useRef

`useRef` is a React Hook that lets you reference a value that's not needed for rendering.

```
const ref = useRef(initialValue)
```

Reference

`useRef(initialValue)`

Call `useRef` at the top level of your component to declare a ref.

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

function MyComponent() {
  const intervalRef = useRef(0);
  const inputRef = useRef(null);
  // ...
```

See more examples below.

Parameters

- `initialValue`: The value you want the ref object's `current` property to be initially. It can be a value of any type. This argument is ignored after the initial render.

Returns

`useRef` returns an object with a single property:

```
{
  current: 0 // The value you passed to useRef
}
```

You can access the current value of that ref through the `ref.current` property. This value is intentionally mutable, meaning you can both read and write to it. It's like a secret pocket of your component that React doesn't track.

- `current`: Initially, it's set to the `initialValue` you have passed. You can later set it to something else. If you pass the ref object to React as a `ref` attribute to a JSX node, React will set its `current` property.

On the next renders, `useRef` will return the same object.

Differences between refs and state

Perhaps you're thinking refs seem less "strict" than state—you can mutate them instead of always having to use a state setting function, for instance. But in most cases, you'll want to use state. Refs are an "escape hatch" you won't need often. Here's how state and refs compare:

refs	state
`useRef(initialValue)` returns `{ current: initialValue }`	`useState(initialValue)` returns the current value of a state variable and a state setter function (`[value, setValue]`)
Doesn't trigger re-render when you change it.	Triggers re-render when you change it.
Mutable—you can modify and update `current`'s value outside of the rendering process.	"Immutable"—you must use the state setting function to modify state variables to queue a re-render.
You shouldn't read (or write) the `current` value during rendering.	You can read state at any time. However, each render has its own snapshot of state which does not change.

Caveats

- You can mutate the `ref.current` property. Unlike state, it is mutable. However, if it holds an object that is used for rendering (for example, a piece of your state), then you shouldn't mutate that object.
- When you change the `ref.current` property, React does not re-render your component. React is not aware of when you change it because a ref is a plain JavaScript object.
- Do not write *or read* `ref.current` during rendering, except for initialization. This makes your component's behavior unpredictable.
- In Strict Mode, React will **call your component function twice** in order to help you find accidental impurities. This is development-only behavior and does not affect production. Each ref object will be created twice, but one of the versions will be discarded. If your component function is pure (as it should be), this should not affect the behavior.

Usage

Referencing a value with a ref

Call `useRef` at the top level of your component to declare one or more refs.

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

function Stopwatch() {
  const intervalRef = useRef(0);
  // ...
```

`useRef` returns a ref object with a single `current` property initially set to the initial value you provided.

On the next renders, `useRef` will return the same object. You can change its `current` property to store information and read it later. This might remind you of state, but there is an important difference.

Changing a ref does not trigger a re-render. This means ***refs are perfect for storing information that doesn't affect the visual output of your component***. For example, if you need to store an interval ID and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its `current` property:

```
function handleStartClick() {
  const intervalId = setInterval(() => {
    // ...
  }, 1000);
  intervalRef.current = intervalId;
}
```

Later, you can read that interval ID from the ref so that you can call clear that interval:

```
function handleStopClick() {
  const intervalId = intervalRef.current;
  clearInterval(intervalId);
}
```

By using a ref, you ensure that:

- You can **store information** between re-renders (unlike regular variables, which reset on every render).
- Changing it **does not trigger a re-render** (unlike state variables, which trigger a re-render).
- The **information is local** to each copy of your component (unlike the variables outside, which are shared).

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use state for that instead. Read more about choosing between `useRef` and `useState`.

Pitfall

Do not write *or read* `ref.current` during rendering.

React expects that the body of your component behaves like a pure function:

- If the inputs (props, state, and context) are the same, it should return exactly the same JSX.
- Calling it in a different order or with different arguments should not affect the results of other calls.

Reading or writing a ref **during rendering** breaks these expectations.

```
function MyComponent() {
  // ...
  // 🚩 Don't write a ref during rendering
  myRef.current = 123;
  // ...
  // 🚩 Don't read a ref during rendering
  return <h1>{myOtherRef.current}</h1>;
}
```

You can read or write refs **from event handlers or effects instead**.

```
function MyComponent() {
  // ...
  useEffect(() => {
    // ✅ You can read or write refs in effects
    myRef.current = 123;
  });
  // ...
  function handleClick() {
    // ✅ You can read or write refs in event handlers
    doSomething(myOtherRef.current);
  }
  // ...
}
```

If you *have to* read or write something during rendering, use state instead.

When you break these rules, your component might still work, but most of the newer features we're adding to React will rely on these expectations. Read more about keeping your components pure.

Manipulating the DOM with a ref

It's particularly common to use a ref to manipulate the DOM. React has built-in support for this.

First, declare a ref object with an initial value of `null`:

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

function MyComponent() {
  const inputRef = useRef(null);
  // ...
```

Then pass your ref object as the `ref` attribute to the JSX of the DOM node you want to manipulate:

Pass it to a built-in DOM element this does not work with custom components we create.

```
  // ...
  return <input ref={inputRef} />;
```

After React creates the DOM node and puts it on the screen, React will set the `current` property of your ref object to that DOM node. Now you can access the `<input>`'s DOM node and call methods like `focus()`:

```
  function handleClick() {
    inputRef.current.focus();
  }
```

React will set the `current` property back to `null` when the node is removed from the screen.

Read more about manipulating the DOM with refs.

Avoiding recreating the ref contents

React saves the initial ref value once and ignores it on the next renders.

```
function Video() {
  const playerRef = useRef(new VideoPlayer());
  // ...
```

Although the result of `new VideoPlayer()` is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating expensive objects.

To solve it, you may initialize the ref like this instead:

```
function Video() {
  const playerRef = useRef(null);
  if (playerRef.current === null) {
    playerRef.current = new VideoPlayer();
  }
  // ...
```

Normally, writing or reading `ref.current` during render is not allowed. However, it's fine in this case because the result is always the same, and the condition only executes during initialization so it's fully predictable.

Troubleshooting

I can’t get a ref to a custom component

If you try to pass a `ref` to your own component like this:

```
const inputRef = useRef(null);

return <MyInput ref={inputRef} />;
```

You might get an error in the console:

```
Console

TypeError: Cannot read properties of null
```

By default, your own components don’t expose refs to the DOM nodes inside them.

To fix this, find the component that you want to get a ref to:

```
export default function MyInput({ value, onChange }) {
  return (
    <input
      value={value}
      onChange={onChange}
    />
  );
}
```

And then add `ref` to the list of props your component accepts and pass `ref` as a prop to the relevent child built-in component like this:

```
function MyInput({ value, onChange, ref }) {
  return (
    <input
      value={value}
      onChange={onChange}
      ref={ref}
    />
  );
};

export default MyInput;
```

Then the parent component can get a ref to it.