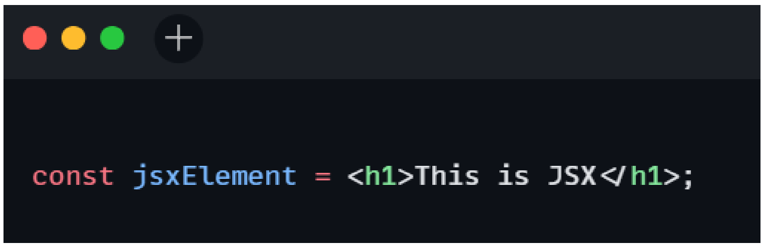
JSX

What is JSX: -

JSX (**JavaScript XML**) is a **syntax extension** for JavaScript, commonly used with React. It allows us to write **HTML-like code** within JavaScript, making it easier to describe the structure of UI components. At first glance, **JSX** looks like a mix of **HTML** and **JavaScript** because of its tag-based structure, but it offers much more functionality.

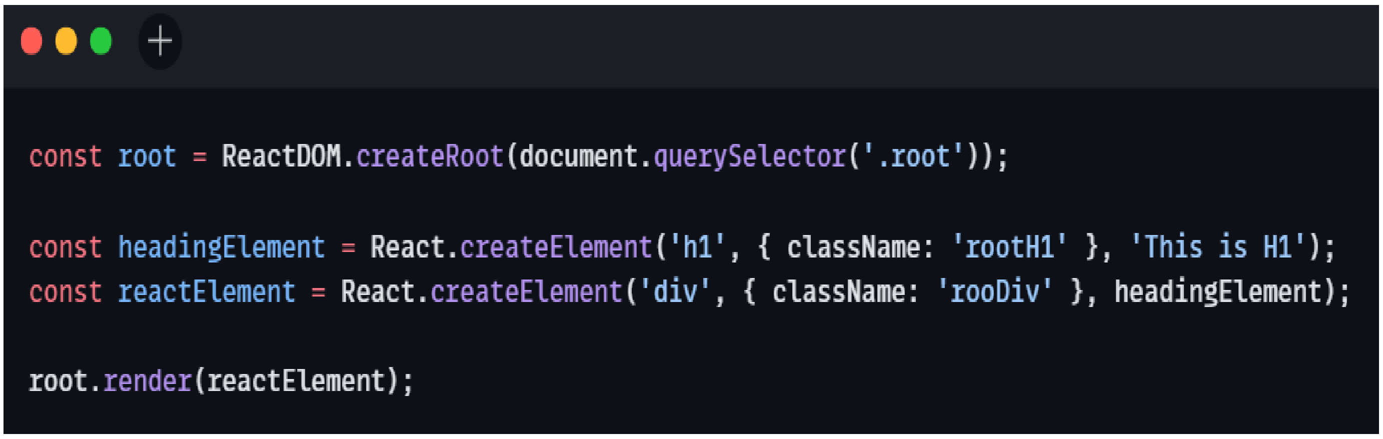
Consider the following JSX example:



```
const jsxElement = <h1>This is JSX</h1>;
```

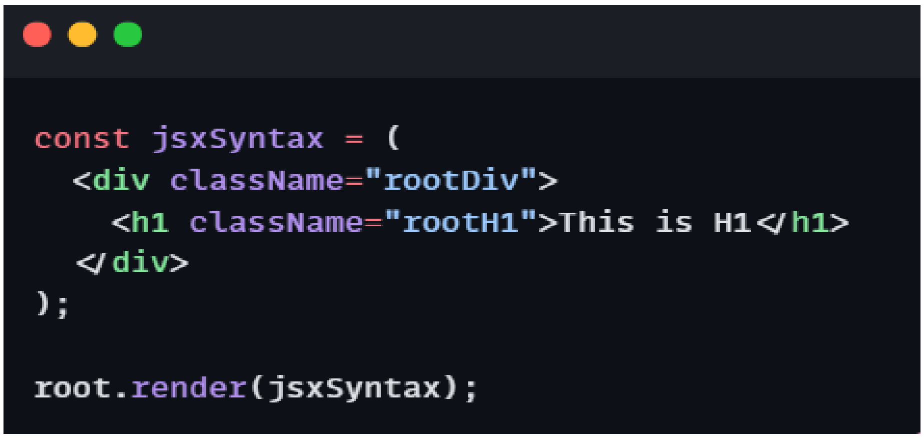
This might look like **HTML**, but it's **JSX**. **JSX** simplifies UI component creation by allowing developers to write readable and expressive code within JavaScript. This eliminates the need for separate templates or string-based methods, leading to a more integrated and efficient development experience.

Without JSX, we'd have to go through the following steps to construct an element:



```
const root = ReactDOM.createRoot(document.querySelector('.root'));

const headingElement = React.createElement('h1', { className: 'rootH1' }, 'This is H1');
const reactElement = React.createElement('div', { className: 'rooDiv' }, headingElement);

root.render(reactElement);
```

The following code is the simplified code using JSX:



```
const jsxSyntax = (
  <div className="rootDiv">
    <h1 className="rootH1">This is H1</h1>
  </div>
);

root.render(jsxSyntax);
```

Understanding JSX Syntax: -

- JSX may seem similar to HTML, but there are some key differences. For instance, while HTML uses the '**class**' attribute, JSX uses '**className**' instead due to **'class'** being a reserved word in JavaScript.
- JSX also requires all tags to be closed. For example, in **HTML**, it's acceptable to leave some tags, like the line break **<br>** or image **<img>** tag, unclosed. However, in JSX, these would need to be closed like so: **<br />** or **<img />.**
- By default, **JSX is not supported as an official syntax of JavaScript** (but it is a way to use it for React development). So, we need to make use of **babel** to **transpile** JSX syntax into plain native JavaScript using babel.
- A unique feature of JSX is its ability to embed JavaScript **expressions** within the code using **curly braces {}.** This allows for dynamic content within the UI. For example, we could embed a JavaScript **function** that returns a value directly into our JSX code.
- One of the most powerful aspects of JSX is its ability to represent components. In React, **components are reusable pieces of code that return a React element** to be rendered on the **DOM**.
- Remember that while JSX enhances **readability** and **maintainability** of our code, it's not native JavaScript and hence needs to be **transpiled** into JavaScript before it can run in a browser. Tools like **Babel** are used for this conversion process during the software build stage.

JSX as Syntactic Sugar: -

JSX is frequently referred to as "**syntactic sugar**" since it makes the creation of React components simpler. JSX is internally converted into JavaScript code using programs like **Babel**. It is changed using **React.createElement() method** into a more complex form.

JSX Advantages & Disadvantages: -

Advantages of JSX:

- JSX enhances the readability and maintainability of code by allowing developers to write **HTML-like syntax** within JavaScript.
- JSX provides the ability to create **reusable components**, leading to more modular and organized code. JSX offers better performance compared to traditional templating solutions by optimizing the rendering process.
- JSX allows for the integration of JavaScript **expressions** within the HTML-like syntax, enabling dynamic rendering of data.
- **Enhanced Security**: JSX automatically eludes dynamic material to protect users from common security flaws like **cross-site scripting** (**XSS**) **assaults**
- **Sanitizes the data:** If someone gets access to our JS code and sends some **malicious data** which will then get displayed on the screen, that attack is called **cross-site scripting**. It can read **cookies**, **local storage**, **session storage**, get **cookies**, get info about our **device**, and

read data. JSX takes care of your data. If some API passes some **malicious data.** JSX will escape it. It prevents **cross-site scripting** and **sanitizes** the data before **rendering**.

Disadvantages of JSX: -

- JSX requires an additional build step to transform the **HTML-like syntax** into valid JavaScript code that browsers can understand.
- JSX can be overwhelming for developers who are not familiar with HTML and CSS concepts.
- JSX can make debugging more challenging due to the mixing of JavaScript and **HTML-like syntax**.
- JSX may not be suitable for all projects, especially those that require **strict separation** of concerns between HTML, CSS, and JavaScript.

How Does JSX Work?

Here are the steps of how JSX functions in React:

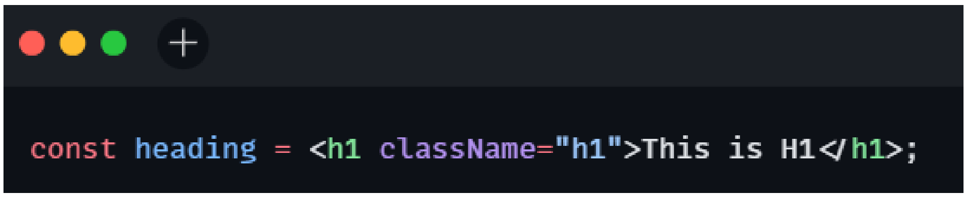
- JSX is written within JavaScript code.
- The JSX code is compiled by a **compiler** (like **Babel**) into JavaScript.
- The compiled code uses `**React.createElement()** ` to create **React elements**.
- **React elements** are converted into **JavaScript objects**.
- **JavaScript objects** are used to update the virtual **DOM**.
- React efficiently updates the actual **DOM** based on changes in the **virtual DOM**.

Is JSX mandatory for React?

JSX is not mandatory for React, but it is **highly recommended** because it provides a more **intuitive** and **declarative syntax** for defining the structure of our user interface.

JSX concepts: -

Elements: -

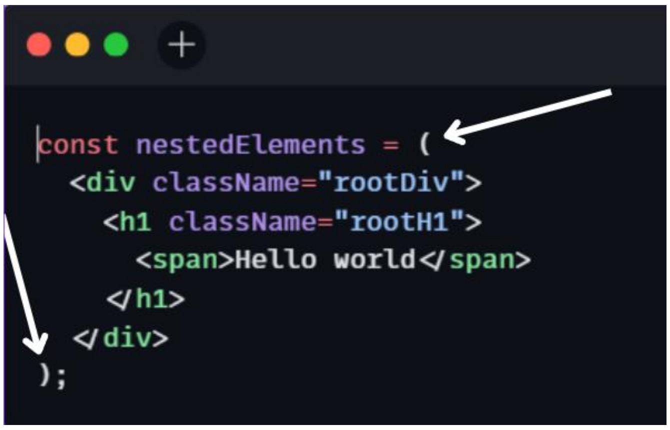


```
const heading = <h1 className="h1">This is H1</h1>;
```

JSX produces **"React elements"** which ultimately become **HTML elements** in the **DOM**.

Nested elements: -

We can also nest HTML elements similarly to how we do in regular HTML syntax, as shown below:



```
const nestedElements = (
  <div className="rootDiv">
    <h1 className="rootH1">
      <span>Hello world</span>
    </h1>
  </div>
);
```

We must **enclose** each element within a container element if we want to use more than **one element**, as shown in the figure above.

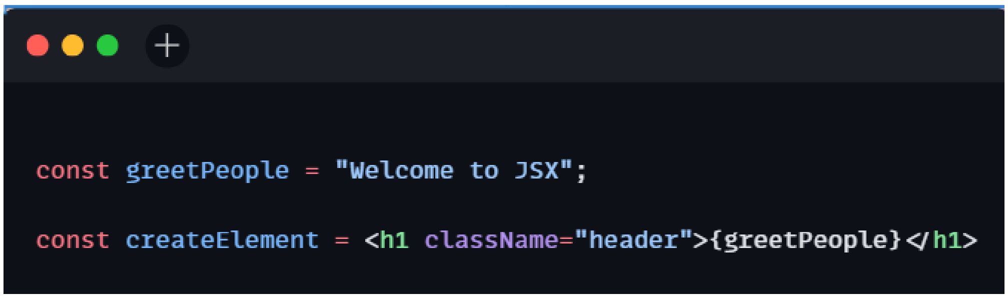
Attributes in JSX: -

Although the **'class'** attribute is commonly used in HTML, we cannot use it in **JSX** since it is rendered as JavaScript and the **'class'** keyword is a **reserved** word in JavaScript. Instead, use the className attribute.

class Vs. className

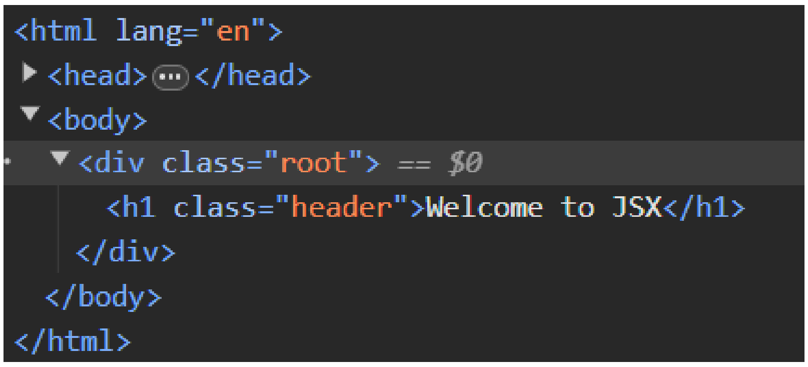
- The **'class'** property in **HTML** is used to tell the browser the **CSS** class to use. Class, on the other hand, is a **JavaScript reserved word**.
- Since JSX is compiled into JavaScript, conflicts can arise. To correct this, we use the **'className'** attribute instead of **class**. It will compile into the **HTML** class property during the **build process**. The distinction between **className** and **class** is purely to assist **JSX** in determining what type of **class** we're working with.

Let us take an example:



```
const greetPeople = "Welcome to JSX";

const createElement = <h1 className="header">{greetPeople}</h1>
```

When we check the element this, we'll get the following:

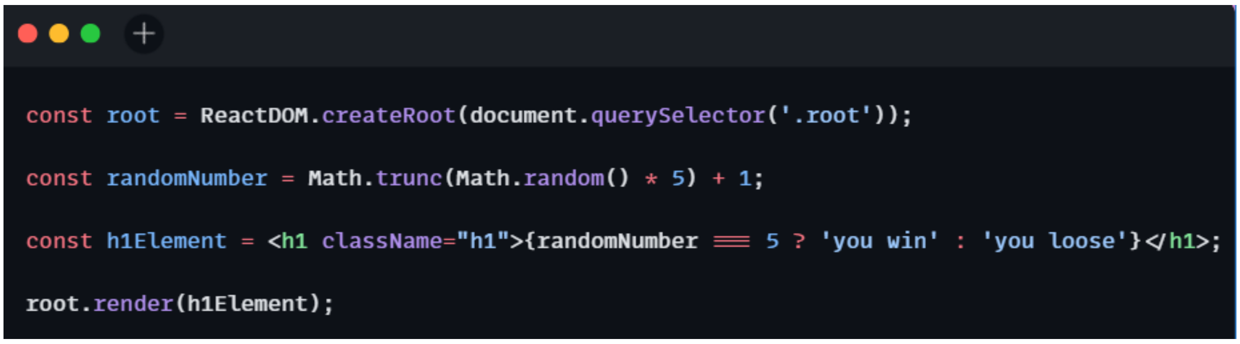


```
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▼<div class="root"> == $0
      <h1 class="header">Welcome to JSX</h1>
    </div>
  </body>
</html>
```

JSX Expressions: -

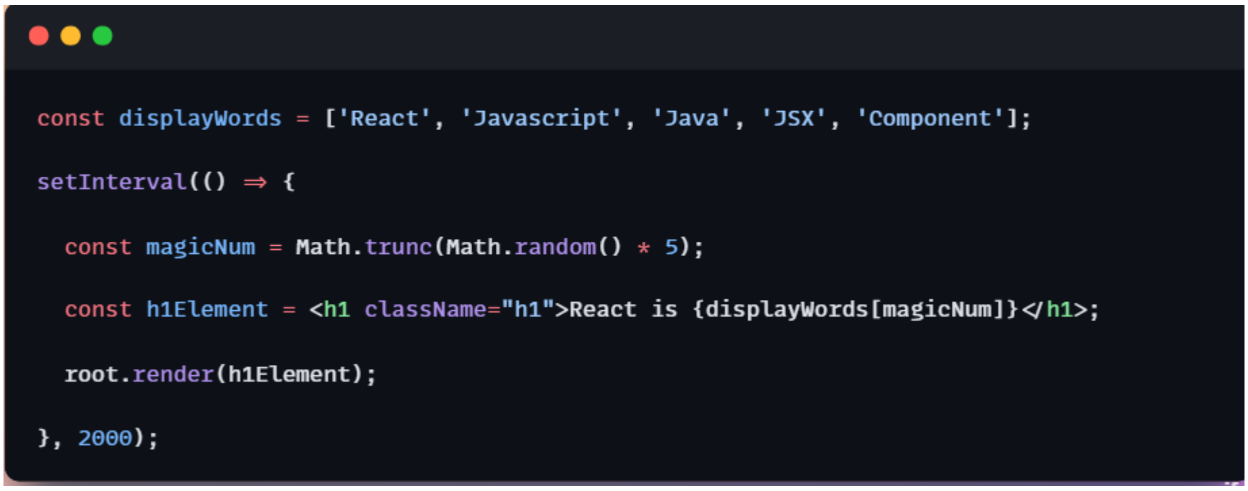
- In JSX, developers use **curly brackets** to include JavaScript expressions, making it easy to add **dynamic content** and **logic** to JSX elements.
- We can still use JavaScript expressions such as **ternary operators**, array methods like **map()** in JSX, though control flow statements like **if-else** or **loops** (**for**, **while**) are **not** used directly inside JSX. JSX is just JavaScript after compilation, so it's easier to write JavaScript along with HTML-like syntax at the same time.

Let's take a look at this example:



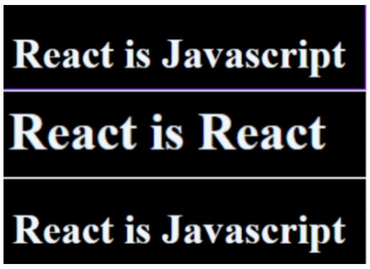
```
const root = ReactDOM.createRoot(document.querySelector('.root'));

const randomNumber = Math.trunc(Math.random() * 5) + 1;

const h1Element = <h1 className="h1">{randomNumber === 5 ? 'you win' : 'you loose'}</h1>;

root.render(h1Element);
```

Let's take a look at this another example:



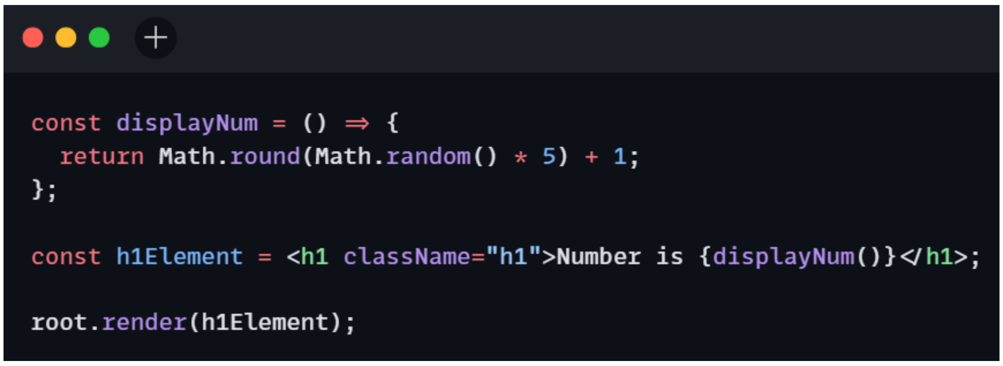
```
const displayWords = ['React', 'Javascript', 'Java', 'JSX', 'Component'];

setInterval(() => {

  const magicNum = Math.trunc(Math.random() * 5);

  const h1Element = <h1 className="h1">React is {displayWords[magicNum]}</h1>;

  root.render(h1Element);

}, 2000);
```

Output:

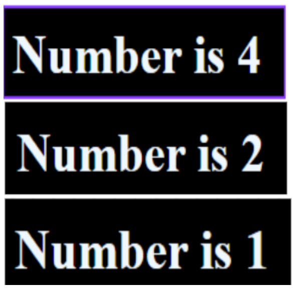


```
React is Javascript
React is React
React is Javascript
```

Another Example: - output:

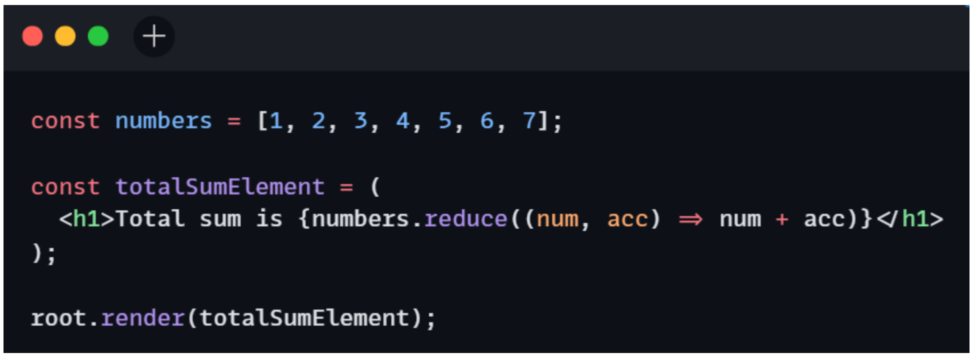


```
const displayNum = () => {
  return Math.round(Math.random() * 5) + 1;
};

const h1Element = <h1 className="h1">Number is {displayNum()}</h1>;

root.render(h1Element);
```

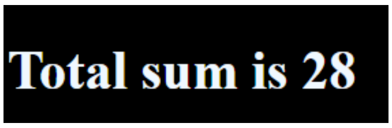


```
Number is 4
Number is 2
Number is 1
```

Another Example: - output: -



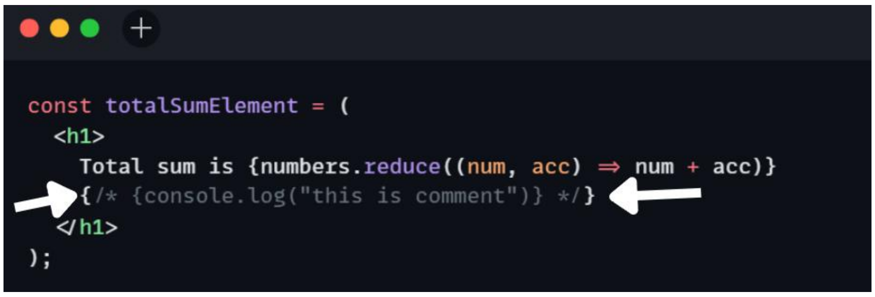
```
const numbers = [1, 2, 3, 4, 5, 6, 7];

const totalSumElement = (
  <h1>Total sum is {numbers.reduce((num, acc) => num + acc)}</h1>
);

root.render(totalSumElement);
```



Total sum is 28

Comments in JSX: -

React comments in JSX differ slightly from comments in standard **JavaScript**. Comments in JSX must be enclosed in **curly brackets {}.**



```
const totalSumElement = (
  <h1>
    Total sum is {numbers.reduce((num, acc) => num + acc)}
    {/* {console.log("this is comment")} */}
  </h1>
);
```

Here are some guidelines for using comments in JSX:

- **Curly braces** must be used to enclose comments.
- There is no way to nest comments.
- Within **attributes**, **properties**, and **children**, **comments** can be used anywhere in JSX.

Introducing Babel: -

Babel is a popular JavaScript **compiler** that allows developers to use the latest **ECMAScript** features (**ES6+**) and **JSX syntax** in their code, even if these features are not yet supported by all **browsers**. In the context of React, **Babel** plays a crucial role:

- **JSX Transformation**: Babel transforms JSX (the **XML-like syntax** used in **React**) into regular JavaScript that browsers can understand.
- **ECMAScript Compatibility**: It allows developers to write modern **JavaScript** (**ES6+**) code, which is then compiled to **ES5** for wider browser compatibility.
- **Polyfills**: Babel can add **polyfills** for newer JavaScript features, ensuring they work in **older browsers**.

Is JSX a valid JavaScript?

The answer is **yes** and **no**.

- **JSX** is not a valid Javascript **syntax** as it's not pure **HTML** or pure **JavaScript** for a browser to understand. Javascript does not have **built-in JSX**. The JS engine does not understand JSX because the **JS engine** understands **ECMAScript** or **ES6**+ code.

If the browser can’t understand JSX, how is it still working?

This is because of **Parcel**.

- Before the code gets to **JS Engine** it is sent to **Parcel** and **Transpiled** there. Then after **transpilation**, the browser gets the code that it can understand.
- **Transpilation** ⇒ **Converting** the code in such a format that the **browsers** can understand.
- **Parcel** is like a **manager** who gives the responsibility of **transpilation** to a package called **Babel**.
- **Babel** is a package that is a **compiler**/**transpiler** of JavaScript that is already present inside **‘node-modules’**. It takes **JSX** and converts it into the code that **browsers** understand, as soon as we write it and save the file. It is not created by Facebook. Learn more about **Babel** on **babeljs.io**.
- **JSX** (**transpiled** by **Babel**) ⇒ **React**.**createElement** ⇒ **ReactElement ⇒ JS Object** ⇒ **HTML Element**(**render**).

What is the difference between HTML and JSX?

- **JSX** is not HTML. It’s **HTML**-**like syntax**.
- **HTML** uses ‘**class’** property whereas **JSX** uses ‘**className’** property.
- **HTML** can use **hypens** in property names whereas **JSX** uses **camelCase** syntax.

React component

What is a React component?

A **React component** is a **reusable**, **self**-**contained piece** of code that defines a part of a **user interface**. Components can be simple (like a **button**) or complex (like an **entire page**). Components can be created as either **functions** or **classes**, and they return **JSX** (a **syntax extension** for **JavaScript** that looks similar to **HTML**) to describe what should be rendered.

Why do we need React components?

- **Reusability**: Components allow us to create **reusable UI elements**. Once we create a component, we can use it multiple times throughout our application.
- **Modularity**: They help break down complex **UIs** into smaller, manageable pieces. This makes our code more organized and easier to understand.
- **Separation of concerns**: Each component can handle its own logic and rendering, promoting a cleaner code structure.
- **Maintainability**: When we need to update a part of our **UI**, we only need to modify the relevant component, and the changes will be reflected everywhere that component is used.
- **State management**: Components can manage their own **state**, making it easier to handle **dynamic data** and **user interactions**.
- **Props system**: Components can receive data and callbacks via **props**, enabling effective communication between different parts of our application.

- **Composition**: Complex **UIs** can be built by composing simpler **components** together, following a clear hierarchy.
- **Performance optimization**: React's **virtual DOM** and **reconciliation** process work efficiently with a component-based structure to optimize rendering.
- **Code splitting**: Components make it easier to implement **code splitting** and **lazy loading**, improving application performance.
- **Testing**: Individual components can be easily unit tested in isolation, improving the overall reliability of our application.

In React, there are primarily two types of components:

- **Functional Components**
- **Class Components**

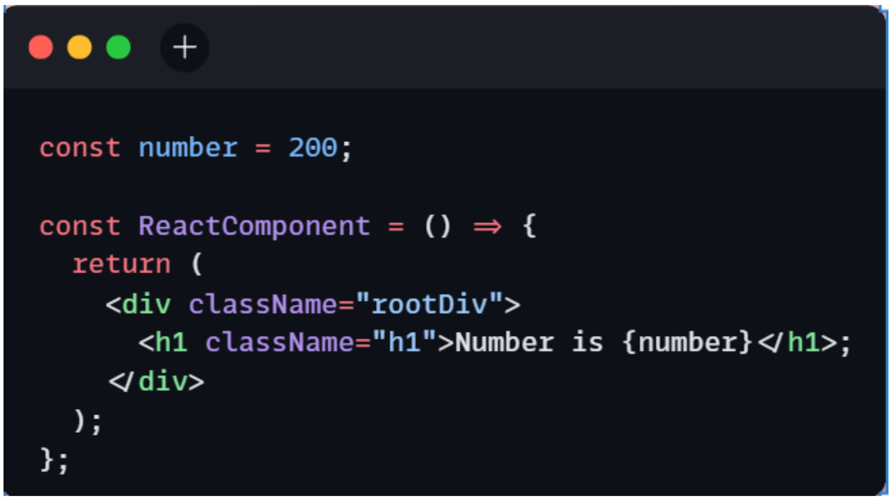
Functional Components:

A Functional component is simply a **plain JavaScript function** which accepts **props** as an **argument** and returns a **React element** or **JSX**.

Key features:

- Simpler syntax
- Can use **Hooks** for **state** and lifecycle features
- Generally preferred in modern React development

Example: -

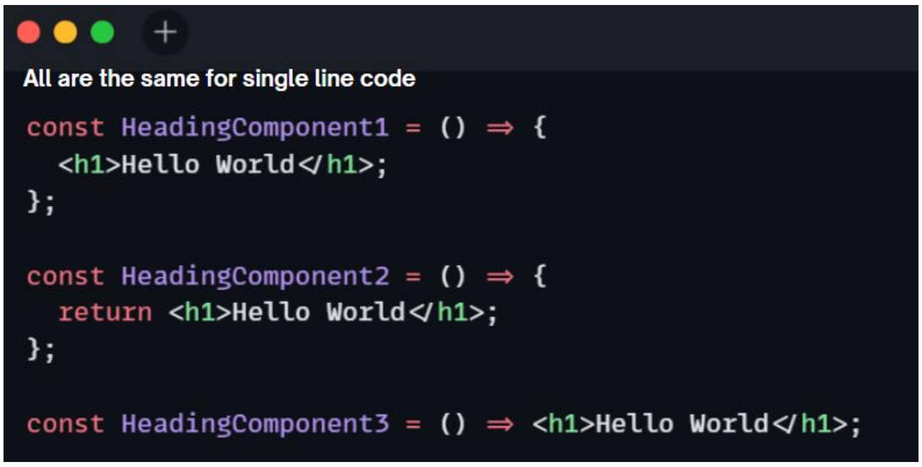


```
const number = 200;

const ReactComponent = () => {
  return (
    <div className="rootDiv">
      <h1 className="h1">Number is {number}</h1>;
    </div>
  );
};
```

Functional component names in React should start with a **capital letter** for these key reasons:

- **Distinction from HTML tags:** React uses capitalization to differentiate custom components from native HTML elements. Lowercase names are interpreted as **HTML tags**, while capitalized names are recognized as **custom components**.
- **JSX compilation:** The React compiler uses this **capitalization** to determine how to process the element. **Capitalized names** are compiled as **custom components**, ensuring proper functionality.

- **Consistency and readability:** This convention aligns with JavaScript's practice of capitalizing **constructor functions** and **classes**, making the code more **consistent** and easier to **read** for developers familiar with React.
- **Error prevention:** Using **lowercase** for custom components can lead to **unexpected behavior** or **errors**, as React would treat them as unknown **HTML** tags instead of **components**.



All are the same for single line code

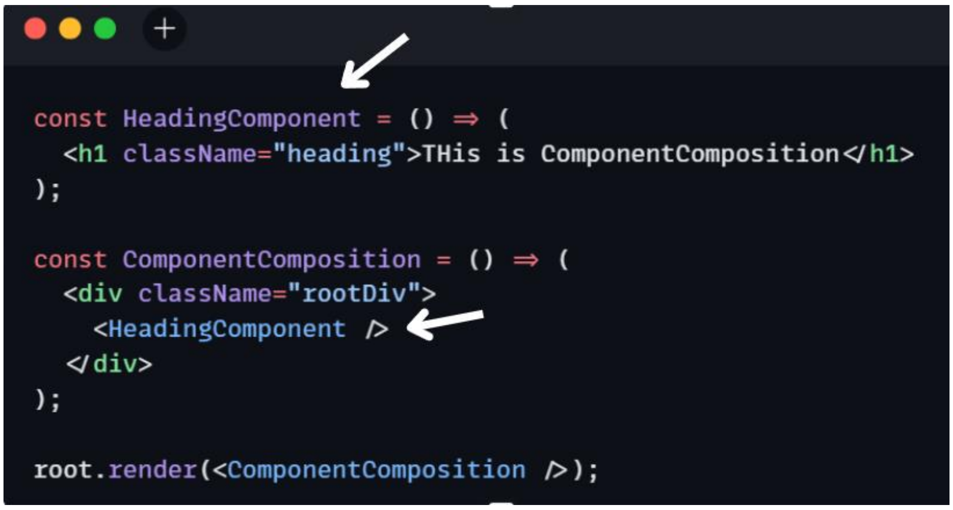
```
const HeadingComponent1 = () ⇒ {
  <h1>Hello World</h1>;
};

const HeadingComponent2 = () ⇒ {
  return <h1>Hello World</h1>;
};

const HeadingComponent3 = () ⇒ <h1>Hello World</h1>;
```

To **render** a **functional component** we call them **'<HeadingComponent1 />'**.
This is the syntax that **Babel understands**. we can also call them using these ways,

- **'<Title></Title>'**
- **'{Title()}'**
- **</Title>**

What is Components Composition?

A component inside a component. Calling a **component** inside another **component** is **Component Composition**.



```
const HeadingComponent = () ⇒ (
  <h1 className="heading">THis is ComponentComposition</h1>
);

const ComponentComposition = () ⇒ (
  <div className="rootDiv">
    <HeadingComponent />
  </div>
);

root.render(<ComponentComposition />);
```

Code inside the **'HeadingComponent'** will be utilized within the **'ComponentComposition' component**; this is known as **component composition**.

How to use JavaScript code inside JSX/Component?

Inside a React **component**, we can write any **JavaScript expression** within ` **{}** ` braces.

```
const number = 200;

const ShowNumComponent = () => {
  return (
    <div className="rootDiv">
      <h1 className="h1">{number}</h1>
    </div>
  );
};

root.render(<ShowNumComponent />);
```

How do we call ReactElement/Plain JSX inside the component?

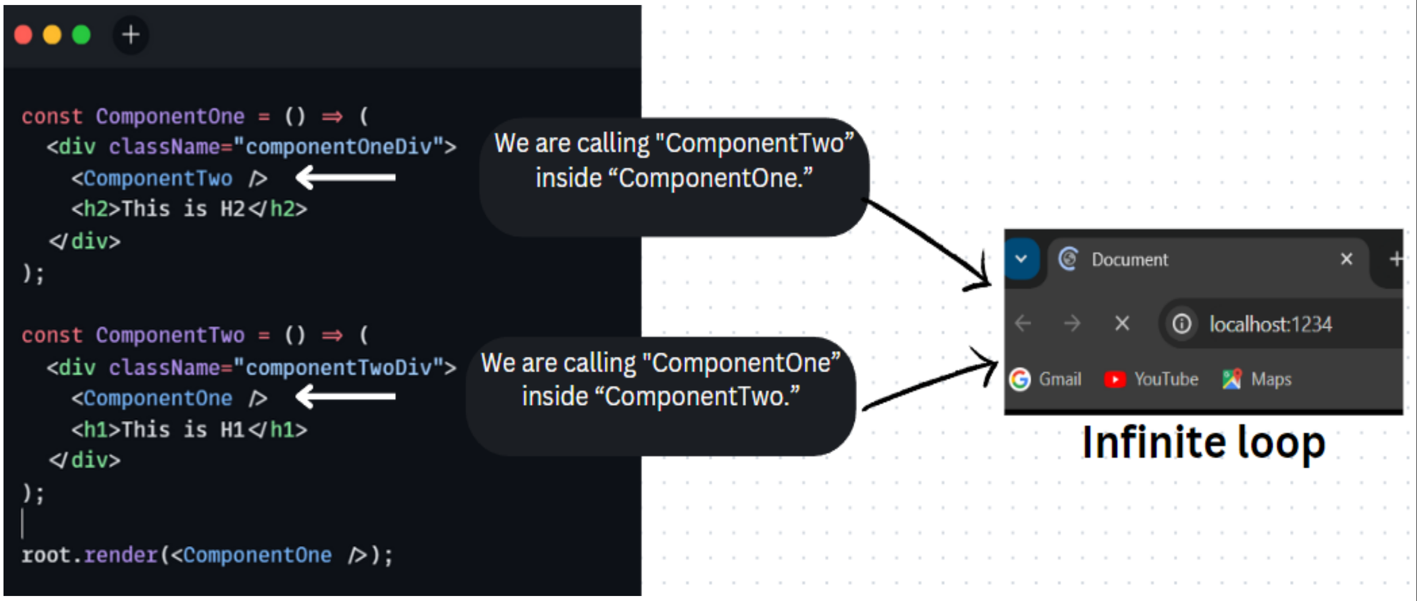
We can use **'{}'** parenthesis.

```
const element1 = <h1 className="h1">This is H1</h1>;

const HeaderComponent = () => {
  return <div className="rootDiv">{element1}</div>;
};

root.render(<HeaderComponent />);
```

What happens if two components call each other?

If we place two **components inside each other**, it will create an **infinite loop** and cause a **stack overflow**, **freezing** our **browser**. Therefore, it is not recommended to do so.

Example: -

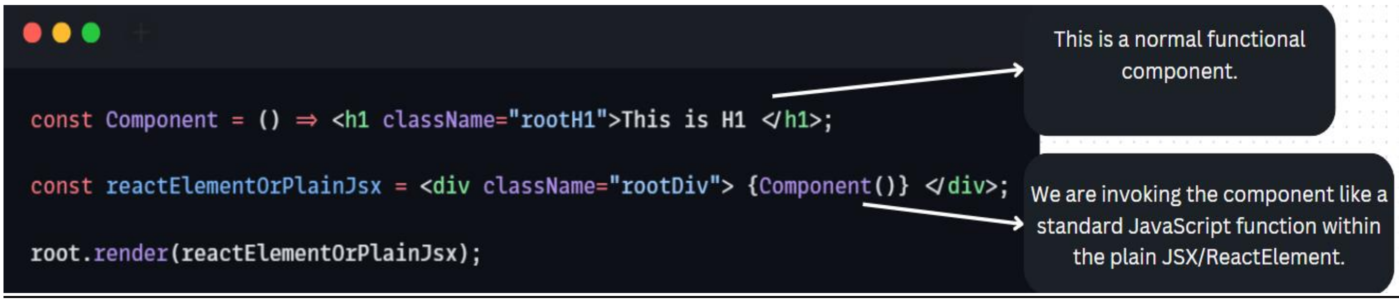


```
const ComponentOne = () => (
  <div className="componentOneDiv">
    <ComponentTwo />
    <h2>This is H2</h2>
  </div>
);

const ComponentTwo = () => (
  <div className="componentTwoDiv">
    <ComponentOne />
    <h1>This is H1</h1>
  </div>
);

root.render(<ComponentOne />);
```

We are calling "ComponentTwo" inside "ComponentOne."
We are calling "ComponentOne" inside "ComponentTwo."
Document
localhost:1234
Gmail
YouTube
Maps
Infinite loop

We can call the component like a regular JavaScript function: -

Example: -



```
const Component = () => <h1 className="rootH1">This is H1 </h1>;

const reactElementOrPlainJsx = <div className="rootDiv"> {Component()} </div>;

root.render(reactElementOrPlainJsx);
```

This is a normal functional component.
We are invoking the component like a standard JavaScript function within the plain JSX/ReactElement.