Inception

`React.createElement()` – The Ultimate Guide

What is it?

```
React.createElement(type, props, children)
```

Breakdown:

Part	Description	Example
`type`	HTML tag or custom component	`"div"` or `MyComponent`
`props`	Attributes for that element	`{ id: "box", className: "a" }`
`children`	What goes *inside* the element	String, another element, or array

Execution Flow – Step-by-Step

```
const el = React.createElement("h1", {}, "Hello!");
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(el);
```

Behind-the-Scenes Flow:

1. `React.createElement` returns a **plain JS object** (virtual DOM).
2. `ReactDOM.createRoot(...).render(...)` **mounts** it to real DOM.
3. Existing content inside `#root` is **cleared** and replaced.

Output from `console.log(el)`:

```
{
  type: "h1",
  props: {
    children: "Hello!"
  },
  ...
}
```

It's NOT an HTML element — it's an object representing what HTML **should** look like.

☑️ Basic Example – Single Element

```
const el = React.createElement("h1", {}, "💡 Hello from React!");
ReactDOM.createRoot(document.getElementById("root")).render(el);
```

👀 Output:

```
<h1>💡 Hello from React!</h1>
```

🪆 Nested Elements – Manual Method

```
const heading = React.createElement("h1", {}, "📌 Nested Element");
const child = React.createElement("div", { id: "child" }, heading);
const parent = React.createElement("div", { id: "parent" }, child);

ReactDOM.createRoot(document.getElementById("root")).render(parent);
```

🧠 Visualization:

```
<div id="parent">
  <div id="child">
    <h1>📌 Nested Element</h1>
  </div>
</div>
```

🎯 Nested Elements – Inline Method

```
const tree = React.createElement(
  "div",
  { id: "parent" },
  React.createElement(
    "div",
    { id: "child" },
    React.createElement("h1", {}, "🌳 Deep Nest")
  )
);
ReactDOM.createRoot(document.getElementById("root")).render(tree);
```

🧮 Same output, but defined inline for quick trees.

👯 Sibling Elements

```
const h1 = React.createElement("h1", {}, "Sibling 1 🧒");
const h2 = React.createElement("h2", {}, "Sibling 2 👧");

const parent = React.createElement("div", {}, h1, h2);
ReactDOM.createRoot(document.getElementById("root")).render(parent);
```

Output:

```
<div>
  <h1>Sibling 1 🧒</h1>
  <h2>Sibling 2 👧</h2>
</div>
```

🔢 Array of Children (Loop Scenario)

```
const kids = [
  React.createElement("li", {}, "👶 Kid A"),
  React.createElement("li", {}, "👶 Kid B"),
];

const ul = React.createElement("ul", {}, kids);
ReactDOM.createRoot(document.getElementById("root")).render(ul);
```

🔁 Replace Existing DOM

📄 Initial HTML:

```
<div id="root">
  <p>This gets overwritten!</p>
</div>
```

🧠 JS:

```
const element = React.createElement(
  "div",
  { id: "new-container" },
```

```
  React.createElement("h1", {}, "🔥 React Overwrites This!")
);
ReactDOM.createRoot(document.getElementById("root")).render(element);
```

> 🧪 React **removes all children** of `#root` and inserts its own.

🖱 On Button Click

```
<button onclick="replaceUI()">Click Me</button>
<div id="root"><p>Old content</p></div>

<script>
  function replaceUI() {
    const newUI = React.createElement("h1", {}, "🎉 Clicked and Replaced!");
    const root = ReactDOM.createRoot(document.getElementById("root"));
    root.render(newUI);
  }
</script>
```

⏱ Delayed Rendering (e.g., API result)

```
setTimeout(() => {
  const delayed = React.createElement("h1", {}, "⏰ Time’s Up!");
  const root = ReactDOM.createRoot(document.getElementById("root"));
  root.render(delayed);
}, 3000);
```

🧪 Conditional Rendering (using ternary)

```
const isLoggedIn = true;

const el = React.createElement(
  "h1",
  {},
  isLoggedIn ? "👋 Welcome Back!" : "🔒 Please Login"
);

ReactDOM.createRoot(document.getElementById("root")).render(el);
```

🎨 Adding `props`, `style`, `className`

```
const el = React.createElement("h1", {
  id: "title",
  className: "main-heading",
  style: { color: "blue", fontSize: "24px" }
}, "✨ Styled Element");

ReactDOM.createRoot(document.getElementById("root")).render(el);
```

⚠️ Common Pitfall: Multiple Roots

🚫 Incorrect:

```
ReactDOM.createRoot(document.getElementById("root")).render(...);
ReactDOM.createRoot(document.getElementById("root")).render(...);
```

☑️ Correct:

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(...);
root.render(...); // safe reuse
```

🧠 JSX vs `createElement`

Feature	`React.createElement()`	JSX
Syntax	Verbose, manual	Clean, declarative
Ideal For	Internals, advanced use cases	Everyday development
Compiles To	JS object	`React.createElement()`
Sample	`React.createElement("div")`	`<div></div>`

📝 JSX is just **syntactic sugar** 🍬 over `createElement`.

📋 Summary Table

Case	Example
Single Element	`React.createElement("h1", {}, "Hi")`
Nested Elements	Parent → Child → Heading
Sibling Elements	Multiple children in `createElement("div", {}, a, b)`

Case	Example
Array of Elements	`React.createElement("ul", {}, [a, b])`
Replace Existing DOM	`render()` into `#root`
Conditional Content	Ternary or `if-else` wrapped content
Delayed Rendering	Use `setTimeout()` or API callback
Event-based Rendering	Inside a click handler or form event
Styled Components	Pass `style` prop as object
Class Name	Use `className` instead of `class`

🧪 Scenario 7: 🕴️ Only the `#root` is Replaced — Siblings Stay Safe!

☑️ HTML Structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Sibling Preservation</title>
  <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>
</head>
<body>

  <!-- 👇 This is the root React will target -->
  <div id="root">
    <p>This will be replaced ☑️</p>
  </div>

  <!-- 👇 This is a sibling and should remain untouched -->
  <div id="non-react">
    <p>I should not be touched 🚫</p>
  </div>

  <script>
    // Replace ONLY #root's content
    const element = React.createElement(
      "div",
      { id: "react-container" },
      React.createElement("h1", {}, "🔥 React has taken over root!")
    );

    const root = ReactDOM.createRoot(document.getElementById("root"));
    root.render(element);
  </script>
```

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

🔍 What Actually Happens?

🧱 The DOM becomes:

```
<body>
  <div id="root">
    <div id="react-container">
      <h1>🔥 React has taken over root!</h1>
    </div>
  </div>

  <div id="non-react">
    <p>I should not be touched 🚫</p>
  </div>
</body>
```

✅ ✅ **Only `#root` is touched!** 🌓 `#non-react` remains **safe and unchanged**.

🧠 Why?

Because:

- `ReactDOM.createRoot(document.getElementById("root"))` tells React: ➡️ "Only manage this DOM node and what's inside it."
- React **never touches outside** of the root container.
- It doesn't replace siblings or parents — just what's inside that root.

📌 Summary Table:

Case	What Happens?
✅ Replacing inner content of `#root`	Works as expected
✅ React root has sibling `<div>` or `<p>`	Siblings remain untouched
❌ Mounting React **without a root container**	❗ Won't work — needs a valid root node

---

💬 Bonus Note: Safe Usage Tip

> Always wrap your React app inside a dedicated container like:

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

This makes sure React doesn’t interfere with static layout around it (e.g., navbars, footers, ads, etc.).
