Advanced Web Programming

Ung Văn Giàu
Email: giau.ung@eiu.edu.vn

Using React Hooks

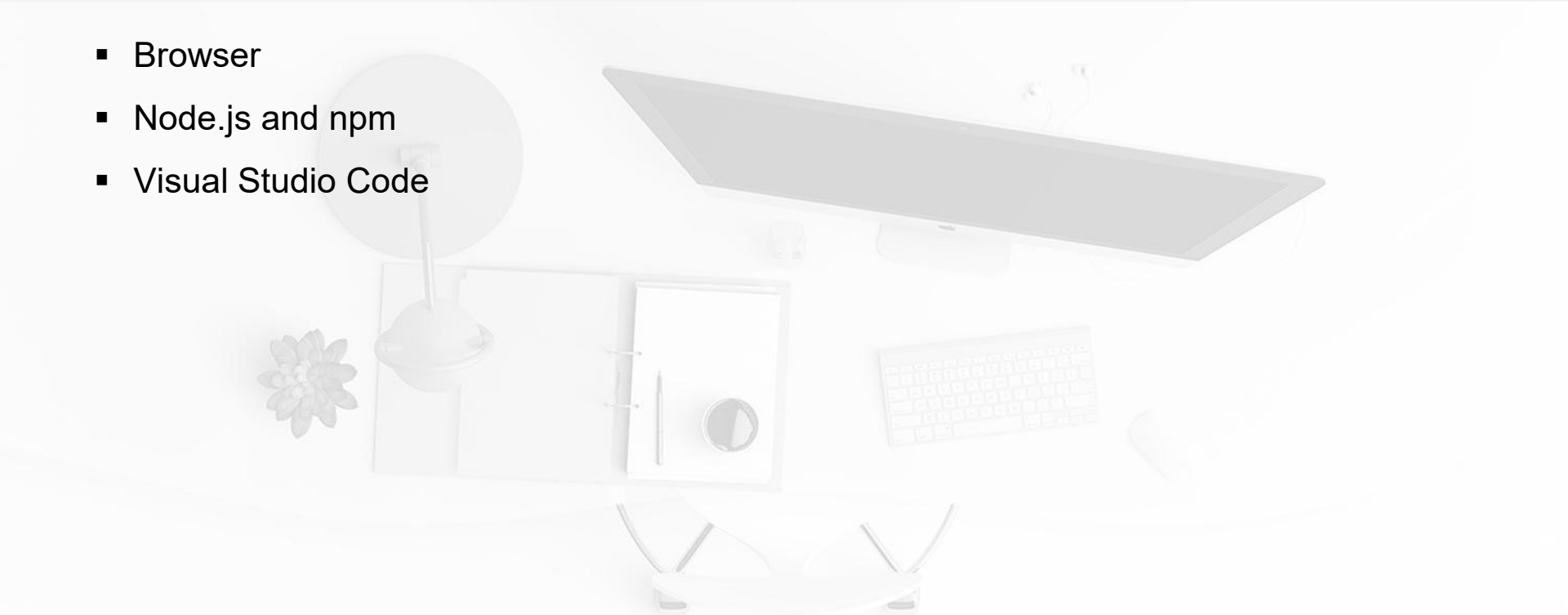
Topic

We'll cover the following topics:

- Using the effect Hook
- Using state Hooks
- Using the ref Hook
- Using the memo Hook
- Using the callback Hook

Technical requirements

- Browser
- Node.js and npm
- Visual Studio Code



1. Using the effect Hook

Understanding the effect Hook parameters

- The effect Hook is used for component side effects.
- A **component side effect** is something **executed outside the scope of the component** such as a web service request.
- The effect Hook is defined using the **useEffect** function from React.
- useEffect contains two parameters:
  - A **function** that executes the effect; at a minimum, this function runs each time the component is rendered
  - An **optional array** of dependencies that cause the effect function to rerun when changed

1. Using the effect Hook

Understanding the effect Hook parameters

- Example:

```
function SomeComponent() {
  function someEffect() {
    console.log("Some effect");
  }
  useEffect(someEffect);
  return ...
}
```

1. Using the effect Hook

Understanding the effect Hook parameters

- The same example but with an anonymous effect function:

```
function SomeComponent() {
  useEffect(() => {
    console.log("Some effect");
  });
  return ...
}
```

1. Using the effect Hook

Understanding the effect Hook parameters

- Another example of an effect:

```
function SomeOtherComponent({ search }) {
  useEffect(() => {
    console.log("An effect dependent on a search prop",
      search);
  }, [search]);
  Return ...;
}
```

1. Using the effect Hook

The rules of Hooks:

- A Hook can **only be called at the top level** of a function component. So, a Hook **can't be called in a loop or in a nested function** such as an event handler.
- A Hook **can't be called conditionally**.
- A Hook can **only be used in function components** and not class components.

1. Using the effect Hook

The rules of Hooks

- Example 1:

```
export function AnotherComponent() {
  function handleClick() {
    useEffect(() => {
      console.log("Some effect");
    });
  }
  return <button onClick={handleClick}>Cause effect</button>;
}
```

1. Using the effect Hook

The rules of Hooks

- A corrected version of Example 1:

```
export function AnotherComponent() {
    const [clicked, setClicked] = useState(false);
    useEffect(() => {
        if (clicked) {
            console.log("Some effect");
        }
    }, [clicked]);
    function handleClick() {
        setClicked(true);
    }
    return <button onClick={handleClick}>Cause effect</button>;
}
```

1. Using the effect Hook

The rules of Hooks

- Example 2:

```
function YetAnotherComponent({ someProp }) {
  if (!someProp) {
    return null;
  }
  useEffect(() => {
    console.log("Some effect");
  });
  return ...
}
```

1. Using the effect Hook

The rules of Hooks

- A corrected version of Example 2:

```
function YetAnotherComponent({someProp}) {
    useEffect(() => {
        if (someProp) {
            console.log("Some effect");
        }
    });
    if (!someProp) {
        return null;
    }
    return ...
}
```

1. Using the effect Hook

Effect cleanup

- **An effect can return a function** that **performs cleanup logic when the component is unmounted**. Cleanup logic ensures nothing is left that could cause a memory leak.
- Example:

```
function ExampleComponent({onClickAnywhere}) {
  useEffect(() => {
    function handleClick() {
      onClickAnywhere();
    }
    document.addEventListener("click", handleClick);
  });
  return ...
}
```

1. Using the effect Hook

Effect cleanup

- A cleanup function example:

```
function ExampleComponent({ onClickAnywhere }) {
  useEffect(() => {
    function handleClick() {
      onClickAnywhere();
    }
    document.addEventListener("click", listener);
    return function cleanup() {
      document.removeEventListener("click", listener);
    };
  });
  return ...;
}
```

1. Using the effect Hook

Effect cleanup

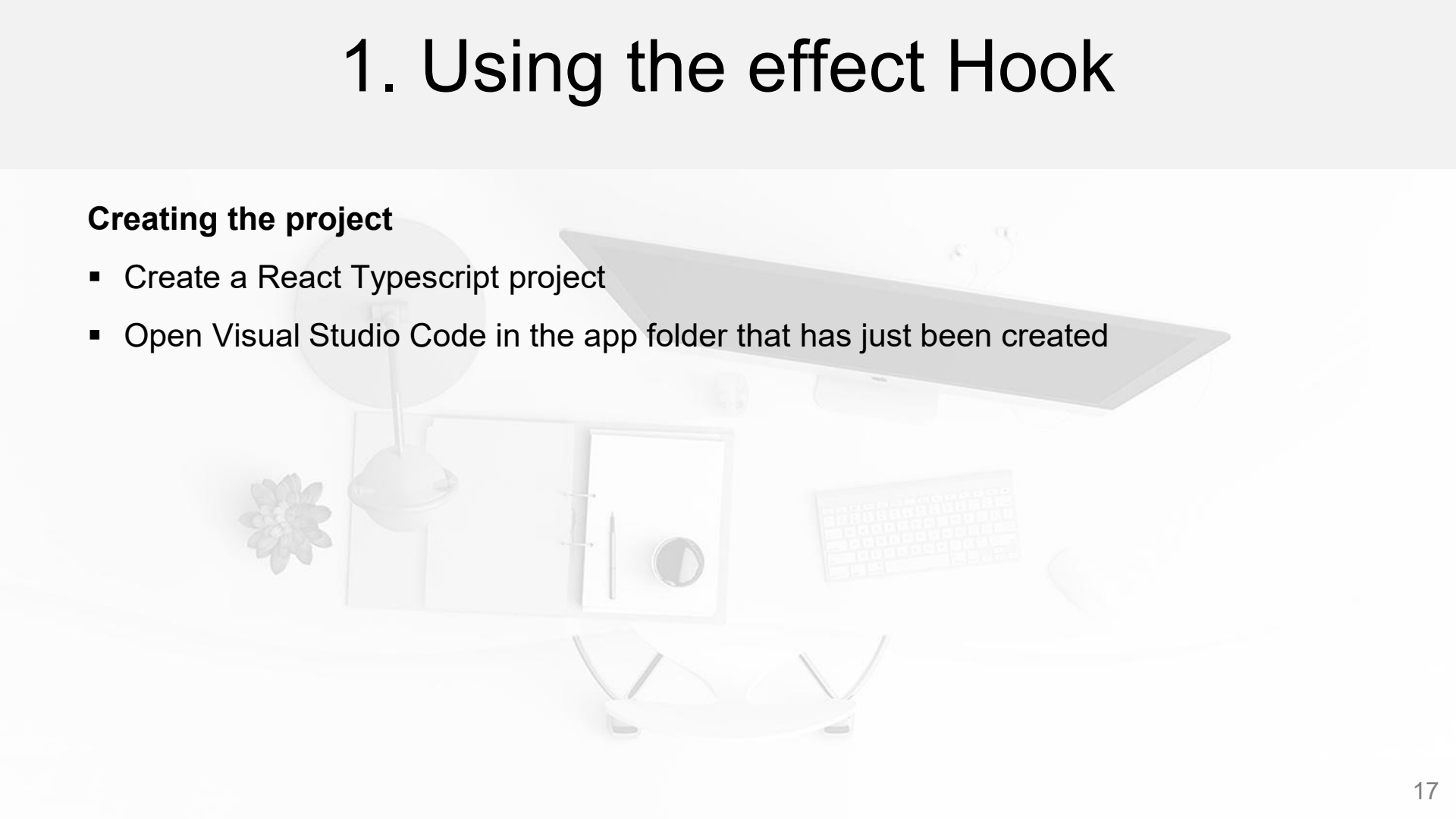
- A cleanup example using an anonymous arrow function:

```
function ExampleComponent({ onClickAnywhere }) {
  useEffect(() => {
    function handleClick() {
      onClickAnywhere();
    }
    document.addEventListener("click", listener);
    return () => {
      document.removeEventListener("click", listener);
    };
  });
  return ...;
```

1. Using the effect Hook

Creating the project

- Create a React Typescript project
- Open Visual Studio Code in the app folder that has just been created



1. Using the effect Hook

Creating the project

- Open **App.tsx** and replace the content with the following:

```
import React from 'react';
import './App.css';
function App() {
    return <div className="App"></div>;
}
export default App;
```

- Start the app running in development mode by running **npm run dev** in the terminal.

1. Using the effect Hook

Fetching data using the effect Hook

- A **common use** of the effect Hook is **fetching data**.
- Carry out the following steps
  - Create a function that will simulate a data request. Create a file called **getPerson. ts** in the **src folder**:

```
type Person = {
  name: string,
};
export function getPerson(): Promise<Person> {
  return new Promise((resolve) =>
    setTimeout(() => resolve({ name: "Bob" }), 1000)
  );
}
```

1. Using the effect Hook

Fetching data using the effect Hook

- Carry out the following steps
  - Create a React component that will eventually display a person and a score. Create a file called **PersonScore.tsx** in the **src folder**:

```
import { useEffect } from 'react';
import { getPerson } from './getPerson';
export function PersonScore() {
    return null;
}
```

1. Using the effect Hook

Fetching data using the effect Hook

- Carry out the following steps
  - Add the following effect above the return statement:

```
export function PersonScore() {
  useEffect(() => {
    getPerson().then((person) => console.log(person));
  }, []);
  return null;
}
```

1. Using the effect Hook

Fetching data using the effect Hook

- Carry out the following steps
  - Open **App.tsx** and render the **PersonScore** component inside the div element:

```
import React from 'react';
import './App.css';
import { PersonScore } from './PersonScore';

function App() {
  return (
    <div className="App">
      <PersonScore />
    </div>
  );
}
export default App;
```

1. Using the effect Hook

Fetching data using the effect Hook

- The **async/await syntax** is an alternative way **to write asynchronous code**.
- Carry out the following steps
  - Go to the running app in the browser and go to the **Console panel** in the browser's **DevTools** to check the result.
  - Refactor how the effect function is called to expose an interesting problem. Open **PersonScore.tsx** and change the **useEffect call** to use the **async/await syntax**:

```
useEffect(async () => {
  const person = await getPerson();
  console.log(person);
}, []);
```

1. Using the effect Hook

Fetching data using the effect Hook

- Carry out the following steps
  - Update the code and use the approach suggested in the error message:

```
useEffect(() => {
  async function getThePerson() {
    const person = await getPerson();
    console.log(person);
  }
  getThePerson();
}, []);
```

1. Using the effect Hook

Recap

- The **effect Hook** is used to **execute component side effects** when a component is rendered or when certain props or states change.
- A common use case for the effect Hook is **fetching data**. Another use case is where DOM events need to be manually registered.
- Any required effect cleanup can be done in a function returned by the effect function.

2. Using state Hooks

Using useState

- The useState Hook allows state to be defined in a variable.
- The syntax for useState is as follows:

const [state, setState] = useState(initialState);

- Example description:

Enhance the PersonScore component created in the last section to store the person's name in state. We will also have **state for a score** that is incremented, decremented, and reset using some buttons in the component. We will also **add the loading state** to the component, which will show a loading indicator when true.

2. Using state Hooks

Using useState

- Carry out the following steps
  - Open **PersonScore.tsx** and add useState to the React import statement:

    import { useEffect, useState } from 'react';

  - Add the following state definitions for name, score, and loading at the top of the component function, above the useEffect call:

```
export function PersonScore() {
  const [name, setName] = useState<string | undefined>();
  const [score, setScore] = useState(0);
  const [loading, setLoading] = useState(true);

  useEffect( ... );

  return null;
}
```

2. Using state Hooks

Using useState

- Carry out the following steps
  - Change the effect function to set the loading and name state values after the person data has been fetched.

```
useEffect(() => {
  getPerson().then((person) => {
    setLoading(false);
    setName(person.name);
  });
}, []);
```

2. Using state Hooks

Using useState

- Carry out the following steps
  - Add the following if statement in between the useEffect call and the return statement:

```
useEffect( ... );
if (loading) {
  return <div>Loading ...</div>;
}
return ...
```

2. Using state Hooks

Using useState

- Carry out the following steps
  - Change the component's return statement:

```
if (loading) {
  return <div>Loading ...</div>;
}
return (
  <div>
    <h3>
      {name}, {score}
    </h3>
    <button>Add</button>
    <button>Subtract</button>
    <button>Reset</button>
  </div>
);
```

2. Using state Hooks

Using useState

- Carry out the following steps
  - Update the Add button so that it increments the score when clicked:

<button **onClick={() => setScore(score + 1)}**>Add</button>

Or

setScore(**previousScore => previousScore + 1**)

  - Add score state setters to the other buttons as follows:

```
<button onClick={() => setScore(score - 1)}>Subtract</button>
<button onClick={() => setScore(0)}>Reset</button>
```

  - Run app, click the buttons, and check the result

2. Using state Hooks

Using useState

- Carry out the following steps
  - Let’s take some time to understand when the state values are actually set. Update the effect function to output the state values after they are set:

```
useEffect(() => {
  getPerson().then((person) => {
    setLoading(false);
    setName(person.name);
    console.log("State values", loading, name);
  });
}, []);
```

  - Updating state values is not immediate – instead, they are batched and updated before the next render.

2. Using state Hooks

Understanding useReducer

- **useReducer** is an alternative method of managing state. It uses a **reducer function** for state changes, which **takes in the current state value** and **returns the new state value**.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- The **dispatch function** takes in **an argument that describes the change**. This object is called an **action**.

```
dispatch({ type: 'add', amount: 2 });
```

2. Using state Hooks

Understanding useReducer

- The **spread syntax** (…state) **copies all the properties from the object** after the three dots.
- The **reducer** function has parameters for the current state value and the action.

```
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'add':
      return { ...state, total: state.total + action.amount };
    case ...
      ...
    default:
      return state;
  }
}
```

2. Using state Hooks

Understanding useReducer

- The types for useReducer can be explicitly defined in its generic parameter:

```
const [state, dispatch] = useReducer<Reducer<State, Action>>(
  reducer,
  initialState
);
```

2. Using state Hooks

Recap

Understanding useReducer

- useReducer is more complex than useState because state changes go through a reducer function that we must implement.
- This benefits complex state objects with related properties or when a state change depends on the previous state value.

2. Using state Hooks

Using useReducer

- Carry out the following steps:
  - Open **PersonScore.tsx** and import useReducer instead of useState from React:

    import { useEffect, useReducer } from 'react';

  - Define a type for the state beneath the import statements:

```
type State = {
    name: string | undefined;
    score: number;
    loading: boolean;
}
```

2. Using state Hooks

Using useReducer

- Carry out the following steps:
  - Define types for all the **action objects**:

```
type Action =
  | {
      type: 'initialize';
      name: string;
    }
  | {
      type: 'increment';
    }
  | {
      type: 'decrement';
    }
  | {
      type: 'reset';
    };
```

2. Using state Hooks

Using useReducer

- Carry out the following steps:
  - Define the following **reducer function** underneath the type definitions:

```
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'initialize':
      return { name: action.name, score: 0, loading:
false };
    case 'increment':
      return { ...state, score: state.score + 1 };
    case 'decrement':
      return { ...state, score: state.score - 1 };
    case 'reset':
      return { ...state, score: 0 };
    default:
      return state;
  }
}
```

2. Using state Hooks

Using useReducer

- Carry out the following steps
  - Inside the **PersonScore** component, replace the useState calls with the following useReducer call:

```
const [{ name, score, loading }, dispatch] = useReducer(
  reducer,
  {
    name: undefined,
    score: 0,
    loading: true,
  }
);
```

2. Using state Hooks

Using useReducer

- Carry out the following steps
  - Update the effect function and dispatch an initialize action:

```
useEffect(() => {
  getPerson().then(({ name }) =>
    dispatch({ type: 'initialize', name })
  );
}, []);
```

2. Using state Hooks

Using useReducer

- Carry out the following steps
  - Dispatch the relevant actions in the button click handlers:

```
<button onClick={() => dispatch({ type: 'increment' })}>
  Add
</button>
<button onClick={() => dispatch({ type: 'decrement' })}>
  Subtract
</button>
<button onClick={() => dispatch({ type: 'reset' })}>
  Reset
</button>
```

2. Using state Hooks

Using useReducer

- The useReducer Hook is **more useful for complex state management situations** than useState, for example, when **the state is a complex object** with related properties and state changes depend on previous state values.
- The **useState Hook** is more appropriate when the state is based on **primitive values** independent of any other state.

3. Using the ref Hook

Understanding the ref Hook

- The ref Hook is called useRef and it **returns a variable whose value is persisted for the lifetime of a component**. This means that the variable doesn't lose its value when a component re-renders.
- The value returned from the ref Hook is often referred to as a **ref**. The ref can be changed without causing a re-render.
- Syntax:

```
const ref = useRef(initialValue);
```

3. Using the ref Hook

Understanding the ref Hook

- An initial value can optionally be passed into useRef. The type of the ref can be explicitly defined in a **generic argument** for useRef:

  const ref = useRef<Ref>(initialValue);

- The generic argument is useful when no initial value is passed or is null.
- The value of the ref is accessed via its **current** property:

  console.log(“Current ref value”, **ref.current**);

- The value of the ref can be updated via its **current** property:

  ref.current = newValue;

3. Using the ref Hook

Understanding the ref Hook

- A common use of the useRef Hook is to access HTML elements imperatively. HTML elements have a **ref** attribute in JSX that can be assigned to a ref.
- The following is an example of this:

```
function MyComponent() {
  const inputRef = useRef<HTMLInputElement>(null);
  function doSomething() {
    console.log(
      "All the properties and methods of the input",
      inputRef.current
    );
  }
  return <input ref={inputRef} type="text" />;
}
```

3. Using the ref Hook

Using the ref Hook

- Carry out the following steps:
  - Open **PersonScore.tsx** and import useRef from React:

    import { useEffect, useReducer, useRef } from 'react';
  - Create a ref for the Add button just below the useReducer statement:

```
const [ ... ] = useReducer( ... );

const addButtonRef = useRef<HTMLButtonElement>(null);

useEffect( ... )
```

  - **Note:** All the standard HTML elements have corresponding TypeScript types for React. Right-click on the HTMLButtonElement type and choose Go to Definition to discover all these types.

3. Using the ref Hook

Using the ref Hook

- Carry out the following steps:
  - Assign the ref to the ref attribute on the Add button JSX element:

```
<button
  ref={addButtonRef}
  onClick={() => dispatch({ type: 'increment' })}
>
  Add
</button>
```

3. Using the ref Hook

Using the ref Hook

- Carry out the following steps:
  - Invoke **focus** method to move the focus to the Add button when the person's information has been fetched.

```
useEffect(() => {
  getPerson().then(({ name }) => dispatch({ type: 'initialize', name }));
}, []);
useEffect(() => {
  if (!loading) {
    addButtonRef.current?.focus();
  }
}, [loading]);

if (loading) {
  return <div>Loading ...</div>;
}
```

3. Using the ref Hook

Using the ref Hook

- Carry out the following steps:
  - We could have moved the focus to the Add button in the existing effect as follows:

```
useEffect(() => {
  getPerson().then(({ name }) => {
    dispatch({ type: 'initialize', name });
    addButtonRef.current?.focus();
  });
}, []);
```

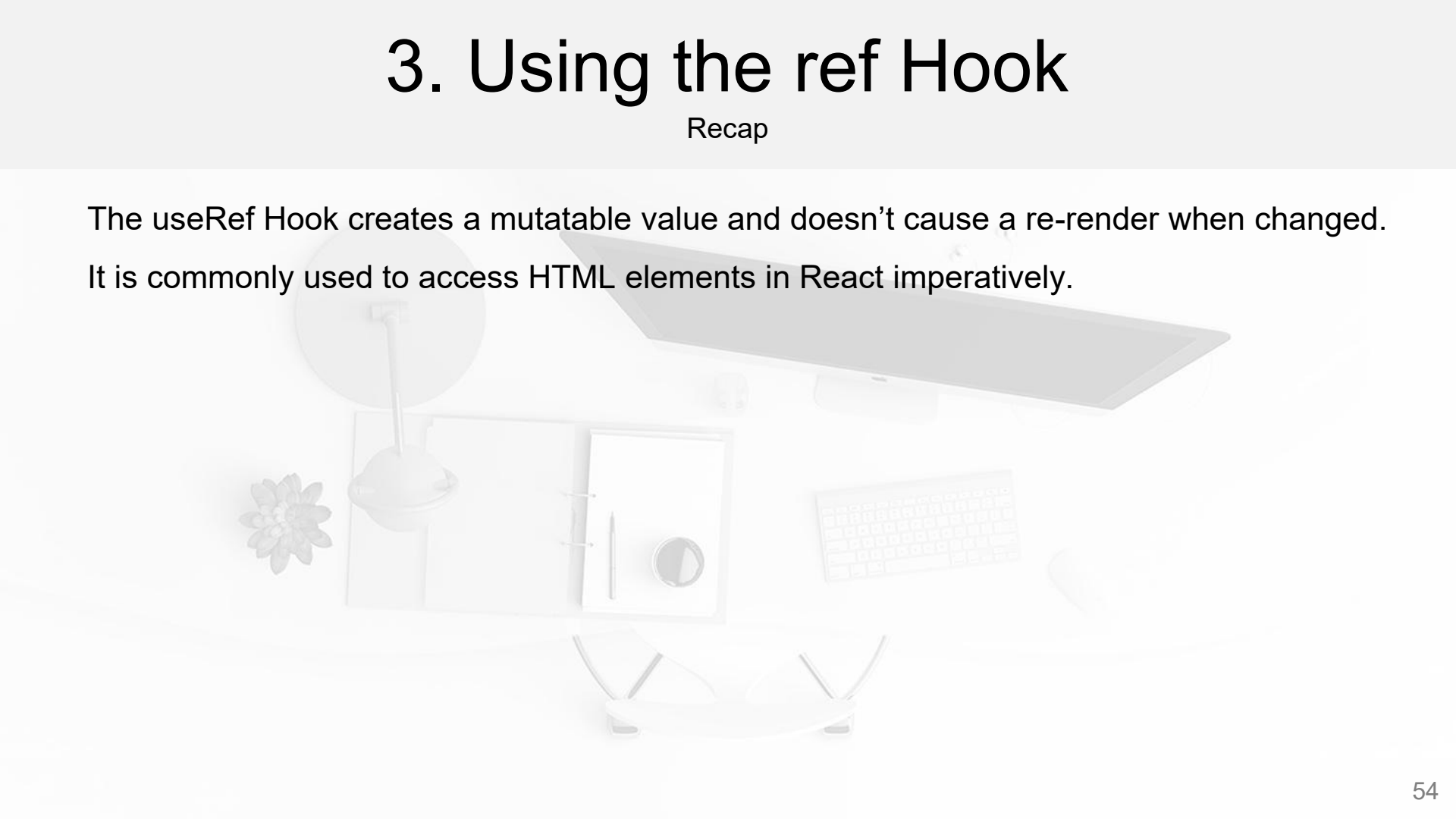
  - If you refresh the browser containing the running app, you will see a focus indicator on the Add button.

3. Using the ref Hook

Recap

The useRef Hook creates a mutatable value and doesn’t cause a re-render when changed.

It is commonly used to access HTML elements in React imperatively.



4. Using the memo Hook

Understanding the memo Hook

- The memo Hook **creates a memoized value** and is **beneficial for values** that have **computationally expensive calculations**.
- Syntax:

```
const memoizedValue = useMemo(() => expensiveCalculation(), []);
```

  - The **first argument** is a function that **returns the value to memoize**. It should perform the expensive calculation.
  - The **second argument** is an **array of dependencies**. When any dependencies change, the function in the first argument is executed again to return a new value to memoize.

4. Using the memo Hook

Understanding the memo Hook

- The type of the memoized value is inferred but can be explicitly defined in a **generic parameter** on useMemo.

```
const memoizedValue = useMemo<number>(
  () => expensiveCalculation(),
  []
);
```

4. Using the memo Hook

Using the memo Hook

- Carry out the following steps:
  - Open **PersonScore.tsx** and import useMemo from React:

```
import { useEffect, useReducer, useRef, useMemo } from 'react';
```

  - Add the following expensive function below the import statements:

```
function sillyExpensiveFunction() {
  console.log("Executing silly function");
  let sum = 0;
  for (let i = 0; i < 10000; i++) {
    sum += i;
  }
  return sum;
}
```

4. Using the memo Hook

Using the memo Hook

- Carry out the following steps:
  - Add a call to the function in the PersonScore component beneath the effects:

```
useEffect( ... );

const expensiveCalculation = sillyExpensiveFunction();

if (loading) {
  return <div>Loading ...</div>;
}
```

4. Using the memo Hook

Using the memo Hook

- Carry out the following steps:
  - Add the result of the function call to the JSX underneath name and score:

```
<h3>
  {name}, {score}
</h3>
<p>{expensiveCalculation}</p>
<button ... >
  Add
</button>
```

  - Refresh the browser containing the app and click the buttons. Look in the console and give some comments.

4. Using the memo Hook

Using the memo Hook

- Carry out the following steps:
  - Rework the call to sillyExpensiveFunction as follows:

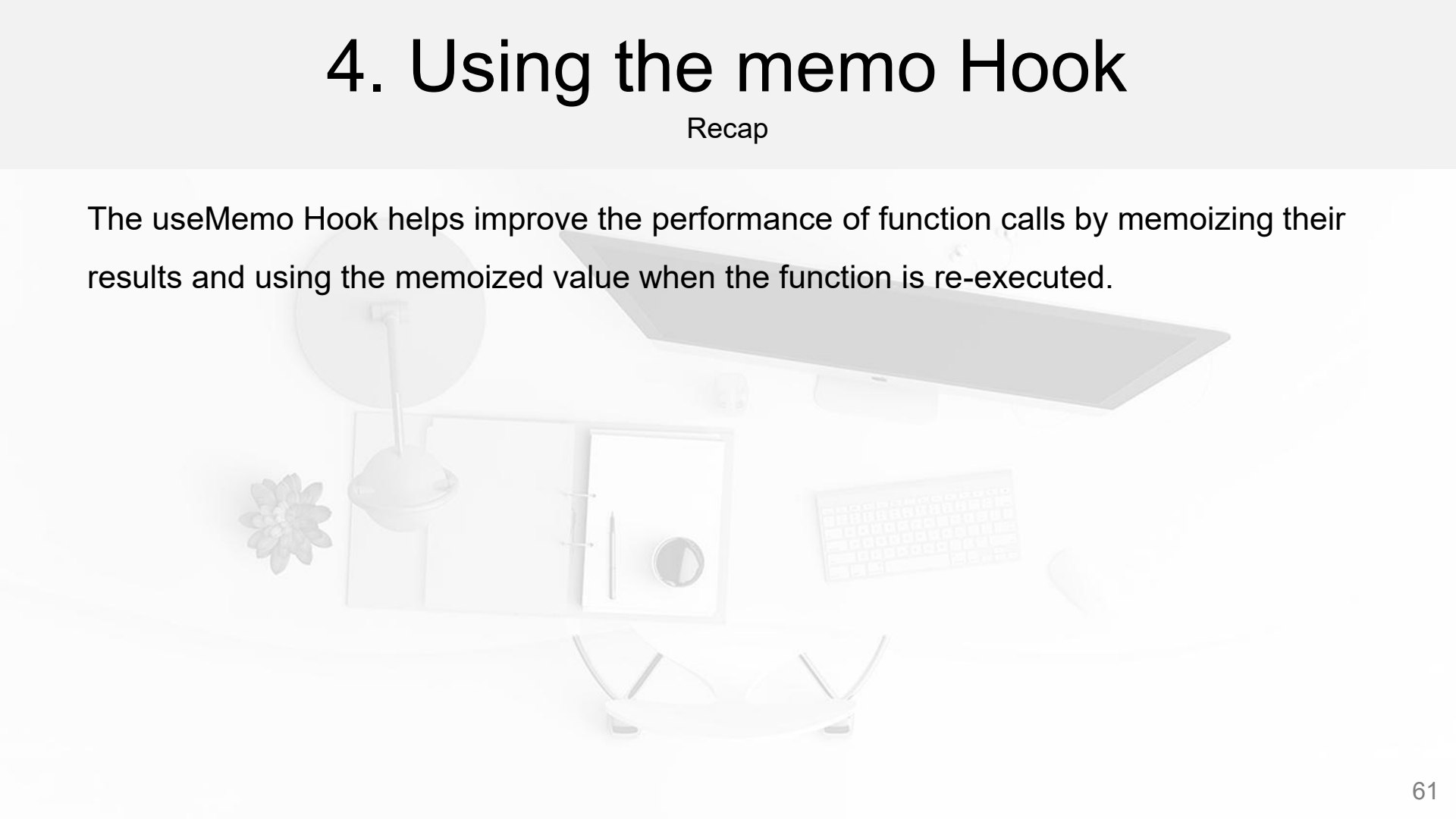
```
const expensiveCalculation = useMemo(
  () => sillyExpensiveFunction(),
  []
);
```

  - The useMemo Hook is used to memoize the value from the function call.
  - Refresh the browser containing the running app and click the buttons. Look in the console and give some comments.

4. Using the memo Hook

Recap

The useMemo Hook helps improve the performance of function calls by memoizing their results and using the memoized value when the function is re-executed.



5. Using the callback Hook

Understanding the callback Hook

- The callback Hook **memoizes a function** so that it isn't recreated on each render.
- Syntax:

```
const memoizedCallback = useCallback(() => someFunction(), []);
```

- The **first argument** is a function that **executes the function to memorize**.
- The **second argument** is an **array of dependencies**. When any dependencies change, the function in the first argument is executed again to return a new function to memoize

5. Using the callback Hook

Understanding the callback Hook

- The type of the memoized function is inferred but can be explicitly defined in a generic parameter on useCallback.
- An example of explicitly defining that the memoized function has no parameters and returns void:

```
const memoizedValue = useCallback<() => void>(
  () => someFunction (),
  []
);
```

5. Using the callback Hook

Understanding when a component is re-rendered

- A component re-renders when its state changes.
- Consider the following component:

```
export function SomeComponent() {
  const [someState, setSomeState] = useState('something');
  return (
    <div>
      <ChildComponent />
      <AnotherChildComponent something={someState} />
      <button
        onClick={() => setSomeState('Something else')}
      ></button>
    </div>
  );
}
```

5. Using the callback Hook

Understanding when a component is re-rendered

- Re-rendering behavior will cause performance problems – particularly when a component is rendered near the top of a large component tree.
- The DOM will only be updated after a re-render if the virtual DOM changes.

5. Using the callback Hook

Understanding when a component is re-rendered

- **The DOM** for ChildComponent **won't be updated** when SomeComponent is re-rendered if it is defined as follows:

```
export function ChildComponent() {
  return <span>A child component</span>;
}
```

5. Using the callback Hook

Understanding when a component is re-rendered

- While this re-rendering behavior generally **doesn't cause performance problems**, it can cause performance issues if a computationally expensive component is frequently re-rendered or a component with a slow side effect is frequently re-rendered.
- There is **a function called memo** in React that can be used **to prevent unnecessary re-renders**.

```
export const ChildComponent = memo(() => {
  return <span>A child component</span>;
});
```

5. Using the callback Hook

Using the callback Hook

- **Description**: We will now **refactor** the **PersonScore component** by **extracting the Reset button** into a separate component called Reset. This will **lead to unnecessary re-rendering** of the Reset component, which we will resolve using React's memo function and the useCallback Hook.

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Create a new file **Reset.tsx** in the src folder for the reset button:

```
type Props = {
  onClick: () => void;
};
export const Reset = memo(({ onClick }: Props) => {
  console.log('render Reset');
  return <button onClick={onClick}>Reset</button>;
});
```

  - Open **PersonScore.tsx** and import the Reset component:

```
import { Reset } from './Reset';
```

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Replace the existing reset button with the new Reset component as follows:

```
<div>
  ...
  <button onClick={() => dispatch({ type: 'decrement'
})}>
    Subtract
  </button>
  <Reset onClick={() => dispatch({ type: 'reset' })} />
</div>;
```

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Go to the app running in the browser and open React's DevTools. Make sure the "**Highlight updates when components render."** option is ticked in the **Components** panel's settings:
  - Click Reset button as well as the Add and Subtract buttons. If you look at the console you'll notice that **Reset is unnecessarily re-rendered** (the re-render highlight around the Reset button).
  - Use the browser's DevTools to inspect the DOM. Click the buttons and see that only the h3 element content was updated .

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - **Add React's memo** function to try **to prevent unnecessary re-renders**. Open **Reset.tsx** and import memo at the top of the file:

    import { memo } from 'react';

  - Wrap **memo** around the Reset component:

```
export const Reset = memo(({ onClick }: Props) => {
  console.log("render Reset");
  return <button onClick={onClick}>Reset</button>;
});
```

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Add the following line beneath the Reset component definition so that it has a meaningful name in React's DevTools:

```
Reset.displayName = 'Reset';
```

  - In the browser, click the Add, Subtract, and Reset buttons. Then, look at the console and give some comments.

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Open React's DevTools > **Profiler** panel and click the cog icon to open the settings. Go to the **Profiler** settings section and make sure **Record why each component rendered while profiling.** is ticked.
  - Click the blue circle icon to start profiling and then click the **Add** button in our app. Click the red circle icon to stop profiling.
  - In the flamegraph that appears, click the Reset bar. This gives useful information about the Reset component re-render.

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Use the **useCallback Hook to memoize the onClick handler and prevent the re-render**. Open **PersonScore.tsx** and start by refactoring the handler into a named function:

```
function handleReset() {
  dispatch({ type: 'reset' });
}

if (loading) {
  return <div>Loading ...</div>;
}

return (
  <div>
    ...
    <Reset onClick={handleReset} />
  </div>
);
```

5. Using the callback Hook

Using the callback Hook

- Carry out the following steps:
  - Add useCallback to the React import statement:

    import {useEffect, useReducer, useRef, useMemo, useCallback} from 'react';
  - Lastly, wrap useCallback around the click handler we just created:

```
const handleReset = useCallback(
  () => dispatch({ type: 'reset' }),
  []
);
```

  - If you click the Add, Subtract, and Reset buttons, Reset is no longer unnecessarily re-rendered.

5. Using the callback Hook

Recap

Using the callback Hook

- **A component is re-rendered when its parent is re-rendered**.
- React's **memo** function can be used **to prevent unnecessary re-renders to child components.**
- **useCallback can be used to memoize functions**. This can be used to create a stable reference for function props passed to child components to prevent unnecessary re-renders.
- React's memo function and useCallback should be used wisely – make sure they help performance before using them because they increase the complexity of the code.

Summary

- All **React Hooks** must be called at the top level of a function component and can't be called conditionally.
- The **useEffect** Hook can be used to execute component side effects when it is rendered.
- **useReducer** is an alternative to useState for using state. useState is excellent for primitive state values. useReducer is great for complex object state values, particularly when state changes depend on previous state values.
- The **useRef** Hook creates a mutatable value and doesn't cause a re-render when changed.
- The **useMemo** and **useCallback** Hooks can be used to memoize values and functions, respectively, and can be used for performance optimization.


Q&A