Hooks!

Hooks!

What are they good for?

Hooks!

What are they good for?

Absoloutely Everything!

How we write components

```
function QuantitySelector() {
  return (
    <div>
      <button onClick={() => null}>-</button>
      <input type="number" value={"1"} />
      <button onClick={() => null}>+</button>
    </div>
  );
}
```

❌ We would like our `QuantitySelector` component to … work. Unfortunately, function components don't provide access to state or lifecycle methods.

Class based component

```
class QuantitySelector extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      quantity: 1
    };
  }

  render() {
    return (
      <div>
        <button onClick={() => null}>-</button>
        <input type="number" value={"1"} />
        <button onClick={() => null}>+</button>
      </div>
    );
  }
}
```

Working class based component

```
class QuantitySelector extends Component {
  constructor(props) {
    super(props);
    this.state = {
      quantity: 1
    };

    this.incrementQuantity = this.incrementQuantity.bind(this);
    this.decrementQuantity = this.decrementQuantity.bind(this);
  }

  incrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity + 1
    }));
  }

  decrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity - 1
    }));
  }

  render() {
    return (
      <div>
        <button onClick={this.decrementQuantity}>-</button>
        <input type="number" readOnly value={this.state.quantity} />
        <button onClick={this.incrementQuantity}>+</button>
      </div>
    );
  }
}
```

Working class based component

```
class QuantitySelector extends Component {
  constructor(props) {
    super(props);
    this.state = {
      quantity: 1
    };

    this.incrementQuantity = this.incrementQuantity.bind(this);
    this.decrementQuantity = this.decrementQuantity.bind(this);
  }

  incrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity + 1
    }));
  }

  decrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity - 1
    }));
  }

  render() {
    return (
      <div>
        <button onClick={this.decrementQuantity}>-</button>
        <input type="number" readOnly value={this.state.quantity} />
        <button onClick={this.incrementQuantity}>+</button>
      </div>
    );
  }
}
```

Working class based component

```
class QuantitySelector extends Component {
  constructor(props) {
    super(props);
    this.state = {
      quantity: 1
    };

    this.incrementQuantity = this.incrementQuantity.bind(this);
    this.decrementQuantity = this.decrementQuantity.bind(this);
  }

  incrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity + 1
    }));
  }

  decrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity - 1
    }));
  }

  render() {
    return (
      <div>
        <button onClick={this.decrementQuantity}>-</button>
        <input type="number" readOnly value={this.state.quantity} />
        <button onClick={this.incrementQuantity}>+</button>
      </div>
    );
  }
}
```

Working class based component

```
class QuantitySelector extends Component {
  constructor(props) {
    super(props);
    this.state = {
      quantity: 1
    };

    this.incrementQuantity = this.incrementQuantity.bind(this);
    this.decrementQuantity = this.decrementQuantity.bind(this);
  }

  incrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity + 1
    }));
  }

  decrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity - 1
    }));
  }

  render() {
    return (
      <div>
        <button onClick={this.decrementQuantity}>-</button>
        <input type="number" readOnly value={this.state.quantity} />
        <button onClick={this.incrementQuantity}>+</button>
      </div>
    );
  }
}
```

Working class based component

```
class QuantitySelector extends Component {
  constructor(props) {
    super(props);
    this.state = {
      quantity: 1
    };

    this.incrementQuantity = this.incrementQuantity.bind(this);
    this.decrementQuantity = this.decrementQuantity.bind(this);
  }

  incrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity + 1
    }));
  }

  decrementQuantity() {
    this.setState(state => ({
      quantity: state.quantity - 1
    }));
  }

  render() {
    return (
      <div>
        <button onClick={this.decrementQuantity}>-</button>
        <input type="number" readOnly value={this.state.quantity} />
        <button onClick={this.incrementQuantity}>+</button>
      </div>
    );
  }
}
```

Classes Considered Harmful

[1] https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-function-components.

Classes Considered Harmful

— Harder to test and reason about. Function components are pure and easy to test

[1] https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-function-components.

Classes Considered Harmful

- Harder to test and reason about. Function components are pure and easy to test
- Tie together behaviour and display. Not easily composable

[1] https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-function-components.

Classes Considered Harmful

- Harder to test and reason about. Function components are pure and easy to test
- Tie together behaviour and display. Not easily composable
- Related code does not live together; it's dispersed across `constrcutor`, `componentDidMount`, various handlers etc

[1] https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-function-components.

Classes Considered Harmful

- Harder to test and reason about. Function components are pure and easy to test
- Tie together behaviour and display. Not easily composable
- Related code does not live together; it's dispersed across `constrcutor`, `componentDidMount`, various handlers etc
- Won't benefit from future React optimizations for function components[1]

[1] https://reactjs.org/blog/2015/10/07/react-v0.14.html#stateless-function-components.

Introducing the `useState` hook

```
import React, { useState } from "react";
// in the top level scope of your function component:
const stateObjValue, setStateObjValue = useState(initialValue);
```

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

const [stateObjValue, setStateObjValue] = useState(initialValue);
// curr value -⬆️
```

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

const [stateObjValue, setStateObjValue] = useState(initialValue);
// function to update value ---⬆️
```

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

const [stateObjValue, setStateObjValue] = useState(initialValue);
// React library function -----------------⬆️
```

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

const [stateObjValue, setStateObjValue] = useState(initialValue);
// the initial value, can be anything ----------------⬆️
```

ℹ️ `useState` allows us to use local state in a function component

Apply useState to our class based component

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

function QuantitySelector() {
  const [quantity, setQuantity] = useState(1);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Apply useState to our class based component

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

function QuantitySelector() {
  const [quantity, setQuantity] = useState(1);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Apply useState to our class based component

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

function QuantitySelector() {
  const [quantity, setQuantity] = useState(1);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Apply useState to our class based component

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

function QuantitySelector() {
  const [quantity, setQuantity] = useState(1);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

❓ What about lifecycle methods (`componentDidMount`)

Let us add a feature in our component to update cart (a side effect) whenever the quantity is updated

Back to a class based approach

```
class QuantitySelector extends Component {
  constructor(props) {
    // …
  }

  componentDidMount() {
    CartAPI.setQuantity(this.state.quantity);
  }

  componentDidUpdate() {
    CartAPI.setQuantity(this.state.quantity);
  }

  incrementQuantity() {
    // …
  }

  decrementQuantity() {
    // …
  }

  render() {
    // …
  }
}
```

Back to a class based approach

```
class QuantitySelector extends Component {
  constructor(props) {
    // …
  }

  componentDidMount() {
    CartAPI.setQuantity(this.state.quantity);
  }

  componentDidUpdate() {
    CartAPI.setQuantity(this.state.quantity);
  }

  incrementQuantity() {
    // …
  }

  decrementQuantity() {
    // …
  }

  render() {
    // …
  }
}
```

💔 We need to get away from classes

Introducing the `useEffect` hook

```
import React, { useEffect } from "react";

// in the top level scope of your function component:
useEffect(() => {
  // imperative, effectful code.
  // will fire on mount and after every update
};
```

```
import React, { useEffect } from "react";

// in the top level scope of your function component:
useEffect(() => {
  // will only fire when `foo` or `bar` change
}, [foo, bar]);
```

```
import React, { useEffect } from "react";

// in the top level scope of your function component:
useEffect(() => {
  // will only fire on component mount
}, []);
```

Apply useEffect to our class based component

```
function QuantitySelector() {
  const [quantity, setQuantity] = useState(1);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Apply `useEffect` to our class based component

```
function QuantitySelector() {
  const [quantity, setQuantity] = useState(1);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

📵 Time to get freaky

Let's consolidate our behaviour into a custom hook!

```
function useQuantitySelect(initialQuantity) {
  const [quantity, setQuantity] = useState(initialQuantity);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return { quantity, setQuantity, incrementQuantity, decrementQuantity };
}
```

Let's consolidate our behaviour into a custom hook!

```
function useQuantitySelect(initialQuantity) {
  const [quantity, setQuantity] = useState(initialQuantity);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return { quantity, setQuantity, incrementQuantity, decrementQuantity };
}
```

Let's consolidate our behaviour into a custom hook!

```
function useQuantitySelect(initialQuantity) {
  const [quantity, setQuantity] = useState(initialQuantity);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return { quantity, setQuantity, incrementQuantity, decrementQuantity };
}
```

Let's consolidate our behaviour into a custom hook!

```
function useQuantitySelect(initialQuantity) {
  const [quantity, setQuantity] = useState(initialQuantity);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return { quantity, setQuantity, incrementQuantity, decrementQuantity };
}
```

Let's consolidate our behaviour into a custom hook!

```
function useQuantitySelect(initialQuantity) {
  const [quantity, setQuantity] = useState(initialQuantity);

  useEffect(() => {
    CartAPI.setQuantity(quantity);
  }, [quantity]);

  const incrementQuantity = () => setQuantity(quantity + 1);
  const decrementQuantity = () => setQuantity(quantity - 1);

  return { quantity, setQuantity, incrementQuantity, decrementQuantity };
}
```

Using a custom hook

```
function QuantitySelector() {
  const { quantity, incrementQuantity, decrementQuantity } = useQuantitySelect(
    1
  );

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Using a custom hook

```
function QuantitySelector() {
  const { quantity, incrementQuantity, decrementQuantity } = useQuantitySelect(
    1
  );

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Using a custom hook

```
function QuantitySelector() {
  const { quantity, incrementQuantity, decrementQuantity } = useQuantitySelect(
    1
  );

  return (
    <div>
      <button onClick={() => decrementQuantity()}>-</button>
      <input type="number" readOnly value={quantity} />
      <button onClick={() => incrementQuantity()}>+</button>
    </div>
  );
}
```

Other hooks

useContext

Returns the current value of the context.

```
const value = useContext(MyContext);
```

useReducer

A useful replacement for `getState` when the state object is complex or the logic for updating it is complex.

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

// get values from the state
const hasDiscont = state.price.hasDiscount;

// dispatch actions
dispatch({ type: "QUANTITY/INCR", payload: 1 });
```

useMemo

Returns a memoized value. For performance optimization - think `shouldComponentUpdate`:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

`useCallback`

Returns a memoized callback. For performance optimization - think `shouldComponentUpdate`:

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

useRef

Allows you to mutate the passed in ref object.

Useful for DOM manipulation, but it can also be used for working with any type of mutable value.

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

💖 And many more!

Benefits of hooks:

Benefits of hooks:

— static typing better than higher order components

Benefits of hooks:

- static typing better than higher order components
- keep related code together (eg, `componentDidMount` and `componentWillUnmount`)

Benefits of hooks:

- static typing better than higher order components
- keep related code together (eg, `componentDidMount` and `componentWillUnmount`)
- resue behaviour and state management

Benefits of hooks:

— static typing better than higher order components

— keep related code together (eg, `componentDidMount` and `componentWillUnmount`)

— resue behaviour and state management

— don't need deep nesting in the component tree - less work for React to do

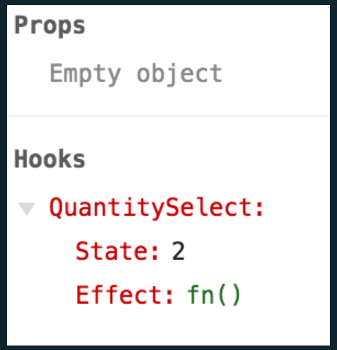
Benefits of hooks:

- static typing better than higher order components
- keep related code together (eg, `componentDidMount` and `componentWillUnmount`)
- resue behaviour and state management
- don't need deep nesting in the component tree - less work for React to do
- no overhead of creating classes and even handler binding

Benefits of hooks:

- static typing better than higher order components
- keep related code together (eg, `componentDidMount` and `componentWillUnmount`)
- resue behaviour and state management
- don't need deep nesting in the component tree - less work for React to do
- no overhead of creating classes and even handler binding
- good devtools support

Benefits of hooks:

- static typing better than higher order components
- keep related code together (eg, `componentDidMount` and `componentWillUnmount`)
- resue behaviour and state management
- don't need deep nesting in the component tree - less work for React to do
- no overhead of creating classes and even handler binding
- good devtools support
- 

```
Props
  Empty object

Hooks
▾ QuantitySelect:
    State: 2
    Effect: fn()
```

❓ So, what's the lowdown?

Hooks!

Hooks!

What are they good for?

Hooks!

What are they good for?

Absoloutely Everything!

Code from deck

— Repl.it project

Official docs

— React Hooks
— Hooks API Reference
— Hooks Rules

Other links

— Functional vs Class-Components in React
— Rangle | Refactor to React Hooks, Not Classes
— Rangle | Simplifying React Forms with Hooks
— State Management with React Hooks — No Redux or Context API
— How to fetch data with React Hooks?