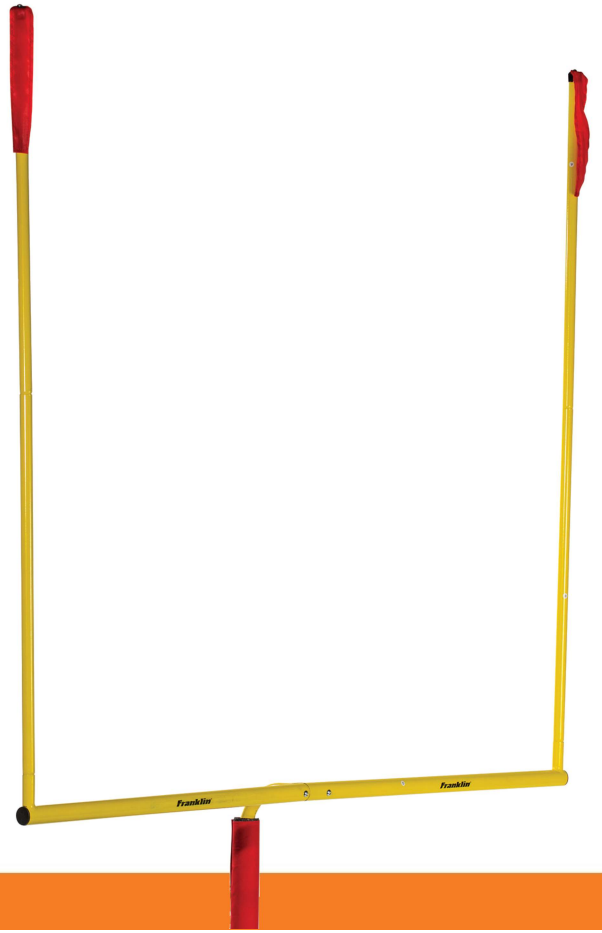
# State

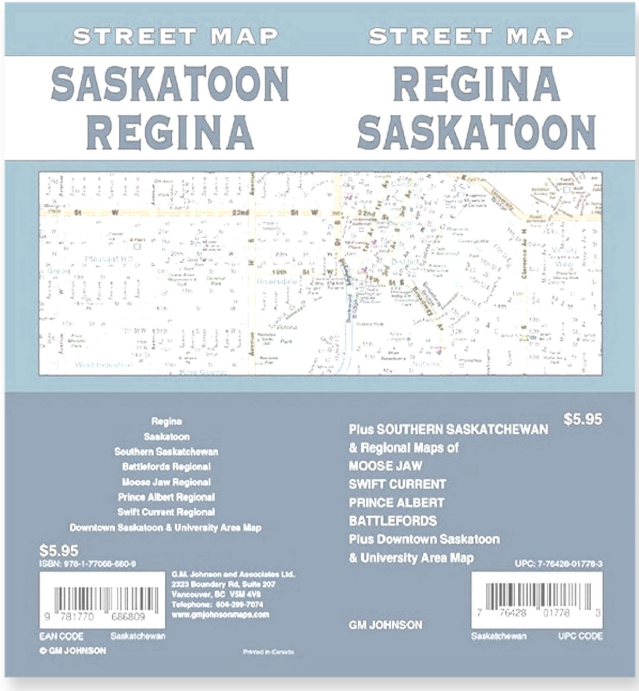
# Goals

1. Describe `useState`
2. Explain `useReducer`
3. List the **rules of hooks**



# Roadmap

1. Overview
2. Hooks
3. State Molecules
4. Reducers



Develop
Intelligence

# Overview

# What's **State**?



- **State** *is any data that directly effects what's on the screen.*
- Components are rendered under 2 conditions:
  - Initial render
  - Updated state
- Functional components are called for each render

# Motivating Example

Develop Intelligence

```
export const App = () => {
  let i = 1;
  return <>
          <h1>Your number is: {i}</h1>
          <button onClick={() => (i = i + 1)}>Increment</button>
        </>;
};
```

- It's just a function:
  - Invocation initializes local variables
  - Exiting recovers memory from the stack
- Nothing tells React that state has changed

# What we Need



- Functional components are just functions
- So we need
  - Something to keep track of state across multiple invocations of a functional component
  - Kind of like data members of a class

# `useState` to the Rescue

Develop Intelligence

- A 'hook' function lets you 'hook' into React behavior
- `useState` is a 'hook' providing
  - A **state variable** to retain the data between renders.
  - A **state setter** function to update the variable and trigger React to render the component again.
- ***Unbreakable rule:*** Only update state through its setter function

```
1 const [i,setI] = useState(1); //Array destructuring
```

# Fixed



- With `useState` even a fuctional component can remember

```
1  import { useState } from 'react';
2
3  export const App = () => {
4    const [i,setI] = useState(1);
5    return <>
6            <h1>Your number is: {i}</h1>
7            <button
8              onClick={() => setI(i+1)}
9            >
10             Increment
11           </button>
12         </>;
13 };
```

# useState Bloopers



## Why isn't this working?

```
export const App = () => {
  let [i,setI] = useState(1);
  return (
    <>
      <h1>Your number is: {i}</h1>
      <button onClick={() => i=i+1}>Increment</button>
    </>
  );
};
```

# State == Snapshot

Develop Intelligence

> *State variables might look like regular JavaScript variables that you can read and write to. However, state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render.*

-- React.dev

# Proof

Develop Intelligence

## Why isn't this working?

```
1  export const App = () => {
2    const [i, setI] = useState(1);
3    const handleClick = () => {
4      setI(i + 1);
5      setI(i + 1);
6      setI(i + 1);
7    };
8    return <>
9            <h1>Your number is: {i}</h1>
10           <button onClick={handleClick}>Increment</button>
11         </>;
12 };
```

Develop
Intelligence

# Hooks

# About Hooks

Develop Intelligence

- A *hook* is a special fuction
  - Only available during rendering
  - Has a `use` prefix
- Lets you 'hook into' react features
- The most fundamental hook is `useState` (maybe)

# Rules of Hooks

Develop Intelligence

- **Always** call a hook at the top level of a component or custom hook
- **Don't** call a hook
  - Inside conditions or loops
  - After a conditional return statement
  - From class components
- **Good news**: The linter is good about warnings

# Multiple useState



- How does React know which state is which?

```
export const UserEditor = () => {
  const [givenName, setGivenName] = useState('');
  const [surname, setSurname] = useState('');
  return <form>
          <h1>Hello, {givenName} {surname}</h1>
          <input type="text"
            onChange={event => setGivenName(event.target.value)}
            value={givenName}
          />
          <input type="text"
            value={surname}
            onChange={event => setSurname(event.target.value)}
          />
        </form>;
```

# Bad Ideas

Develop Intelligence

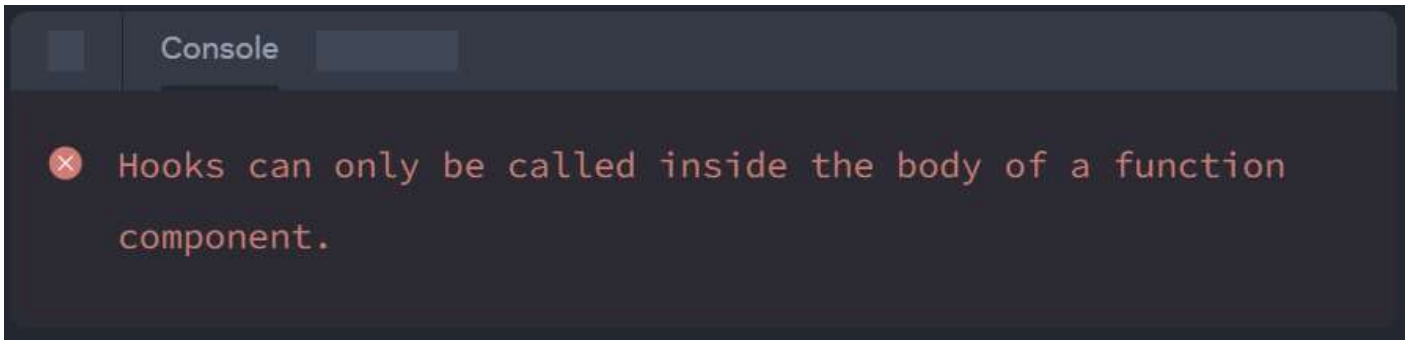
```
export const BadIdea = ({ hasTheme }) => {
  if (hasTheme) {
    const theme = useContext(ThemeContext);
  }
  // ...
}

export const AnotherBadIdea = ({ hidden }) => {
  if (hidden) {
    return;
  }
  const [clicks, setClicks] = useState(0);
  // ...
}
```

# Not *Normal* Functions

Develop Intelligence

**More good news: Error messages are actually helpful!**



# Complaints About Hooks



- Feels weird
  - Sort of magical
  - Different from class-based state management
- Violates the explicit dependency principle
- React lock-in
- Too much magic

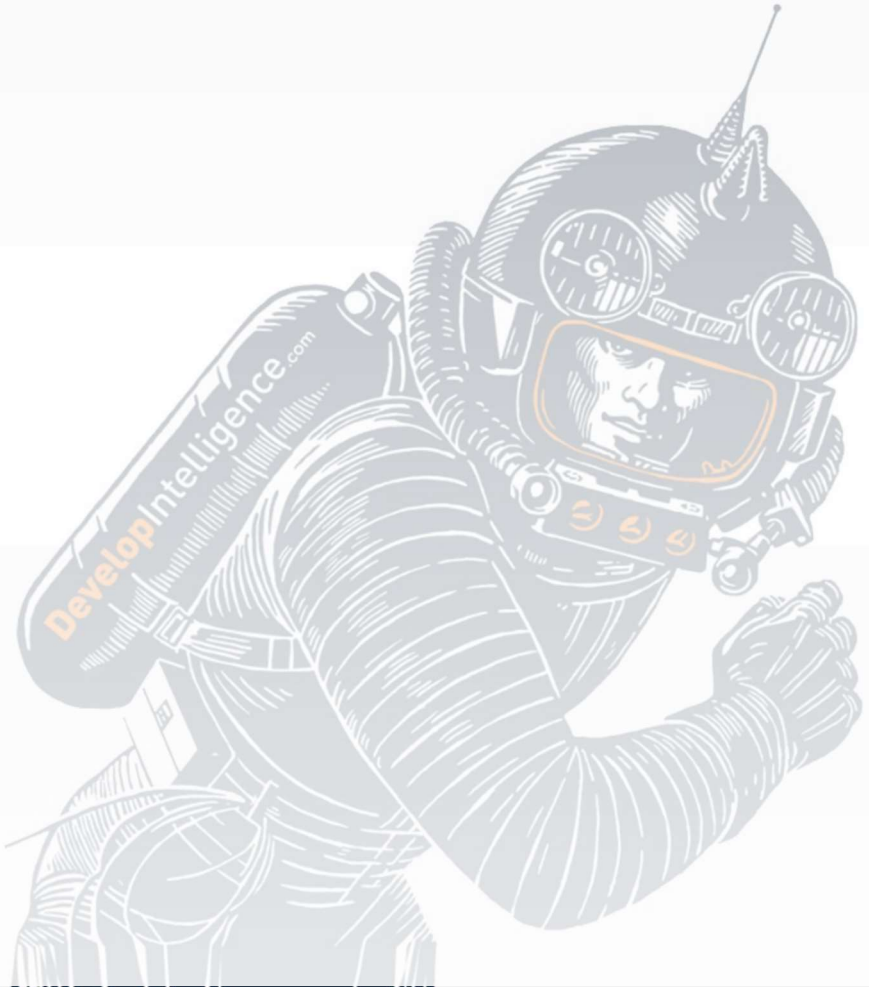
# Reasons to Love Hooks



- Composable
- Functional
- Everyone else uses them

Develop
Intelligence

# State Molecules



# Overview



- Most real-world state is non-atomic
  - Lists
  - Object graphs
- React tracks state changes based on identity

# Multiple useState

Develop Intelligence

```
export const UserEditor = () => {
  const [givenName, setGivenName] = useState('');
  const [surname, setSurname] = useState('');
  return <form>
          <h1>Hello, {givenName} {surname}</h1>
          <input type="text"
            onChange={event => setGivenName(event.target.value)}
            value={givenName}
          />
          <input type="text"
            value={surname}
            onChange={event => setSurname(event.target.value)}
          />
        </form>;
};
```

# Objects as State



- There's nothing wrong with more than one `useState`
- **But** for related data, it's often better to use an object

```
interface User{
  givenName:string;
  surname:string;
}
```

# Failed User Object

Develop Intelligence

```
export const UserEditor = () => {
  const [user, setUser] = useState<User>({givenName:'',surname:''});
  return <form> <h1>Hello, {user.givenName} {user.surname}</h1>
      <input type="text"
        onChange={event => {
          user.givenName = event.target.value;
          setUser(user);
        }}
        value={user.givenName}
      />
      <input type="text"
        onChange={event => {
          user.surname = event.target.value;
          setUser(user);
        }}
        value={user.surname}
```

# Why So Much Fail?

Develop Intelligence

- Two important points
  - React only re-renders based on changes to state
  - React uses Object.is to determine whether state's changed
- So
  - State changes can't be mutations

# The Fix



## Problematic Mutation

```
onChange={event => {
  user.surname = event.target.value;
  setUser(user);
}}
```

## Chad Copy

```
onChange={event => {
  setUser({...user, surname:event.target.value});
}}
```

# Fixed

Develop Intelligence

```
export const UserEditor = () => {
  const [user, setUser] = useState<User>({givenName:'',surname:''});
  return <form> <h1> Hello, {user.givenName} {user.surname}</h1>
      <input type="text"
        onChange={event => {
          setUser({...user,givenName:event.target.value});
        }}
        value={user.givenName}
      />
      <input type="text"
        onChange={event => {
          setUser({...user,surname:event.target.value});
        }}
        value={user.surname}
      />
    </form>;
};
```

# Arrays too

Develop Intelligence

- Mutating an array in-place doesn't change its identity
- If the array changes (or something in it changes), make a copy of the array
- e.g.
  - Array.map
  - Array.filter
  - Array.slice
  - Spread syntax e.g. `[...originalArray]`

# Quiz

Develop Intelligence

## What's the output?

```
1  const ws = [1,2,3];
2  const xs = ws;
3  xs[0] = 500;
4  console.log(`Is xs ws? ${Object.is(xs,ws)}`)
5
6  const ys = ws.map(w=>w);
7  console.log(`Is ys ws? ${Object.is(ys, ws)}`);
8
9  const zs = ws.filter(w=>true);
10 console.log(`Is zs ws? ${Object.is(zs, ws)}`);
```

# Fail #1

Develop Intelligence

```
export const Planets = () => {
  const [planets, setPlanets] = useState(['Mercury','Venus','Earth','Mars']);
  return (
    <>
      {planets.map((planet,i)=>
        <button onClick={
          ()=>planets.splice(i,1)
        }
        >
          Nuke {planet}
        </button>
      )}
    </>
  );
};
```

# Fail #2

Develop Intelligence

```
1  export const Planets = () => {
2    const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars'
3    return (
4      <>
5        {planets.map((planet, i) => (
6          <button onClick={() => {
7            planets.splice(i, 1);
8            setPlanets(planets);
9          }}
10         >Nuke {planet}</button>
11       ))}
12     </>
13   );
14 };
```

# Success #1

Develop Intelligence

```
export const Planets = () => {
  const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars'
  return (
    <>
      {planets.map((planet, i) => (
        <button onClick={() => {
          planets.splice(i, 1);
          setPlanets([...planets]); //Update with shallow copy!
        }}
        >Nuke {planet}</button>
      ))}
    </>
  );
};
```

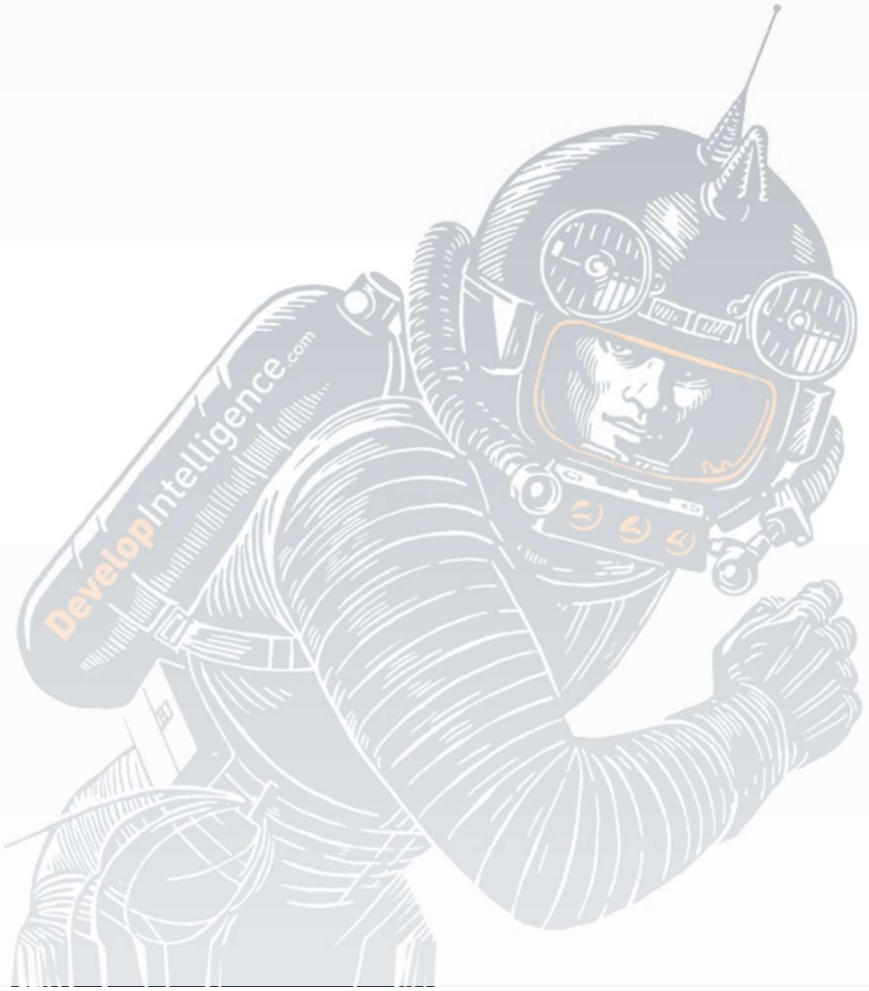
# Success #2

Develop Intelligence

```
export const Planets = () => {
  const [planets, setPlanets] = useState(['Mercury', 'Venus', 'Earth', 'Mars'
  return (
    <>
      {planets.map(planet => (
        <button
          onClick={
            event => setPlanets(planets.filter(p => p !== planet))
          } //Filter creates a new array.
        >
          Nuke {planet}
        </button>
      ))}
    </>
  );
};
```

Develop
Intelligence

# Reducers



# Overview



*Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a reducer.*

-- React.dev

# What's a 'reducer'?

Develop Intelligence

- Function that takes (a) the current state and (b) a piece of data to update the state, returning a new state.
- A.k.a **Fold**
- e.g. ECMAScript's Array.reduce

```
1  const sum = (xs:Array<number)
2    =>xs.reduce(
3      //This is the reducer function
4      (state,x)=>state+x, // Create a new state with additional item
5
6      0  // This is the initial state
7    );
8
9  console.log(sum([1,2,3,4,5]))
```

# Command Pattern

Develop Intelligence

| | |
|---|---|
| Category | Behavioral |
| Problem | • Initiating a request tightly coupled it its fulfillment.<br>• You want a history of actions<br>• Other components want to know |
| Solution | • Create an abstraction to represent the action |

# React's useReducer



*Reducers are a different way to handle state. You can migrate from useState to useReducer in three steps:*

1. *Move from setting state to dispatching actions.*
2. *Write a reducer function.*
3. *Use the reducer from your component.*

# Dumb Example

Develop Intelligence

```
interface CounterIncrementor{
  commandType:'increment';
}

interface CounterDecrementor {
  commandType: 'decrement';
}

type CounterCommand = CounterIncrementor | CounterDecrementor;

const counterReducer = (state: number, command: CounterCommand) => {
  if(command.commandType==='increment')
    return state + 1;
  if(command.commandType==='decrement')
    return state-1;
  throw new Error('Unknown command type');
};
```

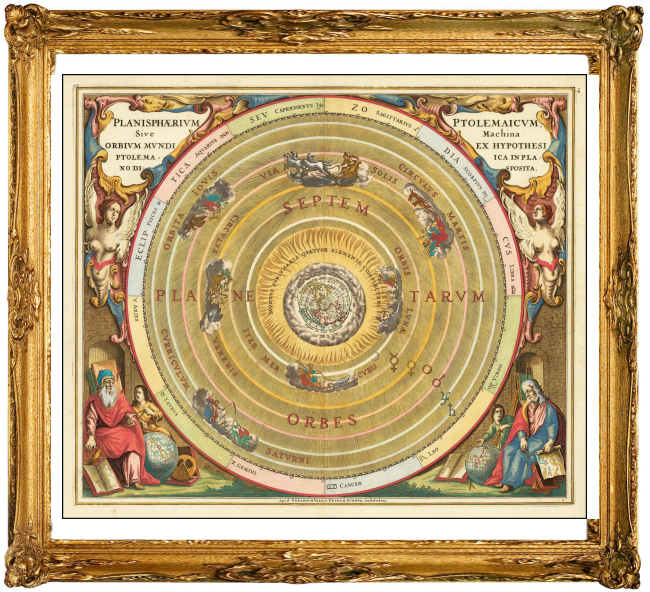
# Dumb Example (Continued)

Develop Intelligence

```
export const Counter = () => {
  const [i, dispatch] = useReducer(counterReducer, 0);
  return (
    <>
      <h1>Your number is: {i}</h1>
      <button onClick={() => dispatch({ commandType: 'increment' })}>
        Increment
      </button>
      <button onClick={() => dispatch({ commandType: 'decrement' })}>
        Decrement
      </button>
    </>
  );
};
```

# Better Example: Planets

- Look under **src/demos/planets/**
- Two actions:
  - `UpdatePlanetDescriptionAction`
  - `DestroySatelliteAction`
- Reducer: `planetsReducer`
- Typescript helps a lot



# Advantages of useReducer

Develop Intelligence

1. Testabilty
2. Fewer bugs (?)
3. Separation of concerns

# Lab: Word Processor

**Instructions: `src/practice/word-processor`**

**Summary:**

- Simple word processor
- Display word count
- Enable simple editing
  - Color

Develop
Intelligence

# Review

1. Describe `useState`
2. Explain `useReducer`
3. List the **rules of hooks**