Hooks at a Glance

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Hooks are backwards-compatible. This page provides an overview of Hooks for experienced React users. This is a fast-paced overview. If you get confused, look for a yellow box like this:

Detailed Explanation
Read the Motivation to learn why we're introducing Hooks to React.

↑↑↑ **Each section ends with a yellow box like this.** They link to detailed explanations.

📌 State Hook

This example renders a counter. When you click the button, it increments the value:

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Here, `useState` is a *Hook* (we'll talk about what this means in a moment). We call it inside a function component to add some local state to it. React will preserve this state between re-renders. `useState` returns a pair: the *current* state value and a function that lets you update it. You can call this function from an event handler or somewhere else. It's similar to `this.setState` in a class, except it doesn't merge the old and new state together. (We'll show an example comparing `useState` to `this.state` in Using the State Hook.)

The only argument to `useState` is the initial state. In the example above, it is `0` because our counter starts from zero. Note that unlike `this.state`, the state here doesn't have to be an object — although it can be if you want. The initial state argument is only used during the first render.

Declaring multiple state variables

You can use the State Hook more than once in a single component:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

The array destructuring syntax lets us give different names to the state variables we declared by calling `useState`. These names aren't a part of the `useState` API. Instead, React assumes that if you call `useState` many times, you do it in the same order during every render. We'll come back to why this works and when this is useful later.

But what is a Hook?

Hooks are functions that let you "hook into" React state and lifecycle features from function components. Hooks don't work inside classes — they let you use React without classes. (We don't recommend rewriting your existing components overnight but you can start using Hooks in the new ones if you'd like.)

React provides a few built-in Hooks like `useState`. You can also create your own Hooks to reuse stateful behavior between different components. We'll look at the built-in Hooks first.

Detailed Explanation
You can learn more about the State Hook on a dedicated page: Using the State Hook.

Effect Hook

You've likely performed data fetching, subscriptions, or manually changing the DOM from React components before. We call these operations "side effects" (or "effects" for short) because they can affect other components and can't be done during rendering.

The Effect Hook, `useEffect`, adds the ability to perform side effects from a function component. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React classes, but unified into a single API. (We'll show examples comparing `useEffect` to these methods in Using the Effect Hook.)

For example, this component sets the document title after React updates the DOM:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

When you call `useEffect`, you're telling React to run your "effect" function after flushing changes to the DOM. Effects are declared inside the component so they have access to its props and state. By default, React runs the effects after every render — *including* the first render. (We'll talk more about how this compares to class lifecycles in Using the Effect Hook.)

Effects may also optionally specify how to "clean up" after them by returning a function. For example, this component uses an effect to subscribe to a friend's online status, and cleans up by unsubscribing from it:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

In this example, React would unsubscribe from our `ChatAPI` when the component unmounts, as well as before re-running the effect due to a subsequent render. (If you want, there's a way to tell React to skip re-subscribing if the `props.friend.id` we passed to `ChatAPI` didn't change.)

Just like with `useState`, you can use more than a single effect in a component:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
```

Hooks let you organize side effects in a component by what pieces are related (such as adding and removing a subscription), rather than forcing a split based on lifecycle methods.

Detailed Explanation
You can learn more about `useEffect` on a dedicated page: Using the Effect Hook.

✌️ Rules of Hooks

Hooks are JavaScript functions, but they impose two additional rules:

- Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks **from React function components**. Don't call Hooks from regular JavaScript functions. (There is just one other valid place to call Hooks — your own custom Hooks. We'll learn about them in a moment.)

We provide a linter plugin to enforce these rules automatically. We understand these rules might seem limiting or confusing at first, but they are essential to making Hooks work well.

Detailed Explanation
You can learn more about these rules on a dedicated page: Rules of Hooks.

💡 Building Your Own Hooks

Sometimes, we want to reuse some stateful logic between components. Traditionally, there were two popular solutions to this problem: higher-order components and render props. Custom Hooks let you do this, but without adding more components to your tree.

Earlier on this page, we introduced a `FriendStatus` component that calls the `useState` and `useEffect` Hooks to subscribe to a friend's online status. Let's say we also want to reuse this subscription logic in another component.

First, we'll extract this logic into a custom Hook called `useFriendStatus`:

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });
  return isOnline;
}
```

It takes `friendID` as an argument, and returns whether our friend is online.

Now we can use it from both components:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

The state of these components is completely independent. Hooks are a way to reuse *stateful logic*, not state itself. In fact, each *call* to a Hook has a completely isolated state — so you can even use the same custom Hook twice in one component.

Custom Hooks are more of a convention than a feature. If a function's name starts with "`use`" and it calls other Hooks, we say it is a custom Hook. The `useSomething` naming convention is how our linter plugin is able to find bugs in the code using Hooks.

You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and probably many more we haven't considered. We are excited to see what custom Hooks the React community will come up with.

Detailed Explanation
You can learn more about custom Hooks on a dedicated page: Building Your Own Hooks.

🔌 Other Hooks

There are a few less commonly used built-in Hooks that you might find useful. For example, `useContext` lets you subscribe to React context without introducing nesting:

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

And `useReducer` lets you manage local state of complex components with a reducer:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
```

Detailed Explanation
You can learn more about all the built-in Hooks on a dedicated page: Hooks API Reference.

Next Steps

Phew, that was fast! If some things didn't quite make sense or you'd like to learn more in detail, you can read the next pages, starting with the State Hook documentation.

You can also check out the Hooks API reference and the Hooks FAQ.

Finally, don't miss the introduction page which explains *why* we're adding Hooks and how we'll start using them side by side with classes — without rewriting our apps.

Edit this page

Previous article
Introducing Hooks

Next article
Using the State Hook