it into the screen.

→ Script types in HTML (MDN Docs)

① async: A boolen attribute which makes sure that the script will be downloaded in parallel to the parsing of the page and it will be executed as soon as it completes the download. It does not block HTML DOM construction during downloading process.

② defer: Similar to asyn, only difference is that, it will be executed only after completing the parsing.

③ blocking: This attribute indicates that certain operations should be blocked on the fetching of the script, The operations to be block can be mentioned as

<Script blocking="render" async src="abc.js" > </script>.

" Module Script defer by-default". → Note it *.

4) cross-origin: Purpose of this attribute is to share resources from one domain to another domain, basically it is used to store CORS request. [cross origin resource sharing]

↓

It checks whether it is safe to allow for sharing the resources from other domain.

```
<script crossorigin src=" " ></script>
```

different script types.

→ learn more about it from developer.mozill.org (MDN web DOCS)

Episode-03 | Laying the foundation.

* **Scripts:** used to make npx commands more smaller, or simpler for us to start our project in development mode or in ~~built~~ production mode.

↳ Go to package.json, find Scripts, and add your script for starting the project or anything else.

↳ You can give your own name to your script.

```
"scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html"
}
```

- your own shortcut
- npx command
- you can name it anything
- entry point of our project
- script name
- package name

After adding Scripts to run them just write **npm run scriptname**

or you can also write **npm start** ↘

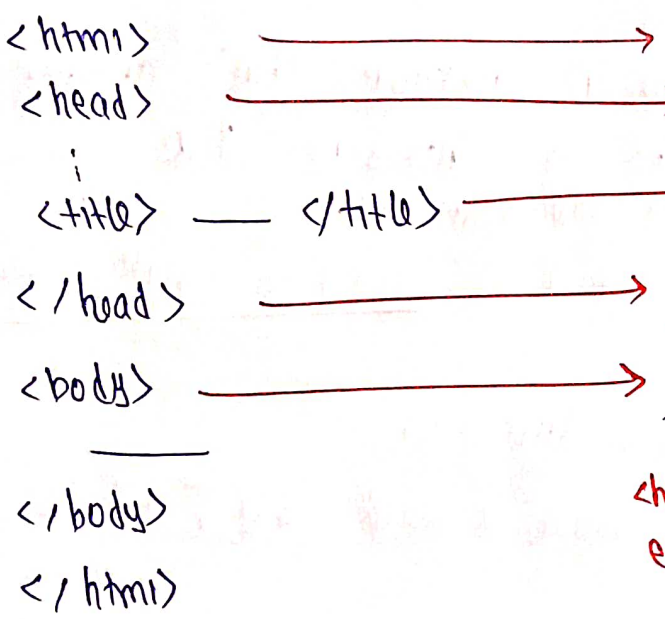
{ but it only works for start, start is a reserved keyword }

npm build ✗ won't work.

npm run start = npm start = npx parcel index.html

npm run build = npm parcel build index.html

It will only work if start & build keywords are added into Script with the their respective commands i.e, parcel index.html and parcel build index.html.



These are our dom elements. head, title, body

Dom elements are nothing but the html elements.
<h1>, <footer>, <article>, <aside>, <canvas> etc.

Note:- HTML elements are basically everything from start tag to end tag.

→ React elements are equivalent to Dom elements.

const heading = React.createElement("h1", {id: "name"}, "hiee");

(h1 → tag name; {id: "name"} → object attribute; "hiee" → children of React element)

→ React elements are just JS objects
→ when we render, it becomes or it translated into html element or we can say dom elements.
→ React dom is a JS library which basically allows react to interact with dom.
→ It is basically used to connect React with dom
→ used to manage dom elements
→ It is used to render or update react elements into dom.

① → we cannot have a single cdn link / file for both react & reactDom because in react native, React 3D, or React Art, same react is used as a component library for adding components.

React → as a component (package) library → reusable in different places such as React Native, React 3D & React Art.

That's why we keep react & react dom (packages) seperated.

② Dom is different for mobile & desktop, that's another reason why react-dom should be a different file.

★ JSX :- → JavaScript extention or JavaScript xml.

It is a JS Syntax which is easier to create ~~crea~~ React elements.

React is different & JSX is different

→ React elements can also be build without JSX, JSX is just to make it easier for us.

★ ~~JSX - just combines html & JS.~~

★ JSX is not html inside JS, It is HTML like Syntax, it just looks like html & xml.

```
const heading = <h1> Hiee </h1>
```

This is how you create react element using JSX.

```
const heading = <h1 id="heading"> Hiee </h1>
```

↕ JSX ↕

Qus for whom do we write code, for humans or machines?

Ans We write code for both, but we want our code to be understand by any other developer / human who see our code is than for machines.

if we just wanted to write code for machines, we would be coding in binary (0 or 1), because machines understand binary.

→ our JS engine understands JavaScript or more precisely our JS engine understands ECMAScript i.e, the pure JS.

→ It understands all the versions of ES i.e, Ecma Script.

→ Browsers uses JS engine.

const heading = <h1> Hiee </h1> } Browsers which uses JS engine cannot understand this, because it is not a valid JS. It will give Syntax error if you write this command into console.

Que: Then how it is working ??

Ans: Parcel is doing this behind the scene. JSX code is transpiled before it reaches the browser

converted into a language which browser understand

→ This is done by Parcel, but Parcel is not doing it by itself, Parcel gives this responsibility to Babel.

→ Babel is a JS compiler or transpiler., babel is not created by facebook.

→ Babel transpiles JSX into plain JS which our browser can understand.

(JSX) → converted into (React element) → (JS object) → HTML element

Babel is doing this

(React element) → (JS object) : same.

(JS object) → HTML element : Through rendering

★ For older versions of browsers which doesnot understands ES6 Babel transpiles it into a language which that browsers can understand.

★ → we need to import react into our JS file to use JSX in older versions of react

```
import React from "react";
```

→ tabindex : It is an attribute which can be used in any Html element. It basically specifies the sequence in which different Html elements will get focus when 'tab' button is used for navigation.

for more better understanding go and see it in w3schools.

→ camelCase → myName → This is camel case.

In html
<h1 class="Nihal"> Hie </h1>

In JSX
<h1 className="Nihal"> Hie </h1>
camel case.

In html
<h1 tableindex="2"> hie </h1>

In JSX
<h1 tableIndex="2"> hie </h1>
↓
In JSX for attributes we use camel case

→ we write class as className in JSX, because class is a reserved keyword in JS. For creating classes used in OOPS.

12/02/24 :

→ for Single line JSX.

```
const h1 = <h1> Hie </h1>
```

→ for multiple line JSX (add a bracket ())

```
const h = (<h1>
            Hie
          </h1>
);
```

we add brackets so that babel can understand from we have started JSX and where it is ending.

★React components.

→ Everything in react is a react component → Button comp. → Header → Footer etc

Types:- 1) class based components → old. → uses JS classes.

★ 2) functional components → New → uses JS functions

Just a normal JS function → name it with a capital letter

and it returns Something (JSX)

It's a react way to understand it is a react component.

→ A react functional component is a JS function which returns Sompiece of JSX.

→ JSX → React element, or we can say a function which returns a JSX or react component is a react functional component

```
const Fn = () => {
    return <h1> Nihal </h1>;
};
```

1st letter have to be capital.

React functional components.

```
const Fn = () => <h1> Nihal </h1>;
```

→ Both the components are same

```
const Fn = () => (
    <h1> class </h1>;
);
```

For using multiple lines, use curly brackets

★ root.render (heading); → Rendering a react element

★ root.render (<Heading />); → Rendering a react component

with '<' & '/>' Babel understands or differentiate between a react element & react component.

→ We can also do nesting in react functional components.

```
const Heading = () => (
    <div id="container">
        <Title />  → React component
        <h1 className="heading"> Hello </h1>
    </div>
);
```

<Title></Title> or

This is called react functional components nesting.

Also known as **component composition**

→ Instead of using arrow function we can also use a normal JS function.

```
const Title = fun () { return (<div> Hellow </div>) };
```

→ Inside any react functional component, inside { } we can write any JavaScript code.

```
const Heading = () => (
    {100+20}
    <h1> Hie </h1>
);
```

JS

output: 120 Hie

you can even write console.log here also, it will be executed

Inside { } you can write any JS code.

```
const Heading = () => (
    <div>
        {element}
        <h1> Hie </h1>
    </div>
);
```

React element

any react element is a JS object, and inside functional component in { } we can write any JS code. So if a wright any react element name inside { } it will work

```
const element = <div> Hellow </div>
```

This is a normal JS variable, So we can write it inside { }.

→ In same way we can write any react element inside any other react element or inside any react component

→ Jsx prevents Cross Site Scripting attacks.

Jsx automatically Sanitizes (removes unsafe code) before executing any Javascript file.

```
const data = api.getData();   → If any attacker sends any malicious code here.
const component = () => {data};   → Then Jsx will sanitize it & then run it.
```

When we use cdn links we do not need to write

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

↓

Because this keyword is coming from cdn links

→ But when we install react & react dom using npm then we get react & react-dom into our node-modules.

But still our code yet doesnot understands from where React keyword is coming.

☆ That's the reason why we need to import react & react-dom

```
import React from "react";
import ReactDOM from "react-dom";
```

we are importing React from a file named as React which is there in our node-module, that React keyword is not there in our code in that JS file in which we are coding. it is not there in our code!!!

That's why we get an error that React is not defined, when we don't use import.

Who downloads the node modules at server because we don't upload it?

There are several ways `node_modules` can be downloaded on a server, and who does it depends on your specific setup. Here are the most common scenarios:

1. **Automated Deployments:**

- **CI/CD Pipeline:** When you push code changes, a CI/CD pipeline (Continuous Integration/Continuous Delivery) automatically builds and deploys your application. This pipeline often includes a step to install dependencies on the server using commands like `npm install` or `yarn install`. These commands fetch dependencies from the public registry and store them in `node_modules` on the server.
- **Deployment Tools:** Specific deployment tools like Capistrano or Deploy.js might have built-in functionality to handle dependency installation during the deployment process.

2. **Containerized Environments:**

- **Docker Image:** If you use containerization with tools like Docker, your containers can include specific tools and configurations to install dependencies at startup. This could involve embedding a package manager and its configuration within the container image itself.

3. Manual Installation:

- **Rarely:** In less common scenarios, you might manually install dependencies on the server using the same `npm install` or `yarn install` commands you use locally. This approach is becoming less frequent due to the rise of automation.

Who Does It?

In most cases, **nobody needs to explicitly download or install dependencies manually** since automated processes handle it. The specific actor depends on your setup:

MODULES

Modules are designed to be reusable, which means that they can be imported and used in multiple places in an application. In contrast, plain old JavaScript code is not organized into reusable units, which makes it more difficult to reuse and maintain.

In modules we have the feature of import and export, which was not available into old js.