Introduction to React JS:

React.js, commonly referred to as React, is a JavaScript library used for building user interfaces (UIs) for web applications. It was developed by Facebook and released as an open-source project in 2013. React allows developers to create interactive, dynamic, and efficient UIs by breaking them down into reusable components.

Here are some key concepts and features of React:

Component-Based Architecture: React follows a component-based architecture, where UIs are divided into small, reusable pieces called components. These components encapsulate the structure, behaviour, and styles of a specific part of the UI.

Virtual DOM (Document Object Model): React uses a virtual DOM to improve performance. Instead of directly manipulating the browser's DOM, React creates a lightweight virtual representation of the DOM in memory. When changes occur, React compares the virtual DOM with the real DOM and only updates the parts that have changed, resulting in faster UI rendering.

JSX (JavaScript XML): JSX is a syntax extension for JavaScript that allows developers to write HTML-like code within JavaScript. It makes React component code more readable and intuitive.

Rich Ecosystem and Community: React has a vast ecosystem of libraries, tools, and resources that make development easier and more efficient. Additionally, React has a large and active community of developers who contribute to its growth and provide support through forums, tutorials, and documentation.

Versatility: React can be used to build various types of applications, including single-page applications (SPAs), progressive web apps (PWAs), mobile apps (using React Native), and even desktop apps (using tools like Electron). Learning React gives you the flexibility to work on a wide range of projects and platforms.

Basic React Component:

```
//create react element
index.html
-----------
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta   name="viewport"   content="width=device-width,   initial-
scale=1.0">
    <title>Document</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstra
p.min.css"            rel="stylesheet"            integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65VohhpuuCOmLASjC"
crossorigin="anonymous">

</head>
<body>
    <div id="root"> This should change</div>
```

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

    <script src="myscript.js"></script>
</body>
</html>
```

```
myscript.js
-----------
let elem = React.createElement("h1",null,"Hello World");
let   MyButton   =   React.createElement("button",{"className":"bg-
primary"},"KMITButton");

ReactDOM.render([elem,MyButton],document.getElementById("root"));
```

Hello World App with JSX

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta   name="viewport"   content="width=device-width,   initial-
scale=1.0">
    <title>Document</title>

</head>
<body>

    <div id="root"> This should change    </div>
    <script                                                    crossorigin
src="https://unpkg.com/react@18/umd/react.production.min.js"></scrip
t>
<script  crossorigin  src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
<script                                                    crossorigin
src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

<script type="text/babel">
        let MyDiv = <div> <p> Hello World, Welcome to KMIT.
            <input/>
            </p></div>
        let parentNode = document.getElementById("root");
        ReactDOM.render(MyDiv,parentNode);
</script>

</body>
</html>
```

JSX (JavaScript XML)

JSX is a syntax extension for JavaScript often used with React, a popular JavaScript library for building user interfaces.

JSX allows developers to write HTML-like code within JavaScript, making it easier to define the structure of UI components.

Understanding JSX:

HTML-like Syntax: JSX resembles HTML syntax, allowing developers to write code that looks similar to HTML within JavaScript files.

Example:

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

Embedding JavaScript Expressions:

JSX allows embedding JavaScript expressions within curly braces {}. This allows dynamic content generation and expression evaluation.

Example:

```
const name = "John";
const element = <h1>Hello, {name}!</h1>;
```

Component Rendering:

JSX makes it easy to define and render React components. Components can be defined as functions or classes and used within JSX syntax.

Example:

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

const element = <Greeting name="John" />;
```

Event Handling:

JSX facilitates event handling by allowing developers to attach event listeners directly within the JSX code.

Example:

```
function handleClick() {
    alert('Button clicked!');
}

const element = <button onClick={handleClick}>Click me</button>;
```

Limitations of JSX:

No Direct Support for HTML Attributes:

JSX may not support some HTML attributes directly. For example, the class attribute in HTML should be written as className in JSX to avoid conflicts with the JavaScript class keyword.

Example:

```
const element = <div className="container">Content</div>;
```

Inline Styling Limitations:

While inline styling is possible with JSX, it can become cumbersome when dealing with complex styles or dynamic styles. Additionally, CSS features like pseudo-selectors (:hover, :active, etc.) are not directly supported.

Example:

```
const style = {
    color: 'red',
    fontSize: '16px'
};

const element = <p style={style}>Styled Text</p>;
```

Complex Logic Inside JSX:

While JSX allows embedding JavaScript expressions, embedding complex logic directly within JSX can make the code harder to read and maintain. It's generally recommended to keep JSX code clean and move complex logic outside JSX whenever possible.

Example:

```
const element = (
    <div>
        {loggedIn ? (
            <button onClick={logout}>Logout</button>
        ) : (
            <button onClick={login}>Login</button>
        )}
    </div>
);
```

Tooling Dependencies:

JSX requires a build step to transform the JSX code into plain JavaScript that browsers can understand. This adds a dependency on build tools like Babel or TypeScript.

Example:

```
// JSX code
const element = <h1>Hello, world!</h1>;

// After transformation
```

```
const element = React.createElement('h1', null, 'Hello, world!');
```

React App Project Directory

create-react-app is a command-line utility provided by the React team for quickly setting up new React projects with a pre-configured build setup.

When you run **npx create-react-app**, it initializes a new React project with all the necessary dependencies and configurations to get started with React development.

Here's how npx create-react-app works and the folder structure it generates:

1. Initializing a New React Project:

You can initialize a new React project in your computer, using npx create-react-app command followed by the name of your project. For example:

```
npx create-react-app my-react-app
```

This command creates a new directory called my-react-app and sets up a basic React project structure inside it.

2. Folder Structure Generated:

The folder structure generated by **create-react-app** typically looks like this:

```
my-react-app/
├── node_modules/        # Dependencies installed by npm/yarn
├── public/              # Public assets (HTML, images, etc.)
│   ├── favicon.ico      # Favicon
│   ├── index.html       # Main HTML file
│   └── ...
├── src/                 # Source code
│   ├── components/      # React components
│   │   ├── App.js       # Main component
│   │   ├── ...
│   ├── App.css          # Component-specific styles
│   ├── index.js         # Entry point for React application
│   ├── index.css        # Global styles
│   └── ...
├── .gitignore           # Git ignore file
├── package.json         # Project metadata and dependencies
├── README.md            # Project documentation
├── yarn.lock            # Yarn lock file (if using Yarn)
└── ...
```

Explanation of Folder Structure:

node_modules/: This directory contains all the dependencies installed by npm or Yarn.

public/: This directory contains static assets like HTML files, images, and the favicon.ico. The index.html file here is the entry point of your application and is where your React app is mounted.

src/: This directory contains the source code of your React application.

> **components/:** This directory typically contains your React components, including the main App.js component.
>
> **App.js:** This is the main component of your React application.
>
> **index.js:** This file is the entry point for your React application. It imports App.js and mounts it into the index.html.
>
> **index.css and App.css:** These files contain global and component-specific styles, respectively.

.gitignore: This file specifies intentionally untracked files to ignore if you're using Git for version control.

package.json: This file contains metadata about the project as well as the list of dependencies required by the project.

README.md: This file typically contains information about your project, including how to set it up and use it.

yarn.lock: This file is generated by Yarn to lock down the versions of dependencies installed.

The folder structure generated by create-react-app provides a solid foundation for building React applications. It follows best practices and conventions, making it easy for developers to get started with React development while also allowing for scalability and maintainability as the project grows.

Types of Components

In React, components are the building blocks used to create user interfaces. There are two main types of components: functional components and class components.

1. Functional Components:

Functional components are JavaScript functions that return JSX. They are also known as stateless components because they don't manage state internally. They are simpler and easier to read compared to class components, making them a preferred choice for many developers, especially for presentational components.

Example:

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

// Usage
const element = <Welcome name="John" />;
```

2. Class Components:

Class components are ES6 classes that extend from React.Component. They have more features and capabilities compared to functional components. Class components have a built-in state and lifecycle methods, making them suitable for managing complex state and behavior.

Example:

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

// Usage
const element = <Welcome name="John" />;
```

Choosing Between Functional and Class Components:

Functional Components:

- ✓ Preferred for simple components that don't need state or lifecycle methods.
- ✓ Easier to read and test.
- ✓ With the introduction of React hooks, functional components can now also manage state and use lifecycle methods, making them even more powerful.

Class Components:

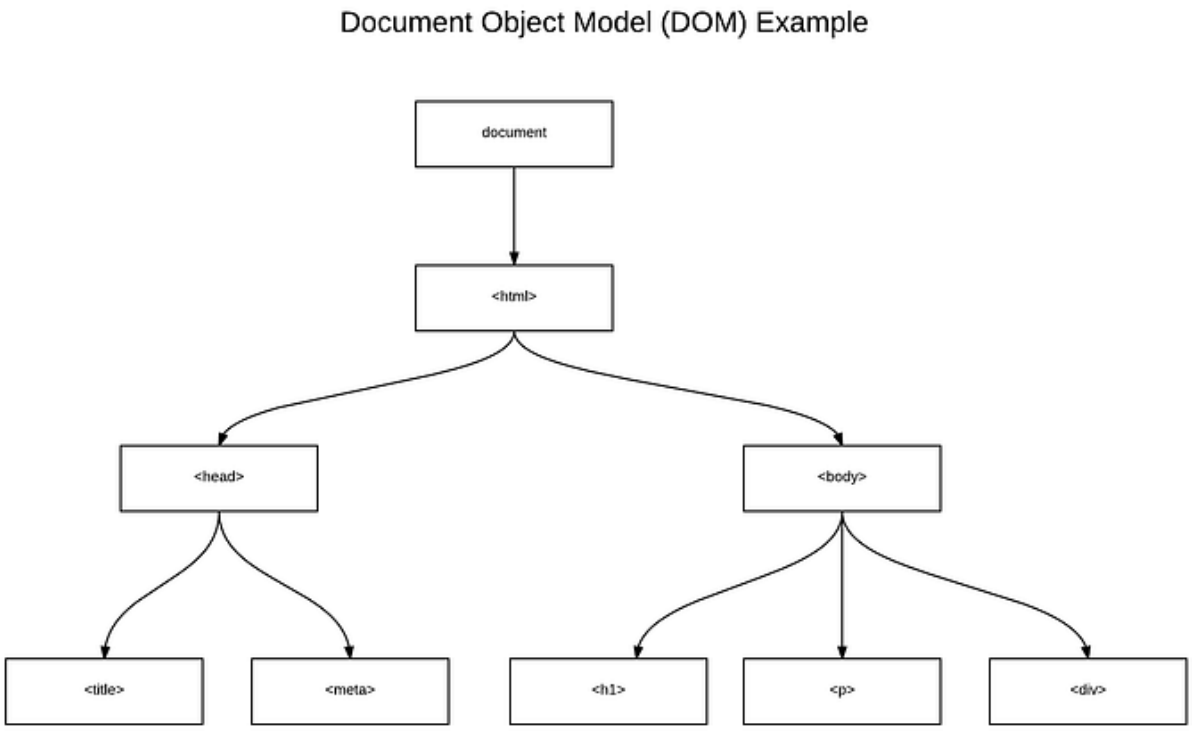
- ✓ Used for components that require state management and lifecycle methods.
- ✓ Suitable for complex components with more logic.
- ✓ Legacy codebases might still use class components, although functional components with hooks are becoming more common.

With the introduction of React hooks, functional components have become more powerful and are now the preferred choice in many cases. However, class components still have their place, especially in legacy codebases or in situations where complex state management and lifecycle methods are required.

What is DOM?

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The DOM is an abstraction of a page's HTML structure. It takes HTML elements and wraps them in an object with a tree-structure — maintaining the parent/child relationships of those nested HTML elements. This provides an API that allows us to traverse nodes (HTML elements) and manipulate them in a number of ways — such as adding nodes, removing nodes, editing a node's content, etc.



DOM inefficiency

DOM was originally intended for static UIs — pages rendered by the server that don't require dynamic updates. When the DOM updates, it has to update the node as well as re-paint the page with its corresponding CSS and layout. Say we have an array.

let fruits = ['Apple', 'Orange', 'Banana']

We want to update here from Orange to lemon. Then we need to create a new array.

let fruits = ['Apple', 'Lemon', 'Banana']

In an efficient way we can just traverse to the fruits[2] and update only this element.

Now it's common to have a thousands node in a single SPA. So repaint the whole page for each change is very-very expensive.

Ideally, we'd like to only re-render items that receive updates, leaving the rest of the items as-is.

Knowing when to update

There are a couple of ways in which components can tell when a data update occurs and whether or not it needs to re-render to the UI:

Dirty Checking (slow) — Checks through all node's data at a regular interval to see if there have been any changes. This is inefficient because it requires traversing every single node recursively to make sure it's data isn't "dirty" (out of date).

Observable (fast) — Components are responsible for listening to when an update takes place. Since the data is saved on the state, components can simply listen to events on the state and if there is an update, it can re-render to the UI. React uses it.

Virtual DOM

The Virtual DOM is a light-weight abstraction of the DOM. You can think of it as a copy of the DOM, that can be updated without affecting the actual DOM. It has all the same properties as the real DOM object, but doesn't have the ability to write to the screen like the real DOM. The virtual DOM gains it's speed and efficiency from the fact that it's lightweight. In fact, a new virtual DOM is created after every re-render.

Reconciliation is a process to compare and keep in sync the two files (Real and Virtual DOM). Diffing algorithm is a technique of reconciliation which is used by React.

How does updates work in React?

On the first load, ReactDOM.render() will create the Virtual DOM tree and real DOM tree.

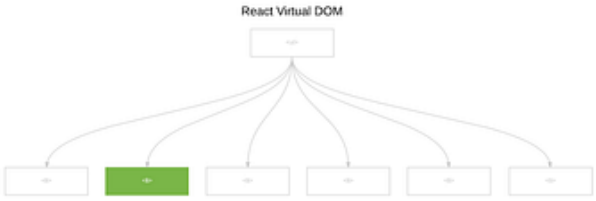
As React works on Observable patterns, when any event(like key press, left click, api response, etc.) occurred, Virtual DOM tree nodes are notified for props change, If the properties used in that node are updated, the node is updated else left as it is.

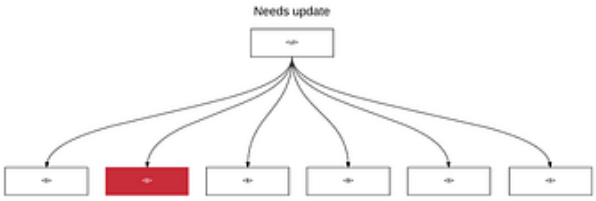
React compares Virtual DOM with real DOM and updates real DOM. This process is called Reconciliation. React uses Diffing algorithm techniques of Reconciliation.

Updated real DOM is repainted on browser.

Reconciliation

React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called Reconciliation.





Difference between virtual and real DOM

The differences between virtual and real DOM are summarized in the table below:

Real DOM	**Virtual DOM**
It is an abstraction of a page's HTML.	It is an abstraction of an HTML DOM.
It can manipulate on-screen elements.	It cannot manipulate on-screen elements.
Any change updates the entire DOM tree.	Any change only updates the relevant node in the tree.
Updating is slow and inefficient.	Updating is fast and efficient.

Additional Notes:

Diffing Algorithm

React first compares the two root elements. The behavior is different depending on the types of the root elements.

React compared the root DOM Elements Types.

Elements of different types: Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from <a> to <img>, or from <Article> to <Comment>, or from <Button> to — any of those will lead to a full rebuild. This will lead to component unmount and mount lifecycle calls too.

DOM Elements Of The Same Type: When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes.After handling the DOM node, React then recurses on the children. This will lead to component update lifecycle calls.

Is the Shadow DOM the same as the Virtual DOM?

No, they are different. The Shadow DOM is a browser technology designed primarily for scoping variables and CSS in web components. The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

Understanding Props and State in React

React is a JavaScript library for building user interfaces, and understanding how props and state work is fundamental to developing React applications effectively. Props and state are essential concepts in React that enable components to manage and pass data. Let's delve into each of them:

Props (Properties):

Props are read-only data that are passed from a parent component to a child component.

They allow you to pass data from one component to another and are immutable (cannot be modified within the component).

Props are passed down the component tree, and each component in the tree can access and use them.

They are defined as attributes in JSX and accessed via this.props in class components or directly as arguments in functional components.

Example:

// ParentComponent.js

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

function ParentComponent() {
    return <ChildComponent name="John" age={30} />;
}

export default ParentComponent;
```

// ChildComponent.js

```
import React from 'react';

function ChildComponent(props) {
    return (
        <div>
            <p>Name: {props.name}</p>
            <p>Age: {props.age}</p>
        </div>
    );
}

export default ChildComponent;
```

State:

State is mutable and represents the internal state of a component. It is managed and controlled by the component itself.

Unlike props, state is local to the component and can be changed using setState() method.

When state changes, React re-renders the component and its children to reflect the updated state.

State should be initialized in the constructor for class components or using useState() hook for functional components.

Example:

// ClassComponent.js

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

class ClassComponent extends Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0
        };
    }

    incrementCount = () => {
        this.setState({ count: this.state.count + 1 });
    }

    render() {
        return (
            <div>
                <p>Count: {this.state.count}</p>
                <button
onClick={this.incrementCount}>Increment</button>
            </div>
        );
    }
}

export default ClassComponent;
```

// FunctionalComponent.js

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

function FunctionalComponent() {
    const [count, setCount] = useState(0);
```

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

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

export default FunctionalComponent;
```

Set State in Depth:

When updating state, it's crucial to understand that setState() may be asynchronous. React may batch multiple setState() calls into a single update for performance reasons.

To perform state updates based on the previous state, you should use the function form of setState() to ensure the correct state is used.

It's important to note that setState() does not immediately mutate this.state. Instead, it creates a pending state transition.

React may merge multiple setState() calls into a single update for better performance, so it's advised to pass an updater function instead of an object when depending on the current state.

Example:

// Using function form of setState

```
this.setState((prevState) => {
    return { count: prevState.count + 1 };
});
```

// Incorrect way of updating state

```
this.setState({ count: this.state.count + 1 }); // May lead to unexpected
behavior
```

// Correct way of updating state when depending on previous state

```
this.setState((prevState) => {
    return { count: prevState.count + 1 };
});
```

Destructuring

Destructuring in JavaScript is a convenient way to extract multiple properties from objects or arrays and assign them to variables.

In React, destructuring is commonly used with props and state to make accessing values more straightforward and to write cleaner and more readable code.

Destructuring Props

Props (short for properties) are a way of passing data from parent components to child components in React.

Destructuring props can simplify the way you access these values within your component.

Example without destructuring:

```
function UserProfile(props) {
  return (
    <div>
      <h1>{props.name}</h1>
      <p>Email: {props.email}</p>
    </div>
  );
}
```

Example with destructuring:

```
function UserProfile({ name, email }) {
  return (
    <div>
      <h1>{name}</h1>
      <p>Email: {email}</p>
    </div>
  );
}


// Usage
<UserProfile name="John Doe" email="john@example.com" />
```

In the above example, the name and email properties are extracted directly in the function parameters. This makes the function body cleaner and easier to read.

Destructuring State

In React, state is used for managing data that changes over time in a component.

When using class components, state is usually accessed via **this.state**.

With the introduction of hooks in React 16.8, functional components can also manage state using the useState hook, and destructuring is often used to access the state variables.

Example in Class Component:

```
import React, { Component } from 'react';
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    const { count } = this.state; // Destructuring state
    return (
      <div>
        <h1>{count}</h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

In the class component example, count is destructured from this.state inside the render method.

Example in functional Component with useState:

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

function Counter() {
  const [count, setCount] = useState(0); // Destructuring with
useState

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

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

In the functional component example, useState returns an array containing the current state and a function to update it, which are destructured right in the variable declaration.

Functional(Stateless) Components Vs Class(Stateful) Components

Functional (Stateless) Components

Functional components are JavaScript functions that return JSX(React elements) .Functional components, also known as stateless components (prior to Hooks).

The primary role of these components was to present UI without involving any lifecycle methods or state management.

However, the introduction of Hooks changed the landscape, allowing functional components to use state, lifecycle events, and more without being classes.

Advantages of Functional Components:

Simplicity and Conciseness: They're straightforward to write and understand, making them more maintainable.

Hooks: The introduction of Hooks like useState, useEffect, useContext, and custom Hooks, adds the ability to manage state, perform side effects, use context, and more, which previously required class components.

Less Boilerplate: Avoids the use of this keyword, which can be confusing for some developers, especially in handling events.

For example:

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

function Welcome(props) {

      const [name, setName] = useState(props.name);

  return <h1>Hello, {name}!</h1>;

}
```

Class(Stateful) Components

Class components are ES6 classes that extend from React.Component or React.PureComponent and provide more features than functional components.

Class Components are more powerfull in terms of lifecycle methods and state management.

They were the main method for managing state and side effects in React applications before Hooks were introduced.

Advantages of Class Components:

Lifecycle Methods: They offer several lifecycle methods like componentDidMount, shouldComponentUpdate, componentDidUpdate, and componentWillUnmount, allowing you to run code at specific points in the component's lifecycle.

State Management: Easily manage local state within a component.

Lifecycle Methods:

These methods provide hooks into specific points in a component's lifetime, such as initialization, updating, and teardown.

They are crucial for managing side effects, subscriptions, and manually controlling component updates.

For example:

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

class Welcome extends Component {

  constructor(props) {

    super(props);

    this.state = { name: props.name };

  }

  render() {

    return <h1>Hello, {this.state.name}!</h1>;

  }

}
```

The Shift Towards Functional Components with Hooks

The introduction of Hooks in React 16.8 was a pivotal moment that blurred the traditional roles of functional and class components. Hooks provided a more powerful and flexible way to use React's features in functional components, aligning with the modern trends towards functional programming in JavaScript.

Why the Shift?

Code Reusability and Composition: Hooks make it easier to share stateful logic between components, enhancing code reusability and composition.

Reduced Complexity: Functional components with Hooks can achieve the same functionality as class components with less code and complexity, leading to easier maintenance and debugging.

Community and Future Direction: The React team and community strongly advocate for functional components with Hooks, indicating a direction towards functional components for future development.