memo

`memo` lets you skip re-rendering a component when its props are unchanged.

```
const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
```

Reference

`memo(Component, arePropsEqual?)`

Wrap a component in `memo` to get a *memoized* version of that component. This memoized version of your component will usually not be re-rendered when its parent component is re-rendered as long as its props have not changed. **But React may still re-render it: memoization is a performance optimization, not a guarantee.**

```
import { memo } from 'react';

const SomeComponent = memo(function SomeComponent(props) {
  // ...
});
```

See more examples below.

Parameters

- `Component`: The component that you want to memoize. The `memo` does not modify this component, but returns a new, memoized component instead. Any valid React component, including functions and `forwardRef` components, is accepted.
- **optional** `arePropsEqual`: A function that accepts two arguments: the component's previous props, and its new props. It should return `true` if the old and new props are equal: that is, if the component will render the same output and behave in the same way with the new props as with the old. Otherwise it should return `false`. Usually, you will not specify this function. By default, React will compare each prop with `Object.is`.

Returns

`memo` returns a new React component. It behaves the same as the component provided to `memo` except that React will not always re-render it when its parent is being re-rendered unless its props have changed.

Usage

Skipping re-rendering when props are unchanged

React normally re-renders a component whenever its parent re-renders. With `memo`, you can create a component that React will not re-render when its parent re-renders so long as its new props are the same as the old props. Such a component is said to be *memoized*.

To memoize a component, wrap it in `memo` and use the value that it returns in place of your original component:

```
const Greeting = memo(function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
});

export default Greeting;
```

A React component should always have pure rendering logic. This means that it must return the same output if its props, state, and context haven't changed. By using `memo`, you are telling React that your component complies with this requirement, so React doesn't need to re-render as long as its props

haven't changed. Even with `memo`, your component will re-render if its own state changes or if a context that it's using changes.

In this example, notice that the `Greeting` component re-renders whenever `name` is changed (because that's one of its props), but not when `address` is changed (because it's not passed to `Greeting` as a prop):

App.js

Download Reset

```
      </label>
      <label>
        Address{': '}
        <input value={address} onChange={e => setAddress(e.target.val
      </label>
      <Greeting name={name} />
    </>
  );
}

const Greeting = memo(function Greeting({ name }) {
```

Show more

Note

You should only rely on `memo` as a performance optimization. ***If your code doesn't work without it, find the underlying problem and fix it first. Then you may add `memo` to improve performance.***

DEEP DIVE

Should you add memo everywhere?

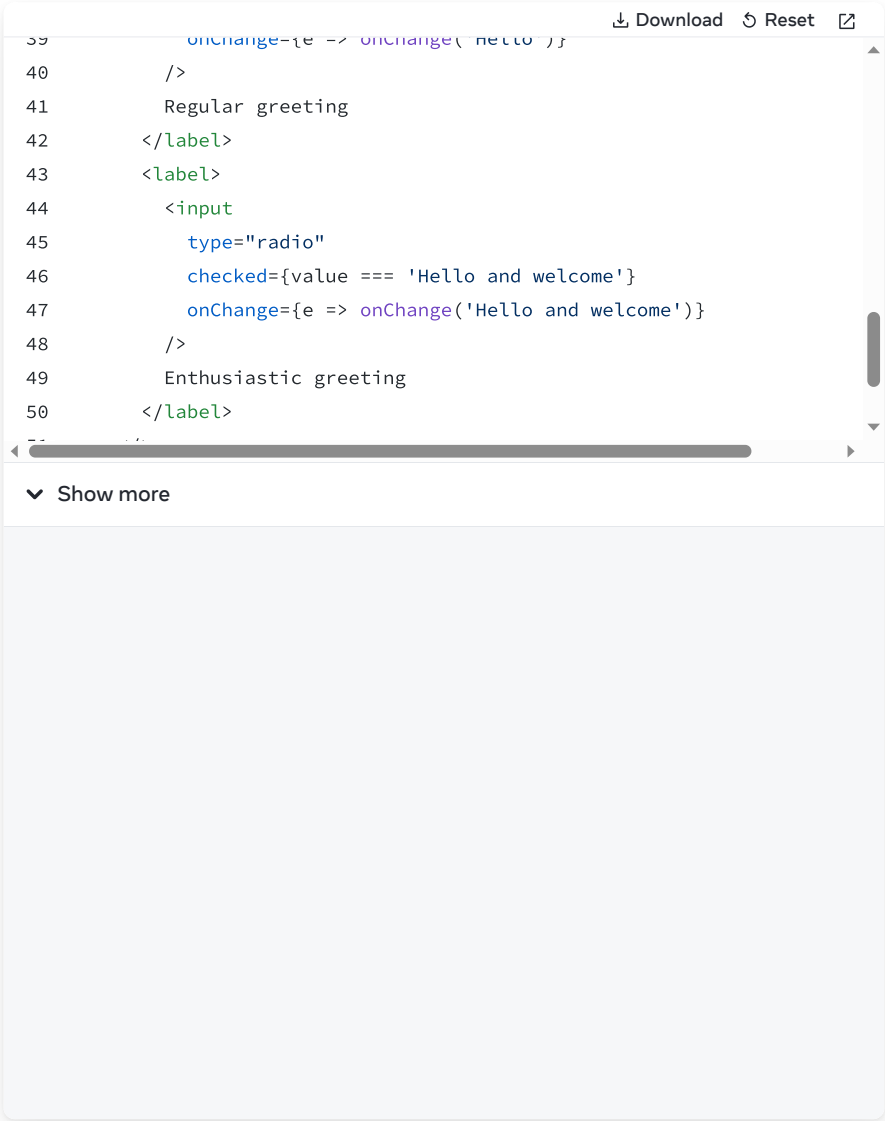
Hide Details

If your app is like this site, and most interactions are coarse (like replacing a page or an entire section), memoization is usually unnecessary. On the other hand, if your app is more like a drawing editor, and most interactions are granular (like moving shapes), then you might find memoization very helpful.

Optimizing with `memo` is only valuable when your component re-renders often with the same exact props, and its re-rendering logic is expensive. If there is no perceptible lag when your component re-renders, `memo` is unnecessary. Keep in mind that `memo` is completely useless if the props passed to your component are *always different*, such as if you pass an object or a plain function defined during rendering (Objects and Functions Change refernces on every render hence they fail the comparision of Object.is() with previous Objects and Functions). This is why you will often need `useMemo` and `useCallback` together with `memo`.

Updating a memoized component using state

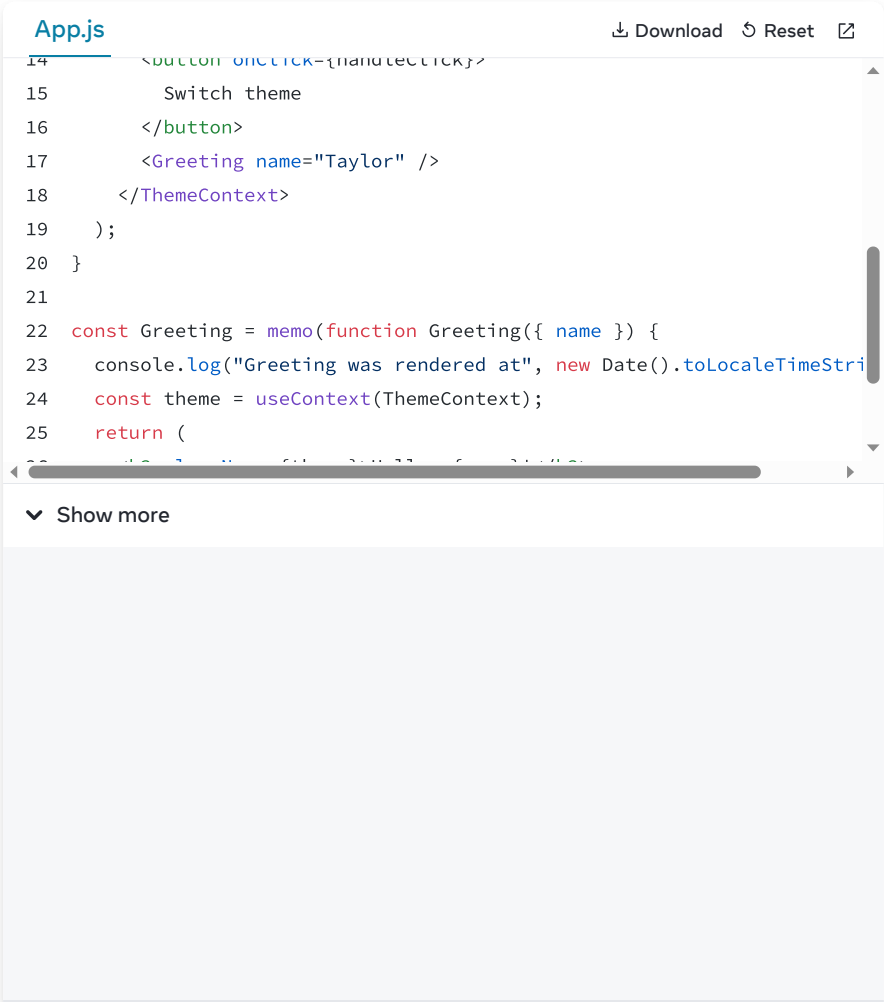
Even when a component is memoized, it will **still re-render when its own state changes.** *Memoization only has to do with props that are passed to the component from its parent.*

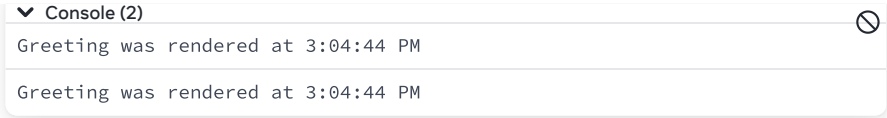


If you set a state variable to its current value, React will skip re-rendering your component even without `memo`. You may still see your component function being called an extra time, but the result will be discarded.

Updating a memoized component using a context

Even when a component is memoized, **it will still re-render when a context that it's using changes.** Memoization only has to do with props that are passed to the component from its parent.





To make your component re-render only when a *part* of some context changes, split your component in two. Read what you need from the context in the outer component, and pass it down to a memoized child as a prop.

Minimizing props changes

When you use `memo`, your component re-renders whenever any prop is not *shallowly equal* to what it was previously. This means that React compares every prop in your component with its previous value using the `Object.is` comparison. Note that `Object.is(3, 3)` is `true`, but `Object.is({}, {})` is `false`.

To get the most out of `memo`, minimize the times that the props change. For example, if the prop is an object, prevent the parent component from re-creating that object every time by using `useMemo`:

```
function Page() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);

  const person = useMemo(
    () => ({ name, age }),
    [name, age]
  );

  return <Profile person={person} />;
}

const Profile = memo(function Profile({ person }) {
  // ...
});
```

A better way to minimize props changes is to make sure the component accepts the minimum necessary information in its props. For example, it could accept individual values instead of a whole object:

```
function Page() {
  const [name, setName] = useState('Taylor');
  const [age, setAge] = useState(42);
  return <Profile name={name} age={age} />;
}

const Profile = memo(function Profile({ name, age }) {
  // ...
});
```

Even individual values can sometimes be projected to ones that change less frequently. For example, here a component accepts a boolean indicating the presence of a value rather than the value itself:

```
function GroupsLanding({ person }) {
  const hasGroups = person.groups !== null;
  return <CallToAction hasGroups={hasGroups} />;
}

const CallToAction = memo(function CallToAction({ hasGroups }) {
  // ...
});
```

When you need to pass a function to memoized component, either declare it outside your component so that it never changes, or `useCallback` to cache its definition between re-renders.

Specifying a custom comparison function

In rare cases it may be infeasible to minimize the props changes of a memoized component. In that case, you can provide a custom comparison

function, which React will use to compare the old and new props instead of using shallow equality. This function is passed as a second argument to `memo`. It should return `true` only if the new props would result in the same output as the old props; otherwise it should return `false`.

```
const Chart = memo(function Chart({ dataPoints }) {
  // ...
}, arePropsEqual);

function arePropsEqual(oldProps, newProps) {
  return (
    oldProps.dataPoints.length === newProps.dataPoints.length &&
    oldProps.dataPoints.every((oldPoint, index) => {
      const newPoint = newProps.dataPoints[index];
      return oldPoint.x === newPoint.x && oldPoint.y === newPoint.y;
    })
  );
}
```

If you do this, use the Performance panel in your browser developer tools to make sure that your comparison function is actually faster than re-rendering the component. You might be surprised.

When you do performance measurements, make sure that React is running in the production mode.

Pitfall

If you provide a custom `arePropsEqual` implementation, **you must compare every prop, including functions.** Functions often close over the props and state of parent components. If you return `true` when `oldProps.onClick !== newProps.onClick`, your component will keep "seeing" the props and state from a previous render inside its `onClick` handler, leading to very confusing bugs.

Avoid doing deep equality checks inside `arePropsEqual` unless you are 100% sure that the data structure you're working with has a known limited depth. **Deep equality checks can become incredibly slow** and can freeze your app for many seconds if someone changes the data structure later.

Do I still need React.memo if I use React Compiler?

When you enable React Compiler, you typically don't need `React.memo` anymore. The compiler automatically optimizes component re-rendering for you.

Here's how it works:

Without React Compiler, you need `React.memo` to prevent unnecessary re-renders:

```
// Parent re-renders every second
function Parent() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(s => s + 1);
    }, 1000);
    return () => clearInterval(interval);
  }, []);

  return (
    <>
      <h1>Seconds: {seconds}</h1>
      <ExpensiveChild name="John" />
    </>
  );
}
```

```
// Without memo, this re-renders every second even though props don't ch
const ExpensiveChild = memo(function ExpensiveChild({ name }) {
  console.log('ExpensiveChild rendered');
  return <div>Hello, {name}!</div>;
});
```

With React Compiler enabled, the same optimization happens automatically:

```
// No memo needed - compiler prevents re-renders automatically
function ExpensiveChild({ name }) {
  console.log('ExpensiveChild rendered');
  return <div>Hello, {name}!</div>;
}
```

Here's the key part of what the React Compiler generates:

```
function Parent() {
  const $ = _c(7);
  const [seconds, setSeconds] = useState(0);
  // ... other code ...

  let t3;
  if ($[4] === Symbol.for("react.memo_cache_sentinel")) {
    t3 = <ExpensiveChild name="John" />;
    $[4] = t3;
  } else {
    t3 = $[4];
  }
  // ... return statement ...
}
```

Notice the highlighted lines: The compiler wraps `<ExpensiveChild name="John" />` in a cache check. Since the `name` prop is always `"John"`, this JSX is created once and reused on every parent re-render. This is exactly what `React.memo` does - it prevents the child from re-rendering when its props haven't changed.

The React Compiler automatically:

1. Tracks that the `name` prop passed to `ExpensiveChild` hasn't changed
2. Reuses the previously created JSX for `<ExpensiveChild name="John" />`
3. Skips re-rendering `ExpensiveChild` entirely

This means **you can safely remove `React.memo` from your components when using React Compiler**. The compiler provides the same optimization automatically, making your code cleaner and easier to maintain.

Note

The compiler's optimization is actually more comprehensive than `React.memo`. It also memoizes intermediate values and expensive computations within your components, similar to combining `React.memo` with `useMemo` throughout your component tree.

Troubleshooting

My component re-renders when a prop is an object, array, or function

React compares old and new props by shallow equality: that is, it considers whether each new prop is reference-equal to the old prop. If you create a new object or array each time the parent is re-rendered, even if the individual elements are each the same, React will still consider it to be changed. Similarly, if you create a new function when rendering the parent component, React will consider it to have changed even if the function has the same definition. To avoid this, simplify props or memoize props in the parent component.