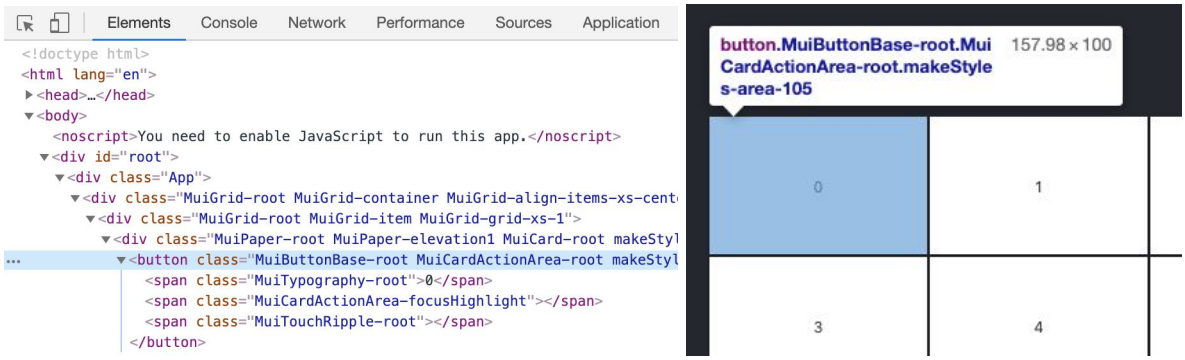
7. React - hooks

Lifecycle of component

So let’s have a closer look at the main job of React (but also other js frameworks), in which point React component comes into existence, what happens at this point and when the component is removed from DOM.
The main job of React is to figure out how to modify the DOM to match what the components want to be rendered on the screen.



We created a simple Tic-Tac-Toe game, where we have the following tree of main components (some components are ignored for clarity): `App -> Board -> Square -> PlayerIcon -> Typography / Cross / Circle`.
We also created a condition if we wanted to show a Cross icon, a Circle icon or index inside the button.

First render (Mounting)

Everything is initialized into default state and rendered as follows.

1. `index.tsx` as a starting point renders App component into `<div id=”root” />`.
2. App renders Board.
3. Board state is initialized with an array of numbers, player state is initialized with “X” and renders 9 Square components with PlayerIcons inside.
4. Each PlayerIcon receives a number, so no icons are rendered at the beginning.

At the moment of component being added to DOM, the component is ***mounting***.

It needs to be mentioned, because usage in practice is very common, while using **class** components, there are lifecycle methods fired during first render in this order (some of them are missing, only needed in advanced usage, out of scope of this course):

- **constructor** - initialize stuff like state
- **render** - only required method in class component
- **componentDidMount** - invoked only once right after first render

Since we are focusing on function components only, these methods are not available to us, and the concepts are slightly different, which we’ll come back to later.

Rerender (Updating)

So what happens after we click on a Square with number 5 on it?

1. Square 5 invokes the onSquareClicked(5) handler passed to it's onClick callback which updates boardValues and also switches player to "O".
2. Now since the state of the Board component has changed, it needs to be rerendered.
3. Code inside its body is run again and all its children are also rerendered.
4. Square 5 now receives different children and therefore renders a Cross icon.

In class component the lifecycle methods are invoked in the order:

- **render** - invoked with new state, props
- **componentDidUpdate** - method invoked after render, but careful Icarus, there is danger of circular rerendering

Unmounting

And what happens to component Typography, when it is replaced with a Cross icon? Simply, it's destroyed, removed, deleted…

There is one lifecycle method invoked before component is unmounted:

- **componentWillUnmount** - use it when you want to clean something after yourself

Hooks - lifecycle of function components

Hooks are a new concept in React world that lets you use state and other lifecycle related features without using class components. Hooks are plain JavaScript functions that need to follow few important rules:

1. **They need to always be called each rerender**, they can't be called inside loops, conditions or nested functions.

```
const Board: FC<Props> = ({ isLoggedIn }) => {
  if (!isLoggedIn) {
    return <div>Oooops</div>;
  }
  // ERROR: If isLoggedIn is false, state hooks below wouldn't be called
  // State
  const [player, setPlayer] = useState<Player>('X');
  const [board, setBoard] = useState<BoardState>({});
  // ...
```

2. **Must be used by Function Components or other hooks**, don't call them from regular JS functions. To properly function they need to be called by React render loop.

"useState" hook

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

During the first render - **mounting phase**, the `state` value is taken from the value of initialState argument. The `setState` is a function used to replace `state` with a new value which triggers a rerender of the component.

During subsequent rerenders - **updating phase**, `state` value stays the same and `initialState` is no longer used.

Functional updates

Setters also accept a function that receives the previous state in its first argument. Example from our Tic Tac Toe assignment.

```
setPlayer(player === 'O' ? 'X' : 'O');
setPlayer(p => (p === 'O' ? 'X' : 'O'));
```

Both examples usually give the same result, so the question is, why would I want to use the longer one? Well imagine a situation, where you want to call setPlayer from a child component. With the first approach, you need to pass both the value and the setter to child the component. With the second approach, there is no need to pass player prop because setPlayer has the information about the previous player, this is a way to avoid unnecessary prop drilling.

Merging state

Important information about the state setting is that setState function (unlike with class component state) does not automatically merge update objects. That's why we need to set new board values in our Tic Tac Toe assignment this way.

```
setBoard(b => ({ ...b, [index]: player }));
```

“useEffect” hook

```
useEffect(() => { /* Do stuff */ }, [/* Dependencies */]);
```

This hook lets you run code in function components only when necessary such as performing side effects like data fetching, setting up a subscription and so on. You can think of this hook as componentDidMount, componentDidUpdate and componentWillUnmount from class components combined.

Example usage of hooks can be added into `Status` component:

```
const Status: FC<Props> = ({ player, winner, onRestart }) => {
  const classes = useStyles({ winner });
  useEffect(() => {
    document.title = `Player ${player}'s turn`;
  });
  // ...
```

By adding this hook, React knows that the Status component needs to do something after render. In this case setup document title. Placing useEffect inside the component lets us access the name prop from the Status function, because the variable is already in a scope. useEffect is by default called right after first render and after every update, in other words useEffect is executed after mounting and also updating.

Dependency array

So far we would be able to achieve the same goal by simply putting all the code inside the hook into the body of the function. To use this hook properly, we need to provide a dependency array as its second argument.

```
useEffect(() => {
  document.title = `Player ${player}'s turn`;
}, [player]);
```

Each rerender, React compares values in this array with the values from previous render and only runs the provided function if these dependencies have changed. In this case, we have a

dependency on `player` prop of `Status` component, which only changes when the `Board` component passes another player to it.

Keep in mind that dependencies of useEffect are compared by reference equality, which means that in the example below, variable copy changes every render because each time it is a new array instance even though it's value is the same.

```
const copy = [...original];
useEffect(() => {
  console.log(`Copy changed.`);
}, [copy]);
```

If you want useEffect hook to be called **exactly once** at the moment of mounting, pass an **empty array** as a second argument. This is not the same as **not providing** a dependencies array since in that case the code runs **every** rerender.

```
useEffect(() => {
  document.title = `Mounting Title FTW`;
}, []);
```

Cleanup

At the moment of unmounting of the component, an optional **cleanup** function is called. Common us case for it is for example with event listener functions.

```
const [scroll, setScroll] = useState(window.scrollY);
useEffect(() => {
  const scrollHandler = () => setScroll(window.scrollY);
  window.addEventListener('scroll', scrollHandler);
  return () => {
    window.removeEventListener('scroll', scrollHandler);
  };
}, []);
```

Implementation

In our Tic Tac Toe app we are calling the `setWinner` setter function inside the `onSquareClicked` handler with the new value of `getWinner` every time a square is clicked.

```
const onSquareClicked = (index: Indexes) => {
  if (board[index]) return;
  const newBoard = { ...board, [index]: player };
  setBoard(newBoard);
  setPlayer((p) => (p === "O" ? "X" : "O"));

  // Update winner
  setWinner(getWinner(newBoard));
};
```

We also need to call it inside the `onBoardRestart` handler to set it back to undefined.

```
const onBoardRestart = () => {
    setPlayer("X");
    setBoard({});
    setWinner(undefined);
};
```

This behavior can be optimized by useEffect. Since board is in the dependency list, this effect will run each time it's value changes, no matter from what source.

```
const onSquareClicked = (index: Indexes) => {
  if (board[index]) return;
  setBoard(b => ({ ...b, [index]: player }));
  setPlayer((p) => (p === "O" ? "X" : "O"));
};

const onBoardRestart = () => {
  setPlayer("X");
  setBoard({});
```

```
};

// Update winner
useEffect(() => {
  setWinner(getWinner(board));
}, [board]);
```

“useMemo” hook

It’s quite similar to useEffect, with one major difference, in that it returns a value. It is used to speed up and to optimise your app by storing the results of expensive function calls by returning the cached result when it’s dependencies don’t change between renders. Inside useMemo, there should not be any side effects running, they should be performed within the useEffect hook instead.

Implementation

With this hook we can simplify our Board component even further. Since the winner is completely dependent on the current board state and calculating it has no side effects, instead of holding it as state we can just memoize it.

```
// State
const [player, setPlayer] = useState<Player>("X");
const [board, setBoard] = useState<BoardState>({});

// Memoization
const winner = useMemo(() => getWinner(board), [board]);

// Handlers
const onSquareClicked = (index: Indexes) => {
  if (board[index]) return;
  setBoard(b => ({ ...b, [index]: player }));
  setPlayer((p) => (p === "O" ? "X" : "O"));
};
```

```
const onBoardRestart = () => {
  setPlayer("X");
  setBoard({});
};
```

“useCallback” hook

Specialized version of useMemo hook, used for memoizing functions. Most commonly it’s used for handlers that are passed as props to other components to prevent unnecessary rerenders.

`useCallback(fn, deps)` is equivalent to `useMemo(() => fn, deps)`.

Implementation

In our app we have two handlers that are passed to child components, `onSquareClicked` and `onBoardRestart`. With our new knowledge of useCallback hook we can now attempt to improve these as well. It may seem like we can simply wrap both in useCallback and call it a day, but it’s also important to think about the usage and benefits.

```
const onBoardRestart = useCallback(() => {
  setPlayer('X');
  setBoard({});
}, []);

// Usage
<Status player={player} winner={winner} onRestart={onBoardRestart} />
```

After converting `onBoardRestart` to useCallback, we can see that it has no dependencies and it’s also directly passed to `onRestart` prop as a value. This means that now the `Status` component will no longer be unnecessarily re-rendering because it was receiving new reference to our previously non hook callback. So far so good.

```
const onSquareClicked = useCallback((index: Indexes) => {
  if (board[index]) return;
  setBoard(b => ({ ...b, [index]: player }));
```

```
  setPlayer(p => (p === 'O' ? 'X' : 'O'));
}, [board]);

// Usage
<Square onClick={() => onSquareClicked(i)}>
```

Second callback now has a dependency to the `board` state value which is okay. What's complicating it now is it's usage. This time it's not passed as a value but called inside an inline arrow function. Because of this, the `Square` component receives different reference each render anyway. In this case there is **no benefit** from using useCallback so we are better off just keeping the handler as it was before.

Custom hook

React provides also other types of hooks, it is even possible to create your own ones. Let's begin with something trivial.

```
const App: FC = () => {
  // Number counter
  const [counter, setCounter] = useState(0);
  // Effect to set number into Title
  useEffect(() => {
    document.title = `${counter}`;
  }, [counter]);
  // Handler for button onClick method
  const incrementCounter = () => setCounter(p => ++p);

  return (
    <div className='App'>
      <header className='App-header'>
        <img src={logo} className='App-logo' alt='logo' />
        <p>{counter}</p>
        <button onClick={incrementCounter}>CLICK</button>
      </header>
    </div>
  );
```

```
};
```

To create a custom hook we just copy out logic from the App component into a single function. One important rule about creating custom hooks is that their name needs to start with “use”. (like useFoo, useCounter, useEtc…), otherwise App fails to compile with an error:

React Hook "useState" is called in function "counter" which is neither a React function component nor a custom React Hook function

After extracting all the logic, out custom hook may look something like this:

```
// Initial value can be passed to the hook as an argument
function useCounter(initialValue = 0) {
  // Number counter
  const [counter, setCounter] = useState(initialValue);
  // Effect to set number into Title
  useEffect(() => {
    document.title = `${counter}`;
  }, [counter]);
  // Handler for button onClick method
  const incrementCounter = () => setCounter(p => ++p);

  // Return only variables that we need to use outside of hook
  return [counter, incrementCounter] as const;
}
```

And finally it’s usage in the App component.

```
function App() {
  const [counter, incrementCounter] = useCounter(5);

  return (
    <div className='App'>
      <header className='App-header'>
        <img src={logo} className='App-logo' alt='logo' />
        <p>{counter}</p>
```

```
        <button onClick={incrementCounter}>CLICK</button>
      </header>
    </div>
  );
}
```